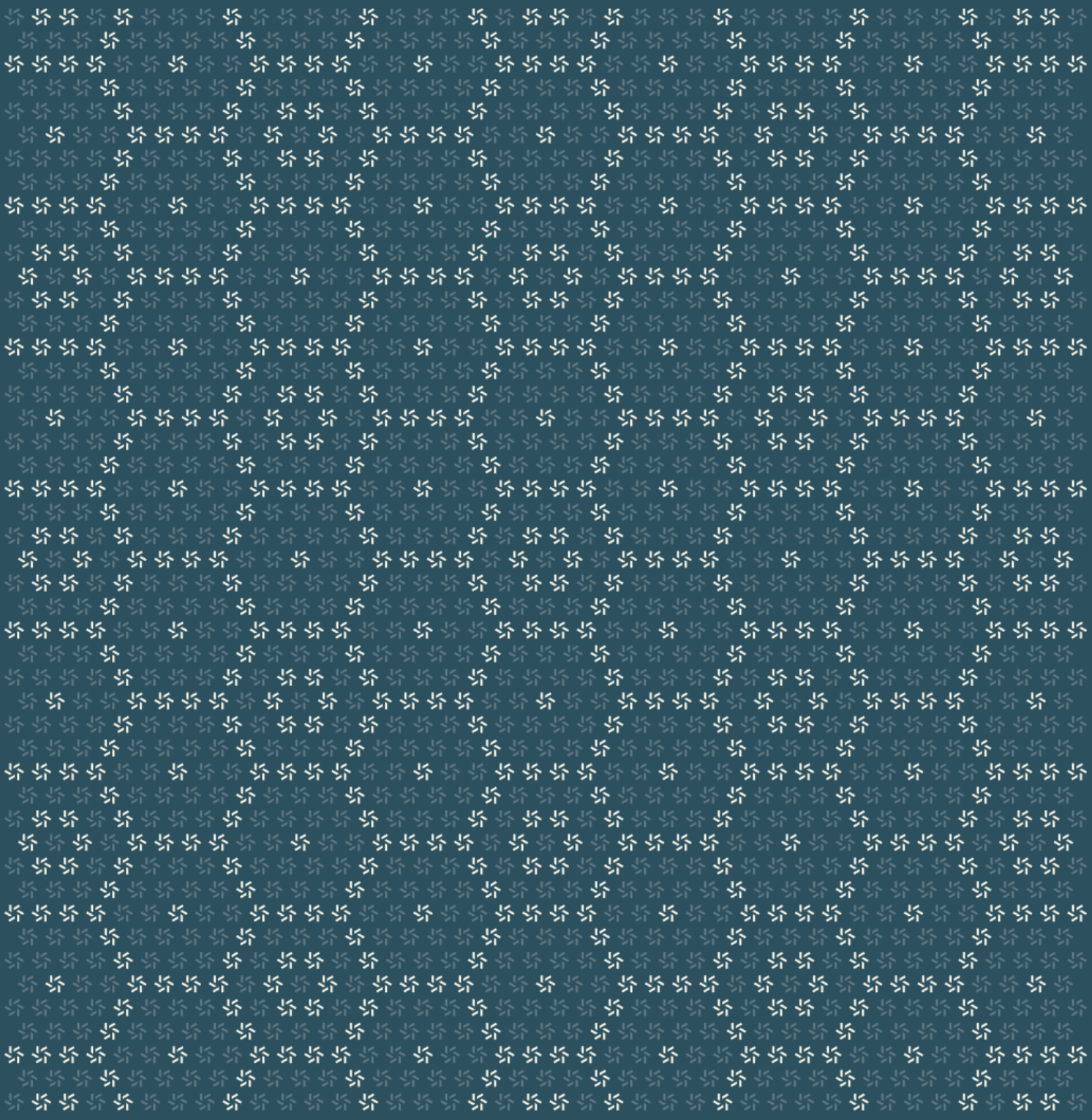
Zellic

Prepared for
Dev Ojha
Valar Group

Prepared by
Mark Pedron
Mohit Sharma
Zellic

August 14, 2026

Zcash Sapling
Cryptographic Security Assessment



Contents

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

1. Overview

1.1. Executive Summary

Zellic conducted a best-effort security assessment for Valar Group from June 6th to July 23rd, 2026. During this engagement, Zellic reviewed Zcash Sapling's code for security vulnerabilities, design issues, and general weaknesses in security posture on a time-boxed, best effort basis.

1.2. Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Does the code in the audit scope implement Zcash Sapling bundle verification according to its specification?
- In particular, do the arithmetic circuits correctly enforce the Spend and Output statements?

1.3. Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Whether the Zcash Sapling design enforces its design goals
- Front-end components
- Infrastructure relating to the project
- Key custody

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide. In addition, this assessment was performed on a best-effort basis within Valar Group's provided time frame, so it should not be considered a comprehensive audit and cannot guarantee the discovery of all potential vulnerabilities or issues.

Our review focused on the critical components of the Sapling repository, along with the portions of the Bellman repository that the Sapling repository depends on, with a particular focus on soundness issues that could lead to loss of funds rather than concerns such as loss of privacy.

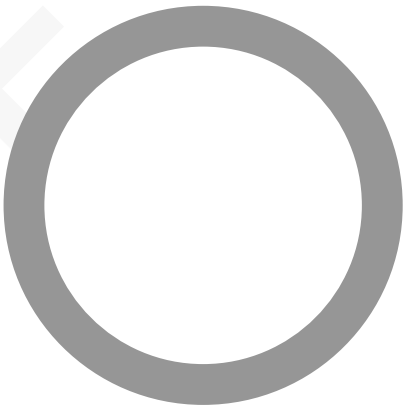
1.4. Results

During our assessment on the scoped Zcash Sapling targets, we discovered six findings, all of which were informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for the benefit of Valar Group in the Discussion section (4. ↗).

