WE INNOVATE TO SECURE YOUR BUSINESS

Code Review of Alice's Ring Signature Implementation

Date	March 13th, 2024
Version	1.1
Page count	92
Authors	Ryad Benadjila Thibauld Feneuil

Contents

1 Introduction

Cypher Lab is a company developing privacy solutions for the Web3 ecosystem. In particular, Cypher Lab develops *Alice's Ring*, a `TypeScript` library implementing a ring signature for the blockchain ecosystem. In this context, Cypher Lab is willing to go through a security audit of their ring signature implementation and has reached CryptoExperts to this purpose.

The present report contains the results of this audit conducted between end of January and beginning of February 2024. We first provide the scope of the audit in Section 1.1 and a summary of the audit methodology and findings in Section 1.2. Then, we present an overview of the code architecture, rationale and APIs in Section 2. Finally, the code review and observations are developed in Section 3.

1.1 Scope of the audit

The code to be reviewed is composed of 22 `TypeScript` source files (3018 lines of code) available at

`https://github.com/Cypher-Laboratory/types-Ring-Signature-audit/tree/main/src`

The version of the code to be reviewed corresponds to the commit `a7da3fe`, from December 26, 2023, available here.

The main external dependencies projects of this code are:

- The `@noble/hashes` library for hashes (`sha3-256` and `sha-512` are used).
- The `@metamask/eth-sig-util` library, used here for encryption and decryption of the partial signature using the `x25519-xsalsa20-poly1305` algorithm for AEAD (Authenticated Encryption with Associated Data).

While their API usage by *Alice's Ring* is in the scope of this audit, these external dependencies source code are **out-of-scope**. `@noble/hashes` has been audited in 2021, and the report is available [12] (although the critical issues reported are marked as resolved, this library is active with many recent commits that have not been audited: the report results should be cross-checked). To the best of our knowledge, the `@metamask/eth-sig-util` library has not been publicly audited.

Parts of the `@noble/secp256k1` and `@noble/ed25519` libraries have been integrated in *Alice's Ring* source code (as per commit 257ba6a and commit d1b2b07 respectively). This code is in the scope of the current audit.

Beyond the mere source code, the inputs of the audit consisted in the relevant documentation pointed by Cypher Lab in order to acquire a good understanding of the ring signature algorithm. This includes the original paper [7], the Monero documentation [6] as well as a technical white paper [5] shared by Cypher Lab[1].

[1] The shared version can be found in this commit

1.2 Methodology and summary of findings

The main goal of this audit was to validate the soundness of *Alice's Ring* implementation of the SAG (Spontaneous Anonymous Group) signatures as described in Section 3.3 of the Monero documentation [6], with the use cases of Cypher Lab for this ring signature in mind. More precisely, this audit aims:

- to validate the proper implementation of the SAG signature scheme in `TypeScript`, i.e. the safety of the exposed API and the code,
- to validate the mathematical soundness of this implementation, namely on the ring signature side but also on the low-level ECC computations (whose abstraction is mostly provided by the `@noble/secp256k1` and `@noble/ed25519` code integration).

The audit methodology consisted in reading the input articles and white paper, and in an in-depth review of the code by two different persons, confronting our understanding of the code and keeping track of our observations.

Our observations are categorized as follows:

- Observations that may impact the security or the soundness of *Alice's Ring* protocol, rated as
  - high risk (flagged ●),
  - medium risk (flagged ●),
  - low risk (flagged ●).
- Observations related to coding practices and implementation choices (flagged ●).
  - These observations do not translate into a direct risk on the security or soundness of *Alice's Ring* protocol, but addressing them would make the code clearer, more efficient and/or less prone to errors.
- Observations related to documentation, comments, variable naming (flagged ■).
  - These observations do not translate into a direct risk on the security or soundness of *Alice's Ring* protocol, but addressing them would facilitate the understanding of the code by third parties (users, developers, auditors).

Each observation comes with an associated recommendation to fix or improve the underlying issue. The observations on a part of code that are duplicates of previous observations on other parts will be marked with the icon: for those neither a detailed description nor a recommendation are directly provided as a reference to the previous observation should be enough to deal with them. Some observations are the result of discussions between CryptoExperts and Cypher Lab providing complementary information that are neither part of the audited code nor the provided documentation, but are still important in *Alice's Ring* context: those will be marked with the icon.

Beyond observations, outstanding remarks (flagged) are also present in the document: these are used to highlight interesting contextual facts.

