Zcash Sapling audit report

Oct 25thth 2018

Revision	Date	Description
1	Sep. 9, 2018	Sapling review delivery
2	Oct. 25, 2018	Blake2s analysis

Overview

The QED-it team has been hired by the Zcash company in order to audit the implementation of the Sapling protocol, as preparation to the Sapling activation in October.

Together with the Zcash team, we have decided to focus mostly on the sapling-crypto (https://github.com/zcash-hackworks/sapling-crypto) and bellman (https://github.com/zkcrypto/bellman) repositories.

These repositories contain the SNARK-critical code, which is the hardest to change and reason about. The Zcash team has implemented from scratch the components comprising Sapling's Spend and Output circuits, a process which is similar to the work we're doing at QED-it.

As a starting point, we have chosen to focus on the following primitives:

- Pedersen hashes + lookup
- Note commitments
- Merkle tree using pedersen hashes
- Spend/Output connection using homomorphic pedersen commitments
- Boolean utils - decomposition into bits, boolean operations
- Group hashes
- Fixed_base_multiplication
- Other parts of protocol (generation of nullifiers, etc)

As a guideline, we have put adding tests and matching the protocol specification as top priorities, and have followed this general process:

- Identify high-value, risky protocol parts using our knowledge, guidance from the Zcash team and issues in the sapling-crypto repository
- Review the theory and the protocol specification
- Review the code and write tests to uphold the theory

This process resulted in a series of audited units, some of which are reviews that allow us to ensure the different corresponding components are in order, and some of which were identified issues and potential problems.

The deliverables in this review are:

- This report, detailing the work that has been done by the team.

- The sapling-crypto-internal repository - fork of the sapling-crypto repository, containing tests, test vectors and comments.
- The go-jubjub repository - containing an independent, spec-oriented implementation in Go of most of the cryptographic primitives in Sapling related to Jubjub.
- The juby repository - containing an independent, spec-oriented implementation in Python of cryptographic primitives in Sapling related to Jubjub.
- The zcash-test-vectors-internal repository - fork of zcash-test-vectors repository, containing test vectors and some analysis.

In conclusion, during the review, we have **not** identified any security issues in the protocols. That said, we have identified some bugs and have recommendations to prevent future potential issues.

The team that worked on the audit:

- Daniel Benarroch
- Aurelien Nicolas
- Kobi Gurkan

Issues

Incorrect assertion - Pedersen hash generators linear relation validation

- **Description:** as part of creating the generators for the pedersen hash, it's important to check the generators are linearly independent, or else you might be able to generate collisions
- **Severity**: Trivial
- **Code reference:** https://github.com/arielgabizon/sapling-crypto/blob/003a150e587bf439dcbf1120006654cac4d97542/src/jubjub/mod.rs#L247
- **Recommendation:** add more comprehensive assertions for linear independence, fix error:
  let sum -> let mut sum
  remove shadowed let

Window base patterns generation not matching the spec

- **Description:** This does not look like a neutral change in fn synth.
- **Severity**: Low
- **Code reference:** https://github.com/zcash-hackworks/sapling-crypto/pull/75/files#diff-0dd64565d600b954396341f14a58f1cfL13
- **Recommendation:** add test. Clarify what formula is being computed by the code.

- Break down of what is computed before and after the change. Basically the difference is whether the constant is included in the negation or not.

Before:

new_assignment = constant - old_assignment
new_eval = old_eval - old_assignment + constant

After:

new_assignment = constant + old_assignment
new_eval = old_eval - old_assignment - constant

jubjub::Fs usage is sometimes confusing regarding the prime subgroup

Description: Fs should describe the scalar field of jubjub, but numbers are taken modulu the order of the prime subgroup, i.e. when multiplying
Severity: Low
Code reference:
https://github.com/zcash-hackworks/sapling-crypto/blob/e4175d81e9dfca3ab4e1e72019f326b9d9bd1b6d/src/jubjub/fs.rs
Recommendation: make the MODULUS Fs uses to be the order of jubjub rather than the prime subgroup, or make jubjub only usable within the prime subgroup

Pedersen hash circuit implementation can not calculate inputs larger than 63*3*4

Description: The circuit implementation in src/circuit/pedersen_hash.rs, when given an input larger than 63*3*4 bits, uses up all the 5 generators, as it should. In the next iteration, rather than stopping, it tries to use the unexisting 6th generator. The non-circuit implementation can correctly support inputs up to 63*3*5.

While not a pressing issue, since all the inputs around the circuit seem to be less than 63*3*4 , it does not seem to be the intended behaviour of the implementation.

Spec notation mismatches and missing parts

Description: when reviewing the sapling-crypto repository and the specification side by side, we found a few notation mismatches which made the comparison less obvious, or missing parts which required to use the code as a reference

References:

- $abst_J$ is not explicitly defined. One condition that wasn't immediately obvious was the fact that y coordinates which are bigger than the modulus should be rejected (after further examination, it is clear that it should be so, since the function needs to be injective). Recommending to add pseudo-code of it:
    - Given bits, assign odd = bits[255] and assign bits[255]=0
    - Interpret bits[0..255] as a y coordinate of Jubjub and assert it is smaller than r_j
    - Try to find a matching x for this y. When found, if $odd \neq (x \mod 2)$, assign $x = (-x \mod r_j)$
- In section 5.4.1.7 when describing PedersenHash function, the spec claims that it "is used in the *incremental Merkle Tree* over *note commitments"* and points to section 3.7. One would expect to find there the matching description of how the PedersenHash function is used in the NoteCommitment, instead there is a very abstract description of the actual Merkle Tree. Would be useful to point instead to section 5.4.1.3 where the spec describes the function MerkleCRH which uses the PedersenHash function, differentiated from its use in the NoteCommit
- The notes in section 5.4.1.3 and 5.4.7.2 which explain that the prefixes used in MerkleCRH and in NoteCommit are used to differentiate the usage of PedersenHashToPoint and prevent a collision. This is not clear from the two notes, which can cause confusion in the sense that it makes one think that the MerkleCRH uses WindowedPedersenCommit function.
- PedersenHash expected constraints amount was not listed. We addressed it in the "Deeper analysis of PedersenHash" section
- r_j is used to denote the order of the prime subgroup in the spec, while called s in the sapling-crypto code
- In section A.3.3.3, the birationally equivalent Montgomery curve equation of Jubjub, specify B_M=1 and A_M=40962.
- Theorem A.3.2. appears to have a typo - seems like the correct condition is $u = 0 \text{ or } v = 0$. Additionally, there's no mention of the fact that this covers the condition $1 - v \neq 0$
- When describing the constraints for the lookup table of the WindowedPedersenCommit, it is not clear how the interpolation was done to vanishing polynomials. It took us some time to figure out that the general way of

