20 SEPTEMBER 2024

PHOENIX AUDIT REPORT

PERFORMED BY JULES DE SMIT

Table of contents

Disclaimer 3
Introduction 4
What is Phoenix? 6
Summary of findings 7
Report structure 8
Correctness evaluation 9
ElGamal 9
Defining a private and public key 9
Encryption 10
Decryption 11
AES 12
Secret key 13
Note secret key computation 13
Ownership check 14
View key 15
Ownership check 15
Public key 16
Stealth address generation 16
Stealth address 18
Note 19
Note minting 20
Note nullification 27
Sender data encryption 31
Phoenix circuits 33
ElGamal gadget 33
Sender encryption gadget 35
Full transaction circuit 39
Issues found 48

Static analysis tool results 48
Cargo audit 48
Test coverage 48
MINOR: External, open source dependencies should be patch-frozen 49
MINOR: Error-prone usage of Zeroize 50
EFFICIENCY: Redundant computation in input note section of circuit 51
STRUCTURAL: Copied code in owns method for secret key 53
STRUCTURAL: View key TODO comment for ct equality check 54
DOC: Confusing documentation comment on Note structure 55

DISCLAIMER

THE CONTENTS OF THIS REPORT ARE PROVIDED "AS IS", WITHOUT REPRESENTATIONS AND WARRANTIES OF ANY KIND.

THE AUTHOR DISCLAIMS ANY LIABILITY FOR DAMAGE ARISING OUT OF, OR IN CONNECTION WITH, THIS REPORT, AS STATED IN THE AGREEMENT BETWEEN THE AUTHOR AND THE CLIENT.

THIS REPORT IS ADDRESSED EXCLUSIVELY TO THE CLIENT. THE AUTHOR UNDERTAKES NO LIABILITY OR RESPONSIBILITY TOWARDS THE CLIENT OR RELEVANT THIRD PARTIES.

INTRODUCTION

This report, commissioned by Dusk on September 4th, 2024, will cover my findings and analysis of the implementation of the Phoenix protocol, as created by Dusk. According to the assignment, I reviewed and audited two repositories:

Repository	https://github.com/dusk-network/phoenix
Commit hash	8d70fac
Scope	All files

Repository	https://github.com/dusk-network/rusk
Commit hash	990dd87
Scope	All Phoenix-related logic in `execution-core/src/transfer/phoenix`

Both works together should provide a complete implementation of the Phoenix protocol; containing both the native code and in-circuit code implementations of this work.

As instructed by the Dusk team, my responsibilities for this audit were twofold:

- Ensuring correctness of implementation, with regards to the specification document provided by the Dusk team.

- Auditing the code for any possible security vulnerabilities.

This report will provide coverage of both points. The correctness evaluation will be performed by providing a self-explanatory understanding of the Dusk-provided Phoenix specification, which then is pitted against the relevant code in either repository listed. This evaluation can then be double-checked with the Dusk team to ensure that there was a correct understanding of the material, and that the material indeed corresponds with the implementation created by Dusk. This should provide full confidence that the team correctly implemented what was described, and, assuming correctness of the security analysis performed, that the protocol is fully sound.

The search for security vulnerabilities is performed by multiple extremely careful reads of the relevant codebases, as well as by the application of simple static analysis tools. The most prominent issue that is looked out for is the potential loss of funds, but other vulnerabilities such as DoS attacks are also considered.

The report was officially commissioned by Dusk on the 4th of September, 2024, and the preliminary findings were provided to Dusk on the 17th of September, 2024. The findings were assessed by the Dusk team and rectified or explained to provide a final report on the 20th of September, 2024.

WHAT IS PHOENIX?

Phoenix is an optionally-obfuscated UTXO transaction model, employed in the Dusk blockchain network. It takes inspiration from projects such as CryptoNote and Zcash, which work in similar fashion - by keeping track of individual units of arbitrary value (in this context, referred to as 'notes') much like a UTXO, and providing a mechanism of spending (or in this context, 'nullifying') such units when they are used in a transaction. With the proper primitives, this allows the network to fully obfuscate transaction values as well as senders and receivers.

Phoenix is a complex protocol which makes use of many cryptographic primitives, such as asymmetric as well as symmetric encryption, multiple hashing algorithms and Pedersen commitments, as well as a corresponding zero-knowledge circuit implementation written with Dusk's Plonk library. The goal is to create an auditable but sufficiently private transaction mechanism for the Dusk blockchain network, achieved by using a three-key mechanism, proving secret keys, public keys, and view keys, such that any third party would be able to 'view' transactions made by an individual, without gaining power to spend any of their holdings.

SUMMARY OF FINDINGS

The implementation of Phoenix was found to be well-specified, and the code well-documented. Only one `TODO` was found in the code, and one comment was confusing with regards to the implementation. Everything else was easily readable and easy to understand due to extensive documentation and support from the protocol specification document provided by the Dusk team. Overall, these findings were highlighted:

- One notational error in the specification.
- 'Supply-chain' level vulnerability with the version specification of open-source code in the imports used, which could potentially lead to draining of funds.
- Error-prone derivation of `Zeroize` for sensitive data, where `ZeroizeOnDrop` would make more sense.
- A slightly non-optimal implementation for input note logic in the Phoenix transaction circuit.
- An instance of repeated code which can be abstracted away with existing functionality.
- A leftover `TODO` concerning the use of constant time equality checks in the view key.
- A confusing comment on the `Note` structure.

REPORT STRUCTURE

The report will be structured as follows. Firstly, there will be a section dedicated to the evaluation of correct implementation by the Dusk team of the provided specification. This will cover all relevant sections of the specification and provide code snippets and explanations of equivalence. Following that, there will be a section dedicated to the found security vulnerabilities and code comments, along with the static analysis results.

The vulnerability findings will be accompanied with responses given by the Dusk team, and potential fixes provided as a result of these findings.

CORRECTNESS EVALUATION

ELGAMAL

Found at `phoenix/core/src/encryption/elgamal.rs`. ElGamal encryption is an asymmetric encryption scheme that works on elliptic curves (technically speaking, any cyclic group will do - Phoenix uses the JubJub elliptic curve group). The scheme is defined as follows (from wikipedia):

DEFINING A PRIVATE AND PUBLIC KEY

The private key is defined as a randomly sampled integer $x \in [1,q-1]$ where q is the order of the cyclic group (in Phoenix, this is supposed to be a random integer in the scalar field of the JubJub curve group). Its corresponding public key is then defined as $h := x \cdot g$, where g is the generator of the cyclic group.

Indeed, in the implementation, the secret key and public key are defined as a `JubJubScalar` and a `JubJubExtended` respectively. We can see this by looking at the encryption and decryption functions defined.

```
34    pub fn decrypt(
35        secret_key: &JubJubScalar,
```

```
19    pub fn encrypt(
20        public_key: &JubJubExtended,
```

ENCRYPTION

According to wikipedia, the encryption algorithm is defined as follows (switching notation to be additive):

- A message M should be mapped to a cyclic group element m.
- Sample a random integer $y \in [1, q-1]$.
- Compute $s := y \cdot h$.
- Compute $c_1 := y \cdot g$.
- Compute $c_2 := m + s$.
- The resulting ciphertext is then (c_1, c_2).