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	0
Medium	0
Low	0
Informational	6



2. Introduction

2.1. About Zcash Sapling

Valar Group contributed the following description of Zcash Sapling:

> Sapling is a network upgrade that introduces significant efficiency improvements for shielded transactions that will pave the way for broad mobile, exchange and vendor adoption of Zcash shielded addresses.

2.2. Methodology

During a security assessment, Zellic works through standard phases of security auditing, including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Protocol design. For novel/nonstandard protocols, we investigate the core design for potential flaws. This does not, however, apply to implementations of established protocols and algorithms. For example, when auditing a library for Yao's garbled circuits, we would not spend time on the garbling scheme itself. On the other hand, the white paper for a novel protocol would be very much considered in scope and examined thoroughly.

Applied cryptography. Zellic has a dedicated team of strong theoretical and applied cryptographers. Implementing cryptographic applications securely, like Web3 wallets, is incredibly difficult, and we help clients navigate a minefield of potential pitfalls and mistakes. We have ensured secure implementation of noncustodial wallets, ERC-4337 (AA), MPC, Shamir's secret sharing (SSS), EOAs, native multi-sig support, enclave solutions, social log-in, and key recovery, among other technologies.

Cryptographic components. We manually review the cryptography usage of the project and examine the relevant studies and standards for any inconsistencies or vulnerabilities. We ensure all of the cryptographic primitives used in the project are used and configured correctly.

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. For cryptographic applications, this includes insufficient checks on the inputs, incorrect handling of edge cases, and completeness issues in low-level arithmetic operations.

Code maturity. We look for potential improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped targets itself. These observations — found in the Discussion (4. ↗) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3. Scope

The engagement involved a review of the following targets:

Zcash Sapling Targets

Type	Rust
Platform	Zcash
Target	sapling-crypto
Repository	https://github.com/zcash/sapling-crypto ↗
Version	c5e596c239dbf9138a74246b843bcd01413f51c7
Programs	circuit/constants.rs circuit/ecc.rs circuit/pedersen_hash.rs circuit.rs verifier.rs verifier/single.rs verifier/batch.rs pedersen_hash.rs group_hash.rs spec.rs note.rs note/commitment.rs note/nullifier.rs value.rs value/sums.rs keys.rs bundle.rs - only verification parts constants.rs

Target	bellman
Repository	https://github.com/zkcrypto/bellman ↗
Version	`e137775023a647716793a362ace008e058679b2a`

Programs	`gadgets.rs`
	`gadgets/*.rs - specific to the parts used by the sapling-crypto circuits`
	`groth16/verifier.rs`
	`groth16/verifier/batch.rs`

2.4. Project Overview

Zellic was contracted to perform a best-effort security assessment for a total of 6.8 person-weeks. The assessment was conducted by two consultants over the course of 7 calendar weeks.

Contact Information

The following project manager was associated with the engagement:

Jacob Goreski
Engagement Manager
jacob@zellic.io ↗

The following consultants were engaged to conduct the assessment:

Mark Pedron
Engineer
mark.pedron@zellic.io ↗

Mohit Sharma
Engineer
mohit@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

June 6, 2026	Start of primary review period
June 8, 2026	Kick-off call
July 23, 2026	End of primary review period

3. Detailed Findings

3.1. Dependence on implementation-specific behavior: `PrimeField` byte representation

Target	sapling_crypto		
Category	Code Maturity	**Severity**	Medium
Likelihood	N/A	**Impact**	Informational

Description

In the trait `ff::PrimeField`, the documentation of the functions `to_repr` and `from_repr` states that the byte representation is implementation-specific and should be treated as opaque:

```
// ff-0.13.0 - src/lib.rs
pub trait PrimeField: Field + From<u64> {
    /// ...
    /// Attempts to convert a byte representation of a field element into an
    element of
    /// this prime field, failing if the input is not canonical (is not
    smaller than the
    /// field's modulus).
    ///
    /// The byte representation is interpreted with the same endianness as
    elements
    /// returned by [`PrimeField::to_repr`].
    fn from_repr(repr: Self::Repr) -> CtOption<Self>;
    /// ...
    /// Converts an element of the prime field into the standard byte
    representation for
    /// this field.
    ///
    /// The endianness of the byte representation is implementation-specific.
    Generic
    /// encodings of field elements should be treated as opaque.
    fn to_repr(&self) -> Self::Repr;
```

The implementations for `jubjub::Fr` (alias: `jubjub::Scalar`) and `bls12_381::Scalar` do not document a stronger or differing claim:

```
// jubjub-0.10.0 - src/fr.rs
impl PrimeField for Fr {
    type Repr = [u8; 32];
```

```
    fn from_repr(r: Self::Repr) -> CtOption<Self> {
        Self::from_bytes(&r)
    }

    fn to_repr(&self) -> Self::Repr {
        self.to_bytes()
    }
```

```
// bls12_381-0.8.0 - src/scalar.rs
impl PrimeField for Scalar {
    type Repr = [u8; 32];

    fn from_repr(r: Self::Repr) -> CtOption<Self> {
        Self::from_bytes(&r)
    }

    fn to_repr(&self) -> Self::Repr {
        self.to_bytes()
    }
```

