Least Authority

PRIVACY MATTERS

Loopring 3.6 Design + Implementation: Circuit

Security Audit Report

Loopring

Final Report Version: 16 March 2021

Table of Contents

Overview

Background

Loopring is a flexible layer-2 scalability solution for basic value transactions as well as a variety of exchanges such as order book and Automated Market Maker (AMM). This system uses advanced cryptography in the form of a limited one-way homomorphic encryption using bilinear pairings, popularized in the implementation of the Zcash protocol. This solution is classified as a validity proof system that ensures that state transition must be correct by the properties provided in the encryption scheme.

Loopring has requested that Least Authority perform a security audit of Loopring 3.6, a zkRollup layer-2 decentralized exchange and payment protocol implementation on the Ethereum blockchain. Loopring 3.6 is an improved version of Loopring 3.1, which is built on top of the same technical stack, and introduces Solidity smart contracts and `libsnark` and `ethsnark`-based circuit code.

Project Dates

- **November 25 - December 16:** Circuit Code review (*Completed)*
- **December 23:** Delivery of Circuit Initial Audit Report (*Completed)*
- **March 11 - 15**: Verification Review (*Completed)*
- **March 16**: Final Audit Report delivered (*Completed)*

Review Team

- JR, Cryptography Researcher and Engineer
- Mirco Richter, Cryptography Researcher and Engineer
- Jan Winkelmann, Cryptography Researcher and Engineer

Coverage

Target Code and Revision

For this audit, we performed research, investigation, and review of the Loopring 3.6 Circuit followed by issue reporting, along with mitigation and remediation instructions outlined in this report.

The following code subdirectory is considered in-scope for the review:

- Loopring 3.6 Circuit: https://github.com/Loopring/protocols/tree/a66e1db6a31879518ab08721bb73009deb15a3a1/packages/loopring_v3/circuit

Specifically, we examined the Git revisions for our initial review:

A66e1db6a31879518ab08721bb73009deb15a3a1

For the verification, we examined the Git revision:

5eb273fe76ba242c6a5f1bb3d1cd0edd57d070b4

This subdirectory was cloned for use during the audit and is linked for reference in this report:

https://github.com/LeastAuthority/loopring-protocols/tree/audit/packages/loopring_v3/circuit

All file references in this document use Unix-style paths relative to the project's root directory.

Supporting Documentation

The following documentation was available to the review team:

- Loopring 3.6 Design: https://github.com/Loopring/protocols/blob/master/packages/loopring_v3/DESIGN.md
- Loopring 3.6 README: https://github.com/Loopring/protocols/blob/master/packages/loopring_v3/README.md
- Loopring 3.6 vs. 3.1: https://github.com/Loopring/protocols/blob/master/packages/loopring_v3/security_audit/LooprinqV3_6_vs_V3_1.pdf
- Loopring 3.6 Circuit Documentation: https://github.com/Loopring/protocols/blob/d0eec91edc9bb195acbeddd38ebbdb71e6938127/packages/loopring_v3/circuit/statements.md
- R. Barbulescu, and S. Duquesne, 2017, "Updating key size estimations for pairings." *IACR Cryptol. ePrint Arch 2017/334* [BD17]
- L. Grassi, D. Khovratovich, C. Rechberger, A. Roy, and M. Schofnegger, 2019, "POSEIDON: A New Hash Function for Zero-Knowledge Proof Systems." *IACR Cryptol. ePrint Arch 2019/458* [Grassi et al.19]
- J. Groth, 2016, "On the Size of Pairing-based Non-interactive Arguments." [Groth16]
- D. Hopwood, 2019, "Designing efficient R1CS circuits." [Hopwood19]
- D. Hopwood, S. Bowe, T. Hornby, and N. Wilcox, 2020, "Zcash Protocol Specification." [Hopwood et al. 20]
- T. Kim, and R. Barbulescu, 2015, "Extended Tower Number Field Sieve: A New Complexity for the Medium Prime Case." *IACR Cryptol. ePrint Arch 2015/1027* [KB15]
- A. Menezes, P. Sakar, and S. Singh, 2016, "Challenges with Assessing the Impact of NFS Advances on the Security of Pairing-based Cryptography." *IACR Cryptol. ePrint Arch 2016/1102* [MSS16]
- Y. Sakemi, Ed. Lepidum, T. Kobayashi, T. Saito, NTT, R. Wahby, 2020, "Pairing-Friendly Curves." [Sakemi et al. 20]
- T. Perrin, 2016, "Curves for pairings." [P16]
- E. Baker, 2020 "Recommendation for Key Management: Part 1 – General." NIST Special Publication 800-57. [B20, Table 4]

