Filecoin Bellman/BLS Signatures Cryptography Review

Protocol Labs

October 20, 2020 – Version 1.0

Prepared for

Friedel Ziegelmayer
Jonathan Victor

Prepared by

Eric Schorn
Paul Bottinelli
Javed Samuel

Executive Summary

Synopsis

In May 2020, Protocol Labs engaged NCC Group's Cryptography Services team to conduct a cryptography review of multiple Filecoin code repositories. Filecoin is a decentralized storage and content distribution network developed by Protocol Labs. These repositories implement finite field and group arithmetic, cryptographic pairings, SHA2 via intrinsics, BLS signatures and zk-SNARK operations. Taken together, these operations deliver the cutting-edge cryptographic primitives which are central to the security of the Filecoin network. This network relies upon *provable* security and authenticity to ensure user data is stored correctly and securely over time. The assessment was open-ended but was time-boxed to eleven person-days of effort. Good communication was established through a kick-off call, a status update call and over Slack. The assessment was followed by a brief retest of several findings in June 2020.

Scope

NCC Group's evaluation included the following GitHub source code repositories comprising approximately 26k lines of Rust. The latest commits were reviewed as noted and particular areas of focus are highlighted below.

- filecoin-project/ff commit `998ddb0`
  - Based on Zcash code, focus on assembly additions
- filecoin-project/group commit `57aa2e7`
  - Interface definitions
- filecoin-project/paired commit `7cc51a4`
  - Focus on implementation of hash-to-curve.
- filecoin-project/rust-sha2ni commit `29532da`
  - Dynamic feature detection, SHA intrinsics
  - Full audit, specific focus on new code
- filecoin-project/bls-signatures commit `0ecd251`
  - Full audit, specific focus on new code and compliance to draft specification
- filecoin-project/bellman commit `300b52d`
  - Comprehensive audit including a number of additions including the OpenCL GPU code.

Limitations

Included test cases relating to SHA2-intrinsics and (some) GPU functions were not run due to advanced hardware requirements. While this is generally an out-of-scope task, it did marginally impact testing convenience and productivity, though it did not materially impact the resulting in-scope coverage. Additional inspection of the various build flows/options is nevertheless recommended.

Key Findings

The target code repositories demonstrated a number of excellent design and implementation choices. These included the use of Rust to avoid the many safety-related challenges of C and C++, a separation of concerns that keeps related levels of cryptographic abstractions tightly coupled and modular, a solid set of test cases that enabled dynamic inspection during testing, and very advanced algorithms (particularly in hash-to-curve). The assessment did uncover several issues, the most notable including:

- A lack of distinct messages enforcement in `bls-signature` verification that could allow a "rogue key" to forge an aggregate signature.
- Two missed length checks related to input data validation. One may lead to a panic and the other could result in an unanticipated public key.
- A missed opportunity to check for errors within randomness generation which may present an issue if the OS entropy is exhausted or otherwise problematic.
- Slightly outdated and inconsistent dependency specifications that may increase the difficulty of debug.
- Missing and inconsistent compiler specification across repositories; Note that the compiler's nightly channel is required for code builds.

Retest Results

In June 2020, Protocol Labs undertook a retest of selected findings uncovered in the original assessment on an updated filecoin-project/bls-signatures (commit `0ecd251`) repository. Findings NCC-PRLB007-004, NCC-PRLB007-005, NCC-PRLB007-006, NCC-PRLB007-007 and NCC-PRLB007-008 were reviewed to determine whether each was fully fixed, partially fixed or not fixed. The detailed entry for each of these findings now includes a brief description of the retest

observations and results. The overall retest results can be summarized as follows:

- Findings NCC-PRLB007-004, NCC-PRLB007-005, NCC-PRLB007-006, NCC-PRLB007-007 and NCC-PRLB007-008 have been fully fixed.

The current status of each finding is also listed in the Table of Findings on page 5.

Strategic Recommendations

The in-scope code repositories follow secure coding practices and demonstrate careful attention to edge-cases. As the code moves towards production deployment, NCC Group recommends prioritizing the following focus areas:

- Ensure all application input is validated as tightly as possible as early as possible.
- Periodically revisit and update all repository dependencies and compiler specifications.
- Consider opportunities to adopt constant time algorithms and secret clearing.
- Document the implemented algorithms more clearly (e.g. in hash-to-curve) via code comments.
- Consider following the draft specification for BLS signatures more closely with explanations for divergence.

Dashboard

Target Metadata

Name	Filecoin BLS Signatures
Type	Cryptography Library
Platforms	Rust
Environment	Source code

Engagement Data

Type	Cryptography review
Method	Manual source inspection
Dates	2020-05-13 to 2020-05-22
Consultants	2
Level of Effort	11 person-days

Targets

https://github.com/filecoin-project/ff	Traits and utilities for working with finite fields
https://github.com/filecoin-project/group	Elliptic curve group traits and utilities
https://github.com/filecoin-project/pairing	Pairing-friendly elliptic curve library
https://github.com/filecoin-project/rust-sha2ni	Dynamic feature detection, SHA intrinsics
https://github.com/filecoin-project/bls-signatures	BLS signatures in Rust
https://github.com/filecoin-project/bellman	zk-SNARK library

Finding Breakdown

Critical issues	0
High issues	0
Medium issues	1
Low issues	5
Informational issues	2
Total issues	**8**

Category Breakdown

Configuration	1
Cryptography	2
Data Exposure	1
Data Validation	3
Patching	1

Component Breakdown

Systemic	3
bls-signatures	5

Key

Critical High Medium Low Informational

Table of Findings

For each finding, NCC Group uses a composite risk score that takes into account the severity of the risk, application's exposure and user population, technical difficulty of exploitation, and other factors. For an explanation of NCC Group's risk rating and finding categorization, see Appendix A on page 20.