Update: Following the audit and a first version of the report, Cypher Lab provided feedback as well as patches for most of the Observations in commit b5ef0b9[2]. CryptoExperts performed a check on all the fixes, and the following notations hold:

- When an Observation is checked to be fixed, it is highlighted with the icon: in this case, a dedicated " **Fix from Cypher Lab**" paragraph is provided along with the Recommendation to discuss the fix.
- Some Observations might not be fully fixed or fully checked by CryptoExperts, these will be indicated with the icon and a dedicated " **Feedback from Cypher Lab**" paragraph should be present to provide more contextual information.
- All the other Observations that are not marked with or are considered neither fixed nor discussed.

Our findings are summarized in the table below (the duplicate observations are not accounted), with the corresponding number of fixed Observations checked by CryptoExperts in commit b5ef0b9.

Category	Number of findings	Fixed
High risk	1	1
Medium risk	2	2
Low risk	14	9
Coding practices	43	32
Documentation	17	11
Total	**77**	**55**
Remarks	5	-

The following list of observation is exhaustive and contains the duplicates (marked with : 0 , 1 , 0 , 2 , 1):

[2] It is to be noted that following the audit report update with Cypher Lab feedback, some Observations numbers might have changed in the current report compared to the ones referenced in commits between a7da3fe and b5ef0b9.

2 Solution overview and source code architecture

2.1 SAG ring signatures

A spontaneous anonymous group (SAG) signature scheme allows a person to sign a message on behalf of a group of people (called a ring) without revealing which member of the ring signed the message. In what follows, we describe the SAG signature scheme of Monero [6].

Setting. The scheme parameters are

- a finite field $\mathbb{F}_q$ of size q,
- an elliptic curve C defined over $\mathbb{F}_q$ of order $N = h \times l$, where l is a large prime number and h is the so-called *cofactor* of the curve,
- a point G of the curve C of order prime l, named *generator*,
- a hash function H producing digest binary strings $H(s)$ of length l_H (usually a power of 2) from binary strings of any size s, and a mapping $\mathcal{H}$ taking outputs of H as big endian big integers in $[0, l_H - 1]$ and producing elements in $[0, l-1]$. Given the fact that in the instances we consider we have $l_H \geq l$, for binary strings s of any length we define $\mathcal{H}(s)$ as:
$$\mathcal{H}(s) = H(s) \pmod{l}$$

Key Generation. The key generation algorithm consists in the following:

1. Sample an integer k satisfying $0 < k < l$ uniformly at random;
2. Compute K as $k \cdot G$ (where $\cdot$ is the scalar multiplication over C);
3. Set K as the public key and k as the corresponding private key.

Signing Algorithm. Let m be the message to sign, $\mathcal{R} = \{K_1, K_2, \ldots, K_n\}$ a set of distinct public keys and k_π the signer's private key corresponding to his public key $K_\pi \in \mathcal{R}$, where π is the signer's secret index.

1. Sample uniformly at random a nonce α and $n - 1$ fake responses $\{r_i\}_{i \neq \pi}$ from $\{1, \ldots, l-1\}$.
2. Compute
$$c_{\pi+1} = \mathcal{H}(\mathcal{R}, m, \alpha \cdot G).$$

3. Compute all the other challenges in a circular way

$$
\begin{aligned}
c_{\pi+2} &= \mathcal{H}(\mathcal{R}, m, r_{\pi+1} \cdot G + c_{\pi+1} \cdot K_{\pi+1}) \\
&\vdots \\
c_n &= \mathcal{H}(\mathcal{R}, m, r_{n-1} \cdot G + c_{n-1} \cdot K_{n-1}) \\
c_1 &= \mathcal{H}(\mathcal{R}, m, r_n \cdot G + c_n \cdot K_n) \\
&\vdots \\
c_\pi &= \mathcal{H}(\mathcal{R}, m, r_{\pi-1} \cdot G + c_n \cdot K_{\pi-1}).
\end{aligned}
$$

4. Compute the final response r_π as $r_\pi = \alpha - c_\pi k_\pi \pmod{l}$.

5. Set $(c_1, r_1, \ldots, r_n)$ as the output signature $\sigma(m)$.

It should be noted that the challenges $\{c_i\}_{i\in[1,n]}$ are all in the range $[0, l-1]$ (as the result of applying $\mathcal{H}$).

Verification Algorithm. Let m be the signed message, $\sigma(m)$ is the signature to verify and $\mathcal{R} = \{K_1, K_2, \ldots, K_n\}$ a set of distinct public keys.