Phoenix implements the algorithm as follows:

```
14    /// Encrypts a JubJubExtended plaintext given a public key and a fresh random
15    /// number 'r'.
16    ///
17    /// ## Return
18    /// Returns a ciphertext (JubJubExtended, JubJubExtended).
19    pub fn encrypt(
20        public_key: &JubJubExtended,
21        plaintext: &JubJubExtended,
22        r: &JubJubScalar,
23    ) → (JubJubExtended, JubJubExtended) {
24        let ciphertext_1 = GENERATOR * r;
25        let ciphertext_2 = plaintext + public_key * r;
26
27        (ciphertext_1, ciphertext_2)
28    }
```

The random integer y is defined here as `r`. c_1 is indeed computed correctly, as we have `GENERATOR * r` which directly corresponds to $y \cdot g$ from the wikipedia spec. c_2 is also indeed computed correctly, as we first compute s by doing `public_key * r`, and then adding it to the plaintext mapped element m defined as `plaintext`. Therefore, encryption is deemed to be implemented correct according to the spec.

DECRYPTION

According to wikipedia, the decryption algorithm is defined as follows:

- Compute $s := c_1 \cdot x$.
- Compute s^{-1}. Since we are in an elliptic curve group, this is defined as a point negation; so we say that we compute $-s$.
- Compute $m := c_2 + s^{-1}$. This translates to $c_2 - s$ in the elliptic curve context.
- Use the reversible mapping function to recover M from m.

Phoenix implements the algorithm as follows:

```
30    /// Decrypts a ciphertext given a secret key.
31    ///
32    /// ## Return
33    /// Returns a JubJubExtended plaintext.
34    pub fn decrypt(
35        secret_key: &JubJubScalar,
36        ciphertext: &(JubJubExtended, JubJubExtended),
37    ) → JubJubExtended {
38        let ciphertext_1 = ciphertext.0;
39        let ciphertext_2 = ciphertext.1;
40
41        // return the plaintext
42        ciphertext_2 - ciphertext_1 * secret_key
43    }
```

s is computed here by `ciphertext_1 * secret_key`, which corresponds to $c_1 \cdot x$. Finally, m is retrieved by the subtraction: `ciphertext_2 - ciphertext_1 * secret_key`, correctly corresponding to $m := c_2 - s$. Since Phoenix does not use any mapping function (it encrypts curve points already), this step does not need to be evaluated. Therefore, the out-of-circuit implementation of ElGamal is deemed correct.

AES

Found at `phoenix/core/src/encryption/aes.rs`. We point out that AES is implemented only as a wrapper over another crate, `aes-gcm`, and therefore make no points about the correctness of the actual cipher. The only thing to point out with this implementation is that extra care should be taken with the defining of `ENCRYPTION_SIZE`, as an incorrect definition may result in a panic. Other than that, all seems sound.

SECRET KEY

Found at `phoenix/core/src/keys/secret.rs`. According to the spec, a Phoenix secret key should be determined by two scalar field elements on the JubJub curve.

- Secret key: $sk = (a, b)$ where $a, b \leftarrow \mathbb{F}_t$.

In the code, this is correctly reflected:

```
51    pub struct SecretKey {
52        a: JubJubScalar,
53        b: JubJubScalar,
54    }
```

NOTE SECRET KEY COMPUTATION

For the creation of a note secret key, the spec mentions that this is computed via $H^{BLAKE2b}(k_{DH}) + b$. In the implementation, the value being hashed is actually $\tilde{k}_{DH}$ as defined in the spec. The spec does mention that only the owner of the secret key can compute this value (knowledge of (a, b) is required), and thus concludes that the implementation is likely correct, but the notation in the spec might be wrong.

For reference, k_{DH} is defined as $r \cdot A$ in the spec, whereas $\tilde{k}_{DH}$ is defined as $a \cdot R$. The code reflects that $\tilde{k}_{DH}$ is used:

```
82        /// Generates a [`NoteSecretKey`] using the `R` of the given
83        /// [`StealthAddress`]. With the formula: `note_sk = H(a · R) + b`
84        pub fn gen_note_sk(&self, stealth: &StealthAddress) → NoteSecretKey {
85            let aR = stealth.R() * self.a;
86
87            NoteSecretKey::from(hash(&aR) + self.b)
88        }
```

The Dusk team clarified that this was merely a notational error, and the math should indeed check out either way. The notational fix was done in pull request #248. Thus, this implementation is correct.

OWNERSHIP CHECK

The ownership check can be performed by checking that the note public key is equal to $H^{BLAKE2b}(\tilde{k}_{DH}) \cdot G + B$, according to the spec. This is done in the code as follows:

```
90      /// Checks if `note_pk ?= (H(R · a) + b) · G`
91      pub fn owns(&self, stealth_address: &StealthAddress) → bool {
92          let aR = stealth_address.R() * self.a();
93          let hash_aR = hash(&aR);
94          let note_sk = hash_aR + self.b();
95
96          let note_pk = GENERATOR_EXTENDED * note_sk;
97
98          stealth_address.note_pk().as_ref() == &note_pk
99      }
```

The code computes $H^{BLAKE2b}(\tilde{k}_{DH}) + b$ (which corresponds to nsk) and then multiplies it by the generator. This gives us the aforementioned note public key, as stated in section 4.2.2 of the specification. Therefore, this code is correct.

VIEW KEY

Found at `phoenix/core/src/keys/view.rs`. A view key is defined in the Phoenix spec as $vk = (a, B)$ where $B = b \cdot G$. In the code, this is duly reflected:

```
28  pub struct ViewKey {
29      a: JubJubScalar,
30      B: JubJubExtended,
31  }
```

OWNERSHIP CHECK

Like the secret key, the view key module indeed does check ownership by checking equality with $H^{BLAKE2b}(\tilde{k}_{DH}) \cdot G + B$. Let's check the code:

```
63      /// Checks `note_pk = H(R · a) · G + B`
64      pub fn owns(&self, stealth_address: &StealthAddress) → bool {
65          let aR = stealth_address.R() * self.a();
66          let hash_aR = hash(&aR);
67          let hash_aR_G = GENERATOR_EXTENDED * hash_aR;
68          let note_pk = hash_aR_G + self.B();
69
70          stealth_address.note_pk().as_ref() == &note_pk
71      }
```

Indeed, the code computes the equality check as defined in the 'note public key' portion of section 4.2.2 in the specification. This is thus the correct implementation, despite it varying slightly in logic from the secret key ownership check since there is no knowledge of b.

PUBLIC KEY

Found at `phoenix/core/src/keys/public.rs`. In the Phoenix spec, a public key is defined as $pk = (A, B)$ where $A = a \cdot G$ and $B = b \cdot G$. This corresponds to the code:

```
25    pub struct PublicKey {
26        A: JubJubExtended,
27        B: JubJubExtended,
28    }
```

STEALTH ADDRESS GENERATION

Phoenix allows a caller to generate a one-time stealth address given a public key. In the spec, this is defined as follows:

- Sample a random $r \leftarrow \mathbb{F}_t$.
- Compute $k_{DH} := r \cdot A$.
- Compute the note public key $npk := H^{BLAKE2b}(k_{DH}) \cdot G + B$.
- Compute $R := r \cdot G$.
- Return (npk, R).

