BLS12-381 and BLS Signature Implementation Review

Anza Technology

June 4, 2026

Version 1.0

Prepared for:
Sam Kim

Prepared by:
Giacomo Pope
Thomas Pornin
Javed Samuel

1 Executive Summary

Synopsis

In April 2026, Anza Technology engaged NCC Group's Cryptography Services team to conduct a security assessment of a new implementation of BLS (Boneh-Lynn-Shacham) signatures over the BLS12-381 elliptic curve, following the draft specification draft-irtf-cfrg-bls-signature. This has been engineered for use with the Solana Alpenglow consensus protocol, but is advertised as generic enough for usage with other projects and so has been considered during the engagement as a generic implementation.

Two consultants spent a total of 8 person-days on this engagement including retesting.

Scope

NCC Group's evaluation targeted two GitHub repositories:

- `bls12-381-syscall` commit `78e9ae2`
- `bls-signatures` commit `31c6346`

Correct integration of this code with the critical dependencies `blst` and `blstrs` was in scope, but the implementation details of the third-party libraries was out of scope.

Limitations

There were no formal limitations to the engagement: the code reviewed is open source, and NCC Group had all the important information about the engagement before the start date.

Key Findings

No severe security issues were found. Memory zeroization of secret values was found to be potentially incomplete, see finding "Complex and Potentially Incomplete Implementation of Zeroization".

In some places related to decoding curve points, the library relies on the backend to still enforce consistency checks (e.g. that the point is on the curve) even when subgroup membership is skipped. This is not well documented by the `blstrs` crate, and not at all by the underlying C library, so these properties might change without notice. However, the `bls12-381-syscall` crate already includes dedicated unit tests to reliably detect that occurrence, should it happen.

Strategic Recommendations

- Refactor zeroization of secret values by deriving the method directly when possible and use `Zeroization` as a wrapper on types which have secrets read into them.
- The library does not enforce individual signature validity, only aggregates; this is fine for the intended usage (Alpenglow protocol), but might be surprising for other applications using the library. Consider including the prominent warning about individual signature validity within the README of the project itself.

Retest

NCC Group were directed to the pull request github.com/anza-xyz/solana-sdk/pull/702 which directly addressed finding "Complex and Potentially Incomplete Implementation of Zeroization", fixing both the zeroization of the secret key as well as modifying various method signatures to ensure secret bytes were wrapped in the `Zeroizing<>` type. As a result, this finding is considered "fixed".

2 Project Information

Assessment Name	bls-signatures and bls12-381-syscall
Assessment Dates	2026-04-09 to 2026-04-17
Number of Consultants	2
Level of Effort	8 person-days
Type	Implementation Review
Method	Code-assisted

Scope

bls-signatures

Targets

https://github.com/anza-xyz/solana-sdk/tree/31c63461356ecd27700a58a7eb796e9af2154815/bls-signatures

bls12-381-syscall

Targets

https://github.com/anza-xyz/agave/tree/78e9ae25f8562bed78b5e4758bb102d2405af693/bls12-381-syscall

3 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.

Title	Status	ID	Risk
Complex and Potentially Incomplete Implementation of Zeroization	Fixed	DLH	Low

4 Finding Details

Low

Complex and Potentially Incomplete Implementation of Zeroization

Overall Risk	Low	**Finding ID**	NCC-E032961-DLH
Impact	Low	**Component**	bls-signatures
Exploitability	Low	**Category**	Cryptography
		Status	Fixed

Impact

The current implementation of zeroize on the `SecretKey` type uses a custom volatile write without any atomic fence. As `Scalar` already has a zeroization method available, it is recommended to derive `Zeroize` on the type instead. Additionally, in some places, secret bytes are copied or created without a `Zeroize` wrapper.