Areas of Concern

Our investigation focused on the following areas:

- Common and case-specific implementation errors in the circuit code;
- Overflow protection against the SNARK scalar field;
- Attacks that impacts funds, such as the draining or the manipulation of funds;
- Mismanagement of funds via transactions;
- Proper management of encryption and signing keys;
- Protection against malicious attacks and other ways to exploit contracts;
- Inappropriate permissions and excess authority;
- Data privacy, data leaking, and information integrity;
- Performance problems or other potential impacts on performance; and
- Anything else as identified during the initial analysis phase.

Findings

General Comments

This audit of the Loopring 3.6 Circuit accompanies a previous audit our team conducted on the Loopring 3.6 Contracts for the Loopring Protocol. The previous report closely investigated how Loopring's smart contracts handle deposit, withdrawal, and block state updates. This report focuses on the way in which the Groth16-based zk-rollup circuits generate and verify proofs for each state update.

At the beginning of our audit, a clear and explicit description of Loopring's highly complex statement did not exist and statement details needed to be extracted from the code, which resulted in increased difficulty in auditing the code. Upon our recommendation, it was determined that the development of a SNARK statement would be a prerequisite to resuming and completing the audit. As a result, the Loopring team created a compact and more readable document containing the SNARK statement. We commend the Loopring team for dedicating their time towards this effort, which allowed for a more successful review by our team and demonstrates their commitment to the security of the protocol.

Our team's review of the Loopring circuit was able to ascertain that proofs work as expected in the intended cases. However, due to the complexity of Loopring's SNARK statement, determining whether the system proves unintended cases requires more robust system design. This includes a comprehensively documented SNARK statement (Suggestion 3) and code that is structured such that it clearly differentiates between the three foundational layers of SNARK development (Suggestion 4). As a result, we recommend that the Loopring team consider a subsequent audit of the circuit once the suggested system design changes have been implemented (See System Design).

Scope

Loopring implements a large and highly complex SNARK circuit [Groth16], with a statement of considerable length. The complexity of the statement presents an added layer of difficulty, in addition to the inherent complexities associated with SNARK circuits. This creates a challenge for reviewers, as it is difficult to estimate the associated `r1cs` solution set and determining if unintended behavior can nevertheless lead to valid `r1cs` solutions.

Dependencies

Loopring uses the established `mcl` library instead of alt_bn128 for curve arithmetic. The Loopring team forked `ethsnarks`, `libsnark`, `libff` and `libfqfft` and modified the libraries to accommodate the newly added dependency. None of these forks were considered in scope for the audit and we estimate the likelihood of potential security issues with these to be very low.

The circuit code depends heavily on `ethsnarks`, which depends on `libsnark`. Due to the heavy use of Git submodules within the dependencies, problems encountered when attempting to build the system from a pinned commit resulted in build issues related to dependencies. These issues prevented our team from running test code during the course of the assessment. Furthermore, the dependencies comprised a series of Git submodules that were compared against sources of known vulnerabilities and no vulnerabilities were identified for these dependencies.

Code Quality

Given the challenges of writing complex SNARK systems, the Loopring team demonstrated good organization of the code by using classes and abstraction where necessary. However, since the high-level circuits and gadgets rely on several tiers of sub-gadgets, navigating the hierarchy to clearly understand the functionality of the circuits is more challenging.

Tests

Tests are present in the circuit/test directory, which demonstrate the intended use and error catching mechanisms. While the test cases are not thoroughly explained in the comments, the test names are often self explanatory vis-à-vis their intended function. We recommend expanding the test cases to model advanced adversaries' potential attempts to violate the assumptions of the respective gadgets (Suggestion 7). This should be done once the security assumptions and the intended use of the circuit are explicitly documented in the SNARK statement and the code comments (Suggestions 3; Suggestion 5).

