Groth16 Proof Aggregation: Cryptography and Implementation Review

Protocol Labs

June 2, 2021 – Version 1.1 Retest

Prepared for

Friedel Ziegelmayer
Nicolas Gailly
Deep Kapur

Prepared by

Eric Schorn
Paul Bottinelli
Kevin Henry

Executive Summary

Synopsis

During April 2021, Protocol Labs engaged NCC Group's Cryptography Services team to conduct a cryptography and implementation review of the Groth16 proof aggregation functionality in the `bellperson` and two other related GitHub repositories. This code utilizes inner product arguments to efficiently aggregate existing Groth16 proofs while re-using existing powers of tau ceremony transcripts. Full source code access was provided with support over Slack. The project concluded with a brief retest several weeks after the initial review.

Scope

NCC Group's evaluation included the following primary materials:

- The `bellperson` GitHub repository, commits `d412078` and `854f254` on branch `feat-ipp2`, containing source code in `src/groth16/aggregate/*`
- The `filecoin-ffi` GitHub repository, commit `c5e646e` on branch `feat-aggregation2`, containing source code in `rust/src/proofs/api.rs` with the primary functions `fil_aggregate_seal_proofs()` and `fil_verify_aggregate_seal_proof()` (along with downstream logic)
- The `rust-fil-proofs` GitHub repository, commit `ab958b5` on branch `feat-aggregation`, containing source code in `filecoin-proofs/src/api/seal.rs` with the primary functions `aggregate_seal_commit_proofs()` and `verify_aggregate_seal_commit_proofs()` (along with downstream logic)
- The technical paper titled "Proofs for Inner Pairing Products and Applications" by Bünz et al.
- The private/preliminary technical paper titled "Proposal: Practical Groth16 Aggregation"[1]
- Retest performed on `bellperson` Pull Request 179,[2] `filecoin-ffi` Pull Request 169[3] and the technical paper "SNARKpack: Practical SNARK Aggregation" by Gailly, Maller and Nitulescu

The testing methodology revolved around documentation review and manual source code review augmented by fuzzing of selected components.

Limitations

While the target source code is part of a much larger out-of-scope system undergoing rapid development, NCC Group was able to achieve robust coverage of all in-scope components.

Key Findings

The in-scope code was well organized and supported by detailed technical documentation. Implementing the target functionality in Rust prevents many common memory-safety related errors, allows for safer use of concurrency and provides for a straightforward build/test environment. Nonetheless, the review uncovered a total of eleven findings with the most notable including:

- Several instances of `panic` from malformed or malicious input that could present denial of service attack vectors or prevent a graceful recovery from errors.
- A known-constant value present in the random vector that determines the linear combinations of proof elements for the inner product process.
- The potential for memory churn and/or exhaustion during the deserialization of carefully crafted objects.
- Input validation checks utilizing `debug_assertions` that the compiler removes during a release build.

After retesting, NCC Group found **all findings were fixed, with one exception** of an informational observation involving toolchain and dependency versioning. This latter issue is expected to be fixed via a normal periodic updating process. Additional informational material is included as Appendix B on page 27.

Strategic Recommendations

Beyond addressing the reported findings, NCC Group recommends prioritizing the following areas during future development:

- Continue the focus on data validation for all input supplied at the API level and also whenever possible within intermediate functions (such as ensuring the lists provided to `zip` are of equal and non-zero lengths). Ensure range checks are implemented for both minimum and maximum expected values/sizes when appropriate.
- To better avoid denial of service scenarios stemming from unanticipated or illegal operations, prefer a `Result` or `Option` over an `assert` which may panic at runtime, and over a `debug_assert` which will be removed during a release build. Continue utilizing the `ensure!()` macro, while avoiding `unwrap()` and `expect()` statements.
- Tighten the externally-visible API by reviewing the visibility of internal functions, perhaps replacing `pub` with `pub(crate)` when possible.

[1] File `Aggregate_Groth16_via_IPP.pdf` dated March 11, 2021 with shasum `ac40456...`
[2] https://github.com/filecoin-project/bellperson/pull/179
[3] https://github.com/filecoin-project/filecoin-ffi/pull/169/commits/7f4effb29d1b0bd78125049c99da21839e778da3

Dashboard

nccgroup

Target Metadata

Name	Filecoin: Groth16 Proof Aggregation
Type	Cryptographic Library
Platforms	Rust with C FFI
Environment	Development

Engagement Data

Type	Cryptography and Implementation Review
Method	Manual Source Code Review, Fuzzing
Dates	2021-04-05 to 2021-04-16
Consultants	3
Level of Effort	15 person-days

Targets

Bellperson	https://github.com/filecoin-project/bellperson/
Filecoin-FFI	https://github.com/filecoin-project/filecoin-ffi/
Rust-Fil-Proofs	https://github.com/filecoin-project/rust-fil-proofs/

Finding Breakdown

Critical issues	0
High issues	0
Medium issues	5
Low issues	4
Informational issues	2
Total issues	**11**

Category Breakdown

Cryptography	5
Data Validation	1
Denial of Service	3
Error Reporting	1
Patching	1

Component Breakdown

all	1
bellperson	9
filecoin-ffi	1

Key

Critical High Medium Low Informational

Table of Findings

For each finding, NCC Group uses a composite risk score that takes into account the severity of the risk, application's exposure and user population, technical difficulty of exploitation, and other factors. For an explanation of NCC Group's risk rating and finding categorization, see Appendix A on page 25.

Title	Status	ID	Risk
Constant Entry in Randomness Vector(s)	Fixed	003	Medium
Brittle Input Validation via Debug Assertions	Fixed	005	Medium
DoS in Aggregated Proof Verification via Malformed Proof	Fixed	006	Medium
Uncaught Panic in FFI Code	Fixed	007	Medium
Memory Exhaustion via Malformed Structured Reference String	Fixed	009	Medium
Potential DoS via Inverse Computation Panic	Fixed	002	Low
Panic when Aggregating Proofs of Length One	Fixed	004	Low
Aggregate Proof Malleability	Fixed	008	Low
Discrepancy between Reference and Implementation in KZG Challenge Point Computation	Fixed	011	Low
Marginally Inconsistent/Outdated Toolchain and Dependencies	Not Fixed	001	Informational
Missing Domain Separation Parameter in Oracle Calls	Fixed	010	Informational

Finding **Constant Entry in Randomness Vector(s)**

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

Identifier NCC-E001405-003

Status Fixed

Category Cryptography

Component bellperson

Location
- `structured_scalar_power()` on lines 28-34 of `bellperson/src/groth16/aggregate/mod.rs`
- Lines 49-57 of `bellperson/src/groth16/aggregate/prove.rs`
- Line 109 of `bellperson/src/groth16/aggregate/verify.rs`

Impact A random r vector with the first element fixed to `1` (or r^0) will result in the first group elements of the Z_{AB} and Z_C inner product terms and the first v_1' and v_2' vector elements having no randomness applied. This impacts a central pillar of the aggregation scheme which depends upon checking a random linear combination of proof elements.

