Zcash NU5 Cryptography Review

Zcash

November 1, 2021

Prepared for

Sean Bowe
Jack Grigg
Daira Hopwood
Taylor Hornby
Ying Tong Lai
Kris Nuttycombe
Larry Ruane
Steven Smith

Prepared by

Parnian Alimi
Aleksandar Kircanski
Thomas Pornin

Executive Summary

Synopsis

In March 2021, Electric Coin Co. engaged NCC Group to perform a review of the upcoming network protocol upgrade NU5 to the Zcash protocol (code-named "Orchard"). The review was to be performed over multiple phases: first, the specification document changes and the relevant ZIPs, then, in June 2021, the implementation itself.

Scope

The scope of the specification review consisted of the following:

- The Zcash protocol specification: https://zips.z.cash/protocol/nu5.pdf (reviewed version was 2021.1.19, dated March 17th, 2021[1]).
- The contents of ZIPs 216, 224, 225, 244 and 252, as of March 15th, 2021:
  - https://zips.z.cash/zip-0216
  - https://zips.z.cash/zip-0224
  - https://zips.z.cash/zip-0225
  - https://zips.z.cash/zip-0244
  - https://zips.z.cash/zip-0252

As for the implementation review, scope included:

- For ZIP 216:
  - primary fix: https://github.com/zkcrypto/jubjub/tree/f192388f3cad327868db4e8a58582083f24ef09f
  - fix integration in `librustzcash`: https://github.com/zcash/librustzcash/pull/396
  - changes to consensus rules: https://github.com/zcash/zcash/pull/5213
- For ZIP 224:
  - Pallas and Vesta curves implementation: https://github.com/zcash/pasta_curves/tree/d8547d2326b16b11b5c3e8ada231065111b51680
  - support for RedPallas added into a fork of the `redjubjub` crate: https://github.com/str4d/redjubjub/tree/d5d8c5f3bb704bad8ae88fe4a29ae1f744774cb2
  - `orchard` crate: https://github.com/zcash/orchard/tree/93a7f1db228479228f768e9d86dd5868e4c2ff1e with the exception of the "circuit" part.
  - note encryption support (shared between Sapling and Orchard): https://github.com/zcash/librustzcash/pull/358 and https://github.com/zcash/librustzcash/pull/390
  - Orchard pool value tracking: https://github.com/zcash/zcash/pull/5228
  - Orchard proof verification consensus rules: https://github.com/zcash/zcash/pull/5232
  - signature validation consensus rules: https://github.com/zcash/zcash/pull/5217
  - block header commitments: https://github.com/zcash/zcash/pull/5220
- For ZIP 225:
  - Rust parser code: https://github.com/zcash/librustzcash/pull/375 and https://github.com/zcash/librustzcash/pull/398
  - C++ integration: https://github.com/zcash/zcash/pull/5202
- For ZIP 244:
  - transaction digest components: https://github.com/zcash/librustzcash/pull/375
  - transaction and signature digests: https://github.com/zcash/zcash/pull/5215 and https://github.com/zcash/zcash/pull/5219
- ZIP 316 specification (https://zips.z.cash/zip-0316) and implementation (https://github.com/zcash/librustzcash/pull/383 and https://github.com/zcash/librustzcash/pull/352) were added to the scope in the second phase.

Limitations

While NCC Group Cryptography Services completed the scope of the security audit there were some challenges with scope adjustments and code not being finalized at the start of the implementation review phase.

Findings and Strategic Recommendations

No serious issue was detected during the review. Some potential issues were reported, but cannot be triggered in the current implementation and the intended usage context; they might induce vulnerabilities later on if some library code were later reused in a different context (e.g. improper implementation of the random point selection function for the iso-Pallas and iso-Vesta curves; see finding NCC-E001151-002 on page 6).

A description of the audit process and some relevant remarks have been assembled in Appendix B on page 13 for the specification review, and Appendix C on page 18 for the implementation review. None of these remarks constitutes a security vulnerability, but they are still worth mentioning and can be viewed as software engineering issues impacting long-term readability and maintenance of the code. Most of them relate to cases of dead code, duplicated code, or ambiguous (or lacking) documentation.

[1] Document hash (SHA-256): fcfe26c583a23d9990bf3fdd836e86e7d8ecd1c89c5882a0516517ba65d21216

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 11.

Title	Status	ID	Risk
Insufficient Checks in Unified Address Parser	Fixed	004	Medium
Incorrect Random Point Generation and Decoding on Isogenous Curves	Reported	002	Low
Non-Constant-Time Operations in Pasta Curves	Reported	003	Low

Finding	**Insufficient Checks in Unified Address Parser**
Risk	**Medium** Impact: Low, Exploitability: Low
Identifier	NCC-E001151-004
Status	Fixed
Category	Data Validation
Component	librustzcash
Location	`components/zcash_address/src/kind/unified.rs`, line 90
Impact	Failure to reject Unified Addresses (UA) that include more than one address type can lead to ambiguous choice of receiver address.

Description

ZIP 316 clearly states that the "Senders MUST reject Unified Addresses in which the same Typecode appears more than once, or that include both P2SH and P2PKH Transparent Addresses, or that contain only a Transparent Address". The unified address parser (used by the sender's wallet) parses the input as a sequence of Tag-Length-Values, but does not check if A) an address type is repeated, B) P2SH and P2PKH are both present, and C) at least one shielded address is included. This can be seen in the code below.

```
90  fn try_from(buf: &[u8]) -> Result<Self, Self::Error> {
91      let encoded =
        ➔   f4jumble::f4jumble_inv(buf).ok_or(ParseError::InvalidEncoding)?;
92
93      // Validate and strip trailing zero bytes.
94      let encoded = match encoded.split_at(encoded.len() - PADDING_LEN) {
95          (encoded, tail) if tail == &[0; PADDING_LEN][..] => Ok(encoded),
96          _ => Err(ParseError::InvalidEncoding),
97      }?;
98
99      iter::repeat(())
100         .scan(encoded, |encoded, _| match encoded {
101             // Base case: we've parsed the full encoding.
102             [] => None,
103             // The raw encoding of a Unified Address is a concatenation of:
104             // - typecode: byte
105             // - length: byte
106             // - addr: byte[length]
107             [typecode, length, data @ ..] if data.len() >= *length as usize => {
108                 let (addr, rest) = data.split_at(*length as usize);
109                 *encoded = rest;
110                 Some(Receiver::try_from((*typecode, addr)))
111             }
112             // The encoding is truncated.
113             _ => Some(Err(ParseError::InvalidEncoding)),
114         })
115         .collect::<Result<_, _>>()
116         .map(Address)
117 }
```

The loop on line 99 scans, decodes, and appends receiver addresses to the `Address` list without performing any checks. As a consequence, it is ambiguous which address the caller of this API (Sender wallet) will use to create the transaction. In order to exploit this, an active attacker has to decode the UA and insert their address at the index they predict the sender

wallet will prefer. This is not easier than simply replacing receiver's UA with the malicious UA, thus the exploitability of this finding is low.

