SmartDec

Beam Security Analysis

This report is public.

Published: June 23, 2020

Abstract

In this report, we consider the implementation of Lelantus protocol for Beam blockchain project. Our task is to check if the implementation of the protocol conforms to the specification and if the implementation is secure.

The security of the protocol itself is out of the audit scope.

Disclaimer

The audit does not give any warranties on the security of the code. One audit cannot be considered enough. We always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of the code. Besides, security audit is not an investment advice.

Summary

In this report, we considered the implementation of Lelantus protocol. We performed our audit according to the procedure described below.

The audit showed that the implementation of the protocol conforms to the specification.

Also, several issues of low severity were found in the code. None of them endanger the project's security.

The developer provided the comments for these issues as well as for some details of the implementation. We placed them in the report.

General recommendations

The low severity issues found in the report do not endanger the project's security. However, we recommend fixing them to avoid problems in the future versions of code.

Procedure

In our audit, we consider the following crucial features of the code:

1. Whether the implementation of the protocol conforms to the specification.
2. Whether the code is secure.
3. Whether the code meets best coding practices.

We perform our audit according to the following procedure:

- automated analysis
    - we scan project's code base with SmartDec Scanner
    - we manually verify (reject or confirm) all the issues found by tools
    - we run tests and check their coverage
- manual audit
    - we inspect the code and revert the initial algorithms of the protocol and then compare them with the specification
    - we manually analyze the code for security vulnerabilities
    - we assess overall project structure and quality
- report
    - we reflect all the gathered information in the report

Project overview

Project description

In our analysis, we consider Lelantus protocol specification and Beam project's code on Git repository, commit 33334578bb879044281b83c88ac09de142211fe8.

Project architecture

For the audit, we were provided with a git repository. The project has tests and specification.

The scope of the audit included:

- **lelantus.cpp/lelantus.h** (complete)
- **shield.cpp** (complete)
- **ecc_native.h, ecc.h** (partial)
- **ecc.cpp** (partial)
  - `void MultiMac::Calculate(Point::Native& res) const` 1435
  - `void SignatureBase::SignRaw(const Config& cfg, const Hash::Value& msg, Scalar* pK, const Scalar::Native* pSk, Scalar::Native* pRes) const` 2343
  - `void SignatureBase::Sign(const Config& cfg, const Hash::Value& msg, Scalar* pK, const Scalar::Native* pSk, Scalar::Native* pRes)` 2336
  - `void SignatureBase::CreateNonces(const Config& cfg, const Hash::Value& msg, const Scalar::Native* pSk, Scalar::Native* pRes)` 2314
  - `void SignatureBase::SetNoncePub(const Config& cfg, const Scalar::Native* pNonce)` 2304
- **eccbulletproof.cpp** (partial)
  - `void InnerProduct::BatchContext::AddCasual(const Point::Native& pt, const Scalar::Native& k, bool bPremultiplied /* = false */)` 68
  - `void InnerProduct::BatchContext::AddPrepared(uint32_t i, const Scalar::Native& k)` 88
  - `void InnerProduct::BatchContext::AddPreparedM(uint32_t i, const Scalar::Native& k)` 93

Automated analysis

The code base was scanned by a program static analysis tool by specifying the URL.

The tool analyzed 1 354 659 lines of code and detected 8 critical and 616 medium level vulnerabilities. All of them were either false positives, or referred the test code, or 3rd-party libraries.

The generated report is not included here as it contains no findings.

Test coverage analysis

All operations were performed on the commit `33334578bb879044281b83c88ac09de142211fe8`.

To compute test coverage, compilation flags `-fprofile-arcs -ftest-coverage` were added to the root `CMakeLists.txt` file, as well as the linker flag `-lgcov`. Then the project was compiled with the Debug profile, after which the tests were run using `make test`.

During compilation, `.gcno` files with information about the blocks and the structure of the source code were automatically generated. During tests execution, `.gcda` files with information about the actual execution of specific lines, blocks and functions were automatically generated.

At the end, `gcov` was called on the list of all the `.gcno` and `.gcda` files. The output of `gcov` was converted using regular expressions to a csv table with information about the amount of code covered by tests in the Beam sources. In addition to the summary data, we get the test coverage stats for each line in source code files

Full sequence of commands for obtaining test coverage data:

```
git checkout 3333457
vim CMakeLists.txt # Changing the CMakeLists.txt

git diff CMakeLists.txt
# diff --git a/CMakeLists.txt b/CMakeLists.txt
# index 72d1523..8b300c8 100644
# --- a/CMakeLists.txt
# +++ b/CMakeLists.txt
# @@ -280,6 +280,8 @@ else()
#      set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -Wno-unused-const-variable")
# so what?
#      set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -Wno-unused-function") #
mostly in 3rd-party libs
#      set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -Wno-unused-value") #
proto.h
# +    set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -fprofile-arcs -ftest-
coverage")
# +    set(CMAKE_EXE_LINKER_FLAGS "${CMAKE_EXE_LINKER_FLAGS} -lgcov")
```

```
#   endif()
#
#   if(BEAM_HW_WALLET)

cmake .
make -j4
make test

mkdir ../cov-res
fd -aiI '.*\.gcno' > ../cov-res/gcno-list.txt
fd -aiI '.*\.gcda' > ../cov-res/gcda-list.txt
cd ../cov-res
cat gcno-list.txt gcda-list.txt | \
    xargs gcov -rmps /home/morgan/SmartDec/MWC/beam-cov > gcov-pct.txt

vim gcov-pct.txt # Converting from gcov output format to .csv
```

Coverage Stats

File type	Total executable LoC	Coverage
All files	62598	72.10%
All except 3rdparty	50544	79.74%
All except 3rdparty and tests	35524	76.25%
Lelantus and lelantus tests	1529	81.88%
Lelantus only	916	94.10%

Conclusion

Tests coverage is sufficient.

Manual analysis of implementation

Here we check whether the implementation of the Lelantus protocol conforms to the specification.

Lelantus protocol implementation analysis

This section contains a detailed review of the implementation of Lelantus in the source code. Here, with red color we denoted the lines that seem to deviate from the specification.