1. Parse the signature $\sigma(m)$ as $(c_1, r_1, \ldots, r_n)$.

2. Compute all the challenges in a circular way

$$
\begin{aligned}
c'_2 &= \mathcal{H}(\mathcal{R}, m, r_1 \cdot G + c_1 \cdot K_1) \\
c'_3 &= \mathcal{H}(\mathcal{R}, m, r_2 \cdot G + c'_2 \cdot K_2) \\
&\vdots \\
c'_n &= \mathcal{H}(\mathcal{R}, m, r_{n-1} \cdot G + c'_{n-1} \cdot K_{n-1}) \\
c'_1 &= \mathcal{H}(\mathcal{R}, m, r_n \cdot G + c_n \cdot K_n).
\end{aligned}
$$

3. If $c_1 = c'_1$ then accept the signature, reject it otherwise.

2.2 *Alice's Ring* SAG signature usage in Web3

The rationale behind the usage of SAG signatures in the *Alice's Ring* framework is provided in the technical white paper [5]. The main *leit motiv* is to preserve anonymity in contexts where transactions are signed by users, and where the public keys allows to trace those transactions signed by the same private keys. This is for instance the case in the various existing blockchains: users usually only benefit from so called "pseudonymity" without true anonymity.

While Zero-Knowledge Proofs can be used to enhance privacy in such contexts, the current frameworks such as zk-SNARKs still face challenges as commonly used blockchains, such as Bitcoin and Ethereum, rely on elliptic curves based signatures (e.g. on `secp256k1` for Bitcoin) that are not pairing-friendly. Adopting new (pairing friendly) curves while preserving backward compatibility would bring user friction as the experience would be severely degraded, with adding more layers to trust.

Another technical solution can be found using ring signatures as described in Section 2.1: the idea is to use a set of multiple public keys with only one being the real signing keys. Monero [6] has been the first to use ring signatures in the context of crypto transactions on their blockchain, using a set of decoy keys to anonymize the user. *Alice's Ring* pushes the usage of ring signatures further: beyond mere transactions, these signatures allow proving that any user of a blockchain shares characteristics with other users without revealing his identity (i.e. his public key). An obvious use case is proving to a bank that your balance is 1 BTC: by including in the ring a large set of users with this characteristic, it is possible to sign an anonymity preserving (in the set) assessment.

This "proof of assessment" takes the form of **badges** in the context of *Alice's Ring*: such badges are the assessments signed by another private key hold by the legitimate signer (unrelated to the blockchain one to preserve anonymity), allowing this signer to exhibit a proof of his badge without revealing his identity. The overview of *Alice's Ring* solution components is exposed on Figure 1:

- An indexer continuously grabs public data from various blockchains ◇1. This includes wallet balances, transactions, etc. This data is stored in a database ◇2 that will be accessed later. The database model focuses on providing sets of public keys with the same assessments (e.g. having a current balance of 1 BTC, etc.).
- In the application layer, the *Alice's Ring* mobile or web application retrieves data from the database to forge a ring of public keys sharing the same assessment with the user of the application ◇3.
- The application forges the ring signature using the user's private key, and a badge is generated and signed using the dedicated private key (uncorrelated with the ring signature private key). Metadata linking the badge to its signature proof is stored on the IPFS (InterPlanetary File System) shared storage file system[3] ◇4, and remains accessible whenever the legitimate signer wants to prove the assessment in the badge to a third party. The badge is assigned to an address on the blockchain accessible to the user and that is different from the ring signature one to preserve anonymity.

Let us dive into the detail of the modules that participate to *Alice's Ring* solution, and how they interact in a typical scenario. These modules can be split across three main parts: the back-end of the solution that is made of servers mostly participating to the indexing layer, and the wallet and front-end that are (usually) executed locally on the client side (either mobile application or browser) in the application layer[4].

We present the modules on Figure 2. In the following, we use the same notations as in Section 2.1:

[3] `https://ipfs.tech/`

[4] The wallet can be deported on another infrastructure or device that is not local to the client machine, using remote communication with the font-end (e.g. network, USB or NFC with a dedicated physical device).