Recommendation Adapt the Unified Address parser to check that there are at most one of each address types in the input and there is at most one type of transparent address present.

Retest Results NCC Group reviewed Pull Request 416 and observed that the `Address` construction API is updated. With this fix, an `Address` can be constructed from:

1. A vector of `Receivers` (via the TryFrom<Vec< Receiver > > API), which returns an error if any of the 3 checks mentioned in this finding do not pass.

Or, 2. A byte array (via the TryFrom<&[u8]> API) which first parses the input into an array of `Receivers` and if successful, constructs the `Address` from it.

Since an `Address` can only be constructed after passing required checks, this finding has been marked as `Fixed`.

Finding	**Incorrect Random Point Generation and Decoding on Isogenous Curves**
Risk	**Low** Impact: Undetermined, Exploitability: None
Identifier	NCC-E001151-002
Status	Reported
Category	Cryptography
Component	pasta_curves
Location	`pasta_curves/src/curves.rs`, lines 69 and 633
Impact	Invalid curve points may be created if decoding from bytes or random point generation are used on the isogenous curves (iso-Pallas and iso-Vesta). Since the implementations of these curves only have crate visibility, and none of the crate code invokes these functionalities, this issue cannot be triggered in the present state of the code.

Description

The `new_curve_impl` macro defines the implementation of all functions attached to the structure types that implement a given curve (in Jacobian and affine coordinates); this is used for the Pallas and Vesta curves, and their isogenous counterparts iso-Pallas and iso-Vesta. Among these functions are the following:

- `random()` generates a random non-neutral curve point.
- `from_bytes()` decodes a sequence of 32 bytes into a curve point.

`random()` uses rejection sampling: a candidate x coordinate is produced as a random field element; the curve equation is then used to compute y^2 from x. If that value is indeed a quadratic residue, a square root is extracted (and an extra random bit is used to choose its sign); otherwise, the process loops with a new candidate x coordinate. The curve equation is nominally $y^2 = x^3 + ax + b$ for two given constants a and b. However, the `random()` function only uses b:

```
let x3 = x.square() * x;
let y = (x3 + $name::curve_constant_b()).sqrt();
```

Thus, the candidate y^2 is computed as $x^3 + b$; this assumes that $a = 0$. This is true for the Pallas and Vesta curves (which both have equation $y^2 = x^3 + 5$), but not for iso-Pallas and iso-Vesta, who both have non-zero a. For these isogenous curves, the `random()` function would generate points which are not part of the curve.

Similarly, `from_bytes()` decodes the x coordinate from the provided bytes and computes y^2 using the curve equation, again assuming that $a = 0$:

```
let x3 = x.square() * x;
(x3 + $name::curve_constant_b()).sqrt().and_then( // ...
```

On iso-Pallas and iso-Vesta, `from_bytes()` would decode incoming bytes into an incorrect point, not the intended one (and not even a point on the curve).

The iso-Pallas curve structures (`IsoEp` and `IsoEpAffine`) are declared with `pub(crate)` visibility; the same applies to the iso-Vesta structures (`IsoEq` and `IsoEqAffine`). Thus, their `random()` and `from_bytes()` functions may be invoked only from the same crate, and none of the crate code currently calls these functions. The issue described above thus cannot be triggered with the `pasta_curves` crate as currently implemented. However, the issue may

resurface in a future version, if for some reason the isogenous curves are made part of the public API.

Recommendation The `random()` and `from_bytes()` functions can be easily fixed to take the a curve parameter into account, so that they are correct for the isogenous curves as well. The extra cost induced by the multiplication by a is negligible with regard to the cost of the square root that immediately follows (a square root cost is about 200 to 250 times that of a multiplication).

Finding	**Non-Constant-Time Operations in Pasta Curves**
Risk	**Low** Impact: Low, Exploitability: Low
Identifier	NCC-E001151-003
Status	Reported
Category	Cryptography
Component	pasta_curves

Location	`pasta_curves/src/curves.rs`
	`pasta_curves/src/fields/fp.rs`
	`pasta_curves/src/fields/fq.rs`
	`pasta_curves/src/arithmetic/fields.rs`

Impact	Non-constant-time processing may leak secret information through timing-based side-channels if operating on secret inputs. Since the `pasta_curves` API is *seemingly* aiming at constant-time processing, but does not actually provide that feature, users of these crate in contexts such as key exchange or signature generation may unknowingly be vulnerable to timing attacks.
Description	The API of the Pasta curves implementation does not *formally* make any assertion of the intent of providing constant-time functions. However, it does not contain any disclaimer either; moreover, the API and the code structure contain strong hints that constant-time behaviour is or at least has been a goal: • The field structures (`Fp` and `Fq`) and the curve structures (`Ep`, `EpAffine`...) implement the `ConstantTimeEq` trait (promising constant-time equality comparisons). • Many functions that have failure conditions on invalid input use the `CtOption` type, imported from the `subtle` crate; its documentation explicitly states that it is intended to be used "in constant-time APIs".[2] Similarly, the `Choice` type from the `subtle` create is also used in several places. • The `to_bytes()` function (on an affine curve point) includes an explicit comment "TODO: not constant time" which indicates that constant-time behaviour is intended (but not yet implemented) for this specific function. • The point multiplication routine (`mul()`) uses a double-and-add algorithm, with the addition computed for every scalar bit, the result being kept or not through a constant-time conditional selection primitive. This is substantially more expensive than using a conditional jump to perform the addition only when the corresponding scalar bit is equal to 1; the extra cost makes sense only if constant-time processing is intended. However, the implementation fails to be constant-time in several places: • The `cmp()` function for field elements uses lexicographic ordering on the binary representations of the field elements. The comparison is done on a byte-by-byte basis and stops at the first differing byte values; this is not constant-time. • The square root implementation in finite fields uses Sarkar's method[3] to optimize the final part of the operation in the specific fields used by Pasta curves. This method relies on precomputed tables, and accesses are performed at data-dependent addresses. This is inherently non-constant-time. • The curve point addition routine uses conditional jumps to handle the exceptional cases

[2] https://docs.rs/subtle/2.3.0/subtle/struct.CtOption.html
[3] https://eprint.iacr.org/2020/1407

that are not handled correctly by the point addition formulas on short Weierstraß curves in Jacobian coordinates (namely adding the point-at-infinity to a non-infinity point, and adding a point to another representation of itself). Execution time and memory access pattern will differ when these cases are encountered.

