KZG10 Ceremony Audit Report

Small Powers of Tau

SECBIT

Sep 14th, 2022

1. Introduction

Small Powers of Tau provides a solution for Ethereum's Powers of Tau (PoT) setup ceremony. From August 15th to September 10th, 2022, SECBIT Labs conducted a review of both the technical specification and the Rust implementation. Our assessment reveals no critical issues in either the specification or the implementation. We report some potential risks in the code and have comments on optimizing the code and the documentation (see section 5 for details).

2. Overview

This part provides an overview of our assessment.

2.1 Basic Information

Our review covers the following repositories:

Name	Small Powers of Tau
Source	https://github.com/crate-crypto/small-powers-of-tau
Initial Commit	168ec297cc45799754d5760c30ad64c44d8a975e
Final Commit	7792ca98041b6e07b10f12efdfee8c9b712b33bf
Languages	Rust
Spec Source	https://github.com/ethereum/kzg-ceremony-specs
Initial Commit	5c562eb2b754456105ae254d44c0fdea54f3e360
Final Commit	5e5d714079aa0b80641edb8ca11367b1211bc479

2.2 File Lists

Listed below are the source code files of Small Powers of Tau that are considered in-scope for the audit:

Name	Lines	Description
lib.rs	8	Crate root with its modules
srs.rs	378	SRS struct with its implementation
update_proofs.rs	36	UpdateProof struct with its implementation
shared_secrets.rs	115	SharedSecretChain struct with its implementation
serialisation.rs	225	serialization for SRS and UpdateProof
sdk.rs	72	Transcript struct with its update and subgroup check

We also conducted a review of the following specification documents in the kzg-ceremony-specs repository:

Name	Lines	Description
BLS.md	63	BLS12-381 curve.
coordinator.md	111	Behavior of a coordinator.
participant.md	119	Behavior of a participant.
sdk.md	64	SDK interface (optional).

2.3 Findings

We outline the issues found in the rust implementation and the specification as follows:

No.	Issue Title	Type	Level	Status
1	Potential overflow or array out-of-bounds errors may cause panic	Security Risk	Medium	Fixed
2	Redundant array allocation	Code Optimization	Low	Fixed
3	Unfixed dependencies versions	Code Optimization	Low	Fixed
4	Redundant file	Code Optimization	Info	Fixed
5	Redundant functions	Code Optimization	Info	Fixed
6	Typo in kzg-ceremony-specs(Correct construction of G1/G2 Powers)	Document Optimization	Low	Fixed
7	Typo in kzg-ceremony-specs(Running Product Subgroup check)	Document Optimization	Low	Fixed
8	Inconsistency between Rust code and kzg-ceremony-specs about 'Witness Subgroup checks'	Document Optimization	Info	Discussed
9	Inconsistency between Rust code and kzg-ceremony-specs about Transcript	Document Optimization	Info	Discussed

3. Project Analysis

This part contains an overview of the Small Powers of Tau ceremony, followed by a function-level analysis of the rust implementation.

3.1 Overview of PoT ceremonies

A PoT Ceremony collaboratively computes a structured reference string that takes the form of

$$\mathsf{srs} = \begin{Bmatrix} [\tau^0]_1, & [\tau^1]_1, & [\tau^2]_1, & \cdots, & [\tau^n]_1, \\ [\tau^0]_2, & [\tau^1]_2, & [\tau^2]_2, & \cdots, & [\tau^m]_2 \end{Bmatrix}. \tag{1}$$

Herein $[\cdot]_1$ and $[\cdot]_2$ mark elements of two distinctive groups $\mathbb{G}_1$ and $\mathbb{G}_2$ both of prime order r, and upon them a bilinear pairing operation is defined $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$. The value of $\tau \in \mathbb{F}_r$ is determined as the product of all participant's private keys $\{\rho_i\}$ s.t. $\tau = \prod_i \rho_i$.

The mainstream of existing PoT implementations follows the specifications of BGM17. However, recent theoretical studies (KMSV21 in particular) have demonstrated that under the algebraic group model with integrated random oracles, some components of BGM17 are redundant, including

- hashes of protocol views in proofs of knowledge for private keys;
- reliance on external random beacons.

Small Powers of Tau honors these conclusions and hence enjoys significant simplifications over BGM17. For an overview of Small Powers of Tau, we summarize the behaviors of the two involving roles, the participant and the coordinator, as follows.

- Participant

  A participant's task is to

  1. Generate a key pair

    The participant chooses a value $\rho \in \mathbb{F}_r$ as its private key and computes $[\rho]_2 \in \mathbb{G}_2$ as its public key.

    2. Update the reference string

       To contribute to the ceremony, the participant receives from the coordinate the latest reference string srs, parse it as as $\{\{[\tau^i]_1\}_{i=0}^n, \{[\tau^j]_2\}_{j=0}^m\}$, and update each element of srs using its private key ρ

       $$\rho^i \cdot [\tau^i]_{1,2} = [(\rho \cdot \tau)^i]_{1,2} = [\tau'^i]_{1,2}. \tag{2}$$

       It submits the the updated $\mathsf{srs}' = \{\{[\tau'^{\mathsf{i}}]_1\}_{\mathsf{i}=0}^{\mathsf{n}}, \{[\tau'^{\mathsf{j}}]_2\}_{\mathsf{j}=0}^{\mathsf{m}}\}$ to the coordinator.

