Protocol Labs

ZK-SNARK proofs audit

26th April 2020

Contents

1

Introduction

Protocol Labs hired us to review the security of their Filecoin storage proofs, which essentially consist of zk-SNARK proofs of tree-based proofs-of-replication. Although the core zk-SNARK logic is well understood and implemented using the established `bellman` implementation (via the `bellperson` fork), the SNARK circuits creation is novel, and a critical component for Filecoin's security.

This report describes the work performed, including our methodology, a general description of the code covered and its security components, a minor specific observations regarding implementation quality.

We would like to thank Protocol Labs for trusting us and for their availability to answer our questions in a timely manner. We greatly enjoyed working on this project, which is among the most interesting auditing works we have done.

The work was performed by Dr. Jean-Philippe Aumasson and Antony Vennard, and took the equivalent of 11 person-days.

1.1 Audit summary

We summarize this security assessment project as follows:

- It took us some time (a few days) to understand what was exactly being done and how it was done. Even for seasoned experts familiar with SNARKS, it **proved challenging** to get a full understanding of the construction, mainly due to the relative complexity of the structure of proofs-of-replication, an unusual type of cryptography mechanism.
- We found that the code base somewhat **lacked explanations and comments**, and that the documentation provided was a reference for developers already familiar with the logic implemented. A more gentle introduction might be beneficial to future new contributors and reviewers.
- The **highly modular structure** of the code, with distinct structures and files modules for each proof type and functionality, as well as the clear separation between vanilla proofs and circuits, makes the workflow easier to follow, and gives a relatively clear

picture of what role various components play and what are the potential security issues.

- The **zk-SNARK creation workflow**, and in particular the circuit synthesis, is clearly separated from the creation of the vanilla proofs (of replication and/or spacetime), which we believe is a good design choice that makes the code more adaptable to future updates, and also easier to audit. The fact that circuit synthesis logic is split across different places in the code is not ideal from an audit perspective, but makes logical sense in terms of software design, so we recommend to stick to the current structure.
- The circuit creation relies as much as possible on **established components** (mainly, the bellman/bellperson crate and its "Groth16" logic), where the only novel component is the Poseidon hash and its implementation, which we believe do not represent a high security risk. The code is written in safe Rust, and enforces the validity, correct encoding, and acceptable size of objects it is processing (in particular, potentially untrusted inputs).
- Our work involved documents review, code review, writing of custom test cases, step-by-step execution using a debugger. We notably evaluated the code behavior against the expected behavior, as documented, and **did not find any significant discrepancy or inconsistency**.
- The circuit definition and synthesis appeared to **fully match the expected functionality**, and did not reveal any behavior that would compromise the zk-SNARKs' security (be it in terms of completeness, soundness, or information leakage). Although we did not review bellperson's logic, we sometimes referred to its code to verify that it behaved as expected. Overall, we **have not found any issue** we consider as having a security impact. The only observations we report are quality improvement suggestions and defense-in-depth controls.
- The testing of the circuits and evaluation under adversarial/invalid input was made more challenging by the relatively **low number of unit tests**, and the complexity and running time of the ones available. Although it does not seem straightforward to create tests that are both much simpler/faster yet meaningful, such tests would be greatly beneficial for more advanced and automated testing. For example, some kind of fuzzing would probably be valuable.

1.2 Scope

The audit concerns components in the filecoin-project/rust-fil-proofs (version v3.0.0), where the main components relevant for this audit are:

- The storage-proof/core components, including
  - The generic definition of `ProofScheme`
  - The compound proof logic defined by the CompoundProof trait, whose proving mechanisms first builds "vanilla" proofs (i.e., non-SNARK proofs) for a series of partitions, before turning said proofs into an actual NIZK proof after circuit synthesis.

- The storage-proofs/porep component, specific to proofs-of-replication, whose code of interest for the audit is mainly in the stacked/circuit subdirectory. Also of interest for the logic's understanding is the stacked/vanilla code.
- The storage-proofs/post component, specific to proofs-of-spacetime, built on top of proofs-of-replication, and whose code of interest for the audit is mainly in fallback/circuit.rs, and where the vanilla proof mechanism is in fallback/vanilla.rs.

Other important components are the bellman zk-SNARK logic (for creating the NIZK proofs), and the neptune implementation of the Poseidon circuit-friendly hash function. These are *not* in the scope of the audit, and were just partially reviewed when needed, for the sake of the system's understanding.