Let's compare this to the code:

```
47      /// Generates new `note_pk = H(A · r) · G + B` from a given `r`
48      pub fn gen_stealth_address(&self, r: &JubJubScalar) → StealthAddress {
49          let G = GENERATOR_EXTENDED;
50          let R = G * r;
51
52          let rA = self.A * r;
53          let rA = hash(&rA);
54          let rA = G * rA;
55
56          let note_pk = rA + self.B;
57          let note_pk = note_pk.into();
58
59          StealthAddress { R, note_pk }
60      }
```

Firstly, R is computed by multiplying the random `r` by the `GENERATOR_EXTENDED`. Next, $H^{BLAKE2b}(k_{DH})$ is computed by multiplying `r` by `A` and hashing the result. The result is then multiplied by `GENERATOR_EXTENDED` to compute $H^{BLAKE2b}(k_{DH}) \cdot G$. Finally, this value is added with `B` to create npk. Both `note_pk` and `R` are then returned in the form of a `StealthAddress` struct. This computation is correct according to the specification.

STEALTH ADDRESS

The file `phoenix/core/src/stealth_address.rs` is deemed to be correct as it simply defines a struct holding R and npk. The rest of the file contains convenience functionality and has no bearing on the spec itself. The `StealthAddress` struct is defined as follows:

```
27  pub struct StealthAddress {
28      pub(crate) R: JubJubExtended,
29      pub(crate) note_pk: NotePublicKey,
30  }
```

Which matches the specification for the note public key in section 4.2.2.

NOTE

Found at `phoenix/core/src/note.rs`. Let's start by checking that a `Note` contains all the needed information as outlined in the specification.

```
75  pub struct Note {
76      pub(crate) note_type: NoteType,
77      pub(crate) value_commitment: JubJubAffine,
78      pub(crate) stealth_address: StealthAddress,
79      pub(crate) pos: u64,
80      pub(crate) value_enc: [u8; VALUE_ENC_SIZE],
81      pub(crate) sender: Sender,
82  }
```

As defined in the specification, a note N is supposed to be made up of $\{type, com, enc, npk, R, enc^{sender}\}$. All declared fields are existant on this struct; though it isn't directly visible. Firstly, the note type and the value commitment are present on the struct. The field `stealth_address` contains both npk and R. Note also that `enc` is contained in the `value_enc` field, and enc^{sender} will be contained in `sender` (although the field can either be encrypted or unencrypted, and care should be taken to ensure that the value is correctly encrypted or unencrypted depending on context). Therefore, My conclusion is that the `Note` struct is correct according to the specification.

Additionally, the `Note` struct contains a `pos` field denoting the position of the `Note` in the note merkle tree as defined in the spec. The spec mentions that this field is not covered as the sender can not determine the position of the note, and it is seemingly only set once the validators include this note in the note merkle tree during consensus.

NOTE MINTING

Note minting can be performed by using the `Note::new` method. This is defined as follows:

```rust
    pub fn new<R: RngCore + CryptoRng>(
        rng: &mut R,
        note_type: NoteType,
        sender_pk: &PublicKey,
        receiver_pk: &PublicKey,
        value: u64,
        value_blinder: JubJubScalar,
        sender_blinder: [JubJubScalar; 2],
    ) -> Self {
        let r = JubJubScalar::random(&mut *rng);
        let stealth_address = receiver_pk.gen_stealth_address(&r);

        let value_commitment = value_commitment(value, value_blinder);

        // Output notes have undefined position, equals to u64's MAX value
        let pos = u64::MAX;

        let value_enc = match note_type {
            NoteType::Transparent => {
                let mut value_enc = [0u8; VALUE_ENC_SIZE];
                value_enc[..u64::SIZE].copy_from_slice(&value.to_bytes());

                value_enc
            }
            NoteType::Obfuscated => {
                let shared_secret = dhke(&r, receiver_pk.A());
                let value_blinder = BlsScalar::from(value_blinder);

                let mut plaintext = value.to_bytes().to_vec();
                plaintext.append(&mut value_blinder.to_bytes().to_vec());

                aes::encrypt(&shared_secret, &plaintext, rng)
                    .expect("Encrypted correctly.")
            }
        };

        Note {
            note_type,
            value_commitment,
            stealth_address,
            pos,
            value_enc,
            sender: Sender::encrypt(
                stealth_address.note_pk(),
                sender_pk,
                &sender_blinder,
            ),
        }
    }
```

Let's revisit the specification on minting notes:

- Set the type of transaction. If transparent, set 0. If obfuscated, set 1.

```
41    pub enum NoteType {
42        /// Defines a Transparent type of Note
43        Transparent = 0,
44        /// Defines an Obfuscated type of Note
45        Obfuscated = 1,
46    }
```

The code does this correctly according to the spec.

- Set v to the amount of money being transferred.

- Set a blinding factor for the commitment, and a nonce for the encryption. If the note type is 0 (transparent), the blinder value should be 0. Else, the blinder value should be randomly sampled from the JubJub scalar field.

- Compute the value commitment using the Pedersen commitment scheme.

Both transparent note creation methods use transparent commitments with 0 valued blinders, as evidenced by the below code:

```rust
    /// Creates a new transparent note
    ///
    /// This is equivalent to [`transparent`] but taking only a stealth address
    /// and a value. This is done to be able to generate a note
    /// directly for a stealth address, as opposed to a public key.
    pub fn transparent_stealth(
        stealth_address: StealthAddress,
        value: u64,
        sender: impl Into<Sender>,
    ) -> Self {
        let value_commitment = transparent_value_commitment(value);

        let pos = u64::MAX;

        let mut value_enc = [0u8; VALUE_ENC_SIZE];
        value_enc[..u64::SIZE].copy_from_slice(&value.to_bytes());

        Note {
            note_type: NoteType::Transparent,
            value_commitment,
            stealth_address,
            pos,
            value_enc,
            sender: sender.into(),
        }
    }
```

```rust
    /// Creates a new transparent note
    ///
    /// The blinding factor will be constant zero since the value commitment
    /// exists only to shield the value. The value is not hidden for transparent
    /// notes, so this can be trivially treated as a constant.
    pub fn transparent<R: RngCore + CryptoRng>(
        rng: &mut R,
        sender_pk: &PublicKey,
        receiver_pk: &PublicKey,
        value: u64,
        sender_blinder: [JubJubScalar; 2],
    ) -> Self {
        Self::new(
            rng,
            NoteType::Transparent,
            sender_pk,
            receiver_pk,
            value,
            TRANSPARENT_BLINDER,
            sender_blinder,
        )
    }
```

Method `transparent_stealth` which is defined first makes use of the `transparent_value_commitment` function defined in `lib.rs`, found in the root of the out-of-circuit part of the Phoenix repository:

```
42    /// Use the pedersen commitment scheme to compute a transparent value
43    /// commitment.
44    pub fn transparent_value_commitment(value: u64) → JubJubAffine {
45        JubJubAffine::from(GENERATOR_EXTENDED * JubJubScalar::from(value))
46    }
```

And the method `transparent` feeds in a `TRANSPARENT_BLINDER`, which is then used in the `value_commitment` method called in `new`:

```
48    /// Use the pedersen commitment scheme to compute a value commitment using a
49    /// blinding-factor.
50    pub fn value_commitment(
51        value: u64,
52        blinding_factor: JubJubScalar,
53    ) → JubJubAffine {
54        JubJubAffine::from(
55            (GENERATOR_EXTENDED * JubJubScalar::from(value))
56                + (GENERATOR_NUMS_EXTENDED * blinding_factor),
57        )
58    }
```

With `blinding_factor` set to `TRANSPARENT_BLINDER` which equals zero, this method is the same as `transparent_value_commitment`. So in all cases, the note computes the transparent commitment correctly.

