BLST Cryptographic Implementation Review

Ethereum Foundation

November 17, 2020 – Version 1.0

Prepared for

Andy Polyakov
Kelly Olson
Sean Gulley
Simon Peffers

Prepared by

Eric Schorn
Thomas Pornin

Executive Summary

Synopsis

In October 2020, Supranational, Protocol Labs and the Ethereum Foundation engaged NCC Group's Cryptography Services team to conduct a cryptographic implementation review of the BLST library. This library implements support for the draft IETF specifications on Hashing to Elliptic Curves and BLS Signatures. The latter specification uses advanced cryptographic-pairing operations to feature aggregation properties for secret keys, public keys and signatures. This functionality is central to the emerging Ethereum 2.0 Proof-of-Stake block-validation mechanism. Full source code and excellent support was provided. The project was delivered by 2 consultants within 23 person-days of effort.

Scope

NCC Group's evaluation included the following source code repository and specifications:

- Commit `414ac6b` of github.com/supranational/blst
  - C source code, Assembly source code for multiple target architectures.
  - Rust and Golang bindings; Other language bindings were out of scope.
- IETF CFRG draft for BLS Signatures v4
- IETF CFRG draft for Hashing to Elliptic Curves v10

Limitations

While the BLST library largely contains sufficient functionality to support the target specifications, the API does not fully correlate to the specification. In some cases a calling application is expected to combine lower-level primitives to perform a particular function, such as the public key subgroup check, which demonstrates library flexibility but also moves relevant functionality outside of the library. As the code is minimally documented, fully matching intent with implementation can be subjective in a few places. Nonetheless, robust coverage of all in-scope code was achieved.

Key Findings

The BLST library demonstrates leadership in high performance through an optimized implementation in C and Assembly, and leadership in broad usability through multiple language, compiler and architecture targets. However, the review uncovered a number of flaws falling into several broad categories:

- Insufficient validation constraints that need to be updated to the latest version of the BLS Signatures specification, which risks interoperability issues.
- Exporting far more functionality and library internals to the calling application than necessary, which increases application complexity, risk of library misuse and impact of changes.
- Missing and/or unevenly tested functionality for lesser used aspects of the target specifications outside of the Ethereum 2.0 usage profile.
- Issues involving modular reduction, point multiplication and clearing of secrets from memory.

Partial Retest Results

The latter portion of the project involved retesting fixes for selected findings. The detailed entry for each of the addressed findings now includes a brief description of retest observations with results summarized in the Table of Findings on page 4. A summary of the final position is as follows:

- There were no critical or high severity findings.
- Initially, there were 3 medium severity findings reported. Subsequently, 2 were fixed and 1 categorized as 'risk accepted'. This leaves 0 open.
- Initially, there were 14 low severity findings reported. Subsequently, 9 were fixed, 3 categorized as 'risk accepted' and 1 partially fixed. This leaves 2 open.
- Initially, there were 3 informational observations reported. Subsequently, 2 were addressed. These are not considered security weaknesses.

Strategic Recommendations

The in-scope code is well structured, highly optimized, performant and demonstrates careful attention to detail. Beyond addressing the reported findings, NCC Group recommends prioritizing the following areas for future development:

- Revisit the API to present the simplest possible interface that tightly corresponds to the target specifications. Additional performance enhancements can build upon this foundation.
- Move all possible functionality from the bindings into the C source to maximize consistency across languages and usefulness to C users. The bindings would then largely contain language-specific validation and threading support.
- Enforce strict validation upon deserialization to externally-opaque data structures to prevent the library from operating on invalid application data.
- Develop robust documentation to maximize developer productivity, application correctness and library uptake.

Dashboard

Target Metadata

Name	BLST Signature Library for BLS12-381
Type	Source code
Platforms	C with Assembly for several architectures; Rust and Golang bindings
Environment	Production commit

Engagement Data

Type	Cryptography Implementation Review
Method	Manual source code analysis
Dates	2020-10-13 to 2020-11-06
Consultants	2
Level of Effort	23 person-days

Targets

BLST Code Repository https://github.com/supranational/blst/tree/414ac6b185f6b2ef2e6364d5716f915af966c465

Finding Breakdown

Critical issues	0
High issues	0
Medium issues	3
Low issues	14
Informational issues	3
Total issues	**20**

Category Breakdown

Configuration	4
Cryptography	7
Data Exposure	2
Data Validation	7

Component Breakdown

A: BLST Library	2
B: Assembly	2
C: C Source	3
D: Rust Bindings	8
E: Golang Bindings	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 C on page 50.

A: BLST Library

Title	Status	ID	Risk
Missing `PopProve()` and `PopVerify()` Functions	Reported	015	Low
Uneven Test Coverage of Primary Functionality	Reported	023	Informational

B: Assembly

Title	Status	ID	Risk
Incomplete Modular Reduction (256 bits)	Fixed	006	Low
Miscomputation of Parity in the Field Extension	Fixed	003	Informational

C: C Source

Title	Status	ID	Risk
Missing `SecretKey` Deserialization Bound Check	Fixed	005	Low
Incorrect Point Multiplication Result	Fixed	016	Low
Non Constant-Time Inversion	Fixed	021	Informational

D: Rust Bindings

Title	Status	ID	Risk
Insufficient Input Validation During Deserialization	Fixed	004	Medium
Missing Checks in Aggregate Verify	Fixed	009	Medium
Extraneous C Exports	Risk Accepted	010	Medium
Unspecified Rust Toolchain Version	Risk Accepted	001	Low
Sensitive Information Not Cleared	Fixed	002	Low
Insufficient `PublicKey` Validation	Fixed	007	Low
Incorrect Result for Zero Length Aggregation	Partially Fixed	008	Low
Struct Fields With Extraneous `pub` Access Modifiers	Fixed	017	Low

E: Golang Bindings

Title	Status	ID	Risk
Sensitive Information Not Cleared	Fixed	012	Low
Missing Public `KeyValidate` Function	Fixed	013	Low
Extraneous Type Exports	Risk Accepted	018	Low
Insufficient Public Key Validation on Deserialization	Risk Accepted	019	Low
Missing Checks in Aggregate Verify	Fixed	020	Low

Finding	**Missing `PopProve()` and `PopVerify()` Functions**
Risk	**Low** Impact: Medium, Exploitability: Undetermined
Identifier	NCC-ETHF002-015
Status	Reported
Category	Cryptography
Component	A: BLST Library
Location	Missing functionality in (for example) Rust bindings `lib.rs`.
Impact	An application intending to target a ciphersuite configuration corresponding to the 'Proof of Possession Scheme' is required to implement the missing `PopProve()` and `PopVerify()` functions from other complex (sub)components, with a high potential for mistakes by casual users.
Description	The BLST code repository `README.md` file states broad compliance with the relevant BLS specifications without describing any specific carve-outs. This library is compliant with the following IETF draft specifications: • IETF BLS Signature V4 • IETF Hash-to-Curve V9 The BLS Signatures specification described in Appendix B on page 44 outlines two BLS12-381 ciphersuites[1,2] (and their many subcomponents) that target the 'Proof of Possession Scheme'.[3] This scheme defends against rogue key attacks by using a separate public key validation step, which then enables an optimization to aggregate signature verification where all signatures are on the same message. This scheme requires specific `PopProve()` and `PopVerify()` functions analogous to sign and verify functions respectively. This functionality is missing in the BLST library. While this functionality can be assembled with detailed pairing-specific components (as Ethereum 2.0 does with DepositData[4]), this is a high risk approach for casual users.
Recommendation	Implement the `PopProve()` and `PopVerify()` functions per the specification. It may be possible to include an optional data field input such that the Ethereum 2.0 case is simply handled in BLST.
Client Response	The functions will be added to the library to complete the API. Currently this functionality is being performed by the application if needed.

[1] `BLS_SIG_BLS12381G1_XMD:SHA-256_SSWU_RO_POP_`
[2] `BLS_SIG_BLS12381G2_XMD:SHA-256_SSWU_RO_POP_`
[3] https://tools.ietf.org/html/draft-irtf-cfrg-bls-signature-04#section-3.3
[4] https://github.com/ethereum/eth2.0-specs/blob/33cfcc4eb34c3cce313adeea9cdb5bc0a4447a89/specs/phase0/beacon-chain.md#depositdata

Finding	**Uneven Test Coverage of Primary Functionality**
Risk	**Informational** Impact: High, Exploitability: Undetermined
Identifier	NCC-ETHF002-023
Status	Reported
Category	Cryptography
Component	A: BLST Library
Location	Absence from C, Go and Rust tests
Impact	Uneven testing of library functionality may not detect existing bugs nor prevent changes from introducing new bugs that unexpectedly break applications.
Description	The BLST library test coverage is uneven. While even the smallest of algorithm-related errors typically cause obviously-detected incorrect results in basic tests of cryptographic code, best practices require a robust testing suite. This entails public vectors for all supported configurations and tests that specifically target validation corner cases alongside the algorithm-specific tests. This ensures that any flaws introduced by future development are (more likely) caught prior to library deployment in applications. Relying on application users to drive test coverage is not sufficient. There are no/minimal test cases for the C and Assembly source code, so the library is relying on application users to drive coverage in this context. Regarding the hash-to-curve specification, the Golang bindings utilize JSON test vectors for the `BLS12381G[1|2]_XMD_SHA-256_SSWU_RO_` ciphersuites, but not for the two `BLS12381G[1|2]_XMD:SHA-256_SSWU_NU_` ciphersuites. These two missing cases correspond to `encode_to_curve()` primary functionality.[5] The existing Golang test case was adapted to incorporate these latter two vectors[6] and ran successfully, so the functionality is indeed present. The Rust bindings do not contain any of the above hash-to-curve JSON test vectors or test functionality. As the most recent BLS Signatures specification contains new validation constraints, such as `1 <= SK`,[7] the presence of finding NCC-ETHF002-005 on page 12 also indicates missing test coverage.
Recommendation	Implement additional testing to specifically cover the following cases from each language binding: • Each BLS12-381 ciphersuite in the Hashing to Elliptic Curves specification • Each validation constraint in the BLS Signatures specification • Public BLS12-381 test vectors when they become available (the specification currently notes "TBA") • Both successful and unsuccessful paths through primary functionality. This is largely present but mentioned for completeness
Client Response	More test vectors will be added to Go and the JSON test vectors will be moved up a directory in order to eventually be shared with Rust.

[5] https://tools.ietf.org/html/draft-irtf-cfrg-hash-to-curve-10#section-8.8.1
[6] https://github.com/cfrg/draft-irtf-cfrg-hash-to-curve/tree/master/poc/vectors
[7] https://tools.ietf.org/html/draft-irtf-cfrg-bls-signature-04#section-2.4

Finding **Incomplete Modular Reduction (256 bits)**

Risk **Low** Impact: Medium, Exploitability: None

Identifier NCC-ETHF002-006

Status Fixed

Category Cryptography

Component B: Assembly

Location `src/asm/mulq_mont_256-x86_64.pl`, line 518
`src/asm/mulx_mont_256-x86-64.pl`, line 371
`src/asm/mul_mont_256-armv8.pl`, line 330

Impact The functions that reduce a 512-bit input modulo r may return values which are not lower than r. In the current code, these functions are called only as part of key pair generation, and the out-of-spec cases happen to be fixed as a side-effect of the Montgomery multiplication that immediately follows.

Description The `redc_mont_256()` function takes as inputs a 512-bit integer x and an odd modulus r (with $r \leq 2^{256} - 2^{192} - 1$), and return $x/2^{256}$ mod r. That function is provided in assembly code for 64-bit x86 (`src/asm/mulq_mont_256-x86_64.pl`) and ARMv8 (`src/asm/mul_mont_256-armv8.pl`). The `redcx_mont_256()` function has the same API and replaces it on 64-bit x86 platforms where the BMI2 (`mulx`) and ADX (`adcx`, `adox`) opcodes are available. These three implementations work as follows:

- Input x is split into its high half $x_1 = \lfloor x/2^{256} \rfloor$ and low half $x_0 = x$ mod 2^{256}.
- Montgomery reduction is applied on x_0, to compute $z = x_0/2^{256}$ mod r.
- $z + x_1$ is computed, then r is subtracted from that sum if it is not already lower than r.

Montgomery reduction of x_0 adds kr to x_0 for a 256-bit integer such that $x_0 + kr$ is a multiple of 2^{256}. Since r is odd, it is invertible modulo 2^{256}, and k is uniquely defined. The value $x_0 + kr$ is then divided by 2^{256} over the integers (the division is exact in that case). The internal `__mulq_by_1_mont_256()` (or `__mulx_by_1_mont_256()` or `__mul_by_1_mont_256()`) function performs this operation. Note that the result z of Montgomery reduction is not necessarily fully reduced modulo r: a value $z = r$ is possible (with $r \approx 2^{254.86}$ the order of the BLS-381 curve subgroup, this happens only if $k = 2^{256} - 1$ or $k = 2^{256} - 2$, which in turn happens for exactly two possible values of x_0).