1.3 Security goals

In our review, we evaluated the proofs' and their circuits implementation against the following security properties:

- *Correctness*: The code should correctly translates the proof logic into the structure processed by Groth proofs, and this logic should be consistent across different proof instances or executions of the code.
- *Completeness and soundness*: A proof (PoRep or PoST) that is incorrectly encoded or invalid should not lead to a zk-SNARK proof that would be verifiable as valid. Likewise, valid PoRep or PoST should yield a valid zk-SNARK proof.
- *Zero-knowledge*: The proof generated should not leak any data that in principle should not be leaked. To a lesser extent, the creation of the proof should not expose information more than needed (note that side channel evaluation was out of scope).
- *Software security*: The implementation cannot be abused by exploiting logic errors or software bugs. Although the audit is not a pure code review, code safety is one relevant aspect of our assessment, for it can jeopardize the above security notions.

The section below explains in details the actions performed to assess these security properties.

2

Methodology

Our main sources of information were, besides the code repositories, documentation provided by Protocol Labs: comprehensive documentation of both the specification and of the lower level encodings and structures. We notably compared this documentation provided with the code, in terms of algorithmic logic, values encoding, bit ordering, etc., and looked for inconsistencies between the two.

When needed for the understanding of the code's logic, we also referred to the Filecoin paper, the proof of replication paper, and the Groth16 paper.

To test the implementation, we relied on built-in unit tests that we modified to test specific (such as porep's `stacked_test_compound()`), and also relied on `fil-proof-tooling`, as well as custom test suites that we wrote in Rust.

In our review of the circuit creation code, we reviewed the proper allocation of inputs, parameters, adequate use of `bellperson` logic, and content of the dedicated gadgets used. We also looked for missing sanity checks, possible acceptance of invalid or incorrect proofs in the circuitization process, correct generation of inputs, consistent circuit synthesis across different runs or instances, redundant operations, suboptimal ordering of operations, and so on. More generally, we looked for any behavior that could compromise the proof's security, in terms of completeness, soundness, or information leak.

When auditing the circuit creation code for pure code safety, we specifically looked for

- The handling of edge cases that could lead to unsafe behavior (for example, potential ambiguous encodings of data)
- Correct and safe usage of other components (mainly bellperson).
- Correct and safe usage of cryptographic APIs (hash functions Poseidon and SHA-256, PRNG)
- Software security bugs, such as:
  - Potential memory corruption issues (in `unsafe` blocks).
  - Memory leaks and unsafe use of dynamic allocators.
  - Arithmetic bugs, such as integer overflow and division by zero.
  - Unhandled or unsafely handled errors.

We did this via a combination of static analysis (reading code, comparing to specifications/expected outcomes) and dynamic analysis (debugging running test harnesses to understand how the code works and what happens if parameters are altered).

As in all Rust audits we do, we ran linters and analyzers to get an idea of the general code hygiene and safety. We notably used the following Rust utilities via `cargo`:

- `audit`, identifying dependencies with known vulnerabilities identified by the RustSec project.
- `clippy`, standard Rust linter, suggesting quality improvements and identifying risky coding patterns.
- `geiger`, reporting statistics about `unsafe` code blocks.
- `outdated`, to identify dependencies used in a version that is not the latest one.
- `asm`, to example how code compiles to machine code.

Except for one minor issue related to deprecated third-party crates, these tools did not reveal any significant issue.

3

Code functionality

We briefly describe the components involved and the general proof workflow. Details can be found in the related papers and documentation.

3.1 Building blocks

The core cryptographic building blocks of the filecoin proofs are:

- zk-SNARKs, the non-interactive zero-knowledge proof technique built on Groth's technique, and implemented by leveraging Zcash's `bellman` crate.
- Hash functions: the circuit-friend Poseidon hash (whose implementation in the `neptune` crate was audited), and SHA-256, as well as "SHA-254", a version of SHA-256 whose output is stripped of its two most significant bits in order to hash to a 254-bit field element.
- The BLS12-381 curve, an established pairing-friendly curve, which targets 128-bit security.
- Merkle tree, or hash trees Trees involved in proofs of replication are binary or of arity 8 ("octotrees"), and use Poseidon or SHA-256 as a hash. For example, column commitments
- Proof-of-membership, a.k.a. Merkle proofs or authentication paths, which consist in showing the set of nodes that "connect" a tree's leaf or node to its root, by hashing one's way up to the tree's too.
- Depth-robust graphs (DRG), a key component of the proof-of-replication encoding scheme, which ensures (to some extent) that replicas remain stored and are not recomputed between different proofs-of-replication. DRGs are graphs

3.2 Schemes implemented

- A proof-of-replication (PoRep) extends the basic concept of proof-of-retrievability by proving that multiple copies (replicas) of the data are stored. This leverages DRGs to "seal" different instances of the data, and showing compact proofs for such DRGs.