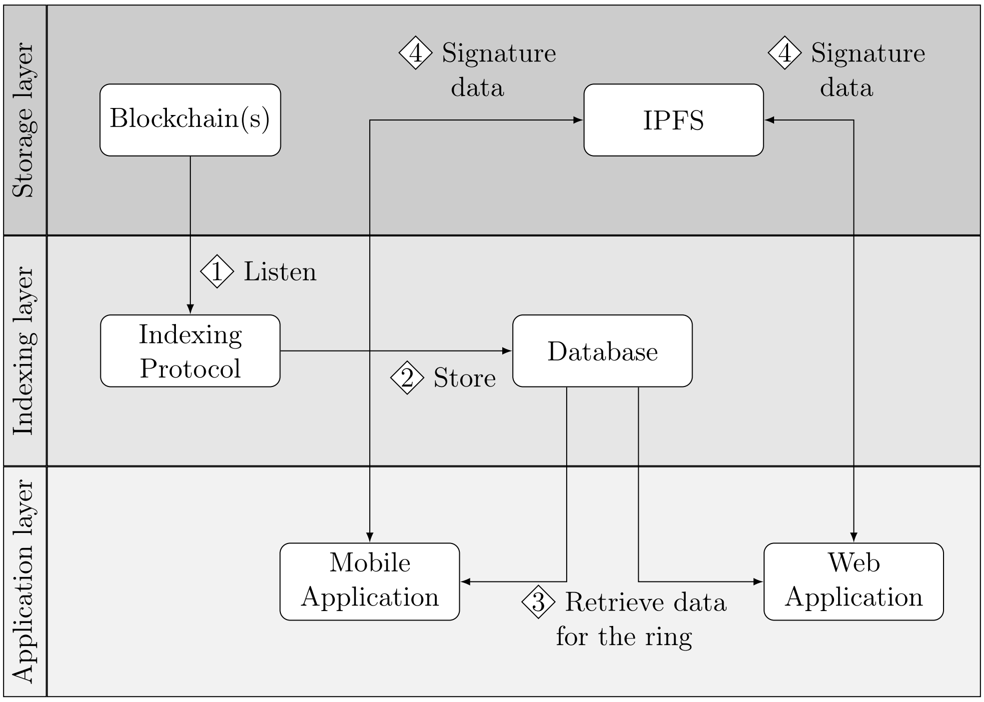

Figure 1: *Alice's Ring* components across the Web3 layers.

- First of all, the front-end initializes a request to prepare a list of some public keys satisfying a specific assessment 1. This request is sent to the back-end servers, while a second request is sent to the wallet where a public `x25519-xsalsa20-poly1305` encryption key is asked (we will denote $K_{\texttt{x25519}}$ the `x25519` public key and $k_{\texttt{x25519}}$ the associated private key).

- The back-end creates and sends a public ring $\hat{\hat{\mathcal{R}}}$ from the assessment using the database where indexed data have been stored. This set $\hat{\hat{\mathcal{R}}}$ should not contain duplicates, but might contain the client's public key: in such case the front-end detects this and removes it from $\hat{\hat{\mathcal{R}}}$ without performing any other modification in this ordered set. In the end, a set of $n-1$ unique public keys $\hat{\mathcal{R}} = \{\hat{K}_1, \hat{K}_2, \ldots, \hat{K}_{n-1}\}$ not containing the client's public key should be available at the front-end level. The responses of requests regarding the same assessment to the back-end are randomized: the back-end sends a random subset of public keys that matches the request[5]. Finally, the wallet sends the public key $K_{\texttt{x25519}}$ in 2.

- Upon reception, the front-end application initiates the ring signature by performing

[5]The details of how the back-end generates the set of public keys from a request, as well as how the front-end deals with the client's public key duplication, are neither part of the audited code nor in the white paper [5]. Since they might be important for *Alice's Ring* security, the elements provided here have been gathered through discussions between CryptoExperts and Cypher Lab.