The returned value $z + x_1$ is equal to $x/2^{256}$ modulo r, but it is not necessarily reduced. Indeed, x_1 may range up to $2^{256} - 1$. The single conditional subtraction of r is not enough to bring the result down to the $0...r - 1$ range; all values up to $2^{256} - 1$ may be returned. Therefore, `redc_mont_256()` and `redcx_mont_256()` may return values that do not comply with the expectations of other functions that perform computations with scalars modulo r.

In practice, the `redc_mont_256()` (or `redcx_mont_256()`) function is called only as part of key pair generation from the `blst_keygen()` function (`src/keygen.c`, line 158) and is immediately followed by a Montgomery multiplication by the `BLS12_381_rRR` constant:

```
redc_mont_256(scratch.key, scratch.key, BLS12_381_r, r0);
mul_mont_sparse_256(scratch.key, scratch.key, BLS12_381_rRR,
                    BLS12_381_r, r0);
```

If the reduction result (`scratch.key`) is not lower than r, then this call is nominally invalid.

However, on 64-bit x86, `mul_mont_sparse_256()` performs Montgomery multiplication by limbs of 64 bits, with reduction rounds delayed by one iteration. This relies on the internal intermediate result to fit on 6 limbs (384 bits). Specifically, each iteration starts with a 5-limb intermediate value, to which is added (with a one-limb shift) the product of the first multiplication operand by one limb of the second operand, and then a multiple of the modulus by another limb-sized value. Since the numerically largest limb of `BLS12_381_rRR` is $m =$ `0xC999E990F3F29C6D`, this sum cannot exceed $2^{320} - 1 + 2^{64}m(2^{256} - 1) + (2^{64} - 1)r$, which happens to be lower than $2^{384} - 1$, which fits the expectations of the algorithm.

On ARMv8, `mul_mont_sparse_256()` is organized slightly differently (there is no one-round delay in application of the reduction), but the same overall conclusion applies: in the specific case of key pair generation in BLST, the out-of-range values that `redc_mont_256()` may return are properly absorbed by the subsequent Montgomery multiplication, yielding a fully reduced result.

Recommendation

Having functions that return out-of-spec values is fragile, since it relies on other functions to tolerate such inputs; this may break whenever the implementation of Montgomery multiplication (`mul_mont_sparse_256()`) is modified.

In the specific case of the subgroup of the BLS-381 curve, the result computed by `redc_mont_256()` could be fully reduced modulo r by performing two extra conditional subtractions of r (since $3r > 2^{256}$). These extra operations should have almost negligible cost compared with the rest of the reduction cost.

If the function is supposed to be usable with much shorter moduli r, many more conditional subtractions of r may be needed, leading to a prohibitive cost. Instead, Montgomery reduction may be applied to x_1 as well, and applied again to x_0, to ensure that both values are in an appropriate range; this would also require an extra conditional subtraction (to make sure that at least one of x_0 or x_1 is strictly lower than r), and the constant for the final multiplication would need to be adjusted to account for the extra Montgomery reductions (i.e. it should be equal to 2^{768} mod r instead of 2^{512} mod r). If such a full reduction is not implemented, then the exact range of acceptable moduli for `redc_mont_256()` should be properly documented to avoid misuse.

Retest Results

NCC Group reviewed changes made in the updated `710a5ea` commit,[8] and observed new code comments related to the exact range of acceptable moduli for `redc_mont_256()` per the recommendation. As such, this finding has been marked as 'Fixed'.

Client Response

The `redc_mont_256()` function is considered to be an internal function, to be used only in conjunction with other steps such as `mul_mont_sparse_256()` to ensure full reduction. This specific constraint on function usage and expectations is documented with explicit comments. See: https://github.com/supranational/blst/commit/710a5ea353ba2106a7a92b34d6a643bff66d7d62

[8] https://github.com/supranational/blst/commit/710a5ea353ba2106a7a92b34d6a643bff66d7d62

Finding	**Miscomputation of Parity in the Field Extension**
Risk	**Informational** Impact: Low, Exploitability: None
Identifier	NCC-ETHF002-003
Status	Fixed
Category	Cryptography
Component	B: Assembly
Location	`src/asm/add_mod_384-x86_64.pl`, lines 1256-1262
Impact	The miscomputed bit is not used in BLST; therefore, there is no immediate security consequence.

Description

The `sgn_pty0_mod_384x()` function computes the *sign* and *parity* of an element of the field $\mathbb{F}_{q^2}$. An element $y \in \mathbb{F}_{q^2}$ can be uniquely written as $y = a + ib$ where a and b are elements of $\mathbb{F}_q$, and i is the conventional square root of -1 in $\mathbb{F}_{q^2}$. Values a and b are the *real part* and *imaginary part* of y, respectively.

The *sign* of y is used for serializing curve points to bytes (in compressed format), while the *parity* is used for hashing arbitrary data into a curve point. Both are used for the same mathematical functionality (disambiguation between the two square roots of a given field element) but are defined differently for historical reasons:

- The sign of y is 1 if its imaginary part (b), as an integer in the $0...q-1$ range, is greater than $(q-1)/2$, or `0` otherwise. If the imaginary part of y is $b = 0$, then the real part (a) is used instead.
- The parity of y is the least significant bit of the real part (a), when represented as an integer in the $0...q-1$ range. If the real part of y is $a = 0$, then the imaginary part (b) is used instead.

The `sgn_pty0_mod_384x()` function first computes the sign and parity of the real part of the input y, then proceeds to compute the same values for the imaginary part. A few instructions then select the proper bits to pack them into the returned value:

```
        not     $r_ptr                  # 2*x > p, which means "negative"

        test    @acc[0], @acc[0]
        cmovnz  $r_ptr, %rax            # a->im!=0? sgn0(a->im) : sgn0(a->re)

        test    @acc[6], @acc[6]
        cmovz   $r_ptr, @acc[7]         # a->re==0? prty(a->im) : prty(a->re)

        and     \$1, @acc[7]
        and     \$2, %rax
        or      @acc[7], %rax           # pack sign and parity
```

In the conventions of the assembly code generator Perl script:

- `@acc[0]` contains zero if and only if the imaginary part of the input y is zero
- `rax` contains the packed sign and parity of the real part of y (bit `0` is the parity, bit 1 is the sign)
- `@acc[6]` contains zero if and only if the real part of y is zero
- `@acc[7]` contains the least significant limb of the imaginary part of y
- `$r_ptr` contains the negation of the sign of the imaginary part of y (i.e. -1 or `0`)

Thus, in the event that the input value y has a real part equal to zero (`@acc[6]` contains zero), then `@acc[7]` is overwritten by the `cmovz` opcode with a copy of `$r_ptr`; its least significant bit, which is then used in the returned value as the parity of y, is then equal to the least significant bit of `$r_ptr`, which is the sign of y, not its parity.

Therefore, `sgn_pty0_mod_384x()` returns the wrong parity for about half of the values y whose real part is zero.

There is no immediate security consequence on BLS signatures, for the two following reasons:

- The parity is used only as part of hashing to a curve point, in which case the value whose parity is computed is obtained as the output of a one-way hash function, and finding an input to that hash function such that the parity is evaluated on a field element whose real part is zero is computationally infeasible (it would require breaking the preimage resistance of the hash function).
- When the parity of an element of $\mathbb{F}_{q^2}$ is used (as part of hashing into group G2, in `src/map_to_g2.c`, function `map_to_isogenous_E2()`), the called function is not `sgn_pty0_mod_384x()`, but one of `sgn_pty0_mont_384x()` or `sgn_pty0x_mont_384x()`, which work with inputs in the Montgomery domain, and do not have the issue described here.

Recommendation The sequence above can be fixed by using the following instead:

```
        not     $r_ptr                  # 2*x > p, which means "negative"

        test    @acc[6], @acc[6]
        cmovnz  %rax, @acc[7]           # a->re!=0? prty(a->re) : prty(a->im)

        test    @acc[0], @acc[0]
        cmovnz  $r_ptr, %rax            # a->im!=0? sgn0(a->im) : sgn0(a->re)

        and     \$1, @acc[7]
        and     \$2, %rax
        or      @acc[7], %rax           # pack sign and parity
```

Retest Results NCC Group reviewed changes made in the updated `668f17b` commit.[9] Code changes in the `src/asm/add_mod_384-x86_64.pl` file similar to those described above were observed per the recommendation. As such, this finding has been marked as 'Fixed'.

[9] https://github.com/supranational/blst/commit/668f17b664e285b02b8158ecb37adc36a8d553ff

Finding **Missing `SecretKey` Deserialization Bound Check**

Risk **Low** Impact: Medium, Exploitability: Low

Identifier NCC-ETHF002-005

Status Fixed

Category Data Validation

Component C: C Source

Location Lines 410-419 of blst/src/exports.c

Impact Allowing the deserialization of a zero `SecretKey` violates the (recently changed) BLS-signature specification and may cause application interoperability issues involving the 'zero signature'.

Description The `deserialize()` function implemented in the Rust bindings `lib.rs`[10] performs a `Fr` check via the `blst_scalar_fr_check()` function found in the C source. This latter function is implemented on lines 410-419 of `exports.c` as shown below.

```
410 limb_t blst_scalar_fr_check(const pow256 a)
411 {
412     vec256 value, zero = { 0 };
413
414     limbs_from_le_bytes(value, a, 32);
415     add_mod_256(zero, zero, value, BLS12_381_r);
416     return vec_is_equal(zero, value, sizeof(zero));
417     vec_zero(zero, sizeof(zero));
418     vec_zero(value, sizeof(value));
419 }
```

The above function unpacks the input into `value` limbs on line 414 and then adds `0` mod `BLS12_381_r` on line 415. If the result is unchanged per the check on line 416, the value is less than `Fr` and a success indicator is returned. Note that lines 417-418 are unreachable as shown – this was addressed in the subsequent commit `c08a9ae`.

This approach is consistent with the output constraints required from the KeyGen algorithm as defined in section 2.3 of `draft-irtf-cfrg-bls-signature-02`.[11]

```
- SK, a uniformly random integer such that 0 <= SK < r.
```

However, in the latest version `draft-irtf-cfrg-bls-signature-04`[12,13] the output constraint has been expanded to disallow `0`.

```
- SK, a uniformly random integer such that 1 <= SK < r.
```

The code does not implement the expanded check for `0` present in the latest specification. An attacker able to deserialize a zero secret key may be able to trigger (likely detected) downstream problems involving the 'zero signature', which has been a subject of recent debate.[14]

Recommendation Add a `!vec_is_equal` check against the `zero` value. This does not necessarily need to be in

[10] https://github.com/supranational/blst/blob/414ac6b185f6b2ef2e6364d5716f915af966c465/bindings/rust/src/lib.rs#L377
[11] https://tools.ietf.org/html/draft-irtf-cfrg-bls-signature-02#section-2.3
[12] https://tools.ietf.org/html/draft-irtf-cfrg-bls-signature-04#section-2.3
[13] https://github.com/ethereum/eth2.0-specs/issues/2072
[14] https://github.com/status-im/nimbus-eth2/issues/555

the `blst_scalar_fr_check()` function shown above, but instead in the deserialization and keyGen paths in C (rather than bindings).

Retest Results

NCC Group reviewed changes made in the updated `15f6383` commit.[15] Code changes in the `bindings/go/blst.go` source file now utilize `blst_sk_check()` check, which is exported by the C source (though implemented in assembly) and checks against the zero case. As such, this finding has been marked as 'Fixed'.

