Code Review and Security Assessment
For
Lelantus Spark
Initial Delivery: December 9, 2022
Final Delivery: December 19, 2022

Prepared For
Reuben Yap | Firo
Levon Petrosyan | Firo
Aaron Feickert | Cypher Stack
Aram Jivanyan| Firo
Peter Shugalev | Firo

Prepared by
Mikerah Quintyne-Collins | HashCloak Inc
Manish Kumar | | HashCloak Inc
Onur Inanc Dogruyol | HashCloak Inc

Table Of Contents

Executive Summary

The Firo community engaged HashCloak Inc for an audit of the implementation of the Lelantus Spark protocol which is Firo next generation privacy protocol. Lelantus Spark provides improvements to the original Lelantus construction and more user-friendly features. The audit was done from October 24, 2022 to November 28, 2022. We were provided the commit 88b499355c8781fad279ba3171ca8938bf0907ae and minimal changes were done as we were undergoing the audit which didn't affect the scope of the audit. The audit scope were the files included in the `libspark` and `spark` folders.

A review of the codebase was done once the Firo development team went over the report. PR #1218 and commit 60dba45a8aee17cae0bc24d12d4071b39583781b were reviewed for the fixes we suggested. Further, several of the findings were reassessed after clarifications by the Firo development team.

Over the course of the audit, we were mainly concerned with finding issues that may lead to the following:

- Double spends
- Minting new coins for free
- Deanonymizing transactions
- Leaking transaction amounts
- Leaking of spending keys
- Unauthorized spends are not possible. An authorized spend means the user has the spend key
- It is not feasible to link transactions to each other or to (diversified) addresses
- It is not feasible to link diversified addresses

Overall, we found issues ranging from medium to informational. We find that the code within scope followed the paper very closely in its description of the protocol. In particular, for cryptographic constructions, the implementations followed very closely to its related paper's description. This exposition leads to ease of maintainability and auditability of the codebase.

Severity	Number of Findings
Critical	0

High	0
Medium	1
Low	5
Informational	2

Overview

Lelantus Spark is Firo's next generation privacy protocol. It includes various improvements to the Lelantus construction such as recipient privacy and more reasoned security guarantees. Further, it includes several usability features such as delegated proving, selective disclosures and multisignatures. The cryptographic primitives and constructions used in the implementation of the Lelantus Spark are well understood and rely on common cryptographic assumptions. Further, they are in use in comparable privacy coins such as Monero.

Scope

The scope for this audit was the following:

- All the files within `src/libspark`
- All the files within `src/spark`

If there were any direct dependencies within the Firo codebase that were used within the above two folders that we believed impacted the security of the audit, we included those minimally within the scope of the audit. In particular, we also considered the following:

- `src/secp256k1`
- `src/crypto`

as needed to better understand and assess the security of the code in scope.

Findings

FLS-01: Memory allocation missing for A_scalars

Type: Low
Files affected: src/libspark/bpplus.cpp

Description: In `BPPlus::prove`, in line 139 & 140, `A_points.reserve(2*N*M + 1);` is declared twice for `A_points` but there is no memory allocation for `A_scalars`.

Impact: Repeated reallocation of memory may take place for `A_scalars` which can affect the efficiency and also may result in memory leak. The memory leak is not very likely because deallocation in vector happens automatically once the vector goes out of scope.

Suggestion: line 140 must be replaced with `A_scalars.reserve(2*N*M + 1);`

Status: After discussions with the development team, this finding's rating score was lowered from Medium to Low. This typo has been fixed at the commit [5dd4cc7154bd0e139b2e4d76da9529f4643a5715](5dd4cc7154bd0e139b2e4d76da9529f4643a5715).

FLS-02: Missing checks if challenges are equal to 0 in various places

Type: Medium
Files affected: src/libspark/bpplus.cpp

Description: In various places, after the transcript is updated, challenges are computed by hashing the transcript and relevant proof elements. However, the challenges are not checked if they are equal to 0 in Scalar.

Impact: If a challenge is equal to 0, then a malicious prover can invert 0, breaking the soundness of the protocol. It would effectively be reduced to mounting a hash preimage attack.

Suggestion: This can be fixed by a check and re-doing computation if the check fails. Write test cases for the ZERO in line 7

```
const Scalar ZERO = Scalar((uint64_t) 0);
```

```
// In Line 155, 387
Scalar y = transcript.challenge("y");

if(y == ZERO) {
  MINFO("y is 0, try again");
  goto try_again;
}

// In Line 156, 395
Scalar z = transcript.challenge("z");

if(z == ZERO) {
  MINFO("z is 0, try again");
  goto try_again;
}

// In Line 255, 403
Scalar e = transcript.challenge("e");

if(e == ZERO){
  MINFO("e is 0, try again");
  goto try_again;
}

// In Line 286, 409
Scalar e1 = transcript.challenge("e");

if(e1 == ZERO){
  MINFO("e is 0, try again");
  goto try_again;
}
```

Status: These zero checks have been added only to the `BPPlus::verify`. It is not added to the `BPPlus::prove` because in order for an attacker to enforce this condition from the perspective of the prover, they would simply ignore the checks altogether and be able to invert 0. Hence, it suffices to simply add these checks to the

verifier to ensure that the verifier catches a malicious prover. The check has been added at the commit 5dd4cc7154bd0e139b2e4d76da9529f4643a5715.

FLS-03: Check missing for double pedersen vector commitment

Type: Low
Files affected: src/libspark/grootle.cpp

Description: In the function `vector_commit` in line 65, the validity of the double pedersen commitment statement is not checked i.e., whether size of vectors `Gi` equals to `a` and `Hi` equals to `b`.