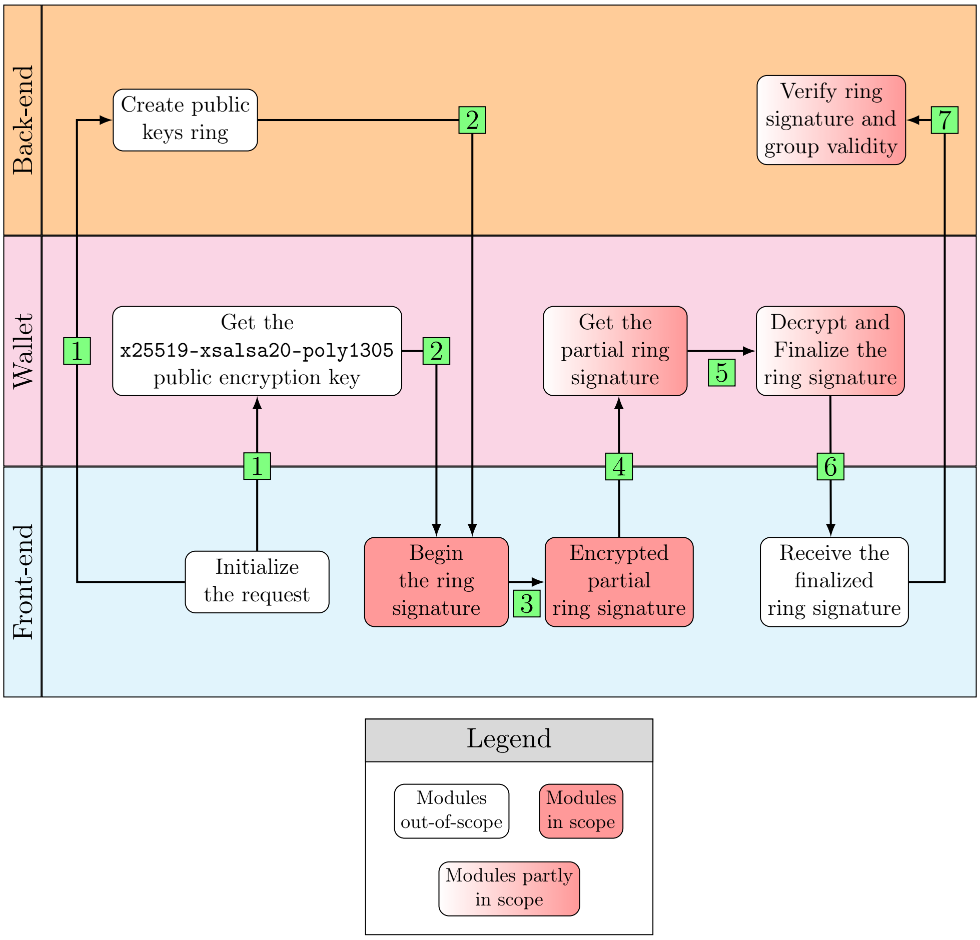

Figure 2: *Alice's Ring* ring signature generation flow. This high level architecture has now changed following Cypher Lab fixes of Observation 1 (●).

the following:

- Generate a random index π for the client public key, and finalize the ring by inserting the client's public key K_π in the ring: $\mathcal{R} = \{K_1, \ldots, K_\pi, \ldots, K_n\}$. The indices larger than π in $\hat{\mathcal{R}}$ are incremented, meaning that:

$$K_i = \begin{cases} \hat{K}_i, & \forall\ 0 \leq i < \pi. \\ \hat{K}_{i-1}, & \forall\ \pi < i \leq n. \end{cases}$$

- Then, $n-1$ random elements $r_1, \ldots, r_{\pi-1}, r_{\pi+1}, \ldots, r_n$ are generated by the front-end as well as a random nonce value α in $]0, l[$, and $c_{\pi+1}$ is computed using the following equation:

$$c_{\pi+1} = \mathcal{H}(\mathcal{R}, H(m), \alpha \cdot G) \tag{1}$$

– Finally, it computes for $i = \pi + 1, \ldots, n, 1, \ldots, \pi - 1$ replacing $n+1$ with 1 (i.e. circular indexing):

$$c_{i+1} = \mathcal{H}(\mathcal{R}, H(m), r_i \cdot G + c_i \cdot K_i) \tag{2}$$

Remark 1: *Alice's Ring* and Monero

It should be highlighted here that *Alice's Ring* corresponds to SAG from Monero as described in Section 2.1 using the message digest $H(m)$ instead of the raw message m when computing the challenges.

The **partial signature** $\hat{\sigma}(m) = (m, \mathcal{R}, c_1, c_\pi, r_1, \ldots, r_{\pi-1}, r_{\pi+1}, \ldots, r_n, \pi, \alpha)$ is then encrypted using the `x25519-xsalsa20-poly1305` public key 3, and the encrypted value $E_{K_{\text{x25519}}}(\hat{\sigma}(m))$ is sent to the wallet 4.

- The wallet achieves to compute the final ring signature $\sigma(m)$ by doing the following:

  – Decrypt the partial signature using the `x25519-xsalsa20-poly1305` private key 5:

$$\begin{aligned}\hat{\sigma}(m) &= D_{k_{\text{x25519}}}(E_{K_{\text{x25519}}}(\hat{\sigma}(m))) \\ &= (m, \mathcal{R}, c_1, c_\pi, r_1, \ldots, r_{\pi-1}, r_{\pi+1}, \ldots, r_n, \pi, \alpha).\end{aligned}$$

  – Compute:

$$r_\pi = \alpha - c_\pi \times k_\pi \pmod{l} \tag{3}$$

  This will ensure that $\alpha \cdot G = (r_\pi + c_\pi \times k_\pi) \cdot G = r_\pi \cdot G + c_\pi \cdot K_\pi$, hence satisfying the circular chaining.

  – Finalize the ring signature $\sigma(m)$ computation by inserting r_π at index π (this is called "combining" the partial signature with the wallet response in *Alice's Ring* implementation):

$$\sigma(m) = (m, \mathcal{R}, c_1, r_1, \ldots, r_{\pi-1}, r_\pi, r_{\pi+1}, \ldots, r_n).$$

  The wallet finally returns $\sigma(m)$ to the front-end application 6.