Title	Status	ID	Risk
Distinct Messages not Enforced in Aggregate Verify	Fixed	006	Medium
Outdated Dependencies and Inconsistent Compiler Specification	Reported	001	Low
Secrets in Memory Not Cleared	Reported	003	Low
Missing Error Check and Non-Crypto Randomness Generator	Fixed	004	Low
Missing Length Check on Private Key Deserialization	Fixed	005	Low
Missing Length==0 Validation Check	Fixed	008	Low
Non Constant-Time Cryptography Implementation	Reported	002	Informational
Private Key Generation not Compliant	Fixed	007	Informational

Finding	**Distinct Messages not Enforced in Aggregate Verify**
Risk	**Medium** Impact: Low, Exploitability: Medium
Identifier	NCC-PRLB007-006
Status	Fixed
Category	Data Validation
Component	bls-signatures
Location	bls-signatures/src/signature.rs
Impact	A specially crafted public key (commonly referred to as the "rogue" key) can potentially be used to forge an aggregate signature.
Description	The BLS signature scheme allows users to produce aggregate signatures, combinations of a set of signatures, which can be verified more efficiently than the individual signatures separately.

The draft specification defines three signature schemes for BLS: basic, message augmentation and proof of possession. The difference between these variants lies in the way they protect against rogue key attacks. In this context, a rogue key is a maliciously crafted public key that can be used to forge aggregate signatures. The EUROCRYPT 2003 paper by Boneh, Gentry, Lynn, and Shacham, entitled *Aggregate and verifiably encrypted signatures from bilinear maps*,[1] describes this attack in Section 3.2 under *A potential attack on aggregate signatures* and proposes the distinct messages countermeasure in the following paragraph *A simple defense for aggregate signatures*. A copy of the paper can be found in the archives of the conference proceedings.[2]

In the basic scheme, which is what is implemented in the package `bls-signatures`, rogue key attacks are handled by requiring all messages signed in an aggregate signature to be distinct. This requirement is specified in Section 3.1. of the draft specification `draft-irtf-cfrg-bls-signature-02`[3] and is enforced during signature verification by the function `AggregateVerify`.

The code does not enforce that messages be different from one another, and as such is vulnerable to rogue key attacks. The following code excerpt, taken from `bls-signatures/src/signature.rs` shows the aggregate signature verification procedure. It is easy to see that no check is performed regarding the distinctiveness of the `hashes`.

```
/// Verifies that the signature is the actual aggregated signature of hashes -
 ➜  pubkeys.
/// Calculated by `e(g1, signature) == \prod_{i = 0}^n e(pk_i, hash_i)`.
pub fn verify(signature: &Signature, hashes: &[G2], public_keys: &[PublicKey]) ->
 ➜  bool {
    if hashes.len() != public_keys.len() {
        return false;
    }

    let mut prepared: Vec<_> = public_keys
        .par_iter()
```

[1] https://link.springer.com/chapter/10.1007%2F3-540-39200-9_26
[2] https://www.iacr.org/archive/eurocrypt2003/26560416/26560416.pdf
[3] https://tools.ietf.org/html/draft-irtf-cfrg-bls-signature-02#section-3.1

```
        .zip(hashes.par_iter())
        .map(|(pk, h)| (pk.as_affine().prepare(), h.into_affine().prepare()))
        .collect();

    let mut g1_neg = G1Affine::one();
    g1_neg.negate();
    prepared.push((g1_neg.prepare(), signature.0.prepare()));

    let prepared_refs = prepared.iter().map(|(a, b)| (a, b)).collect::<Vec<_>>();

    if let Some(res) =
    ➔  Bls12::final_exponentiation(&Bls12::miller_loop(&prepared_refs)) {
        Fq12::one() == res
    } else {
        false
    }
}
```

Additionally, it should be noted that the function `CoreAggregateVerify` defined in Section 2.9. of the draft specification, expects the messages themselves as arguments and not their hashes. It is unclear whether this has a direct impact, but depending on the code path followed and the provenance of this data, this might give unnecessary freedom to an attacker.

Recommendation Implement the function `AggregateVerify` as defined in section 3.1. of the draft specification. This function should first ensure that all messages are distinct, before proceeding with the aggregate signature verification. Additionally, this function should hash the messages instead of expecting hashed data.

Retest Results The updated `signature.rs` source file was inspected to confirm the recommended addition of logic on lines 106-112 corresponding to step 1 of the `AggregateVerify` procedure defined in section 3.1.1 of the draft specification. This logic ensures that no two messages are equal before proceeding with verification. As such, this finding has been marked as fixed.

Finding	**Outdated Dependencies and Inconsistent Compiler Specification**
Risk	**Low** Impact: Low, Exploitability: Low
Identifier	NCC-PRLB007-001
Status	Reported
Category	Patching
Component	Systemic
Location	All six code repositories
Impact	An attacker will attempt to identify and then utilize any vulnerabilities stemming from outdated dependencies to exploit the application. Silently changing toolchain versions can introduce instability and increase debug difficulty.
Description	Incorporating outdated dependencies is one of the most common, serious and exploited application vulnerabilities. Unspecified toolchain versions will not insulate the application and developer from silent changes that can introduce instability and greatly increase debug difficulty.