Description The technical reference paper titled "Proposal: Practical Groth16 Aggregation"[4] section 3.3 contains an overview of the aggregation protocol stating "…it is sufficient to prove that only one inner pairing product of a random linear combination of these initial equations defined by a verifier's random challenge $r \in \mathbb{Z}_p$ holds", with section 3.3.2 item 5 defining the randomness vector as $\mathbf{r} = \{r^i\}_{i=1}^n$. These statements are consistent with the approach articulated in the other technical paper "Proofs for Inner Pairing Products and Applications"[5] section 2.2.2 (although there are subsequent references to the first index and power starting at zero). The code comments on lines 51 and 53 below are aligned with the approach described above for the randomness vector and its inverse.

```
49 // Random linear combination of proofs
50 let r = oracle!(&com_ab.0, &com_ab.1, &com_c.0, &com_c.1);
51 // r, r^2, r^3, r^4 ...
52 let r_vec = structured_scalar_power(proofs.len(), &r);
53 // r^-1, r^-2, r^-3
54 let r_inv = r_vec
55     .par_iter()
56     .map(|ri| ri.inverse().unwrap())
57     .collect::<Vec<_>>();
```

However, the implementation of `structured_scalar_power()` function in `mod.rs` excerpted below initializes the first element of the `powers` vector with a fixed value of `F::one()` on line 29.

```
28 fn structured_scalar_power<F: Field>(num: usize, s: &F) -> Vec<F> {
29     let mut powers = vec![F::one()];
30     for i in 1..num {
31         powers.push(mul!(powers[i - 1], s));
32     }
33     powers
34 }
```

[4] Aggregate_Groth16_via_IPP.pdf dated March 11, 2021
[5] https://eprint.iacr.org/2019/1177.pdf

As a result, the first entries in the `r_vec` and `r_inv` vectors on lines 52 and 54 of the earlier code snippet are fixed and predictable without randomness. These vectors ultimately factor into subsequent calculations for the Z_{AB} and Z_C inner product terms, and the v_1' and v_2' vectors. While the test cases demonstrate that a constant value still works, this conflicts with the random linear combination pillar of the aggregation algorithm.

Recommendation

In the `structured_scalar_power()` function, initiate the `powers` vector with `s` rather than `F::one()` on line 29.

Separately and informationally, since the intended length of the `powers` vector is known prior to its declaration, utilizing the `Vec::with_capacity()` initializer will eliminate repeated resizing which may marginally improve performance.

Retest Results

Regarding the Client Response noted immediately below, Protocol Labs offered the newly released paper titled "SnarkPack: Practical SNARK Aggregation" at https://eprint.iacr.org/2021/529. Notably, the `i` index for constructing the randomness vector now starts at `0` in step 5 of the Prove algorithm described on page 13. This corresponds to the earlier external paper titled "Proofs for Inner Pairing Products and Applications" at https://eprint.iacr.org/2019/1177.pdf which clearly indicates that the verifier constructs the randomness vector with a `1` in the leading position (which corresponds to r^i with an initial index of `0`) on page 26. Pull Request 179 contained no code changes pertinent to this finding. While allotted time and project scope prevented NCC Group from performing a full analysis of the paper's implications, the code and technical documentation are now aligned here. As such, this finding has been marked 'Fixed'.

Client Response

Developer discussions over Slack greatly clarified the context to this finding. The constant value is proposed to be safe as-is and an internal writeup by Protocol Labs is being prepared to describe the supporting rationale. Additionally (lightly edited),

> There is an ongoing PR which is bringing the paper and the implementation into "sync": namely, before we wrote the paper our scheme was using commitment keys as g^{a^i} and $g^{a^{n+1+i}}$ for $i : [1, n]$. We realized we could use the "smaller" bases g^{a^i} and $g^{a^{n+i}}$ later on for $i : [0, n-1]$, so we wrote the paper using these bases, because it allows to aggregate more proofs. But the implementation was still using the old way. Because it led to some confusion, the PR is here to close that gap.

Finding Brittle Input Validation via Debug Assertions

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

Identifier NCC-E001405-005

Status Fixed

Category Data Validation

Component bellperson

Location
- Lines 13 and 30 of bellperson/src/groth16/aggregate/inner_product.rs
- Lines 78-81 of bellperson/src/groth16/aggregate/prove.rs
- Lines 288-291 of bellperson/src/groth16/aggregate/srs.rs

Impact Depending upon the execution path, unvalidated values from external input may result in unexpected behavior and potential Denial of Service attacks.

Description The `pairing_miller_affine()` function implemented in `inner_product.rs` returns the miller loop evaluated on pairs of inputs, as shown below. This function is usable externally as it appears to be marked `pub` from the crate root down to the function modifier[6] on line 12 shown below (though this may be a transient development artifact). Note that Rust's strict type checking ensures that proper parameter types are passed, and that the `debug_assert_eq!()` statement on line 13 validates the parameter contents.

```
12 pub fn pairing_miller_affine<E: Engine>(left: &[E::G1Affine], right:
   ➜ &[E::G2Affine]) -> E::Fqk {
13     debug_assert_eq!(left.len(), right.len());
14     let pairs = left
15         .par_iter()
16         .map(|e| e.prepare())
17         .zip(right.par_iter().map(|e| e.prepare()))
18         .collect::<Vec<_>>();
19     let pairs_ref: Vec<_> = pairs.iter().map(|(a, b)| (a, b)).collect();
20 
21     E::miller_loop(pairs_ref.iter())
22 }
```

However, the Rust compiler eliminates debug assertions such as `debug_assert_eq!` when code is built for release, thus removing the input validation. When a test case was modified to invoke the above function with unequal length parameters and run via `cargo test --release`, the error was only caught when the test case compared actual versus expected results at its conclusion. This is somewhat fortuitous, as some functions such as `copy_from_slice()` [7] will panic on malformed input, which can lead to a denial of service. In this instance, if the length of `left` and `right` lists do not match, elements from the longer list will be considered extraneous and silently ignored by the `zip` function. See also finding NCC-E001405-006 on page 9 for another example of input validation and further perspective on consequences.