[15] https://github.com/supranational/blst/commit/15f63834dad3a3d9588574b302669f34d9f252bc

Finding	**Incorrect Point Multiplication Result**
Risk	**Low** Impact: Medium, Exploitability: None
Identifier	NCC-ETHF002-016
Status	Fixed
Category	Cryptography
Component	C: C Source
Location	`src/ec_mult.h`, line 165
Impact	In the context of BLS signatures, signature generation and public key derivation from the private key fail for a specific private key value. Since it is very improbable that a properly generated private key has that value, the impact is mostly negligible.
Description	The `POINTonE1_mult_w5()` and `POINTonE2_mult_w5()` functions compute products of a curve point by a scalar, for points with coordinates in the base field $\mathbb{F}_q$ or the extended field $\mathbb{F}_{q^2}$, respectively. The scalar is an integer modulo the prime r. These functions assume that the point which is multiplied is in the proper subgroup of order r, and that the scalar is fully reduced. The implementation of both functions is provided by the `POINT_MULT_SCALAR_WX_IMPL` macro, which uses a classic double-and-add algorithm, with window optimizations (here, with 5-bit windows) to avoid most of the point additions. Booth recoding is applied to the scalar: • For w-bit windows, a scalar value k is written as $k = \sum_{i=0}^{d-1} k_i 2^{wi}$ with each k_i in the range $-(2^{w-1} - 1) ... + 2^{w-1}$. The number of digits is $d = \lceil (n+1)/w \rceil$, where n is the maximum size of the scalar, expressed in bits (normally equal to the size of r). The top digit k_{d-1} is nonnegative. • For the base point P, the window is filled with points jP for $j = 1$ to 2^{w-1} (inclusive). • The algorithm operates over an accumulator point Q, whose initial value is $k_{d-1}P$ (which is either the point-at-infinity, if $k_{d-1} = 0$, or one of the window points). Then, each iteration consists in first multiplying Q by 2^w (i.e. applying w successive point doublings), then adding $k_i P$, which is either a point from the window (if $k_i \geq 0$), or the negation of a point of the window (if $k_i < 0$), the latter being inexpensive to compute dynamically. Iteration number i ranges from $d - 2$ down to 0. In the BLST implementation, the specialized `POINTinE1_add()` (or `POINTinE2_add()`) function is used. This function uses the classic point addition formulas for short Weierstraß curves in Jacobian coordinates, with cost 11M+5S. Crucially, these formulas have exceptional cases when one of the operands is the point-at-infinity, or when both operands designate the same curve point. The implementation in `POINTinE1add()` handles the former with two constant-time `vec_select()` calls, but not the latter: if the two points to add are the same point T, then the function returns an all-zero point, i.e. a representation of the point-at-infinity, instead of the expected $2T$. Unfortunately, it is possible for the last call to `POINTinE1_add()` (respectively `POINTinE2_add()`) in `POINTinE1_mult_w5()` (respectively `POINTinE2_mult_w5()`) to be in that problematic situation. With the BLS12-381 subgroup order r (of size 255 bits), with 5-bit windows, this happens when the scalar has value exactly $r - 2$. In that case, the `POINTinE1_add()` and `POINTinE2_add()` functions incorrectly return the point-at-infinity for any source point in the subgroup.

It can be shown that when all of the following hold:

- the source point P is in the right subgroup of order r;
- the order r is prime;
- the scalar is fully reduced modulo r (i.e. is in the range 0 to $r - 1$);

then the problematic situation may happen only in the point addition of the last iteration, and for only a single scalar value. Therefore, **in the context of BLS signatures**, this issue is unlikely to be exploitable: all uses of that multiplication function use the private key as a scalar, and the private key is, by definition, not adversarially chosen, and unlikely to be equal to the unique scalar that triggers the bug.

Recommendation

Since the problem may occur only on the last iteration, it is sufficient to call the generic addition function (`POINTinE1_dadd()`, `POINTinE2_dadd()`) for that last iteration, which would incur only negligible overhead compared to the whole point multiplication cost.

In full generality, this is not sufficient if the function is to be used on points whose order is not prime, or with scalars which are not fully reduced. If such extended, generic functionality is needed, then all point addition calls should use a complete routine that does not have exceptional cases, such as the `POINTinE1_dadd()` function.

Retest Results

NCC Group reviewed changes made in the updated `f17999f` commit.[16] Code changes in the `src/ec_mult.h` file causing `POINTinE2_dadd()` to be called for the last iteration were observed per the recommendation. As such, this finding has been marked as 'Fixed'.

[16] https://github.com/supranational/blst/commit/f17999fcdf29b33bfbb6b0313a36e5fd41d38b49

Finding **Non Constant-Time Inversion**

Risk **Informational** Impact: High, Exploitability: None

Identifier NCC-ETHF002-021

Status Fixed

Category Cryptography

Component C: C Source

Location
- Lines 533-548 of blst/src/fp12_tower.c

Impact Non constant-time code operating on secret data risks the potential for exposure through timing side-channels.

Description The `inverse_fp2()` function as implemented in the `fp12_tower.c` source file is shown below. The comment on lines 541-542 pertains to the following non constant-time `eucl_inverse_fp()` function, and highlights an assumption of only operating on public data.

```
533 static void inverse_fp2(vec384x ret, const vec384x a)
534 {
535     vec384 t0, t1;
536
537     /* 1/(a0^2 + a1^2) */
538     sqr_fp(t0, a[0]);
539     sqr_fp(t1, a[1]);
540     add_fp(t0, t0, t1);
541     /* It's assumed that "higher-dimension" operations are performed
542      * on public data, hence no requirement for constant-time-ness. */
543     eucl_inverse_fp(t1, t0);
544
545     mul_fp(ret[0], a[0], t1);
546 ...
```

When deriving a $\mathbb{G}_2$ public key `PK` from a secret key `SK` via `skToPk()`,[17] it is true that the inversion happens at the end of the process on essentially the 'public Z coordinate'. However, while the resulting `PK` point is meant to be public, its Z coordinate does contain traces of private operations, i.e., different `SK` values will translate into different Z values. Thus, there is the potential for side-channel leakage of secret information via timing differences[18] in `eucl_inverse_fp`. Since `skToPk()` is not a publicly-observable repeated operation, in this case the risk is minimal.

The `min-pk` configuration's signing process `CoreSign()`[19] function may very well be a publicly-observable repeated operation of the above nature, and thus of higher risk. Proof-of-stake users in this configuration emit signatures under a high-value key on a regular basis. Note that Ethereum 2 operates in the `min-sig` configuration.

In BLST, the `inverse_fp2()` appears to be used strictly inside tower operations rather than curve operations involving $\mathbb{G}_2$. As such, this finding has been marked 'Informational'.

Recommendation Review the usage of `inverse_fp2()` to confirm that it is not used within curve operations. If it is used within curve operations, there are at least two paths described below for mitigation.

[17] https://tools.ietf.org/html/draft-irtf-cfrg-bls-signature-04#section-2.4
[18] Section 3, Projective coordinates leak: https://tches.iacr.org/index.php/TCHES/article/view/8596/8163
[19] https://tools.ietf.org/html/draft-irtf-cfrg-bls-signature-04#section-2.6

Preferably, implement a constant-time inversion for $\mathbb{F}_{p^2}$ utilizing Fermat's Little Theorem or another suitable algorithm.

Alternatively, mask the operand with randomness – multiply by a non-zero random value `t`, invert the product, and multiply that result again by `t`. Given a random and uniformly distributed `t`, the inversion of the product will not reveal anything to an attacker provided the multiplications themselves are constant-time.[20] This will require a random number generator.

Retest Results NCC Group has confirmed that Supranational has reviewed the usage of `inverse_fp2()` and confirms that it is not used within curve operations. As such, this finding has been marked as 'Fixed'.

Client Response Inversion functionality is being reworked for performance and to be constant time everywhere.

[20] https://botan.randombit.net/handbook/side_channels.html#rsa

Finding **Insufficient Input Validation During Deserialization**

Risk **Medium** Impact: High, Exploitability: Medium

Identifier NCC-ETHF002-004

Status Fixed

Category Data Validation

Component D: Rust Bindings

Location
- Lines 373-382 and 388-390 of blst/blob/bindings/rust/src/lib.rs
- Lines 473 and 879 of blst/blob/bindings/rust/src/lib.rs

Impact Attempting to deserialize undersized input may cause an unsafe out-of-bounds read, while oversized input will result in the excess data to be ignored. Each case may result in inconsistent behavior that is difficult to debug.

Description As a positive example, in the Rust bindings `lib.rs` source file, the public `serialize()` function shown below returns a byte array of exactly 32 bytes. Also note that `&self` resolves to a fixed-size 32-byte struct.

```
364 pub fn serialize(&self) -> [u8; 32] {
365     let mut sk_out = [0; 32];
366     unsafe {
367         blst_bendian_from_scalar(sk_out.as_mut_ptr(), &self.value);
368     } ....
```

However, the sibling (adjacent) public `deserialize()` function shown below accepts `sk_in` as an arbitrary-sized byte slice. This slice is then passed as a pointer to the C `blst_scalar_from_bendian()` function inside an `unsafe` block. This latter function is implemented in `exports.c`[21] and assumes a 32-byte input.

```
373 pub fn deserialize(sk_in: &[u8]) -> Result<Self, BLST_ERROR> {
374     let mut sk = blst_scalar::default();
375     unsafe {
376         blst_scalar_from_bendian(&mut sk, sk_in.as_ptr());
377         if !blst_scalar_fr_check(&sk) {
378             return Err(BLST_ERROR::BLST_BAD_ENCODING);
379     }   }
380     Ok(Self { value: sk })
381 }
```

As a result, if a BLST library user calls the `deserialize()` function with incorrectly sized input, this input will be passed into unsafe code that does not expect it. Data in excess of 32-bytes in length will be ignored, while undersized data may cause an out-of-bounds read.

Note that the nearby `from_bytes()`[22] function has the same input type declaration and internally utilizes the above `deserialize()` function (and thus has the same issue). Since the corresponding `PublicKey` routines implement a size check, they do not have this issue.

Separately, note that two `from_bytes()` functions containing line 473 and line 879 may at-

[21] https://github.com/supranational/blst/blob/414ac6b185f6b2ef2e6364d5716f915af966c465/src/exports.c#L372-L378

[22] https://github.com/supranational/blst/blob/414ac6b185f6b2ef2e6364d5716f915af966c465/bindings/rust/src/lib.rs#L388-L390

tempt to access the first element of zero-length input before testing its length.

Recommendation Change the `deserialize()` and `from_bytes()` function declarations to require input of type `[u8; 32]`, or perform length check.

Adapt both `from_bytes()` functions on lines 473 and 879 to test for input length before accessing input contents.

Retest Results NCC Group reviewed changes made in the updated `aae0c7d` commit.[23] Code changes in the `bindings/rust/src/lib.rs` file involving a check for `sk_in.len() != 32` were observed per the recommendation. As such, this finding has been marked as 'Fixed'.

[23] https://github.com/supranational/blst/commit/aae0c7d70b799ac269ff5edf29d8191dbd357876

Finding	**Missing Checks in Aggregate Verify**
Risk	**Medium** Impact: Medium, Exploitability: Medium
Identifier	NCC-ETHF002-009
Status	Fixed
Category	Data Validation
Component	D: Rust Bindings
Location	• Lines 589-680 of blst/bindings/rust/src/lib.rs • Lines 703-817 of blst/bindings/rust/src/lib.rs
Impact	A zero-length list of `msgs`/`pks` input provided to the `aggregate_verify()` function does not return `INVALID` as the BLS Signatures specification requires, but instead may cause a panic. Allowing non-distinct messages violates the Message Augmentation Scheme (sub)specification and may allow attacks involving a rogue key.
Description	There are two instances of this finding, each containing the same two aspects.

The first instance involves the `aggregate_verify()` function as implemented on lines 589-680 `lib.rs` which is partially excerpted below.

```
pub fn aggregate_verify(&self, msgs: &[&[u8]], dst: &[u8], pks: &[&PublicKey] )
 ➜  -> BLST_ERROR {
    let n_elems = pks.len();
    if msgs.len() != n_elems {
        return BLST_ERROR::BLST_VERIFY_FAIL;
    }

    // TODO - check msg uniqueness?
    // TODO - since already in object form, any need to subgroup check?

    let pool = da_pool();
    let (tx, rx) = channel();
    let counter = Arc::new(AtomicUsize::new(0));
    let valid = Arc::new(AtomicBool::new(true));

    // Bypass 'lifetime limitations by brute force. It works,
    // because we explicitly join the threads...
    let raw_pks = unsafe {
        transmute::<*const &PublicKey, usize>(pks.as_ptr())
    };
    let raw_msgs =
        unsafe { transmute::<*const &[u8], usize>(msgs.as_ptr()) };
    let dst =
        unsafe { slice::from_raw_parts(dst.as_ptr(), dst.len()) };
...
```

The first aspect relates to the BLS Signature specification requiring that the `aggregate_verify()` function must return `INVALID` when provided with zero-length `msgs`/`pks` lists as input. The function shown above will instead progress down to the three `transmute` statements shown at the bottom without checking for zero-length input (on each of the three parameters). The Rust documentation states[24]:

[24] https://doc.rust-lang.org/std/mem/fn.transmute.html

> `transmute` is **incredibly** unsafe. There are a vast number of ways to cause undefined behavior with this function. `transmute` should be the absolute last resort.

The second aspect is indicated by the first code comment above: the implemented logic does not check for distinct messages as the Message Augmentation Scheme (sub)specification requires.[25]

As an aside, note that the BLS Signature specification section 3.3.4 outlines a `FastAggregateVerify` function that does not require message uniqueness (and in fact only takes a single message). The code implements this function on lines 682-691 where most of its functionality is delegated to the function shown above.

The second instance of the above two aspects involves nearly duplicate code for the `verify_multiple_aggregate_signatures()` function implemented on lines 703-817. This code is also sensitive to zero-length input and non-distinct messages as described above.

Recommendation For each instance, implement checks for zero-length input (`msgs`, `pks`, `dst`). To support the Message Augmentation Scheme, implement checks in each instance for message uniqueness. Consider documenting which function in the specification message schemes map to which function in the code.