Impact: Passing an incorrect argument to `vector_commit` can lead to segmentation fault.

Suggestion: There should be a check for the pedersen statement.

```
if (Gi.size() != a.size() || Hi.size() != b.size()) {
     Throw std::invalid_argument(" Bad pedersen statement");
}
```

Status: This check has been implemented at the commit 5dd4cc7154bd0e139b2e4d76da9529f4643a5715

FLS-04: Using secret data for loop bounds

Type: Low
Files affected: src/libspark/keys.cpp

Description: In keys.cpp within `GetHex()`, the loop on line 187 is directly dependent on `buffer`. `buffer` is not directly dependent on secret data but does depend on `Q1` and `Q2` on lines 179 and 180 respectively.
Impact: Loops with a bound derived from a secret value directly expose a program to timing attacks

Suggestion:

```
// In Line 187, instead of using the following

for (const auto b : buffer) {
    ss << (b >> 4);
    ss << (b & 0xF);
}

// First use the following, then use the loop

size_t buffer_size = buffer.size();
```

Status: The loop bound bounds were not dependent on the secret data and hence this change is not made. But, the `GetHex()` and `SetHex()` were outdated and hence both those functions were removed at the commit [60dba45a8aee17cae0bc24d12d4071b39583781b](60dba45a8aee17cae0bc24d12d4071b39583781b).

FLS-05: Shift values need to be typecasted in order to minimize undefined behavior

Type: Low
Files affected: src/libspark/bech32.cpp

Description: If it's not typecasted, the bitwise left-shift operator may be undefined and may cause the runtime error.

Impact: It may cause a runtime error.

Suggestion: In Line 116

```
uint8_t c0 = (uint8_t) c >> 25;
```

Status: After discussion with the development team, it was found that this change breaks the beach32 capability to detect certain address malleations. Therefore, this change was not made.

FLS-06: Randomize( ) function should be able to return zero

Type: Low
Files affected: secp256k1/src/Scalar.cpp

Description: The `randomize()` function used in src/libspark and src/spark for generating any random scalar does not generate the `zero` scalar. In particular, the line 220 in Scalar.cpp is incorrect.

Impact: Excluding zero in `randomize()` leads to a biased randomize function where distribution of all the numbers might not be equally likely.

Suggestion: The line 220 should be replaced by

```
while(!this->isMember());
```

Status: This issue is fixed at the commit [60dba45a8aee17cae0bc24d12d4071b39583781b]

FLS-07: Optimization in Chaum verification

Type: Informational
Files affected: src/libspark/chaum.cpp

Description: In line 141, optimization can be done in the aggregation of A2 by adding the proof.A2[i] first and then calling points.emplace_back on the sum. In this way, n-1 scalar space and n-1 points space can be saved.

Suggestion:

```
GroupElement A2_sum;
A2_sum = proof.A2[0];
for(std::size_t i = 1; i < n; i++) {
```

```
    A2_sum += proof.A2[i];
}
scalars.emplace_back(w);
points.emplace_back(A2_sum);
```

Status: This suggestion was useful for the efficiency and hence is added at the commit [5dd4cc7154bd0e139b2e4d76da9529f4643a5715]

FLS-08: Results of the curve/group operations needs to be double checked to be valid

Type: Informational
Files affected: src/libspark/bpplus.cpp, src/libspark/chaum.cpp, src/libspark/grootle.cpp, src/libspark/schnorr.cpp

Description: In various places, curve/group operations are used. Results of these operations need to be double checked to be sure they are valid.

Impact:

Suggestion: In the specified files above, the results needs to be checked. The following functions are need to be inserted after curve/group operations.

```
// In src/secp256k1/include/GroupElement.h, the following functions
// can be used

isMember();
isInfinity();

// In src/secp256k1/include/Scalar.h, the following function  // can
be used

isMember();
```

In src/libspark/bpplus.cpp

```
// In Line 151

proof.A = A_multiexp.get_multiple();

// In Line 248, 249

L_ = L_multiexp.get_multiple();
R_ = R_multiexp.get_mutliple();

// In Line 517

Return multiexp.get_multiple().isInfinity();

// In Line 260, 261

Gi1[i] = Gi1[i] * e_inverse + Gi1[i+N1] * (e*y_N1_inverse);
Hi1[i] = Hi1[i] * e + Hi1[i+N1]*e_inverse;

// In Line 281, 282

proof.A1 = Gi1[0]*r_ + Hi1[0]*s_ + G*(r_*y*b1[0] + s_*y*a1[0]) +
H*d_;

proof.B = G*(r_*y*s_) + H*eta_;


In Line 288, 289, 290

proof.r1 = r_ + a1[0]*e1;
proof.s1 = s_ + b1[0]*e1;
proof.d1 = eta_ + d_*e1 + alpha*e1.square();
```

In src/libspark/chaum.cpp

```
// In Line 62
```

```
proof.A1 = H*t;

// In Line 65, 66

proof.A1 += F*r[i] + G*s[i];
proof.A2[i] = T[i]*r[i] + G*s[i];

// In Line 72, 75, 76, 77, 78

proof.t3 = t;
proof.t1[i] = r[i] + c_power*x[i];
proof.t2 += s[i] + c_power*y[i];
proof.t3 += c_power*z[i];
c_power *= c;
```

In src/libspark/schnorr.cpp

```
// In Line 43

proof.A = G*r;

// In Line 50, 51

proof.t += y[i].negate()*c_power;
c_power *= c;

// In Line 78
c_power *= c;
```

Status: After discussing with the development team, it was concluded that these closure checks should be done at the curve/scalar wrapper level and not in the `libspark`. Therefore, a separate closure check in the `libspark` is not required.