Algorithm outline

Taken from https://github.com/BeamMW/beam/wiki/Beam-signature-schemes

oracle <-- Sigma parameters (n,M)
oracle <-- Commitment
oracle <-- SpendPk
oracle <-- N’ (public nonce of the Schnorr’s multi-signature)
oracle --> Challenge for Commitment
oracle --> Challenge for SpendPk
<-- Schnorr’s multi-signature: kG, kH
oracle <-- Sigma protocol part 1 (A, B, C, D, G-vector) lines 758 - 763 in `lelantus.cpp` count these values and the analysis will be below.
oracle --> Challenge for Sigma protocol - line 702, file `lelantus.cpp`
<-- Sigma protocol part 2 (a, c, r, f-vector) - lines 704 - 739, file `lelantus.cpp`

Lelantus analysis

Comparison of symbols

- $m\_Witness.V.m\_SpendSk$ – private key (k_{mw} in the documentation). It is also the secret key for generating $SpendPk$.

- $m\_Witness.V.m\_R\_Adj, m\_Witness.V.m\_R\_Output$ – these are two private keys that form k_{out}. The difference between them is that the H^* generator is hidden for confidential assets in the following way $H' = k1 \cdot G + H^*$. This means that $C_{out} = k_{out} \cdot G + v \cdot H' = k_{out} \cdot G + v \cdot (k1 \cdot G + H^*) = (k_{out} + v \cdot k1) \cdot G + v \cdot H^*$.
  In other words, if H^* is hidden, i.e. $phGen$ is set (in the source), then $m\_Witness.V.m\_R\_Adj =$ (k_{out} is used as the secret key for the G generator $+ v \cdot k1$), otherwise $m\_Witness.V.m\_R\_Output = k_{out}$. But the most important thing is that both keys will be called sk later in the code, meaning that this is the key that is multiplied by the G generator in the c_{out} commitment.

- $m\_Proof.m\_SpendPk = G \cdot k_{mw}$. In other words, it is generated from the sender's secret key to generate a serial number.

- $phGen$ – this $H' = k1 \cdot G + v \cdot H^*$ or this H^*.

- `Scalar::Native kSer` – serial number, calculated using $m\_Proof.m\_SpendPk$ and oracle.

- $ptBias = C_{out} + J \cdot s = G \cdot sk + H^* \cdot v + J \cdot s = G \cdot (k_{out} + v \cdot k1) + H^* \cdot v + J \cdot s$.

- $m\_Witness.V.m\_V$ – hidden value v.

- $m\_Proof.m\_Commitment$ – this is C_{out}.

- $m\_Witness.V.m\_R$ – this is $(k_s + k_{mw})$.

- $m\_NoncePub$ – this is $R = G \cdot sk_{rand} + H' \cdot v_{rand}$.

Code review

Reference: Schnor's signature [1].

1. `bool Proof::IsValid(InnerProduct::BatchContext& bc, Oracle& oracle, Scalar::Native* pKs, const Point::Native* pHGen) const`
   core/lelantus.cpp, lines 789-842

   To check the signature, check the equality of $g^u h^v = RC^x$.

In our case, it should be like this:

$G \cdot m\_Signature.m\_pK[0] + H' \cdot m\_Signature.m\_pK[1] + R + Cout \cdot x_1 + SpendPk \cdot x_2 == 0$, where $R = [R = m\_Signature.m\_NoncePub] = G \cdot sk_{rand} + H' \cdot v_{rand}$, $C_{out} = sk \cdot G + v \cdot H'$, $m\_Signature.m\_pK[0] = -sk_{rand} - sk \cdot x_1 - SpendSk \cdot x_2$, $m\_Signature.m\_pK[1] = -v_{rand} - v \cdot x_1$, where x_1 and x_2 are challenges.

`818` – adds $SpendPk \cdot x_2$.

`820` – adds $G \cdot m\_Signature.m\_pK[0]$.

`822-826` – adds $H' \cdot m\_Signature.m\_pK[1]$.

`827` – adds R.

`831-834` – Sigma verification.

`835` – adds $C_{out} \cdot (kBias + x_1) \cdot \sum_{i=0}^{N-1} c_i \cdot \prod_{j=0}^{M-1} f_{j,i_j} = \sum_{i=0}^{N-1}(C_i - bias) \cdot \prod_{j=0}^{M-1} f_{j,i_j} = \sum_{i=0}^{N-1} C_i \cdot \prod_{j=0}^{M-1} f_{j,i_j} - bias \cdot kBias = \sum_{i=0}^{N-1} C_i \cdot \prod_{j=0}^{M-1} f_{j,i_j} - (C_{out} + s \cdot J) \cdot kBias$.

`837-840` – adds $s \cdot kBias \cdot J$.

> ***Comment from the developers:*** *For the verification of the Schnorr's generalized signature only $C_{out} \cdot x_1$ must be added. But those code lines actually add 2 terms into the equation. We can rewrite them as: $sum += C_{out} \cdot x_1$ and $sum += kBias * (C_{out} + s * J)$.*
>
> *So, the 1st term is related to Schnorr's signature, whereas the 2nd term is related to the Sigma proof. The expression $(C_{out} + s * J)$ is the so-called "bias". It should be subtracted from all the commitments in the commitment list before the Sigma protocol is applied. So, instead of subtracting this from each commitment in the list (which is very ineffective), we subtract it only once with appropriate coefficient. This coefficient $kBias$ is returned from the `Sigma::Proof::IsValid()` funciton, and is equal to the negated sum of all the used commitments with appropriate coefficients.*
>
> *So, we use the above expressions for optimization. We use C_{out} only once, with the coefficient that accounts for both Schnorr's signature and the "bias". And the multiplier for J is accumulated when multiple proofs are verified at once.*

2. `void Prover::Generate(const uintBig& seed, Oracle& oracle, const Point::Native* pHGen)` core/lelantus.cpp, lines 844-890

   `846-848` – generation of the secret key sk.