Standards and Best Practices

SNARK research and development is still in its early stages and there are currently only a few well-established standards and best practices. As a result, we recommend that active developers in this field maintain a clear distinction between the three fundamental layers of SNARK development: protoboard allocation of variable slots, r1cs-generation, and witness computation. In addition, we recommend consistently maintaining a distinction between computation and constraint enforcement, as well as a clear delineation between instance and witness variables. In a considerable number of instances throughout the code base, it is unclear whether witness variables are being passed in through the constructor or whether those are public instance variables. As a result, we recommend that these distinctions be made clear in the design phase of the statement, prior to implementing them in the code (Suggestion 4).

Documentation

Our team found that a justification for the used Poseidon parameters was not given in the documentation. The Loopring team explained they were derived using a script found in the repository. This script was compared to the Poseidon paper [Grassi et al.19] and found to be properly implemented in the Loopring circuit.

Code Comments

While the aforementioned statement descriptions contribute to the understanding of how the system works, the inclusion of more extensive code comments is strongly recommended, given the complexity of the project. Code comments should explain the use of each building block within the system, as well as the reasons for using particular constraints, explicitly specifying the intended purpose and function of each component and, more importantly, the purpose and function it should not perform. In addition, a comprehensive list of all assumptions made by the gadget should be provided (Suggestion 5). This would help delineate the proper use of the circuit from its potential misuse, while facilitating better understanding and easier review of the code, thus making potential issues more visible and resulting in a more robust and secure system.

SNARK Statement

The high-level SNARK statement developed by the Loopring team contributed significantly to our understanding of how the system works. However, it is critical that the documentation is maintained and updated regularly as an accurate point of reference for the coded implementation, as inconsistency in the documentation and the implementation could result in confusion or errors (Suggestion 3). Furthermore, the existing preliminary draft of the statement should adhere to the conventions of proper statement design (See SNARK Statement Design).

System Design

Use of Cryptographic Algorithms

The Groth16 proof system internally uses elliptic curve pairings. These computations are non-trivial and require support from precompiled contracts in order to keep gas costs at a reasonable level. This allows the computation to be performed natively on the machine executing the smart contract instead of being performed on the Ethereum Virtual Machine (EVM). Currently, precompiled contracts are available only for the BN254 curve (which, among other names, is also called BN128). However, due to more recent advances in number theory, this curve is no longer considered to provide the security we have come to expect from cryptographic algorithms. In light of this, Ethereum developers have decided to include precompiled contracts for operations on the pairing-friendly curve BLS12-381, which is assumed to provide sufficient security. The precompiled contract is found in the Ethereum Berlin hard fork, which is expected to take place in January 2021. As a result, we recommend that the Loopring team implement the curve pairing-friendly curve BLS12-381 (Issue A).

SNARK Statement Design

The design of SNARK circuit systems should begin with proper statement design, as recommended by [Hopwood19]. If this is not possible due to the complexity of the statement, parallel development of both statement design and implementation is strongly recommended. As previously noted, the Loopring circuit statement was written following the completion of the coded implementation by extracting it from the code. Given that this does not adhere to recommended best practices of SNARK development, we suggest that an extensive and rigorous statement definition be written (Suggestion 3).

Floating Point Values

In the `FloatGadget`, monetary values are represented as floats, which is not considered best practice since precision and rounding errors occur on floating point number calculations. While intended as a means to save space in transactions, the added complexity in the circuit did not make the optimization appear advantageous. As a result, we recommend using regular integer representation instead (Suggestion 6).

Specific Issues & Suggestions

We list the issues and suggestions found during the review, in the order we reported them. In most cases, remediation of an issue is preferable, but mitigation is suggested as another option for cases where a trade-off could be required.

