RADICALLY OPEN SECURITY

# Cryptographic Analysis Report

## Nimue library

V 1.1
Amsterdam, July 9th, 2024

## Document Properties

| | |
|---|---|
| Client | Nimue library |
| Title | Cryptographic Analysis Report |
| Target | Nimue library. |
| Version | 1.1 |
| Pentester | Younes Talibi Alaoui |
| Authors | Younes Talibi Alaoui, Peter Mosmans |
| Reviewed by | Peter Mosmans |
| Approved by | Melanie Rieback |

## Version control

| Version | Date | Author | Description |
|---|---|---|---|
| 0.1 | June 25th, 2024 | Younes Talibi Alaoui | Initial draft |
| 0.2 | June 26th, 2024 | Peter Mosmans | Minor textual edits |
| 0.3 | June 27th, 2024 | Younes Talibi Alaoui | Minor edits |
| 1.0 | June 27th, 2024 | Peter Mosmans | 1.0 Release |
| 1.1 | July 9th, 2024 | Younes Talibi Alaoui | Additional edits |

## Contact

For more information about this document and its contents please contact Radically Open Security B.V.


| | |
|---|---|
| Name | Melanie Rieback |
| Address | Science Park 608<br>1098 XH Amsterdam<br>The Netherlands |
| Phone | +31 (0)20 2621 255 |
| Email | info@radicallyopensecurity.com |

# Table of Contents

# 1 Executive Summary

## 1.1 Introduction

Nimue is a library designed to aid in creating multi-round public coin zero-knowledge proofs, by assisting the Prover generate the protocol transcript and assisting the Verifier in the verification process.

In this report, we provide a code review over the library [3] , targeting the latest version available at the time of the start of the review. [1]

## 1.2 Scope of work

The code review consisted of inspecting the different components of the library. Note however that the `group`plugin, the `legacy`module, the `Anemoi`hash function, and the `bulletproof`example, received less focus than the rest of the library.

## 1.3 Summary of findings

We identified five findings during our code review:

1. The first one is of high severity, in that it might affect the soundness of the proofs.
2. The second one is of high severity, in that it leads to raising an error by the library when the protocol transcript is being constructed, or in an extreme case, it might affect the soundness of the proofs.
3. The third finding is of minor severity, in that it is problematic only in a use case that is less likely to happen.
4. The fourth finding is of minor severity, in that we could not identify how to exploit it.
5. The fifth finding is minor, in that it only concerns a test file, and when fixed the test still passes.

Note however that all these findings are fixable. We also provide in this report how to fix them. The findings and the fixes are given in Section 3. We also identified some typos in the library: These are given in the appendix of this report.

[1] commit `923e0fde49610dfff76c23ad0ca811442ae21e98`

# 2 Overview of the Nimue library

The Nimue library [2] , [3] enables the Prover to generate the protocol transcript for a public-coin zero knowledge-proof, by both provisioning the necessary randomness for the Prover in order to execute the protocol, as well as the generation of the public coins for the zero-knowledge proof. The library enables as well the Verifier to verify the zero-knowledge proof, by deserializing the transcript and generating the public coins, which can be further used in order to perform the necessary calculation in order to verify the proof.

The library does so, by performing a Fiat Shamir transform over the zero knowledge-proof. The hash function used to perform the transform, takes as input a well defined tag, unique to every protocol. This tag is derived from the protocol description, referred to as the IO pattern. That is, similarly to the SAFE paper [1] , Nimue uses a duplex sponge [10] in order to absorb and squeeze data. The data squeezed serves as the public coins used in the protocol. This sponge is referred to as the Public Sponge in Nimue. When the Verifier receives the protocol transcript from the Prover, he uses the same sponge, namely he seeds the sponge with the same tag used by the Prover, and generates the public coins by squeezing the sponge, after performing the necessary absorbs of data that he extracts from the transcript. The Verifier then can use the public coins in order to perform the necessary checks for verifying the proof. Note however that Nimue supports an extra operation compared to the SAFE paper that consists of ratcheting the state of the sponge. More details about this operation are given in the next section.

As for the randomness needed by the Prover in order to execute the protocol, the library provides it by using another duplex sponge (referred to as the Private Sponge) that works similarly to the Public Sponge. The design of this sponge is similar to how the randomness is generated in the Merlin library [4] , in that the generation of the randomness involves the protocol transcript throughout the execution of the protocol. That is, Nimue uses for the Private Sponge a Keccak Permutation, sets the tag to be the empty string, then performs an absorb of the IO pattern of the protocol. further on, whenever the Public Sponge absorbs data (which is the data that composes the transcript, hence the protocol transcript being involved in the randomness generation for the Prover), the Private Sponge as well absorbs the same data, then before each squeeze, the Private Sponge absorbs randomness generated by a cryptographic random number generator [7] . After the squeeze a ratchet operation is performed (see next section for more details regarding this operation).