- Curve point decoding (`from_bytes()`), random point selection (`random()`), and hash-to-curve operations all use square root computations and thus inherit the non-constant-time behaviour. The hash-to-curve operation also ends with a point addition (the results of two distinct map-to-curve invocations are added together) which may conceptually hit one of the addition exceptional cases, albeit with a negligible probability.
- Point multiplication by a scalar (`mul()` functions on both Jacobian and affine structures) uses a double-and-add algorithm. The accumulator point (`acc` variable) is initialized with the point-at-infinity; thus, all point additions in the loop will exercise the special case of adding a point to the point-at-infinity, up to the first non-zero bit in the scalar (in high-to-low order). Since the special case of point addition has a much shorter execution time, the overall execution time of `mul()` will leak the actual binary length of the scalar; in an ECDSA or Schnorr signature generation context, this leak would be a very serious vulnerability.
- Point encoding (`to_bytes()` function) has a special case for the point-at-infinity (this non-constant-time behaviour is acknowledged in an internal code comment).

In the main intended usage context of the Pasta curves, i.e. *verification* of Halo 2 proofs, there is no secret value, and side-channel leakages are irrelevant. In that context, none of the above matters. However, for *generation* of such proofs, secret values are used, and side-channel resistance matters, unless the overall operational context is such that proof generation can be assumed to happen on physical systems that cannot be observed by attackers.

Recommendation

The `pasta_curves` crate should explicitly document its stance with regard to constant-time implementations. If constant-time behaviour is not an intended feature of the code, then there should be disclaimers to that effect, especially for the documentation of API functions that *appear* to aim at constant-time behaviour (e.g. through the use of a constant-time primitive such as `CtChoice`).

In order to make the whole crate constant-time, the following changes would be needed:

- Square root extraction must be changed into a constant-time routine. Using Sarkar's methodology, this can be achieved by producing the final 32-bit exponent t bit by bit instead of using 8-bit chunks. This would involve some extra computational overhead to the square root operation, though probably not in catastrophic amounts.
- Curve point addition must be made constant-time. A simple way would be to use projective coordinates with Renes-Costello-Batina formulas,[4] which are complete as long as the curve has odd order (which is the case for the Pasta curves).
- For the specific operation of point multiplication of a curve point by a scalar, the constant-time complete addition routine can be used. Alternatively, Jacobian coordinates can still be leveraged to improve performance, using the following method:
  - The source point can be converted to Jacobian coordinates when entering the function, and converted back on output, if the overall representation format uses projective coordinates (these conversions are relatively inexpensive).
  - The scalar is nominally a modular integer (modulo q for points on the Pallas curve). It can be converted into an integer n in the 0 to $q-1$ range; the integer $n' = n + 2q$ can then be used as an equivalent scalar value. This alternate integer is such that $2^{255} < n' < 2^{256}$, which guarantees that n', as a 256-bit integer, starts with a 1. This removes the issue with the accumulator containing the point-at-infinity.

[4] https://eprint.iacr.org/2015/1060

  - In the double-and-add algorithm, it can be shown that none of the point additions for the first 253 steps may hit one of the exceptional cases of Jacobian coordinates formulas. Only for the final bits should the implementation revert to projective coordinates and use the complete routines.
- The `cmp()` (on fields) and `to_bytes()` functions (on curve points) can be made constant-time in straightforward ways.

Appendix A: Finding Field Definitions

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

Risk Scale

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

Overall Risk

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

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

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

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

Low Implies a relatively minor threat to the application.

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

Impact

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

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

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

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

Exploitability

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

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

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

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

Category

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

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

Appendix B: Protocol Specification and ZIPs Review

This appendix documents the notes and comments that were presented in the first phase of the NU5 Zcash release review, which focused solely on the specification and accompanying ZIPs, as outlined by the NU5 release web page. At this stage, NCC Group consultants looked at the specification and a subset of the ZIPs in scope.

Specification review

In this section, we list remarks on the specification itself. None is a security issue *per se*, but they were deemed worth reporting nonetheless, if only because they may create confusion in readers or have otherwise detrimental indirect effects. They are listed below in order of appearance in the specification document nu5.pdf.[5] Not included are the multiple places marked as TODO or broken references to sections that do not exist yet (NCC Group assumes that these placeholders will be systematically searched for and completed before the formal release of the NU5 protocol specification).