    3. Produce an update proof

       Each update should be accompanied by an update proof $\pi = ([\rho]_2, [\tau']_1)$. It is essentially a proof of knowledge for the private key ρ.

  This minimalist update proof features significant simplifications over that of BGM17. In BGM17, the participant has to collect the protocol view up to this moment, employs a hash-to-group step to produce $H \in \mathbb{G}_2$, and uses $\rho \cdot H \in \mathbb{G}_2$ as the proof of knowledge for the private key ρ.

- Coordinator

  The coordinator acts as the middle man of the ceremony. Its tasks are to

  1. Initiate the ceremony

     The coordinator initiate the ceremony by creating a clean-slate srs with $\tau = 1$.

  2. Sequence participants

     The coordinator sends the latest srs to a participant and collects the updated srs' and the accompanying proof π in due time. It has to decide whether to accept or discard the update. Then the coordinator proceeds the ceremony by sending either the updated srs' or the original srs to the next contributor.

  3. Verify updates

     To verify whether an update is valid, the coordinator has to check that all elements of the updated srs' are non-empty, non-zero, and in the correct prime-ordered subgroups. Then it parses the update proof π as $([\rho]_2, [\sigma]_1)$, and confirms that $[\sigma]_1$ agrees with the degree-1 element $[\tau']_1$ of srs'. It check whether $[\tau']_1$ is corrected related to $[\tau]$ by pairing equation

     $$e([\tau]_1, [\rho]_2) \overset{?}{=} e([\tau']_1, [1]_2), \tag{3}$$

     and whether srs' conforms with the powers of tau structure.

$$e([\tau'^{i}]_1, [\tau']_2) \overset{?}{=} e([\tau'^{i+1}]_1, [1]_2) \qquad \text{for } i \in [1, n-1], \tag{4}$$

$$e([\tau']_1, [\tau'^{j}]_2) \overset{?}{=} e([1]_1, [\tau'^{j+1}]_2) \qquad \text{for } j \in [1, m-1]. \tag{5}$$

The update will be thrown away when any checks fail.

Specific to Ethereum's PoT setup, there are four sets of **srs** to be generated. They spell out in the **transcript** transmitted between the coordinator and the participants as

$$\textsf{transacript} = \left\{ \begin{array}{l} srs = \left\{ \begin{array}{ccccc} [\tau_0^0]_1, & [\tau_0^1]_1, & [\tau_0^2]_1, & \cdots, & [\tau_0^{4095}]_1, \\ [\tau_0^0]_2, & [\tau_0^1]_2, & [\tau_0^2]_2, & \cdots, & [\tau_0^{64}]_2 \end{array} \right\}, \\ srs = \left\{ \begin{array}{ccccc} [\tau_1^0]_1, & [\tau_1^1]_1, & [\tau_1^2]_1, & \cdots, & [\tau_1^{8191}]_1, \\ [\tau_1^0]_2, & [\tau_1^1]_2, & [\tau_1^2]_2, & \cdots, & [\tau_1^{64}]_2 \end{array} \right\}, \\ srs = \left\{ \begin{array}{ccccc} [\tau_2^0]_1, & [\tau_2^1]_1, & [\tau_2^2]_1, & \cdots, & [\tau_2^{16383}]_1, \\ [\tau_2^0]_2, & [\tau_2^1]_2, & [\tau_2^2]_2, & \cdots, & [\tau_2^{64}]_2 \end{array} \right\}, \\ srs = \left\{ \begin{array}{ccccc} [\tau_3^0]_1, & [\tau_3^1]_1, & [\tau_3^2]_1, & \cdots, & [\tau_3^{32767}]_1, \\ [\tau_3^0]_2, & [\tau_3^1]_2, & [\tau_3^2]_2, & \cdots, & [\tau_3^{64}]_2 \end{array} \right\} \end{array} \right\}, \tag{6}$$

The values $\tau_0, \tau_1, \tau_2, \tau_3$ are produced by four separate sets of private keys provided by the participants. The implementation is based on curve BLS12-381

3.2 Function Description

This section presents a detailed analysis of the Rust implementation of Small Powers of Tau. The source code consists of 7 main structures distributed in 8 files.

lib.rs

The root of the crate.

keypair.rs

The keypair.rs defines a struct `PrivateKey` which stores the secret value `tau`.

```
pub struct PrivateKey {
    pub(crate) tau: Fr,
}
```

Four methods are implemented for the `PrivateKey` struct to create a cryptographically secure secret value and compute the corresponding public key.

1. `rand<R: Rng>(mut rand: R) -> Self`

   It generates a `PrivateKey` using entropy from an RNG.

2. `from_bytes(bytes: &[u8]) -> Self`

   It converts the input byte array to a `PrivateKey`.

3. `to_public(self) -> G2Projective`

   It generates the corresponding public key.

4. `from_u64(int: u64) -> Self`

   It converts the input to a `PrivateKey`. This function is only used for testing.

srs.rs

The srs.rs defines structs `SRS` and `Parameters`.