- A proof-of-spacetime (PoSt) extends PoRep by proving that multiple copies of the data are stored for a given period of time. A PoSt involves a series of PoReps.

(Here, “proving” should of course be understood as “proving a certain security bound that depends on certain computational assumptions and on several parameters”.)

3.3 BLS12-381 background

This section will be very brief, as the literature on BLS12 curves is fairly widely established and understood. BLS12-381 allows us to target a 128-bit security level type-3 pairing, usually implemented as an optimal ate pairing, that transforms between two groups to a third:

$$e : \mathbb{G}_1 \times \mathbb{G}_2 \mapsto \mathbb{G}_T$$

In usual implementations, the first group is formed from the curve $E(\mathbb{F}_p)$ and the second group from the sextic twist of the 12th degree extension field, which is $E(\mathbb{F}_{p^2})$.

A third field exists, $\mathbb{F}_r$, where r is the size of the cyclic subgroup used for cryptography over the curves. Elements of this field therefore describe the order of any point in the scheme.

3.4 Groth16 background

The objective of the Groth16 work is to provide a proof via a non-interactive protocol in zero-knowledge. Ideally such a proof is succinct, hence the Succinct Non-Interactive Argument of Knowledge (SNARK) acronym.

The Groth16 technique uses a generalized arithmetic circuit to build a binary relation R, comprised of statement wires and witness wires. The witness wires satisfy the arithmetic circuit and make the overall circuit consistent (or not).

The generalized arithmetic circuit is expressed as:

$$\sum a_i u_{i,q} \cdot \sum a_i v_{i,q} = \sum a_i w_{i,q}$$

Where $a_0 = 1$ and $a_1, \ldots, a_m \in K$ are variables and u, v, w are constants also in K specifying the appropriate equation.

This expression is reformulated by taking arbitrary but distinct $r_1, \ldots, r_n \in K$, defining

$$t(x) = \prod_{q=1}^{n} (x - r_q)$$

and then set $u_i(x)$ to be a polynomial degree $n - 1$ where:

$$u_i(r_q) = u_{i,q}$$

that is, the polynomial chosen when evaluated at the respective r_q is equal to the constant chosen above. Repeat this for v, w.

The above generalized arithmetic circuit can then be written as:

$$\sum a_i u_i(X) \cdot \sum a_i v_i(X) = \sum a_i w_i(X) \mod t(X)$$

The overall binary relation becomes:

$$R = (K, \mathsf{aux}, l, \{u_i(X), v_i(X), w_i(X)\}_{i=0}^{m}, t(X))$$

Here the parameter l is chosen such that $1 \leq l \leq m$. Those terms in $\{u_i(X), v_i(X), w_i(X)\}_{i=0}^{l}$ are statements in the way explained above, and those terms $\{u_i(X), v_i(X), w_i(X)\}_{i=l+1}^{m}$ are witness circuits. K is an appropriately large finite field, aux is some auxiliary information and $t(X)$ a polynomial described above.

Thus l delimits which of the m available wires are statement wires and which are witness wires.

The system proposed by Groth and used here is concretely a pairing based variant of this generalized description.

In the type-3 pairing case frequently used by BLS12-381, $\mathbb{G}_1 \neq \mathbb{G}_2$ in the pairing groups and there is no efficiently computable isomorphism between them - these are usually implemented as optimal ate pairings.