Note that the function cannot return any indication of failure, such as a `Result`[8] or `Option`.

There are two additional instance of the same scenario present in the `prove.rs` and `srs.rs` source files.

[6] https://doc.rust-lang.org/reference/visibility-and-privacy.html
[7] https://doc.rust-lang.org/std/primitive.slice.html#method.copy_from_slice
[8] https://doc.rust-lang.org/std/result/

Recommendation This finding is at the intersection of several concerns – function visibility, input validation and return types. Typically, a library should only expose the minimal API necessary for its use, each function in the exposed API should carefully validate all inputs, and the return type of each function should be able to indicate failure. In this specific instance:

- Consider whether the function should be externally visible. The `pub(crate)` modifier may instead be appropriate.
- Convert the `debug_assert_eq!()` statement to a functional non-debug equivalent, and consider whether additional input attributes can be validated.
- Modify the function return type to `Result` or `Option` to support input validation failures.

Note that there are other (out of scope) `debug_assert` statements present in the codebase in `src/multicore.rs`, `src/groth16/proof.rs`, `src/groth16/multiscalar.rs` and `src/groth16/verifier.rs`.

Retest Results A review of Pull Request 179 indicates the noted `pairing_miller_affine()` function now has the `pub(crate)` visibility modifier, no longer contains a `debug_assert` and returns a `Result<>`. Separately, developer discussions indicate that the `debug_assert` in `prove.rs` is to support debugging rather than production. Similarly, the multiple `debug_assert` in `srs.rs` are contained within the `setup_fake_srs()` function which is also not a production path. As such, this finding was marked 'Fixed'.

Finding	**DoS in Aggregated Proof Verification via Malformed Proof**
Risk	**Medium** Impact: Medium, Exploitability: Medium
Identifier	NCC-E001405-006
Status	Fixed
Category	Denial of Service
Component	bellperson
Location	`bellperson/src/groth16/aggregate/verify.rs:307`
Impact	By tampering with an aggregated proof, an attacker is able to trigger a crash during the verification process, effectively exercising a Denial of Service attack on the verifier.

Description

Aggregated proofs are deeply nested structures composed of a large number of fields. All proof types are declared in the `bellperson/src/groth16/aggregate/proof.rs` file. Specifically, the structure `AggregateProof` includes commitment values, aggregators, and a structure named `TippMippProof`. The latter contains KZG openings for the v and w values, as well as another proof structure called `GipaProof`. An adversary able to tamper with elements of this last `GipaProof` structure is able to perform a DoS on the verifier.

The representation of these different proofs is provided below, for reference.

```
pub struct AggregateProof<E: Engine> {
    pub com_ab: commit::Output<E>,
    // ...
    pub tmipp: TippMippProof<E>,
}

pub struct TippMippProof<E: Engine> {
    pub gipa: GipaProof<E>,

    pub vkey_opening: KZGOpening<E::G2Affine>,

    pub wkey_opening: KZGOpening<E::G1Affine>,
}

pub struct GipaProof<E: Engine> {
    pub nproofs: u32,

    pub comms_ab: Vec<(commit::Output<E>, commit::Output<E>)>,

    pub comms_c: Vec<(commit::Output<E>, commit::Output<E>)>,

    pub z_ab: Vec<(E::Fqk, E::Fqk)>,

    pub z_c: Vec<(E::G1, E::G1)>,
    pub final_a: E::G1Affine,

    // ...
}
```

When submitting an aggregated proof for verification with an empty vector for either `tmipp.gipa.comms_ab`, `tmipp.gipa.comms_c`, `tmipp.gipa.z_ab`, or `tmipp.gipa.z_c`, a crash is triggered in the function `verify_tipp_mipp()`, on line 207, where a call to the `first()`

function returns `None`, which is then `unwrap()`-ed and leads to a panic. An excerpt of that function is provided below for reference.

```
/// verify_tipp_mipp returns a pairing equation to check the tipp proof.  $r$ is
/// the randomness used to produce a random linear combination of A and B and
/// used in the MIPP part with C
fn verify_tipp_mipp<E: Engine>(
    v_srs: &VerifierSRS<E>,
    proof: &AggregateProof<E>,
    r_shift: &E::Fr,
) -> PairingCheck<E> {

    // ...

    // (T,U), Z for TIPP and MIPP  and all challenges
    let (final_res, mut challenges, mut challenges_inv) = gipa_verify_tipp_mipp(
     ➜  &proof);

    // ...

    // KZG challenge point
    let c = oracle!(
        &challenges.first().unwrap(),
        &fvkey.0,
        &fvkey.1,
        &fwkey.0,
        &fwkey.1
    );
    // ...
```

Note that the panic is triggered relatively deep into the call stack, which might indicate some oversights in the validations of parameters. Indeed, to verify an aggregated proof, one calls the `verify_aggregate_proof()` function, which in turns calls the `gipa_verify_tipp_mipp()` function (in which no error is triggered regarding the empty fields), before finally calling `verify_tipp_mipp()`.

The reason for that crash is that the `challenges` vector returned by the `gipa_verify_tipp_mipp()` function is empty. This is because the `for`-loop in that function (highlighted in the code excerpt below), tries to iterate over one of the empty vectors, and thus immediately stops, without triggering any error.

```
fn gipa_verify_tipp_mipp<E: Engine>(
    proof: &AggregateProof<E>,
) -> (GipaTUZ<E>, Vec<E::Fr>, Vec<E::Fr>) {
    info!("gipa verify TIPP");
    // ...

    // We first generate all challenges as this is the only consecutive process
    // that can not be parallelized then we scale the commitments in a
    // parallelized way
    for ((comm_ab, z_ab), (comm_c, z_c)) in comms_ab
        .iter()
        .zip(zs_ab.iter())
        .zip(comms_c.iter().zip(zs_c.iter()))
    {
```

Recommendation Perform stricter parameter validation, as early as possible in the execution, and in particular around functions that accept potential adversarial input. Additionally, consider writing functions that validate whether structures, such as proofs, are well-formed.

Retest Results A `parsing_check()` method was added as part of Pull Request 173 (and also contained in Pull Request 179) to the `AggregateProof` structure, which ensures that proofs are well-formed.

This function is now called as a first step during the verification process performed by the `verify_aggregate_proof()` function, mitigating this finding.