In the case of an obfuscated note, it seems that the method relies on the caller to provide a sufficiently random blinder:

```
194     /// Creates a new obfuscated note
195     ///
196     /// The provided blinding factor will be used to calculate the value
197     /// commitment of the note. The tuple (value, value_blinder), known by
198     /// the caller of this function, must be later used to prove the
199     /// knowledge of the value commitment of this note.
200     pub fn obfuscated<R: RngCore + CryptoRng>(
201         rng: &mut R,
202         sender_pk: &PublicKey,
203         receiver_pk: &PublicKey,
204         value: u64,
205         value_blinder: JubJubScalar,
206         sender_blinder: [JubJubScalar; 2],
207     ) -> Self {
208         Self::new(
209             rng,
210             NoteType::Obfuscated,
211             sender_pk,
212             receiver_pk,
213             value,
214             value_blinder,
215             sender_blinder,
216         )
217     }
```

The `new` method then calls the correctly implemented Pedersen commitment function with the given value and blinder.

- Encrypt the opening of the commitment. In case of a transparent note, we simply set the encryption to be the cleartext value. Otherwise, we use AES encryption to hide it.

Let's re-paste the encryption section of the `new` method:

```
111         let value_enc = match note_type {
112             NoteType::Transparent => {
113                 let mut value_enc = [0u8; VALUE_ENC_SIZE];
114                 value_enc[..u64::SIZE].copy_from_slice(&value.to_bytes());
115 
116                 value_enc
117             }
118             NoteType::Obfuscated => {
119                 let shared_secret = dhke(&r, receiver_pk.A());
120                 let value_blinder = BlsScalar::from(value_blinder);
121 
122                 let mut plaintext = value.to_bytes().to_vec();
123                 plaintext.append(&mut value_blinder.to_bytes().to_vec());
124 
125                 aes::encrypt(&shared_secret, &plaintext, rng)
126                     .expect("Encrypted correctly.")
127             }
128         };
```

Here, you can see that in the case of a transparent note, the value is simply copied into a slice as bytes, leaving the unencrypted value on the note. In the obfuscated case, the value is correctly encrypted with AES. See also that the key for the AES encryption can be safely recovered by the receiver with the knowledge of R from the stealth address and a from the private key.

However, it is important to note that the encryption function if left ambiguous in the specification - likely because it could either be AES or ElGamal depending on the context. From the specification, it seems that AES is used for the out-of-circuit case so it should be correct here.

The decryption of the value should be correctly implemented for the obfuscated case:

```
233     fn decrypt_value(
234         &self,
235         vk: &ViewKey,
236     ) -> Result<(u64, JubJubScalar), Error> {
237         let R = self.stealth_address.R();
238         let shared_secret = dhke(vk.a(), R);
239 
240         let dec_plaintext: [u8; PLAINTEXT_SIZE] =
241             aes::decrypt(&shared_secret, &self.value_enc)?;
242 
243         let value = u64::from_slice(&dec_plaintext[..u64::SIZE])?;
244 
245         // Converts the BLS Scalar into a JubJub Scalar.
246         // If the `vk` is wrong it might fails since the resulting BLS Scalar
247         // might not fit into a JubJub Scalar.
248         let value_blinder =
249             match JubJubScalar::from_slice(&dec_plaintext[u64::SIZE..])?.into()
250             {
251                 Some(scalar) => scalar,
252                 None => return Err(Error::InvalidData),
253             };
254 
255         Ok((value, value_blinder))
256     }
```

As specified, the receiver can retrieve the AES key with his view key - recovered from R and a. The value and blinder are then separated and returned after being decrypted.

- Compute the note public key.

This is correctly computed as the stealth address module has been confirmed to work according to spec, and the second line of the `new` function computes this stealth address.

The note is then returned accordingly. This functionality is therefore deemed fully correct.

NOTE NULLIFICATION

Note nullification logic is implemented between the two repositories. The specification roughly explains it as follows:

- Compute the nullifier as a poseidon hash of the nullification key and the position of the note.
- If the note is obfuscated, then we decrypt the value commitment and get the value and blinder. Otherwise, we just retrieve the value from the plaintext array.
- Retrieve the old note private key.
- Hash the transaction payload with Blake2b.
- Generate a Schnorr signature of the hash of the tx payload.

The Phoenix repository only provides functionality for the first step, in this function:

```
258     /// Create a unique nullifier for the note
259     ///
260     /// This nullifier is represeted as `H(note_sk · G', pos)`
261     pub fn gen_nullifier(&self, sk: &SecretKey) → BlsScalar {
262         let note_sk = sk.gen_note_sk(&self.stealth_address);
263         let pk_prime = GENERATOR_NUMS_EXTENDED * note_sk.as_ref();
264         let pk_prime = pk_prime.to_hash_inputs();
265 
266         let pos = BlsScalar::from(self.pos);
267 
268         Hash::digest(Domain::Other, &[pk_prime[0], pk_prime[1], pos])[0]
269     }
```

This first step is performed correctly; the nullification key is defined in section 4.2.2 as the note secret key multiplied by the NUMS generator. This nullification key is then correctly concatenated and hashed with the Poseidon hash function.

For the rest of the steps, we need to look at the transfer contract code located in `rusk/execution-core/src/transfer/phoenix.rs`, method `new`.

```
133         for (note, _opening) in &inputs {
134             let note_nullifier = note.gen_nullifier(sender_sk);
135             for nullifier in &input_nullifiers {
136                 if note_nullifier == *nullifier {
137                     return Err(Error::Replay);
138                 }
139             }
140             input_nullifiers.push(note_nullifier);
141             input_values.push(note.value(Some(&sender_vk))?);
142             input_value_blinders.push(note.value_blinder(Some(&sender_vk))?);
143         }
```

Indeed, for each input note, it computes a nullifier, and then retrieves the encrypted value and blinder for each input note as well.

The value and blinder are retrieved with the following logic in the Phoenix repository:

```
330     /// Attempt to decrypt the note value provided a [`ViewKey`]. Always
331     /// succeeds for transparent notes, might fails or return random values for
332     /// obfuscated notes if the provided view key is wrong.
333     pub fn value(&self, vk: Option<&ViewKey>) -> Result<u64, Error> {
334         match (self.note_type, vk) {
335             (NoteType::Transparent, _) => {
336                 let value =
337                     u64::from_slice(&self.value_enc[..u64::SIZE]).unwrap();
338                 Ok(value)
339             }
340             (NoteType::Obfuscated, Some(vk)) => {
341                 self.decrypt_value(vk).map(|(value, _)| value)
342             }
343             _ => Err(Error::MissingViewKey),
344         }
345     }
346
347     /// Decrypt the blinding factor with the provided [`ViewKey`]
348     ///
349     /// If the decrypt fails, a random value is returned
350     pub fn value_blinder(
351         &self,
352         vk: Option<&ViewKey>,
353     ) -> Result<JubJubScalar, Error> {
354         match (self.note_type, vk) {
355             (NoteType::Transparent, _) => Ok(TRANSPARENT_BLINDER),
356             (NoteType::Obfuscated, Some(vk)) => self
357                 .decrypt_value(vk)
358                 .map(|(_, value_blinder)| value_blinder),
359             _ => Err(Error::MissingViewKey),
360         }
361     }
362 }
```

With the `decrypt_value` function being correct, these values should be correctly retrieved.