   $ptBias$ – this is $Bias = commitment + s \cdot J = k \cdot G + H' \cdot v + s \cdot J$.

   `850` – adds $pHGen \cdot v$, where $pHGen$ is H' or $H*$.

   `851` – $Commitment = k \cdot G + H' \cdot v$. Commitment in the source is $m\_Proof.m\_Commitment$.

   `852` – Calculation of $SpendPk$. In the source it is $m\_Proof.m\_SpendPk$ and $m\_Witness.V.m\_SpendSk$ – the secret key for $SpendPk$ generating.

   `856-867` – Calculation of parameters for `CreateNonces`.

`868` – `m_Proof.m_Signature.CreateNonces(Context::get().m_Sig.m_CfgGH2, hv, pSk, pRes);` calculates two keys that are written to $pRes$. $pRes[0]$ is associated with sk; $pRes[1]$ is associated with v (in documentation) and $m\_Witnewss.V.m\_V$ (in source).

Notice: `CreateNonces`'s location is `core/ecc.cpp`.

`869-872` – calculation of $m\_Proof.m\_Signature.m\_NoncePub = G \cdot pRes[0] + phGen \cdot pRes[1] = G \cdot sk_{rand} + phGen \cdot v_{rand}$.

`874-875` – the signature `SignRaw` (the case when $phGen$ is set). `m_Proof.m_Signature.SignRaw(Context::get().m_Sig.m_CfgGH2, hv, m_Proof.m_Signature.m_pK, pSk, pRes);`

Notice: `SignRaw` is located in `core/ecc.cpp`, line 343. First, receive
$pRes[0] = sk_{rand}$ (previous value $pRes[0]$) $+ sk \cdot x_1 + SpendSk \cdot x_2$;
$pRes[1] = v_{rand}$ (similarly) $+v \cdot x_1 + zero \cdot x_2$, where x_1 and x_2 are challenges.

In other words, we get the signature $m\_Proof.m\_Signature$, in which the field m_pK consists of two commitments:

$m\_pk[0] = -sk_{rand} - sk \cdot x_1 - SpendSk \cdot x_2$,

$m\_pk[1] = -v_{rand} - v \cdot x_1 - zero \cdot x_2 = -v_{rand} - v \cdot x_1$.

`876-878` – the signature `sign` (the case when $phGen$ is not set and H^* is not hidden). In the output, we get $m\_NoncePub = sk_{rand} \cdot G + v_{rand} \cdot H$.

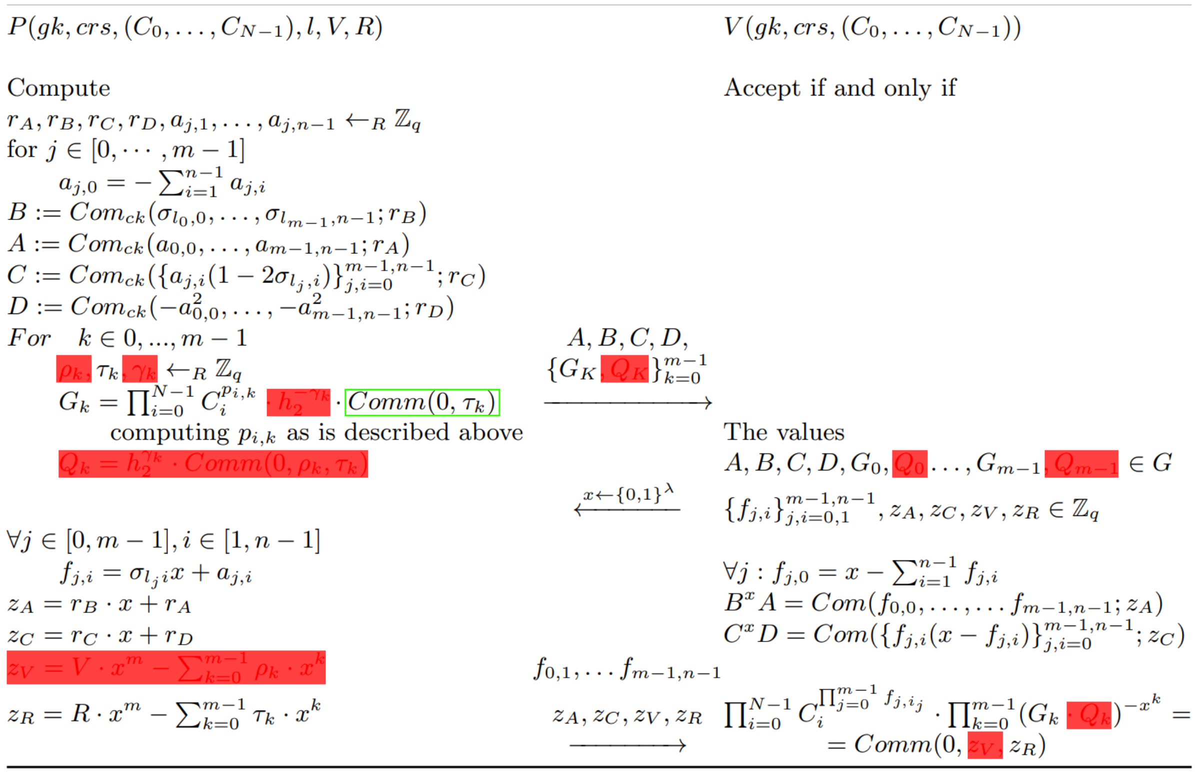
`879-881` – In these lines, the serial number is calculated ($char$ in the source, $\boldsymbol{s}$ in the documentation) and $J \cdot s$ is added to $Bios$.

`882-890` – forming a proof for Sigma.

Sigma implementation review

The verification of this Protocol was based on article [2].

The image below is necessary for a better understanding of the scheme and for comparing variables from the code with the notation from the article.