ISSUE / SUGGESTION	STATUS
Issue A: The BN254 Curve Provides Insufficient Security	Unresolved
Suggestion 1: Use the `emplace_back` Method Consistently Across Gadgets	Resolved
Suggestion 2: Remove Unnecessary Bitness Check in `ArraySelectGadget`	Resolved
Suggestion 3: Write A Comprehensive Accompanying SNARK Statement	Partially Resolved
Suggestion 4: Clearly Distinguish Between Computing Witnesses and Enforcing Constraints	Resolved
Suggestion 5: Expand Code Comments	Unresolved

Suggestion 6: Use Regular Integer Representation Instead of 24 Bit Floating Point Values	Unresolved
Suggestion 7: Expand Test Suite to Enforce Security Assumptions	Unresolved

Issue A: The BN254 Curve Provides Insufficient Security

Location

/CMakeLists.txt#L47-L53

Synopsis

Loopring uses a variant of the BN254 curve. In 2016, advances in number theory led to a lower security estimate of that curve. Specifically, its security is now considered to be around 96 bits. This is significantly lower than the 112 bits required by NIST for new products.

Impact

Use of the BN254 curve undermines the security of the zk-SNARK scheme, such that the feasibility of computing valid, forged proofs cannot be ruled out. Such proofs would pass validation, yet violate the constraints imposed by the circuit. For example, a valid, forged proof could increase or reduce the balance of accounts arbitrarily.

Preconditions

In Loopring, only proofs published by the operator are considered. As a result, an attacker needs access to the operators keys, either through being the operator or through having gained access to them by different means.

Feasibility

The exact feasibility is difficult to estimate. However, the potential gains from a successful attack are high, which suggests that an attacker would have incentive to invest significant resources.

Technical Details

Attacks based on the Tower Number Field Sieve (TNFS) and the derivative exTNDS and SexTNFS have led to a reduced estimate of the security level of BN254. The several scholars and practitioners working on this issue do not entirely agree on the new estimate, but opinions range from 96 bits to 110 bits [KB15, MSS16, BD17, Perrin16]. Regardless of where on this spectrum the real value falls, it is still too low. Even for applications that only need to remain secure until 2030, NIST requires a security level of at least 112 bits [B20, Table 4], which BN254 does not achieve.

Mitigation

This attack can be detected, however, it would require that one or more parties permanently check for forged proofs.

Remediation

We recommend using the curve BLS12-381. According to the draft RFC on pairing-friendly curves [Sakemi et al. 20], it has a security level of ~128 bits, which is above the 112 bits considered sufficient by NIST until 2030.

The Berlin hard fork, which is planned to take place in January 2021, will bring precompiled contracts support for operations on this curve. This will make using the curve viable. Given the potentially high incentive for launching such an attack, running a new trusted setup computation to generate a required common reference string for BLS12-381 should be strongly considered. As a result, we recommend this approach as an appropriate long-term solution for Loopring.

Status

The Berlin hard fork upgrade will no longer contain EIP-2537, as originally planned at the time we delivered the *Initial Audit Report*. As a result, 384 bit arithmetic in the EVM (i.e. EVM-384) is currently unavailable. Given this change, there is currently no efficient method for secure pairing based cryptography that is able to achieve at least 112 bits of security as required by NIST for new products. Thus, a long-term remediation is not currently possible and we recommend that the Loopring team continue to monitor developments with EIP-2537.

Verification

Unresolved.

Suggestions

Suggestion 1: Use the `emplace_back` Method Consistently Across Gadgets

Location

/circuit/Gadgets/MathGadgets.h#L1645

/circuit/Gadgets/MathGadgets.h#L1809

Synopsis

The `emplace_back` method of `std::vector` is usually called with the constructor arguments and then constructs the new value inside the new vector. Compared to constructing the value and then appending it with `push_back`, this approach saves one copy. While most of the code uses this pattern, in `FloatGadget` and `SelectGadget`, the `TernaryGadget` is explicitly constructed, which introduces an unnecessary copy.

Mitigation

Remove the explicit constructor call and let `emplace_back` perform the creation of the new value.

Status

The Loopring team has removed the unnecessary constructor allowing `emplace_back` to construct the object itself.

Verification

Resolved.

Suggestion 2: Remove Unnecessary Bitness Check in `ArraySelectGadget`

Location

/circuit/Gadgets/MathGadgets.h#L1871

Synopsis

`ArraySelectGadget` is analogous to `SelectGadget`, except that it constrains a `VariableArrayT` instead of a `VariableT`. In `SelectGadget`, the bitness checks of the conditional value in the `Ternary` operator are explicitly disabled (because they are not required). However, in `ArraySelectGadget`, they are not disabled.