- **Section 4.11.1 (page 30):** "reasonble" is a typo for "reasonable"
- **Section 4.14 (page 50):** in $\mathsf{v}^{\mathsf{balanceOrchard}}$, the word "Orchard" is unduly in bold font (in the LaTeX source, `\vBalance{\Orchard}` is used, but it should probably be `\vBalance{Orchard}`).
- **Section 5.2 (page 67):** some spaces are missing in "Sapling*and*Orchard"
- **Section 5.4.1.9 (page 75):** value c is defined as "the largest integer such that $2^n \leq (r_{\mathbb{P}} - 1)/2$". This "$2^n$" should be "$2^c$".
- **Section 5.4.1.9 (page 75):** the incomplete addition operator is incompletely defined, since its output can be the special no-value $\perp$, which may then be used as input to other applications of the incomplete addition operator. Similarly, SinsemillaHash is defined as applying $\mathsf{Extract}_{\mathbb{P}}$ on the output of SinsemillaHashToPoint. This last function normally outputs a point on $\mathbb{P}$ (the Pallas curve) but it may also conceptually return $\perp$, and $\mathsf{Extract}_{\mathbb{P}}$ is not defined for an input of $\perp$ (page 97). The intent was probably to make $\perp$ an alias to $\mathcal{O}_{\mathbb{P}}$.
- **Section 5.4.2 (page 79):** the formula defines the PRF output as $\mathsf{Poseidon}(\mathsf{nk}, \rho)$, but the Poseidon *function* was not defined (the function is called PoseidonHash).
- **Section 5.4.4 (page 80):** the FF1 encryption mode for format preserving encryption is used. This mode was originally created by Voltage (which was later acquired by HP) and submitted to the NIST standardization process for NIST SP 800-38G.[6] A "letter of assurance"[7] was sent by Voltage to signify their intent, in case their submission (called FFX in the letter) were to be selected for inclusion in the NIST standard, to "make available a non-exclusive license, under reasonable rates with or without compensation" for use of their relevant patents. FFX was included in NIST SP 800-38G, as FF1. Therefore, the current intellectual property status of FF1 is unclear. NCC Group recommends that Electric Coin performs some additional legal analysis to ascertain the limitations that may arise from the use of FF1 in the NU5 protocol specification.
- **Section 5.4.9.6 (page 96):** the offered definition of a *short Weierstrass elliptic curve*, as a curve of equation $y^2 = x^3 + a \cdot x + b$ for two field elements a and b such that $4 \cdot a^3 + 27 \cdot b^2 \neq 0$, is valid only for finite fields of characteristic different from 2 and 3. For fields with characteristic 2 or 3, notions of "short Weierstrass curve" have been defined, but with different equation formats.
- **Section 5.4.9.6 (page 96):** the text uses $\mathbb{G}$ without introducing it. NCC Group recommends the addition of a sentence such as "Let $\mathbb{G}$ be either $\mathbb{P}$ or $\mathbb{V}$" as is done in section 5.4.9.8.
- **Section 5.4.9.6 (page 96):** the text implicitly switches between integers and field elements. For instance, the encoding of a point into bits uses "$x + 2^{255} \cdot \tilde{y}$", which should be computed over the integers ($\mathbb{N}$), not the finite field in which the point coordinates are defined (though the expression is mathematically defined in the finite field); a few lines later, a square root is extracted from the value "$x^3 + b_{\mathbb{G}}$", an expression which, this time, must be evaluated in the finite field. Explicit conversions between finite field elements and integers would help clarify the intent for implementers.
- **Section 5.4.9.6 (page 98):** In the sentence: "Define $\mathsf{abst}_{\mathbb{G}}$ [...] such that $\mathsf{abst}_{\mathbb{J}}(P\star)$ is computed as follows", the second "$\mathsf{abst}_{\mathbb{J}}$" should be "$\mathsf{abst}_{\mathbb{G}}$".

[5] https://zips.z.cash/protocol/nu5.pdf, retrieved on 2021-03-19, SHA-256 = fcfe26c583a23d9990bf3fdd836e86e7d8ecd1c89c5882a0516517ba65d21216

[6] https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-38G.pdf

[7] https://csrc.nist.gov/CSRC/media/Projects/Block-Cipher-Techniques/documents/BCM/proposed-modes/ffx/ffx-voltage-ip.pdf

- **Section 5.4.9.8 (page 98):** the proposed **hash_to_field** is supposed to align on the hash-to-curve draft[8]; this intent is explicitly stated on page 99. However, the NU5 specification diverges from the hash-to-curve draft in two respects:
  - The padding sequence "[`0x00`]^{64}" is a sequence of 64 bytes of value zero; it corresponds to the value `Z_pad` of the draft (section 5.4.1). However, the draft makes that string have the same length as the internal block size of the hash function used. Here, the hash function used is BLAKE2b-512, whose internal block size is 128 bytes (16 words of 64 bits, see RFC 7693[9]). Thus, the padding sequence should have length 128 bytes, not 64.
  - As per the hash-to-curve draft, extraction of each field element should use a sequence of `L = ceil((ceil(log 2(p)) + k) / 8)` bytes, with `p` being the field size, and `k` the "claimed security level" of the curve. For the Pallas and Vesta curves, `p` is slightly above 2^{254}. The proposed **hash_to_field** specification uses 64 bytes for each of the two field elements, which would imply a claimed security level `k` between 250 and 257 bits. This is way more than the theoretical resistance of the curves against generic attacks; Electric Coin's own blog explicitly claims "126-bit security against Pollard rho attacks".[10]
- **Section 5.4.9.8 (page 99):** the text defines the output space of **sqrt_ratio** to be the finite field $\mathbb{F}_{q_{\mathbb{G}}}$, but then immediately offers formulas that output a pair of values (a finite field element *and* an integer of value 0 or 1).
- **Section 5.4.9.8 (page 99):** the value $\lambda_{\mathbb{G}}$ is not provided anywhere in the specification. This is normal (the value vanishes through the computations, so the actual choice has no impact on the final output of **map_to_curve_simple_swu**), but this is not obvious and should probably be made explicit somewhere in the text.
- **Section 5.4.9.8 (page 100):** the input string D to **GroupHash** is formally a byte sequence of arbitrary length, but it is in practice limited because the total size of **DST** must not exceed 255 bytes (the operations in the hash-to-curve draft can use an arbitrary length domain separation tag through the use of an extra hashing operation when the tag is too large, but this specific construction has not been retained in the NU5 specification, which is thus limited to tags of 255 bytes or fewer).
- **Section 7.1 (page 117):** in the consensus rules, some spaces are missing after `flagsOrchard`, after `enableSpendsOrchard` (three times), and before `spendAuthSig`. Also, the second `enableSpendsOrchard` should be `enableOutputsOrchard` since that sentence is output transaction outputs. Finally, a "2^16" should be "2^{16}".
- **Section 7.3 (page 119):** a space is missing before `spendAuthSig`.

ZIPs review

NCC Group reviewed the following ZIPs: 216, 224, 225, 244 and 252. As for ZIP 252, it describes the deployment of the NU5 upgrade and NCC Group has not found any issues with it. See the review of the remaining ZIPs below.

ZIP 216

ZIP 216 modifies the rules concerning non-canonical encodings of some Jubjub points. Initially, the decoding function ($\mathsf{abst}_{\mathbb{J}}$) rejected any non-canonical encoding, but it was found that this did not match the behaviour of the implementation (zcashd), which accepted two non-canonical encodings, for point $(0, 1)$ and $(0, -1)$. The specification was thus amended in July 2020 to align with the implementation.[11] This behaviour was later found to be inconvenient for implementations, since consistency across decoding/encoding then requires maintaining the encoding variant information; a failure to do so was deemed likely to be undetected, and may lead to inadvertent forks. ZIP 216 thus reinstantiates the enforcement of canonical encodings, with a scheduled application at the activation height of Orchard.