The audited code calls these functions directly in several places and relies on the fact that the byte encoding is always the little-endian byte encoding of the canonical modular representative:

```
// sapling-crypto - src/keys.rs
pub struct ExpandedSpendingKey {
    pub ask: SpendAuthorizingKey,
    pub nsk: jubjub::Fr,
    pub ovk: OutgoingViewingKey,
}
// ...
impl ExpandedSpendingKey {
    // ...
    pub fn to_bytes(&self) -> [u8; 96] {
        let mut result = [0u8; 96];
        result[0..32].copy_from_slice(&self.ask.to_bytes());
        result[32..64].copy_from_slice(&self.nsk.to_repr());
        result[64..96].copy_from_slice(&self.ovk.0);
        result
    }
```

```
// sapling-crypto - src/keys.rs
pub struct SaplingIvk(pub jubjub::Fr);
// ...
```

```
impl SaplingIvk {
    // ...
    pub fn to_repr(&self) -> [u8; 32] {
        self.0.to_repr()
    }
```

```
// sapling-crypto - src/keys.rs
impl SpendAuthorizingKey {
    // ...
    pub(crate) fn from_bytes(bytes: &[u8]) -> Option<Self> {
        <[u8; 32]>::try_from(bytes)
            .ok()
            .and_then(|b| {
                // ...
                jubjub::Scalar::from_repr(b)
```

```
// sapling-crypto - src/keys.rs
pub struct SpendAuthorizingKey(redjubjub::SigningKey<SpendAuth>);
// ...
impl SpendAuthorizingKey {
    // ...
    pub(crate) fn to_scalar(&self) -> jubjub::Scalar {
        jubjub::Scalar::from_repr(self.0.into()).unwrap()
    }
```

```
// sapling-crypto - src/keys.rs
impl ExpandedSpendingKey {
    // ...
    pub fn from_bytes(b: &[u8]) -> Result<Self, DecodingError> {
        // ...
        let nsk =
    Option::from(jubjub::Fr::from_repr(b[32..64].try_into().unwrap()))
```

```
// sapling-crypto - src/note/commitment.rs
pub struct ExtractedNoteCommitment(pub(super) bls12_381::Scalar);

impl ExtractedNoteCommitment {
    // ...
    pub fn from_bytes(bytes: &[u8; 32]) -> CtOption<Self> {
        bls12_381::Scalar::from_repr(*bytes).map(ExtractedNoteCommitment)
    }
    // ...
    pub fn to_bytes(self) -> [u8; 32] {
```

```
        self.0.to_repr()
    }
```

```
// sapling-crypto - src/pedersen_hash.rs
pub fn pedersen_hash<I>(personalization: Personalization, bits: I) ->
    jubjub::SubgroupPoint
// ...
{
    // ...
    loop {
        let mut acc = jubjub::Fr::zero();
        // ...
        let acc = acc.to_repr();
```

```
// sapling-crypto - src/spec.rs
pub(crate) fn crh_ivk(ak: [u8; 32], nk: [u8; 32]) -> jubjub::Scalar {
    let mut h: [u8; 32] = // ...
    // ...
    jubjub::Fr::from_repr(h).unwrap()
}
```

```
// sapling-crypto - src/value.rs
impl ValueCommitTrapdoor {
    // ...
    pub fn from_bytes(bytes: [u8; 32]) -> CtOption<Self> {
        jubjub::Scalar::from_repr(bytes).map(ValueCommitTrapdoor)
    }
```

```
// sapling-crypto - src/verifier.rs
impl SaplingVerificationContextInner {
    // ...
    fn check_output(
        // ...
    ) -> bool {
        // ...
        public_input[4]
    = bls12_381::Scalar::from_repr(cmu.to_bytes()).unwrap();
```

Impact

Future implementations of `jubjub` and `bls12_381` could change the encoding for `to_repr` and `from_repr`. The calling functions would then deviate from their specified behavior. The consequences vary by caller; for example, a deviation in `pedersen_hash` would result in consensus-breaking behavior.

Recommendations

The functions `from_bytes` and `to_bytes` guarantee that their encoding is as expected by the calling code. Use these functions instead of `from_repr` and `to_repr`. In both crates, the versions of `from_repr` and `to_repr` in use are implemented via `from_bytes` and `to_bytes`, so this change does not alter the code's behavior.

Remediation

TBD

3.2. Incoming viewing key `ivk` not range checked

Target	sapling_crypto::keys		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The Zcash Protocol Specification ↗ states:

> `ivk` MUST be in the range $\{1 \ldots 2^{l_{ivk}} - 1\}$ as specified in section 4.2.2 'Sapling Key Components' on page 36. That is, a decoded incoming viewing key MUST be considered invalid if `ivk` is not in this range.

The code does not enforce this range condition in several places, most importantly during decoding. Note that the constant l_{ivk} has value 251. The struct `SaplingIvk` simply wraps an $\mathbb{F}_{r_J}$ scalar and does not have a dedicated constructor, so it lacks any additional range validation.

```
// src/keys.rs
#[derive(Debug, Clone, PartialEq, Eq)]
pub struct SaplingIvk(pub jubjub::Fr);
```

Decoding

The `SaplingIvk` struct is wrapped in the `IncomingViewingKey` struct, which is parsed:

```
// src/zip32.rs
#[derive(Clone, Debug, PartialEq, Eq)]
pub struct IncomingViewingKey {
    dk: DiversifierKey,
    ivk: SaplingIvk,
}

impl IncomingViewingKey {
    /// Parses a `IncomingViewingKey` from its raw byte encoding.
    ///
    /// Returns `None` if the bytes do not contain a valid encoding of a
    diversifiable
```

```
    /// Sapling incoming viewing key.
    pub fn from_bytes(bytes: &[u8; 64]) -> CtOption<Self> {
        jubjub::Fr::from_bytes(&bytes[32..].try_into().unwrap()).map(|fr|
    IncomingViewingKey {
            dk: DiversifierKey::from_bytes(bytes[..32].try_into().unwrap()),
            ivk: SaplingIvk(fr),
        })
    }
```

The decoding code does not check the range condition.