The struct `Parameters` specifies the length of `SRS`, i.e., the numbers of elements in $\mathbb{G}_1$ and $\mathbb{G}_2$.

```
pub struct Parameters {
    pub num_g1_elements_needed: usize,
    pub num_g2_elements_needed: usize,
}
```

The struct `SRS` stores the powers of tau in $\mathbb{G}_1$ and $\mathbb{G}_2$.

```
pub struct SRS {
    pub(crate) tau_g1: Vec<G1Projective>,
    pub(crate) tau_g2: Vec<G2Projective>,
}
```

Nine methods are implemented for the `SRS` so that users can create, update and verify an `SRS`. All the checks defined in the kzg-ceremony-specs are included.

1. `new(parameters: Parameters) -> SRS`

   It creates a PoT ceremony using parameters in `Parameters`. In the clean-slate `SRS`, all elements of `tau_g1` and `tau_g2` equal the respective generators of $\mathbb{G}_1$ and $\mathbb{G}_2$.

2. `new_for_kzg(num_coefficients: usize) -> SRS`

   It creates a powers of tau ceremony with `num_g2_elements_needed=2`. This function is only used for testing.

3. `update(&mut self, private_key: PrivateKey) -> UpdateProof`

It updates the `SRS` by calling `.update_srs()` with `PrivateKey` and produces a proof `UpdateProof`.

4. `update_srs(&mut self, private_key: Fr)`

   It is a private function that updates `SRS` elements by `private_key`. It calls the `vandemonde_challenge()` function for generating powers of `private_key` and then uses the wNAF algorithm to compute scalar multiplications.

5. `subgroup_check(&self) -> bool`

   It is used to check that the list of $\mathbb{G}_1$ and $\mathbb{G}_2$ elements are in the prime order subgroup by calling `is_in_correct_subgroup_assuming_on_curve(p: &GroupAffine<Parameters>)` on each point.

6. `verify_update(before: &SRS, after: &SRS, update_proof: &UpdateProof) -> bool`

   It is a special case of `verify_updates()` where a single update is involved.

7. `verify_updates(before: &SRS, after: &SRS, update_proofs: &[UpdateProof]) -> bool`

   It verifies whether the transition from one SRS to the other was valid by the `UpdateProof`.

   There are five points to check in this function:

   - Non-empty check
   - Consistency check for `update_proofs` and the `after` SRS
   - Proofs verification by `verify_chain` function
   - Non-zero check
   - Structure check for `after` SRS

8. `structure_check(&self) -> bool`

   It is a private function checking whether `SRS` conforms with the powers of tau structure. It performs pair-wise checks on $\mathbb{G}_1$ elements by comparing `pairing(tau_i_next, tau_g2_0)` and `pairing(tau_i, tau_g2_1)`, and likewise on $\mathbb{G}_2$ elements, `pairing(tau_g1_0, tau_i_next)` and `pairing(tau_g1_1, tau_i)`.

9. `structure_check_opt(&self, random_element: Fr) -> bool`

   It is a private function and an optimized version of `structure_check` based on the note. It constructs linear combinations of elements in `tau_g1` and `tau_g2`. It probabilistically checks the structure of `SRS` using only two pairings.

update_proof.rs

The `update_proof.rs` defines a struct `UpdateProof` which stores a degree-1 element `new_accumulated_point` of the updated SRS and a witness of the secret value tau as `commitment_to_secret`.

```
pub struct UpdateProof {
    // A commitment to the secret scalar `p`
    pub(crate) commitment_to_secret: G2Projective,
    // This is the degree-1 element of the SRS after it has been
    // updated by the contributor
    pub(crate) new_accumulated_point: G1Projective,
}
```

Only one method is implemented for the `UpdateProof`, which verifies a list of `UpdateProof`.

1. `verify_chain(starting_point: G1Projective,update_proofs: &[UpdateProof],) -> bool`

   It uses `SharedSecretChain` as a subroutine. It accumulates `update_proofs` to a `SharedSecretChain`, and then calls `verify()` on the chain.

shared_secret.rs

The shared_secret.rs contains a struct `SharedSecretChain` which reconstructs a chain of proofs from a specified starting point. A shared secret proof proves that a point was necessarily created by multiplying the discrete log of a series of previous points.

```
pub struct SharedSecretChain {
    accumulated_points: Vec<G1Projective>,
    witnesses: Vec<G2Projective>,
}
```

There are four methods implemented for the `SharedSecretChain`.

1. `starting_from(starting_point: G1Projective) -> Self`

   It creates a `SharedSecretChain` using a $\mathbb{G}_1$ point as the first element.

2. `extend(&mut self, new_accumulated_point: G1Projective, witness: G2Projective)`

   It extends a shared secret chain with a new accumulated point and a new witness.

3. `remove_last(&mut self)`

It removes the last element of `accumulated_points` and `witnesses` in `SharedSecretChain`. It is for testing.

4. `verify(&self) -> bool`

   It verifies a shared secret chain such that each $\mathbb{G}_1$ point in `accumulated_points` is updated from the previous one using the specified witness.

serialisation.rs

The serialisation.rs contains some functions for serializing/deserializing `SRS` or `UpdateProof` into/from JSON arrays.

1. `serialise(&self) -> (Vec<String>, Vec<String>)`

   This function use the `.to_json_array()` method to serialize `SRS` or `UpdateProof` into JSON array.