Mitigation

We suggest either letting the parent gadget decide whether the check should be performed or disabling it for consistency.

Status

The Loopring team implemented a fix where the `enforceBitness` parameter of `generate_r1cs_constraints` is explicitly set to false, thus removing the unnecessary bitness check.

Verification

Resolved.

Suggestion 3: Write A Comprehensive Accompanying SNARK Statement

Location

/circuit/statements.md

Synopsis

SNARKS are short and computationally sound proofs for the existence of witnesses to given statements. In order to correctly perform a security audit on a SNARK, it is fundamental to start with a clear and formal definition of a SNARK statement. Without such an abstract definition, there is no foundation to compare the implementation against. It is insufficient to have the statement implicit in the code, as this would force a circular approach for reviewers, consisting of comparing the code against a statement that is implicit in the code.

The statement documentation created by the Loopring team is helpful, however, we identified subtle errors in comparison to the actual functionality of the gadgets. For example, in the `RequireFillsGadget`, it appeared as if a `TernaryGadget` was being used improperly when in fact, it was used correctly but the documentation itself was incorrect and therefore misleading. These inconsistencies were reported to the Loopring team and were promptly corrected once they concluded that the circuit implementation was correct.

In addition, there are repeated instances in the documentation of computation (witness generation) and r1cs-enforcement being used interchangeably and it is often unclear what exactly is computed and what is constrained. SNARKS are system critical, cryptographic primitives and any implementation should adhere to the same rigor and documentation as every other crypto-primitive (e.g. hash functions). As a result, our team has determined that the statement definition requires further improvement.

Finally, gadget descriptions are not provided with a list of assumptions that a gadget has to make on its inputs. For example, in the `UpdateAccountGadget`, the address is assumed to be Boolean constrained elsewhere, which is not clear from the description of the gadget. In addition to the assumptions made, a reference should be given to where those assumptions are satisfied. We consider this essential, given that auditing the statement on an abstract mathematical level is as important as performing an audit on its corresponding coded implementation.

Mitigation

Write an extensive and rigorous statement definition, adhering to the best practices of SNARK development. In addition, institute regular documentation reviews to ensure the documentation remains up to date and consistent with the implementation.

Status

In response to our suggestion during the course of the audit, the Loopring team began the efforts of writing a statement description, which was iteratively improved based on our feedback and recommendations. However, the final version is absent of cryptographic rigor, a proper list of assumptions that the gadget witnesses must satisfy, and a clear distinction between computation and constraining of witness variables. Importantly, there are still repeated instances in the documentation of computation (witness generation) and r1cs-enforcement being used interchangeably and it is often unclear what exactly is computed and what is constrained. We recommend that the Loopring team continue to improve the SNARK statement until it satisfies the requirements of SNARK statement design best practices.

Verification

Partially Resolved.

Suggestion 4: Clearly Distinguish Between Computing Witnesses and Enforcing Constraints

Location

Example:
[/circuit/Gadgets/StorageGadgets.h#L31]

Synopsis

According to [[Groth16]], inputs to the verifier are called instance variables, while all other factors in any r1cs solution are called the witness. Therefore, strictly speaking, Loopring's SNARK only has a single instance value (the hash of the public inputs). Those variables should be handled in `ethsnark`'s `generate_r1cs_witness()` function. In contrast, the constructor should only assign the slots for these variables (allocate them on the protoboard), but not assign actual values. However, such a clear distinction has not been made in the code, making it difficult to judge in both the prover phase or in the verifier phase which constraints and constants are known at compile time.

In contrast, adhering to a clear distinction would highlight the separation between the various phases (generator, prover, and verifier) during code execution, therefore greatly increasing the ability for successful review of the project.

For example, the `StorageGadget` associates the actual values of `data` and `storageID`, both in a constructor and in the `generagte_r1cs_witness`() function. Despite the fact that the mentioned constructor is never used, it makes execution-phase separation harder to understand from the reviewers' perspective.

As a result, a distinction between witness computation and constraint enforcement is necessary to draw attention to the separation between the various phases during code execution, which is not currently present in the substatements. In order to adhere to SNARK development best practices and make reviews of the code more feasible, we recommend that the code be restructured as such.