Dummy `OutputInfo`

Additionally, the construction of dummy Sapling `OutputInfo` structs makes use of invalid `ivk` values:

```
// src/builder.rs
impl OutputInfo {
    /// Constructs a new dummy Sapling output.
    pub fn dummy<R: RngCore>(mut rng: &mut R) -> Self {
        // This is a dummy output
        let dummy_to = {
            let mut diversifier = Diversifier([0; 11]);
            loop {
                rng.fill_bytes(&mut diversifier.0);
                let dummy_ivk = SaplingIvk(jubjub::Fr::random(&mut rng));
                if let Some(addr) = dummy_ivk.to_payment_address(diversifier)
    {
                    break addr;
                }
            }
        };

        Self::new(None, dummy_to, NoteValue::ZERO, [0u8; 512])
    }
```

This is not an encoding issue; at this point, the `ivk` is only an implementation detail used to derive a type-correct payment address. The Zcash Protocol Specification ↗ only states:

> As in Sprout, a dummy Sapling output note is constructed as normal but with zero value, and sent to a random shielded payment address.

From this specification, it is not clear whether a *random shielded payment address* must

correspond to a valid (random) `ivk`.

Tests

Several tests and one doc example in src/note_encryption.rs use random $\mathbb{F}_{r_J}$ values for `ivk`, which are likely to be larger than `2^l_ivk`.

Impact

No impact on consensus or on keys derived by this library. Both derivation paths mask the value into the required range: `crh_ivk` clears the top five bits of the BLAKE2s output before the scalar conversion, and the circuit truncates the in-circuit `ivk` bits to `jubjub::Fr::CAPACITY` (251) before the `pk_d = g_d^ivk` computation. An out-of-range `ivk` therefore cannot be produced by `FullViewingKey::ivk`, and cannot be smuggled into a proof.

The missing check only affects `ivk` values entering the library from outside, via `IncomingViewingKey::from_bytes`, which accepts any canonical $\mathbb{F}_{r_J}$ element — including the roughly 32 out of every 33 scalars that exceed $2^{l_{ivk}} - 1$. Consequences are limited to:

- **Spec deviation.**

  The specification requires such an encoding to be rejected.

  Callers that rely on `from_bytes` for validation of user-supplied unified-key data will accept keys that another conformant implementation rejects, so the same encoding is interoperable in one direction only.
- **Keys with no corresponding spending key.**

  An accepted out-of-range `ivk` still yields type-correct payment addresses via `address_at`/`address`, and round-trips through `to_bytes`, but no `FullViewingKey` can derive it.

  A wallet importing such a key would present addresses that cannot be linked back to any spend authority and whose incoming notes it can detect but never spend.

Recommendations

Add a constructor to `SaplingIvk` that checks the range condition, and use it consistently at all decoding boundaries. The dummy-output and test uses discussed above would then need an explicit escape hatch.

Remediation

TBD

3.3. Missing `esk == 0` handling for pre-ZIP-212 notes

Target	sapling_crypto::note		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

For pre-ZIP-212 Sapling notes, the Zcash Protocol Specification section 4.7.2 ↗ requires the ephemeral private key `esk` to be sampled uniformly from $\mathrm{KA}^{\mathrm{Sapling}}.\mathrm{Private} \setminus \{0\}$.

The implementation represents pre-ZIP-212 note randomness using `Rseed::BeforeZip212`. `Note::derive_esk` returns `None` for this variant:

```
// src/note.rs
pub(crate) fn derive_esk(&self) -> Option<EphemeralSecretKey> {
    match self.rseed {
        Rseed::BeforeZip212(_) => None,
        Rseed::AfterZip212(rseed) =>
    Some(EphemeralSecretKey(jubjub::Fr::from_bytes_wide(
            &PrfExpand::SAPLING_ESK.with(&rseed),
        ))),
    }
}
```

`Note::generate_or_derive_esk_internal` handles this `None` by sampling a random Jubjub scalar:

```
// src/note.rs
pub(crate) fn generate_or_derive_esk_internal<R: RngCore>(
    &self,
    rng: &mut R,
) -> EphemeralSecretKey {
    match self.derive_esk() {
        None => EphemeralSecretKey(jubjub::Fr::random(rng)),
        Some(esk) => esk,
    }
}
```

`jubjub::Fr::random` samples from the complete scalar field, including zero. Therefore, the pre-ZIP-212 branch can return `EphemeralSecretKey(0)` instead of sampling from the required

nonzero subset.

If `esk == 0`, the ephemeral public key is $epk = [esk]g_d = [0]g_d = \mathcal{O}_J$. The Sapling verifier rejects small-order `epk` values:

```
// src/verifier.rs
if epk.is_small_order().into() {
    return false;
}
```

An affected output therefore cannot be accepted as part of a valid transaction.

Impact

Assuming a cryptographically secure random-number generator, the probability of sampling zero is approximately $1/r_J$. If zero is sampled, transaction construction can complete successfully. However, the resulting transaction is rejected during Sapling verification because the identity `epk` is a small-order point.

Recommendations

Implement rejection sampling in the `None` branch of `generate_or_derive_esk_internal`, repeating the sampling operation until a nonzero scalar is obtained.

Remediation

TBD

3.4. Incomplete handling of `ask == 0` and `ivk == 0`

Target	sapling_crypto::note / sapling_crypto::keys / sapling_crypto::zip32		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The Zcash Protocol Specification ↗ requires `sk` to be resampled when either the derived `ask` or `ivk` is zero.

Section 4.8.2 instructs dummy-note generation to choose a random `sk` and to generate the `ak` and `nk` components, as well as a diversified payment address, as described in section 4.2.2. It is not explicit whether this cross-reference incorporates the resampling requirements from the complete key-generation procedure. However, tracing the implementation from the top-level dummy-note generation path shows that neither condition causes `sk` to be resampled. Instead, both conditions eventually cause a panic.