- The front-end can send the resulting ring signature $\sigma(m)$ to the back-end so that a badge can be emitted 7. The back-end can then verify this signature with:

  – Extract the components $(m, \mathcal{R}, c_1, r_1, \ldots, r_n) = \sigma(m)$, and compute for $i = 1, 2, \ldots, n$ replacing $n+1$ by 1 (i.e. circular indexing):

$$c'_{i+1} = \mathcal{H}(\mathcal{R}, H(m), r_i \cdot G + c_i \cdot K_i) \tag{4}$$

  – Check that $c'_1 = c_1$, accept the signature if this is the case, refuse it otherwise.

It is to be noted that the logic involving the badge signature with the user (badge) private key has not been included in this description, and is out-of-scope as it is orthogonal to the core computation of the ring signature.

The code provided for the audit (see Section 1.1) does not cover all the modules described in the previous ring signature generation and verification scenario. This is materialized on Figure 2 where represent modules that are fully part of the audited code, are only partially implemented, and are not implemented at all in the scope of the audited code. The implementation for the partially in scope and out-of-scope modules is present in other repositories from Cypher Lab's organization (e.g. parts of the wallet logic are implemented in a wallet dedicated project).

In the current report, the focus will be on the modules, and the parts of the modules present in the audited code. A scrutiny of the APIs exposed to out-of-scope code is also performed in order to ensure their safety in the context of *Alice's Ring*.

2.3 TypeScript source code architecture

The source code of the library in `TypeScript` in the `src` folder can be split in 7 categories that can be seen as abstractions dedicated to specific purposes. These are represented on Figure 3:

- The highest level abstraction 1 concerns the signature APIs, i.e. those that are used by the modules described in Section 2.2. The `ringSignature.ts` file contains the main class and methods to perform a ring signature, with possible partial computation without the private key as presented in step 3 of the flow described on Figure 2. The rationale behind partial computation is that the private key should not leave the wallet perimeter, and hence the front-end only computes the ring signature part that does not need this private key. The `signature/piSignature.ts` file contains the method to compute the missing piece with the private key, and a `combine` method in `ringSignature.ts` terminates the full ring signature as performed in step 5. The communication between the front-end and the wallet is encrypted to protect against MitM sniffing as we will precise in the sequel (this is step 4 on Figure 2). The verification logic to check ring signatures is also implemented in `ringSignature.ts`, corresponding to step 7 of Figure 2 performed in the back-end. Finally, this abstraction also contains `signature/schnorrSignature.ts` that is not used in the library, but is exposed for sanity checks (exploiting the fact that a ring signature with a ring of size exactly one coincides with the classical Schnorr signature algorithm).

- The signature abstraction 1 relies on an ECC (Elliptic Curve Cryptography) abstraction 2 that exposes classes and methods for curves and points manipulation (for instance, a ring is an array of `Point`). Only the `secp256k1` and `ed25519` curves are allowed. Methods mainly consist in instantiating curves, dealing with affine points (with operations such as multiplication, addition), checking if they are on the curve, serializing and deserializing these objects, etc. The elliptic curves abstraction rely on the `noble-secp256k1` and `noble-ed25519` libraries 3 for the implementation of everything related to ECC low-level operations: dealing with projective coordinates and the transfer to affine coordinates, scalar multiplication, addition, negation, etc. Modular operation on the fields (prime of the curve and the order) are performed there over `TypeScript BigInt`. The code of `@noble-secp256k1` and `@noble-ed25519` has been taken untouched from the original repositories, and the code for signature operations (ECDSA and Ed25519) is present but not used by *Alice's Ring* (only the points and operations on them are used).

- Signatures also make use of hash functions in the 4 abstraction, exposing an abstract `hash` operation. At low-level, this makes use of the `@noble-hash` library (out-of-scope in this audit).

- Random number generation is used and exposed through 5: it is a wrapper using `node-js node:crypto/randomBytes` internal RNG (Random Number Generator), advertised as a CSPRNG (Cryptographically Secure Random Number Generator)[6]. Random number generation is used in various places of *Alice's Ring*, e.g. when generating the index π, the nonce α, the responses $\{r_i\}_{i\neq\pi}$, etc.

- The partial ring signature makes use of encryption (between the front-end and the wallet, see steps $\boxed{3}$ and $\boxed{4}$ on Figure 2), which uses `x25519-xsalsa20-poly1305` in 6 imported from the external `@metamask/eth-sig-util` library (out-of-scope of the audit). The encryption API is explicitly called from `ringSignature.ts` 1, but the decryption API is not explicitly called as it is part of out-of-scope code from Cypher Lab (code in the wallet implementation).

- Finally, various utility functions (used for serialization, hexadecimal to binary conversion, errors and exceptions handling, public interfaces, etc.) are exposed by 7 and used all over the other abstractions.