The pairing-based NIZK is specified by:

$$R = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, g, h, l, \{u_i(X), v_i(X), w_i(X)\}_{i=0}^{m}, t(X))$$

Then:

1. Setup: pick $\alpha, \beta, \gamma, \delta, x \in \mathbb{Z}_p^*$.

   Define:

   $$\tau = (\alpha, \beta, \gamma, \delta, x)$$

   Then let:

   $$\sigma_1 = \left(\begin{array}{c} \alpha, \beta, \gamma, \delta, \{\dfrac{\beta u_i(x) + \alpha v_i(x) + w_i(x)}{\gamma}\}_{i=0}^{l} \\ \left\{\dfrac{\beta u_i(x) + \alpha v_i(x) + w_i(x)}{\gamma}\right\}_{i=l+1}^{m}, \left\{\dfrac{x^i t(x)}{\delta}\right\}_{i=0}^{n-1} \end{array} \right)$$

   and let:

   $$\sigma_2 = (\beta, \gamma\delta, \{x^i\}_{i=0}^{n-1})$$

   Then return $\sigma = (\sigma_1, \sigma_2)$

   These expressions are complex, but are simply parameter sets expressing appropriately manipulated equations that are circuits, or quadratic arithmetic programs, with parameters in appropriate domains for use in pairings.

2. Prove: pick $r, s \in \mathbb{Z}_p^*$. Compute:

$$A = \alpha + \sum_{i=0}^{m} a_i u_i(x) + r\delta$$
$$B = \beta + \sum_{i=^0}^{m} a_i v_i(x) + s\delta$$
$$C = \frac{\sum_{i=l+1}^{m} a_i(\beta u_i(x) + \alpha v_i(x) + w_i(x)) + h(x)t(x)}{\delta} + As + Br - rs\delta$$

   The proof statement is then: $\pi = (A \cdot G_1, B \cdot G_1, B \cdot G_2)$

3. Verify:

$$e(A, B) == e(\alpha, \beta) + e\left(\sum_{i=0}^{l} a_i \left[\frac{\beta u_i(x) + \alpha v_i(x) + w_i(x)}{\gamma}\right], \gamma\right) + e(C, \delta)$$

   where e is the bilinear pairing.

In the Groth paper, $[x]_1$ is used to mean $x \in \mathbb{G}_1$, that is, the bracket subscript indicates the group in question. This notation has not been used here.

3.5 Rank-1 constraint systems

This concept originates in linear algebra but is relevant in the zkSNARK literature. The Groth16 scheme enables one to implement such a constraint system.

The rank of a matrix A is the number of linearly independent columns in the matrix. A circuit from groth16 can be reduced to a rank-1 constraint system and so the language of constraint systems is used frequently throughout crates used by this project.

3.6 Implementation approach

The implementation approach is partially documented in https://hackmd.io/@jake/Bko2WrFhL and https://hackmd.io/@jake/rym1Ggj2L, regarding the algorithms implemented, primitives and encoding used. Below we briefly describe our understanding of the general workflow of a proof's generation logic, as context for our review and further comments.

The main steps involved in a creation of a (compound) proof, as well as the software components involved, are the following (we omit details of the PoRep construction) after an initial setup:

The first step is a creation of (vanilla) a batch of proofs of replication, using the `ProofScheme`'s `prove_all_partitions()` method. The actual `prove_all_partitions()` is therefore that of a vanilla proof, ultimately relying on the logic of `prove_layer()` in porep/stacked/vanilla/proof.rs.

These vanilla proofs are not NIZK proofs yet, and will be used as input for creating circuit proofs. This is done via (as defined in compound_proofs.rs):

```
fn circuit_proofs(
    pub_in: &S::PublicInputs,
    vanilla_proof: Vec<S::Proof>,
    pub_params: &S::PublicParams,
    groth_params: &groth16::MappedParameters<Bls12>,
    priority: bool,
)
```

which takes a vector of vanilla proofs. Each proof in this vector is then circuitized via:

```
 Self::circuit(
     &pub_in,
     C::ComponentPrivateInputs::default(),
     &vanilla_proof,
     &pub_params,
     Some(k),
 )
```

The actual `circuit()` being called is from porep/stacked/circuit/proof.rs, which performs some sanity checks and returns a `StackedCircuit` object.