The top-level dummy-note generation function is:

```
// src/note.rs
pub(crate) fn dummy<R: RngCore>(mut rng: R) -> (ExpandedSpendingKey,
    FullViewingKey, Self) {
    let mut sk_bytes = [0; 32];
    rng.fill_bytes(&mut sk_bytes);

    let extsk = ExtendedSpendingKey::master(&sk_bytes[..]);
    let fvk = extsk.to_diversifiable_full_viewing_key().fvk().clone();
    let recipient = extsk.default_address();

    let mut rseed_bytes = [0; 32];
    rng.fill_bytes(&mut rseed_bytes);
    let rseed = Rseed::AfterZip212(rseed_bytes);

    let note = Note::from_parts(recipient.1, NoteValue::ZERO, rseed);

    (extsk.expsk, fvk, note)
}
```

`ask == 0`

During dummy-note generation, the relevant spending-key derivation path is:

```
Note::dummy→
 ExtendedSpendingKey::master→
 ExpandedSpendingKey::from_spending_key→
 SpendAuthorizingKey::from_spending_key→
 SpendAuthorizingKey::from_scalar
```

`SpendAuthorizingKey::from_scalar` detects `ask == 0` and returns `None`:

```
// src/keys.rs
pub(crate) fn from_scalar(ask: jubjub::Scalar) -> Option<Self> {
    if ask.is_zero().into() {
        None
    } else {
        Some(SpendAuthorizingKey(ask.to_bytes().try_into().unwrap()))
    }
}
```

However, `ExpandedSpendingKey::from_spending_key` calls `expect` on this result:

```
// src/keys.rs
pub fn from_spending_key(sk: &[u8]) -> Self {
    let ask =
        SpendAuthorizingKey::from_spending_key(sk).expect("negligible chance
    of ask == 0");
    // ...
}
```

Consequently, `ask == 0` causes an immediate panic at the `expect`, and `sk` is not resampled.

`ivk == 0`

After deriving the extended spending key, `Note::dummy` requests its default payment address by calling `extsk.default_address()`.

The relevant call path is:

```
Note::dummy→
 ExtendedSpendingKey::default_address→
 DiversifiableFullViewingKey::default_address→
```

```
DiversifiableFullViewingKey::to_external_ivk→
DiversifiableFullViewingKey::to_ivk→
ViewingKey::ivk→
IncomingViewingKey::find_address→
SaplingIvk::to_payment_address
```

`ViewingKey::ivk` wraps the output of `crh_ivk` without checking whether it is zero:

```
// src/keys.rs
pub fn ivk(&self) -> SaplingIvk {
    SaplingIvk(crh_ivk(self.ak.to_bytes(), self.nk.0.to_bytes()))
}
```

If `crh_ivk` returns zero, address generation computes $pk_d = [ivk]g_d = [0]g_d = \mathcal{O}_J$. `DiversifiedTransmissionKey::derive` returns `Some(DiversifiedTransmissionKey(identity))`, after which `PaymentAddress::from_parts` rejects the identity point:

```
// src/address.rs
pub(crate) fn from_parts_unchecked(
    diversifier: Diversifier,
    pk_d: DiversifiedTransmissionKey,
) -> Option<Self> {
    if pk_d.is_identity() {
        None
    } else {
        Some(PaymentAddress { pk_d, diversifier })
    }
}
```

The resulting `None` propagates through `SaplingIvk::to_payment_address` and `IncomingViewingKey::find_address`. `DiversifiableFullViewingKey::default_address` then calls `unwrap` on it:

```
// src/zip32.rs
pub fn default_address(&self) -> (DiversifierIndex, PaymentAddress) {
    self.to_external_ivk().find_address(0u32).unwrap()
}
```

Consequently, `ivk == 0` also causes the dummy-note generation path to panic instead of resampling `sk`.

Impact

Neither zero-valued key is retained in a successfully generated dummy note. Instead, the operation unwinds or terminates the process, depending on the panic configuration. This is inconsistent with the resampling behavior in section 4.2.2.

Assuming a cryptographically secure random-number generator and the standard pseudorandomness assumptions for the key-derivation functions, the probabilities are negligible: $\frac{1}{r_J}$ for `ask == 0` and 2^{-251} for `ivk == 0`. Even if either condition occurs, the panic prevents transaction construction from completing.

Recommendations

Make `ExpandedSpendingKey::from_spending_key` fallible when `ask == 0`, returning the failure to its caller rather than calling `expect`.

In `Note::dummy`, wrap the initial `sk_bytes` sampling and key derivation in a resampling loop. Use a fallible address-generation path, such as `find_address`, and resample the initial key material if the derived `ivk` is zero or address generation otherwise fails. This allows both exceptional conditions to resample `sk` instead of reaching `expect` or `unwrap`.

Remediation

TBD

3.5. Duplicate entries in linear combination not handled correctly if they cancel out

Target	bellman::LinearCombination		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

Bellman constraints are specified by explicitly defining the A, B, and C vectors of the R1CS constraint, which includes a coefficient for each variable in the constraint. These vectors are tracked by the `LinearCombination` type defined in src/lib.rs. Since `LinearCombination` is an append-only structure, a user-defined circuit can have a constraint in which a linear combination holds multiple entries for the same variable using different scalars. This is expected behavior and is used by the `AllocatedBit::xor` gadget.

```
cs.enforce(
    || "xor constraint",
    |lc| lc + a.variable + a.variable,
    |lc| lc + b.variable,
    |lc| lc + a.variable + b.variable - result_var,
);
```