Nimue also provides bindings to some popular zero-knowledge proofs libraries, namely the algebra crate from Arkworks [9] and the group crate from Zkcrypto [8] .

For more details about the technical terms employed in this section, one can refer to [2] and [1] .

## 2.1 IO patterns in Nimue

While ABSORB and SQUEEZE are the only operations admitted in an IO pattern of the SAFE paper, Nimue admits an extra one called RATCHET. RATCHET performs a permute operation on the current state, then fills in the rate with zeros. This serves to start from a clean rate after a SQUEEZE, particularly for the Private Sponge.

In addition to this, while the IO pattern in the SAFE paper operates over only field elements, Nimue is more general, in the sense that it can admit other native elements, such as group elements and bytes, and even allows combinations of them, such as having an IO pattern that operates over both bytes and field elements. Moreover, in both Nimue and the

SAFE paper, there is a field called the domain separator, that can be used in order to describe the IO pattern. In addition to this, the operations ABSORB and SQUEEZE in Nimue are attached to a field set by the user (called label), this field can be used to describe the operation.

Furthermore, the Sponge of Nimue supports both algebraic hash functions (hash functions that operate over field elements), and hash functions that operate over bytes, whereas in the SAFE paper, the Sponge admits only algebraic hash functions. In fact, Nimue allows different combinations between the IO pattern and the hash function used, such as having an algebraic hash function while the IO pattern is defined over bytes, while in the SAFE paper, the field used of the IO pattern is the same as the one of the algebraic hash function.

Some of the hash functions already supported in Nimue are Keccak, Poseidon, Anemoi. It provides as well support for the legacy hash functions that satisfy the Digest trait [6] . The implementation of this is inspired from the libsignal [5] , for more details one can refer to `legacy.rs`

As such, in both the Nimue and the SAFE paper, only knowing the IO pattern does not give all the details about the protocol, such as the hash function being used for the Public Sponge, the native elements in the case of Nimue or the field in the case of the SAFE paper over which the IO pattern is defined. Therefore, the Prover and the Verifier, before using either Nimue or SAFE, should be able to know what the IO pattern is defined over, as well as which hash function is being used.

## 2.2 Tag

In the SAFE paper, there is a transformation that happens to the IO pattern before hashing it to obtain the tag. The transformation consists of encoding the IO pattern as a list of 32-bit words, then aggregating any contiguous ABSORB or SQUEEZE calls, then serializing the IO pattern. In Nimue, the tag is calculated by hashing directly the string representing the IO pattern.

In the SAFE paper, patterns that belong to the same equivalence class (roughly speaking, patterns that lead to the same execution of what the IO pattern describes) lead to the same tag. This is thanks to the transformation that happens to obtain the tag. In Nimue, we can have different tags for the same equivalence class, as the user can add labels next to the operations (which means that what the user sets for the labels will be part of what is being hashed to obtain the tag). Besides, no aggregating step happens in Nimue before calculating the tag. Note however that this difference in calculating the tags should not be an issue, as long as the Prover and the Verifier use the same IO pattern.

# 3 Code Review

## 3.1 Squeeze calls of bytes for the case of an algebraic hash function

In the context of an IO pattern that squeezes bytes with an algebraic hash function, we found that in some cases, the function `bytes_uniform_modp`was used in order to determine how many uniformly random bytes we can obtain out of a uniformly random field element (which is less in size than the bytes we need to represent the field element), while actually `bytes_uniform_modp` determines how many uniformly random bytes we need in order to calculate a uniformly random field element (which is larger in size than the bytes we need to represent the field element). This happened in the trait implementation of `ByteChallenges` for `Merlin<H, Fp<C, N>, R>` and for `Arthur<'a, H, Fp<C, N>>` in `nimue/src/plugins/ark/common.rs`, and in the trait implementation of `ByteIOPattern` for `IOPattern<H, Fp<C, N>>` in `nimue/src/plugins/ark/iopattern.rs`.

As a consequence, this function leads to generating bytes for the user that are supposed to be uniformly random while they are not.

If the generated bytes were intended to construct afterwards an element from the same field of the algebraic sponge, then it would have been secure to use these bytes, and it actually even allows the user to have a uniformly random field element without relying on a statistical security parameter. That is, when generating a uniformly field element from uniformly random bytes, we rely on a statistical security parameter to obtain field elements from a distribution close to uniformly random, and that is why we need to generate more bytes than the ones we need to represent the field element.