Description

The `SecretKey` type, defined in *bls-signatures/src/secret_key.rs*, implements a custom zeroization method (automatically invoked on value drop) which uses a volatile write over the secret with the zero value in the scalar field:

```
36 impl Zeroize for SecretKey {
37     fn zeroize(&mut self) {
38         unsafe {
39             core::ptr::write_volatile(&mut self.0, Scalar::ZERO);
40         }
41     }
42 }
43
44 impl Drop for SecretKey {
45     fn drop(&mut self) {
46         self.zeroize();
47     }
48 }
```

The zeroization methods, that the `zeroize_derive` crate (available through `zeroize`) allows to produce automatically, also include an atomic fence to prevent unwanted compiler optimizations that would break the zeroization guarantees. This fence is not present in the custom implementation for `SecretKey`, which means that in some situations the zeroization could be incomplete, or at least delayed more than expected. Using the automatic derivation of the `Zeroize` and `ZeroizeOnDrop` traits would allow a simpler and more robust code. This derivation requires that the type of the fields for `SecretKey` have their own `zeroize()` methods, and indeed, looking at the source of the `Scalar` type in the `blstrs` crate, we see:

```
impl Zeroize for Scalar {
    fn zeroize(&mut self) {
        self.0.l.zeroize();
    }
}
```

As a result, this would allow `Zeroize` to be directly implemented through derivation on the type, as follows:

```
#[derive(Zeroize, ZeroizeOnDrop)]
pub struct SecretKey(pub(crate) Scalar);
```

Creation of secrets without zeroization

In *bls-signatures/src/keypair.rs*, a key pair can be encoded into bytes:

```
122 impl From<&Keypair> for [u8; BLS_KEYPAIR_SIZE] {
123     fn from(keypair: &Keypair) -> Self {
124         // WARNING: `bytes` contains raw secret-key material. Callers should zeroize the
        ↳ returned
125         // buffer as soon as they are done using it.
126         let mut bytes = [0u8; BLS_KEYPAIR_SIZE];
127         let secret_bytes = Zeroizing::new(Into::<[u8;
        ↳ BLS_SECRET_KEY_SIZE]>::into(&keypair.secret));
128         bytes[..BLS_SECRET_KEY_SIZE].copy_from_slice(secret_bytes.as_slice());
129         bytes[BLS_SECRET_KEY_SIZE..].copy_from_slice(&keypair.public.to_bytes_uncompressed());
130         bytes
131     }
132 }
```

The explicit warning (in the comment) states that since the bytes include the secret key, they should be zeroized by the caller after usage. However, this required zeroization is not enforced through the Rust type system, and the caller might omit that step. This could be avoided, creating a less fragile API, by implementing the `From` trait for `Zeroizing<[u8; BLS_KEYPAIR_SIZE]>`. This change may also be done for the `SecretKey` type, whose implementation contains a similar warning:

```
141 impl From<&SecretKey> for [u8; BLS_SECRET_KEY_SIZE] {
142     fn from(secret_key: &SecretKey) -> Self {
143         // WARNING: The returned buffer contains raw secret-key bytes. Callers should
        ↳ zeroize it
144         // as soon as they are done using it.
145         secret_key.0.to_bytes_le()
146     }
147 }
```

A third case is in the JSON encoding of key pairs, in *bls-signatures/src/keypair.rs*:

```
148     /// WARNING: The returned JSON string contains secret-key material. Callers should
        ↳ clear it as
149     /// soon as they are done using it.
150     pub fn write_json<W: Write>(&self, writer: &mut W) -> Result<String, Box<dyn error::Err
        ↳ or>> {
151         let bytes = Zeroizing::new(Into::<[u8; BLS_KEYPAIR_SIZE]>::into(self));
152         let json = serde_json::to_string(bytes.as_slice())?;
153         writer.write_all(json.as_bytes())?;
154         Ok(json)
155     }
156
157     pub fn write_json_file<F: AsRef<Path>>(
158         &self,
159         outfile: F,
160     ) -> Result<String, Box<dyn core::error::Error>> {
```

```
        let outfile = outfile.as_ref();

        if let Some(outdir) = outfile.parent() {
            fs::create_dir_all(outdir)?;
        }

        // Remove the file or symlink if it already exists.
        let _ = fs::remove_file(outfile);

        let mut f = {
            #[cfg(not(unix))]
            {
                OpenOptions::new()
            }
            #[cfg(unix)]
            {
                use std::os::unix::fs::OpenOptionsExt;
                OpenOptions::new().mode(0o600)
            }
        }
        .write(true)
        .create_new(true)
        .open(outfile)?;

        self.write_json(&mut f)
    }
}
```