The project incorporates several direct and transitive dependencies that are marginally out of date. For example, output from the `cargo outdated` tool is shown below for the `bls-signatures` repository.

```
eschorn@ataraxy:~/work/filecoin/bls-signatures$ cargo outdated
Name                          Project  Compat   Latest   Kind         Platform
----                          -------  ------   ------   ----         --------
base64                        0.12.0   0.12.1   0.12.1   Development  ---
base64-serde->base64          0.12.0   0.12.1   0.12.1   Normal       ---
base64-serde->serde           1.0.106  1.0.110  1.0.110  Normal       ---
fff_derive->proc-macro2       1.0.10   1.0.12   1.0.12   Normal       ---
fff_derive->quote             1.0.3    1.0.5    1.0.5    Normal       ---
fff_derive->syn               1.0.18   1.0.21   1.0.21   Normal       ---
getrandom->libc               0.2.69   0.2.70   0.2.70   Normal       cfg(unix)
groupy->thiserror             1.0.16   1.0.17   1.0.17   Normal       ---
hermit-abi->libc              0.2.69   0.2.70   0.2.70   Normal       ---
num_cpus->libc                0.2.69   0.2.70   0.2.70   Normal       ---
paired                        0.19.0   0.19.1   0.19.1   Normal       ---
quote->proc-macro2            1.0.10   1.0.12   1.0.12   Normal       ---
rand->libc                    0.2.69   0.2.70   0.2.70   Normal       cfg(unix)
serde                         1.0.106  1.0.110  1.0.110  Development  ---
serde->serde_derive           1.0.106  1.0.110  1.0.110  Normal       ---
serde_derive->proc-macro2     1.0.10   1.0.12   1.0.12   Normal       ---
serde_derive->quote           1.0.3    1.0.5    1.0.5    Normal       ---
serde_derive->syn             1.0.18   1.0.21   1.0.21   Normal       ---
serde_json                    1.0.52   1.0.53   1.0.53   Development  ---
serde_json->serde             1.0.106  1.0.110  1.0.110  Normal       ---
syn->proc-macro2              1.0.10   1.0.12   1.0.12   Normal       ---
syn->quote                    1.0.3    1.0.5    1.0.5    Normal       ---
thiserror                     1.0.16   1.0.17   1.0.17   Normal       ---
thiserror->thiserror-impl     1.0.16   1.0.17   1.0.17   Normal       ---
thiserror-impl->proc-macro2   1.0.10   1.0.12   1.0.12   Normal       ---
thiserror-impl->quote         1.0.3    1.0.5    1.0.5    Normal       ---
thiserror-impl->syn           1.0.18   1.0.21   1.0.21   Normal       ---
```

In addition, note that the in-scope repositories do not fully refer to the latest version of them-

selves. For example:

- The `ff/Cargo.toml` file declares this code to be version 0.2.2. However, the `group`, `pairing`, `bellman`, and `bls-signatures` repositories specify version 0.2.0.
- The `group/Cargo.toml` file declares this code to be version 0.3.1. However, the `pairing` and `bls-signatures` repositories specify version 0.3.0.
- The `rust-sha2ni/Cargo.toml` file declares this code to be version 0.8.5. However, the `bls-signatures` repository specifies version 0.8.1.

Separately, the `bellman`, `bls-signatures` and `pairing` repositories each include a file named `rust-toolchain` that specifies an exact compiler version. The first file specifies a different compiler version than the latter two, specifically:

```
nightly-2020-01-08
nightly-2020-02-17
```

The `ff`, `group` and `rust-sha2ni` repositories lack this file altogether. As a result, it can be difficult to determine the precise compiler version used on a particular build which may then increase debug challenges. Note that the code requires the 'nightly' compiler which is intended to change rapidly.

Recommendation Add a periodic gating milestone to the development process that involves reviewing toolchain and all application dependencies for inconsistent, outdated or vulnerable versions. In the first instance, working from the bottom of the dependency tree towards the top (e.g. from `ff` towards `bellman`):

- Specify the same recent/latest version of the compiler in the `rust-toolchain` file.
- Update the `Cargo.toml` files to refer to the latest version of themselves.
- Update the `Cargo.toml` files to refer to the latest external dependencies via `cargo update`
- Confirm 'freshness' with a tool such as `cargo outdated`.

This would be a good opportunity to bump the version number of each repository for completeness.

Review the `.circleci/config.yaml` files to ensure they are picking up the expected code and toolchain versions.

Finding	**Secrets in Memory Not Cleared**
Risk	**Low** Impact: Medium, Exploitability: Undetermined
Identifier	NCC-PRLB007-003
Status	Reported
Category	Data Exposure
Component	Systemic
Location	Systemic, starting with `ff` repository
Impact	If regions of memory are accessed by an attacker, perhaps via a core dump or debugger, the attacker may be able to extract sensitive secret values.
Description	Typically, all of a function's local variables remain in process memory after the function goes out of scope, unless they are overwritten by new data. Data that is still stored on the stack or heap but is no longer in scope is referred to as stale (or garbage) data. Stale data is vulnerable to disclosure through means such as data leaks and debugging if the process is running with low privileges. As a result, sensitive data should not remain in memory once it goes out of scope. For example, in the event of a program crash, stale data may be core dumped. If an attacker has access to read a core dump, then they will be able to read any sensitive data that was not zeroed-out before the crash. Note that the `ff-zeroize` crate[4] is a version of the `ff` crate that incorporates 'zeroize' features.
Recommendation	Consider incorporating the `ff-zeroize` library as a temporary substitute for `ff` and implement similar extensions to the other repositories. The `ff-zeroize` crate uses the `Zeroize` trait to explicitly clear secret (or sensitive) values in memory when they are no longer needed. This trait is provided by the `zeroize` crate[5] which utilizes `core::ptr::write_volatile` and `core::sync::atomic` memory fences to automate zero-on-drop and guarantees the operation will not be "optimized away". Note that the `zeroize` crate can be used to zero values from either the stack or the heap. Further, the `pin`[6] crate can be leveraged to ensure data kept on the stack isn't moved.

[4] https://crates.io/crates/ff-zeroize
[5] https://docs.rs/zeroize/1.1.0/zeroize/
[6] https://doc.rust-lang.org/std/pin/struct.Pin.html

Finding	**Missing Error Check and Non-Crypto Randomness Generator**
Risk	**Low** Impact: Medium, Exploitability: Low
Identifier	NCC-PRLB007-004
Status	Fixed
Category	Cryptography
Component	bls-signatures
Location	Line 88 of bls-signatures/src/key.rs
Impact	An error condition that surfaces while generating random values, perhaps due to insufficient entropy, may not be detected and may impact the characteristics of secret values.