creating similar polynomials for constraints derived from 3 bits b_0, b_1, b_2 and encoding outputs $c_1, ..., c_8$ is with the following constraint:
$C(b_0, b_1, b_2) = c_0(1 - b_0)(1 - b_1)(1 - b_2) + c_1(b_0)(1 - b_1)(1 - b_2) + \cdots + c_7(b_0)(b_1)(b_2)$

- NoteCommit, MerkleCRH and MixingPedersenHash, while defined as separate functions in the spec, are defined only implicitly as unseparated code sections inside the Spend circuit, making analysis more difficult - we recommend to abstract them into distinct functions

The bit unpacking between layers of the Merkle tree

Description: This enforces congruent equality only (using into_bits_le, not into_bits_le_strict). About 10% of PH outputs have a congruency in Fr. In these cases, the prover can choose between two distinct 255 bits representations as input to the next layer. This reduces the second-preimage resistance by a fraction of a bit, which is not significant.
Severity: Low

Testing is done with another implementation of ConstraintSystem

Description: TestConstraintSystem is used throughout testing while ProvingAssignment is used in proving, both implementing the ConstraintSystem trait.
While our focus was testing constraints and cryptography, we'd like to note that this is also a potential location of danger - when adding constraints and allocating variables, there could be a different behaviour.

params.montgomery_2a doesn't seem to be used anywhere

Description: while doesn't seem like an issue, this raises the question of what was the author's intended use for it.

Review

Num::into_bits_le_strict

- **Description:** used to convert a Num into its bits representation while making sure the bits representation is in the field and not just a congruence, used by EdwardsPoint for the (x,y) coordinates
- **Code reference:** https://github.com/zcash-hackworks/sapling-crypto/blob/21084bde2019c04bd34208e63c3560fe2c02fb0e/src/circuit/num.rs#L91
- **Requirements:** as mentioned, making sure the bits representation is in the field and not just a congruence

  Pseudo-code (assuming $r-1$ is char(F) - 1, and n is the number):

  $bits_{r-1} =$ bits of r−1 in descending order

  $bits_n =$ bits of n in descending order

  - In loop on bits of $r-1$:
    - Find the next bit sequence of 1s in $bits_{r-1}$
    - Constrain:
      - If the same bit sequence in $bits_n$ is all 1s, make sure that the next bit is 0 (or else, n is larger than $r-1$
      - Else, that means that in that sequence, the represented number by that sequence is shorter, and there's no constraint on the next bit
  - Constraint: Make sure the $bits_n$ bit representation is the unpacking of n

- **Test link:** https://github.com/QED-it/sapling-crypto-internal/commit/97d5675eecfc49468b8231b7de4a05389473f461

boolean::alloc_conditinoally

- **Description:** alloc conditionally receives a constant and a variable, and enforces the following: if constant is true, variable must be false. Else, variable can be anything.
- **Code reference:** https://github.com/zcash-hackworks/sapling-crypto/blob/45c73dbd8f21b11ab28c836f785333e31c506335/src/circuit/boolean.rs#L39
- **Requirements:**
    - Since constant is allocated in place, it must also validate that the allocated bit is also a boolean
    - The variable must be false if constant is true
    - The constraint used there to uphold both requirements:

$$(1 - c - v) * v = 0$$

- **Test link:** https://github.com/QED-it/sapling-crypto-internal/tree/alloc_conditionally

JubJub Implementations for generation of test vectors

- **Description:** tested the different ECC operations used inside the circuit, and generated test vectors by writing independent implementations for JubJub, based solely on the spec. One implementation in Python and the second in Go, both tested independently against each other and then against the rust implementation.
- **Code reference:** https://github.com/QED-it/juby/ & https://github.com/QED-it/go-jubjub
- **Test vectors**:
    - pedersen hashes - test vectors, testing of both non-circuit and circuit sides (where a non-security bug seems to have been identified for bit lengths > 63*3*4)
    - ph_test_vectors - many test vectors for pedersen hash
    - input_circuit_test_vectors - test vectors for NoteComit, in the context of a Spend circuit
    - group_hash - test vectors for FindGroupHash

Unbiased sampling of base points for Pedersen Hash generators

Description:
The base points for pedersen hashes should be unbiased (pseudo-random) to prevent linear relations between generators, allowing the creation of collisions.

Pseudo-code (as read from the code):
Hash the data below with Blake2, try to interpret the 32 bytes output as a point using a number of tests. If invalid, increment the trial counter and repeat.
This avoids biases in the selection (assuming the hash function is unbiased).

Hash input:

- 8 bytes of personalization (domain). For Pedersen Hash it is "Zcash_PH" in ASCII.
- 1 bytes for the segment number, 0 to 4 inclusive.
- 3 null bytes.
- 1 byte for the trial counter, starting at 0.

Conversion:

- Convert to a number from little-endian bytes.
- Heaviest bit is used as the sign of x.
- Heaviest bit is reset to give y.
- If y is bigger than the Jubjub field, discard.
- Deduce x using the curve equation. If x doesn't exist (the square root fails), discard.
- Multiply by the cofactor to remove the small subgroups component.
- If the point is zero, discard.
- Try up to 254 times.