In this case, the `String` instance returned by `write_json()` is documented as needing zeroization by the caller; the `write_json_file()` function, located immediately afterwards in the same file, calls `write_json()` and does *not* perform the zeroization (the string is implicitly dropped but not implicitly or explicitly zeroized), which highlights that the warning is certainly not sufficient to ensure correct usage.

Parsing of Secret Bytes without Zeroization

When parsing a secret key from bytes (in *bls-signatures/src/secret_key.rs*):

```
    /// Parses a canonical, non-zero secret scalar from little-endian bytes.
    fn parse_scalar(bytes: &[u8; BLS_SECRET_KEY_SIZE]) -> Result<Scalar, BlsError> {
        let scalar: Option<Scalar> = Scalar::from_bytes_le(bytes).into();
        let scalar = scalar.ok_or(BlsError::FieldDecode)?;
        if bool::from(scalar.is_zero()) {
            return Err(BlsError::FieldDecode);
        }
        Ok(scalar)
    }
```

The accepted secret bytes are not ensured to be auto-zeroizing. By modifying this method signature to only accept `Zeroize<[u8; BLS_SECRET_KEY_SIZE]>`, the code would be hardened against secret bytes living in the stack or the heap for longer than necessary (or at least force the caller to explicitly skip zeroization if they really want it that way).

Recommendation

Rather than implementing zeroization as a volatile write, derive methods using `zeroize_derive` to ensure a complete and comprehensive implementation. Additionally, use

Rust's type system to ensure input and output values which depend on secret data are correctly zeroized.

Location

- *bls-signatures/src/keypair.rs*
- *bls-signatures/src/secret_key.rs*

Retest Results

2026-05-11 – Fixed

Although the upstream repository has a method for `Zeroization` for the `Scalar` type, added in December of 2025, this new feature is not available from the most recent version published via crates.io, which has not been updated since August 2023. As a result, the recommendation to use this directly with the derive trait is incorrect with the current workflow. Instead, the Azna team discussed alternatives with NCC Group and have added an atomic fence along with the volatile write to replicate the same behavior seen in the upstream code. NCC Group note that if the `blsrs` crate is updated from `0.7.1` in the future, this code could be simplified.

Additionally, Anza have updated various method signatures to ensure that the input/output of functions accepts or returns `Zeroizing` wrapped bytes and strings. As a result, this finding has been marked as "fixed". NCC Group note that it could be worth adding a comment for users of `impl TryFrom<&[u8]> for SecretKey` to encourage the use of the `Zeroizing` wrapper for the secret bytes which they load into memory. This is more complex to enforce at the type level if Anza wish to allow this to be a slice of unknown length. Additionally, Anza could expose `impl TryFrom<&Zeroizing<[u8; BLS_SECRET_KEY_SIZE]>> for SecretKey` which would encourage a coding practice which matches Anza's library design.

5 Audit Notes

This section contains various remarks which were deemed noteworthy, but none of them constitutes a security issue.

JSON Files for Secret Keys

In *bls-signatures/src/keypair.rs*, JSON files are used to encode and store key pairs. A key pair contains in particular the secret key, which is secret data that should therefore be handled in a way which does not leak information about it through side-channels, in particular timing-based side-channels.

However, JSON encodes binary objects in a text-based format, normally using Base64. The encoding and decoding functions for Base64 are very likely to be based on look-up tables, which is not constant-time: attackers observing fine-grained timing side-channels may thus obtain information on secret keys when they are encoded or decoded. If strict constant-time discipline is desired, then a custom Base64/JSON encoder with no lookup table for byte/character conversions is needed.