Description

When generating a secret key, it is imperative to draw from a cryptographically secure source of entropy and pay particular attention to potential error conditions. Unusual (and/or unexpected) error conditions can stem from a wide variety of sources as suggested by the following excerpt from the Ubuntu manual page for `getrandom`[7]:

> **GRND_RANDOM**
> If this bit is set, then random bytes are drawn from the random source ... instead of the urandom source. The random source is limited based on the entropy that can be obtained from environmental noise. If the number of available bytes in the random source is less than requested in buflen, the call returns just the available random bytes. If no random bytes are available, the behavior depends on the presence of **GRND_NONBLOCK** in the flags argument.

The `generate()` function implemented on `PrivateKey` in `keys.rs` shown below uses the RngCore `fill_bytes()` method which does not explicitly return an error indicator. The `rand_core` crate documentation[8] indicates that the `try_fill_bytes()` method is a variant of `fill_bytes()` method that does allow for explicit error detection and handling.

```
84 pub fn generate<R: RngCore>(rng: &mut R) -> Self {
85     // IKM must be at least 32 bytes long:
86     // https://tools.ietf.org/html/draft-irtf-cfrg-bls-signature-00#section-2.3
87     let mut ikm = [0u8; 32];
88     rng.fill_bytes(&mut ikm);
89     Self::new(ikm)
90 }
```

Note that there are two similar usages in test code, which are not a security issue.

- Line 586 of `pairing/src/bls12_381/fq.rs` (test code)
- Line 853 of `pairing/src/bls12_381/ec/mod.rs` (test code)

Separately, the trait `rand_core::CryptoRng`[9] is used to indicate that an RngCore or BlockRngCore implementation is supposed to be cryptographically secure. This is absent in the above code.

Recommendation

Utilize the `try_fill_bytes()` method along with logic to detect and handle errors. Utilize the `rand_core::CryptoRng` trait to prevent users from inadvertently supplying a determin-

[7] http://manpages.ubuntu.com/manpages/bionic/man2/getrandom.2.html
[8] https://rust-random.github.io/rand/rand_core/trait.RngCore.html#tymethod.try_fill_bytes
[9] https://rust-random.github.io/rand/rand_core/trait.CryptoRng.html

istic random number generator.

Retest Results

The updated `key.rs` source file was inspected to confirm the recommended use of the `try_fill_bytes()` method (with `expect` clause) on lines 86-87 as well as the additional `CryptoRnd` attribute on lines 8 and 82. As such, this finding has been marked as fixed.

Finding	**Missing Length Check on Private Key Deserialization**
Risk	**Low** Impact: Low, Exploitability: Low
Identifier	NCC-PRLB007-005
Status	Fixed
Category	Data Validation
Component	bls-signatures
Location	bls-signatures/src/keys.rs
Impact	Oversized key material provided to the deserialization function will have its trailing portion ignored, which may result in an unexpected private key used in downstream logic.
Description	All input should be aggressively validated for correctness as early and completely as possible to reject incorrect values and reduce the complexity requirements placed on downstream logic. The `from_bytes()` function implemented for the `Serialize` trait of `PublicKey` in `keys.rs` first checks the length of its raw bytes input for the correct size of `G1Compressed::size()`. This is very good practice, and the check is shown below.

```
177 if raw.len() != G1Compressed::size() {
178     return Err(Error::SizeMismatch);
179 }
```

However, the `from_bytes()` function implemented for the `Serialize` trait of `PrivateKey` in `keys.rs` does not check the length of its raw bytes input. Note that there is an expected correct length that *can* be checked. This function is shown below.

```
129 fn from_bytes(raw: &[u8]) -> Result<Self, Error> {
130     let mut res = FrRepr::default();
131     let mut reader = Cursor::new(raw);
132     let mut buf = [0; 8];
133
134     for digit in res.0.as_mut().iter_mut() {
135         reader.read_exact(&mut buf)?;
136         *digit = u64::from_le_bytes(buf);
137     }
138
139     Ok(Fr::from_repr(res)?.into())
140 }
```

If the function is provided with undersized raw bytes input, it will report an error on line 135. If the function is provided with oversized raw bytes input, the trailing bytes are ignored. The latter could become an issue if an attacker were able to prepend malicious data as the key would then become controllable. In any event, the code should reject incorrectly sized input.

Note that the error portion of `Result` can be returned via the `?` operator (e.g., line 135/139) but is not otherwise explicitly set in this function. Thus, all errors are effectively passed-through (which indicates minimal validation).

Recommendation	Perform a check on the expected length of raw bytes input for the private key, similar to that done on the public key.

Retest Results The updated `key.rs` source file was inspected to confirm the addition of the recommended length check for the supplied raw bytes on lines 129-132. As such, this finding has been marked as fixed.

Finding	**Missing Length==0 Validation Check**
Risk	**Low** Impact: Low, Exploitability: Low
Identifier	NCC-PRLB007-008
Status	Fixed
Category	Data Validation
Component	bls-signatures
Location	bls-signatures/src/signature.rs
Impact	When supplied with operands of length 0, the `verify()` function progresses into the Miller loop before panicking and potentially allowing for denial of service.
Description	The `verify()` function in `signature.rs` shown below checks for inconsistent lengths between the supplied input `hashes` and `public_keys`. Inconsistent lengths return an immediate false. However, when supplied with `Vec`s of length 0, this check passes and the code progresses into the Miller loop near the end before panicking when attempting to reference the operand (rather than returning false).

```
88  pub fn verify(signature: &Signature, hashes: &[G2], public_keys: &[PublicKey]) ->
    ➜  bool {
89      if hashes.len() != public_keys.len() {
90          return false;
91      }
92
93      let mut prepared: Vec<_> = public_keys
94          .par_iter()
95          .zip(hashes.par_iter())
96          .map(|(pk, h)| (pk.as_affine().prepare(), h.into_affine().prepare()))
97          .collect();
98
99      let mut g1_neg = G1Affine::one();
100     g1_neg.negate();
101     prepared.push((g1_neg.prepare(), signature.0.prepare()));
102
103     let prepared_refs = prepared.iter().map(|(a, b)| (a, b)).collect::<Vec<_>>();
104
105     if let Some(res) =
        ➜  Bls12::final_exponentiation(&Bls12::miller_loop(&prepared_refs)) {
106         Fq12::one() == res
107     } else {
108         false
109     }
110 }
```

Recommendation	Adapt the length check to include a zero length condition.
Retest Results	The updated `signature.rs` source file was inspected to confirm the addition of the recommended length check condition (equal to zero) for the supplied `hashes` and `public_keys` structures on lines 93-95. As such, this finding has been marked as fixed.