ZIP 216 contains a list of references to the "orchard.pdf" file, which does not exist yet; these links are thus (currently) broken, but will become valid when the Orchard PDF file is published. The *text* of references `[6]` to `[9]`, however, is incorrect in that it contains the wrong section number, because section numbering changed in the most recent drafts of the protocol specification. The following mappings should be applied:

- Reference `[6]` (Jubjub): 5.4.8.3 → 5.4.9.3
- Reference `[7]` (Pallas and Vesta): 5.4.8.6 → 5.4.9.6

[8] https://tools.ietf.org/html/draft-irtf-cfrg-hash-to-curve-10
[9] https://tools.ietf.org/html/rfc7693#section-2.1
[10] https://electriccoin.co/blog/the-pasta-curves-for-halo-2-and-beyond/
[11] https://github.com/zcash/zips/commit/154da511c63f270427ae38efb693824f52f8900b

- Reference [8] (Sapling Payment Addresses): 5.6.4 → 5.6.3.1
- Reference [9] (Sapling Full Viewing Keys): 5.6.7 → 5.6.3.3

In the "Specification" section, the following intent is expressed:

> The above is intended to be a complete list of the places where compressed encodings of Jubjub points occur in the Zcash consensus protocol and in plaintext, address, or key formats.

Such a guarantee cannot be ensured indefinitely, unless the ZIP contents are actively updated to match future protocol versions, which would prevent the ZIP from reaching "Final" status. It might be more robust to indicate explicitly that the provided list relates to the Zcash consensus protocol *at the time of Orchard activation*.

ZIP 224

ZIP 224 specifies the use of the new Orchard shielded protocol. The protocol itself is not described in details in the ZIP; the complete description is in the network protocol document itself, with explanations and additional material in the halo2 and Orchard books, to which the ZIP links. The following remarks can be offered:

- In the "Nullifiers" section, the meaning of p in the formula is not defined. In the protocol specification, this is $q_{\mathbb{P}}$, i.e. the coordinate field of the Pallas curve; crucially, it is *not* the scalar field of Pallas, a non-trivial point that the protocol document specifically highlights, but that ZIP 224 glosses over. (This remark also applies to the Orchard book, section 3.5.)
- ZIP 224 uses (twice) the string "c/f" as an abbreviation for the Latin verb *conferatur*. The correct abbreviation is: "*cf.*".
- In "Commitments", the previous commitment mechanism is referred to as "Bowe--Hopwood" with two dash signs, but in "Commitment tree" a single dash sign is used instead, which is inconsistent.
- In "Proving system", the link to the Halo 2 protocol is missing (it is specified as "TODO").
- Due to changes in the draft specification document, section numbering has been modified, and some references in the table of references of ZIP 224 are now off:
  - Reference [6] (Mainnet and Testnet): 3.11 → 3.12
  - Reference [14] (RedDSA, RedJubjub, and RedPallas): 5.4.6 → 5.4.7
  - Reference [15] (Sinsemilla commitments): 5.4.7.4 → 5.4.8.4
  - Reference [16] (Pallas and Vesta): 5.4.8.6 → 5.4.9.6
  - Reference [17] (Group Hash into Pallas and Vesta): 5.4.8.8 → 5.4.9.8
  - References [18] to [21] point to sections 5.6.4.1 to 5.6.4.4 for encodings of Orchard addresses and keys; in the current specification draft, there are now *five* sections (5.6.4.1 to 5.6.4.5) that also describe the new unified payment address format, that Orchard addresses leverage.

ZIP 225

ZIP 225 lays out the V5 transaction format together with modifications to the transaction ID and signature digest specified in ZIP 244 . The new transaction format supports already existing pools and adds fields specific to Orchard. As mentioned, since the transaction structure is extended, this ZIP depends on ZIP 244 which specifies new methods for computing the transaction ID and Authorizing Data Commitments.

The following issues in the context of ZIP 225 can be observed:

- In new transaction format table on p. 114, the NU5 specification outlines that the size of `vSpendsSapling` and `vOutputsSapling` fields is `362*nSpendsSapling` and `948*nOutputsSapling`, respectively. This is different than in ZIP 225, which specifies the sizes of those fields as `128*nSpendsSapling` and `756*nOutputsSapling`. In V5 transactions, the size of the `SpendDescriptionV5` type is 96 bytes, since it contains three 32-byte values, according to the ZIP and section 7.3 of the spec. As for the `OutputDescriptionV5` type, the ZIP is correct in what the actual size is, however the spec includes fields that are only present in V4. NCC Group suggests updating the ZIP and the table on p. 114 of the spec to reflect the correct sizes.
- In ZIP 225, just below the transaction field table, `valueBalanceSapling`, `anchorSapling`, and `bindingSigSapling` fields are mentioned to be present if and only if `nSpendsSapling + nOutputsSapling > 0`. On the other hand,

the NU5 specification on p. 114 also includes `vSpendProofsSapling`, `vSpendAuthSigsSapling` and `vOutputProofsSapling` under that same condition, which appears to be a discrepancy between the spec and ZIP 225. Now, in the ZIP, `vSpendProofsSapling` and `vSpendAuthSigsSapling` are conditioned to have a 1:1 correspondence to the elements of `vSpendsSapling` and as for `vOutputProofsSapling`, it is conditioned to have a 1:1 correspondence to `vOutputsSapling`. However, the two 1:1 conditions aren't equivalent with the `nSpendsSapling + nOutputsSapling > 0` stated in the spec. In particular, the conditions in ZIP 225 do not explicitly state whether the field is present or not and, when it comes to the spec, it does not include the 1:1 condition and the sorting requirement. NCC Group suggests unifying the two sets of conditions in both the spec and ZIP 225.

- On page 114, the NU5 specification lists a number of action-related fields as present if and only if `nActionsOrchard > 0`. ZIP 225 does the same, but excludes the following fields: `flagsOrchard`, `sizeProofsOrchard`, `proofsOrchard` and `vSpendAuthSigsOrchard`. The ZIP does not mention any restrictions on `flagsOrchard` and `proofsOrchard`.
- In the "Orchard Action Description" section of ZIP 225, `ephemeralKey` is described to be a Pallas public key, while the spec states this is a Jubjub key. The spec should be corrected to state this is a Pallas key.
- ZIP 225 names the least significant and next to least significant bit inside the `flagsOrchard` as `spendsEnabledOrchard` and `outputsEnabledOrchard`, respectively. These bits are referred to differently in the specification: `enableSpendsOrchard` and `enableOutputsOrchard`
- In the "Sapling Output Description" section in ZIP 225, the data type of the `outCiphertext` "bytes" and "data type" columns don't match. The bytes field is stated to be `80` and data type `byte[580]`. The similar issue exists in the Orchard Action Description section of the same ZIP.
- The § symbol is used to refer to sections inside the NU5 spec, however this is not explicitly stated before the first such reference.

