Security assessment

Lurk – Protocol Labs

inference

20230411 – FINAL

Contents

1 Introduction

This report presents the results of our collaboration with Protocol Labs' Lurk team, to improve the security assurance of the novel Lurk system, a Lisp dialect used to generate succinct zero-knowledge proofs of arbitrary deterministic programs. Lurk aims to offer program execution proved in zero-knowledge, using different proving back-ends for the recursive zkSNARK computation.

Our main goal was to assess the security of Lurk in terms of soundness, correctness, and zero-knowledge, by reviewing in detail:

- The end-to-end processing of Lurk programs from Lurk code to a proof, and in particular
- The circuits used to perform that processing, including the main "circuit frame" circuit as well as related gadgets.

We also looked for security defects including:

- Choice of cryptographic primitives and protocols (including the proof systems Groth16 and Nova, and their internals).
- Correctness and safety of cryptographic mechanisms built in Lurk (hashing, commitments, functional commitments).
- General correctness of the code against specification and standards implemented.
- Software security bugs and safety issues (including side channels), including deserialization bugs.
- "Supply-chain" risks related to the dependencies used.
- Randomness issues (generation, sampling, entropy, etc.).

Our methodology included manual code review and dynamic analysis based on modified versions of the test suite.

We would like to thank the Lurk team for trusting us, for their timely responses to our questions, and for joining us in biweekly meetings to provide feedback on our findings and explain design choices.

1.1 Scope

The code reviewed is mainly in the Lurk Rust implementation repository, lurk-lab/lurk-rs. In this repository, the most critical part of the implementation is the circuit and associated gadgets in src/circuits. We also reviewed other components of the Lurk implementation, such as the ("vanilla") reduction logic, code parser, arithmetic of field elements and 64-bit unsigned integers, among others.

For security-critical operations, Lurk relies on dependencies from other repositories, include some of other organizations. These include:

- bellperson: https://github.com/filecoin-project/bellperson (Groth16 and R1CS/circuit creation)
- neptune: https://github.com/filecoin-project/neptune (Poseidon-based hashing)
- nova: https://github.com/Microsoft/Nova (Nova recursive proof system)
- pasta curves: https://github.com/zcash/pasta_curves (elliptic curves)
- pasta msm: https://github.com/supranational/pasta-msm (multi-scalar multiplication)
- multihash: https://github.com/multiformats/rust-multihash (generic hashing)

The description of findings below include links to the version/commit concerned, and/or to the GitHub issue or PR.

In a second phase of the audit, we specifically reviewed the following PRs, some of which modified the code previously reviewed in order to reduce the number of constraints in the circuit:

- https://github.com/lurk-lab/lurk-rs/pull/245: Implement more syntax errors.
- https://github.com/lurk-lab/lurk-rs/pull/253: Make calling functions with malformed bodies an error.
- https://github.com/lurk-lab/lurk-rs/pull/254: Fix binding syntax.
- https://github.com/lurk-lab/lurk-rs/pull/258: Optimize `def_head_sym`.
- https://github.com/lurk-lab/lurk-rs/pull/259: Prefer constants to allocations where possible.
- https://github.com/lurk-lab/lurk-rs/pull/263: Optimize variadic and/or.
- https://github.com/lurk-lab/lurk-rs/pull/309/: Slight store performance improvements (low-hanging fruits).
- https://github.com/lurk-lab/lurk-rs/pull/309/: Refactor Scalar(Cont)Ptr.

1.2 Documentation

Aside from the code, the following pages and documents proved useful:

- The post A Programmer's Introduction to Lurk.
- Lurk Language Specification in lurk-lab/lurk/spec/v0-1.md.
- Language specs + circuit overview in lurk-lab/lurk-rs/spec/main.pdf.
- Eval specs in lurk-lab/lurk-rs/spec/eval.md.
- Reduction notes in lurk-lab/lurk-rs/spec/reduction-notes.md.
- Filecoin's post Introducing Lurk: A programming language for recursive zk-SNARKs.
- Protocol Labs' post SnarkPack: How to aggregate SNARKs efficiently.
- The talk A Zero-Knowledge circuit for the Lurk language by Eduardo from the Lurk team.
- NCC's public audit report of Protocol Labs' Groth16 proof aggregation.

- The Nova paper.

1.3 Activities

Our strategy over the 3-month security review can be broadly split up into 3 phases, each lasting approximately one month. (Note that we were not working full-time on the project, but approximately 0.5 FTE – a month here is thus "wall time" rather than "CPU time").