That is why if the intention of the user consisted of generating a uniformly random element from the same field of the algebraic sponge, then using the bytes generated by the library to construct the random field element is secure as they consist of a representation of uniformly random field element of the same field. However, if the bytes were intended to be used as uniformly random bytes, then some of the most significant bytes have to be discarded.

The library actually contains a script ( `useful_bits_modp.py` ) that implements a function that is intended to output the number of uniformly distributed bits we can obtain out of a uniformly random mod `p` integer, where `p` is a prime number given as input to the function. However this function was not used in the library.

### 3.1.1 Fix for finding

To address this issue, the `bytes_uniform_modp` has to be replaced with a function that determines how many uniformly random bytes we can obtain from a uniformly random field element, such as the function implemented in the python script. We also suggest to attach a reference to the script, to the result it is implementing. In addition to this, we suggest adopting a different name for `bytes_uniform_modp` as it is misleading.

## 3.2 `fill_challenge_bytes` for the case of an algebraic hash function

For the case when the function `fill_challenge_bytes` is used in the context of an IO pattern that contains byte squeezes, with a sponge defined with an algebraic hash function (in `here` and `here`), we found that there are situations where the Prover does not obtain all the randomness he asked for. That is, let us denote by `l` the number of bytes that the Prover wants to generate from the sponge, by `m` the number of bytes that we need in order to represent a field element coming from the sponge, and by `u` the number of uniformly random bytes we can obtain from a field element coming from the sponge. The function `fill_challenge_bytes` was written in a way that the number of the bytes filled by randomness depends on the result of the comparison between `m` and `min(l,u)`, which leads in some cases to the library raising an error while constructing the transcript, or in an extreme case filling some bytes with zeros if there was no absorbs afterwards in the IO pattern. Note however that the usage of Nimue requires that the last response is also absorbed in the sponge so that the repsonse is added to the transcript, thus the latter case was not going to happen if the user was following the usage by default of Nimue.

### 3.2.1 Fix for finding

In order to fix this issue, we recommend removing the dependence of how many squeezes to perform inside the function with the comparison explained above. Instead, we recommend performing all the necessary squeezes inside the function, till all the randomness requested by the user is provided. The number of squeezes to ensure this was already calculated in the trait implementation of `ByteIOPattern` for `IOPattern<H, Fp<C, N>>` in `nimue/src/plugins/ark/iopattern.rs`, in order to determine how many squeezes from the sponge are needed. It can be also recalculated inside the function `fill_challenge_bytes` using `l` and `u`. Then after every squeeze of a field element, one can keep the least significant `u` bytes and discard the rest of bytes.

## 3.3 `swap_field` function

The `swap_field` function defined in the ark plugin, is intended to provide an injective map `f`, between field `F_1`, and field `F_2`, so that every element `x` in `F_1`, has exactly one representative `y` in the field `F_2`, so that when mapped back to `F_1` (using the swap function from `F_2` to `F_1`), `y` will result in `x`, and that no value `y` not in `f(F_1)` can be mapped back to `F_1`.

The context in which this function was used is that the Prover before including an element `x` from `F_1` in the transcript, might move it first to an element `y` in field `F_2`, then when the Verifier receives the transcript, he transforms back `y` to `x`. This is the case for instance in the `schnorr_algebraic_hash.rs`.

The issue with this function is that guaranteeing the injective property is only possible if `F_2` is larger than `F_1`.

However, thanks to a check included in the swap function, if `F_1` is larger than `F_2`, the swap function will be only able to map `|F_2|` elements from `F_1` to `F_2`, while an error will be raised when another element not in the `|F_2|` elements that the function can map is being mapped.

As a consequence, we might have two outcomes. The first one is that an error is raised when the Prover is generating the transcript.
The second outcome is that no error will be raised and the Prover sends the proof. In this case, if the Verifier knew about this issue, then this might leak to him information about the distribution of the secrets involved in the proof (and thus this might affect the zero knowledge proof properties), however further analysis has to be done in order to determine the leakage, which is application specific.

However, we consider this to be a minor issue as in the context of Nimue, the swap function is used in order to move from the scalar field of an elliptic curve to the base field, and typically, the elliptic curves over which we use zero knowledge proofs have a base field larger than the scalar field. Note also that in case when a malicious Prover still crafts a proof corresponding to a case where an error was raised by the library and sends it to the Verifier, the swap function will still raise an error when the Verifier parses the transcript.

### 3.3.1 Fix for finding

We recommend adding a warning to the user in the library, that this function cannot be used when we are moving elements from a field to a smaller one.

## 3.4 Usage of two representations of field elements through bytes

We noticed that the library uses two different representations of field elements, when expressed through bytes, due to the usage of the functions from_le_bytes_mod_order, from_be_bytes_mod_order, and from_bytes_mod_order. Note however that we did not identify any malicious exploitation of this.