ZIP 244

In ZIP 244, the transaction ID computation is modified to exclude attestations to transaction validity, such as transaction signatures and proofs. In addition, a new way of computing the transaction digest for signature validation is specified. Finally, an existing block field is repurposed to contain new transaction commitments. It should be noted that ZIP 244 is Orchard-agnostic and that Orchard-specific amendments to transaction ID and signature hash can be found in ZIP 225 (Version 5 Transaction Format).

It is worth observing that:

- The ZIP 244 Requirements section contains a requirement that's not satisfied. The requirement is: "It should be possible to use the transaction id unmodified as the value that is used to produce a signature hash in the case that the transaction contains no transparent inputs, or in the case that only the `SIGHASH_ALL` flag is used.". However, if the transaction has transparent inputs and only `SIGHASH_ALL` flag is used, the transaction ID and signature hash will still not be the same. This is since the `txin_sig_digest` intermediate hash will be non-empty as long as there are transparent inputs, regardless of the actual signature flag used.
- Inside ZIP 225's section that's relevant to ZIP 244, there is an omitted field inside the `signature_digest` section. The `sprout_digest` is mentioned in the tree inside the `Signature Digest` section of ZIP 225, but not right next below in the `signature_digest` subsection.
- The `flagsOrchard` field does not participate in a transaction's signature hash. The flags control whether Orchard spends or outputs are enabled or not. In a transaction that's already without Orchard spends, it's possible to toggle the `enableOrchardSpends` flag without knowing any secrets. There do not appear to be any issues with this, since the transaction intent is not modified in any way. **Note:** As per the Zcash team, this flag was intended to participate inside the signature hash and the reviewed ZIP will be modified to reflect that.

ZIP 316

ZIP 316 bundles the transparent addresses (P2PKH and P2SH), and shielded addresses (i.e. Sapling and Orchard as of NU5) into a unified address encoding. With this approach the Receiver/Producer wallet will send its preferred addresses to the Sender's wallet and allows them to pick their preferred address type. This simplifies the communication between the Sender and Receiver, and allows wallets to be compatible with a range of older and newer wallets. By design, wallets that upgrade to support unified address format will ignore address types that they do not recognize,

which will result in forward compatibility as more address types are introduced.

The Open Issues and Known Concerns and Reference implementation sections of ZIP 316 are left as `TODO`.

ZIP 316 corresponds with section 5.6.4 of NU5 specification.

- It is mentioned that one of the goals of the unified addresses is to "Provide a "bridging mechanism" to allow shielded wallets to successfully interact with conformant Transparent-Only wallets.", however the ZIP does not describe how this bridging mechanism works, and in fact it emphasizes that unified addresses cannot be transparent only.
- This statement: "The string encoding is resilient against typos, transcription errors, cut-and-paste errors, unanticipated truncation, or other anticipated UX hazards." implies that the formatting has error correcting capabilities, when it can only detect errors via Bech32m's checksum. It is worth noting that, by design, Bech32m could be used to locate the position of a few substitution errors, however it is not advised to correct errors without notifying the user[12] as it could, unintentionally, result in losing funds.
- One of the unified address decoding rules is that the sender must reject a UA in which the same typecode appears more than once, and another rule is that it must ignore typecodes that it does not recognize. The ZIP's description does not clearly specify whether the first rule applies to typecodes that the Sender does not recognize or not.

[12] https://github.com/bitcoin/bips/blob/master/bip-0350.mediawiki#cite_ref-3-0

Appendix C: Implementation Review

This section includes notes and various remarks about the audit process of the implementations in scope. Some of these remarks could be considered issues to fix, but are not security vulnerabilities, even in a loose sense.

ZIP 216

The ZIP 216 implementation is straightforward. The patch is split over three pull requests:

- Primary fix: https://github.com/zkcrypto/jubjub/pull/46
  This small patch simply adds an explicit (and constant-time) test for the edge condition where $u = 0$ with a non-zero sign bit, and the `zip_216_enabled` flag is set.
- Fix integration into `librustzcash`: https://github.com/zcash/librustzcash/pull/396
  This patch adds the `zip216_enabled` flag to the API so that the new behaviour can be activated at a specific chain height.
- Consensus rule change in `zcash`: https://github.com/zcash/zcash/pull/5213
  The new `zip216_enabled` parameter is passed to conditionally enable the new behaviour when the call is part of the consensus and the current height reaches a specific activation threshold (outside of consensus rules, the new behaviour is unconditionally enforced).

These three patches collectively implement the provisions of ZIP 216.

ZIP 224

Implementation of the Pasta Curves

The implementation of the Pasta curves is in a dedicated crate, in the following repository and specific commit: https://github.com/zcash/pasta_curves/tree/d8547d2326b16b11b5c3e8ada231065111b51680

The code appears to be mathematically correct and behave as expected, with two caveats which have been detailed in specific findings:

- The point decoding and random point generation functions, as implemented, do not work correctly for the isogenous curves (iso-Pallas and iso-Vesta); this is not a critical issue because these curves are only used internally for hash-to-curve support (see finding NCC-E001151-002 on page 6).
- Some of the implementation appears to be aiming at constant-time behaviour, but fails to do so in a number of parts. The actual intent is not documented in the API; see finding NCC-E001151-003 on page 8.