The first month was spent getting familiarized with the code base and its accompanying specification documents, and understanding the fundamental concepts underpinning the Lurk project. These include the language itself and its connection to Lisp, but also the theoretical constructions like Nova, SnarkPack and incrementally verifiable computation.

The next phase was spent reviewing the primitives, R1CS gadgets and macros and understanding how they are used in the full reduction circuit. This phase involved manual checking of all custom R1CS constraints, as well as modifications to the test cases to trigger more edge cases. In parallel, we also started writing custom Lurk programs in order to assess the capabilities of the language, and discover potential errors in the actual reduction specification.

The remainder of our time was spent analyzing the actual reduction circuit, by carefully comparing each step against the Rust specification side-by-side. During this whole time, we would discuss our findings with the team as we discovered them, either through Telegram or as part of our bi-weekly calls. We also reviewed pull requests to ensure fixes were applied correctly, as well as new code that modified parts of the code base we had already reviewed.

2 Lurk's security concepts

After reading through enough documentation and code to get a reasonable grasp of Lurk's internals, we started the security assessment project by writing down an overview of the security properties to achieve across the Lurk workflow, which goes as follows (from a very high level):

1. Creation of a program in the Lurk language.
2. Reduction step breaking down the program in small components.
3. R1CS witness generation for each iteration of the reduction circuit.
4. zkSNARK proving & aggregation of each reduction using Groth16+SnarkPack or Nova+Spartan.

Bugs in steps 2, 3, and 4 could potentially compromise completeness, soundness, and zero-knowledge of the Lurk system. Below we'll examine the security assurance level with respect to each of these notions, starting with the foundational property, cryptographic security.

2.1 Cryptographic security level

Lurk targets a security level of the order of 128 bits (or slightly less), and its cryptography components consistently achieve that level, including:

- The *Pallas curve*, with respect to discrete logarithm (likewise for the Vesta and BLS12-381 curves, other possible curves). Pallas has a 254-bit group order, thus offering approx. 126-bit discrete logarithm security.
- The *Groth16 zkSNARK*, based notably on filecoin-project/bellperson implementation, secure assuming:
  - The hardness of the discrete logarithm problem in the group used (that of the BLS12-381 curve, having security allegedly lower than 120 bits), as proven in the generic group model, with tight bounds.
  - A trusted setup securely performed.
- The *Nova recursive folding scheme*, implemented in microsoft/Nova. The folding mechanism used to aggregate iterations' proofs is secure assuming the hardness of the discrete logarithm problem in the group used, as proven in the random oracle model, with tight bounds. The proof is finalized and compressed using the *Spartan* SNARK, instantiated with the *Inner Product Argument* (IPA) as polynomial commitment scheme and is secure assuming hardness of the discrete logarithm problem in the group used. The current implementation does not support zero-knowledge and therefore the Lurk proof isn't either.
- The *Poseidon hash function* in *SAFE* mode, as implemented in filecoin-project/neptune, in a configuration targeting at least 126-bit collision resistance.
- Other cryptography components:
  - 256-bit hashes: BLAKE2s and SHA-256 in bellperson, BLAKE3 in multihash and CID generation, offering 128-bit collision security.
  - The OS PRNG, designed in Linux to offer at least 256-bit security (including the new 2022 kernel PRNG design).

The security of Lurk thus requires the theoretical security of the above, as well as the correctness and security of their implementations and dependencies thereof. We found that the cryptographic security level was consistent across primitives, and satisfying the stated security goals.

2.2 Completeness assurance

Completeness means that:

- Given a valid Lurk program, the proof system captures all the program's code and internal logic.
- Invalid/faulty Lurk programs are rejected by the proof compiler.

This requires a definition of a *valid Lurk program*, that one that:

- Is syntactically correct.
- Is deterministic.

Completeness requires that the *Evaluator* correctly processes syntactically correct Lurk programs, as sequence of *frames*. Edge cases in terms of program size and recursion depth are covered by the iteration limit set by the Evaluator (by default, 1000 iterations).

Completeness further requires completeness of the proof systems and their implementations (Groth16/SnarkPack+ or Nova).

In the version of Lurk we tested, syntactically invalid programs run in the REPL interpreter would either yield an error or cause the interpreter to panic, depending on the error type.

2.3 Soundness assurance

Soundness means that:

- Given a valid Lurk program, a proof covers accurately all operations in the program, and not that of another program.
- Invalid proofs are always rejected as invalid.
- Proofs are not malleable.
- Circuits are well-defined, and all variables are correctly constrained.