2. `to_json_array(&self) -> (Vec<String>, Vec<String>)`

   It is a private function. It converts `SRS` or `UpdateProof` into two string vectors.

3. `deserialise(json_arr: (Vec<String>, Vec<String>),parameters: Parameters,) -> Option<Self>`

   This function use the `.from_json_array()` method to deserialize`json_arr` to `SRS` or `UpdateProof` structure.

4. `from_json_array(json_array: (Vec<String>, Vec<String>),parameters: Parameters,) -> Option<Self>`

   It is a private function. This function converts a `String` to a point element in $\mathbb{G}_1$ or $\mathbb{G}_2$.

sdk.rs

The sdk.rs contains a struct `Transcript` using the parameters specified in the kzg-ceremony-specs.

```
pub struct Transcript {
    sub_ceremonies: [SRS; NUM_CEREMONIES],
}
```

```
const NUM_CEREMONIES: usize = 4;

const CEREMONIES: [Parameters; NUM_CEREMONIES] = [
    Parameters {
        num_g1_elements_needed: 4096,
        num_g2_elements_needed: 65,
    },
    Parameters {
        num_g1_elements_needed: 8192,
        num_g2_elements_needed: 65,
    },
    Parameters {
        num_g1_elements_needed: 16384,
        num_g2_elements_needed: 65,
    },
    Parameters {
        num_g1_elements_needed: 32768,
        num_g2_elements_needed: 65,
    },
];
```

There are two functions defined in the sdk.rs.

1. `update_transcript(mut transcript: Transcript,secrets: [String; NUM_CEREMONIES],) -> Option<(Transcript, [UpdateProof; NUM_CEREMONIES])>`

   It takes a `Transcript` object and a list of hex-encoded secrets. First, it confirms that each `SRS` parameters are correct and decodes the hex-encoded secrets to `PrivateKey` structs. Then it updates the `Transcript` using `.update()` method on `SRS`, and finally returns a new `Transcript`.

2. `transcript_subgroup_check(transcript: Transcript) -> bool`

   It performs the prime-ordered subgroup checks on a `Transcript` object by calling `.subgroup_check()` for each `SRS`.

4. Audit Process

This part describes the audit steps and audit content.

4.1 Audit Steps

The audit strictly followed the audit specification of SECBIT Lab. The process consists of the following steps:

- Background research by going through all documentation and external references.
- Comparative evaluation of code versus related specification and documentation.
- Fully analysis of source code line by line.
- Evaluation of vulnerabilities and potential risks revealed in the source code.
- Communication on assessment and confirmation.
- Audit report writing.

4.2 Audit Scope and Checklist

The following are the audit goals and the scope of the audit.

4.2.1 Specs

- Check the correctness of the *Powers of Tau Specification* document by CarlBeek
- Check the correctness of the underlying algorithm described in the document *Powers of Tau - Notes* written by kevaundray

Note: The Powers of Tau Specification also contains specs for other parts of the KZG ceremony besides the cryptographic module, which are not included in the audit scope. We reviewed the files named `participant.md`, `coordinator.md`, `BLS.md`, and `sdk.md` in the repository for this audit.

4.2.2 Code

- Check consistency between specs and algorithm description notes mentioned above
- Check consistency between specs and the *small-powers-of-tau* code
- Check the correctness and security of the *small-powers-of-tau* code
- Analyze the potential risk of using *small-powers-of-tau* as a library
- Check whether the libraries that the code depends on have known vulnerabilities that could affect this project and whether they have had recent suspicious commits
- Check for recent threats of supply chain-like attacks related to the underlying codebase

Note: The ceremony-specs written by kevaundray are used as another reference implementation to help deepen the understanding of the code and double-check the consistency between specs and code.

4.2.3 The Essential Steps in Detail

The `Coordinator` and `Contributor` are vital actors in the KZG ceremony process. The *Powers of Tau Specification* document describes these two actors and what they should do. We carefully checked the code implementation against the specs for consistency between them. Some of the specs described were not in the scope of the code we targeted for audit. We have listed below the essential steps related to the code.

4.2.3.1 Coordinator

- Point Checks
  - Prime Subgroup checks
    - G1 Powers Subgroup check
    - G2 Powers Subgroup check
    - Witness Subgroup checks
  - Non-zero check
  - Witness continuity check
- Pairing Checks
  - Running Product construction
  - Correct construction of G1 Powers
  - Correct construction of G2 Powers

4.2.3.2 Contributor (aka. Participant)

- Verifying the transcript
    - Point Checks
        - G1 Powers Subgroup check
        - G2 Powers Subgroup check
        - Running Product Subgroup check
- Updating the transcript
    - Generate the secrets
    - Update Powers of Tau
        - Multiply each of the powers_of_tau.g1_powers by incremental powers of x and overwrite the powers_of_tau.g1_powers in the transcript
        - Multiply each of the powers_of_tau.g2_powers by incremental powers of x and overwrite the powers_of_tau.g2_powers in the transcript
    - Update Witness
        - Multiply witness.running_products[-1] by x and append it to the witness.running_products list
        - Append bls.G2.mul(x, bls.G2.g2) to witness.pot_pubkeys
    - Clearing the memory

4.3 Secure Rust Development

The *small-powers-of-tau* code written in Rust is the audit target for the ceremony deployment in production.