The actual circuit synthesis happens in groth16's `create_proof_batch_priority()`, called via various wrappers:

```
let mut provers = circuits
    .into_par_iter()
    .map(|circuit| -> Result<_, SynthesisError> {
        let mut prover = ProvingAssignment {
            a_aux_density: DensityTracker::new(),
            b_input_density: DensityTracker::new(),
            b_aux_density: DensityTracker::new(),
            a: vec![],
            b: vec![],
            c: vec![],
            input_assignment: vec![],
            aux_assignment: vec![],
        };

        prover.alloc_input(|| "", || Ok(E::Fr::one()))?;

        circuit.synthesize(&mut prover)?;
```

The `synthesize()` of the stacked circuit object is then called, for example the PoRep's: this takes public parameters, a list of proofs, a replica ID, and commitments to the tree's roots (or hash thereof):

```
fn synthesize<CS: ConstraintSystem<Bls12>>(self, cs: &mut CS) -> Result<(), SynthesisError> {
    let StackedCircuit {
        public_params,
```

```
        proofs,
        replica_id,
        comm_r,
        comm_d,
        comm_r_last,
        comm_c,
        ..
    } = self;
```

This then allocates commitments as numbers, defines the public input, reverses the order of the ID's bit, and synthesizes each of the proofs:

```
        for (i, proof) in proofs.into_iter().enumerate() {
            proof.synthesize(
                &mut cs.namespace(|| format!("challenge_{}", i)),
                public_params.layer_challenges.layers(),
                &comm_d_num,
                &comm_c_num,
                &comm_r_last_num,
                &replica_id_bits,
            )?;
        }
```

Non-vanilla proofs are verified using `groth16::verify_proof` according to the logic in the groth16 section above. Proof verification can be batched using `groth16::verify_proofs_batched`. The `CompoundProof` trait in storage core offers two functions, `verify` and `verify_batch`. We include only the signature of `verify` here:

```
fn verify<'b>(
        public_params: &PublicParams<'a, S>,
        public_inputs: &S::PublicInputs,
        multi_proof: &MultiProof<'b>,
        requirements: &S::Requirements,
    ) -> Result<bool> {
```

Here a `multi_proof` is exactly a vector of groth16 compatible proofs:

```
pub struct MultiProof<'a> {
    pub circuit_proofs: Vec<groth16::Proof<Bls12>>,
    pub verifying_key: &'a groth16::VerifyingKey<Bls12>,
}
```

In this case, `StackedCompound`, which forms the target scope for our review, implements `CompoundProof`, providing the `prove` and `verify_*` methods necessary to create and prove SNARG proofs.

Our review then focuses on what happens from this point. During this review we looked primarily at the circuit being synthesized and its appropriate use in Groth proofs. Full circuit synthesis takes place in `porep/src/stacked/circuit/params.rs`.

Here, as described in the specification a number of proof of replication circuits are synthesized and added as “inclusions” in the constraint system. The constraint system object is reasonably complex: it supports multiple namespaces under the root each with their own unique namespace, as implemented by the namespace struct:

```
/// This is a ”namespaced” constraint system which borrows a constraint system (pushing
/// a namespace context) and, when dropped, pops out of the namespace context.
pub struct Namespace<’a, E: ScalarEngine, CS: ConstraintSystem<E>>(&’a mut CS, PhantomData<E>);
```

This is a newtype declaration, so the Rust syntax appears unusual; the constraint system is then referenced as `self.0` inside this struct. The constraint system itself is implemented as a trait, for which a test harness is provided. Its code comment describes its role:

```
/// Represents a constraint system which can have new variables
/// allocated and constrains between them formed.
pub trait ConstraintSystem<E: ScalarEngine>: Sized {
    /// Represents the type of the ”root” of this constraint system
    /// so that nested namespaces can minimize indirection.
    type Root: ConstraintSystem<E>;
```

The Constraint system is built of `LinearCombination` objects:

```
/// This represents a linear combination of some variables, with coefficients
/// in the scalar field of a pairing-friendly elliptic curve group.
#[derive(Clone)]
pub struct LinearCombination<E: ScalarEngine>(HashMap<Variable, E::Fr>);
```

This may not seem to match the definitions of the Groth16 but in fact does if we consider that any elliptic curve point may be expressed as the generator multiplied by some value k in a cyclic group. Thus it is possible to evaluate the pairings demanded by the groth16 proof.

The multiple namespaces in the constraint system allow different commitments to be made. We were not asked to analyse the larger scheme in this review, so we have not included report findings or a description of said scheme here. Nevertheless, we have checked that these variables are of the desired type, how they are instantiated and how well the system tolerates possible manipulations.