Finding	**Non Constant-Time Cryptography Implementation**
Risk	**Informational** Impact: Medium, Exploitability: Undetermined
Identifier	NCC-PRLB007-002
Status	Reported
Category	Cryptography
Component	Systemic
Location	Systemic
Impact	Many significant operations which handle sensitive data are not written to execute in constant time. This may make timing and other microarchitectural attacks possible, leaking sensitive data to an external attacker.
Description	Timing side-channel attacks allow the extraction of information about secret data by measuring the time required to perform operations involving the data. They were first described in 1996 in the context of RSA private key operations.[10] In 2005, timing attacks were extended into cache attacks,[11] in which a secret-dependent memory access pattern is revealed through timing measures made later on by the attacker. A variety of usage contexts allow attackers to perform such measures with enough precision to enact private key recovery, e.g. when the attacker can run his own code as an unprivileged process or another virtual machine co-hosted on the same hardware. Even remote measurements over a network have been demonstrated to be possible.[12] Other relevant examples[13,14,15] stem from OpenSSL's bignum non-constant time operations such as modular reduction, comparisons, and multiplication. Even SGX enclaves have proven vulnerable[16,17] to timing side-channel attacks. The Protocol Labs in-scope code repositories cannot provide constant time assurances. There are several operations where execution time is dependent upon the supplied data. The foundational `ff` repository performs a number of operations where execution time may depend upon secret data, including computing the inverse via a variant of the Euclidean Algorithm. The repository's `README.md` file includes a constant-time disclaimer: > This library does not provide constant-time guarantees. The foundational `pairing` library also performs a number of operations where execution time may depend upon secret data, such as point arithmetic on a curve. The repository's `README.md` file also includes a constant-time disclaimer: > This library does not make any guarantees about constant-time operations, memory access patterns, or resistance to side-channel attacks. As it is understood that timing side-channels are not part of Protocol Labs' threat model, this finding has been categorized as 'Informational'.
Recommendation	Unfortunately there are no readily-available libraries that deliver this functionality in constant

[10] https://www.paulkocher.com/TimingAttacks.pdf
[11] https://cr.yp.to/antiforgery/cachetiming-20050414.pdf
[12] https://crypto.stanford.edu/~dabo/abstracts/ssl-timing.html
[13] https://www.nccgroup.trust/us/our-research/return-of-the-hidden-number-problem/
[14] https://eyalro.net/project/cat/
[15] https://eprint.iacr.org/2013/448.pdf
[16] https://arxiv.org/pdf/1703.06986.pdf
[17] https://people.cs.kuleuven.be/~jo.vanbulck/ccs18.pdf

time. Constant-time operation does appear to be a long-term goal for these libraries, but not a high priority. Thus, be aware of these limitations in context of the application's threat model.

Finding	**Private Key Generation not Compliant**
Risk	**Informational** Impact: Low, Exploitability: None
Identifier	NCC-PRLB007-007
Status	Fixed
Category	Configuration
Component	bls-signatures
Location	bls-signatures/src/key.rs
Impact	The private keys generated are not fully compliant with the draft specification. This could prevent interoperability between different implementations. It could also potentially reduce the security of the keys, since different inputs might result in the same private key generated.

Description In `bls-signatures/src/key.rs`, private keys are generated with the `key_gen` function, as excerpted below. The function mostly follows the draft specification in that it uses HKDF correctly in its Extract/Expand paradigm (the call to `new` implicitly calls `extract`).

```
/// Hash a secret key sk to the secret exponent x'; then (PK, SK) = (g^{x'}, x').
fn key_gen<T: AsRef<[u8]>>(data: T) -> Fr {
    let mut result = GenericArray::<u8, U48>::default();

    // `result` has enough length to hold the output from HKDF expansion
    assert!(Hkdf::<Sha256>::new(Some(SALT), data.as_ref())
        .expand(&[], &mut result)
        .is_ok());
    Fr::from_okm(&result)
}
```

However, the function does not completely follow the draft specification for private key generation `draft-irtf-cfrg-bls-signature-02`,[18] which is provided below for reference.

```
SK = KeyGen(IKM)
Inputs:
   - IKM, a secret octet string. See requirements above.
Outputs:
   - SK, a uniformly random integer such that 0 <= SK < r.
Parameters:
   - key_info, an optional octet string.
If key_info is not supplied, it defaults to the empty string.
Definitions:
   - HKDF-Extract is as defined in RFC5869, instantiated with hash H.
   - HKDF-Expand is as defined in RFC5869, instantiated with hash H.
   - I2OSP and OS2IP are as defined in RFC8017, Section 4.
   - L is the integer given by ceil((3 * ceil(log2(r))) / 16).
   - "BLS-SIG-KEYGEN-SALT-" is an ASCII string comprising 20 octets.
Procedure:
   1. PRK = HKDF-Extract("BLS-SIG-KEYGEN-SALT-", IKM || I2OSP(0, 1))
   2. OKM = HKDF-Expand(PRK, key_info || I2OSP(L, 2), L)
   3. SK = OS2IP(OKM) mod r
   4. return SK
```

More specifically, in step 1 of the `Procedure` above, a 0-byte should be appended to `IKM`

[18] https://tools.ietf.org/html/draft-irtf-cfrg-bls-signature-02#section-2.3

during the `Extract` call. The `key_gen` function does not append this `0`-byte during the call to `new`. Additionally, in step 2, the draft specification requires appending the quantity $L = \lceil (3\lceil \log_2(r) \rceil)/16 \rceil$ encoded over two bytes. Again, the `key_gen` function does not append this data during the `expand` call. The reason for adding `L` to the second argument is given in the HKDF RFC,[19] where Section 3.2 states that it is a way to "bind the key material to its length". Indeed, without providing this value, HKDF-Expand calls with different output lengths might result in the same private key.