Finding	**Uncaught Panic in FFI Code**
Risk	**Medium** Impact: Medium, Exploitability: Medium
Identifier	NCC-E001405-007
Status	Fixed
Category	Error Reporting
Component	filecoin-ffi
Location	`fil_verify_aggregate_seal_proof()` on lines 544-598 and `convert_aggregation_inputs()` on lines 452-472 of `filecoin-ffi/rust/src/proofs/api.rs`
Impact	An external caller to the Rust FFI function `fil_verify_aggregate_seal_proof()` may encounter a panic that does not gracefully return control (e.g., potentially a core dump).

Description

The `fil_verify_aggregate_seal_proof()` function implemented on lines 544-598 of `api.rs` is called by an external FFI function to verify the output of an aggregated seal. The function returns a pointer to a constructed `fil_VerifyAggregateSealProofResponse` struct defined in filecoin-ffi/rust/src/proofs/types.rs which includes an `is_valid` boolean field. After some initial input validation, the function calls `convert_aggregation_inputs()` on each element of a slice of the `commit_inputs`.

The `convert_aggregation_inputs()` function is implemented on lines 452-472 of `api.rs` contains a statement involving `.unwrap_or_else(|_| {  panic!(....`. When this is encountered control may not be returned to the caller in a graceful manner.

Note that other nearby functions implement a similar pattern surrounded by a `catch_unwind` function.[9] Rust documentation indicates that:

> It is currently undefined behavior to unwind from Rust code into foreign code, so this function is particularly useful when Rust is called from another language (normally C). This can run arbitrary Rust code, capturing a panic and allowing a graceful handling of the error.
>
> It is not recommended to use this function for a general try/catch mechanism. The Result type is more appropriate to use for functions that can fail on a regular basis. Additionally, this function is not guaranteed to catch all panics, see the "Notes" section below.

However, the `catch_unwind()` function is absent from the noted location. Further, lines 580-592 indicate functionality that does involve the `is_valid` boolean flag (and is thus unused for this panic case).

Recommendation

Indicate function failure through the `Result` or `Option` mechanism (e.g., through the `is_valid` flag) rather than through a panic. The `catch_unwind` mechanism can be a secondary 'defense in depth' mechanism, but should not be the primary intended functional path.

Retest Results

A review of Pull Request 169 indicates the `convert_aggregation_inputs()` function no longer panics but rather returns a `Result` per the recommendation. Additionally, the primary return path for the `fil_aggregate_seal_proofs()` function now involves a `Result<>` and also contains a `catch_panic_response` mechanism (with `catch_unwind`) similar to its siblings. As such, this finding has been marked 'Fixed'.

[9] https://doc.rust-lang.org/std/panic/fn.catch_unwind.html

Finding	**Memory Exhaustion via Malformed Structured Reference String**
Risk	**Medium** Impact: High, Exploitability: Medium
Identifier	NCC-E001405-009
Status	Fixed
Category	Denial of Service
Component	bellperson
Location	• `bellperson/src/groth16/aggregate/srs.rs:345` • `bellperson/src/groth16/aggregate/srs.rs:216`
Impact	An adversary may trigger the allocation of large amounts of memory, eventually impeding the normal behavior of processes.

Description

In the file `srs.rs`, the functions `read()` and `read_mmap()` are used to parse a Structured Reference String (SRS). Both functions use helper functions to read vectors (`read_vec()` and `mmap_read_vec()`, respectively), and follow the same pattern when doing so; they first read a 32-bit integer, and then allocate a vector of that length. This can be seen in the highlighted portion of the `read_vec()` function provided below for reference.

```
fn read_vec<G: CurveAffine, R: Read>(r: &mut R) -> io::Result<Vec<G>> {
    let vector_len = r.read_u32::<BigEndian>()? as usize;
    let mut data = vec![G::Compressed::empty(); vector_len];
    for encoded in &mut data {
        r.read_exact(encoded.as_mut())?;
    }
    Ok(data
        .par_iter()
        .map(|enc| {
            enc.into_affine()
                .map_err(|e| io::Error::new(io::ErrorKind::InvalidData, e))
                .and_then(|s| Ok(s))
        })
        .collect::<io::Result<Vec<_>>>()?)
}
```

In the case of a malformed (potentially adversarial) SRS, a large amount of memory may be incorrectly allocated, which would impede normal functioning. Specifically, a 5-byte vector may result in the allocation of more than 4GB of memory.

Recommendation

Consider enforcing an upper bound on the expected size of SRS fields when parsing them. Additionally, check possible mismatches between the parsed lengths and expected buffer sizes (possibly also enforcing `0` length checks) where appropriate.

Retest Results

A `MAX_SRS_SIZE` constant was added as part of Pull Request 173 (and also contained in Pull Request 179) to the `bellperson/src/groth16/aggregate/srs.rs` file. When reading data using the functions `read_vec()` and `mmap_read_vec()`, an error is now returned if the vector length is larger than this upper bound, mitigating this finding.

Finding	**Potential DoS via Inverse Computation Panic**
Risk	**Low** Impact: Medium, Exploitability: Low
Identifier	NCC-E001405-002
Status	Fixed
Category	Cryptography
Component	bellperson
Location	`bellperson/src/groth16/aggregate/verify.rs:73`
Impact	With overwhelmingly low probability, a panic may be triggered during the aggregated proof verification process, effectively exercising a Denial of Service attack on the verifier. This condition may also be leveraged by an attacker.

Description

The function `verify_aggregate_proof()` is the main function used to verify aggregated proofs. During the verification process, a random challenge `r` is computed and then used to check the aggregate pairing product equation. During this computation, one is subtracted to `r` before computing the inverse of that quantity. The code excerpted below highlights this process.

```
// Check aggregate pairing product equation
// SUM of a geometric progression
// SUM a^i = (1 - a^n) / (1 - a) = -(1-a^n)/-(1-a)
// = (a^n - 1) / (a - 1)
info!("checking aggregate pairing");
let mut r_sum = r.pow(&[public_inputs.len() as u64]);
r_sum.sub_assign(&E::Fr::one());
let b = sub!(r, &E::Fr::one()).inverse().unwrap();
r_sum.mul_assign(&b);
```

If the variable `r` were equal to one, the result of the subtraction would be `0`, which does not have a modular inverse in the field. Thus, the call to `inverse()` above would return `None`, which would trigger a panic when `unwrap()`-ing. However, since that variable is computed as the output of a hash function (which is used to model a random oracle based on the Fiat-Shamir heuristic, see below), the probability of it being one is negligible.

```
// Random linear combination of proofs
let r = oracle!(
    &proof.com_ab.0,
    &proof.com_ab.1,
    &proof.com_c.0,
    &proof.com_c.1
);
```

Note that the input to the random oracle may be controlled by an adversary. However, it does not seem feasible for an attacker to craft inputs hashing to such value.