Moving on, we still need to ensure that the transaction payload hash is correctly signed with each input note secret key.

```
202         // Now we can set the tx-skeleton, payload and get the payload-hash
203         let tx_skeleton = TxSkeleton {
204             root,
205             // we also need the nullifiers for the tx-circuit, hence the clone
206             nullifiers: input_nullifiers.clone(),
207             outputs,
208             max_fee,
209             deposit,
210         };
211         let payload = Payload {
212             chain_id,
213             tx_skeleton,
214             fee,
215             data,
216         };
217         let payload_hash = payload.hash();
218
219         // --- Create the transaction proof
220
221         // Create a vector with all the information for the input-notes
222         let mut input_notes_info = Vec::with_capacity(input_len);
223         inputs
224             .into_iter()
225             .zip(input_nullifiers)
226             .zip(input_values)
227             .zip(input_value_blinders)
228             .for_each(
229                 |(
230                     (((note, merkle_opening), nullifier), value),
231                     value_blinder,
232                 )| {
233                     let note_sk = sender_sk.gen_note_sk(note.stealth_address());
234                     let note_pk_p = JubJubAffine::from(
235                         crate::GENERATOR_NUMS_EXTENDED * note_sk.as_ref(),
236                     );
237                     let signature = note_sk.sign_double(rng, payload_hash);
238                     input_notes_info.push(InputNoteInfo {
239                         merkle_opening,
240                         note,
241                         note_pk_p,
242                         value,
243                         value_blinder,
244                         nullifier,
245                         signature,
246                     });
247                 },
248             );
```

As according to the specification, we generate a double-key Schnorr signature of the transaction payload hash for each input note. Therefore, the nullification logic is correct according to the specification.

SENDER DATA ENCRYPTION

The last piece of functionality for the note is sender data encryption. Let's break it down:

- We use the sender's static key a and b to generate sig_A and sig_B respectively of the transaction payload hash from the previous section.

- Using the note public key, we encrypt the sender's public key A and B individually using the ElGamal encryption scheme.

The Phoenix repository only implements the second step:

```
454     /// Create a new [`Sender`] enum by encrypting the sender's [`PublicKey`] in
455     /// a way that only the receiver of the note can decrypt.
456     pub fn encrypt(
457         note_pk: &NotePublicKey,
458         sender_pk: &PublicKey,
459         blinder: &[JubJubScalar; 2],
460     ) → Self {
461         let sender_enc_A =
462             elgamal::encrypt(note_pk.as_ref(), sender_pk.A(), &blinder[0]);
463
464         let sender_enc_B =
465             elgamal::encrypt(note_pk.as_ref(), sender_pk.B(), &blinder[1]);
466
467         let sender_enc_A: (JubJubAffine, JubJubAffine) =
468             (sender_enc_A.0.into(), sender_enc_A.1.into());
469         let sender_enc_B: (JubJubAffine, JubJubAffine) =
470             (sender_enc_B.0.into(), sender_enc_B.1.into());
471
472         Self::Encryption([sender_enc_A, sender_enc_B])
473     }
```

Additionally, it does also provide decryption logic:

```rust
    /// Decrypts the [`PublicKey`] of the sender of the [`Note`], using the
    /// [`NoteSecretKey`] generated by the receiver's [`SecretKey`] and the
    /// [`StealthAddress`] of the [`Note`].
    ///
    /// Note: Decryption with an *incorrect* [`NoteSecretKey`] will still yield
    /// a [`PublicKey`], but in this case, the public-key will be a random one
    /// that has nothing to do with the sender's [`PublicKey`].
    ///
    /// Returns an error if the sender is of type [`Sender::ContractInfo`].
    pub fn decrypt(&self, note_sk: &NoteSecretKey) -> Result<PublicKey, Error> {
        let sender_enc = match self {
            Sender::Encryption(enc) => enc,
            Sender::ContractInfo(_) => {
                return Err(Error::InvalidEncryption);
            }
        };

        let sender_enc_A = sender_enc[0];
        let sender_enc_B = sender_enc[1];

        let decrypt_A = elgamal::decrypt(
            note_sk.as_ref(),
            &(sender_enc_A.0.into(), sender_enc_A.1.into()),
        );
        let decrypt_B = elgamal::decrypt(
            note_sk.as_ref(),
            &(sender_enc_B.0.into(), sender_enc_B.1.into()),
        );

        Ok(PublicKey::new(decrypt_A, decrypt_B))
    }
```

Both of which match the specification. The first step is performed in the transfer contract portion of the code, located at `rusk/execution-core/src/transfer/phoenix.rs`, in method `new`:

```rust
        // Sign the payload hash using both 'a' and 'b' of the sender_sk
        let schnorr_sk_a = SchnorrSecretKey::from(sender_sk.a());
        let sig_a = schnorr_sk_a.sign(rng, payload_hash);
        let schnorr_sk_b = SchnorrSecretKey::from(sender_sk.b());
        let sig_b = schnorr_sk_b.sign(rng, payload_hash);
```

Therefore, the sender encryption portion of the specification seems correctly implemented.

PHOENIX CIRCUITS

The circuit logic is wholly contained in `phoenix/circuits/src` and I'll cover all relevant spec implementation.

The circuit work is structured across three files mainly, one of them implementing the sender encryption sequence as a gadget (`phoenix/circuits/src/sender_enc.rs`), one of them implementing the in-circuit version of ElGamal (`phoenix/circuits/src/encryption/elgamal.rs`) and the other implementing the full transaction circuit (`phoenix/circuits/src/circuit_impl.rs`).

ELGAMAL GADGET

Let's check the ElGamal gadget against the out-of-circuit implementation as it was previously deemed correct - so correspondence to the out-of-circuit implementation should mean correctness of the circuit.

The encryption is defined as follows:

```
15    /// Encrypt in-circuit a plaintext WitnessPoint.
16    ///
17    /// ## Return
18    /// Returns a ciphertext (WitnessPoint, WitnessPoint).
19    pub fn encrypt_gadget(
20        composer: &mut Composer,
21        public_key: WitnessPoint,
22        plaintext: WitnessPoint,
23        r: Witness,
24    ) → Result<(WitnessPoint, WitnessPoint), Error> {
25        let R = composer.component_mul_point(r, public_key);
26        let ciphertext_1 = composer.component_mul_generator(r, GENERATOR)?;
27        let ciphertext_2 = composer.component_add_point(plaintext, R);
28
29        Ok((ciphertext_1, ciphertext_2))
30    }
```

Let's pull up the out-of-circuit implementation again for reference:

```
14  /// Encrypts a JubJubExtended plaintext given a public key and a fresh random
15  /// number 'r'.
16  ///
17  /// ## Return
18  /// Returns a ciphertext (JubJubExtended, JubJubExtended).
19  pub fn encrypt(
20      public_key: &JubJubExtended,
21      plaintext: &JubJubExtended,
22      r: &JubJubScalar,
23  ) → (JubJubExtended, JubJubExtended) {
24      let ciphertext_1 = GENERATOR * r;
25      let ciphertext_2 = plaintext + public_key * r;
26
27      (ciphertext_1, ciphertext_2)
28  }
```

From this, it's easy to conclude that `ciphertext_1` is computed correctly; it is computed by multiplying `r` with the `GENERATOR`, just like in the out-of-circuit code. Additionally, the computation of `ciphertext_2` also matches; firstly, we compute `R` which is equal to `public_key * r`. Then, the `plaintext` and `R` are added together, matching the out-of-circuit code. The encryption gadget is therefore correct.