### 3.4.1 Fix for finding

We suggest unifying in the whole library the representation of field elements through bytes, by considering either the big-endian system or the little-endian system.

## 3.5 Basic test for randomness

One of the tests in the library ( `test_statistics` ) was intended to perform a basic statistical test, to check if squeezed bytes from a sponge that uses Keccak look random. The test generates `n` bytes through the sponge, then checks if every possible byte value (values from 0 to 255) has occurred on average `n/256` times. This is done by setting an interval `[a,b]` such that `a = n/256 - e` and `b = n/256 + e` , for some value `e` not too far from `n/256` , i.e, an interval centered with a term that each byte value has to appear on average.

However, the interval that was used is actually of the form `[a, b+ another_term]`, leaving more room for the test to pass even if the occurrences of the bytes' values are not in the interval intended.

Note however that the test still passes (and thus the outcome of the test checks out with the fact that these values are random) when fixing the interval to be the right one and increasing the number of samples.

### 3.5.1 Fix for finding

We suggest removing the extra term in the interval, if the code related to the function being tested is modified.

# 4 Conclusion and recommendations

This report provided a code review over the Nimue library. The main issues found (first and second findings) concerned the case where an algebraic hash function is being used for the Public Sponge while the IO pattern operates over bytes. However these issues are fixable, and we have provided solutions to address them. The remaining issues are of minor severity, and we have also proposed suggestions to mitigate them.

# 5 Acknowledgement

We would like to thank Michele for the discussions related to the library.

# 6 Bibliography


[1] *SAFE: Sponge API for Field Elements*. https://eprint.iacr.org/2023/522.pdf. Accessed: 2024-05-28.

[2] *Nimue Documentation*. https://docs.rs/nimue/latest/nimue/index.html. Accessed: 2024-05-28.

[3] *Nimue Library*. https://github.com/arkworks-rs/nimue/tree/main. Accessed: 2024-05-28.

[4] *Merlin library*. https://github.com/dalek-cryptography/merlin. Accessed: 2024-07-08.

[5] *shosha256 implementation of Libsignal*. https://github.com/signalapp/libsignal/blob/main/rust/poksho/src/shosha256.rs. Accessed: 2024-07-08.

[6] *Digest trait*. https://docs.rs/digest/latest/digest/trait.Digest.html. Accessed: 2024-07-08.

[7] *Cryptographic Random Number Generator*. https://crates.io/crates/getrandom. Accessed: 2024-07-08.

[8] *Zkcrypto group crate* . https://github.com/zkcrypto/group. Accessed: 2024-07-08.

[9] *Arkworks algebra crate* . https://github.com/arkworks-rs/algebra. Accessed: 2024-07-08.

[10] *Duplex Sponge*. https://keccak.team/files/SpongeDuplex.pdf. Accessed: 2024-07-08.

# Appendix 1 Testing team

| | |
|---|---|
| Younes Talibi Alaoui | Younes Talibi Alaoui received his PhD in cryptography from KU Leuven, with a focus on privacy-preserving technologies such as Multi-Party Computation. After his PhD, he started working as a cryptographer across companies. |
| Melanie Rieback | Melanie Rieback is a former Asst. Prof. of Computer Science from the VU, who is also the co-founder/CEO of Radically Open Security. |

# Appendix 2 Typos

We list here the typos we spotted in the comments of the code:

1. "Parse the givern IO Pattern into a sequence of [Op]'s".
2. "Users can build the top of it via extension traits".
3. "There are prover just proceeds with concatenation".
4. "They a string encoding the sequence".
5. "easy an easier inspection of the Fiat-Shamir transform".
6. "Basic scatistical".
7. "contains the veirifier state".
8. "Hashes in nume operate over some native".
9. "we this means squeeze".
10. "not hold eny information".
11. "Bindings for some popular libearies using zero-knowledge".
12. "Retrieve field elements from the protocol trainscript".
13. "Every time the prover's sponge is squeeze".
14. "One of the witness".
15. "This operations makes".
16. "some dangers lie is non-trivial for algebraic hashes".
17. "and have other protocol compose the IO pattern.".
18. "/// Bits needed in order to encode an element of F. pub(super) const fn bytes_modp(modulus_bits: u32) ->usize { (modulus_bits as usize + 7) / 8 }" Bits should be Bytes.
19. "https://github.com/dalek-cryptography/arthur". The correct link is : "https://github.com/dalek-cryptography/merlin"
20. " /// Bits needed in order to obtain a (pseudo-random) uniform distribution in F. pub(super) const fn bytes_uniform_modp(modulus_bits: u32) ->usize { (modulus_bits as usize + 128) / 8 }" Bits here should be bytes
21. " /// Get a challenge of `count` elements". This comment is misplaced.