Apart from that, the following remarks may be offered:

- **Square root computation description in the Halo 2 book:** the Halo 2 book contains a dedicated section that explains Sarkar's method for speeding up square root computations in the kind of field that the Pasta curves use. The `pasta_curves` repository contains a file (`book/src/design/implementation/fields.md`) which appears to have been later copied into the Halo 2 book as the source of that section. It contains some formulas that reference the value "$g^{2^{-24}}$", where g is a primitive 2^{32}-th root of unity modulo p (or q). This value is not defined, since g has order 2^{32}, and 2^{24} is not invertible modulo 2^{32}. The correct expression here would be "$g^{-2^{24}}$".
- **Code duplication:** the two fields $\mathbb{F}_p$ and $\mathbb{F}_q$ (base fields of Pallas and Vesta, respectively) are implemented into two separate but very similar files `src/fields/fp.rs` and `src/fields/fq.rs`. The two files are essentially the same code, modulo the replacement of type `Fp` with type `Fq`, a few constant changes, and some differences in the function that uses an addition chain to implement the first step of the square root. Code duplication is, in general, frowned upon, since it increases maintenance efforts. These two files could be merged into a single one by generating most of the functions through macros (in the same way as it is done for elliptic curves in `src/curves.rs`).
- **Undocumented ranges:** the field implementations rely on internal functions which have some specific range requirements, which are fulfilled in the code, but not documented; this may lead to issues if this code is later adapted to other finite fields. Namely, the following information should be explicitly stated in the `fp.rs` and `fq.rs` source files:
  - The field modulus must be lower than 2^{255}. A 256-bit modulus is not supported; otherwise, carries may be lost in additions (line 412) and Montgomery reductions (line 354).

  - While element values should normally be represented as integers in the 0 to $p - 1$ range (for a modulus p), some functions can accept larger operands. In particular, the `sub()` function tolerates that its second operand (but not the first) is equal to p; this is leveraged in the implementation of additions, as well as the conditional subtraction at the end of Montgomery reduction.
  - The `montgomery_reduce()` function takes as input a 512-bit integer, but it cannot work will all integers up to $2^{512} - 1$. Its actual maximum input value is $2^{256}p$, for a modulus p. This limit is larger than p^2; this fact is leveraged in the `from_u512()` function, which performs modular reduction of a 512-bit input and can, contrary to `montgomery_reduce()`, handle inputs up to $2^{512} - 1$.
- **Constant-time negation in the fields:** the `neg()` function, on input x, computes $p - x$, but must fix the value in case the input was zero (since p is not in the expected range of values). This fix uses at some point a Boolean value, converted to a 64-bit integer:

```
    let mask = (((self.0[0] | self.0[1] | self.0[2] | self.0[3]) == 0) as u64).wrapping_sub(1);
```

  Depending on the target architecture and the compiler flags, this construction *might* induce the compiler to use a conditional jump, since it is aware of the Boolean nature of that value. This should normally not happen on usual architectures (x86, ARM...) when optimizations are active. A possibly safer alternative would be to implement `neg()` by calling the `sub()` function, to subtract the operand from zero.
- **Range limitations in curves:** similarly to finite fields, the curve implementation macros in `src/curves.rs` have some inherent limitations with regard to the fields and scalars they may handle. In particular, the size of encoded points (32 bytes) is hardcoded (e.g. lines 226 and 623), with the top bit of the last byte being reserved for the sign of y (line 627); this effectively prevents use with a base field modulus larger than 255 bits. Similarly, when multiplying a curve point by a scalar, the top bit of the scalar is ignored (lines 453 and 564): scalars must fit on 255 bits too. The Pasta curves naturally comply with these limitations, but in the interest of preventing issues in case of reuse of this code for other curves, it is recommended that a couple of `assert!()` clauses be added, in order to check that both the coordinate and scalar fields have `NUM_BITS` constant values in the proper range (i.e. at most 255).
- **Unnecessary checks in point doubling:** the point doubling functions for points in Jacobian coordinates in `src/curves.rs` (lines 787 and 824) include a final constant-time check to specially handle the point-at-infinity. This check is not actually needed, because the doubling formulas are complete: if $z = 0$ on input, then the output will correctly have $z = 0$ as well. In fact, even if the curves contained points of order 2, the formulas would still work for these points, making the comments on lines 790 and 827 technically correct but irrelevant.

RedPallas Support

The `redjubjub` crate was forked in order to add some support code for RedPallas. The reviewed repository and commit were: https://github.com/str4d/redjubjub/tree/d5d8c5f3bb704bad8ae88fe4a29ae1f744774cb2

In this fork, only the `src/orchard.rs` file contained Orchard-specific modifications. These are simple functions that seem correct and in line with the Orchard specification. Of note, the `non_adjacent_form()` function (on line 81, defined on the Pallas scalar type) is an almost identical copy of the function with the same name in `src/scalar_mul.rs` (line 68); the two functions differ only in the type of scalar structure they attach to (Pallas vs Jubjub scalars) and the function call used to obtain the little-endian 32-byte representation of the scalar. In the interest of future code maintenance, it is recommended that these two functions be merged, at least in the source code (with a macro).

Implementation of the Orchard network upgrade

The implementation of the Orchard network upgrade is in a dedicated crate, in the following repository and specific commit: https://github.com/zcash/orchard/tree/1182d8d5a7e73644fe36d06af4a4727fc0544304

The whole crate was in scope, but in particular the following elements: Bundle and Action structures, Key structures, Notes, Commitments, Nullifiers, Note encryption, Bundle builder, Sinsemilla primitive, and Poseidon primitive and circuit gadget.

- **MerklePath's hash_layer() function could panic:** the helper function that hashes left and right nodes on a Merkle path unwraps the result of Sinsemilla hash, which could potentially panic and crash the calling process. This has a

very low probability of happening, as it will require to find a message that makes `SinsemillaHashToPoint` algorithm produce `None`. One of the conditions where that can happen is if Acc_{i-1} and $S(m_i)$, in the *SinsemillaHashToPoint* algorithm, are equal. This maps to the `hash_to_point_inner` function in `primitives/sinsemilla.rs` (line 124). It is worth emphasizing that the probability of that happening is very low; see Theorem 5.4.4 for proof.

Update: Since this note was reported, Orchard's Merkle authentication path calculation has been updated to map the hash outputs that are `None` to zero (Relevant Orchard Pull Request). As a result Merkle path calculation will no longer panic and crash the calling process, and therefore this note has been addressed. The specification has also been updated to allow zero to be a valid Merkle tree node (see Zcash Protocol Specification).