Retest Results NCC Group reviewed changes made in the updated `b280835` commit.[26] Code changes in the `bindings/rust/src/lib.rs` file involving a check for zero length input lists (2X) were observed per the recommendation. The calling function must ensure message uniqueness. As such, this finding has been marked as 'Fixed'.

[25] https://tools.ietf.org/html/draft-irtf-cfrg-bls-signature-04#section-3.2
[26] https://github.com/supranational/blst/commit/b280835065fcfb3102ffb82bdde5adcb2975ab7c

Finding	**Extraneous C Exports**
Risk	**Medium** Impact: High, Exploitability: Low
Identifier	NCC-ETHF002-010
Status	Risk Accepted
Category	Configuration
Component	D: Rust Bindings
Location	blst/bindings/rust/src/bindings.rs
Impact	An application may bypass the Rust bindings to incorrectly utilize internal C library functions. The application then risks functionality misuse, overlooked validation checks or misinterpreted returned values which can impact stability, correctness, interoperability and thus consensus. Allowing dependencies upon internal functionality will make library evolution brittle.
Description	The `bindings.rs` source file contains declarations for functions within `blst.h`, such as those shown below. The highlighted `pub` keyword effectively makes all of these functions accessible to a Rust application developer.

```
extern "C" {
    pub fn blst_scalar_from_uint64(out: *mut blst_scalar, a: *const u64);
}
extern "C" {
    pub fn blst_core_verify_pk_in_g1( pk: *const blst_p1_affine, signature:
    ➔  *const blst_p2_affine, hash_or_encode: bool, msg: *const byte, msg_len:
    ➔  usize, DST: *const byte, DST_len: usize, aug: *const byte, aug_len:
    ➔  usize) -> BLST_ERROR;
}
extern "C" {
    pub fn blst_miller_loop(ret: *mut blst_fp12, Q: *const blst_p2_affine, P:
    ➔  *const blst_p1_affine);
}
```

The first internal function shown above is utilized by the Sigma Prime Lighthouse[27] application in a sensitive `verify_signature_sets()` function.

The second function is commonly used in subgroup checks, such as that BLST itself performs in the (public key) `key_validate()` function.[28] Providing this check may lead the application to neglect the 'non-identity point' check required by the latest BLS Signatures specification,[29] as it was not present in an earlier version.[30]

The third function involves the intermediate internals of a pairing calculation and should not be used by an application under any circumstances. Specifying this function (and others like it) will make evolving library internals difficult and brittle.

Recommendation	The Rust bindings should export functions directly matching those in the BLS Signature specification and no others. If some extraneous exports are required for interim third-party logistical reasons, these functions should be prefixed by a term like "unsafe" or "dangerous".

[27] https://github.com/sigp/lighthouse/blob/ncc-october/crypto/bls/src/impls/blst.rs#L71
[28] https://github.com/supranational/blst/blob/414ac6b185f6b2ef2e6364d5716f915af966c465/bindings/rust/src/lib.rs#L406
[29] https://tools.ietf.org/html/draft-irtf-cfrg-bls-signature-04#section-5.1
[30] https://tools.ietf.org/html/draft-irtf-cfrg-bls-signature-02#section-5.1

Client Response	This is a concisous decision, for two core reasons: 1) The underlying structures are exported so that other languages can handle all memory management. 2) Library users have requested these exports due to their high performance for use in other applications such as zk-proof systems. These structures are being formally verified to improve their assurance.

Finding	**Unspecified Rust Toolchain Version**
Risk	**Low** Impact: Medium, Exploitability: Undetermined
Identifier	NCC-ETHF002-001
Status	Risk Accepted
Category	Configuration
Component	D: Rust Bindings
Location	A `rust-toolchain` file should be placed in the `rust` directory alongside the `Cargo.toml` file.
Impact	An unspecified toolchain version may cause divergence in application behavior between developers and users with different environments, as well as allowing silently changing toolchain versions that can introduce consensus instability which is difficult to debug and audit. A specific toolchain version also highlights support expectations.
Description	The Cargo package manager for Rust[31] allows the developer to specify the exact toolchain version to be used via the `rust-toolchain`[32] file. This allows a consistent, known and auditable process for building an application that will reduce the potential for confusion and poor debug visibility. This is particularly important for consensus-oriented projects that are currently undergoing rapid development and change. The missing `rust-toolchain` file typically indicates both a channel along with an exact numeric or dated version (though their specification varies; see below).
Recommendation	Specify an explicit version of the Rust toolchain in a `rust-toolchain` file placed at the root of the code. Place this file under version control to ensure consistent builds across all users and environments. Add a periodic gating milestone to the development process that involves reviewing and updating the toolchain version along with project dependencies. As an example, on an updated Ubuntu 20.04 machine, the following shell session demonstrates how to find the latest stable toolchain version, create the necessary file, place that file under version control and consistently build the project.

```
# Find out the latest stable toolchain version by installing it
$ rustup toolchain install stable
  ...<snip>...
  stable-x86_64- ... rustc 1.47.0 (18bf6b4f0 2020-10-07)...

# Create `rust-toolchain` file at root of repository; carefully note format
$ echo '1.47.0' > rust-toolchain

# Add file to git so other developers will use the same version
git add rust-toolchain

# Project will now build consistently
$ cargo test    # Or cargo bench
```

Note that the `rust-toolchain` file is not well documented and nightly versions are specified as `nightly-2020-10-13` (and only appears to support dates rather than an explicit version number).

Client Response	Given the stable toolchain is utiilized and the Rust code is minimal, the rust-toolchain is un-

[31] https://rust-lang.github.io/rustup/concepts/toolchains.html
[32] https://rust-lang.github.io/rustup/overrides.html#the-toolchain-file

necessary in our view. The expectation is the library will function for all recent versions. We rely on a diversity of CI platforms (Travis and Github actions) with the intention of catching any issues quickly. In addition, we spoke with multiple application developers using the Rust bindings and their opinion is that the file is not needed.

Finding	**Sensitive Information Not Cleared**
Risk	**Low** Impact: High, Exploitability: Low
Identifier	NCC-ETHF002-002
Status	Fixed
Category	Data Exposure
Component	D: Rust Bindings
Location	• blst/bindings/rust/Cargo.toml • `blst_scalar` in blst/bindings/rust/src/bindings.rs • `SecretKey`, `serialize()` and `deserialize()` in blst/bindings/rust/src/lib.rs
Impact	If regions of memory become accessible to an attacker, perhaps via a core dump, attached debugger or disk swapping, the attacker may be able to extract non-cleared secret values.
Description	Typically, all of a function's local stack variables and heap allocations remain in process memory after the function goes out of scope, unless they are overwritten by new data. This stale data is vulnerable to disclosure through means such as core dumps, an attached debugger and disk swapping. As a result, sensitive data should be cleared from memory once it goes out of scope. As suggested in code comments,[33] BLST as a library sensibly aims to minimize ownership of memory containing secret material, leaving responsibility for clearing secrets to the larger application. The repository `README.md` states "...these ultimately belong in language-specific bindings". The Rust bindings are the intermediary to the larger application, and the location where the responsibility begins. The secret key `serialize()` and `deserialize()` functions on lines 364-382 of `lib.rs` have an opportunity to include simple memory-clearing functionality. Since the results of memory-clearing functions are not used for functional purposes elsewhere, these functions can become the victim of compiler optimizations and be eliminated. While there are a variety of "tricks"[34] to attempt to avoid compiler optimizations and ensure that a clearing routine is performed reliably, the Rust community has largely adopted the approach provided by the Zeroize[35] crate.
Recommendation	Utilize the Zeroize crate to derive the zeroize-on-drop trait for senstive values such as secret keys and scalars. As an example, this entails minor edits to the three files as shown below (line numbers are approximate). 1. Include the `zeroize` crate dependency in the `Cargo.toml` file.

```
38 [dependencies]
39 threadpool = "^1.8.1"
40 zeroize = { version = "1.1.1", features = ["zeroize_derive"] }
```

2. Attach a derived zeroize-on-drop to the `blst_scalar` (which is needed for the following step) in the `bindings.rs` file. Note that this file is created by `rust-bindgen` so upstream

[33] https://github.com/supranational/blst/blob/414ac6b185f6b2ef2e6364d5716f915af966c465/bindings/rust/src/lib.rs#L357
[34] https://www.usenix.org/sites/default/files/conference/protected-files/usenixsecurity17_slides_zhaomo_yang.pdf
[35] https://docs.rs/zeroize/1.1.1/zeroize/

modifications[36] will be required.

```
use zeroize::Zeroize;

#[derive(Zeroize)]
#[zeroize(drop)]
#[repr(C)]
#[derive(Debug, Default, Clone, PartialEq, Eq)]  // Copy not needed
pub struct blst_scalar {
    pub b: [byte; 32usize],
}
```

3. Attach a derived zeroize-on-drop trait to the `SecretKey` in the `lib.rs` file.

```
/// Secret Key
#[derive(Zeroize)]
#[zeroize(drop)]
#[derive(Default, Debug, Clone)]
pub struct SecretKey {
    pub value: blst_scalar,
}
```

4. Adapt the secret key `serialize()` functions to return a `type alias` struct with the zeroize-on-drop trait attached per step 2 above (an example can be found in Sigma Prime's Lighthouse[37,38]). The `deserialize()` function can remain unchanged as `Self` is addressed by step 3 above.

After the above shown edits, `cargo test` and `cargo bench` run successfully.

Ensure the same approach is taken to attach the zeroize-on-drop trait to all secret material found in the Rust bindings.

Retest Results NCC Group reviewed changes made in the updated `2c8038d` commit.[39] Code changes in the `bindings/rust/Cargo.toml`, `bindings/rust/src/bindings.rs` and `bindings/rust/src/lib.rs` files involving the `zeroize` crate were observed per the recommendation. As such, this finding has been marked as 'Fixed'.

[36] https://github.com/rust-lang/rust-bindgen/issues/1089
[37] https://github.com/sigp/lighthouse/blob/ncc-october/crypto/bls/src/generic_secret_key.rs#L24
[38] https://github.com/sigp/lighthouse/blob/ncc-october/crypto/bls/src/zeroize_hash.rs#L6-L9
[39] https://github.com/supranational/blst/commit/2c8038d3eef7f01e773089be93f4e21b2ad7981d

Finding **Insufficient `PublicKey` Validation**

Risk **Low** Impact: Medium, Exploitability: Low

Identifier NCC-ETHF002-007

Status Fixed

Category Data Validation

Component D: Rust Bindings

Location Lines 402-412 of blst/bindings/rust/src/lib.rs

Impact An incomplete `key_validate()` function which permits an identity public key, where the corresponding secret key is equal to zero, does not meet the BLS Signature specification and may allow a valid signature for every message under this key. A malicious signer could then introduce uncertainty about which messages were signed.

Description This finding is a sibling to finding NCC-ETHF002-005 on page 12 involving deserialization of `SK==0`.

The `key_validate()` function as implemented in `lib.rs` is shown below. After deserializing a candidate public key on line 403, the code confirms it is in the correct subgroup on line 406 (which is required to ensure that subsequent pairing operations is defined). If this check is successful, the key is returned to the function caller.

```
402 pub fn key_validate(key: &[u8]) -> Result<Self, BLST_ERROR> {
403     let pk = PublicKey::from_bytes(key)?;
404     let err: bool;
405     unsafe {
406         err = $pk_in_group(&pk.point);
407     }
408     if err != true {
409         return Err(BLST_ERROR::BLST_POINT_NOT_IN_GROUP);
410     }
411     Ok(pk)
```

The above check is sufficient to meet the requirements of section 5.1 of the BLS Signature specification version 2.[40] A subsequent version (v4[41]) excerpted below has an additional requirement: that the public key not correspond the the identity-point. The code shown above does not perform this check.

> A non-identity point is required because the identity public key has the property that the corresponding secret key is equal to zero, which means that the identity point is the unique valid signature for every message under this key. A malicious signer could take advantage of this fact to equivocate about which message he signed. ...equivocation is infeasible for BLS signatures under any nonzero secret key ... Prohibiting SK == 0 eliminates the exceptional case, which may help to prevent equivocation-related security issues in protocols that use BLS signatures.

Recommendation Add a check that disallows the identity point. As noted separately, the bindings should have minimal logic, so ultimately these checks (including subgroup) should migrate to the C source.

[40] https://tools.ietf.org/html/draft-irtf-cfrg-bls-signature-02#section-5.1
[41] https://tools.ietf.org/html/draft-irtf-cfrg-bls-signature-04#section-5.1

Retest Results NCC Group reviewed changes made in the updated `d9c8de7` commit.[42] Code changes in the `bindings/rust/src/lib.rs` file involving a check for `pk_is_inf` were observed per the recommendation. As such, this finding has been marked as 'Fixed'.