Deeper analysis of PedersenHash

Security review of pedersen construction

- **Description:** the use of pedersen hashes and commitments comes with several non-trivial security needs and we check that these are meet and proven correctly
- **Requirements:**
    a. PedersenHash must be collision resistant: this boils down to Enc(m_j) and Extractor being injective which is achieved by skipping a bit for the image points.
        - Demonstrate injectivity by sum of powers of 2
            - First: lower bits are in a + or - range, next bit is greater or lower (in negative)
            - Second: No bigger than +-s
            - Third: Adding s any number of times is indifferent modulo s
        - Second-preimage and collision resistances of PH are effectively the same. If given a single collision pair, one can work out the relationship between segment generators, breaking the scheme.
    b. GroupHash, acting as a random oracle, derives a generator of the subgroup of order r_j of the JubJub curve: by using the BLAKE2s hash function to derive a point in JubJub, it is ensured that it acts as a random oracle. Then multiplying this point by the co-factor, h_j, ensures (with high probability) the new point is in the prime subgroup, where any element is a generator.
    c. MixingPedersenHash[WindowedPedersenCommit_r(M),x] must be collision-resistant on (r, M, x). This boils down to WindowedPedersenCommit function being collision-resistant on (r, M) since otherwise one could break the DLP from $P + [x]J = P' + [x']J$.
    d. WindowedPedersenCommit must be both computationally binding (collision-resistant), which boils down to PedersenHashToPoint being collision resistant (which it is) and computationally hiding (but is statistically hiding based on the addition of a random point of the curve, as long as “r” is randomly generated.

Other Security aspects

1. **Faerie gold protection**

a. **Description:** These are the steps to prevent the faerie gold attack, by making nullifiers unique:
    i. The nullifier includes the note position in rho.
    ii. Rho is the sum of the PH of the note, and [position].NullifierPosition.
    iii. The generators of PH and NullifierPosition are independent, which binds the nullifier to the position.
b. **Note**: The faerie gold protection does not apply for dummy notes. When value=0, the note and position are not validated at all, so one may be able to create and nullify faerie gold notes of 0 value. The nullifiers involved are bound to the 0 value since it is part of the note.

2. **Malleability:** PH is malleable.
    a. Given a hash and some known or guessed preimage bits, one can construct the hash of the preimage with known bits replaced by any other values.
    b. For instance PH(b || secret) from PH(a || secret).
    c. One can generate a pair of inputs that hash to values that are opposite in the Fr field, by flipping every third preimage bits.
    d. Similarly, given a hash but not its preimage, one can find the hash of a preimage with every third bits flipped by negating the hash value.
3. **General attacks:** Attacks against PH affect all instances. This is a practical advantage to a potential attacker. No general attacks on the construction have been found.

Padding

PH is padded with zeros to a length multiple of 3 bits. The spec elaborates about the fact that there is a maximum of 63*5*3 = 945 bits as input to the PedersenHash function, yet in particular, sapling uses fixed-sized inputs of 516 and 582 bits, so no padding occurs since these are already multiples of 3. Indeed, it seems like the protocol does not use the fifth generator, see issue above.

Constraints checking

1. **Lookup table:** Find (x, y) from a and b bits in a table (Xi, Yi).
    a. Precompute:

    Xa = X1 - X0

    Xb = X2 - X0

    Xab = X3 - X0 - Xa - Xb

    And the same for Y.

    b. Constraints: Bitness of a, b, c is guaranteed by code calling PH:

i. For note commitment: expose_value_commitment, EdwardsPoint.repr()
ii. For Merkle path: into_bits_le()

c. Constraint: ab = a AND b
(a) * (b) = (ab)

d. Coordinates are just linear combinations of the a, b, ab bits.
x = X0 + a.Xa + b.Xb + ab.Xab
y' = Y0 + a.Ya + b.Yb + ab.Yab

e. Constraint: conditionally negate the point. In Montgomery coordinates, negate the y coordinate.
(2 * y') * (c) = (y' - y)

2. **Breakdown of PH constraints:**

a. **Spec**: « The cost of a Pedersen hash over l bits (where l includes the fixed bits) is ... constraints. In particular, for the Merkle tree hashes l = 516, so the cost is ... constraints. »

i. *Per chunk:*
3-bits lookup: 2 constraints
A AND b
Conditional negation of y

ii. Between chunks, 62 times per full segments:
Montgomery add: 3 constraints
lambda = (y' - y) / (x' - x)
(lambda) * (lambda) = (A + x + x' + x'')
y' + y = lambda(x - x')

iii. Per segments:
into_edwards: 2 constraints
u = (scale*x) / y
v = (x - 1) / (x + 1)

iv. Between segments:
Edwards add: 6 constraints
A = y2 * x1
B = y1 * x2
C = d*A*B
x3 = (A + B) / (1 + C)Per chunk:

v. 3-bits lookup: 2 constraints
a AND b
Conditional negation of y

vi. Between chunks, 62 times per full segments:

Montgomery add: 3 constraints

lambda = (y' - y) / (x' - x)

(lambda) * (lambda) = (A + x + x' + x'')

y' + y = lambda(x - x')

vii. Per segments:

into_edwards: 2 constraints

u = (scale*x) / y

v = (x - 1) / (x + 1)

viii. Between segments:

Edwards add: 6 constraints

A = y2 * x1

B = y1 * x2

C = d*A*B

x3 = (A + B) / (1 + C)

y3 = (U - A - B) / (1 - C)df

ix. Example: 516 bits input.