We checked this code with the standard of secure Rust development. Specifically, we refer to the checklist from the *Secure Rust Guideline*.

Below is a list of the highlighted items we have focused on for this project during the audit.

4.3.1 Libraries

- Check for outdated dependencies versions (cargo-outdated) (LIBS-OUTDATED)
- Check for security vulnerabilities report on dependencies (cargo-audit) (LIBS-AUDIT)
- Check for unsafe code in dependencies (LIBS-UNSAFE)

4.3.2 Language Generalities

- Don't use unsafe blocks (LANG-UNSAFE)
- Use appropriate arithmetic operations regarding potential overflows (LANG-ARITH)
- Use the?operator and do not use thetry!` macro (LANG-ERRDO)
- Don't use functions that can cause `panic!` (LANG-NOPANIC)
- Test properly array indexing or use the `get` method (LANG-ARRINDEXING)

4.3.3 Memory Management

- Do not use `forget` (MEM-FORGET)
- Use clippy lint to detect use of `forget` (MEM-FORGET-LINT)
- Do not leak memory (MEM-LEAK)
- Do release value wrapped in `ManuallyDrop` (MEM-MANUALLYDROP)
- Always call `from_raw` on `into_raw` ed value (MEM-INTOFROMRAW)
- Do not use uninitialized memory (MEM-UNINIT)
- Zero out the memory of sensitive data after use (MEM-ZERO)

4.3.4 Type System

- Justify `Drop` implementation (LANG-DROP)
- Do not panic in `Drop` implementation (LANG-DROP-NO-PANIC)
- Do not allow cycles of reference-counted `Drop` (LANG-DROP-NO-CYCLE)
- Do not rely only on `Drop` to ensure security (LANG-DROP-SEC)
- Justify `Send` and `Sync` implementation (LANG-SYNC-TRAITS)
- Respect the invariants of standard comparison traits (LANG-CMP-INV)
- Use the default method implementation of standard comparison traits (LANG-CMP-DEFAULTS)
- Derive comparison traits when possible (LANG-CMP-DERIVE)

4.4 Out of Scope

The following are excluded from the audit scope due to time constraints and workload considerations.

- Security of algorithms and underlying implementations of the dependent libraries
- External academic papers referenced by the code are assumed to be safe, and only their consistency with the code will be checked, not the correctness of the papers
- Logical correctness of the code being called by external users as a library
- Security related to coordinator software and processes

5. Audit Result

This part describes the audit's detailed results and demonstrates the issues.

5.1 Summary of Audit Findings

We briefly summarize the findings of the audit process as follows.

5.1.1 Compiler and Linter Warnings

We inspected all warnings reported by the Rust compiler and Clippy Linter to ensure there were no threats to the security of the *small-powers-of-tau* code.

5.1.2 Dependencies

We used tools like `cargo-outdated` and `cargo-audit` to check the dependency security of the code quickly.

The output of cargo-outdated.

```
Name                             Project  Compat   Latest   Kind
Platform
----                             -------  ------   ------   ----    -----
---
ahash->once_cell                 1.13.0   1.13.1   1.13.1   Normal
cfg(not(all(target_arch = "arm", target_os = "none")))
atty->libc                       0.2.131  0.2.132  0.2.132  Normal
cfg(unix)
bstr->serde                      1.0.143  1.0.144  1.0.144  Normal  ---
cpufeatures->libc                0.2.131  0.2.132  0.2.132  Normal
aarch64-apple-darwin
criterion->plotters              0.3.2    0.3.3    0.3.3    Normal  ---
criterion->serde                 1.0.143  1.0.144  1.0.144  Normal  ---
criterion->serde_derive          1.0.143  1.0.144  1.0.144  Normal  ---
criterion->serde_json            1.0.83   1.0.85   1.0.85   Normal  ---
crossbeam-epoch->once_cell       1.13.0   1.13.1   1.13.1   Normal  ---
```

```
crossbeam-utils->once_cell       1.13.0   1.13.1   1.13.1   Normal  ---
csv->serde                       1.0.143  1.0.144  1.0.144  Normal  ---
getrandom->libc                  0.2.131  0.2.132  0.2.132  Normal
cfg(unix)
hermit-abi->libc                 0.2.131  0.2.132  0.2.132  Normal  ---
itertools->either                1.7.0    1.8.0    1.8.0    Normal  ---
num_cpus->libc                   0.2.131  0.2.132  0.2.132  Normal
cfg(not(windows))
plotters->plotters-svg           0.3.2    0.3.3    0.3.3    Normal  ---
rand->libc                       0.2.131  0.2.132  0.2.132  Normal
cfg(unix)
rayon->either                    1.7.0    1.8.0    1.8.0    Normal  ---
semver-parser->pest              2.2.1    2.3.0    2.3.0    Normal  ---
serde_cbor->serde                1.0.143  1.0.144  1.0.144  Normal  ---
serde_json->serde                1.0.143  1.0.144  1.0.144  Normal  ---
sha2->cpufeatures                0.2.2    0.2.4    0.2.4    Normal
cfg(any(target_arch = "aarch64", target_arch = "x86_64", target_arch =
"x86"))
tinytemplate->serde              1.0.143  1.0.144  1.0.144  Normal  ---
tinytemplate->serde_json         1.0.83   1.0.85   1.0.85   Normal  ---
wasm-bindgen-backend->bumpalo    3.10.0   3.11.0   3.11.0   Normal  ---
wasm-bindgen-backend->once_cell  1.13.0   1.13.1   1.13.1   Normal  ---
```