Mitigation

Restructure the code to adhere to a clear distinction between r1cs generation, proof generation, and proof verification.

Status

The Loopring team [has updated] the `DualVariableGadget` by splitting it into `FromBitsGadget` and `ToBitsGadget`. In addition, the team removed the unused constructor from the

`DynamicVariableGadget` and the `StorageGadget` to facilitate a better separation of witness computation and constraining.

Verification

Resolved.

Suggestion 5: Expand Code Comments

Location

Examples:
/circuit/Circuits/AccountUpdateCircuit.h

/circuit/Circuits/BaseTransactionCircuit.h

Synopsis

We found the code comments to be insufficient in a considerable number of areas in the code and that the existing descriptive comments require further clarification. Code comments within the codebase are critical for developers and reviewers, as they help to define and explain the purpose of each gadget and a description of the intended functionality. It would be helpful for each gadget to be commented on, clearly describing the assumptions made within the gadget and providing references to the part of the code where those assumptions are satisfied (e.g. enforcing Booleanness of an address in the `AccountUpdateGadget`).

Mitigation

We recommend that the Loopring team expand code comment coverage, edit existing comments for clarity, and update the gadget comments such that they describe the intended behavior.

Status

The Loopring team has acknowledged this suggestion and have stated that they intend to improve code comment coverage if the opportunity arises. At the time of this verification, code comments have not been further expanded.

Verification

Unresolved.

Suggestion 6: Use Regular Integer Representation Instead of 24 Bit Floating Point Values

Location

/packages/loopring_v3/circuit/Gadgets/MathGadgets.h#L1617

/packages/loopring_v3/circuit/Gadgets/MathGadgets.h#L1628

Synopsis

While using floating point numbers in the calculation of monetary values is not generally considered best practice, the reasons for using them in this context are particularly unclear. The 24 bit float encoding uses 5 bits for the exponent and 19 bits for the mantissa, resulting in a maximum value of 2^19*10^5-1, which in regular integer representation can be encoded using 36 bits. This results in a storage cost of 12 bits more than the float representation, but provides perfect accuracy, no conversions, and very simple arithmetic throughout the code base. However, the representation is inaccurate and the conversions incur a cost in the form of constraints, thus reducing proving performance. The decision for this approach is unclear as the tradeoff of saving a few bits of storage per transaction does not appear to be worthwhile.

There are larger space savings for the 16 bit encoding, however, inaccuracies further increase. These floats are only used for protocol fees, in which this is permissible.

Mitigation

Use regular integer representation instead of 24 bit floating point representations.

Status

The Loopring team has responded that the use of floating point numbers was a design decision intended to reduce gas costs as well as prove generation costs. They have noted that since no calculations are performed on float point numbers, they consider the resulting inaccuracies to be an acceptable trade-off. We recommend that the Loopring team continue to consider the security implications of such design decisions, in order to make informed decisions about security trade-offs.

Verification

Unresolved.

Suggestion 7: Expand Test Suite to Enforce Security Assumptions

Location

Everything included in circuit/tests/

Synopsis

At present, the tests largely verify whether variables are correctly formatted, however, they were not found to model advanced adversaries potential attempts to violate the assumptions of the respective gadgets. For example, in the MerkleTree tests, apart from the `Everything correct` test case, the tests mainly test off-by-one errors and an incorrect index.

While protecting against code regression, these test cases do not contribute to maintaining the security of the system by modeling actions by malicious actors. This follows from what seems to be ambiguity in the circuit documentation, where clear statements would give insight into the security assumptions of the circuit in addition to its functionality (Suggestions 3; Suggestion 5). We suggest incorporating tests that perform this function, in order to help protect against the potential for malicious actions.

Mitigation

We recommend expanding the test cases after a more thorough documentation of the security assumptions of the circuit are generated, with an eye to modelling against malicious actors attempting to forge proofs.

Status

The Loopring team has responded that, by design, the unit tests for higher level gadgets are intended only to check for success and failure cases and that more thorough testing is performed in the underlying gadgets tests. They also note that testing if all linked variables are constrained together in a test would require the modification of the internal variables of all the gadgets, which they believe will introduce additional complexity, and potentially, bugs into the code base.