2 segments of 63 chunks

2 * (63 * 2 + 62 * 3)

1 segment of 46 chunks

46 * 2 + 45 * 3

3 Edwards conversions

3 * 2

2 Edwards add

2 * 6

= 869 constraints

Deeper analysis of blake2s

High level overview

Following the specification in https://blake2.net/blake2.pdf, blake2s has the following functions:

- XORs (pages 19-20)
- Compress function using G function (page 5):
    - 32-bit unsigned integer additions, ignoring overflows (in the sense that addition is mod 2^32)
    - Right rotations
    - XORs

It also has the following data structures:

- 32-byte constant **IV**
- 32-byte parameter block **P**, containing a personalization, which is constant for each of the two uses
- 32-byte chain value **h**
- Counters **t0, t1**, each 4 bytes
- Finalization flags **f0, f1**, each 4 bytes

Without describing the entire hash function, which matches the protocol specification, the following danger areas seem important when considering a circuit implementation:

- XORs assumes inputs are bits, outputs are bits as well
- 32-bit unsigned integer additions assumes the inputs are of 32-bit size, which would have to be verified, and overflows must be handled correctly
- Rotations are bit operations, so inputs must be bits

Code and constraints checking

This analyzes the constraints for the case where we have 512 input bits and all the input bits are variables.

Bits allocation

All bits are represented using a Boolean structure. A bit can be either a constant, or an allocated variable. A bitness constraint is always added to a new variable.

Initialize h

This initializes **h** as the XOR of IV and the parameter block (which constraints the personalization). Since all of these are constant for each use, no constraints are involved yet.

Outer function

Blake2s operates on blocks of 512 bits, each fed to a **compress** function, receiving the current chain value **h**, the counters **t0, t1** and the finalization flags **f0, f1**.

We note the in the two circuit uses, there is only one block: for the nullifier, there is an input of 256-bit and for the ivk, it's 512 bits.

Therefore, the circuit code for inputs larger than 512 bits is unused in the protocol, for which we have added tests.

Additionally, the circuit handles both constant input bits and variable input bits with padding, while the two uses only use variable input bits without padding, to which we have also added tests.

Lastly, input bits are given as little-endian sequences, in the sense that bytes are ordered as little-endian and the bits are ordered from least significant to most significant.

XOR module

The module assumes that the inputs **a** and **b** are Booleans, which is correct in this case. If one of those is a constant, then no constraints are added, and the appropriate linear combination is returned. If both are variable, there's a single constraint (a xor b = c): (a + a)*b = a + b -c, which has the following truth table:

```
a=0,b=0 => 0*0 = 0 + 0 - c                => c = 0
a=0,b=1 => 0*1 = 0 + 1 - c => 0 = 1 - c => c = 1
a=1,b=0 => 2*0 = 1 + 0 - c => 0 = 1 - c => c = 1
a=1,b=1 => 2*1 = 1 + 1 - c => 2 = 2 - c => c = 0
```

which is the appropriate truth table for XOR. It also allocates a variable for c.

MultiEq module

Uses a method of calculating the following linear combination and requiring it to be zero:
$\sum 2^{\texttt{cur\_bits}} * (lhs_i - rhs_i)$

and advance **cur_bits by** b_i, where b_i is the number of bits in lhs_i/rhs_i, and **cur_bits** is the number of bits accumulated so far.
Eventually, as long as the number of bits is less than the field size (255 bits), this is one constraint.
When the number of bits goes over the field size, another constraint is added automatically.

This is used throughout blake2s to gather as many equality checks as possible into one constraint.

Uint32 module

Since Blake2s works mostly with 32-bit integers, the input bits and other data structures are stored as the **Uint32** type defined in sapling-crypto. This type still maintains the integers as bits sequences.

from_bits

Builds a Uint32 from a sequence of little-endian bits, which it takes as references.

addmany

Adds multiple Uint32s together, accounting for whether the operands are all constant or at least one of them isn't. If all are constant, no constraints are added, and the constant value is returned.

Additionally, depending on the number of operands, extra bits are allocated. Essentially, the number of operands is multiplied by the maximum Uint32 value, and a rounded up to an integer number of bits is allocated.

If at least one of the operands isn't constant, then **result_bits** are allocated, which will contain the bits of the result. These are equated, using the MultiEq module, to the value of the linear combination of the bits of each operand multiplied by the appropriate power of 2.

rotr

Rotates the bits to the right. This just changed the order of references inside the Uint32 so no new variables and constraints are added.

mixing_g

In the uses in blake2s:

- 2 times:
    - Addmany is used to add 2 **v** values and a Uint32 from the message - 3 operands, so we have to allocate `ceil(log_2(3*(2^32-1))) = 34` bits and boolean constraint them. In total, 34 constraints.
    - XOR is used, which contributes 32 constraints (when not involving a constant), and then ROTR which doesn't contribute constraints.
    - Addmany is used to add 2 **v** values - 2 operands, so we have to allocate `ceil(log_2(2*(2^32-1))) = 33` bits and boolean constraint them. In total, 33 constraints.
    - We are using XOR, which contributes 32 constraints, and then ROTR which doesn't contribute constraints.

- It's important to note that in the first pass, the first two **XOR**s in the first 4 mixing_g invocations, has **v[d]** and **v[b]** as constants (d=12,13,14,15, b=4,5,6,7). This causes them to be constants. So all in all, it saves 2*4*32 = 256 constraints for the **XOR**s.
- Eventually, we get: 2*(34+2*32+33) = 262 constraints (without equality checks), or 198 constraints for the first 4 invocations.

blake2_compression

Starts with the 8 Uint32s of **h** and extends it with the 8 Uint32s of **IV** and stores them in **v** as a sequence of Uint32s, as references and constants.
Then 2 XORs to update v[12] and v[13] with the counters **t0, t1**. At this points, the values are constant, so no constraints are added.

The XORs themselves are of two Uint32s, which operate bit-wise. Each bit-wise operation is implemented as mentioned in the XOR module analysis.

Then, depending on whether this is the final block, XORs v[14] with 0xFFFFFFFF. Since we always have <= 512 bits inputs, this always happens.
These XORs operate on constants - **v** is constant for each of the uses, so no constraints are added.