Next, let's look at decryption:

```
32  /// Decrypt in-circuit a ciphertext (WitnessPoint, WitnessPoint).
33  ///
34  /// ## Return
35  /// Returns a plaintext WitnessPoint.
36  pub fn decrypt_gadget(
37      composer: &mut Composer,
38      secret_key: Witness,
39      ciphertext_1: WitnessPoint,
40      ciphertext_2: WitnessPoint,
41  ) → WitnessPoint {
42      let c1_sk = composer.component_mul_point(secret_key, cipherte
43      let neg_one = composer.append_constant(-JubJubScalar::one());
44      let neg_c1_sk = composer.component_mul_point(neg_one, c1_sk);
45
46      // return plaintext
47      composer.component_add_point(ciphertext_2, neg_c1_sk)
48  }
```

And again, let's pull up the out-of-circuit code for reference:

```rust
/// Decrypts a ciphertext given a secret key.
///
/// ## Return
/// Returns a JubJubExtended plaintext.
pub fn decrypt(
    secret_key: &JubJubScalar,
    ciphertext: &(JubJubExtended, JubJubExtended),
) -> JubJubExtended {
    let ciphertext_1 = ciphertext.0;
    let ciphertext_2 = ciphertext.1;

    // return the plaintext
    ciphertext_2 - ciphertext_1 * secret_key
}
```

The gadget first computes `ciphertext_1 * secret_key` by computing `c1_sk`. `c1_sk` is then negated, and finally we add `ciphertext_2` to the negated `neg_c1_sk`, which matches the out-of-circuit implementation. Both ElGamal gadgets are therefore considered correct.

SENDER ENCRYPTION GADGET

The sender encryption gadget specifically should constrain steps 6 and 7 of the transaction validity circuit in the specification. These steps are defined as follows:

- Prove that the sender owns the transaction. We do this by proving knowledge of the static key (a, b) corresponding to the public key (A, B). Instead of putting the static key in the witness, we simply prove that the transactor has the capability of signing a message that corresponds to the public key. The transaction payload hash is signed and the signature is verified in-circuit.

- Prove that the encrypted data is encrypted correctly so that only the recipient can decrypt it.

As covered in the sender data encryption segment, the sender will sign the transaction payload hash with both a and b of the static key. So the method indeed takes in two Schnorr signatures. The circuit then correctly includes both public keys A and B into the witness, as well as both signatures. Subsequently, both signatures are verified. Let's single out this part of the circuit for reference:

```
29      // VERIFY A SIGNATURE FOR EACH KEY 'A' AND 'B'
30      let sender_pk_A = composer.append_point(sender_pk.A());
31      let sender_pk_B = composer.append_point(sender_pk.B());
32
33      let sig_A_u = composer.append_witness(*signatures.0.u());
34      let sig_A_R = composer.append_point(signatures.0.R());
35
36      let sig_B_u = composer.append_witness(*signatures.1.u());
37      let sig_B_R = composer.append_point(signatures.1.R());
38
39      gadgets::verify_signature(
40          composer,
41          sig_A_u,
42          sig_A_R,
43          sender_pk_A,
44          payload_hash,
45      )?;
46      gadgets::verify_signature(
47          composer,
48          sig_B_u,
49          sig_B_R,
50          sender_pk_B,
51          payload_hash,
52      )?;
```

This covers the first step of the sender encryption correctness in-circuit.

Next, let's cover the correctness of the encryption. Let's again single out the part of the circuit for reference:

```
54      // ENCRYPT EACH KEY 'A' and 'B' USING EACH OUTPUT 'NPK'
55      let note_pk_0 = composer.append_public_point(output_npk[0]);
56      let note_pk_1 = composer.append_public_point(output_npk[1]);
57
58      let blinder_A_0 = composer.append_witness(sender_blinder[0][0]);
59      let blinder_B_0 = composer.append_witness(sender_blinder[0][1]);
60
61      let blinder_A_1 = composer.append_witness(sender_blinder[1][0]);
62      let blinder_B_1 = composer.append_witness(sender_blinder[1][1]);
63
64      // assert that the sender encryption of the first note is correct
65      // appends the values of sender_enc_out0 as public input
66      assert_sender_enc(
67          composer,
68          sender_pk_A,
69          sender_pk_B,
70          note_pk_0,
71          (blinder_A_0, blinder_B_0),
72          sender_enc_out0,
73      )?;
74
75      // assert that the sender encryption of the second note is correct
76      // appends the values of sender_enc_out1 as public input
77      assert_sender_enc(
78          composer,
79          sender_pk_A,
80          sender_pk_B,
81          note_pk_1,
82          (blinder_A_1, blinder_B_1),
83          sender_enc_out1,
84      )?;
```

Since every possible Phoenix circuit always contains 2 output notes (one for the recipient, one for the remainder to be sent back to the sender), the circuit also checks two encryptions. It starts by including two note public keys into the witness, one for each output note. Since we encrypt both points in each public key, we also include 2 blinders for each output note. Finally, the circuit asserts that the sender information encryption was performed correctly on both notes with the `assert_sender_enc` function.

```
89  fn assert_sender_enc(
90      composer: &mut Composer,
91      sender_pk_A: WitnessPoint,
92      sender_pk_B: WitnessPoint,
93      note_pk: WitnessPoint,
94      blinder: (Witness, Witness),
95      sender_enc: [(JubJubAffine, JubJubAffine); 2],
96  ) → Result<(), Error> {
97      let blinder_A = blinder.0;
98      let (enc_A_c1, enc_A_c2) =
99          elgamal::encrypt_gadget(composer, note_pk, sender_pk_A, blinder_A)?;
100
101     let blinder_B = blinder.1;
102     let (enc_B_c1, enc_B_c2) =
103         elgamal::encrypt_gadget(composer, note_pk, sender_pk_B, blinder_B)?;
104
105     let sender_enc_A = sender_enc[0];
106     let sender_enc_B = sender_enc[1];
107
108     composer.assert_equal_public_point(enc_A_c1, sender_enc_A.0);
109     composer.assert_equal_public_point(enc_A_c2, sender_enc_A.1);
110
111     composer.assert_equal_public_point(enc_B_c1, sender_enc_B.0);
112     composer.assert_equal_public_point(enc_B_c2, sender_enc_B.1);
113
114     Ok(())
115 }
```

This function indeed takes all sender information and the provided blinders and then computes the encryption in-circuit with the ElGamal gadget. The note's encrypted sender information is then compared with the results of the gadget. All variables are placed correctly here and thus this sender encryption gadget should be entirely correct.

FULL TRANSACTION CIRCUIT

The full transaction circuit, minus the previously covered gadgets, should prove correctness of the transaction in 5 steps:

- Prove that all notes in the circuit are included in the note merkle tree.
- Prove that the sender owns the notes that are being used as inputs to the transaction.
- Prove correct computation of the nullifier for each input note.
- Prove correct computation of the value commitments for each output note.
- Prove that the balance of the inputs notes is equal to the balance of the output notes plus the maximum fee and deposit values.