The output of cargo-audit.

```
    Scanning Cargo.lock for vulnerabilities (108 crate dependencies)
Crate:      serde_cbor
Version:    0.11.2
Warning:    unmaintained
Title:      serde_cbor is unmaintained
Date:       2021-08-15
ID:         RUSTSEC-2021-0127
URL:        https://rustsec.org/advisories/RUSTSEC-2021-0127
Dependency tree:
serde_cbor 0.11.2
└── criterion 0.3.6
    └── small-powers-of-tau 0.1.0

warning: 1 allowed warning found
```

We have reviewed the results individually and found no issues that affect this project. We also checked the recent commits of all the dependencies for about half a year and found no security vulnerabilities related to these dependencies.

5.1.3 Test Cases

To confirm their validity, we reviewed all the test cases in the *small-powers-of-tau* code. We also checked the test coverage and added test cases or double-checked the code that was not covered.

5.1.4 Fuzz Testing

We fuzz-tested the main interface of *small-powers-of-tau* code to see if it would panic on a specific input. We found several potential overflows and out-of-bounds errors. Details are documented in the next section.

5.1.5 Secure Memory Zeroing for Private Keys

We checked if the Rust code correctly handled sensitive information in memory as described in the specs. We found that the code uses the `ZeroizeOnDrop` trait of the zeroize zero crate to automatically and securely zeroing memory. It is a good and common practice.

5.1.6 Specs vs. Code

We found minor typo-related errors in *Powers of Tau Specification* and reported them to the team for confirmation.

During the audit, we found that the specs and the code did not align in some small spots. The main reason for this was that they did not handle the witness in the same way. We reported these issues to the team and documented them in this report.

In addition, the *small-powers-of-tau* code does not have an apparent role abstraction as defined in the Spec documentation. Since some of the key steps are included in the lower-level functions, we recommend that the application using *small-powers-of-tau* as a library requires careful reference to the specs to ensure the correctness of each step. As the underlying cryptography library, the *small-powers-of-tau* also does not handle the validation of transcript files or JSON Schema. After confirming with the developer, these should be accomplished by the upper-level applications.

Update: A transcript check function is added in commit 80ba440.

5.1.7 Supply Chain Attack Check

We inspected the commit records of some essential dependencies over the recent year and found no apparent traces of supply chain attacks.

We do this to reduce the risk of this project being targeted or poisoned by the supply chain, i.e., someone disrupting the ceremony by inserting malicious code into the dependencies.

The start and end commit hash of the dependencies we checked are as follows.

- serde
  - First commit: a6690ea2fe83924e5bb37bbb6a1341444d26a65b
  - Last commit: d208762c81883a181e8c6a9ca3f303e040105c7d
- rayon
  - First commit: f9292a7efc3dfb083cb1f22a8749c2944d263a32
  - Last commit: c00b997fc5f47ccd70dff99ab341e8da71f849d9
- itertools
  - First commit: 599ae8cced7ede91c85a0e507808e8dcbf1acd27
  - Last commit: 677900a0dd817638db718faa8e26b8df3b99cf07
- zeroize
  - First commit: ec11298b5556c898cf81a35ed023630e9dd3003c
  - Last commit: 1261e29ff91aeb26b010832ea6a855c0f360ea04
- hex
  - First commit: bfe146e0bb92ae99c302f0e5e87115e1036f37ad
  - Last commit: c333cf5128b6f5135d8f561b217f68e670275031

5.2 Issues

1. Potential overflow or array out-of-bounds errors may cause panic

   Security Risk Medium

   - Description: Multiple branches in the code can enter overflow or array out-of-bounds errors. Due to Rust's protection mechanism, the entire program will panic.

```
fn vandemonde_challenge(x: Fr, n: usize) -> Vec<Fr> {
    let mut challenges: Vec<Fr> = Vec::with_capacity(n);
    challenges.push(x);
    for i in 0..n - 1 { // @audit n=0, then panic with 'attempt to subtract with overflow' error
        challenges.push(challenges[i] * x);
    }
    challenges
}
```

```
pub fn verify_updates(before: &SRS, after: &SRS, update_proofs:
&[UpdateProof]) -> bool {
    ...
    if after.tau_g1[1] ! = last_update.new_accumulated_point {
        return false;
    }

   // 2. Check the update proofs are correct and form a chain
of updates
    if !UpdateProof::verify_chain(before.tau_g1[1],
update_proofs) {
        return false;
    }
    if after.tau_g1[1].is_zero() { // @audit could panic with
'index out of bounds' error
        return false;
    }
    if after.tau_g2[1].is_zero() {
        return false;
    }
    ...
}

fn structure_check(&self) -> bool {
    let tau_g2_0 = self.tau_g2[0];
    let tau_g2_1 = self.tau_g2[1];

    let tau_g1_0 = self.tau_g1[0];
    let tau_g1_1 = self.tau_g1[1];
    ...
}

pub fn update(&mut self, private_key: PrivateKey) ->
UpdateProof {
    ...
    let updated_tau = self.tau_g1[1];
    ...
}

pub fn structure_check_opt(&self, random_element: Fr) -> bool {
    ...
    let tau_g2_0 = self.tau_g2[0];
    let tau_g2_1 = self.tau_g2[1];
    let tau_g1_0 = self.tau_g1[0];
    let tau_g1_1 = self.tau_g1[1];
    ...
}
```

- Consequence

  In extreme cases, misusing this library can cause the whole program to panic, leading to a DOS risk.