The NCC Group team also noted that the implementation of the oracle currently guards against another such edge case. Specifically, if the value computed as the output of the hash function does not have an inverse, the oracle will perform additional iterations in order to generate a value for which an inverse exists.

Recommendation Consider adding a check in the oracle macro to protect against the generation of one.

Retest Results As part of Pull Request 173 (and also contained in Pull Request 179), the following check that mitigates this finding was added to `bellperson/src/groth16/aggregate/macros.rs`:

```
if c == one {
    continue;
}
```

Finding	**Panic when Aggregating Proofs of Length One**
Risk	**Low** Impact: Medium, Exploitability: Medium
Identifier	NCC-E001405-004
Status	Fixed
Category	Denial of Service
Component	bellperson
Location	`bellperson/src/groth16/aggregate/prove.rs:109`
Impact	A panic may be triggered during the proof aggregation process, effectively exercising a Denial of Service attack on the prover.

Description The proof aggregation process, implemented in the `aggregate_proofs()` function, converts an array of `n` zkSNARK proofs to an aggregate proof. Currently, the number of aggregated proofs must be a power of 2. The `aggregate_proofs()` function starts by performing a few sanity checks on its inputs before proceeding with the aggregation, as can be seen in the code excerpt below.

```rust
/// Aggregate `n` zkSnark proofs, where `n` must be a power of two.
pub fn aggregate_proofs<E: Engine + std::fmt::Debug>(
    srs: &ProverSRS<E>,
    proofs: &[Proof<E>],
) -> Result<AggregateProof<E>, SynthesisError> {
    if !proofs.len().is_power_of_two() {
        return Err(SynthesisError::NonPowerOfTwo);
    }

    if !srs.has_correct_len(proofs.len()) {
        return Err(SynthesisError::MalformedSrs);
    }

    // ...

    // Random linear combination of proofs
    let r = oracle!(&com_ab.0, &com_ab.1, &com_c.0, &com_c.1);
    // r, r^2, r^3, r^4 ...
    let r_vec = structured_scalar_power(proofs.len(), &r);
    // r^-1, r^-2, r^-3

    // ...

    // we prove tipp and mipp using the same recursive loop
    let proof = prove_tipp_mipp::<E>(&srs, &a, &b_r, &c, &wkey_r_inv, &r_vec)?;
```

However, if the number of individual zkSNARK proofs passed in to the `aggregate_proofs()` function is equal to 1, a panic will be triggered in the `prove_tipp_mipp()` function, highlighted above. Note that 1 is a power of two ($2^0 = 1$), and as such the check `proofs.len().is_power_of_two()` above will succeed.

More specifically, the `prove_tipp_mipp()` function tries to access the `r_vec` array at index 1, which is the same length as the proof array, resulting in an 'index out of bounds' panic if the latter is of length one. This is highlighted in the code excerpt below.

```
fn prove_tipp_mipp<E: Engine>(
    srs: &ProverSRS<E>,
    a: &[E::G1Affine],
    b: &[E::G2Affine],
    c: &[E::G1Affine],
    wkey: &WKey<E>, // scaled key w^r-1
    r_vec: &[E::Fr],
) -> Result<TippMippProof<E>, SynthesisError> {
    if !a.len().is_power_of_two() || a.len() != b.len() {
        return Err(SynthesisError::MalformedProofs);
    }
    let r_shift = r_vec[1].clone();
```

Reproduction Steps Change the `NUM_PROOFS` value to 1 in the test `test_groth16_aggregation()` of the file `bellperson/tests/groth16_aggregation.rs` and run the test to observe the panic.

```
const NUM_PROOFS: usize = 1;
```

Recommendation Consider adding a guard to prevent aggregation of less than 2 proofs. Additionally, consider performing stricter parameter validation, as early as possible in the function executions.

Retest Results As part of Pull Request 173 (and also contained in Pull Request 179), the following check was added to the function `aggregate_proofs()` in `bellperson/src/groth16/aggregate/prove.rs`:

```
if proofs.len() < 2 {
    return Err(SynthesisError::MalformedProofs(
        "aggregating less than 2 proofs is not allowed".to_string(),
    ));
}
```

This check now mitigates the finding.

Finding **Aggregate Proof Malleability**

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

Identifier NCC-E001405-008

Status Fixed

Category Cryptography

Component bellperson

Location `bellperson/src/groth16/aggregate/verify.rs:307`

Impact An adversary may be able to arbitrarily inflate aggregated proofs. If other components were relying on assumptions surrounding proof non-malleability, unexpected issues might occur.

Description Aggregated proofs are deeply nested structures composed of a large number of fields. All proof types are declared in the `bellperson/src/groth16/aggregate/proof.rs` file. Specifically, the structure `AggregateProof` includes commitment values, aggregators, and a structure named `TippMippProof`. The latter contains KZG openings for the v and w values, as well as another proof structure called `GipaProof`. A sibling finding (see finding NCC-E001405-006 on page 9) shows the nesting between these proofs and provides code excerpts.

An adversary able to tamper with elements of this `GipaProof` structure (given a valid instance of an `AggregateProof`) may arbitrarily inflate the proof, and the verification will still succeed. Note that this *does not mean* that an adversary is able to forge individual (or aggregated) zkSNARK proofs.

Similar to the sibling finding referred to above, the reason for the malleability of this proof structure is twofold: the `verify_aggregate_proof()` function performs limited parameter checks, and the Rust `zip()` operator on iterators will return as soon as one of the iterators is exhausted. An excerpt of vulnerable code from the `verify_aggregate_proof()` function is provided below.

```
    for ((comm_ab, z_ab), (comm_c, z_c)) in comms_ab
        .iter()
        .zip(zs_ab.iter())
        .zip(comms_c.iter().zip(zs_c.iter()))
    {
    // ...
```

As a result, when submitting an aggregated proof for verification with vectors of different sizes for either `tmipp.gipa.comms_ab`, `tmipp.gipa.comms_c`, `tmipp.gipa.z_ab`, or `tmipp.gipa.z_c`, the function will loop over elements of these fields until it reaches the end of the shortest vector, at which point it will exit. Thus, an attacker may inflate the proof structure by appending an arbitrary number of elements to either of these fields, and the verification will succeed.