- **Sinsemilla hash API does not check the message's length:** the Sinsemilla hash function assumes the message length is less than or equal to $K * C$, but it does not explicitly check it. As a result if the `msg`'s length is too large `assert!(self.len <= K * C);` will panic. This code is only accessible via the `MerklePath`'s `root()` API at the moment, which hashes Merkle tree nodes (Pallas bases) and therefore is fine, but it is an unexpected behaviour for a hash function to panic on large messages. We suggest that this assumption be documented or return an error code.
- **Undefined padding rule:** in the protocol specification (`nu5.pdf`), the pad() function is defined with three steps, the first of which stating that the input bit string M shall be padded to $n \cdot k$ bits; however, it is not said what value the additional bits should have, nor where they should be added. A previous version of the document specified that the padding should be performed by appending bits of value zero, but these details have been removed. The implementation indeed adds the extra bits of value zero after the input M.
- **Non-genericity of Poseidon implementation:** the `src/primitives/poseidon.rs` file defines numerous parameterized traits and implementations to define generic sponge constructions that can potentially accommodate various values of the state size (`T` parameter) and the rate (`RATE` parameter), both expressed in number of field elements. The sponge **capacity**, by definition, is the difference of these two values (`T - RATE`). However, on lines 172-174, the starting state is initialized with an implicit assumption that the capacity is equal to 1, since only a single state element is set to the relevant value:

```
let input = [None; RATE];
let mut state = [F::zero(); T];
state[RATE] = initial_capacity_element;
```

  If this code were to be used with different parameters that lead to a capacity of 2 or more, then the initialization would be incorrect (but it would still compile successfully).
- **Unused function:** in `src/constants.rs`, the `find_zs_and_us()` function is not public, and never called anywhere.
- **Default value assumption:** in `src/constants/util.rs`, on line 40, the evaluation of a polynomial with given coefficients uses Horner's method, starting with an accumulator variable (`acc`) initialized with the default value for the field. This implicitly assumes that the default value of the field is zero. Since the used trait for the field includes a function called `zero()` that explicitly returns a zero, it would be clearer and more robust to use that function instead of `default()`.
- **Duplicated constants:** in `src/spec.rs`, the domain separation strings (`"z.cash:Orchard-CommitIvk"` on line 174, and `"z.cash:Orchard-gd"` on line 195) are given as literal strings, instead of using the constants `COMMIT_IVK_PERSONALIZATION` and `KEY_DIVERSIFICATION_PERSONALIZATION` defined in `src/constants.rs`.

Unified Address Support

The support for parsing and encoding Unified Addresses (as outlined in ZIP 316) was added to the `librustzcash` repo. The reviewed repository and commit hash were: https://github.com/zcash/librustzcash/tree/4ac5977c913ab123eea5ee4323b170e0ff659c8f. More specifically NCC Group reviewed 2 Pull Requests: https://github.com/zcash/librustzcash/pull/383 and https://github.com/zcash/librustzcash/pull/352.

The implementation closely follows ZIP 316's description with the exception of finding NCC-E001151-004 on page 4. It is worth noting that the implementation does not yet support encoding or decoding a unified `ivk` or `fvk`. Another minor remark is that `Typecode` constants (values `0x00` to `0x03` for receiver types) are included as literal integers in two separate functions (`Receiver`'s `TryFrom` implementation and its `typecode` getter); using symbolic constants, defined

in a single place, would improve maintainability by reducing the risk of typographic errors leading to the two functions using different values.

ZIP 244

The implementation of ZIP 244 was reviewed as in the following commit. Three different ways to compute transaction digests are introduced: TxId digest, Signature Digest and Authorizing Data Commitment. A modification to the semantics and a renaming of the `hashLightClientRoot` block field is also introduced. The implementation review looked for any value omissions in digest computation, discrepancies between the implementation and specification and DoS vectors common in deserialization code. No issues have been identified in the implementation of ZIP 244.

As for the TxId digest, it is realized in code by the `txid::TxIdDigester` implementation. ZIP 244 mentions that the `transparent_digest` hash is based on `prevout_digest`, `sequence_digest` and `output_digest` values, however, the method that computes `transparent_digest` also includes the `per_input_digest` field, if it exists. This is not an issue since the transparent digest's `per_input_digest` is set to `None` before being passed to the corresponding function. If Orchard actions are non-existent, the `orchard_digest` field is just the personalization field hash: this is ensured using the `hash_sapling_txid_empty` method.

When it comes to the Authorizing Data Commitment, it is realized in code by the `txid::BlockTxCommitmentDigester` implementation. ZIP 244 mentions that `ZTxAuthHash_ || CONSENSUS_BRANCH_ID` should be used as a personalization string for the commitment. This is implemented inside the `BlockTxCommitmentDigester::combine` method. It is worth noting that ZIP 244 mentions that only if Orchard Actions are present in the Orchard-related transaction field, `orchard_auth_digest` will be computed on non-static data. Technically, even if Orchard bundle's actions are empty, the Authorizing Data Commitment implementation will include the `zkproof` and `binding_signature` fields. This is not an issue since Orchard entries with empty actions are rejected by the `read_v5_orchard` method (as also specified by ZIP 244).

As for the Signature Digest, it implemented by `sighash_v5::v5_signature_hash`. The key part discussed in the ZIP is in the `transparent_input_sigdigests` function. The review effort was on that function, since the remaining aspects of the Signature Digest computation are similar to the previously discussed digest computations.

ZIP 225

The implementation of ZIP 225 was reviewed in the following commit. As for the threat model around ZIP 225, it was noted that there has been a fair amount of code refactoring to support the new `TransactionData` memory structure. Several `TransactionData` fields have been packed into bundles. For example, pre NU5-release, the `TransactionData` fields included:

```
    pub value_balance: Amount,
    pub shielded_spends: Vec<SpendDescription>,
    pub shielded_outputs: Vec<OutputDescription>,
    // ....
    pub binding_sig: Option<Signature>
```

In NU5, these `TransactionData` fields are bundled, see sapling.rs:

```
pub struct Bundle<A: Authorization> {
    pub shielded_spends: Vec<SpendDescription<A>>,
    pub shielded_outputs: Vec<OutputDescription<A::Proof>>,
    pub value_balance: Amount,
    pub authorization: A,
}
```

With respect to V4 transactions, the NU5 release needs to re-implement the legacy behaviour, but with the new `TransactionData` "bundled" memory format. This introduces a premature fork risk.

For example, the `read_v4` function in NU5 needs to behave exactly in the same way as the pre-NU5 `read` method. This was an important aspect of the ZIP 225 implementation review, however, no issues were identified. The `read_v4` function was refactored to call a number of sub-functions, e.g., in NU5, reading transparent inputs has been replaced by a specific `read_transparent` function. NCC Group spent time validating that the code is fully equivalent and no issues have been found.

As an aside, it is worth mentioning that the ZIP states that for coinbase transactions, the `enableSpendsOrchard` Orchard flag must be set to `0`, however this is not enforced at the `librustzcash` level.