- Suggestion

  Given that resisting DOS is vital for this project, we recommend adding more checks. It is recommended that the `SRS::new()` function be given an upper and lower length check for `tau_g1` and `tau_g2`. We explained the reason for adding a lower bound above. The upper bound prevents someone from passing in a value that is too large, causing a memory allocation error, or causing the program to take an infinitely long time to execute. These basic checks help reduce the risk of DOS. We also recommend adding a length check to `SRS::verify_updates()` since it accepts dynamic size arrays directly.

- Status

  The team has adopted this suggestion, and added length check for `tau_g1` and `tau_g2` at the `SRS::new()` function and the `SRS::deserialise` function.

2. Redundant array allocation

   Code Optimization Low

   - Description

     The vandemonde_challenge() function calculates a set of values in a challenge value by doing exponential operations with increasing exponents. It is used at `update_srs()` function(#L75) and `vandemonde_challenge()` function(#L185).

     The parameter n, representing the array's length, is `max_number_elements` in both places, but `max_number_elements-1` is enough.

```
// #L75
let powers_of_priv_key = vandemonde_challenge(private_key,
max_number_elements);
// #L185
let rand_pow = vandemonde_challenge(random_element,
max_number_elements);
```

```
fn vandemonde_challenge(x: Fr, n: usize) -> Vec<Fr> {
    let mut challenges: Vec<Fr> = Vec::with_capacity(n);
    challenges.push(x);
    for i in 0..n - 1 {
        challenges.push(challenges[i] * x);
    }
    challenges
}
```

- Consequence

  It won't cause an error. But it will simply make the for loop in the vandemonde_challenge() function do one more unnecessary operation.

- Suggestion

  Change both `max_number_elements` to `max_number_elements - 1`

- Status

  The team has adopted this suggestion and changed both `max_number_elements` to `max_number_elements - 1`

3. Unfixed dependencies versions

   Code Optimization Low

   - Description

     In Cargo.toml, the version of hex and zeroize are "*", which is rather dangerous because we are not sure whether the future updates of the hex repository will cause small-powers-of-tau not to work.

     And the official documentation of rust also mentions that 'Avoid * requirements, as they are not allowed on crates.io, and they can pull in SemVer-breaking changes during a normal cargo update.'

   - Consequence

     Future changes to these two dependencies may cause the small-powers-of-tau library not to work as expected.

   - Suggestion

     Change to a fixed version.

   - Status

     The team has adopted this suggestion and replaced the version of "*" with a fixed version.

4. Redundant file

   Code Optimization Info

   - Description

     The `ceremony.rs` file is not referenced as a module by `lib.rs`, so the contents in this file will not work.

   - Consequence

     Unnecessary file.

   - Suggestion

     Remove the `ceremony.rs` file.

   - Status

     The team has adopted this suggestion and removed the `ceremony.rs` file.

5. Redundant functions

   Code Optimization Info

   - Description

     In the `serialisation.rs` file, two `from_bytes()` and two `to_bytes()` functions are not used.

```
impl SRS {
  fn to_bytes(&self) -> Vec<u8> {
   ...
  }
  fn from_bytes(bytes: &[u8], parameters: Parameters) ->
Option<Self> {
     ...
  }
}

impl UpdateProof {
  fn to_bytes(&self) -> Vec<u8> {
     ...
  }
  fn from_bytes(bytes: &[u8]) -> Option<Self> {
     ...
  }
}
```

- Suggestion

  Remove those functions.

- Status

  The team has adopted this suggestion and removed those functions.

6. Typo in kzg-ceremony-specs(Correct construction of G1/G2 Powers)

   Code Optimization Low

   - Description

     In the `coordinator.md` file in the kzg-ceremony-specs. There are some typos in the pseudo code of `Correct construction of G1 Powers` and `Correct construction of G2 Powers`.

     ```
     // in "Correct construction of G1 Powers"
     def g1_powers_check(transcript: Transcript) -> bool:
         for sub_ceremony in transcript.sub_ceremonies:
             powers = sub_ceremony.powers_of_tau.g1_powers
             pi = sub_ceremony.witness.running_products[-1]
             for power, next_power in zip(powers[:-1], powers[1:]):
                 if bls.pairing(pi, power) != bls.pairing(bls.G1.g1,
     next_power):
                     return False
         return True
     ```

     Both `next_power` and `power` are the G1 points, which are used as the G2 points in `bls.pairing(pi, power) != bls.pairing(bls.G1.g1, next_power)`.

     The `pi` is a G2 point, which is used as a G1 point.