Beyond the `src/` folder that contains the source code, a `test/` folder contains various test cases in `TypeScript` for many of the abstractions previously described. Finally, `json` package configuration files are also present in the source tree, allowing to provide *Alice's Ring* configuration and dependencies.

[6] `https://nodejs.org/api/crypto.html`

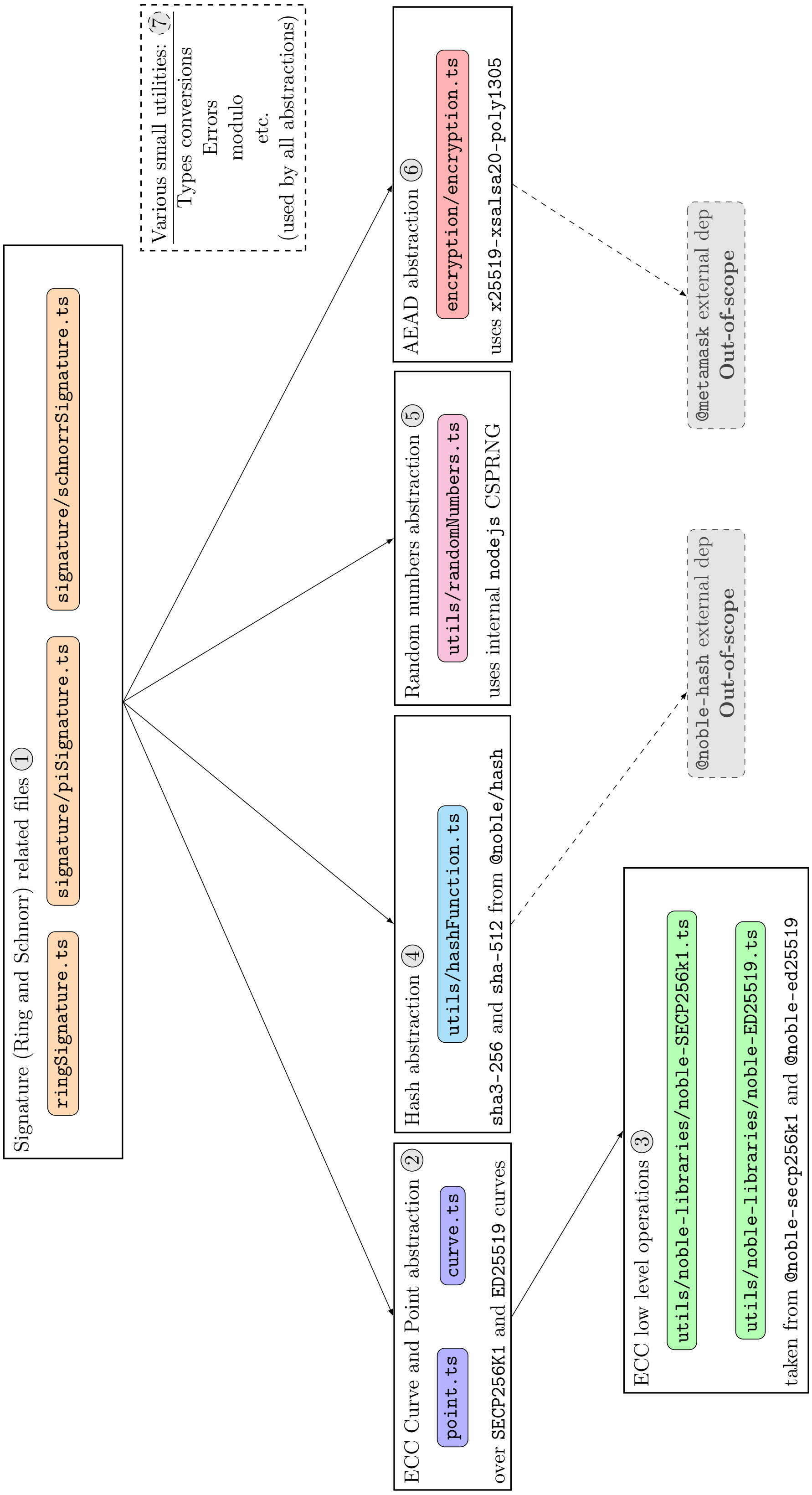

Figure 3: Overview of the TypeScript files of *Alice's Ring* and their interaction.

2.4 Threat and attack model

The purpose of this section is to provide the threat model considered in this audit, hence embracing the attacks that are considered as relevant for *Alice's Ring* typical usage scenario as described on Figure 2. The list of sensitive assets is also provided, with their level of sensitivity (i.e. how critical guessing them would impact the solution).