$P(gk, crs, (C_0, \dots, C_{N-1}), l, V, R)$

Compute

$r_A, r_B, r_C, r_D, a_{j,1}, \dots, a_{j,n-1} \leftarrow_R \mathbb{Z}_q$

for $j \in [0, \cdots, m-1]$

$a_{j,0} = -\sum_{i=1}^{n-1} a_{j,i}$

$B := Com_{ck}(\sigma_{l_0,0}, \dots, \sigma_{l_{m-1},n-1}; r_B)$

$A := Com_{ck}(a_{0,0}, \dots, a_{m-1,n-1}; r_A)$

$C := Com_{ck}(\{a_{j,i}(1 - 2\sigma_{l_j,i})\}_{j,i=0}^{m-1,n-1}; r_C)$

$D := Com_{ck}(-a_{0,0}^2, \dots, -a_{m-1,n-1}^2; r_D)$

$For \quad k \in 0, ..., m-1$

$\rho_k, \tau_k, \gamma_k \leftarrow_R \mathbb{Z}_q$

$G_k = \prod_{i=0}^{N-1} C_i^{p_{i,k}} \cdot h_2^{-\gamma_k} \cdot \boxed{Comm(0, \tau_k)}$

computing $p_{i,k}$ as is described above

$Q_k = h_2^{\gamma_k} \cdot Comm(0, \rho_k, \tau_k)$

$\xrightarrow{A, B, C, D, \{G_K, Q_K\}_{k=0}^{m-1}}$

$\xleftarrow{x \leftarrow \{0,1\}^\lambda}$

$\forall j \in [0, m-1], i \in [1, n-1]$

$f_{j,i} = \sigma_{l_j i} x + a_{j,i}$

$z_A = r_B \cdot x + r_A$

$z_C = r_C \cdot x + r_D$

$z_V = V \cdot x^m - \sum_{k=0}^{m-1} \rho_k \cdot x^k$

$z_R = R \cdot x^m - \sum_{k=0}^{m-1} \tau_k \cdot x^k$

$f_{0,1}, \dots f_{m-1,n-1}$

$\xrightarrow{z_A, z_C, z_V, z_R}$

$V(gk, crs, (C_0, \dots, C_{N-1}))$

Accept if and only if

The values

$A, B, C, D, G_0, Q_0 \dots, G_{m-1}, Q_{m-1} \in G$

$\{f_{j,i}\}_{j,i=0,1}^{m-1,n-1}, z_A, z_C, z_V, z_R \in \mathbb{Z}_q$

$\forall j : f_{j,0} = x - \sum_{i=1}^{n-1} f_{j,i}$

$B^x A = Com(f_{0,0}, \dots, \dots f_{m-1,n-1}; z_A)$

$C^x D = Com(\{f_{j,i}(x - f_{j,i})\}_{j,i=0}^{m-1,n-1}; z_C)$

$\prod_{i=0}^{N-1} C_i^{\prod_{j=0}^{m-1} f_{j,i_j}} \cdot \prod_{k=0}^{m-1} (G_k \cdot Q_k)^{-x^k} =$

$= Comm(0, z_V, z_R)$

Fig. 2. Σ-protocol for double-blinded commitment to 0 in list $C_0, \dots, C_{N-1}$

> ***Comment from the developers:*** *It is based on the transcript from Aram's Lelantus paper, but simplified because we don't prove balance, after subtraction of the bias the being-spent element must consist of the blinding factor only. We united the* G_k *and* Q_k, *and removed original* z_V. *By red I marked what we removed, and the green frame - this is what moved into* G_k *from* Q_k.

Brief explanation of the proof

In this case, there are N commits: $C_0, \dots C_{N-1}$, one of them of the form $Comm(0; r)$. This commit is under the number l. we need to prove that we know r without revealing (l, r).

For the proof, the commits A, B, C, D are formed, where $l_j \in \{0,1\}$, that is, l_j is the j^th^ bit of l.
In addition to these four commits, G_k, where $k = 0, \dots N-1$ are considered.
After getting a random x, Prover counts z_R, z_A, z_C and for each $j = 0, \dots M-1$, $i = 1, \dots n-1$ counts $f_{j,i}$.

Notice: In this case, $Comm(a, b) = a \cdot G + b \cdot H$.

Following the note, we get that $G_k = \sum_{i=0}^{N-1}(C_i p_{i,k} + Comm(0, \tau_k))$.

As a result, you need to check the following expressions for equality:

(The first part of the given notes)

$B \cdot x + A == Comm_{ck}(f, z_A); C \cdot x + D == Comm_{ck}(f(x - f), z_C)$.

(The second part of the given notes)

$\sum_{i=0}^{N-1} C_i \prod_{j=0}^{M-1} f_{j,i_j} + \sum_{k=0}^{M-1} G_k \left(-x^k\right) == Comm(0, z_R)$.

How the second part was obtained:

$$Comm(0, z_R) = 0 \cdot G + \left(rx^M - \sum_{k=0}^{M-1} \tau_k x^k\right) \cdot H = [rx^M \cdot H = Comm(0, r) \cdot x^M = C_l \cdot x^M] =$$

$$= C_l x^M + \left(-\sum_{k=0}^{M-1} \tau_k\, x^k\right) \cdot H = C_l x^M + \sum_{k=0}^{M-1} Comm(0, \tau_k) \cdot (-x^k) =$$

$$= C_l x^M + \sum_{k=0}^{M-1} \sum_{i=0}^{N-1} C_i p_{i,k} x^k + \sum_{k=0}^{M-1} \sum_{i=0}^{N-1} \left(-C_i p_{i,k} x^k\right) + \sum_{k=0}^{M-1} Comm(0, \tau_k) \cdot \left(-x^k\right) =$$

$$= \sum_{i=0}^{N-1} C_l x^M \cdot \delta_{il} + \sum_{k=0}^{M-1} \sum_{i=0}^{N-1} C_i p_{i,k} x^k + \sum_{k=0}^{M-1} \left(\sum_{i=0}^{N-1} \left(C_i p_{i,k} + Comm(0, \tau_k)\right) \cdot \left(-x^k\right)\right) =$$

$$= [\delta_{il} = 1, if(i = l), else\ 0] = \sum_{i=0}^{N-1} \left[C_i x^M \delta_{il} + \sum_{k=0}^{M-1} C_i p_{i,k} x^k\right] + \sum_{k=0}^{M-1} G_k \cdot \left(-x^k\right) =$$

$$= \sum_{i=0}^{N-1} C_i \left(x^M \delta_{il} + \sum_{k=0}^{M-1} p_{i,k} x^k\right) + \sum_{k=0}^{M-1} G_k \cdot \left(-x^k\right) = \sum_{i=0}^{N-1} C_i \cdot \prod_{j=0}^{M-1} f_{j,i_j} + \sum_{k=0}^{M-1} G_k \cdot \left(-x^k\right).$$