Note however that this is not a strict requirement in the draft specification. Namely, the specification states the following:

> Other key generation approaches meeting these requirements MAY also be used; details of such methods are beyond the scope of this document.

Recommendation Add the appropriate fields for the computation of the key, namely one `0`-byte after the secret octet string `IKM` as well as `L` encoded over two bytes, as in the `Procedure` provided above.

Retest Results The updated `key.rs` source file was inspected to confirm the recommended modifications. Specifically, it was noted:

- A zero byte is appended to the secret octet string on line 199.
- The call to hkdf `expand()` on line 205 now includes 'L' as `&[0,48]`.

As such, this finding has been marked as fixed.

[19] https://tools.ietf.org/html/rfc5869

Appendix A: Finding Field Definitions

The following sections describe the risk rating and category assigned to issues NCC Group identified.

Risk Scale

NCC Group uses a composite risk score that takes into account the severity of the risk, application's exposure and user population, technical difficulty of exploitation, and other factors. The risk rating is NCC Group's recommended prioritization for addressing findings. Every organization has a different risk sensitivity, so to some extent these recommendations are more relative than absolute guidelines.

Overall Risk

Overall risk reflects NCC Group's estimation of the risk that a finding poses to the target system or systems. It takes into account the impact of the finding, the difficulty of exploitation, and any other relevant factors.

Critical Implies an immediate, easily accessible threat of total compromise.

High Implies an immediate threat of system compromise, or an easily accessible threat of large-scale breach.

Medium A difficult to exploit threat of large-scale breach, or easy compromise of a small portion of the application.

Low Implies a relatively minor threat to the application.

Informational No immediate threat to the application. May provide suggestions for application improvement, functional issues with the application, or conditions that could later lead to an exploitable finding.

Impact

Impact reflects the effects that successful exploitation has upon the target system or systems. It takes into account potential losses of confidentiality, integrity and availability, as well as potential reputational losses.

High Attackers can read or modify all data in a system, execute arbitrary code on the system, or escalate their privileges to superuser level.

Medium Attackers can read or modify some unauthorized data on a system, deny access to that system, or gain significant internal technical information.

Low Attackers can gain small amounts of unauthorized information or slightly degrade system performance. May have a negative public perception of security.

Exploitability

Exploitability reflects the ease with which attackers may exploit a finding. It takes into account the level of access required, availability of exploitation information, requirements relating to social engineering, race conditions, brute forcing, etc, and other impediments to exploitation.

High Attackers can unilaterally exploit the finding without special permissions or significant roadblocks.

Medium Attackers would need to leverage a third party, gain non-public information, exploit a race condition, already have privileged access, or otherwise overcome moderate hurdles in order to exploit the finding.

Low Exploitation requires implausible social engineering, a difficult race condition, guessing difficult-to-guess data, or is otherwise unlikely.

Category

NCC Group categorizes findings based on the security area to which those findings belong. This can help organizations identify gaps in secure development, deployment, patching, etc.

Access Controls	Related to authorization of users, and assessment of rights.
Auditing and Logging	Related to auditing of actions, or logging of problems.
Authentication	Related to the identification of users.
Configuration	Related to security configurations of servers, devices, or software.
Cryptography	Related to mathematical protections for data.
Data Exposure	Related to unintended exposure of sensitive information.
Data Validation	Related to improper reliance on the structure or values of data.
Denial of Service	Related to causing system failure.
Error Reporting	Related to the reporting of error conditions in a secure fashion.
Patching	Related to keeping software up to date.
Session Management	Related to the identification of authenticated users.
Timing	Related to race conditions, locking, or order of operations.

Appendix B: Audit Discussion Notes and Observations

1. Overview

This informational section highlights selected portions of the engagement methodology used, priority concerns investigated, and a few observations that do not warrant security-related findings but are worth discussion. The primary strategy for this project relied heavily on manual source code inspection, supported by some execution of the included test cases. Priority was given to the correctness of cryptographic algorithms and implementation, the specific focus areas highlighted in the executive summary, GPU and assembly-level optimizations, and general secure coding practices that could potentially impact legitimate operation.

2. Related References

- IETF CFRG draft Hash to Curve specification at https://tools.ietf.org/pdf/draft-irtf-cfrg-hash-to-curve-07.pdf.
- IETF CFRG draft BLS Signatures specification at https://tools.ietf.org/pdf/draft-irtf-cfrg-bls-signature-02.pdf.
- "Fast and simple constant-time hashing to the BLS12-381 elliptic curve" at https://eprint.iacr.org/2019/403.pdf. Pertinent to the `osswu_map()` logic in `pairing/src/bls12_381/ec/g2.rs`.
- "Efficient hash maps to G2 on BLS curves" at https://eprint.iacr.org/2017/419.pdf (clearing cofactor).
- Hash related code at https://github.com/algorand/pairing-plus/blob/master/src/bls12_381/osswu_map/g2.rs.

3. Initial Survey

The following primary code targets were cloned and briefly surveyed prior to the kick-off conference call.