The exploitability (and corresponding risk rating) of this finding was set to *Low*, since the function to read an `AggregateProof` performs appropriate checks on the fields of the proof structures and the function `verify_aggregate_proof()` is not expected to be publicly accessible, as per the Filecoin team.

Reproduction Steps In `bellperson/tests/groth16_aggregation.rs`, add the following lines to the `test_groth16_aggregation()` and observe that the test still succeeds.

```
// 1. Valid proofs
let mut aggregate_proof =
    aggregate_proofs::<Bls12>(&pk, &proofs).expect("failed to aggregate proofs");

// Adding some vector elements
aggregate_proof.tmipp.gipa.comms_c.push(aggregate_proof.tmipp.gipa.comms_c[0]);
aggregate_proof.tmipp.gipa.z_ab.push(aggregate_proof.tmipp.gipa.z_ab[0]);
aggregate_proof.tmipp.gipa.z_c.push(aggregate_proof.tmipp.gipa.z_c[0]);

let result = verify_aggregate_proof(&vk, &pvk, &mut rng, &statements,
 ➜  &aggregate_proof)
        .expect("these proofs should have been valid");
assert!(result);
```

Recommendation Consider adding checks surrounding the lengths of the different parameters in the `verify_aggregate_proof()` for the purpose of defense in depth.

Retest Results A `parsing_check()` method was added as part of Pull Request 173 (and also contained in Pull Request 179) to the `AggregateProof` structure, which ensures that proofs are well-formed.

This function is now called as a first step during the verification process performed by the `verify_aggregate_proof()` function, mitigating this finding.

Finding	**Discrepancy between Reference and Implementation in KZG Challenge Point Computation**
Risk	**Low** Impact: Low, Exploitability: Medium
Identifier	NCC-E001405-011
Status	Fixed
Category	Cryptography
Component	bellperson
Location	• `bellperson/src/groth16/aggregate/prove.rs:124` • `bellperson/src/groth16/aggregate/verify.rs:196`
Impact	Discrepancies between reference and implementation may invalidate security proofs, resulting in potential flaws in the protocol.

Description

During the proof aggregation process, commitment to a polynomial using KZG is performed, which utilizes the output of a random oracle as a challenge point.

The computation of the challenge is performed during the proof aggregation process in `prove.rs`, on line 123 (see the excerpt below), and in the proof verification process in the file `verify.rs`, on line 195.

```
    // KZG challenge point
    let z = oracle!(
        &challenges[0],
        &proof.final_vkey.0,
        &proof.final_vkey.1,
        &proof.final_wkey.0,
        &proof.final_wkey.1
    );
```

This computation differs from the (internal) reference paper "Proposal: Practical Groth16 Aggregation" in two ways.

First, the paper states that the KZG challenge point is computed with all the challenges **x** (in bold, referring to a vector). Namely, towards the end of the `MIPP.Prove` procedure, at the top of p. 20, it states:

> Draw challenge $z = Hash_2(\mathbf{x}, v1, v2)$ (from all challenges **x** and $(v1, v2)$)

However, the implementation uses a single challenge, the first element of the `challenges` vector (as can be seen above), instead of using the vector containing all the challenges computed. While the element used in the point computation *does* incorporate all the previous challenges (since it is iteratively computed as a hash of the previous challenge), this does not strictly follow the specification.

Second, the computation of the KZG challenge point is performed in two instances in the paper; once during the TIPP proof and once during the MIPP proof. The inputs to the hash function differ for these two proofs. Consider the `TIPP.Prove` procedure, in which the challenge point is computed as follows (as can be seen on p. 22):

> Draw challenge $z = Hash_2(\mathbf{x}, v1, v2, w1, w2)$

This differs from the computation performed in `MIPP.Prove` shown above, with the additional inclusion of the commitment keys $w1, w2$. However, the implementation uses the same challenge computation for both proofs, namely the one performed in the `TIPP.Prove` procedure. It is unclear whether this has a security impact.

Finally, the NCC Group team noted that the paper uses a number of different notations for the computation of this challenge point. In addition to the two listed above, the following two notations appear in the paper.

On p. 20:

1. Reconstruct KZG challenge point: $z = H(x_{log(n)}, v1, v2)$

On p. 22:

1. Reconstruct KZG challenge point: $z = H(\{x_i\}_{i=0}^{log(n)}, v1, v2, w', w')$

Recommendation Ensure that the reference paper and the implementation match, and that notation is consistent throughout the paper.

If possible, consider providing an argument (in the implementation or the reference paper) supporting the use of a single KZG challenge point for both proofs.

Retest Results Pull Request 172 (which is also contained in Pull Request 179) combines a number of changes aiming to synchronize the implementation with the reference paper.

Instead of addressing the discrepancies listed in this finding, the reference paper[10] was updated in the following way.

1. Some arguments supporting the security of the combination of the TIPP and MIPP proofs and of their respective challenges were included.
2. The challenge KZG point is now computed using the last `challenge` only.

This addresses the concerns listed in this finding.

[10] https://eprint.iacr.org/2021/529

Finding	**Marginally Inconsistent/Outdated Toolchain and Dependencies**
Risk	**Informational** Impact: Undetermined, Exploitability: Undetermined
Identifier	NCC-E001405-001
Status	Not Fixed
Category	Patching
Component	all
Location	• `bellperson/rust-toolchain` • `bellperson/Cargo.toml` • `filecoin-ffi/rust/rust-toolchain` • `filecoin-ffi/rust/Cargo.toml` • `rust-fil-proofs/rust-toolchain` • `rust-fil-proofs/Cargo.toml`

Impact Attackers may attempt to identify and utilize publicly known vulnerabilities in outdated dependencies to exploit the functionality of the target code.

Description Incorporating outdated dependencies is one of the most common, serious and exploited application vulnerabilities. Inconsistent toolchains can also increase the difficulty of build and debug.

The `bellperson` repository contains a `rust-toolchain` file set to stable version 1.46.0, which is about 6-months out of date.[11] The `filecoin-ffi` repository contains a `rust-toolchain` file set to nightly-2020-10-5 of similar vintage. Note that the `rust-fil-proofs` repository contains a `rust-toolchain` file set to stable version 1.51.0 which is fully up to date. These are inconsistent and the former two are outdated.