Comparison of symbols

m_Tau – these are coefficients τ_k for x^k. They are generated randomly at the very beginning.

mz_R is z_R. r in this case is $(k_s + k_{mw} - k_{out})$.

m_a is the vector of random values a_j.

m_p – these are coefficients $p_{i,k}$.

m_Witness.V.m_L is commitment's number l, in which pair of keys is $(0, r)$.
l_j is j^th bit of number l.

m_vF (size $M(n - 1)$) is the vector f_j, that is, the j^th element is equal to $f_{j,1}$ or $f_{j,0}$ (see [2]).

m_Part1.m_vG (size M) is vector G.

m_A is A.

m_B is B.

m_C – is C.

m_D is D.

Code overview

1. `Sigma::CommitmentStd::FillEquation(MultiMac& mm, const Scalar::Native& blinding, const Scalar::Native* pMultiplier = nullptr)` lines 56-80

   In this function, $mm.m\_pKPrep$ is filled in depending on the commit as follows:
   Commitment mA: filled in with coefficients m_a. At the end, the blinding factor rA is added.

   Commitment mB: filled in with coefficients l_j that are equal to `0` or `1` (`1` if $L \% n$ is equal to j, otherwise `0`). At the end, the blinding factor rB is added.

   Commitment mC: filled in with coefficients $\pm m\_a_j$ (with preceding minus sign if $L \% n$ is not equal to j, otherwise with plus sign). At the end, the blinding factor rC is added. Here, $a_{j,i}(1-2\sigma_{l_j,i})$ part from the article is $\pm m\_a[j*n+i]$ in the code.

   Commitment mD: filled in with coefficients $-m\_a_j^2$. At the end, the blinding factor rD is added.

2. `Calculate(Point& res, MultiMacMy& mm, const Scalar::Native& blinding)` lines 81-89

   m_pKPrep is filled with coefficients (scalars that points will be multiplied by), and then the corresponding commit of the form $(a \cdot A + b \cdot B + \dots)$ is calculated.

3. `bool IsValid(InnerProduct::BatchContext& bc, const Point& ptA, const Point& ptB, const Scalar::Native& x, const Scalar& z)` lines 90-104

   Checks whether $ptA + ptB \cdot x == Commitment(\dots, z)$.

4. `void CmList::Import(MultiMac& mm, uint32_t iPos, uint32_t nCount)` lines 105-119

   It seems that here points are imported to mm, namely points in $m\_pCasual$.

5. `void CmList::Calculate(Point::Native& res, uint32_t iPos, uint32_t nCount, const Scalar::Native* pKs)` lines 120-144

   This method calculates the commitment. The result is the sum of $a \cdot A + b \cdot B + \dots$

6. `bool Proof::IsValid(InnerProduct::BatchContext& bc, Oracle& oracle, const Cfg& cfg, Scalar::Native* pKs, Scalar::Native& kBias) const` lines 205-349

`238-256` – $FillKs$. It is considered $m\_kBias = \sum_{k=0}^{M-1} \sum_{i=0}^{n} \prod_{j=k}^{M-1} f_{j,i}$ and m_pKs, $(k \cdot n + i)$th element of which is $\prod_{j=k}^{M-1} f_{j,i}; i = 0, \dots, n, k = 0, \dots M - 1$.

`259-276` – calculates $f_{j,0}$ using $f_{j,1}$. $\forall j$: $f_{j,0} = x - \sum_{i=1}^{n-1} f_{j,i}$.

`279-295` – checks whether $m\_A + m\_B * x == Commitment(f, m\_zA)$.

`297-320` – checks whether $m\_D + m\_C * x == Commitment(f(x - f), m\_zC)$.

`321-334` – in the second part of the check, there is G_k on the left. In this case, G_k is stored in vector $m\_Part1.m\_vG$. This is where G_k is multiplied by $(-x^k)$.

`335-348` – scalar $m\_Part2.m\_zR$ is added, which will be multiplied by G in the commit. This is the right part of the second check.

Reminder: the right part is equal to $Comm\ (0, m\_zR)$ in our notation.

7. `void Prover::UserData::Recover(Oracle& oracle, const Proof& p, const uintBig& seed)` lines 360-390

   `UserData` function is not used yet.

8. `void Prover::InitNonces(const uintBig& seed)` lines 391-422

   This method generates random values:

   rA, rB, rC, rD, m_Tau (size = M), m_a (size = $M \cdot (n - 1)$).

   Compute $r_A, r_B, r_C, r_D, a_{j,1}, \dots, a_{j,n-1} \leftarrow_R \mathbb{Z}_q$
   for $j \in [0, \dots, m - 1]$

   $$a_{j,0} = -\sum_{i=1}^{n-1} a_{j,i}.$$

9. `void Prover::CalculateP()` lines 423-476

   calculates coefficients $p_{i,k}$. These coefficients are used when $f_{i,j}$ are multiplied.

10. `void Prover::ExtractABCD()` lines 478-560

    `483-494` – *Commitment m_A:* `get_At` outputs the values m_a. As a result, we get the commit m_A with scalars: $m\_a[j \cdot n + i], i = 0, \dots n - 1, j = 0, \dots M - 1$ and rA.

    `496-517` – *Commitment m_B:* `get_At` returns `1` if `L % n == i` and `0` otherwise. This results in commit m_B with scalars: `1` or `0`, $i = 0, \dots n - 1, j = 0, \dots M - 1$, and rB.

    `519-542` – *Commitment m_C*. `get_At` returns $-m\_a[j \cdot n + i]$ if `L % n == i` and $m\_a[j \cdot n + i]$ otherwise.