- https://github.com/filecoin-project/ff
- https://github.com/filecoin-project/group
- https://github.com/filecoin-project/pairing
- https://github.com/filecoin-project/rust-sha2ni
- https://github.com/filecoin-project/bls-signatures
- https://github.com/filecoin-project/bellman

Dependencies were reviewed, resulting in finding NCC-PRLB007-001 on page 8. Each `Cargo.toml` was edited to ensure builds utilized in-scope repositories (e.g. adapting version and adding `path="../ff"`). Repositories were then built and test cases run.

Variable-time algorithms were found, resulting in finding NCC-PRLB007-002 on page 16. Secrets are not cleared after use, resulting in finding NCC-PRLB007-003 on page 10.

4. Detailed Review, Functionality

Each repository was reviewed in detail, running from the bottom of the dependency tree towards the top.

- The `ff` repository was compared to the Zcash fork source, as well as reviewed holistically. The primary differences centered on the `ff->fff` name change and `asm/mul_4.S`. This implements the `mod_mul_4w()` function by composing the `mul_256` multiplication and `red_256` Montgomery reduction macros. The function is 'bound' in `src/asm.rs` and then incorporated on line 765 of `ff_derive/src/lib.rs`. The latter file also defines the `add_assign_asm_impl()` function via intrinsics. Finally, the `build.rs` file selectively builds the assembly based upon target machine architecture.
  - Note that the constant on line 19/20 of `ff/ff_derive/src/lib.rs` is the (255-bit) BLS12-381 curve order (rather than the field prime).
  - Line 405 of `ff/ff_derive/src/lib.rs` seems repeated on second half of line 411. JetBrains Clion warns about a number of unused variables (such as lines 405 and 411) primarily involving `biguint_to_u64_vec()`. Clippy reports nothing.
  - The assembly integration was reviewed. This gets picked up when field prime is set to BLS constant; it is not picked up in `bls-signatures` (as this works with six limbs), but rather in `bellman`. This was also confirmed by perturbing the code and re-running tests.

  - All code was reviewed in detail.
- The `group` repository was compared to the Zcash fork source, as well as reviewed holistically. Minor deltas were observed related to naming and the `GroupDecodingError` enum. Code revolves around trait definitions for `CurveProjective`, `CurveAffine` and `EncodedPoint`.
  - Minimal logic is present, all code was reviewed in detail.
- The `pairing` repository was compared to the Zcash fork source, as well as reviewed holistically. This code has significant changes, with key items including hash to curve/field, `ec` refactoring, cofactor clearing, some serialization/deserialization helper functions, and supporting test cases. Large amount of code and changes are present.
  - Regarding section 3 of `draft-irtf-cfrg-hash-to-curve-07`, the function `encode_to_curve()` on line 39 of `pairing/src/hash_to_curve.rs` should be 'dead code'. Line 40 of `bls-signatures/src/signature.rs` calls the correct function. Note that commenting out a line of each function causes `pairing` tests to fail, but not `bellman` tests (as expected).
  - The `osswu_map()` on line 835 of `pairing/src/bls12_381/ec/g2.rs` corresponds to section 6.6.3, appendix C.3 and appendix D.2.1 (of aforementioned document) but the code does not match the algorithm. However, it does match https://github.com/algorand/pairing-plus/blob/master/src/bls12_381/osswu_map/g2.rs.
  - Similar code from Algorand is at https://github.com/algorand/pairing-plus. See above reference to "Fast and simple constant-time hashing to the BLS12-381 elliptic curve".
  - Note that the spec implements a method to clear the cofactor as described by Budroni and Pintore. However, the code implements a different algorithm where the comment notes "...used to clear cofactor compatibly with Budroni-Pintore GLV-based method". See the reference to "Efficient hash maps to G2 on BLS curves". The function `chain_h2_eff()` implements a mul/add chain.
  - Note that the test data on line 1737 of `pairing/src/bls12_381/ec/g2.rs` matches that in Appendix G.10.1 on page 147 of the draft specification (expected matches P.x and P.y). So we know that the input/output test vector correspondence is solid.
  - XNUM and associated constants do not match the spec. This may be due to Jacobian coordinates but the missing heritage needs further investigation.
- The `rust-sha2ni` repository was compared to the fork RustCrypto source, as well as reviewed holistically. Additionally, the `sha256_intrinsics.rs` source file was compared with the reference C implementation at https://github.com/noloader/SHA-Intrinsics/blob/master/sha256-x86.c. The objective of the repository is to deliver SHA2 functionality using x86 'intrinsics' when available and a generic implementation otherwise. It appears to have redundant functionality - namely, the source file `sha256_intrinsics.rs`, as well as pulling from `sha2_asm` crate.
  - Minimal new logic is present, all code was reviewed in detail.
  - Given the number of different SHA2 implementations, and as discussed with the Protocol Labs Team, it should be ensured that the correct ones are being used.
- The `bls-signatures` was compared to the draft BLS signature specification https://tools.ietf.org/pdf/draft-irtf-cfrg-bls-signature-02.pdf as well as reviewed holistically. The implementation does not strictly follow the draft reference. Namely, the functions defined in the spec are not implemented as is. In order to assess compliance with the RFC, we identified the following requirements and studied their conformance.

    2.2.: H, a hash function that MUST be a secure cryptographic hash function, e.g., SHA-256. For security, H MUST output at least ceil(log2(r)) bits, where r is the order of the subgroups G1 and G2 defined by the pairing-friendly elliptic curve.

  - Fulfilled; Conforming hash function used.

    2.2.: When the signature variant is minimal-signature-size, this function (i.e., hash_to_point) MUST output a point in G1. When the signature variant is minimal-pubkey size, this function MUST output a point in G2. For security, this function MUST be either a random oracle encoding or a nonuniform encoding, as defined in `draft-irtf-cfrg-hash-to-curve-07`.