Soundness requires non-ambiguous evaluation of Lurk programs, as discussed in detail in the Soundness Notes document. This operation requires collision-resistant hashing with Poseidon, and sound encoding of expression types and their results, via Lurk's pointer-based internal data structures.

Soundness further requires completeness of the proof systems and their implementations (Groth16 and Nova).

Lower-level R1CS gadgets can be a source of circuit-soundness bugs, since they may be implemented using non-deterministic computation. Any variable whose value is computed "outside the circuit" must be properly constrained. Encapsulating this type of computation into small modules ensures they are easy to verify, and allows for safer higher-level circuit that don't need to know about the underlying constraint system.

Circuit-soundness can also be compromised when public inputs are not constrained or compared against other variables computed inside the circuit. In some sense, it is the same issue as with the

low-level gadgets since these values are computed externally and are therefore unconstrained by default.

The Lurk reduction circuit exposes two public input triples `input` and `output` of the form `(expr, env, cont)`. Each of these elements defines a "pointer" to some data, represented as a pair `(tag, hash)`, where `hash` is the Poseidon hash of the data, domain-separated by one of `tag` constants representing the type. In order to prove that the reduction of `input` produces `output`, the prover must provide the preimages of all pointers, and constrain the output values to be correctly derived from the inputs.

In order to handle all possible inputs, the circuit computes the output for all expression types, and conditionally selects the correct result depending on whether certain conditions are met. A classic reduction implementation would usually make use of many `if` and `match` expressions to manage this type of control-flow. Within a circuit, these can be emulated by constructing Boolean variables mapping to all possible branch conditions, along with the corresponding return value, and conditionally selecting one of the values depending on whether its condition was met. Several techniques are employed throughout the code base to simplify this branching emulation, such as specialized R1CS gadgets and Rust macros that enhance the functionality of the `bellperson` library.

2.4 Zero-knowledge assurance

Zero-knowledge means that:

- Given a valid Lurk program, its (aggregated) proof does not leak data on the underlying program.
- The creation of a proof does not entail the leakage of secret data to other parties than the prover.

Lurk supports creating proofs using either the Groth16+SnarkPack or Nova proof system implementation. These dependencies are responsible for ensuring that the proofs do not leak any information about the program's secret inputs.

The Groth16+SnarkPack backend satisfies the required zero-knowledge properties, but the current implementation of the Nova proof system requires some modifications to provide the same guarantees. Since it is based on Microsoft's Spartan implementation which does satisfy zero-knowledge, similar techniques can easily be adapted to Nova. In particular, the polynomial commitment scheme can be made perfectly hiding, ensuring zero-knowledge for Nova's folding scheme. The ZK versions of the Sumcheck and IPA sub-protocols from Spartan can then be used in Nova to ensure that the full proof satisfies zero-knowledge.

3 Security issues

3.1 S-LRK-01: [circuits] Soundness failure due to 0 value not enforced

In `selector_dot_product()`, the allocated variable `zero` was allocated but not enforced to be 0. This variable is then used as the default in `pick()`, which means it would have been possible to manipulate the final `result` variable to be anything. This was fixed in commit 4a61333 by passing a previously allocated `zero` variable from the global store.

3.2 S-LRK-02: [circuits] Soundness failure due to accumulator 1 initial value not enforced

In `multi_case_aux`, the initial accumulator variable `acc` is initialized to 1 but not enforced. By setting it to 0 instead, the variable `is_selected` would always be true. This was fixed in commit 4a61333 by passing the global store as input, and using the previously allocated `true_num` variable.

3.3 S-LRK-03: [circuits] Vectors not constrained to be of the same size

Some vectors were not constrained to be the same size, and could lead to uncaught bugs or runtime crashes:

- In `multi_case()`, adding the assertion `cases.len()== defaults.len()` led to the discovery of several soundness bugs due to a smaller `defaults` vector. The zipped iterator would skip over the trailing `cases` and skip adding the required constraints.
- In `selector_dot_product()`, checking `selector.len()== value_vector.len()` ensures the zipped iterator does not skip any entries. This was however correctly enforced by the callers, so no constraints were ignored.
- In `multi_case()`, adding `c.len()== cases[0].len()` would prevent a runtime crash when accessing `cases[0][j].key`, if the number of elements in each vector of cases is not equal.
- A check could be added to `multi_case_aux` to ensure that the all keys are unique, as this would otherwise create an invalid proof.
- An assertion could be added in `selector_dot_product()` to ensure that the provided `zero` variable actually has a value of 0. However, this would be incompatible with some constraint systems that do not actually assign values.