    The result is m_C commit with $\pm m\_a[j \cdot n + i], i = 0, \dots n - 1, j = 0, \dots M - 1$ and rC.

    `544-560` – *Commitments m_D*: `get_At` outputs $-m\_a^2[j \cdot n + i]$. The result is the commit m_D with $-m\_a^2[j \cdot n + i], i = 0, \dots n - 1, j = 0, \dots M - 1$ and rD.

IMPORTANT: in the article, the commit is denoted as c_i. In our case, this commit refers to the difference of commitments, as written in [3]: $c_i = C_i - ptBias = [C_i$ is from Shielded pool, $ptBias = C_{out} + s \cdot J] == (k_s + k_{MW} - k_{out}) \cdot G$.

Methods `ExtractG_Part()` and `ExtractG()` are used to calculate the vector m_vG. Its elements correspond to the values of $G_k = \sum_{i=0}^{N-1} C_i p_{i,k} + Comm(0, \tau_k)$.

In this case, values $\left(-\sum_{i=0}^{N-1} ptBias \cdot p_{i,k}\right)$ are added to $G_k = \sum_{i=0}^{N-1} \left(C_i p_{i,k} - ptBias \cdot p_{i,k} + Comm(0, \tau_k)\right)$.

Value $\sum_{i=0}^{N-1} C_i \cdot p_{i,k}$ is calculated in `ExtractG_Part()` method, whereas $\left(-\sum_{i=0}^{N-1} ptBias \cdot p_{i,k}\right)$ value is calculated in `ExtractG()` method.

11. `void Prover::ExtractG_Part(GB* pGB, uint32_ti 0, uint32_t i1)` lines 568-613

    Here, structure vector m_vGB is filled with the following values:

    $m\_vGB.m\_G = \sum_{i=0}^{N-1} C_i p_{i,k}$, $m\_vGB.m\_kBias = \sum_{i=0}^{N-1} p_{i,k}$.

12. `void Prover::ExtractG(const Point::Native& ptBias)` lines 615-689

    `681` – when `Calculate()` method is called, commit $Comm(0, \tau_k)$ is calculated ($Comm(0, m\_Tau_k)$ in the code). m_kBias is multiplied by $(-ptBias)$. Thus, all three components of G_k are calculated. The next G_k is added to $m\_Proof.m\_Part1.m\_vG$.

13. `void Prover::ExtractPart2(Oracle& oracle)` lines 699-739

    oracle >> x1; getting challenge (as in article).

    `ExtractBlinded(m_Proof.m_Part2.m_zA, m_vBuf[Idx::rB], x1, m_vBuf[Idx::rA]);` calculates $m\_zA = rB \cdot x1 + rA$.

    `ExtractBlinded(m_Proof.m_Part2.m_zC, m_vBuf[Idx::rC], x1, m_vBuf[Idx::rD]);` calculates $m\_zC = rC \cdot x1 + rD$.

    `707-717` – calculates $m\_zR = -(m\_Tau[0] + m\_Tau[1] \cdot x_1 + m\_Tau[2] \cdot x_1^2 + m\_Tau[3] \cdot x_1^3 + m\_Tau[4] \cdot x_1^4) + (k_s + k_{mw} - k_{out}) \cdot x_1^5$.

    m_Tau is a coefficient of p_k for x^k. They are generated randomly at the very beginning. $(k_s + k_{mw} - k_{out})$ is r in the documentation.

    `718-739` – calculates $f_{j,1}$, $f_{j,0}$ which are added to the vector m_vF:
    $\forall j \in [0, m-1], i \in [1, n-1] \quad f_{j,i} = \sigma_{l_j i} x + a_{j,i}$.

14. `void Prover::Generate(const uintBig& seed, Oracle& oracle, const Point::Native& ptBias)` lines 741-765

    In this method, the order of functions calls is clear.

    `InitNonces(seed);` – generates rA, rB, rC, rD, and $m\_Tau(\tau_k)$, and $m\_a(a_j)$.

    `ExtractABCD();` – calculates m_A, m_B, m_C, m_D.

`CalculateP();` – calculates $p_{i,k}$.

`ExtractG(ptBias);` – calculates m_vG, or G_k in the documentation.

`m_Proof.m_Part1.Expose(oracle);` – values m_A, m_B, m_C, m_D, and m_vG vector are sent to an oracle.

`ExtractPart2(oracle);` – calculates m_zA, m_zC, m_zR, and vector m_vF.

> ***Comment from the developers:*** *We use batch-verification technique throughout the code extensively. We need to verify that many different expressions of the form* $Sum(k[i,j] * A[j]) = 0$*, verify for each* i*. Instead of verifying each of them individually we multiply each expression by a pseudo-random multiplier and verify that sum of them all is zero. Means,* $Sum(k[i,j] * multiplier[i] * A[j]) = 0$*.*
>
> *This is an important optimization. If a point* $A[j]$ *is shared for different expressions, then obviously adding scalars is more efficient than points. But even if all the points are different, still calculating multi-exponentiations of many points at once is beneficial.*
>
> *So, the whole Lelantus proof is converted into one big equation. Moreover, many such proofs are also combined, and the whole block (or even many blocks verified at once) is verified as a single multi-exponentiation, which includes Lelantus proofs, bulletproofs, and Schnorr's signatures.*
>
> *All this logic is handled in* `BatchContext` *class. The* `EquationBegin()` *member function regenerates the pseudo-random multiplier, and functions* `AddCasual()` *and* `AddPrepared()` *automatically multiply the given scalar by the current multiplier.*
>
> *However, in some cases we use the multiplier explicitly as an optimization. For example if there's a sequence of many scalars derived from each other (like powers of a number), then we multiplier explicitly to calculate the initial values, and then use versions* `AddCasual`*(*`bPremultiplied` *set to* `true`*) and* `AddPreparedM()` *that assume multiplier was already applied.*

Shield.cpp review

https://github.com/BeamMW/beam/wiki/MW-CLA

`bool ShieldedTxo::IsValid(ECC::Oracle& oracle, ECC::Point::Native& comm, ECC::Point::Native& ser) const` line 48

Validation of Schnorr's signature. RangeProof validation.