Indeed this can be seen in the code, for example:

```
Thread 1 ”filecoinrunner” hit Breakpoint 4, paired::Engine::pairing (p=..., q=...)
    at paired-0.19.1/src/lib.rs:97
(gdb) bt
#0  paired::Engine::pairing (p=..., q=...) at paired-0.19.1/src/lib.rs:97

#1  0x0000555555971505 in bellperson::groth16::verifier::prepare_batch_verifying_key
    (vk=0x7fffffffc9c0) at bellperson-0.8.0/src/groth16/verifier.rs:32

#2  0x00005555557c45e3 in storage_proofs_core::compound_proof::CompoundProof::verify
```

```
    (public_params=0x7fffffffaaa8, public_inputs=0x7fffffffb340,
    multi_proof=0x7fffffffd358, requirements=0x55555653cc00)
    at rust-fil-proofs/storage-proofs/core/src/compound_proof.rs:118

#3  0x0000555555c60b16 in filecoinrunner::filecoinutils::run_compound_proof ()
    at src/filecoinutils.rs:377

#4  0x00005555558a5ad6 in filecoinrunner::main () at src/main.rs:46
```

In public parameters and inputs supplied to proof and verify functions, Poseidon "domains" are supplied. The Poseidon hash function was designed for more efficient hashing in SNARKs and SNARGS, where the cost of SHA-256 circuits is computationally very expensive and a quicker, yet secure, alternative is desired. This is used in the compound proof as desired.

From our verification and analysis of the inputs, including attempts to induce logic errors with a debugger and a custom running harness and having compared the scheme to its specification, we have we believe demonstrated to our own satisfaction that the code implements the desired scheme correctly.

4

Security issues

In the course of our analysis we have not found any issues we consider as having a security impact. The code is written in safe Rust, and uses established and well implemented dependencies for its circuit construction, evaluation and for Groth proofs.

As always, security issues cannot be entirely ruled out.

5

Observations

5.1 Suggested additional pre-conditions checks

In porep/stacked/circuit/params.rs, `synthesize()` may perform early additional sanity checks such as checking the length of `comm_d_path` (although an invalid path will prevent `enforce_inclusion()` from succeeding). The number of layers `layers` could be checked to be the number of parent columns (being used in `parent_col.get_value(layer)`), otherwise the `assert!` in `get_value()` will fail. Likewise, parents proofs argument in this function might gain to be checked for consistency and non-emptiness.

In porep/stacked/circuit/create_label.rs, `create_label_circuit()` accepts `replica_id` that is a very short or even empty slice, a minimal length should thus be enforced.

5.1.1 Resolution

This issue was resolved by the customer in this pull request. Checks now exist for empty input values:

```
assert!(!drg_parents_proofs.is_empty());
assert!(!exp_parents_proofs.is_empty());
```

as suggested. In addition, as suggested, layers and parent column equality are also checked:

```
assert_eq!(layers, parent_col.len());
```

The remaining issues suggested here are also resolved as part of these commits.

5.2 Unhandled overflow

The (unlikely) case of an integer overflow is not handled in the multiplications in `satisfies_requirements()`:

```
fn satisfies_requirements(
    public_params: &Self::PublicParams,
    requirements: &Self::Requirements,
    partitions: usize,
) -> bool {
    partitions * public_params.sector_count * public_params.challenge_count
        >= requirements.minimum_challenge_count
}

fn satisfies_requirements(
    public_params: &PublicParams<Tree>,
    requirements: &ChallengeRequirements,
    partitions: usize,
) -> bool {
    let partition_challenges = public_params.layer_challenges.challenges_count_all();

    partition_challenges * partitions >= requirements.minimum_challenges
}
```

5.2.1 Resolution

This issue was resolved by the customer in this pull request. `satisfies_requirements` now includes an overflow check in the form of Rust/LLVM's `checked_mul` primitive:

```
assert_eq!(
    partition_challenges.checked_mul(partitions),
    Some(partition_challenges * partitions)
);
```

5.3 Deprecated dependencies

Two dependencies are reported as deprecated by `cargo audit`, and could be replaced with new crates:

- `tempdir`, used in `neptune` and `rust-fil-proofs`, to be replaced with `tempfile`
- `net2`, used in `rust-fil-proofs`, to be replaced with `socket2`

5.3.1 Resolution

This issue was resolved by the customer in this pull request. `tempdir` has been replaced by `tempfile`.

`socket2` is used by an upstream dependency and will need to be resolved by the relevant upstream package.

5.4 Naming improvement

In compound_proofs.rs' `circuit_proofs()`, the `vanilla_proof: Vec<S::Proof>` might be renamed to `vanilla_proofs` for clarity.

5.4.1 Resolution

This issue was resolved by the customer in this pull request. The name is changed as suggested.