Each repository includes a `Cargo.toml` file specifying dependencies. Some dependencies are marginally out of date, including:

- `bellperson`
  - `ff` specifies v0.2.0, while the latest `fff` version is v0.2.3
  - `rust-gpu-tools` specifies v0.2.0, while the latest version is v0.3.0
  - `rand_core` specifies v0.5, while the latest version is v0.6.2
  - `byteorder` specifies v1, while the latest version is v1.4.3
  - `rand` specifies v0.7, while the latest version is v0.8.3
  - `rayon` specifies v1.3.0, while the latest version is v1.5.0
  - `itertools` specifies v0.9.0, while the latest version is v0.10.0
- `filecoin-ffi`
  - `bls-signatures` specifies v0.8, while the latest version is v0.9.0
  - `ff` specifies v0.2.1, while the latest `fff` is v0.2.3
  - `rust-gpu-tools` specifies v0.2.0, while the latest version is v0.3.0
- `rust-fil-proofs`
  - No significant outdated dependencies

In addition, the `Cargo.toml` files make use of the `[patch.crates-io]` directive which further increases the challenge of version management, since branches rather than commits, versions or digests are specified.

Recommendation Update the `rust-toolchain` files to the (same) latest stable version recommended for production deployment, which is currently 1.51.0. If a repository requires the nightly channel, use a version of the same vintage. Update the `Cargo.toml` files to include the most recent

[11] https://github.com/rust-lang/rust/blob/master/RELEASES.md#version-1460-2020-08-27

versions of dependencies and minimize the complexity of `[patch.crates-io]` clauses.

Retest Results A review of Pull Request 179 does not indicate changes relevant to the `rust-toolchain` or `Cargo.toml` files nor the dependency versions. Note that this informational observation does not indicate an immediate security issue. The developers indicate this will be resolved through the regular update process. As such, this finding has been marked 'Not Fixed' at this time.

Finding Missing Domain Separation Parameter in Oracle Calls

Risk **Informational** Impact: Undetermined, Exploitability: Low

Identifier NCC-E001405-010

Status Fixed

Category Cryptography

Component bellperson

Location
- `bellperson/src/groth16/aggregate/prove.rs:124`
- `bellperson/src/groth16/aggregate/verify.rs:196`
- `bellperson/src/groth16/aggregate/prove.rs:233`
- `bellperson/src/groth16/aggregate/verify.rs:318`

Impact Calls to different random oracles may result in the same output, contradicting the random oracle model on which the security proofs are built.

Description When implementing cryptography protocols that were proven in the Random Oracle Model, it is common practice to instantiate random oracles using hash functions. Generally, when using a single hash function to model multiple random oracles, the hash functions are instantiated using different *domain separators*. This ensures that calls to different oracles with the same inputs will result in different outputs, so as not to invalidate the assumptions made in the security proof.

The (internal) reference paper "Proposal: Practical Groth16 Aggregation" describes two different hash functions used in the course of the proof aggregation and verification, $Hash_1$ and $Hash_2$, which are used to derive challenges for the proof commitments.

In the implementation, both hash functions are implemented by the `oracle` macro, in `bellperson/src/groth16/aggregate/macros.rs`, which hashes all its inputs using SHA256.

The NCC Group team noticed that no domain separator was passed in the oracle calls. As such, the two hash functions upon which the proofs are built are effectively the same, possibly invalidating the security proof. An example present in the `verify.rs` file is excerpted below.

```
let c = oracle!(
    &challenges.first().unwrap(),
    &fvkey.0,
    &fvkey.1,
    &fwkey.0,
    &fwkey.1
);
```

Recommendation Consider adding a domain separator tag to the oracle calls, different for the $Hash_1$ and $Hash_2$ calls.

Retest Results As part of Pull Request 172 (and also contained in Pull Request 179), two different domain separation tags (`"randomr"` and `"randomgipa"`, respectively) were added to the oracle calls, addressing this finding.

Appendix A: Finding Field Definitions

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

Risk Scale

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

Overall Risk

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

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

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

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

Low Implies a relatively minor threat to the application.

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

Impact

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

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

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

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

Exploitability

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

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

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

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

Category

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

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

This informational section highlights selected portions of the engagement methodology used, some of the priority concerns investigated, and observations that do not warrant security-related findings on their own. The primary strategy for this project relied heavily on manual source code inspection, supported by some execution of the included test cases and light fuzzing of some of the parsing functions. Priority was given to the correctness of cryptographic algorithms and implementation, the specific focus areas highlighted in the executive summary, data validation, control flow and general secure coding practices that could potentially impact legitimate operation.

Build Environment

The team evaluated the build environment (which resulted in finding NCC-E001405-001 on page 22) and ran the tool `cargo audit` on the three repositories under review. Two vulnerabilities were discovered: one crate used by the three repositories (`fil-ocl`) was found to be vulnerable, while another crate (`raw-cpuid`) used by `rust-fil-proofs` was also vulnerable. In addition, the `cargo audit` tool highlighted several other deprecated crates in the three repositories. The latter item shown below is addressed as part of the noted finding's recommendation.

```
ID:       RUSTSEC-2021-0011
Crate:    fil-ocl
Version:  0.19.6
Date:     2021-01-04
URL:      https://rustsec.org/advisories/RUSTSEC-2021-0011
Title:    EventList's From<EventList> conversions can double drop on panic.
Solution:  No safe upgrade is available!
Dependency tree:
fil-ocl 0.19.6
├── rust-gpu-tools 0.3.0
└── rust-gpu-tools 0.2.2

ID:       RUSTSEC-2021-0013
Crate:    raw-cpuid
Version:  8.1.2
Date:     2021-01-20
URL:      https://rustsec.org/advisories/RUSTSEC-2021-0013
Title:    Soundness issues in `raw-cpuid`
Solution:  upgrade to >= 9.0.0
Dependency tree:
raw-cpuid 8.1.2
└── fil-proofs-tooling 5.5.0
```

Fuzzing

The project supports two different backends for field, curve and pairing operations, namely the `blst` and `paired` crates. The integration with external dependencies introduces the potential for serialization/deserialization bugs. NCC Group built and ran custom fuzzers on the following blocks for both backends:

- `crate::bellperson::groth16::aggregate::AggregateProof::<Bls12>::read()`[12]
- `crate::bellperson::groth16::aggregate::GipaProof::<Bls12>::read()`[13]
- `crate::bellperson::groth16::aggregate::TippMippProof::<Bls12>::read()`[14]

One finding arose from this effort (see finding NCC-E001405-009 on page 13), while a second finding (see finding NCC-E001405-005 on page 7) was stimulated by the fuzzing development process.