However, an edge case occurs when the different coefficients of a variable cancel out to zero, as in a constraint such as `lc + x - x`. Parameter generation handles this correctly, but the prover errors out during proof generation. The prover's `eval` function is defined as follows:

```
fn eval<S: PrimeField>(
    lc: &LinearCombination<S>,
    mut input_density: Option<&mut DensityTracker>,
    mut aux_density: Option<&mut DensityTracker>,
    input_assignment: &[S],
    aux_assignment: &[S],
) -> S {
    let mut acc = S::ZERO;

    for &(index, coeff) in lc.0.iter() {
        let mut tmp;

        if !coeff.is_zero_vartime() {
            match index {
```

```
                Variable(Index::Input(i)) => {
                    tmp = input_assignment[i];
                    if let Some(ref mut v) = input_density {
                        v.inc(i);
                    }
                }
                Variable(Index::Aux(i)) => {
                    tmp = aux_assignment[i];
                    if let Some(ref mut v) = aux_density {
                        v.inc(i);
                    }
                }
            }

            if coeff != S::ONE {
                tmp *= coeff;
            }
            acc += tmp;
        }
    }

    acc
}
```

Impact

If a variable has multiple coefficients in a constraint that ultimately cancel out to zero, the calculation is performed correctly, but the density is still incorrectly marked as true for that variable, even though the variable effectively has a zero coefficient for that constraint. Further down the pipeline, this causes `multiexp` to error out with `"expected more bases from source"`, which means the circuit and verifying key are both valid but the circuit cannot be proved with Bellman.

This is therefore a completeness issue, and it is only triggered for circuits that contain specifically crafted constraints. Sapling does not use a malformed constraint of the required type, and any affected circuit would be identified at synthesis time. This finding therefore does not affect any currently deployed circuits.

Recommendations

Alter density tracking in the `eval` method to set density to zero if the accumulated coefficients for a variable cancel out to zero.

Remediation

TBD

3.6. Dependence on implementation-specific behavior: `clear_cofactor`

Target	sapling_crypto::group_hash / sapling_crypto::spec		
Category	Code Maturity	**Severity**	Medium
Likelihood	N/A	**Impact**	Informational

Description

In the trait `group::cofactor::CofactorGroup`, the documentation of the function `clear_cofactor` states that the implementation is allowed to differ from multiplication by the cofactor:

```
// group-0.13.0 - src/cofactor.rs
pub trait CofactorGroup:
// ...
{
    // ...
    /// Maps `self` to the prime-order subgroup by multiplying this element by
    some
    /// `k`-multiple of the cofactor.
    ///
    /// The value `k` does not vary between inputs for a given implementation,
    but may
    /// vary between different implementations of `CofactorGroup` because some
    groups have
    /// more efficient methods of clearing the cofactor when `k` is allowed to
    be
    /// different than `1`.
    ///
    /// If `Self` implements [`PrimeGroup`], this returns `self`.
    fn clear_cofactor(&self) -> Self::Subgroup;
```

The implementation for `jubjub::ExtendedPoint` does not document a stronger or differing claim:

```
// jubjub-0.10.0 - src/lib.rs
impl CofactorGroup for ExtendedPoint {
    type Subgroup = SubgroupPoint;

    fn clear_cofactor(&self) -> Self::Subgroup {
        SubgroupPoint(self.mul_by_cofactor())
    }
```

The audited code calls this function directly in two places and relies on the fact that `clear_cofactor` for `jubjub::ExtendedPoint` is implemented as multiplication by the cofactor:

```
// sapling-crypto - src/group_hash.rs
pub fn group_hash(tag: &[u8], personalization: &[u8]) ->
    Option<jubjub::SubgroupPoint> {
    // ...
    let p = jubjub::ExtendedPoint::from_bytes(h.as_array());
    if p.is_some().into() {
        // <ExtendedPoint as CofactorGroup>::clear_cofactor is implemented
    using
        // ExtendedPoint::mul_by_cofactor in the jubjub crate.
        let p = CofactorGroup::clear_cofactor(&p.unwrap());
        // ...
```

```
// sapling-crypto - src/spec.rs
pub(crate) fn ka_sapling_agree_prepared(
    // ...
) -> jubjub::SubgroupPoint {
    // [8 sk] b
    // <ExtendedPoint as CofactorGroup>::clear_cofactor is implemented using
    // ExtendedPoint::mul_by_cofactor in the jubjub crate.

    (b * sk).clear_cofactor()
}
```

The comments in the code show that the developers are aware of this dependence. The comments do not, however, provide evidence that the called function guarantees this behavior in future implementations.

Impact

Future implementations of `jubjub` could change the behavior of `clear_cofactor` and choose an implementation that does not return the point multiplied by the cofactor. This would cause `group_hash` and `ka_sapling_agree_prepared` to deviate from the specified behavior. A deviation in `group_hash` would result in deviations from the consensus rules. A deviation in `ka_sapling_agree_prepared` would result in a different key agreement for note encryption; the decrypting party would have to use the same implementation as the encrypting party to successfully decrypt the note.

Recommendations

Communicate this issue to the upstream `jubjub::ExtendedPoint` implementation, and document there that the function returns the multiplication by the cofactor. This clarifies that this is part of the function's contract for `jubjub::ExtendedPoint`.

If the upstream behavior cannot be contractually established, call `mul_by_cofactor` directly.

Remediation

TBD

4. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1. Additional checks

We recommend adding the following checks to enforce assumptions the code relies on:

Merkle path length in Spend circuit

In sapling-crypto, in the file src/circuit.rs, the circuit generated by the `Spend` struct via its `synthesize` method depends on the length of the Merkle path `auth_path`. Assert that this length equals $\text{MerkleDepth}^{\text{Sapling}} = 32$.

Elliptic curve `fixed_base_multiplication` could silently truncate argument `by`

In sapling-crypto, in the file src/circuit/ecc.rs, the function `fixed_base_multiplication` assumes that `by.len() <= 3 * base.len()`. Otherwise, it silently uses only the first `3 * base.len()` elements of `by`, which is not intended. Assert that the assumed upper bound on `by.len()` holds.

4.2. Incorrect or misleading code comments

We recommend clarifying or correcting the following doc comments.