It was also noted (in finding "Complex and Potentially Incomplete Implementation of Zeroization") that the resulting JSON strings are not zeroized after use.

Implementation of `Eq` and `PartialEq` for `SecretKey`

As `Eq` and `PartialEq` are variable time comparisons, it would be safer to not implement these methods on the type and instead use `ConstantTimeEq`, which is already implemented on the `Scalar` type on which the `SecretKey` depends on.

Maximum Number of Pairings

In *bls12-381-syscall/src/pairing.rs*, a maximum number of pairings (8) per system call is defined; it is enforced in the implementation. However, this maximum value does not seem to be part of the `sol_curve_pairing_map` system call specification. In the interest of application reliability, the maximum should be specified explicitly.

Typographic Errors in Comments

- In *bls-signatures/src/proof_of_possession/bytes.rs*: “Zeraoble” should be “Zeroable”. Same typo is in *src/pubkey/bytes.rs* and *src/signature/bytes.rs*.
- In *bls-signatures/src/pubkey/verify.rs*: “valud” should be “value”.
- In *bls12-381-syscall/src/encoding.rs*, the link to the Zcash specification is broken. The referenced file was removed in 2020; it seems to now be in the documentation for the *bls12_381* crate.

Typographic Errors in Tests

- In the test `test_signature_aggregate()`, `let test_message = b"test message";` is defined twice, once for each signature. To make it clearer for the test reader, it would be good to remove one to show the intention is that both signatures are over the same message.

6 Finding Field Definitions

This section explains the risk rating system used for issues within NCC Group reports. The headings below refer to some of the attributes displayed in the headings displayed at the start of each finding in the Findings Details section.

Overall Risk

The Overall Risk rating reflects NCC Group's evaluation of the risk that a finding poses. It takes into account the impact of the finding, the difficulty of exploitation, and any other relevant factors, such as the environment. While NCC Group strive to place all identified issues with the relevant context, it should be recognized that accurately quantifying the business risk posed by any issue will typically require more information than is available within a security assessment. So, for example, a risk may be reported as high from a technical perspective, although a business may consider the risk acceptable, as a result of other controls, beyond the scope of the assessment.

Risk Rating	Description
Critical	Implies an immediate, easily accessible threat of total compromise.
High	Implies an immediate threat of system compromise, or an easily accessible threat of a significant breach.
Medium	A difficult to exploit threat of a significant breach, or easy compromise of a small component of a system.
Low	Implies a relatively minor threat.
Informational	No immediate threat. An Informational issue may provide suggestions for improvements, describe functional problems that were encountered, or something that could become exploitable if conditions changed.

Mappings between CVSS scores and the verbose descriptions above (informational, low, etc) are provided in this page (for CVSS versions 3 and higher). These descriptions are often taken to be equivalent to descriptions of risk, although as that page makes clear, these are vulnerability severity descriptions, not risk descriptions.

The mapping from vulnerability severity to risk rating may not be a simple like for like mapping. That is, a CVSS score in the medium vulnerability severity range *could* be reported with a different risk rating, although in practice they are often the same.

Impact

Impact reflects the effects that successful exploitation could have upon the target system. It takes into account potential losses of confidentiality, integrity and availability.

Rating	Description
High	An attacker could read or modify all data in a system, execute arbitrary code on the system, or escalate their privileges to superuser level.
Medium	An attacker could read or modify data on a system without authorization, deny access to that system, or gain significant internal technical information.
Low	An attacker could gain small amounts of information without authorization or slightly degrade system performance. The issue could have a negative effect on the public perception of the organization or its security.

Exploitability

Exploitability reflects the ease with which an attacker may exploit a finding. It takes into account factors such as the level of access required, availability of exploitation information, requirements relating to social engineering, race conditions, brute-forcing and will also reflect any impediments to exploitation.

Rating	Description
High	An attacker could unilaterally exploit the finding without special permissions or significant roadblocks.
Medium	An attacker 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 would require implausible social engineering, a difficult race condition, the recovery of difficult-to-guess data, or the fulfilment of some other similarly unlikely requirement.