The commit d2c7c69 addresses the above issues by adding necessary assertions and modifying tests to validate these edge cases.

3.4 S-LRK-04: [circuits] Multi case number of defaults not enforced

The function `apply_continuation()`, would call `multi_case` with an incorrectly sized **`default`** argument. This led to some constraints not being enforced on some of the `cases`.

This was fixed in commit edcda97 by providing the correct number of arguments to the function, and adding an assertion to detect incorrect usage in the future.

3.5 S-LRK-05: [circuit] Under-constrained outputs in `reduce_sym`

In the case where the input expression is symbol, the circuit enforces the output of the reduction to be one of 10 possible results. All outcomes must be computed, but they are constrained to equal the output if and only if a specific set of conditions are satisfied. The specification defined by the Rust evaluation circuit returns the correct result by branching over different conditions. The circuit must mirror this behavior by computing all possible branching predicates, and constructing the correct conjunction that corresponds to a specific branch having been executed. As an optimization, the circuit creates disjunctions of all the predicates that would result in the same value being returned.

Originally, the circuit was written in a way that mirrored the Rust specification by constructing each of the 10 output predicates separately, and creating the disjunctions for the values that would be the same. The large number of branches added a lot of complexity, resulting in missed optimizations, unconstrained outputs, and unsatisfiable circuits in certain situations. These observations were noted in a review of PR 213 at commit 748c1e8. These observations were fixed in commit 7236df5 and commit 5e6409f.

The function `reduce_sym` was refactored and split into three parts. First, the circuit computes all the branch conditions and possible output values. Each one is then assigned a predicate that is true when that is the value that should be returned. Finally, constraints are added to enforce the outputs. These changes made the code easier to understand and compare with the Rust specification, while also reducing the total number of constraints.

3.6 S-LRK-06: [neptune] Zero padding not enforced

In the function `poseidon_hash_allocated()` from `filecoin-project/neptune/src/circuit2.rs`, if the `hash_type` is `ConstantLength`, and the `length` is smaller than the `arity`, the preimage buffer is padded with newly allocated zero variables. However, these are not actually constrained to be equal to zero.

This would make it possible to create a different hash than what is expected, but still pass Poseidon validation.

This was fixed in commit 2415d64, by constraining the padding values to be zero, via the new function `enforce_zero()`.

3.7 S-LRK-07: Leak of the number of iterations

The incrementally verifiable computation (IVC) construction implemented in the Nova proof system requires the number of iterations of the reduction circuit to be a public input. The number of iterations of a Lurk program is thus not confidential, which may leak information about what program has been executed or its inputs (unless the program is constant-time with respect to private inputs).

Fixing this would require modifications to the recursive Nova verification circuit, and ensuring the "recursion bound" are satisfied. We also recommend warning users of this limitation, and providing code examples or gadgets that are constant-time.

3.8 S-LRK-08: Nova lack of zero-knowledge guarantees

In its current form, the SNARK proofs produced by the Nova implementation do not satisfy the zero-knowledge property. The final "compressing" SNARK, based on the Spartan proof system, may leak some information about the last folded reduced R1CS instance due to the use of the Sum-check protocol, and a non-hiding variant of the IPA commitment scheme.

It's however unclear how such theoretical leakage can compromise the confidentiality of Lurk programs.

4 Other issues

We report other observations that are not security issues per se, but should probably be addressed nonetheless to improve Lurk's reliability.

4.1 O-LRK-01: [REPL] Inconsistent u64 division by zero failure (error vs panic)

As reported in the GitHub Issue 197, failure upon division by zero will either return an error or cause a panic, depending on whether the integer Euclidean division operator or the remainder operator is used:

```
1 > (/ (u64 5) (u64 0))
2 [7 iterations] => ERROR!
3 > (% (u64 5) (u64 0))
```

```
thread 'main' panicked at 'attempt to calculate the remainder with a
    divisor of zero', src/uint.rs:80:55
note: run with `RUST_BACKTRACE=1` environment variable to display a
    backtrace
```

4.2 O-LRK-02: [REPL] Assert of undefined symbols yields T

As reported in the GitHub Issue 198, asserts of undefined symbol appeared to yield `T`, instead of exiting with non-0 code.

4.3 O-LRK-03: [REPL] Relative path processing failure in REPL terminal

As reported in the GitHub Issue 205, the REPL interface failed to properly handle relative paths, but it appears that it had been fixed independently of our report.