Let's go over each step in the full transaction circuit found in `phoenix/circuits/src/circuit_impl.rs`. Firstly, let's check the membership proof. I'll show the setup for this function, and show the meat of it below:

```
50          // Append the root as public input
51          let root_pi = composer.append_public(self.root);
```

```
55          // Check membership, ownership and nullification of all input notes
56          for input_note_info in &self.input_notes_info {
```

The merkle tree root is first added to the witness. Then, for each input note, we compute the following:

```
100             // Commit to the value of the note
101             let pc_1 = composer.component_mul_generator(value, GENERATOR)?;
102             let pc_2 = composer
103                 .component_mul_generator(value_blinder, GENERATOR_NUMS)?;
104             let value_commitment = composer.component_add_point(pc_1, pc_2);
105
106             // Compute the note hash
107             let note_hash = HashGadget::digest(
108                 composer,
109                 Domain::Other,
110                 &[
111                     note_type,
112                     *value_commitment.x(),
113                     *value_commitment.y(),
114                     *note_pk.x(),
115                     *note_pk.y(),
116                     pos,
117                 ],
118             )[0];
119
120             // Verify: 1. Membership
121             let root = opening_gadget(
122                 composer,
123                 &input_note_info.merkle_opening,
124                 note_hash,
125             );
126             composer.assert_equal(root, root_pi);
```

We compute its hash and finally use the Plonk `opening_gadget` to compute the root from the note's merkle opening path and its hash. Finally, we ensure that for each note, this produced root equals the note merkle tree root supplied by the prover. This step is thus considered correct.

Next up, we will check the ownership part of the circuit:

```
55          // Check membership, ownership and nullification of all input notes
56          for input_note_info in &self.input_notes_info {
57              let (
58                  note_pk,
59                  note_pk_p,
60                  note_type,
61                  pos,
62                  value,
63                  value_blinder,
64                  nullifier,
65                  signature_u,
66                  signature_r,
67                  signature_r_p,
68              ) = input_note_info.append_to_circuit(composer);
69
70              // Verify: 2. Ownership
71              gadgets::verify_signature_double(
72                  composer,
73                  signature_u,
74                  signature_r,
75                  signature_r_p,
76                  note_pk,
77                  note_pk_p,
78                  payload_hash,
79              )?;
```

In this context, we verify ownership by verifying the given signatures on each note with the corresponding public keys. As long as all signatures verify correctly, we can assume that the sender indeed owns the notes that are being proven in the circuit. Therefore, this step is also correct.

Let's move on to the nullifier computation part of the circuit:

```
55          // Check membership, ownership and nullification of all input notes
56          for input_note_info in &self.input_notes_info {
57              let (
58                  note_pk,
59                  note_pk_p,
60                  note_type,
61                  pos,
62                  value,
63                  value_blinder,
64                  nullifier,
65                  signature_u,
66                  signature_r,
67                  signature_r_p,
68              ) = input_note_info.append_to_circuit(composer);
```

```
81                  // Verify: 3. Nullification
82                  let computed_nullifier = HashGadget::digest(
83                      composer,
84                      Domain::Other,
85                      &[*note_pk_p.x(), *note_pk_p.y(), pos],
86                  )[0];
87                  composer.assert_equal(computed_nullifier, nullifier);
```

The circuit re-computes the nullifier according to the spec. That is, we take the supplied note public key and the note's position in the merkle tree, and perform a Poseidon hash to arrive at the nullifier value. We then assert equality between this produced value and the provided value on each note. The nullifier computation part of the circuit is therefore correct.

Next up is the checking of the correctness of the value commitments on the output notes (we like to clarify that it should only be done for output notes, as input notes will have this checked in the circuit in which they were minted previously):

```
131         // Commit to all output notes
132         for output_note_info in &self.output_notes_info {
133             // Append the witnesses to the circuit
134             let value = composer.append_witness(output_note_info.value);
135             // Append the value-commitment as public input
136             let expected_value_commitment =
137                 composer.append_public_point(output_note_info.value_commitment);
138             let value_blinder =
139                 composer.append_witness(output_note_info.value_blinder);
```

```
149             // Commit to the value of the note
150             let pc_1 = composer.component_mul_generator(value, GENERATOR)?;
151             let pc_2 = composer
152                 .component_mul_generator(value_blinder, GENERATOR_NUMS)?;
153             let computed_value_commitment =
154                 composer.component_add_point(pc_1, pc_2);
155
156             // Verify: 4. Minting
157             composer.assert_equal_point(
158                 expected_value_commitment,
159                 computed_value_commitment,
160             );
161         }
```

For each output note, we perform the pedersen commitment computation with the given `value` and `value_blinder`. This computed commitment is then constrained to be equal to the provided value commitment of the output note in question. This part of the circuit is thus deemed correct.

Finally, we need to do the balance integrity check. Besides checking that the sum of input notes minus the sum of output notes, deposit and maximum fee corresponds to zero, we also need to ensure that each note's value is below 2^{64}. Each input and output note is indeed range checked to be below 64 bits in the following parts of the circuit:

```
55        // Check membership, ownership and nullification of all input notes
56        for input_note_info in &self.input_notes_info {
```

```
89            // Perform a range check ([0, 2^64 - 1]) on the value of the note
90            composer.component_range::<32>(value);
```

```
131       // Commit to all output notes
132       for output_note_info in &self.output_notes_info {
```

```
141           // Perform a range check ([0, 2^64 - 1]) on the value of the note
142           composer.component_range::<32>(value);
```

Since the range gadget needs to take in the bits of range divided by two, these range checks are indeed performed correctly.

Next up, we ensure that the sum

$$\sum_{i=1}^{s} w_i - \sum_{j=1}^{r} v_j - deposit - max\_fee = 0$$

is indeed checked correctly:

```
53      let mut input_notes_sum = Composer::ZERO;
54
55      // Check membership, ownership and nullification of all input notes
56      for input_note_info in &self.input_notes_info {
57          let (
58              note_pk,
59              note_pk_p,
60              note_type,
61              pos,
62              value,
63              value_blinder,
64              nullifier,
65              signature_u,
66              signature_r,
67              signature_r_p,
68          ) = input_note_info.append_to_circuit(composer);
```

```
92              // Sum up all the input note values
93              let constraint = Constraint::new()
94                  .left(1)
95                  .a(input_notes_sum)
96                  .right(1)
97                  .b(value);
98              input_notes_sum = composer.gate_add(constraint);
```

Here, we compute $\sum_{i=1}^{s} w_i$. Note that each input note's value is added to the `input_notes_sum` variable, and the result of this is set back to the `input_notes_sum` variable, so the sum is indeed correctly computed.

Next up, the summing of output note values:

```
129         let mut tx_output_sum = Composer::ZERO;
130
131         // Commit to all output notes
132         for output_note_info in &self.output_notes_info {
133             // Append the witnesses to the circuit
134             let value = composer.append_witness(output_note_info.value);
135             // Append the value-commitment as public input
```

```
144             // Sum up all the output note values
145             let constraint =
146                 Constraint::new().left(1).a(tx_output_sum).right(1).b(value);
147             tx_output_sum = composer.gate_add(constraint);
```

Similarly, for each value we sum it to `tx_output_sum` and then assign the result back to `tx_output_sum`. Thus, $\sum_{j=1}^{r} v_j$ is indeed computed correctly as well. Finally, we check the identity:

```
163         // Append max_fee and deposit as public inputs
164         let max_fee = composer.append_public(self.max_fee);
165         let deposit = composer.append_public(self.deposit);
166
167         // Add the deposit and the max fee to the sum of the output-values
168         let constraint = Constraint::new()
169             .left(1)
170             .a(tx_output_sum)
171             .right(1)
172             .b(max_fee)
173             .fourth(1)
174             .d(deposit);
175         tx_output_sum = composer.gate_add(constraint);
176
177         // Verify: 5. Balance integrity
178         composer.assert_equal(input_notes_sum, tx_output_sum);
```

The circuit appends the max fee and deposit values to the witness, and adds them to the `tx_output_sum` variable. It is not needed to range check these variables as they are public knowledge and can be checked for correctness outside of the circuit.