[42] https://github.com/supranational/blst/commit/d9c8de70b433fb4f798eb2244759bbbcca974b01

Finding **Incorrect Result for Zero Length Aggregation**

Risk **Low** Impact: Low, Exploitability: Low

Identifier NCC-ETHF002-008

Status Partially Fixed

Category Data Validation

Component D: Rust Bindings

Location
- Lines 521-557 of blst/bindings/rust/src/lib.rs
- Lines 928-964 of blst/bindings/rust/src/lib.rs

Impact Zero-length input given to four `aggregate*()` functions does not meet the BLS Signature specification behavior and will panic.

Description The `aggregate_serialized()` function on `AggregatePublicKey` from `lib.rs` are shown below. As indicated in the first comment, the scenario involving a zero length `pks` input array is not handled.

```
537 pub fn aggregate_serialized(
538     pks: &[&[u8]],
539 ) -> Result<Self, BLST_ERROR> {
540     // TODO - handle case of zero length array?
541     // TODO - subgroup check
542     // TODO - threading
543     let mut pk = PublicKey::from_bytes(pks[0])?;
544     let mut agg_pk = AggregatePublicKey::from_public_key(&pk);
545     for s in pks.iter().skip(1) {
546         pk = PublicKey::from_bytes(s)?;
547         unsafe {
548             // TODO - does this need add_or_double?
549             $pk_add_or_dbl_aff(
550                 &mut agg_pk.point,
551                 &agg_pk.point,
552                 &pk.point,
553             );
554         }
555     }
556     Ok(agg_pk)
557 }
```

When the above function is called with a zero-length `pks` input array, line 543 will cause a panic.[43] Note that the adjacent `aggregate()` function on lines 522-535 has the same issue. However, its function signature returns `Self` rather than a `Result<>` as in the function shown above. Finally, both functions are again implemented for `AggregateSignature` on lines 928-964 with the same issue again.

The BLS Signature specification[44] requires that `n >= 1 or return INVALID`.

Recommendation Implement a check for a zero-length `pks` input array in each of the four functions, and return an appropriate `Err(BLST_ERROR)` if found. Two of the four functions will need their return type modified to `Result<Self, BLST_ERROR>`.

[43] https://doc.rust-lang.org/reference/expressions/array-expr.html#array-and-slice-indexing-expressions
[44] https://tools.ietf.org/html/draft-irtf-cfrg-bls-signature-04#section-2.8

Retest Results NCC Group reviewed changes made in the updated `1e288d9` commit.[45] Code changes in the `bindings/rust/src/lib.rs` file involving checks for `.len() == 0` on both `pks` and `sigs` were observed per the recommendation. However, line 530 of the adjacent `aggregate()` function will panic when given an empty list. As such, this finding has been marked as 'Partially Fixed'.

Client Response The `aggregate()`-related portion of this finding will be resolved in the nearest API change. For now, presenting a crash is preferred over returning incorrect data.

[45] https://github.com/supranational/blst/commit/1e288d96ed2a336aacb380bcbd8eab6fc2ddb1e4

Finding	**Struct Fields With Extraneous `pub` Access Modifiers**
Risk	**Low** Impact: Low, Exploitability: Undetermined
Identifier	NCC-ETHF002-017
Status	Fixed
Category	Configuration
Component	D: Rust Bindings
Location	• Lines 289, 395, 501, 574 and 907 of blst/bindings/rust/src/lib.rs • Line 19 of blst/bindings/rust/src/bindings.rs
Impact	Exposing internal library struct fields to the application increases the risk of misuse and the impact of field changes, and may be indicative of an incomplete API.
Description	The Rust bindings implemented in `lib.rs` define the `SecretKey`, `PublicKey`, `AggregatePublicKey`, `Signature`, and `AggregateSignature` structs for use by the application, with the first struct shown below.

```
288 pub struct SecretKey {
289     pub value: blst_scalar,
290 }
```

Annotating the internal field with the `pub` modifier, as highlighted above, causes it to be accessible to the calling application. This accessibility allows the application to bypass the API and directly perform operations on the internal fields which is generally inadvisable. For example, the application may attempt to deserialize a `SecretKey`[46] without performing the necessary bounds check.[47] The application may (or may not) be performing these direct operations due to the lack of a suitable API function.

In addition, structs with public fields may become entwined with application code such that internal BLST library changes cause unanticipated application breakage,[48] e.g., through brittle pattern matching logic.

Line 19 of `bindings.rs` adds the `pub` access modifier to an internal field of the `blst_scalar` struct as described above.

Recommendation	Remove the `pub` access modifier from the internal fields of the six structs noted. The above change does not impact `cargo test` but may initially impact application users. This impact will then inform either proper API usage or further API development.
Retest Results	NCC Group reviewed changes made in the updated `f1a2b9f` commit.[49] Code changes in the `bindings/rust/src/lib.rs` file involving the removal of the `pub` keyword from the `SecretKey`, `PublicKey`, `AggregatePublicKey`, `Signature` and `AggregateSignature` structs were observed per the recommendation. As such, this finding has been marked as 'Fixed'.

[46] https://github.com/sigp/lighthouse/blob/ncc-october/crypto/bls/src/impls/blst.rs#L71
[47] https://tools.ietf.org/html/draft-irtf-cfrg-bls-signature-04#section-2.3
[48] https://users.rust-lang.org/t/structs-with-public-fields-are-brittle/989
[49] https://github.com/supranational/blst/commit/f1a2b9f53c4685c55c632d3147aaa47c6d501770

Finding	**Sensitive Information Not Cleared**
Risk	**Low** Impact: High, Exploitability: Low
Identifier	NCC-ETHF002-012
Status	Fixed
Category	Data Exposure
Component	E: Golang Bindings
Location	Line 42 of blst/bindings/go/blst.go
Impact	If regions of memory become accessible to an attacker, perhaps via a core dump, attached debugger or disk swapping, the attacker may be able to extract plaintext secret values.
Description	This finding is a sibling of finding NCC-ETHF002-002 on page 26 which relates to the Rust bindings. Typically, all of a function's local stack variables and heap allocations remain in process memory after the function goes out of scope, unless they are overwritten by new data or garbage collection is performed. This stale data is vulnerable to disclosure through means such as core dumps, an attached debugger and disk swapping. As a result, sensitive data should be cleared from memory once it is no longer needed. As suggested in code comments,[50] BLST as a library sensibly aims to minimize ownership of memory containing secret material, leaving responsibility for clearing secrets to the larger application. The repository `README.md` states "...these ultimately belong in language-specific bindings". The Golang bindings are the intermediary to the larger application, and the location where the responsibility begins. The secret key struct on line 42 of `blst.go` has no associated functionality to assist in clearing its contents from memory after use. Since the results of memory-clearing functions are not used for functional purposes elsewhere, these functions can become the victim of compiler optimizations and be eliminated. While there are a variety of tooling-specific "tricks"[51] to attempt to avoid compiler optimizations and ensure that a clearing routine is performed reliably, Golang offers the opportunity for the programmer to either A) explicitly call a C function to immediately perform this task, or B) utilize runtime functionality[52] to defer zeroizing to garbage collection time.

Recommendation	Preferably, expose and document the usage of the existing `vec_zero()`[53] function to clear secrets from memory after use.
	
	Alternatively, document the usage of `runtime.SetFinalizer()`[54] as shown below (from https://github.com/golang/go/issues/21865#issuecomment-494054993):
	
	`type Secret struct { key [16]byte }`
	
	`s := &Secret{key: ...}`
	`runtime.SetFinalizer(s, func(s *Secret) { s.key = [16]byte{} })`

[50] https://github.com/supranational/blst/blob/414ac6b185f6b2ef2e6364d5716f915af966c465/bindings/rust/src/lib.rs#L357
[51] https://www.usenix.org/sites/default/files/conference/protected-files/usenixsecurity17_slides_zhaomo_yang.pdf
[52] https://github.com/golang/go/issues/21865#issuecomment-494054993
[53] https://github.com/supranational/blst/blob/f3647bc54593258eecc995c88045e02ab48e1996/src/vect.h#L315
[54] https://golang.org/pkg/runtime/#SetFinalizer

Retest Results NCC Group reviewed changes made in the updated `10346f4` commit.[55] Code changes in the `bindings/go/blst.go` file involving the addition of a `Zeroize()` function for execution at garbage collection time, along with corresponding test cases, were observed per the recommendation. As such, this finding has been marked as 'Fixed'.

[55] https://github.com/supranational/blst/commit/10346f4bfc57c13c9955f27a03454dc3bc1e78c6

Finding	**Missing Public `KeyValidate` Function**
Risk	**Low** Impact: High, Exploitability: Low
Identifier	NCC-ETHF002-013
Status	Fixed
Category	Cryptography
Component	E: Golang Bindings
Location	Absent from Golang bindings
Impact	An absent `KeyValidate()` function (which is present in the Rust bindings) will require the application to correctly validate public keys, which may introduce oversights.
Description	Section 5.1[56] of the BLS Signatures specification discusses the necessity of validating public keys prior to their use, though requirements vary by scheme as excerpted below: > 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. For the proof of possession scheme, each public key MUST be accompanied by a proof of possession, and use of PopVerify is REQUIRED. This suggests the need for a `KeyValidate()` API function for the Golang bindings (as is provided in the Rust bindings). Section 2.5 of the specification defines the central functionality to be "... ensures that a public key represents a valid, non-identity point that is in the correct subgroup". Without this Golang API, an application is required to implement this logic and there is significant potential for mistakes. If the user refers to an older version of the specification,[57] the check for a non-identity key may be missed. If the user refers to the BLST root `README.md` file[58] for direction, this same check may be missed. If the user refers to the existing Rust bindings,[59] this same check may be missed.
Recommendation	Implement and expose a `KeyValidate()` function that matches section 2.5 of the BLS Signatures specification; the Golang and Rust interfaces should have an identical signature except for necessary language-specific differences.
Retest Results	NCC Group reviewed changes made in the updated `e75e2ad` commit.[60] Code changes in the `bindings/go/blst.go` file involving the addition of a `KeyValidate()` function for each point type was observed per the recommendation. As such, this finding has been marked as 'Fixed'.

[56] https://tools.ietf.org/html/draft-irtf-cfrg-bls-signature-04#section-5.1
[57] https://tools.ietf.org/html/draft-irtf-cfrg-bls-signature-02#section-2.5
[58] https://github.com/supranational/blst/blob/414ac6b185f6b2ef2e6364d5716f915af966c465/README.md#signature-verification
[59] https://github.com/supranational/blst/blob/master/bindings/rust/src/lib.rs#L402
[60] https://github.com/supranational/blst/commit/e75e2add28767eeeaac11a55548c3aa06dd84e33

Finding	**Extraneous Type Exports**
Risk	**Low** Impact: Medium, Exploitability: Low
Identifier	NCC-ETHF002-018
Status	Risk Accepted
Category	Configuration
Component	E: Golang Bindings
Location	Lines 31-39 of blst/bindings/go/blst.go
Impact	Exposing internal library types to the application increases the risk of misuse and the impact of internal library changes on application users, and may be indicative of an incomplete API.
Description	This finding is a sibling to finding NCC-ETHF002-010 on page 22 and finding NCC-ETHF002-017 on page 32.

The `blst.go` source file implements the Golang bindings and exports a variety of types, as shown below.

```
31 type Scalar = C.blst_scalar
32 type Fp = C.blst_fp
33 type Fp2 = C.blst_fp2
34 type Fp6 = C.blst_fp6
35 type Fp12 = C.blst_fp12
36 type P1 = C.blst_p1
37 type P2 = C.blst_p2
38 type P1Affine = C.blst_p1_affine
39 type P2Affine = C.blst_p2_affine
40 type Message = []byte
41 type Pairing = []uint64
42 type SecretKey = Scalar
```

The code above effectively aliases the internal C types to Golang. However, when the first character of a Golang identifier is capitalized, that identifier is exported[61] and becomes usable to the calling application.

Several identifiers, such as `Fp2` and `Fp6`, correspond to strictly internal types that should not be used outside of the library. These types are exported along with additional associated functionality such as `Equals()`, `FromBEndian()`, `ToBEndian()` etc.

The latter three identifiers shown in the above code fragment relate to the library's top-level functionality and thus are appropriate. However, the absence of `PublicKey` and `Signature` is notable and explains the need to export the variety of 'points'. This suggests an incomplete API that requires unnecessary expertise on the part of the library user (e.g., to differentiate which type of point is used for a particular value in a specific ciphersuite configuration).

Recommendation	Export all necessary types corresponding to the top-level library functionality only. Remove the internal types from export by using a lower-case leading character.
Client Response	Same rationale as with Rust in NCC-ETHF002-010 finding NCC-ETHF002-010 on page 22.