Then we have 10 iterations of the loop containing **mixing_g** function which is invoked 8 times, contributing 9*8*262+4*262+4*198 = 20704 constraints.

Lastly, there's an update to the 8 Uint32 composing **h** by XORing each **h**[i] twice with **v**[i] and **v**[8+i]. Since the **h[i]**s are constants to begin with, these contribute 8*32=256 constraints.

Lastly, we get the batched equality checks. In our checks, we had the following in the mixing_g function:

- 33 bits
- 34 bits
- 33 bits
- 34 bits

We invoked mixing_g 8 times in the loop in compression_function, and then iterated 10 times. The MultiEq module, when operated in this way, can batch 7 equality checks: 33+34+33+34+33+34+33=234, and then start a new batch. In total, we had 10*8*4=320 of these operations, which fit in ceil(320/7) = 46 constraints.

In total, we have 20704+46+256 = 21006 constraints.

Security requirements

Blake2s is used in the circuit in two places (and three in the protocol over all) - as the PRF for the nullifier and the hash function used to generate the “incoming viewing key”.
In both of those places, it seems reasonable to use a hash function rather than a pedersen commitment, since breaking DLP would seemingly allow attackers to either nullify an existing note (in the case of the nullifier PRF), or even spend it (in the case of the ivk). Additionally, there is a requirement for the nullifier derivation in the spec to be a PRF which is not entirely obvious.

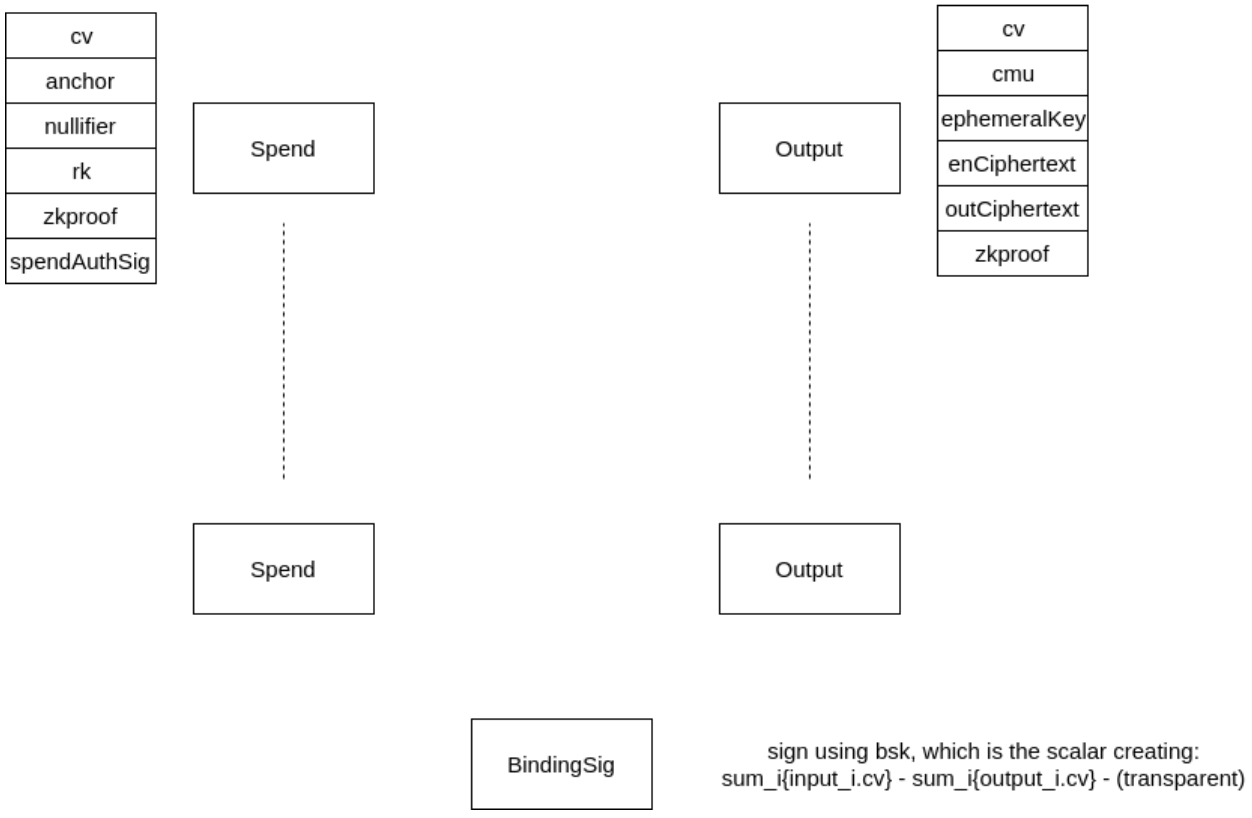
Review and suggestions

1. The total number of constraints seems to be wrong in the specification, adding extra 256 constraints from the first 4 invocations of the **mixing_g** function, when some of the values are constants, as described above. The code, however, does assert for the same amount of constraints as was calculated above - the code asserts for 21518 constraints, which is the 21006 calculated here + 512 constraints for the boolean input values. There are constraints for all the needed steps.
2. The blake2s function, for the uses in the protocol, matches bit-by-bit to the Go standard library implementation for blake2s, after us adding support for personalization to the standard library code. This code can be found in the go-jubjub repository.
3. We have added tests showing different uses of the blake2s function, which supports more features than is used and tested in the protocol, using test vectors from go-jubjub. The tests exist in https://github.com/QED-it/sapling-crypto-internal/tree/blake2s_review
4. The review went to describe the code step by step, and the specification, in its large part, matches the code very closely. The different constraints and the invariants required for them to hold are described here and seem to hold.
5. The motivation for the use of blake2s in the two cases in the circuit is not obvious.
    a. For the nullifier, the requirement of it being a PRF gives some hint about the attacks it might prevent (regarding being able to nullify notes you don’t own), although some description of possible attacks would have been helpful, and whether they are critical.
    b. For the ivk, it would be helpful to address what justifies the jump from a Pedersen commitment to blake2s (i.e., as the example we provided above).