Our team believes that this design decision is costly in the absence of a complete SNARK statement, which clearly and thoroughly defines the assumptions (see Suggestion 3). In addition to a comprehensive list of assumptions that would provide insight into edge cases, we recommend increasing test coverage to model advanced adversaries potential attempts to violate the assumptions of the respective gadgets.

Verification

Unresolved.

Recommendations

We recommend that the unresolved and partially resolved *Issue* and *Suggestions* stated above are addressed as soon as possible and followed up with verification by the auditing team.

We consider the security level of the BN254 curve to be insufficient, as it has been determined by more recent research to be less suitable than more secure alternatives, such as BLS12-381. We recommend switching to BLS12-381 once that opportunity becomes possible, as the high rewards of a successful attack provide sufficient incentive to potential attackers to exploit this vulnerability.

We encourage the Loopring team to consider the importance of an abstract statement definition that is separate from the code itself by expanding on the current specification and checking for consistency, explicitness, and adherence to best practices. These best practices should also be enforced in the code, ensuring a clear distinction between r1cs generation, proof generation, and proof verification.

This effort can be considerably aided by improving documentation coverage, including the addition of code comments and more comprehensive test coverage to model actions of potential attackers attempting to forge proof, as well as better consistency between the specification and corresponding coded implementations. The application of these development best practices will facilitate an easier understanding for users, implementers and reviewers, in addition to enhancing the overall security of the code.

Finally, as Loopring's design documentation and mathematically rigorous statement specification matures, we recommend that follow up audits of the Loopring circuit be conducted, once the findings of this report are addressed and verified.

About Least Authority

We believe that people have a fundamental right to privacy and that the use of secure solutions enables people to more freely use the Internet and other connected technologies. We provide security consulting services to help others make their solutions more resistant to unauthorized access to data and unintended manipulation of the system. We support teams from the design phase through the production launch and after.

The Least Authority team has skills for reviewing code in C, C++, Python, Haskell, Rust, Node.js, Solidity, Go, and JavaScript for common security vulnerabilities and specific attack vectors. The team has reviewed implementations of cryptographic protocols and distributed system architecture, including in cryptocurrency, blockchains, payments, and smart contracts. Additionally, the team can utilize various tools to scan code and networks and build custom tools as necessary.

Least Authority was formed in 2011 to create and further empower freedom-compatible technologies. We moved the company to Berlin in 2016 and continue to expand our efforts. Although we are a small team, we believe that we can have a significant impact on the world by being transparent and open about the work we do.

For more information about our security consulting, please visit https://leastauthority.com/security-consulting/.

Our Methodology

We like to work with a transparent process and make our reviews a collaborative effort. The goals of our security audits are to improve the quality of systems we review and aim for sufficient remediation to help protect users. The following is the methodology we use in our security audit process.

Manual Code Review

In manually reviewing all of the code, we look for any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behavior when it is relevant to a particular line of investigation.

Vulnerability Analysis

Our audit techniques included manual code analysis, user interface interaction, and whitebox penetration testing. We look at the project's web site to get a high level understanding of what functionality the software under review provides. We then meet with the developers to gain an appreciation of their vision of the software. We install and use the relevant software, exploring the user interactions and roles. While we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation. We hypothesize what vulnerabilities may be present, creating Issue entries, and for each we follow the following Issue Investigation and Remediation process.

Documenting Results

We follow a conservative, transparent process for analyzing potential security vulnerabilities and seeing them through successful remediation. Whenever a potential issue is discovered, we immediately create an Issue entry for it in this document, even though we have not yet verified the feasibility and impact of the issue. This process is conservative because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this we analyze the feasibility of an attack in a live system.

Suggested Solutions

We search for immediate mitigations that live deployments can take, and finally we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinized by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our report, and before the details are made public.

Responsible Disclosure

Before our report or any details about our findings and suggested solutions are made public, we like to work with your team to find reasonable outcomes that can be addressed as soon as possible without an overly negative impact on pre-existing plans. Although the handling of issues must be done on a case-by-case basis, we always like to agree on a timeline for resolution that balances the impact on the users and the needs of your project team. We take this agreed timeline into account before publishing any reports to avoid the necessity for full disclosure.