[61] https://golang.org/ref/spec#Exported_identifiers

Finding **Insufficient Public Key Validation on Deserialization**

Risk **Low** Impact: Medium, Exploitability: Low

Identifier NCC-ETHF002-019

Status Risk Accepted

Category Data Validation

Component E: Golang Bindings

Location
- Lines 1323-1336 of blst/bindings/go/blst.go
- Lines 1515-1528 of blst/bindings/go/blst.go

Impact Incomplete validation which permits an identity public key, where the corresponding secret key is equal to zero, does not meet the BLS Signature specification and may allow a valid signature for every message under this key. A malicious signer could then introduce uncertainty about which messages were signed.

Description This finding is partially related to finding NCC-ETHF002-007 on page 28.

As noted in finding NCC-ETHF002-018 on the preceding page, the `blst.go` source file does not export a `PublicKey` struct. Thus, the library user must utilize the correct exported 'point' type to perform any necessary operations. One significant operation is deserialization in which the BLS Signatures specification requires the public key to be a valid, non-identity point in the correct subgroup. The relevant function for the `P1Affine` type is shown below.

```
1323 func (p1 *P1Affine) Deserialize(in []byte) *P1Affine {
1324     if len(in) != BLST_P1_SERIALIZE_BYTES {
1325         return nil
1326     }
1327     if C.blst_p1_deserialize(p1,
1328     (*C.byte)(&in[0])) != C.BLST_SUCCESS {
1329         return nil
1330     }
1331
1332     if !bool(C.blst_p1_affine_in_g1(p1)) {
1333         return nil
1334     }
1335     return p1
1336 }
```

The validation test on lines 1332-1334 checks subgroup membership via a technique outlined in the paper "Faster Subgroup Checks for BLS12-381".[62] The required check for the identity point[63] is not performed.

This same scenario is also present in the `Deserialize()` function for `P2Affine` as implemented on lines 1515-1528.

Recommendation Ensure the deserialized public key is validated against the identity point as required by the specification. This may necessitate a new API function if the current deserialization function is used for multiple purposes with different requirements.

Client Response Added KeyValidate function (see NCC-ETHF002-013 finding NCC-ETHF002-013 on page 35).

[62] https://eprint.iacr.org/2019/814.pdf
[63] https://tools.ietf.org/html/draft-irtf-cfrg-bls-signature-04#section-5.1

The expectation is the application will deserialize/uncompress an object and call KeyValidate if that object is a previously unvalidated public key. The deserialize interface is agnostic to the higher level object type.

Finding	**Missing Checks in Aggregate Verify**
Risk	**Low** Impact: Medium, Exploitability: Low
Identifier	NCC-ETHF002-020
Status	Fixed
Category	Data Validation
Component	E: Golang Bindings
Location	• Lines 282-309 in blst/bindings/go/blst.go • Lines 313-365 in blst/bindings/go/blst.go • Lines 830-857 in blst/bindings/go/blst.go
Impact	A zero-length list of `msgs`/`pks` input provided to the `AggregateVerify()` function incorrectly returns `true`, rather than `INVALID` (or `false`) as the BLS Signatures specification requires.

Description	This finding is a sibling to finding NCC-ETHF002-009 on page 20. There are multiple instances of this finding.
	
	The first instance involves the `AggregateVerify()` function as implemented on lines 282-309 `blst.go` which is partially excerpted below.
	
	`func (sig *P2Affine) AggregateVerify(pks []*P1Affine, msgs []Message, dst []byte, optional ...interface{}) bool { // useHash bool, augs [][]byte`
	`// sanity checks and argument parsing`
	`if len(pks) != len(msgs) {`
	`return false`
	`}`
	`_, augs, useHash, ok := parseOpts(optional...)`
	`...<snip>...`
	`return coreAggregateVerifyPkInG1(sigFn, pkFn, msgs, dst, useHash)`
	`}`
	
	The BLS Signatures specification requires that the `AggregateVerify()` function return `INVALID` (or `false`) when provided with zero-length `msgs`/`pks` lists as input. The function shown above will instead progress down to its return statement with a function closure passed to `coreAggregateVerifyPkInG1()` shown below. This latter function will then return `true`.
	
	`func coreAggregateVerifyPkInG1(sigFn sigGetterP2, pkFn pkGetterP1, msgs []Message, dst []byte, optional ...bool) bool { // useHash`
	`n := len(msgs)`
	`if n == 0 {`
	`return true`
	`}`
	`...`
	
	Additional instances of the same issue can be found on lines 313-365, 830-857 and 861-913. In the same way, this pattern is repeated for several functions calling `multipleAggregateVerifyPkInGx()`.

Recommendation	For each instance, implement early checks for zero-length input that return `false`.

Retest Results NCC Group reviewed changes made in the updated `9b4b16f` commit.[64] Code changes in the `bindings/go/blst.go` file involving the addition of early checks for `length==0` returning `false` was observed per the recommendation. As such, this finding has been marked as 'Fixed'.

[64] https://github.com/supranational/blst/commit/9b4b16fb42692370ba8a6ccfbca1803691225413

In this section, we present a few suggestions for achieving performance improvements in BLST.

Modular Inversion

Inversion modulo a prime p is classically performed through two possible methods: Fermat's Little Theorem (FLT), i.e. raising to the power $p - 2$, and the extended Euclidean algorithm. There are several variants of the latter; the most used in cryptographic applications are of the "binary GCD" type, which uses only subtractions and right shifts. It is generally considered that the FLT method should be used when processing secret values, to avoid timing-based side channels.

In 2019, Bernstein and Yang published a new constant-time algorithm to compute modular inverses[65]; they found it to be faster than FLT when working modulo $2^{255} - 19$ on 64-bit x86 systems, even though such a situation is known to favour the FLT, owing to the efficient 64-bit multipliers of the CPU and the fast modular reduction allowed by the special format of the modulus. In 2020, Pornin described a variant of the binary GCD which is also constant-time, and even faster.[66] Using this latter article, we find that for modulus $p = 2^{255} - 19$, the optimized binary GCD completes with a cost roughly equal to that of 180 squarings in that field. As another data point, using the same field but a much smaller target architecture (ARM Cortex M0+), an assembly implementation of the binary GCD can be made in 54973 cycles, i.e. about 55 squarings.[67]

In the case of curve BLS12-381, the field of interest uses a 381-bit modulus p. Since the binary GCD has a quadratic cost in the size of the modulus, we can expect its cost in that field to be no more than 180 squarings, and probably less than that, since multiplications in that field are comparatively more expensive due to the use of Montgomery reduction instead of the fast reduction allowed by quasi-Mersenne primes. As a rough estimate, we may hope for an implementation of the optimized binary GCD to have a cost equivalent to about 140 squarings, i.e. less than a third of the cost of FLT-based inversion in that field (461 squarings, as implemented in BLST with an optimized addition chain).

Variable-time variants of both the Bernstein-Yang and the Pornin algorithms are possible, and can yield some additional speed-ups, applicable to situations where the value to invert is not secret (e.g. when handling public keys and verifying signatures). In another 256-bit field with a special format (used for curve secp256k1), on 64-bit x86 CPU, inversion times below 4000 cycles have been reported.[68]

Normalization to Affine Coordinates

Even with an optimized modular inversion, as described in the previous section, doing all computations on elliptic curve points in strict affine coordinates is still much more expensive than using fractional coordinates, e.g. Jacobian coordinates, as currently implemented in BLST. However, a fast modular inversion may make it worthwhile to convert the precomputed points in the window back to affine coordinates, when multiplying a non-fixed point by a scalar.

The current implementation of `POINTonE1_add()` uses the classic 11M+5S formulas. The alternate function `POINTonE1_add_affine()`, which expects the second operand to be in affine coordinates, does so in 7M+4S, i.e. saves a cost of 4M+1S. When multiplying a point in the subgroup of order r (e.g. to generate a signature), scalars have size 255 bits, and the point multiplication routine with 5-bit windows makes 51 calls to the generic point addition routine, using a point from the window (or its opposite) as second operand. If all window points are in affine coordinates, then the cumulated savings are 204M+51S. The question is then whether such savings are enough to offset the cost of converting all window points to affine coordinates.

To convert a point from Jacobian $(X:Y:Z)$ coordinates to affine coordinates $(x, y) = (X/Z^2, Y/Z^3)$, one needs to compute one inversion (to obtain Z^{-1}), then 3M+1S to finish the conversion. If several inversions must be computed, then they can be mutualized, using a trick due to Montgomery: for any two values a and b that must be inverted, one can do a single inversion $1/(ab)$ and then obtain $1/a = b/(ab)$ and $1/b = a/(ab)$. Applied recursively on a list of n values to invert, this method allows computing the n inverses with a single modular inversion, and $3(n - 1)$ extra

[65] https://eprint.iacr.org/2019/266
[66] https://eprint.iacr.org/2020/972
[67] https://github.com/pornin/x25519-cm0
[68] https://github.com/bitcoin-core/secp256k1/pull/767#issuecomment-687583246

multiplications. In the case of BLST, with 5-bit windows, there are $n = 16$ points to convert to affine coordinates, so the total cost of that conversion should be that of one modular inversion, and 93M+16S extra cost.

Using these figures as estimates, the conversion of window points to affine coordinates should incur net overall savings if the cost of the inversion is less than 111M+35S. Since we estimated the cost of the optimized binary GCD at about 140S in the field of interest, and since squarings are normally faster than multiplications, we conclude that it is probable that this technique would lower the cost of multiplying group elements on the base curve (E1) by scalars (and, in particular, the cost of signature generation).

On the extended curve in the field extension $\mathbb{F}_{p^2}$, larger savings are expected. This can be seen as a two-step process: first, point coordinates can be modified so that their Z coordinates are part of the base field $\mathbb{F}_p$; then, these simplified coordinates may be inverted to fully convert all window points to affine coordinates, similarly to above.

In the description below, we will denote the costs of multiplication and squaring in $\mathbb{F}_p$ by M and S, respectively, and the costs of multiplication and squaring in $\mathbb{F}_{p^2}$ by M′ and S′. With the implementation techniques used in BLST, we have the following approximate correspondences:

- Generic multiplication in $\mathbb{F}_{p^2}$: 1M′ = 3M
- Generic squaring in $\mathbb{F}_{p^2}$: 1S′ = 2M
- Mixed multiplication (one element of $\mathbb{F}_{p^2}$ by one element of $\mathbb{F}_p$): 2M

Any field element $z \in \mathbb{F}_{p^2}$ can be uniquely represented as $z = a + ib$ where a and b are elements of $\mathbb{F}_p$ (the "real part" and "imaginary part" of z, respectively). Then, one can notice that $1/z = (a - ib)/(a^2 + b^2)$. Using this technique, a point in Jacobian coordinates $(X{:}Y{:}Z)$ can be converted to $(X'{:}Y'{:}Z')$ with $Z' \in \mathbb{F}_p$ with cost 3M′+1S′+2S = 11M+2S. Applying this conversion on all 16 window points thus costs 176M+32S. On the other hand, when the conversions are applied, cost of generic point addition (`POINTinE2_add()` function) is reduced, from 11M′+5S′ = 43M down to 37M+1S; over the 51 point additions involved in the point multiplication routine (with a 255-bit scalar), these amount to saving 255M, and furthermore replacing 51 multiplications with cheaper squarings. The savings exceed the cost of the conversion, thus making the optimization worthwhile.

As a second step, when all Z coordinates are in $\mathbb{F}_p$, they can be inverted with the techniques previously described; that way, all window points can be converted to affine coordinates at an extra cost of one inversion in $\mathbb{F}_p$, and 125M+16S. This brings the total cost of conversion of all 16 window points to affine coordinates to 301M+48S and one inversion in $\mathbb{F}_p$. Mixed point addition over the extended field has cost 7M′+4S′ = 29M, which is 14M lower than the generic point addition; over 51 point additions, this leads to savings of 714M, thus net savings overall as long as the inversion is less expensive than 365M, which is very likely to be true if using the optimized binary GCD, as detailed in a previous section.

All these figures are estimates based on the number of multiplications and squarings; computations on points also involve other operations, such as additions and subtractions, which are cheaper but not necessarily negligible. Therefore, benchmarks should be performed to measure the savings that can thus be obtained. At least the first step of reduction for curve E2 (reducing Z coordinates to the base field) is independent of the modular inversion technique and is likely to have tangible benefits by itself.

Alternate Formulas

BLST currently uses the classic 11M+5S addition formulas for point addition over short Weierstraß curves in Jacobian coordinates (13M+5S for generic addition that can also compute doublings). These are described in various places, notably the Explicit-Formulas Database.[69] However, the EFD is not fully up-to-date, especially with regard to any formulas discovered after 2013 or so. In late 2015, Renes, Costello and Batina published new formulas for short Weierstraß curves.[70] These formulas, when applied to points of a short Weierstraß curve in projective coordinates, compute a generic point addition in only 12M, which is noticeably faster. Furthermore, these formulas are *complete*, i.e. they handle all combinations of operands, including when either (or both) is the point-at-infinity, or the result is

[69] http://hyperelliptic.org/EFD/g1p/auto-shortw-jacobian-0.html#addition-add-2007-bl
[70] https://eprint.iacr.org/2015/1060

the point-at-infinity, or the two operands represent the same point.[71] This fully avoids issues such as the one reported in finding NCC-ETHF002-016 on page 14.