     ```
     // in "Correct construction of G2 Powers"
     def g2_powers_check(transcript: Transcript) -> bool:
         for sub_ceremony in transcript.sub_ceremonies:
             g1_powers = sub_ceremony.powers_of_tau.g1_powers
             g2_powers = sub_ceremony.powers_of_tau.g2_powers
             for g1_power, g2_power in zip(g1_powers, g2_powers):
                 if bls.pairing(bls.G1.g1, g1_power) !=
     bls.pairing(g2_power, bls.G2.g2):
                     return False
         return True
     ```

     `g1_power` is a G1 point, and `g2_power` is a G2 point, but both are misused.

The team fixed the first mistake in commit 50b5e12, but it is still inaccurate.

```
// in "Correct construction of G1 Powers"
if bls.pairing(bls.G1.g1, next_power) != bls.pairing(power,
pi):
```

In "Correct construction of G1 Powers", the G1 point `next_power` is used as a G2 point.

- Suggestion

  Change the pseudo code as follows.

```
// in "Correct construction of G1 Powers"
if bls.pairing(next_power, bls.G2.g2) != bls.pairing(power,
pi):

// in "Correct construction of G2 Powers"
if bls.pairing(bls.G1.g1, g2_power) != bls.pairing(g1_power,
bls.G2.g2):
```

- Status

  The team has adopted this suggestion and fixed those typos.

7. Typo in kzg-ceremony-specs(Running Product Subgroup check)

   Document Optimization Info

- Description

  There is a typo in the `Running Product Subgroup check` pseudo code of the `participant.md` file in the kzg-ceremony-specs.

```
// Running Product Subgroup check
if not
bls.G1.is_in_prime_subgroup(sub_ceremony.witness.running_produc
ts[:-1]):
        return False
```

`[:-1]` in python means "Remove the last element of the list". To "get the last element of the list", we should use `[-1]`.

- Suggestion

  Change the pseudo code as follows.

```
// Running Product Subgroup check
if not
bls.G1.is_in_prime_subgroup(sub_ceremony.witness.running_produc
ts[-1]):
       return False
```

- Status

  The team has adopted this suggestion and fixed those typos.

8. Inconsistency between Rust code and kzg-ceremony-specs about 'Witness Subgroup checks'

   Document Optimization Info

   - Description

     The coordinator must do 'Witness Subgroup checks'(For each of the points in witness, check that they are elements of their respective subgroups.) according to the kzg-ceremony-specs. But in the Rust code, there is no such check at all.

   - Consequence

     Inconsistency between Rust code and kzg-ceremony-specs

   - Suggestion

     Add the check.

   - Status

     The team explains the issue. This check was supposed to be added to match the specs, but it's not necessary for the actual code because the pairings check would fail if the witness were not in the correct group.

9. Inconsistency between Rust code and kzg-ceremony-specs about Transcript

   Document Optimization Info

   - Description

     All the witnesses should be transferred between the coordinator and participants according to the kzg-ceremony-specs. But in the Rust code, participants do not need to download witnesses and only upload their witnesses to the coordinator.

   - Consequence

     Inconsistency between Rust code and kzg-ceremony-specs

  - Suggestion

    Change the kzg-ceremony-specs to match the Rust code.

  - Status

    The team explains this issue. The full transcript is no longer sent to participants. Only a contribution.json file will be sent, a stripped-down version that doesn't contain the complete witnesses, just the necessary powers.

6. Conclusion

This trusted setup ceremony is the first step in a series of essential cryptographic-related upgrades to Ethereum. The SECBIT team audited the core cryptography module of the ceremony. After carefully reviewing the specification and source code, the SECBIT team found no fatal bugs or flaws. The specification is well documented, and the code is concise and efficient. The SECBIT team found some issues and proposed corresponding suggestions, as shown above. The SECBIT team reported to the development team and confirmed with them. They were responsive and fixed promptly. The SECBIT team then confirmed that the updated code addressed all of the issues raised in the report. Besides, we recommend that any developer who needs to use the cryptography library of this project should read the relevant documentation carefully to ensure that the overall process is understood correctly.

Disclaimer

The security audit service by SECBIT Labs assesses the code's correctness, security, and performability in code quality, logic design, and potential risks. The report is provided "as is", without any warranties about the code practicability, business model, management system's applicability, and anything related to the contract adaptation. This audit report is not to be taken as an endorsement of the platform, team, company, or investment.

APPENDIX

Appendix 1: Vulnerability/Risk Level Classification

Level	Description
High	Severely damage to the system's integrity
Medium	Damage to the application's security under given conditions
Low	Cause no actual impairment to the application
Info	Relevant to practice or rationality of the code could possibly bring risks.
Discussion	Some suggestions for optimizing the code logic

Appendix 2: Type Classification

Type	Description
Security Risk	The risk of compromising system security directly
Code Optimization	Optimize code implementation
Logical Implementation	Vulnerability in design or implementation logic
Potential Risk	The potential risk of compromising system security
Code Revising	Non-standard usage of code writing
Document Optimization	Optimize the description in the documents

SECBIT Labs is devoted to construct a common-consensus, reliable and ordered blockchain economic entity.

🌎 http://www.secbit.io

✉ audit@secbit.io

@secbit_io