Since we have the identity

$$\sum_{i=1}^{s} w_i - \sum_{j=1}^{r} v_j - deposit - max\_fee = 0$$

We can rearrange this to

$$\sum_{i=1}^{s} w_i = \sum_{j=1}^{r} v_j + deposit + max\_fee$$

And indeed, we check this equality on the last line of the above code block. Therefore, the balance integrity step is entirely correct.

With all these steps checked for correctness, as well as all the peripheral gadgets deemed correct, we dictate that the Phoenix transfer circuit is entirely correctly implemented according to the specification.

ISSUES FOUND

Let's now proceed to cover the security vulnerabilities found while auditing the two codebases, as well as the results from the static analysis tools.

STATIC ANALYSIS TOOL RESULTS

For Rust, there do not exist many out-of-the-box security tools that do not make use of some sort of description language or extraneous implementation of code. I also want to note that fuzzing seems to be of little use to this implementation; it would be deemed useful for things such as the underlying cryptography implementations, but these are out of scope for this audit. Similarly, constructing a fuzzer for the circuit would be a large undertaking, and was not deemed possible for the audit timeline.

CARGO AUDIT

Cargo audit returned no vulnerable imports. However, imports should be secured as per the minor issue found where imports are not patch-frozen.

TEST COVERAGE

Test coverage reports revealed that all critical logic is tested, and will require no further coverage work.

From a static analysis perspective, the respective codebases have a clean bill of health.

MINOR: EXTERNAL, OPEN SOURCE DEPENDENCIES SHOULD BE PATCH-FROZEN

When importing external crates to your repository, you open yourself up to potential attacks. Specifically, an attacker could manage to sneak in malicious code through a patch in a crate, and then exploit another codebase relying on this crate when it is merged in. Most recently, such an attack was performed on BitTensor and caused a drain of wallets. Attackers made out with the money through various exchanges and recovery efforts are still ongoing.

A good way to avoid this is to patch-freeze any external dependencies that accept open source contributions. Since the dusk code is open source and (potentially) could accept contributions from outsiders, it is recommended that all imports are frozen on

Let's provide an example of patch-freezing. Consider the `rkyv` import, found in the out-of-circuit portion of the Phoenix repository:

```
28    rkyv = { version = "0.7", default-features = false, features = ["size_32"] }
```

We can patch-freeze it by this change:

```
28    rkyv = { version = "0.7.43", default-features = false, features = ["size_32"] }
```

TEAM RESPONSE

The Dusk team mentioned that this fix should apply to their whole stack, and as of the date of this document will be working on implementing it across the entire range of their repositories.

MINOR: ERROR-PRONE USAGE OF ZEROIZE

A few structs containing sensitive information in the Phoenix repository derive the `Zeroize` trait, to reduce potential attacks from malicious software that would read memory past its legal bounds. `Zeroize` allows the programmer to zero out any memory used before the struct goes out of scope.

It is instead advised to derive `ZeroizeOnDrop` where `Zeroize` is derived, as this will make the zeroing out of memory happen automatically as the struct goes out of scope, and takes the mental burden of ensuring this is done correctly everywhere off the programmer.

TEAM RESPONSE

Dusk's response to this issue was that the `ZeroizeOnDrop` derivation was tested, but found to be unreliable and would not always work on `drop()`. Therefore, the advisory is adjusted instead to ensure that `Zeroize` is always called when dealing with sensitive information, as usage of the Phoenix libraries go outside of the scope of this audit. More information on the team's response can be found in issue #244 in the Phoenix repository, and an explanatory comment was added in pull request #245.

EFFICIENCY: REDUNDANT COMPUTATION IN INPUT NOTE SECTION OF CIRCUIT

In `phoenix/circuits/src/circuit_impl.rs`, for each input note, to compute the note hash the circuit first computes the value commitment with the given note information. I postulate that it is not necessary to compute this commitment in-circuit and can instead be appended to the witness to save on constraints and make the circuit slightly less complex. To retain soundness of the circuit, we instead suggest that, since the cleartext values and blinders are known to the prover, all witness-appended commitments can instead be summed up, and the circuit's summed up input values can then similarly be committed to with the summed up blinders, and an equality check can be performed. This should reduce some amount of constraints between the different transaction circuits for different input note amounts.

For reference, this is the section of code affected:

```
55          // Check membership, ownership and nullification of all input notes
56          for input_note_info in &self.input_notes_info {

100             // Commit to the value of the note
101             let pc_1 = composer.component_mul_generator(value, GENERATOR)?;
102             let pc_2 = composer
103                 .component_mul_generator(value_blinder, GENERATOR_NUMS)?;
104             let value_commitment = composer.component_add_point(pc_1, pc_2);
```

TEAM RESPONSE

After speaking with the Dusk team about this optimization, we came to the conclusion that, while it would cut constraints, it's likely not going to have a noticeable effect. In the Dusk Plonk implementation, the performing of fixed-base multiplication on the JubJub curve costs around 260 constraints. Therefore, we cut maximally ~1500 constraints in the worst case. Xavier listed the different circuit constraint counts:

- 1 input: 46321
- 2 inputs: 73646

- 3 inputs: 100971

- 4 inputs: 128296

It thus seems that for none of the circuits, the cut in constraints would be large enough to drop the circuit in its degree. Thus we came to the conclusion that this optimization would not be necessary, especially as the change in code would need to be more closely audited for soundness.

STRUCTURAL: COPIED CODE IN OWNS METHOD FOR SECRET KEY

In `phoenix/core/src/keys/secret.rs`, the method `owns` computes whether or not a given note public key corresponds to an owned private key. The first 3 lines compute the note secret key, which are also implemented in the method `gen_note_sk`. It would be advised to not repeat code and instead use the `gen_note_sk` method to compute the note secret key, and then retrieving the value out of the generated struct for further computation. This will increase readability and reduce potential error if this logic needs changing or fixing.

TEAM RESPONSE

This issue was addressed in the Phoenix repository, pull request #247.

STRUCTURAL: VIEW KEY TODO COMMENT FOR CT EQUALITY CHECK

In `phoenix/core/src/keys/view.rs`, there is a lingering `TODO` comment in the implementation of the constant time equality check:

```
33  impl ConstantTimeEq for ViewKey {
34      fn ct_eq(&self, other: &Self) → Choice {
35          // TODO - Why self.a is not checked?
36          self.B.ct_eq(&other.B)
37      }
38  }
```

It's indeed a valid question, especially since constant time equality checking should be deemed more safe than any standard method; and also because this is actually properly done on the secret key structure, where two JubJub scalars are present.

TEAM RESPONSE

This was fixed in pull request #250.

DOC: CONFUSING DOCUMENTATION COMMENT ON NOTE STRUCTURE

On the `Note` struct found at `phoenix/core/src/note.rs`, the following comment is found:

```
68    /// A note that does not encrypt its value
```

The file does offer functionality for creating encrypted notes, so this documentation comment is invalid and should be updated to reflect the current state of the logic, to avoid confusion.

TEAM RESPONSE

The Dusk team fixed this comment in pull request #249.