The Renes-Costello-Batina operate in projective coordinates (coordinates $(X_p{:}Y_p{:}Z_p)$ correspond to the affine point $(X_p/Z_p, Y_p/Z_p)$), not in the Jacobian coordinates currently used by BLST. The article contains formulas specialized for curves with equation $y^2 = x^3 + b$, a category which includes BLS12-381; the point doubling formulas have cost 6M+2S, which is fast, but not as fast as the 3M+4S formulas currently used in Jacobian coordinates. One possible strategy is to convert from projective to Jacobian before performing a sequence of doublings, then back to projective afterwards. This can be equivalently described as converting from Jacobian to projective coordinates before performing point addition, and back to Jacobian afterwards.

Conversion from projective $(X_p{:}Y_p{:}Z_p)$ to Jacobian $(X_j{:}Y_j{:}Z_j)$ coordinates can be done in cost 2M+1S (with $X_j = X_p Z_p$, $Y_j = Y_p Z_p^2$ and $Z_j = Z_p$), while conversion back from Jacobian to projective coordinates has cost 2M+1S (with $X_p = X_j Z_j$, $Y_p = Y_j$ and $Z_p = Z_j^3$. **Important:** the Renes-Costello-Batina formulas require the point-at-infinity to be represented as $(0{:}Y_p{:}0)$ for a non-zero Y_p; therefore, conversion from Jacobian to projective should make sure that Y_p is set to a non-zero value when $Z_j = 0$.

The cost of conversion between the two coordinate systems is such that keeping to projective coordinates is a better choice except for long runs of successive point doublings; the number of doublings at which the switch to Jacobian coordinates is worthwhile depends on the relative costs of multiplications and squarings. Starting from a point in projective coordinates and going back to projective coordinates, a sequence of n point doublings has cost $6n$ multiplications and $2n$ squarings if keeping to projective coordinates, but $3n + 4$ multiplications and $4n + 2$ squarings if converting to Jacobian coordinates and back. If squarings and multiplications have the same cost, the use of Jacobian coordinates implies savings only when $n > 6$; if the cost of a squaring is 2/3 of that of a multiplication, then Jacobian coordinates take the advantage when $n \geq 4$. For long runs of doublings (e.g. as part of testing whether a point is in the expected subgroup, in the `POINTonE1_times_zz_minus_1_div_by_3()` function, where up to 41 doublings may be performed in a row), Jacobian coordinates are certainly more efficient.

In the context of multiplying a point by a scalar, with 5-bit windows, then it is unclear whether remaining on projective coordinates is faster than converting to Jacobian coordinates for doubling; this matter could be resolved only by benchmarking. No big savings (or extra costs) are expected, compared to the current BLST implementation in pure Jacobian coordinates. However, when doing a linear combination of points (a sum of products of multiple points, as implemented by `POINTonE1s_mult_w5()` and `POINTonE2s_mult_w5()`), the use of projective coordinates and the Renes-Costello-Batina formulas for at least the point additions (if not the doublings) will probably induce savings since their cost per point addition (12M, or 11M if the windows were normalized to affine coordinates) is lower than the cost of generic point addition (13M+5S, or 8M+5S with normalization to affine coordinates); this is where the completeness of formulas is most useful.

Use of complete formulas, in general, is recommended for security reasons; it makes the analysis simpler, and the implementation more robust (for instance, when verifying whether an incoming point is in the right subgroup, complete formulas don't have trouble even with low order points, as discussed for instance in GitHub issue #16[72]). These security considerations, by themselves, justify exploring implementations that use complete formulas. But it is also possible that such formulas yield, in practice, some performance improvements.

[71] Completeness is ensured as long as neither of the operands is a point of order 2 (i.e. such that $y = 0$). However, curve BLS12-381 does not contain any point of order 2; therefore, the formulas are indeed complete for the whole curve.
[72] https://github.com/supranational/blst/issues/16

Appendix B: BLS Signatures API Notes

1.0 Overview

This informational appendix contains API notes and observations related to the BLS Signatures specification and the BLST library implementation. The next section below presents a simplified view of the specification from the perspective of the user, alongside informal notes regarding input parameter formats and validation checks. This is followed by a section summarizing the implementation interface delivered by the Rust bindings for BLST. The appendix concludes with observations and recommendations on prioritizing future usability development to better support safe and reliable applications.

2.0 BLS Signatures API

The BLS Signature specification[73] specifies two configurations – the first features a minimum signature size (`min-sig`) and the second features a minimum public key size (`min-pk`). The specification then outlines a number of core operations involving key generation, signing, verifying and aggregation. Next, three different signature schemes are elaborated which differ in the ways that they defend against rogue key attacks. These signature schemes incorporate various combinations of the core operations via inheritance, delegation and wrapping them alongside additional logic. Finally, specific ciphersuites are defined to unambiguously specify the exact schemes and their configurations to enable application interoperability. As a result of this complexity, the definition and correct use of the API for each ciphersuite can be challenging to understand.

The following three subsections are intended to 'flatten' the API specification for each of three signature schemes (and their ciphersuite set) to aid in understanding. The content originated as an intermediate project artifact, but is presented here to support a comparison of specification against implementation. Each section can be considered an API, with the appropriate functions and combinations of input highlighted for the two configurations, and the required input validation noted. One item that becomes somewhat more clear in the sections that follow is that the zero-public-key and the zero-signature may occur when using the aggregation functions, but cannot occur outside of this.

2.1 BLS Signatures 'Basic Scheme' API From Specification Section 3.1

Relevant Ciphersuites:
> `BLS_SIG_BLS12381G1_XMD:SHA-256_SSWU_RO_NUL_` – Basic scheme, `min-sig`
> `BLS_SIG_BLS12381G2_XMD:SHA-256_SSWU_RO_NUL_` – Basic scheme, `min-pk`

- `KeyGen()` is the core operation defined in section 2.3
  - min-sig and min-pk: `KeyGen(IKM) -> SK_scalar`
  - BLS Signatures specification of `KeyGen` algorithm is *RECOMMENDED* (but not a required algorithm)
  - `key_info` is optional octet string input parameter; when not supplied, it defaults to an empty string
  - `SK_scalar` output must be deterministic
  - **Input check:** `IKM` must be at least 32 bytes long; it can be longer
  - **Output:** A uniform `SK` such that 1 <= `SK` < r
- `SkToPk()` is the core operation defined in section 2.4
  - min-sig: `SkToPk_p2(SK_scalar) -> PK_p2`
  - min-pk: `SkToPk_p1(SK_scalar) -> PK_p1`
  - **Input check:** `SK` input such that 1 <= `SK` < r
  - **Output:** `PK` is valid, non-identity point in the correct subgroup
- `Sign()` delegates to the identical `CoreSign` operation defined in section 2.6
  - min-sig: `Sign_p1(SK_scalar, message) -> Sig_p1`
  - min-pk: `Sign_p2(SK_scalar, message) -> Sig_p2`
  - **Input check:** `SK` input in the format output by `KeyGen` (note range constraint)
  - **Output:** `Sig` cannot be invalid or in the wrong subgroup; calculation precludes identity result

[73] https://tools.ietf.org/html/draft-irtf-cfrg-bls-signature-04

- `Verify()` delegates to the identical `CoreVerify` operation defined in section 2.7
  - min-sig: `Verify_p2_p1(PK_p2, message, Sig_p1) -> VALID/INVALID`
  - min-pk: `Verify_p1_p2(PK_p1, message, Sig_p2) -> VALID/INVALID`
  - **Input check:** `PK` must be checked via `KeyValidate` 2.5: valid, non-identity point in the correct subgroup
  - **Input check:** `Sig` from `Sign` is valid, non-identity point in the correct subgroup

- `Aggregate()` is the core operation defined in section 2.8
  - min-sig: `Aggregate_p1(list of Sig_p1) -> Sig_p1/INVALID`
  - min-pk: `Aggregate_p2(list of Sig_p2) -> Sig_p2/INVALID`
  - **Input check:** List of signatures must be >= 1, otherwise return `INVALID`
  - **Input check:** Each `Sig` from `Sign` is valid, non-identity and in the correct subgroup
  - **Output:** `Sig` cannot be invalid or in the wrong subgroup; calculation does not preclude identity

- Section 3.1.1 `AggregateVerify()` utilizes `CoreAggregateVerify` (2.9) routine **with extra logic**
  - min-sig: `Aggregate_p2_Verify_p1((list of PK_p2), (list of messages), Sig_p1)`
  - min-pk: `Aggregate_p1_Verify_p2((list of PK_p1), (list of messages), Sig_p2)`
  - **Input check:** List of messages must be >= 1, otherwise return `INVALID`
  - **Input check:** If any two messages are equal, return `INVALID`
  - **Input check:** Length of `PK` must match length of messages, otherwise return `INVALID`
  - **Input check:** `CoreAggregateVerify` checks `PKs` via `KeyValidate` for valid, non-identity point in the correct subgroup
  - **Input check:** `CoreAggregateVerify` checks `Sig` for point validity and the correct subgroup (identity not stated)

2.2 BLS Signatures 'Message Augmentation Scheme' API From Specification Section 3.2

Relevant Ciphersuites:
> `BLS_SIG_BLS12381G1_XMD:SHA-256_SSWU_RO_AUG_` – Message augmentation, `min-sig`
> `BLS_SIG_BLS12381G2_XMD:SHA-256_SSWU_RO_AUG_` – Message augmentation, `min-pk`

- `KeyGen()` is the core operation defined in section 2.3
  - min-sig and min-pk: `KeyGen(IKM) -> SK_scalar`
  - BLS Signatures specification of `KeyGen` algorithm is *RECOMMENDED* (but not a required algorithm)
  - `key_info` is optional octet string input parameter; when not supplied, it defaults to an empty string
  - `SK_scalar` output must be deterministic
  - **Input check:** `IKM` must be at least 32 bytes long; it can be longer
  - **Output:** A uniform `SK` such that 1 <= `SK` < r

- `SkToPk()` is the core operation defined in section 2.4
  - min-sig: `SkToPk_p2(SK_scalar) -> PK_p2`
  - min-pk: `SkToPk_p1(SK_scalar) -> PK_p1`
  - **Input check:** `SK` input such that 1 <= `SK` < r
  - **Output:** `PK` is valid, non-identity point in the correct subgroup

- Section 3.2.1 `Sign()` utilizes `CoreSign` (2.6) routine **with extra logic**
  - min-sig: `Sign_p1(SK_scalar, message) -> Sig_p1`
  - min-pk: `Sign_p2(SK_scalar, message) -> Sig_p2`
  - Roughly `CoreSign(SK, PK || message)`
  - **Input check:** `SK` input in the format output by KeyGen (note range constraint)
  - **Output:** `Sig` cannot be invalid or in the wrong subgroup; calculation precludes identity result

- Section 3.2.2 `Verify()` utilizes `CoreVerify` (2.7) **with extra logic**
  - min-sig: `Verify_p2_p1(PK_p2, message, Sig_p1) -> VALID/INVALID`
  - min-pk: `Verify_p1_p2(PK_p1, message, Sig_p2) -> VALID/INVALID`
  - Roughly `CoreVerify(PK, PK || message, signature)`

  - **Input check:** PK must be checked via KeyValidate 2.5: valid, non-identity point in the correct subgroup
  - **Input check:** CoreVerify checks Sigs to be valid and in correct subgroup

- Aggregate() is the core operation defined in section 2.8
  - min-sig: Aggregate_p1(list of Sig_p1) -> Sig_p1/INVALID
  - min-pk: Aggregate_p2(list of Sig_p2) -> Sig_p2/INVALID
  - **Input check:** List of signatures must be >= 1, otherwise return INVALID
  - **Input check:** Each Sig from Sign is valid, non-identity point in the correct subgroup
  - **Output:** Sig cannot be invalid or in the wrong subgroup; calculation does not preclude identity

- Section 3.2.3 AggregateVerify() utilizes CoreAggregateVerify (2.9) **with extra logic**
  - min-sig: Aggregate_p2_Verify_p1((list of PK_p2), (list of message), Sig_p1) -> VALID/INVALID
  - min-pk: Aggregate_p2_Verify_p1((list of PK_p2), (list of message), Sig_p2) -> VALID/INVALID
  - Roughly: concat message/PK pairs, CoreAggregateVerify((PKs), (concat-pairs), Sig)
  - **Input check:** List of messages must be >= 1, otherwise return INVALID
  - **Input check:** Length of PK must match length of messages, otherwise return INVALID
  - **Input check:** CoreAggregateVerify checks PKs via KeyValidate for valid, non-identity point in the correct subgroup
  - **Input check:** CoreAggregateVerify checks Sig for point validity and the correct subgroup (identity not stated)