Comparison between Reference Paper and Implementation

The two technical references were used as a support to assess the implementation. Some discrepancies with limited impact were identified and are discussed in more details in finding NCC-E001405-003 on page 5 and finding NCC-

[12] https://github.com/filecoin-project/bellperson/blob/d4120782d27e2971bd19ac5e12d206a7ccffbb7a/src/groth16/aggregate/proof.rs#L82
[13] https://github.com/filecoin-project/bellperson/blob/d4120782d27e2971bd19ac5e12d206a7ccffbb7a/src/groth16/aggregate/proof.rs#L233
[14] https://github.com/filecoin-project/bellperson/blob/d4120782d27e2971bd19ac5e12d206a7ccffbb7a/src/groth16/aggregate/proof.rs#L355

E001405-011 on page 20.

The team paid particular attention to the good generation and use of randomness, which is especially important to ensure the security of the aggregate proof verification. Indeed, the aggregate proof verification process checks a single randomized equation, which, if it were not random, might be exploited by an attacker.

The generation of a Structured Reference String (SRS) is typically a very sensitive operation; in other protocols such as the ZCash powers of tau ceremony, extensive care[15] was taken to erase the toxic waste. Since the proof aggregation and verification make use of two existing SRSs, the need for memory zeroization in that regard is not necessary.

Input Validation

Input validation was inspected manually, by debugger-introduced variations, and by modified test cases where possible. Initial focus was placed on field range and curve subgroup validation, such as the deserialization and appropriate range check of `Fq`, or the correct subgroup check on `G1Projective` during deserialization.

The team noted a number of instances that could benefit from better input validation, as evidenced by findings such as finding NCC-E001405-006 on page 9 and finding NCC-E001405-004 on page 16. In addition, in the spirit of defense in depth, it might be beneficial to include such checks throughout the code, even if the functions do not seem susceptible to be used externally. For example, in `bellperson/src/groth16/aggregate/mod.rs`, in the `compress()` function (provided below for reference), if the value of `split` were larger than the length of the vector, the assertion in `split_at_mut()` would fail.

```
fn compress<C: CurveAffine>(vec: &mut Vec<C>, split: usize, scaler: &C::Scalar) {
    let (left, right) = vec.split_at_mut(split);
    // ...
```

```
pub fn split_at_mut(&mut self, mid: usize) -> (&mut [T], &mut [T]) {
    assert!(mid <= self.len());
    // ...
```

A second example from the `rust-fil-proofs` repository source file `seal.rs:801` is excerpted below. The function `verify_aggregate_seal_commit_proofs()` performs an unchecked division via the remainder operator `%` on an unvalidated caller-provided function parameter. If `aggregated_proofs_len` is zero, this will result in a `panic`.

```
ensure!(
    commit_inputs.len() % aggregated_proofs_len == 0,
    "invalid number of inputs provided"
);
```

Rust Programming Practices

The team noted a few instances of potentially unsafe Rust programming practices, for example in the use of `unwrap()` (such as in finding NCC-E001405-002 on page 14). Generally speaking, explicit error handling should be preferred instead of calling functions that might result in panics, such as `unwrap()` or `expect()`. The Secure Rust Guidelines provide some helpful pointers to that effect.

As an example, the function `aggregate_seal_commit_proofs()` contains two instances of `.expect()` shown below, which will cause a panic if triggered. The same function (and others in this file) use the preferred `ensure!()` macro in other places to pass error messages rather than panic.

```
let mut proofs: Vec<_> = commit_outputs
    .iter()
    .fold(Vec::new(), |mut acc, commit_output| {
```

[15] https://z.cash/technology/paramgen/

```
        acc.extend(
            MultiProof::new_from_reader(
                Some(partitions),
                &commit_output.proof[..],
                &verifying_key,
            )
            .expect("failed to construct multi proof from bytes")
            .circuit_proofs,
        );

        acc
    });
```

```
proofs.push(
    proofs
        .last()
        .expect("failed to access last proof for duplication")
        .clone(),
);
```

Adapting the above logic to utilize the preferred `ensure!()` macro will increase the ability for callers to gracefully handle errors.

Separately, another particular item of note is the use of the `zip()` operator to iterate through multiple vectors at the same time. Together with the lack of confirmation of equal-length lists prior to `zip`-ing them, this has shown to be a potential vector of issues (see finding NCC-E001405-006 on page 9 and finding NCC-E001405-008 on page 18) that could be avoided by adding an inexpensive check that will prevent an "orphan" from being ignored (and potentially enabling an escape). There are many instances where this operator is used. For example (as also noted in finding NCC-E001405-005 on page 7), the following computation in `bellperson/src/groth16/aggregate/inner_product.rs` might return incorrect results in non-debug builds.

```
pub fn pairing_miller_affine<E: Engine>(left: &[E::G1Affine], right: &[E::G2Affine]) -> E::Fqk {
    debug_assert_eq!(left.len(), right.len());
    let pairs = left
        .par_iter()
        .map(|e| e.prepare())
        .zip(right.par_iter().map(|e| e.prepare()))
        .collect::<Vec<_>>();
    let pairs_ref: Vec<_> = pairs.iter().map(|(a, b)| (a, b)).collect();

    E::miller_loop(pairs_ref.iter())
}
```

Optimization Potential

Generally, the cost of performing a field inversion is significantly larger than the cost of computing a field multiplication. In `bellperson/src/groth16/aggregate/prove.rs`, the inverses of the powers of r are computed, as can be seen in the code excerpt below.

```
// r, r^2, r^3, r^4 ...
let r_vec = structured_scalar_power(proofs.len(), &r);
// r^-1, r^-2, r^-3
let r_inv = r_vec
    .par_iter()
    .map(|ri| ri.inverse().unwrap())
    .collect::<Vec<_>>();
```

Thus, for an array of length n, the code above computes n modular inversions. A possible optimization to this piece of code might be to compute r^{-n} (only one inversion) and then successively multiply it by r, thus obtaining $r^{-n+1}, \ldots, r^{-2}, r^{-1}$.

Additionally, there are instances of iteratively building long vectors where the length is known in advance. Declaring new vectors with `Vec::with_capacity`[16] may significantly reduce the otherwise repeated memory allocation overhead.

[16] https://doc.rust-lang.org/std/vec/struct.Vec.html#capacity-and-reallocation