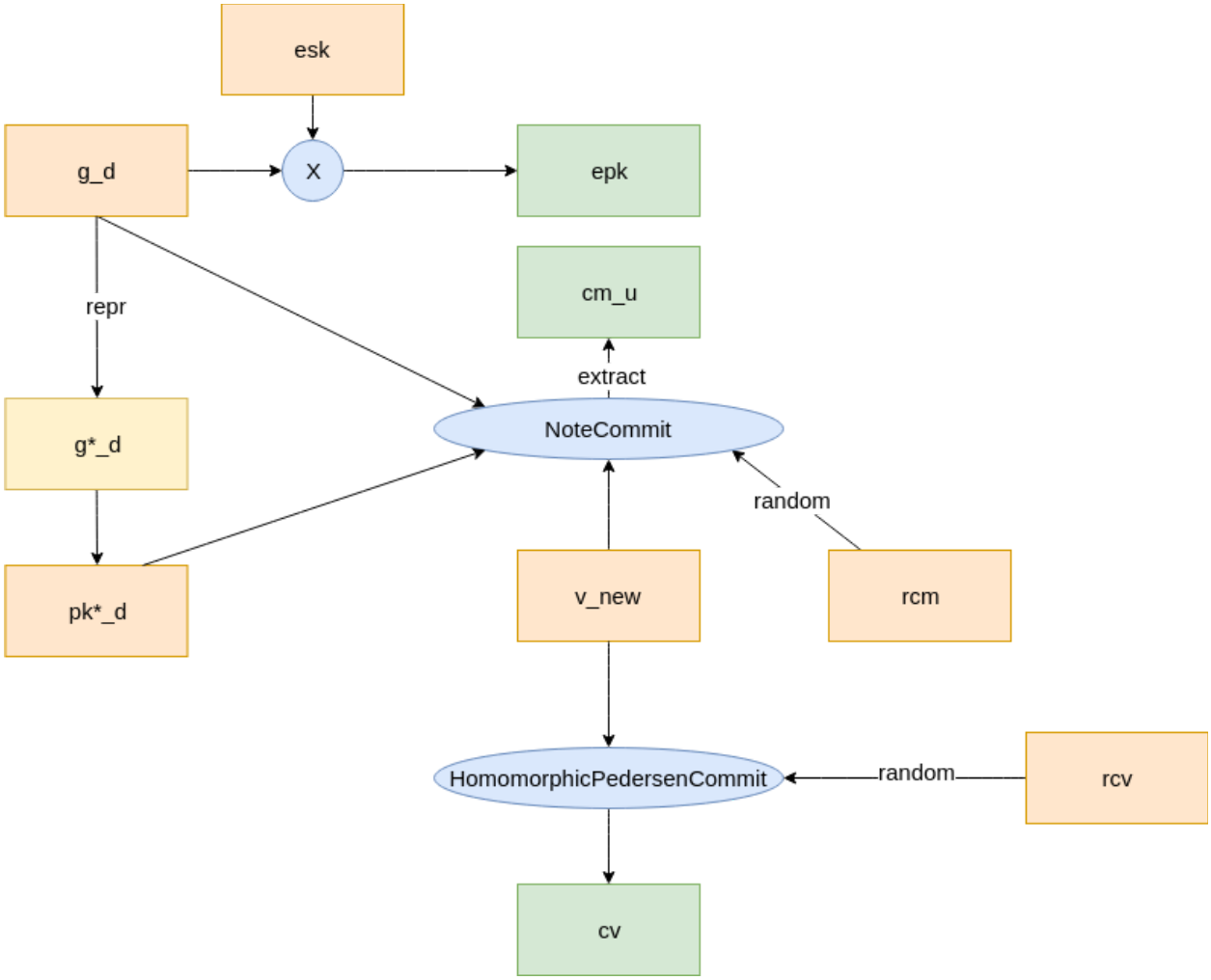
SNARK protocol overview

- **Description:** the protocol built for sapling increases the complexity of the zcash system and mapping out the different components and their relations is important for the understanding of the functionality.
- **Reference:** https://drive.google.com/file/d/1-F0gGwcjT-GxEsL2LxSxdWVVU4ASxr2A/view?usp=sharing
- **Color code:** green - public, orange - private, blue - operation
- **Transaction structure:**