Batch verifier behavior if `check_bundle` returns `false`

The doc comment on this function states:

```
// sapling-crypto - src/verifier/batch.rs
/// Returns `false` if the bundle doesn't satisfy all of the consensus rules.
    This
/// `BatchValidator` can continue to be used regardless, but some or all of
    the proofs
/// and signatures from this bundle may have already been added to the batch
    even if
/// it fails other consensus rules.
pub fn check_bundle<V: Copy + Into<i64>>(
```

The phrase "can continue to be used regardless" is ambiguous about the guarantees the verifier still makes. The current doc comment explains the internal behavior of the batch verifier but not the remaining external contract. Clarify how the verifier may be used after a `check_bundle` call returns `false`.

Elliptic curve addition of points with the same x-coordinate

The doc comment on this function states:

```
// sapling-crypto - src/circuit/ecc.rs
/// Performs an affine point addition, not defined for
/// points with the same x-coordinate.
pub fn add<CS>(&self, mut cs: CS, other: &Self) -> Result<Self,
    SynthesisError>
```

This comment does not make clear that the generated circuit is unsound if `self` is not guaranteed to be different from `other`, and incomplete if `self` and `other` have the same `x`- but different `y`-coordinate. Document these restrictions.

Confusion of note and value commitment

The doc comment on this function incorrectly refers to a *note* commitment as a *value* commitment. Both types exist in the codebase, but they are different.

```
// sapling-crypto - src/note/commitment.rs
impl ExtractedNoteCommitment {
    /// ...
    /// Serialize the value commitment to its canonical byte representation.
    pub fn to_bytes(self) -> [u8; 32] {
        self.0.to_repr()
    }
```

Constant `MAX_NOTE_VALUE` is not the upper bound of a note value

The following constant is defined with a misleading name:

```
// sapling-crypto - src/value.rs
/// Maximum note value.
pub const MAX_NOTE_VALUE: u64 = u64::MAX;
```

This is not the upper bound of a Sapling note's value, which the specification bounds by MAX_MONEY $= 2.1 \cdot 10^{15}$ zatoshi. If this constant refers to the upper bound of a note plaintext

value (which is a u64), consider renaming it to `MAX_NOTE_PLAINTEXT_VALUE`.

4.3. Incorrect error message

The function `ExpandedSpendingKey::read` reports every decoding failure of `ask` with the message `ask not in field`:

```
// sapling-crypto - src/keys.rs
impl ExpandedSpendingKey {
    // ...
    pub fn from_bytes(b: &[u8]) -> Result<Self, DecodingError> {
        // ...
        let ask =
    SpendAuthorizingKey::from_bytes(&b[0..32]).ok_or(DecodingError::InvalidAsk)?;
        // ..
    }

    pub fn read<R: Read>(mut reader: R) -> io::Result<Self> {
        let mut repr = [0u8; 96];
        reader.read_exact(repr.as_mut())?;
        Self::from_bytes(&repr).map_err(|e| match e {
            DecodingError::InvalidAsk => {
                io::Error::new(io::ErrorKind::InvalidData, "ask not in field")
            }
            // ..
        })
    }
```

However, a non-canonical field encoding is not the only reason parsing could fail. The implementation of `SpendAuthorizingKey::from_bytes` also rejects the value 0 because a `SpendAuthorizingKey` is restricted within the Sapling key tree to be a nonzero scalar.

```
// sapling-crypto - src/keys.rs
impl SpendAuthorizingKey {
    pub(crate) fn from_bytes(bytes: &[u8]) -> Option<Self> {
        <[u8; 32]>::try_from(bytes)
            .ok()
            .and_then(|b| {
                // RedJubjub.Private permits the full set of Jubjub scalars
    including
                // zero. However, a SpendAuthorizingKey is further restricted
    within the
                // Sapling key tree to be a non-zero scalar.
```

```
                jubjub::Scalar::from_repr(b)
                    .and_then(|s| {
                        CtOption::new(
                            redjubjub::SigningKey::try_from(b)
                                .expect("RedJubjub permits the set of valid
    SpendAuthorizingKeys"),
                            !s.is_zero(),
                        )
                    })
                    .into()
            })
            .map(SpendAuthorizingKey)
    }
```

Therefore, if the value 0 were given for `ask`, it would be incorrectly reported as `ask not in field`, even though 0 is in the field. Consider distinguishing the two cases in the error message or rewording it to state that `ask` is not a valid `SpendAuthorizingKey`.

4.4. Verifying keys have insufficient validation

In src/groth16/verifier.rs, `prepare_verifying_key` does not enforce any structural validation on the input verifying key. `VerifyingKey::read` rejects identity entries only for `ic`, not for `alpha_g1`, `beta_g2`, `gamma_g2`, or `delta_g2`. This is in contrast to the behavior documented by the comments on `VerifyingKey`.

```
pub struct VerifyingKey<E: Engine> {
    // alpha in g1 for verifying and for creating A/C elements of
    // proof. Never the point at infinity.
    pub alpha_g1: E::G1Affine,

    // beta in g1 and g2 for verifying and for creating B/C elements
    // of proof. Never the point at infinity.
    pub beta_g1: E::G1Affine,
    pub beta_g2: E::G2Affine,

    // gamma in g2 for verifying. Never the point at infinity.
    pub gamma_g2: E::G2Affine,

    // delta in g1/g2 for verifying and proving, essentially the magic
    // trapdoor that forces the prover to evaluate the C element of the
    // proof with only components from the CRS. Never the point at
    // infinity.
    pub delta_g1: E::G1Affine,
```

```
    pub delta_g2: E::G2Affine,
```

In the context of Sapling, this is not an exploitable issue since production Sapling circuits are intended to be verified against publicly available verifying keys generated from a trusted setup ceremony. Clients are expected to reject untrusted verifying keys regardless of whether they are malformed.