4.4 O-LRK-04: Non-constant-time zero check

In num.rs, the check for zero value does not use the constant-time `is_zero()` function, but instead `is_zero_vartime()`, which might leak information about the tested value to an attacker measuring execution time:

```
impl<F: LurkField> Num<F> {
    pub fn is_zero(&self) -> bool {
        match self {
            Num::Scalar(s) => s.is_zero_vartime(),
            Num::U64(n) => n == &0,
        }
    }
```

This was fixed in PR 320 by replacing the call to `is_zero_vartime()` with `is_zero()`.

4.5 O-LRK-05: Potential undefined behavior in `uint` arithmetic

In uint.rs, the use of `unchecked` intrinsics in `unsafe` blocks can lead to undefined behavior. We suggest implementing safe versions, as in num.rs.

For example, `unchecked` multiplication is used in

```
impl Mul for UInt {
    type Output = Self;
    fn mul(self, other: Self) -> Self {
        match (self, other) {
```

```
            (UInt::U64(a), UInt::U64(b)) => UInt::U64(unsafe { a.
                unchecked_mul(b) }),
        }
    }
}
```

This was fixed in commit de95c5a, by using the `wrapping` operations instead of the unsafe unchecked ones.

4.6 O-LRK-06: Fast exponentiation in Lurk via parity trick

The Lurk library initially provided an exponentiation routine using the naive method (linear in the exponent value). This was done because Lurk provides no direct way to implement the parity checks required for square-and-multiply exponentiation. However, we noticed that the following field element sign convention can be leveraged to simulate such an operator:

```
    /// A field element is defined to be negative if it is odd after
        doubling.
    fn is_negative(&self) -> bool {
        self.double().is_odd().into()
```

This allows for these simple parity checks:

```
    (odd (lambda (a) (< (/ a 2) 0)))
    (even (lambda (a) (eq NIL (odd a))))
```

which in turn allow for the following efficient exponentiation implementation:

```
    ; computes b raise to power e
    (fastexp (lambda (b e)
        (if (= e 0) 1
            (if (even e) (fastexp (* b b) (/ e 2))
                (* b (fastexp (* b b) (/ (- e 1) 2)))))))
```

4.7 O-LRK-07: `unwrap_or()` optimization

The newly added code includes this function:

```
pub fn alloc_num_is_zero<CS: ConstraintSystem<F>, F: PrimeField>(
    mut cs: CS,
    num: Num<F>,
) -> Result<Boolean, SynthesisError> {
    let num_value = num.get_value();
    let x = num_value.unwrap_or(F::zero());
    let is_zero = num_value.map(|n| n == F::zero());
(...)
```

Here, the `unwrap_or()` should be replaced by `unwrap_or_else()` to only evaluate the code needed, as documented in RS-W1031.

This has been fixed in commit 5cd44dc.

5 Disclaimer

This [security assessment] report ("Report") by Inference AG ("Inference") is solely intended for Protocol Labs ("Client") with respect to the Report's purpose as agreed by the Client. The Report may not be relied upon by any other party than the Client and may only be distributed to a third party or published with the Client's consent. If the Report is published or distributed by the Client or Inference (with the Client's approval) then it is for information purposes only and Inference does not accept or assume any responsibility or liability for any other purpose or to any other party.

Security assessments of a software or technology cannot uncover all existing vulnerabilities. Even an assessment in which no weaknesses are found is not a guarantee of a secure system. Generally, code assessments enable the discovery of vulnerabilities that were overlooked during development and show areas where additional security measures are necessary. Within the Client's defined time frame and engagement, Inference has performed an assessment in order to discover as many vulnerabilities of the technology or software analyzed as possible. The focus of the Report's security assessment was limited to the general items and code parts defined by the Client. The assessment shall reduce risks for the Client but in no way claims any guarantee of security or functionality of the technology or software that Inference agreed to assess. As a result, the Report does not provide any warranty or guarantee regarding the defect-free or vulnerability-free nature of the technology or software analyzed.

In addition, the Report only addresses the issues of the system and software at the time the Report was produced. The Client should be aware that blockchain technology and cryptographic assets present a high level of ongoing risk. Given the fact that inherent limitations, errors or failures in any software development process and software product exist, it is possible that even major failures or malfunctions remain undetected by the Report. Inference did not assess the underlying third party infrastructure which adds further risks. Inference relied on the correct performance and execution of the included third party technology itself.