Input

Consists of the following:

- Range within the shielded pool, that contains the being-spent element.
- *SpendKey* is revealed, and the whole shielded input is signed by the appropriate private key.
- Optionally asset info: the blinded asset generator + asset surjection proof.
- Output commitment $C_{out} = k_{out} \cdot G + v \cdot H$.
    - It should commit to the same value, but the blinding factor k_{out} is different from that used in shielded output.
- Generalized Schnorr's signature, that proves the C_{out} is indeed of this form.
- Sigma proof for the rest.
- *m_pK[2]* – k_s, s are stored here.
- *m_SerialPub* – commitment $C_s = k_s \cdot G + s \cdot J$.

  `void ShieldedTxo::Data::TicketParams::DoubleBlindedCommitment(ECC::Point::Native& res, const ECC::Scalar::Native* pK)` line 135

  calculates $r \cdot J + k \cdot G$.

  `void ShieldedTxo::Data::TicketParams::set_FromkG(Key::IPKdf& gen, Key::IKdf* pGenPriv, Key::IPKdf& ser)` line 151

  Receives serial number and initializes *m_pK[1]* as this sn.

  `void ShieldedTxo::Data::TicketParams::GenerateInternal(Ticket& s, const ECC::Hash::Value& nonce, Key::IPKdf& gen, Key::IKdf* pGenPriv, Key::IPKdf& ser)` line 189

  Calculates C_{out} and forms Schnorr's signature.

  `void ShieldedTxo::Data::TicketParams::set_SharedSecretFromKs(ECC::Point& ptSerialPub, Key::IPKdf& gen)` line 206

  calculates $C_s = k_s \cdot G + s \cdot J$.

`bool ShieldedTxo::Data::TicketParams::Recover(const Ticket& s, const Viewer& v)` line 229

checks the signature and gets the challenge.

`268` – checks the serial number

output

Consists of the following:

- Blinded serial number commitment: $C_s = k_s \cdot G + s \cdot J$.
- Generalized Schnorr's signature that proves the above commitment is indeed of this from.
- Optionally asset info: the blinded asset generator + asset surjection proof.
- UTXO commitment $C_{MW} = k_{MW} \cdot G + v \cdot H$.
- Rangeproof.

This applies to Shielded input:

- m_k - k_{MW}.
  `void ShieldedTxo::Data::OutputParams::Generate(ShieldedTxo& txo, const ECC::Hash::Value& hvShared, ECC::Oracle& oracle, bool bHideAssetAlways /* = false */)` line 337
  Here commitment $C_{MW} = k_{MW} \cdot G + k_1 \cdot v \cdot G + H *\cdot v$ is formed. Then `RangeProof.CoSign` is called.
  `bool ShieldedTxo::Data::OutputParams::Recover(const ShieldedTxo& txo, const ECC::Hash::Value& hvShared, ECC::Oracle& oracle)` line 383
  `408-418` – calculates $C_{MW} = k_{MW} \cdot G + k_1 \cdot v \cdot G + H *\cdot v$ and checks for equity of commitments.

Manual Security Analysis

Here we inspect the code manually and check whether it is secure and meets best coding practices.

Critical issues

Critical issues seriously endanger project security. We highly recommend fixing them.

The audit showed no critical issues

Medium severity issues

Medium issues can influence project operation in current implementation. We highly recommend addressing them.

The audit showed no issues of medium severity.

Low severity issues

Low severity issues can influence project operation in future versions of code. We recommend taking them into account.

Zeroing objects

In numerous locations various objects are cleared by using `memset` or equivalent. This behavior is well defined only for so-called *standard layout* types. Apparently, all the current memory zeroing cases in the code deal with such types. However, this assumption makes those types *fragile* for potential changes.

If the use of `memset` is preferred for performance reasons, it is recommended to guard the uses against possible breakage with `static_assert` as in `common.h`.

```
template <typename T>
inline void ZeroObject(T& x)
{
      // TODO: uncomment and fix
      //static_assert(std::is_standard_layout_v<T>);
      static_assert(std::is_trivially_destructible_v<T>);
      ZeroObjectUnchecked(x);
}
```

`static_assert` guard incurs no performance penalty but increases the type safety of the code. For compatibility reasons, the existing code asserts trivially destructable object rather than standard layout. This should be reviewed and fixed.

Use of `volatile`

volatile keyword is used in the code. Some examples are as follows.

File `block_crypt.h`

```
bool Combine(IReader&& r0, IReader&& r1, const volatile bool& bStop);
```

And the implementation (`block_rw.cpp`)

```
while (true)
{
    if (bStop)
        return false;
```

The intent of using `volatile` is to prevent storing `bStop` variable in a CPU register, so another thread can request premature finishing of the function. However, `volatile` keyword is not intended to work in multi-threading environment. Volatile variables are still prone to *data races*.

Instead of `volatile` keyword, it is recommended to use `std::atomic`, for example:

```
bool Combine(IReader&& r0, IReader&& r1, const std::atomic_bool& bStop);
```

> ***Comment from the developers:*** *We know that `volatile` is prone to data races, memory i/o reordering, etc. But we assume they give better performance than `atomic` operations (which translate to asm instructions with `lock` semantics), especially when used in loops.*
> *In those specific cases you mentioned we prefer to use `volatile`, because data races are not important. It is an abortion flag; we do not care if it will have effect immediately or with some minimal delay.*
>
> ***SmartDec response:*** *To avoid excessive costly synchronization, it is better to use weak memory ordering: `memory_order_relaxed`.*
> *For example, `atomic_var.load(memory_order_relaxed).`*

Return value ignored

Throughout the code the following idiom is used for *getter* methods:

```
bool get_X(X& x) { x = whatever; return success; }
```

However, the return code is not checked consistently. For example, `block_rw.cpp`:

```
Merkle::Hash hv;
v.get_Live(hv);

if (!(m_Cwp.m_hvRootLive == hv))
    ThrowBadData();
```

In this code snipped the following check remedies.