2.3 BLS Signatures 'Proof of Possesion Scheme' API From Specification Section 3.3

Relevant Ciphersuites:

> BLS_SIG_BLS12381G1_XMD:SHA-256_SSWU_RO_POP_ – Proof of possesion, min-sig
> BLS_SIG_BLS12381G2_XMD:SHA-256_SSWU_RO_POP_ – Proof of possesion, min-pk

- KeyGen() is the core operation defined in section 2.3
  - min-sig and min-pk: KeyGen(IKM) -> SK_scalar
  - BLS Signatures specification of KeyGen algorithm is *RECOMMENDED* (but not a required algorithm)
  - key_info is optional octet string input parameter; when not supplied, it defaults to an empty string
  - SK_scalar output must be deterministic
  - **Input check:** IKM must be at least 32 bytes long; it can be longer
  - **Output:** A uniform SK such that 1 <= SK < r

- SkToPk() is the core operation defined in section 2.4
  - min-sig: SkToPk_p2(SK_scalar) -> PK_p2
  - min-pk: SkToPk_p1(SK_scalar) -> PK_p1
  - **Input check:** SK input such that 1 <= SK < r
  - **Output:** PK is valid, non-identity point in the correct subgroup

- Sign() delegates to the identical CoreSign operation defined in section 2.6
  - min-sig: Sign_p1(SK_scalar, message) -> Sig_p1
  - min-pk: Sign_p2(SK_scalar, message) -> Sig_p2
  - **Input check:** SK input in the format output by KeyGen (note range constraint)
  - **Output:** Sig cannot be invalid or in the wrong subgroup; calculation precludes identity result

- Verify() delegates to the identical CoreVerify operation defined in section 2.7
  - min-sig: Verify_p2_p1(PK_p2, message, Sig_p1) -> VALID/INVALID
  - min-pk: Verify_p1_p2(PK_p1, message, Sig_p2) -> VALID/INVALID
  - **Input check:** PK must be checked via KeyValidate 2.5: valid, non-identity point in the correct subgroup
  - **Input check:** Sig from Sign is valid, non-identity point in the correct subgroup

- AggregateVerify() delegates to the identical CoreAggregateVerify operation defined in section 2.9
  - min-sig: CoreAggregate_p2_Verify_p1((list of PK_p2), (list of messages), Sig_p1) -> VALID/INVALID

  - min-pk: `CoreAggregate_p1_Verify_p2((list of PK_p1), (list of messages), Sig_p2) -> VALID/INVALID`
  - **Input check:** List of messages must be >= 1, otherwise return `INVALID`
  - **Input check:** Length of `PK` must match length of messages, otherwise return `INVALID`
  - **Input check:** `CoreAggregateVerify` checks `PKs` via `KeyValidate` for valid, non-identity and in the correct subgroup
  - **Input check:** `CoreAggregateVerify` checks Sig for point validity and the correct subgroup (identity not stated)
- Section 3.3.2 `PopProve()`
  - min-sig and min-pk: `Pop_p1_Prove(SK_scalar) -> proof`
  - Roughly: `Q = hash_pubkey_to_point, R = SK * Q, return point_to_sig(R)`
  - **Input check:** `SK` input in the format output by `KeyGen` (note range constraint)
  - **Output:** `proof` is an octect string
- Section 3.3.3 `PopVerify()`
  - min-sig: `Pop_p2_Verify(PK_p1, proof) -> VALID/INVALID`
  - min-pk: `Pop_p1_Verify(PK_p2, proof) -> VALID/INVALID`
  - **Input check:** `PK` is checked via `KeyValidate` 2.5: valid, non-identity and in the correct subgroup
  - **Input check:** derive sig and check validity, subgroup
- Section 3.3.4 `FastAggregateVerify()`
  - min-sig: `FastAggregate_p2_Verify_p1((list of PK_p2), message, Sig_p1) -> VALID/INVALID`
  - min-pk: `FastAggregate_p2_Verify_p1((list of PK_p2), message, Sig_p1) -> VALID/INVALID`
  - Roughly: `aggregate PKs then CoreVerify(aggPK, message, sig)`
  - **Input check:** `CoreVerify` checks `PK` via `KeyValidate` 2.5: valid, non-identity and in the correct subgroup
  - **Input check:** `CoreVerify` checks `Sigs` to be valid and in correct subgroup

3.0 BLST Library API (by Struct)

The BLS bindings for Rust contained in `lib.rs` are presented below for the `min-sig` configuration. Each API function is listed under the particular struct it is 'attached' to. Note that the extraneous public exports (noted in finding NCC-ETHF002-010 on page 22) from `bindings.rs` are not shown.

```
Struct blst::min_sig::SecretKey
   pub fn key_gen(ikm: &[u8], key_info: &[u8]) -> Result<Self, BLST_ERROR>
   pub fn sk_to_pk(&self) -> PublicKey
   pub fn sign(&self, msg: &[u8], dst: &[u8], aug: &[u8]) -> Signature
   pub fn serialize(&self) -> [u8; 32]
   pub fn deserialize(sk_in: &[u8]) -> Result<Self, BLST_ERROR>
   pub fn to_bytes(&self) -> [u8; 32]
   pub fn from_bytes(sk_in: &[u8]) -> Result<Self, BLST_ERROR>

Struct blst::min_sig::PublicKey
   pub fn key_validate(key: &[u8]) -> Result<Self, BLST_ERROR>
   pub fn from_aggregate(agg_pk: &AggregatePublicKey) -> Self
   pub fn compress(&self) -> [u8; 96]
   pub fn serialize(&self) -> [u8; 192]
   pub fn uncompress(pk_comp: &[u8]) -> Result<Self, BLST_ERROR>
   pub fn deserialize(pk_in: &[u8]) -> Result<Self, BLST_ERROR>
   pub fn from_bytes(pk_in: &[u8]) -> Result<Self, BLST_ERROR>
   pub fn to_bytes(&self) -> [u8; 96]

Struct blst::min_sig::Signature
   pub fn verify(&self, msg: &[u8], dst: &[u8], aug: &[u8], pk: &PublicKey) -> BLST_ERROR
```

```
    pub fn aggregate_verify(&self, msgs: &[&[u8]], dst: &[u8], pks: &[&PublicKey]) -> BLST_ERROR
    pub fn fast_aggregate_verify(&self, msg: &[u8], dst: &[u8], pks: &[&PublicKey]) -> BLST_ERROR
    pub fn fast_aggregate_verify_pre_aggregated(&self, msg: &[u8], dst: &[u8], pk: &PublicKey)
            -> BLST_ERROR
    pub fn verify_multiple_aggregate_signatures(msgs: &[&[u8]], dst: &[u8], pks: &[&PublicKey],
            sigs: &[&Signature], rands: &[blst_scalar], rand_bits: usize) -> BLST_ERROR
    pub fn from_aggregate(agg_sig: &AggregateSignature) -> Self
    pub fn compress(&self) -> [u8; 48]
    pub fn serialize(&self) -> [u8; 96]
    pub fn uncompress(sig_comp: &[u8]) -> Result<Self, BLST_ERROR>
    pub fn deserialize(sig_in: &[u8]) -> Result<Self, BLST_ERROR>
    pub fn from_bytes(sig_in: &[u8]) -> Result<Self, BLST_ERROR>
    pub fn to_bytes(&self) -> [u8; 48]

Struct blst::min_sig::AggregatePublicKey
    pub fn from_public_key(pk: &PublicKey) -> Self
    pub fn to_public_key(&self) -> PublicKey
    pub fn aggregate(pks: &[&PublicKey]) -> Self
    pub fn aggregate_serialized(pks: &[&[u8]]) -> Result<Self, BLST_ERROR>
    pub fn add_aggregate(&mut self, agg_pk: &AggregatePublicKey)
    pub fn add_public_key(&mut self, pk: &PublicKey)

Struct blst::min_sig::AggregateSignature
    pub fn from_signature(sig: &Signature) -> Self
    pub fn to_signature(&self) -> Signature
    pub fn aggregate(sigs: &[&Signature]) -> Self
    pub fn aggregate_serialized(sigs: &[&[u8]]) -> Result<Self, BLST_ERROR>
    pub fn add_aggregate(&mut self, agg_sig: &AggregateSignature)
    pub fn add_signature(&mut self, sig: &Signature)
```

In the API shown above, the signing functionality is associated with the `SecretKey` struct and the verify functionality is associated with the `Signature` struct.

The `PopProve()` and `PopVerify()` functions from the 'Proof of Possession Scheme' are missing from the `SecretKey` and `PublicKey` structs (respectively) shown above. While these functions can be assembled from other components, the risk of user mistakes is high. This issue is captured in finding NCC-ETHF002-015 on page 6.

The `PublicKey::key_validate()` function should operate on `&self` rather than `key: &[u8]`. Arguably, the `SecretKey::sk_to_pk()` function should return `Result<PublicKey, BLST_ERROR>` rather than `PublicKey` to account for the possibility of bad input.

There are several functions, such as `fast_aggregate_verify_pre_aggregated()` and `verify_multiple_aggregate_signatures()` that appear complex, undocumented, and are not present in the BLS Signatures specification. These cannot be reviewed for correctness and risk misuse.

Note that several of the Rust bindings API perform detailed validation (e.g., see finding NCC-ETHF002-007 on page 28). This implies its absence in the C source which risks oversights by users of the C library[74] and divergence across the different bindings.

Finally, the BLST repository top-level `README.md` takes a much lower-level view of the API as a very flexible set of operations. This reflects the exports noted in finding NCC-ETHF002-010 on page 22.

API

The BLST API is defined in the C header bindings/blst.h. The API can be categorized as follows, with some example operations:

[74] https://github.com/status-im/nim-blscurve/blob/master/blscurve/blst/blst_abi.nim#L5

- Field Operations (add, sub, mul, neg, inv, to/from Montgomery)
- Curve Operations (add, double, mul, to/from affine, group check)
- Intermediate (hash to curve, pairing, serdes)
- BLS12-381 signature (sign, verify, aggregate) Note: there is also an auxiliary header file, bindings/blst_aux.h, that is used as a staging area for experimental interfaces that may or may not get promoted to blst.h.

Complementing the view above, the `README.md` also states: "The essential point to note is that it's the caller's responsibility to ensure that public keys are group-checked with blst_p1_affine_in_g1". The level of detail in the above API and statement around validation responsibilities places a heavy and complex burden on the library user. There is a risk of specification changes (such as requiring non-identity keys[75]) being overlooked by users.

4.0 Observations and Recommendations

The BLST library offers a tremendous amount of optimized technology, performance and flexibility for applications to deploy BLS Signature functionality. As suggested by the immediately preceding content above, there are some uneven aspects to the API and some pairing-cryptography expertise is required to correctly use the library. History suggests that users are challenged to correctly use basic cryptography[76] and mistakes can have significant consequences that remain unnoticed over extended periods of time[77, 78].

The largest opportunity for BLST improvement involves robust user-facing documentation (and perhaps limited API restructuring) that encourages simple and failsafe operation by non-experts. Improvements may include:

- Ciphersuite definitions alongside a description of the correct API usage for each.
- Articulate examples targeting prominent usage scenarios (e.g., Ethereum 2.0).
- Minimizing the API surface, make internal pairing opaque, and move validation checks into the C source where possible.
- Documentation around each individual function specifying input parameters and type, external expectations/requirements, internal checks and corresponding specification heading.
- Perform point validation at the point of deserialization to ensure all internal operations utilize valid data.
- Relegate all functions not specifically in the BLS Signatures specification to `bindings/blst_aux.h` for use by power-users only.

[75] https://tools.ietf.org/html/draft-irtf-cfrg-bls-signature-04#section-5.1
[76] https://securityboulevard.com/2020/04/simple-illustration-of-zoom-encryption-failure/
[77] Zerologon: Unauthenticated domain controller compromise by subverting Netlogon cryptography (CVE-2020-1472) https://www.secura.com/pathtoimg.php?id=2055
[78] https://www.zdnet.com/article/zerologon-attack-lets-hackers-take-over-enterprise-networks/

Appendix C: 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 D: Project Contacts

The team from NCC Group has the following primary members:

- Eric Schorn — Technical Lead
  eric.schorn@gmail.com
- Thomas Pornin — Consultant
  thomas.pornin@nccgroup.com
- Javed Samuel — Vice President, Practice Director Cryptography Services
  javed.samuel@nccgroup.com

The team from Ethereum Foundation has the following primary members:

- Andy Polyakov — Supranational
  appro@cryptogams.org
- Kelly Olson — Supranational
  kelly@supranational.net
- Sean Gulley — Supranational
  sean@supranational.net
- Simon Peffers — Supranational
  simon@supranational.net