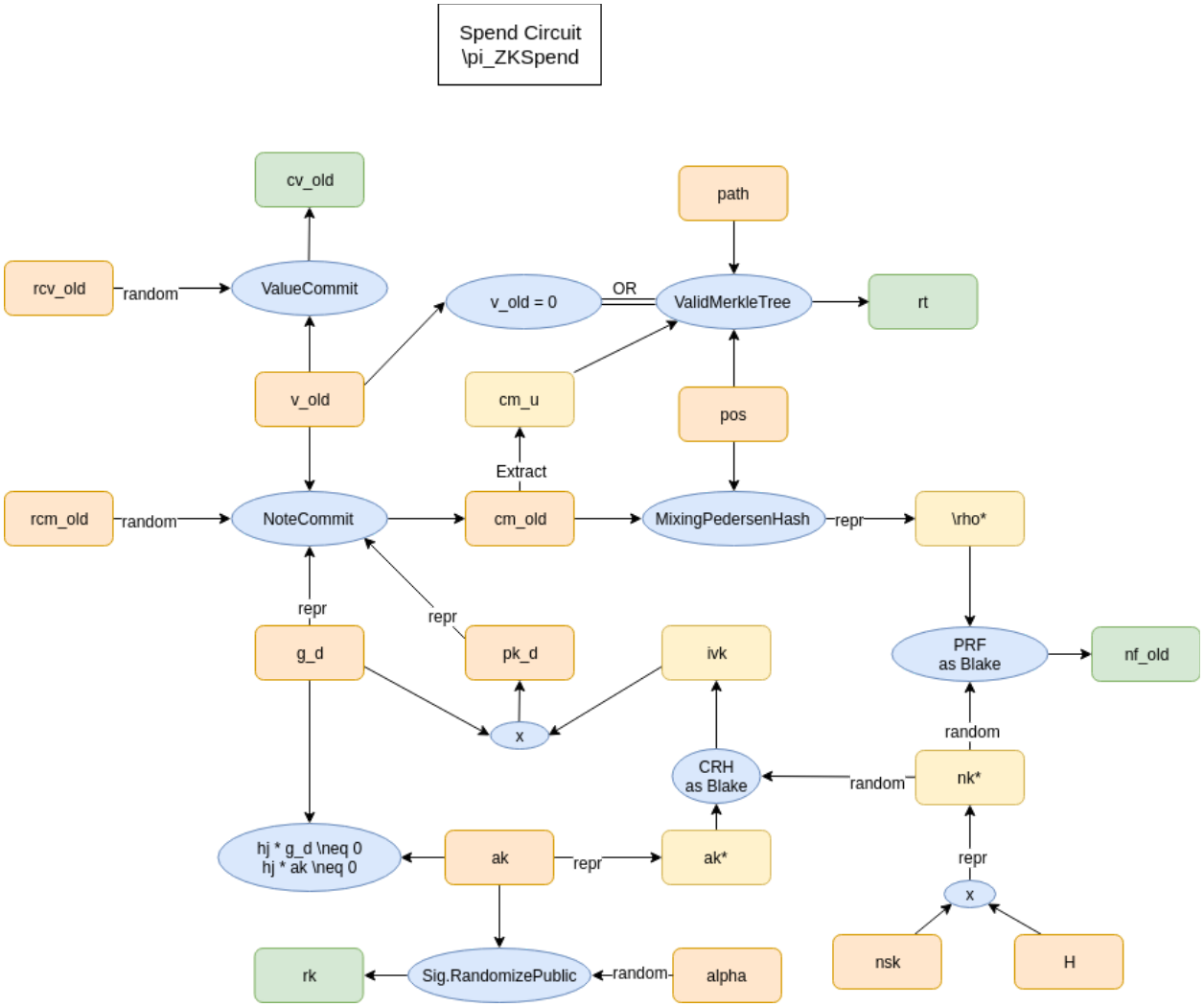
Subset of a Sapling transaction - only shielded Sapling parts:



- **Output circuit:**



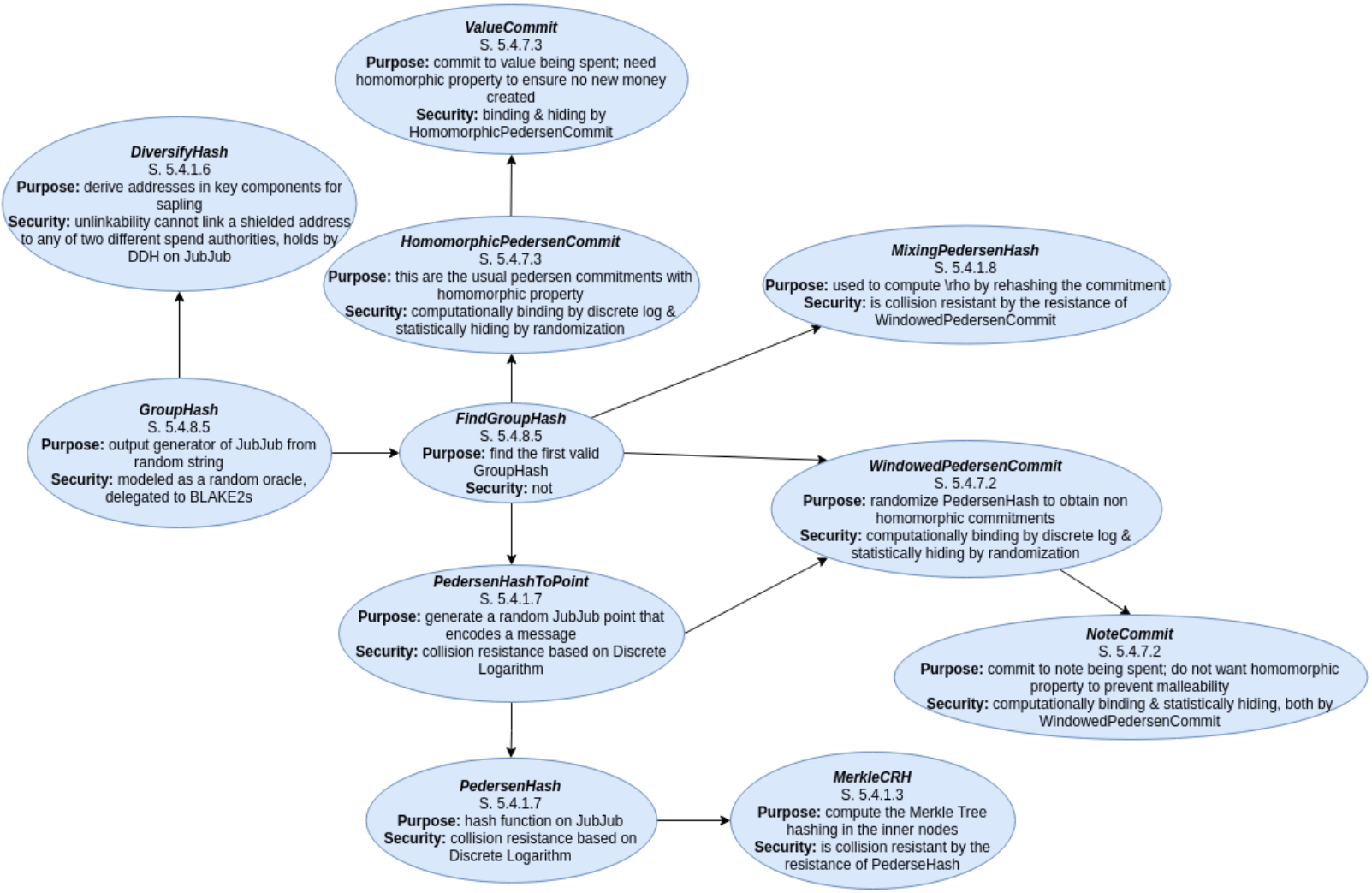
- **Spend circuit:**



- **Commitments and hashes:**

Review comments in docs:

- Test vectors: https://docs.google.com/document/d/1e4I0DSq-FzmhdeRv3JU7GE1cSx3jClBQGQG-zDnmrmU/edit#bookmark=id.o5cfosb09mwo
- Pedersen Hash deeper analysis: https://docs.google.com/document/d/1e4I0DSq-FzmhdeRv3JU7GE1cSx3jClBQGQG-zDnmrmU/edit#bookmark=id.x



Protocol Review

1. Correct usage of value - used three times in the Spend circuit
    - As part of the note shown to exist in the tree.
    - To decide whether the note has to exist (value != 0).
    - A Pedersen commitment is publicly exposed.
2. Notes and nullifiers relationship
    - **Description:** This relates to how the nullifier correctly shares some bits between the notes and the nullifier:
        - The note commitment includes g_d and pk_d = g_d^ivk.
        - ivk is a CRH of ivk_preimage.
        - ivk_preimage has 256 bits in common with nf_preimage.
        - The nullifier bits are a CRH of nf_preimage.
        - The nullifier bits are packed in public inputs.
3. Nullifier position
    - **Description:** Used to generate rho, in order to generate the nullifier. Correct process was observed:
        i. Bitness
        ii. multiply an independent base
    - **Note:** This could be a helper function.
4. Value_num packing
    - **Description:** In sapling/mod.rs, note value_bits are correctly packed into value_num as a sum of bits times powers of two.
5. Commitment_randomness
    - **Description**: Used in the WindowedHomomorphicCommitment, correct process was observed:
        - commitment_randomness bitness verification
        - unpacking
        - multiply by an independent base
6. Expose_value_commitment
    - **Description:** Takes a value, and exposes a HomomorphicPedersenCommitment as a public input.
    - **Constraints:**
        - Public input equals Pedersen commitment.
        - Edwards addition.
        - Multiplication of base point ValueCommitmentValue * value.
        - Multiplication of base point ValueCommitmentRandomness * randomness.

- Bitness of input value.
- Bitness of input randomness.
- The ValueCommitmentValue and ValueCommitmentRandomness points are different and of unknown relationship.

Assert_not_small_order

Description: This function ensures that a point is not part of the small subgroup of order 8, by doubling the point 3 times. As noted in the spec, this only excludes the small subgroup, but not the combinations (a point of the prime subgroup + a point of the small subgroup).
Usages: The circuit checks a few points to make sure they are indeed on-curve and are not of small order. This happens for:

- Spend / proof_generation_key.ak
- Spend / payment_address.diversifier.g_d
- Output / payment_address.diversifier.g_d

Test reference: https://github.com/QED-it/sapling-crypto-internal/tree/assert_not_small_order
Notes:

- It's possible that different g_d with the same ivk yield the same pk_d (similarly with esk and epk). There are no obvious issue with this because g_d is part of the note commitment.

Recommendation:

- We should require that points from the prover be in the prime subgroup anyway.
    - No additional circuit cost.
    - Currently the circuit checks P * 8 != 0. Instead, the prover should provide (P/8) and the circuit calculates P = (P/8) * 8,and still checks P != 0.

Elliptic Curve aspects

1. Affine-Edwards arithmetic

a. **Description:** Checked the 6 constraints and witness calculations. Renamed variables to match the spec.

2. Montgomery into_edwards conversion

a. **Description:** The code lives in into_edwards in src/circuit/ecc.rs, and essentially performs these two operations:
u = (scale*x)/y
v = (x-1) / (x+1)

We have reviewed the two constraints related to the conversion, and had no special comments.

3. Montgomery addition of PH chunks
    a. **Description:** Matched the spec to the code, renaming variables when needed. In circuit/pedersen_hash.rs, constraints "addition of segment {}, window {}".
    P1 is `tmp` and P2 is `segment_mont`.
    P1 is computed with increasing `[2^4j].base`, so k1 > k2 in absolute values.
    Checked the 3 constraints and witness calculations.
4. Negation in Montgomery and Edwards
    a. **Description:** Negation in Montgomery form is done by negating the y coordinate and in Edwards form by negating the x coordinate. Correct process was observed.
5. EdwardsPoint.repr()
    a. **Description:** This gets a representation of an EdwardsPoint out of an (x,y) tuple, returning y || x[0]. It uses correctly the into_bits_le_strict method.
    b. **Note:** While only the first bit in x is needed, a bunch of constraints are used with into_bits_le_strict for x.
6. Using types to mark which representation (Edwards vs Montgomery)
    a. **Description:** The circuit uses distinct Rust types in order to differentiate between points in different representation, in which the type system is the main factor in ensuring the user uses the correct representation. In summary, the usage looks fine and upholds the intended use.
7. Constraints that a point is on Jubjub in EdwardsPoint::*witness()*
    a. **Description:** We have reviewed the constraints relating to a point being on Jubjub in the Edwards form. We note that there's a small deviation from the spec: 4 constraints instead of the 3 in the spec.
    Spec A.3.3.1:
    *(d·uu) * (vv) = (-uu + vv − 1)*
    Implementation in EdwardsPoint.interpret:
    *(uu) * (vv) = (uuvv)*
    *(-uu + vv) * (1) = (1 + d.uuvv)*
    In addition to the square constraints:
    *(u) * (u) = (uu)*
    *(v) * (v) = (vv)*