> ***Comment from the developers:*** *This is a correct point. We do not "ignore" return values on purpose. But we assume specific conventions. In this specific place, we assumed that, apart of returning `false`, the called function also zeroes the hv. But apparently it does not.*
>
> *P.S. In this specific point there should be no situation when false is returned. But to make it more obvious, at least `assert()` should be placed.*

Struct `BigFloat` inside struct `Difficulty`

File `difficulty.cpp` contains definition of `BigFloat` structure. Consider moving this structure into a separate file.

Inefficient use of file descriptors

File `block_crypt.cpp` defines `GenRandom`. The POSIX version of the function opens `/dev/urandom` each time it is called.

Inefficient code

File `lelantus.cpp` defines function `Cfg::get_N()`.

Surrogate scoped enums

File `navigator.h` contains the following definition.

```
struct Type {
    enum Enum {
        Tag,
        count
    };
};
```

This definition places the constants `Tag` and `count` to the scope of `Type` structure. However, the same intent is expressed better with enum classes:

```
enum class Type { Tag, count };
```

Custom `offsetof` implementation

File `common.h` contains the definition of `IMPLEMENT_GET_PARENT_OBJ`.

```
#define IMPLEMENT_GET_PARENT_OBJ(parent_class, this_var) \
    parent_class& get_ParentObj() const { \
        parent_class*  p = (parent_class*) (((uint8_t*) this) + 1 - (uint8_t*) (&((parent_class*) 1)->this_var)); \
        assert(this == &p->this_var); /* this also tests that the variable of the correct type */ \
        return *p; \
    }
```

This macro generates function `get_ParentObj`, which returns the reference to the object that aggregates this object.

However, the expression `&((parent_class*) 1)->this_var` results in *Undefined Behavior*.

It is recommended to rewrite the macro using `offsetof` standard macro. `offsetof` is included into the modern standards and has specified behavior for standard layout types.

Thread joining in destructors

File `treasury.cpp` contains the following code.

```
~ThreadPool()
{
    for (size_t i = 0; i < m_vThreads.size(); i++)
        if (m_vThreads[i].joinable())
            m_vThreads[i].join();
}
```

The destructor of class `ThreadPool` joins the threads in vector `m_vThreads`.

Suboptimal implementation for arbitrary-precision arithmetics

File `uintBig.cpp` contains an in-house implementation for arbitrary-precision arithmetics. The implementation uses simple multiplication and division algorithms with time complexity $O(n^2)$.

Off-by-one error

File `http_msg_creator.cpp` contains the following function.

```
37  bool write_fmt(io::FragmentWriter& fw, const char* fmt, ...) {
38      static const int MAX_BUFSIZE = 4096;
39      char buf[MAX_BUFSIZE];
40      va_list ap;
41      va_start(ap, fmt);
42      int n = vsnprintf(buf, MAX_BUFSIZE, fmt, ap);
43      va_end(ap);
44      if (n < 0 || n > MAX_BUFSIZE) {
45          return false;
46      }
47      fw.write(buf, n);
48      return true;
49  }
```

Line 44 checks that the formatted string fits into the buffer `buf` but does not consider `\0` terminator byte. The correct check should look as follows.

```
if (n < 0 || n >= MAX_BUFSIZE) {
```

Missing integer overflow check

File `http_msg_reader.cpp`.

```
char* e = 0;
int64_t ret = strtol(val.data(), &e, 10);
if (size_t(e - val.data()) != val.size()) return defValue;
```

This code snippet does not check that string to integer conversion does not overflow. It may be fixed as follows:

```
char* e = 0;
errno = 0; // errno may contain stale error code
int64_t ret = strtol(val.data(), &e, 10);
if (errno || size_t(e - val.data()) != val.size()) return defValue;
```

Undefined behavior in functions from `<cctype>`

File `http_msg_reader.cpp`.

```
inline bool equal_ci(const char* a, const char* b, size_t sz) {
    const char* e = a + sz;
    for (; a !=e; ++a, ++b) {
        if (*a != tolower(*b)) return false;
    }
    return true;
}
```

According to the language standard `tolower` function accepts values in range `[-1...255]`. If the argument falls out of this range, the behavior of the function is undefined. On x86 platform `char` is a signed type, so the range of values of type char is `[-128...127]`. The standard-conforming usage of `toupper` is as follows.

```
if (*a != tolower((unsigned char) *b)) return false;
```

Usually nobody cares much about casting the argument to `unsigned char`, as all known standard library implementations support negative arguments and actually accept values in range `[-128...255]`, but it must be noted for the sake of correctness.

Insecure `SECURE_ERASE_OBJ`

File `hw_crypto.c`.

```
#define SECURE_ERASE_OBJ(x) memset(&x, 0, sizeof(x))
```

Use of `memset` is not considered secure. Compilers may optimize it out on high levels of optimization. Consider using a properly secure implementation of memory zeroing, as provided by libsodium or likes.

> ***Comment from the developers:*** *I agree. This is a reference code to be used in the HW wallet implementation (ledger, trezor, and similar devices). They will need to re-define `SECURE_ERASE_OBJ`, as long as several other things suitable for them. We keep this source code in our project to test that it performs identical to our C++ implementation of the similar functionality.*

List of references

[1] Gary Yu “Simple Schnorr Signature with Pedersen Commitment as Key”, p.4. Feb. 22, 2020. Link: https://eprint.iacr.org/2020/061.pdf

[2] Aram Jivanyan “Lelantus: Towards Confidentiality and Anonymity of Blockchain Transactions From Standard Assumptions”. Link: https://lelantus.io/lelantus.pdf

[3] https://github.com/BeamMW/beam/wiki/MW-CLA

This analysis was performed by SmartDec.

Katerina Troshina, Chief Executive Officer
Alexander Chernov, Chief Research Officer
Daria Korepanova, Security Analyst
Aleksey Ivushkin, Security Analyst
Boris Nikashin, Analyst
Alexander Seleznev, Chief Business Development Officer

June 23, 2020