  - Fulfilled; `minimal-pubkey-size` variant is used (namely, public keys are points in G1). The function outputs a point in G2 conforming to the draft specification, as can be seen in the following excerpt from `bls-signatures/src/signature.rs`.

```
pub fn hash(msg: &[u8]) -> G2 {
    <G2 as HashToCurve<ExpandMsgXmd<sha2ni::Sha256>>>::hash_to_curve(msg, CSUITE)
}
```

2.3.: KeyGen uses HKDF instantiated with the hash function H. For security, IKM MUST be infeasible to guess, e.g., generated by a trusted source of randomness. IKM MUST be at least 32 bytes long, but it MAY be longer.

- Partially Fulfilled; See finding NCC-PRLB007-007 on page 18 for discussions around the use of the HKDF. Additionally, it should be noted that the possibility exists to generate a key directly from bytes. This should be used only for testing, or provided a good source of randomness, as per the draft RFC.

```
/// Generate a deterministic private key from the given bytes.
///
/// They must be at least 32 bytes long to be secure.
pub fn new<T: AsRef<[u8]>>(msg: T) -> Self {
    PrivateKey(key_gen(msg))
}
```

2.4.: SK MUST be indistinguishable from uniformly random modulo r (Section 2.2) and infeasible to guess, e.g., generated using a trusted source of randomness.

- Fulfilled if using the `generate` function; Additionally, finding NCC-PRLB007-004 on page 11 describes some shortcomings around the usage of `fill_bytes`.

```
/// Generate a new private key
pub fn generate<R: RngCore>(rng: &mut R) -> Self {
    // IKM must be at least 32 bytes long:
    // https://tools.ietf.org/html/draft-irtf-cfrg-bls-signature-00#section-2.3
    let mut ikm = [0u8; 32];
    rng.fill_bytes(&mut ikm);

    Self::new(ikm)
}
```

3.3.: All public keys used by Verify, AggregateVerify, and FastAggregateVerify MUST be accompanied by a proof of possession, and the result of evaluating PopVerify on the public key and proof MUST be VALID.

- Not applicable since this variant is not implemented.

3.3.1.: For security, this function (i.e., `hash_pubkey_to_point`) MUST be domain separated from the hash_to_point function. In addition, this function MUST be either a random oracle encoding or a nonuniform encoding, as defined in `draft-irtf-cfrg-hash-to-curve-07`.

- Not applicable since this variant is not implemented.

4.1.: SC_TAG is a string indicating the scheme and, optionally, additional information. The first three characters of this string MUST chosen as follows: "NUL" if SC is basic, ...

- Fulfilled; the ciphersuite (`CSUITE`) is defined as `BLS_SIG_BLS12381G2_XMD:SHA-256_SSWU_RO_NUL_` in `signature.rs`.

4.1.: hash_to_point: a hash from arbitrary strings to elliptic curve points. hash_to_point MUST be defined in terms of a hash-to-curve suite `draft-irtf-cfrg-hash-to-curve-07`.

- Fulfilled; discussed above.

4.1.: hash_pubkey_to_point MUST be defined in terms of a hash-to-curve suite `draft-irtf-cfrg-hash-to-curve-07`. The hash-to-curve domain separation tag MUST be distinct from the domain separation tag used for hash_to_point.

- Not applicable in basic variant

5.1.: All algorithms in Section 2 and Section 3 that operate on public keys require first validating those keys. For the basic and message augmentation schemes, the use of KeyValidate is REQUIRED.

- Partially fulfilled; while the function `KeyValidate` is not explicitly defined, this validation is enforced during creation of the different objects. For example, the trait `CurveProjective` in `group/src/lib.rs` is defined as follows:

```
/// Projective representation of an elliptic curve point guaranteed to be
/// in the correct prime order subgroup.
pub trait CurveProjective:
```

5.2.: This check (i.e., the signature_subgroup_check invocation in CoreVerify) is REQUIRED of conforming implementations.

- Partially fulfilled; while the function `signature_subgroup_check` is not explicitly defined, this validation is enforced during creation of the different objects.

  5.4.: The IKM input to KeyGen MUST be infeasible to guess and MUST be kept secret. Secret keys MAY be generated using other methods; in this case they MUST be infeasible to guess and MUST be indistinguishable from uniformly random modulo r.

- Fulfilled; see discussion above.

  5.5.: The security analysis models hash_to_point and hash_pubkey_to_point as random oracles. It is crucial that these functions are implemented using a cryptographically secure hash function. For this purpose, implementations MUST meet the requirements of `draft-irtf-cfrg-hash-to-curve-07`. In addition, ciphersuites MUST specify unique domain separation tags for hash_to_point and hash_pubkey_to_point. The domain separation tag format used in Section 4 is the RECOMMENDED one.

- Fulfilled; since `hash_pubkey_to_point` is not used in the basic variant and the tag used is the ciphersuite.

- The `bellman` repository was compared to the Zcash fork source, as well as reviewed holistically.
  - Key changes revolve around gpu, particularly around `fft`, `multiexp` and some `groth`.
  - Parallelism/build options were implemented: rayon, crossbeam, futures-cpupool, and ocl.
  - Relatively more (necessary) unsafe logic.
  - Note that `.circleci/config.yaml` picks up a Ubuntu 16.04 image.
  - The `'% 32'` on lines 153 and 171 of `src/gadgets/uint32.rs` is nice in that it does not generate an error, and is not-nice for the same reason. Trivia.
  - The `temp_path()` function on lines 8-12 of `src/gpu/locks.rs` would benefit from a check to prevent directory traversal, but this function is just a trivial helper at the moment.

5. Additional Spot Checks

In addition to the different considerations listed above, the NCC Group team also paid particular attention to specific areas traditionally known to be sources of security vulnerabilities. These areas are the following:

- Randomness generation, error detection and handling
- Deserialization (particularly length checks) and opportunities to tamper with the data
- Public parameters review (mostly around BLS12-381) and key generation steps
- Code dependencies (particularly around outdated dependencies)
- Non-constant time algorithms and the potential introduction of oracles (that could be leveraged for side-channel attacks)