4.5. Type deviations from the specification

Many function implementations in the audited code claim to correspond to functions in the Zcash Protocol Specification. In several cases, this is only correct up to type conversions or restrictions applied to the arguments. In this section, we describe differences between the types used in the implementation and those used in the specification.

Bits and bytes

The specification distinguishes between bit $\mathbb{B}$ and byte $\mathbb{B}^{\mathbb{Y}}$ sequences. If a function specification uses bit sequences but the implementation uses a byte slice, the corresponding bit sequence is the sequence of little-endian bits of the byte sequence.

Merkle tree position

Sapling uses a Merkle tree of depth $\mathsf{MerkleDepth}^{\mathsf{Sapling}} = 32$ for its note commitments, and the positions are therefore unsigned 32-bit integers. In the specification, the position is supplied as argument to the function MixingPedersenHash. The type of this argument is $\{0 \mathinner{..} r_J - 1\}$. Note that $2^{32} < r_J$, so this is well-defined.

The corresponding function `mixing_pedersen_hash` in the implementation uses `u64` as the type for the same argument. If this argument value is smaller than r_J, it agrees with the specified function. The implementation also uses the type `u64` for the Merkle tree position. These functions only agree with their specified counterparts if the provided value is smaller than 2^{32}, because they are only specified for this range.

Specification is ambiguous about the signature of $\mathsf{PRF}^{\mathsf{nfSapling}}$

The Zcash Protocol Specification uses conflicting types for the first argument of $\mathsf{PRF}^{\mathsf{nfSapling}}$. In its definition ↗, the first argument is a non-zero jubjub subgroup-point. In its invocations (1 ↗, 2 ↗, 3 ↗), the representations of jubjub points of type $\mathbb{B}^{[l_J]}$ are provided.

4.6. Secret scalars use variable-time arithmetic

In the Sapling key-agreement helpers, the type `PreparedScalar` is instantiated via `group::WnafScalar`. The windowed non-adjacent form (WNAF) representation uses precomputation to improve the speed of scalar multiplication. Multiplying by scalars in this representation is therefore variable time, and timing measurements leak information about the scalar value.

`PreparedIncomingViewingKey::new` stores the wallet incoming viewing key as a `PreparedScalar`, and for key agreement, `PreparedEphemeralPublicKey::agree` multiplies `ivk` by the ephemeral public key.

```
impl PreparedEphemeralPublicKey {
    pub(crate) fn new(epk: EphemeralPublicKey) -> Self {
        PreparedEphemeralPublicKey(PreparedBase::new(epk.0))
    }

    pub(crate) fn agree(&self, ivk: &PreparedIncomingViewingKey) ->
    SharedSecret {
        SharedSecret(ka_sapling_agree_prepared(&ivk.0, &self.0))
    }
}
```

This could potentially be used to leak information about a long-lived `ivk`. While switching to constant-time scalar multiplication for key agreement would close this potential attack surface, such an attack is extremely difficult for a remote adversary to mount in practice. Extracting any meaningful information, if it is possible at all, requires a large number of queries, and each query requires a key agreement — which implies an equally large number of transactions between the two parties.

5. Design

This section provides a description of the audited code's design with regard to the audit's goals. The Zcash Protocol Specification ↗ describes the intended behavior in detail. In this section, we describe how the audited code relates to this specification.

5.1. Zcash bundle verification

The audited code contains two entry points for the verification process:

- The struct `SaplingVerificationContext` can be used to verify the Sapling bundle of an individual transaction.
- The struct `BatchValidator` can be used to verify the Sapling bundle of multiple transactions.

The `SaplingVerificationContext` is intended to be used in the following way:

1. Construct an instance via `new()` for each transaction.
2. Ensure that all value commitments `cv` for all spend and output descriptions are not small order.
3. For each Sapling spend description in the transaction, call `check_spend` on its fields.
4. For each Sapling output description in the transaction, call `check_output` on its fields.
5. Finally, call `final_check`.

The Sapling bundle of the transaction is valid if these calls all return `true`, and each `anchor` refers to an earlier block's final treestate.

The `BatchValidator` is intended to be used in the following way:

1. Construct an instance via `new()`.
2. Parse the Sapling bundle of each transaction into a `Bundle` struct.
3. Ensure that all value commitments `cv` for all spend and output descriptions are not small order.
4. For each bundle, call `check_bundle`.
5. Finally, call `validate`.

The Sapling bundles of all transactions are valid if these calls all return `true`, and each `anchor` refers to an earlier block's final treestate.

To check the validity of a bundle, the code must validate that

- the Spend and Output statements hold for all spend and output descriptions;

- the spend authorizations are valid, as defined in the spend authorization signature ↗ segment of the specification; and
- the balance claim matches the claimed Sapling balance, which is enforced by validating the binding signature as described in the Sapling balance ↗ segment of the specification.

Spend and Output statements

The consensus rules for validating the Spend and Output statements are given in the spend description ↗ and output description ↗ segments of the specification.

Neither `SaplingVerificationContext` nor `BatchValidator` checks that `cv` is not small order; this is the duty of the parser. Both do, however, carry out the small-order checks on `rk` and `epk`.

The zero-knowledge proofs are then validated against the Spend and Output circuits defined in src/circuit.rs. These circuits encode the Spend statement ↗ and Output statement ↗.

6. Assessment Results

During our assessment on the scoped Zcash Sapling targets, we discovered six findings, all of which were informational in nature.

6.1. Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Where Zellic states that a finding has been addressed or fixed in one or more specific commits, this indicates only that those commits introduce changes that, in our assessment, address the finding relative to the codebase version originally reviewed. This does not imply that Zellic reviewed other changes contained in those commits, the overall state of the codebase at those commits, or the interplay between remediation measures for different findings.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.