2.4.1 Considered attacks

The threat model we consider in this audit is the following:

- Attacks on the ring signature correctness: an attacker that does not possess any of the private keys associated to the ring public keys must not be able to forge a signature that is accepted by the verifier.
- Attacks on the implementation interfaces in order to recover sensitive assets (see below for the list of sensitive assets) or bypass the signature verification. The attack surface that can be used by an attacker includes:
  - The exposed API accessible to attackers, i.e. where they can inject corrupted data to gain an advantage.
  - Remote and local timing side-channels where the time taken by a module (either running on a remote machine or the same machine) can be exploited.
  - Local software based side-channels (microarchitectural attacks exploiting cache, branch prediction or speculative execution) on the same machine [8]. An example of this is the scenario where the attacker runs his spying code on the user physical machine (PC or mobile) where the wallet runs in order to guess the private key of the user, or where the front-end runs in order to guess the private index π of the user.

  Software based fault injections, such as Rowhammer [9] (DRAM flipping bits) or CLKscrew [11] (voltage and power frequency management induced faults), are evaluated as too anecdotal and hard to exploit to be considered in the scope of the current audit. However, best practices could be provided to prevent them if they are not too heavy on the implementation.

Beyond mere exploitable attacks, observations on coding practices will also be provided when needed.

2.4.2 List of sensitive assets

The list of the assets considered as sensitive, with the list of modules that legitimately know them as designed by CYPHER LAB, is presented in Table 1.

The rationale behind Table 1 is that the most valuable and critical asset to protect is the user's private key k_π, since leaking it will completely compromise the user's account on the blockchains. This private key is supposed to be kept secret inside the wallet. The

Sensitive assets of ***Alice's Ring*** as designed		
Asset	Sensitivity	Modules knowing the asset
k_π	Critical	Wallet
$k_{\texttt{x25519}}$	High	Wallet
π	High	Front-end, Wallet, Indexer

Table 1: Sensitive assets as interpreted from *Alice's Ring* design.

other assets are related to the ring signature itself, and are considered as a bit less sensitive as compromising their confidentiality would equivalently lead to the discovery of the real signer in the ring: the index π is a direct representation of the signer public key, and $k_{\texttt{x25519}}$ would allow to recover it by sniffing the communication between the front-end and the wallet. The value of π must remain secret, shared between the front-end, the wallet and the indexer: the back-end modules (except the indexer) must not be able to know or guess this value. The indexer should be trusted, since it can recover the signer's index π by comparing the rings associated to the signatures $\mathcal{R}$ with the provided (incomplete) rings $\hat{\mathcal{R}}$. The other exchanged data in the ring signature ($\sigma(m)$, $H(m)$, $\{c_i\}_{i\in[1,n]}$, $\{r_i\}_{i\in[1,n]}$, G, $\{K_i\}_{i\in[1,n]}$) are not sensitive as they are public data.

Although deemed critical, the private assets used for the badge signature (e.g. the other user private key used in *Alice's Ring*) are considered out-of-scope here as the code handling these operations is not part of the audit.

3 Source code review

In this section, we perform a review of the solution and the source code of *Alice's ring*. In Section 3.1, we provide feedback on the general design of the solution as well as elements regarding the white paper [5] review. Then, following the high level architectural abstractions described in Section 2.3, we give a detailed review of each file in the `TypeScript` source code. Some observations might be generic or transversal to many files: in such cases and in order to preserve the report readability, we will explicit them once and make reference to them each time it is relevant.

3.1 Solution design and white paper feedback

3.1.1 Design issues

Nonce α sensitivity A first critical issue can be spotted in the partial signature $\hat{\sigma}(m)$ design: the nonce α is not considered as sensitive as the private key, which is wrong. Indeed, knowing α allows from the publicly known r_π and c_π to recover the private key:

$$k_\pi = (\alpha - r_\pi) \times c_\pi^{-1} \pmod{l}$$

This means that the front-end is able to recover the private key from the wallet using the existing API, which is a serious issue.

Observation 1: The API between the front-end and the wallet leaks the private key

Because the front-end knows (actually fixes) α, it is able to extract the user's private key k_π from the wallet using the public data of the ring signature.

Recommendation:

Modify the code design so that α is randomly drawn inside the wallet, and the front-end does not know anything about it. The partial signature rationale is not very clear (as the ring is completely sent to the wallet, and the wallet finalizes the whole ring signature). It would be safer to perform the complete ring signature inside the wallet (as this one must be modified to be compatible with SAG anyways).

If splitting the signature computation is a desired rationale, the wallet could generate the nonce α and send $\alpha \cdot G$ to the front-end. Then, the front-end could compute the fake responses and the challenges c_1 and c_π. The wallet can thus finalize the signature by computing the last response using the private key and the secret nonce. To implement this solution, several approaches are possible. The wallet could store the nonce in its local memory and retrieve it to finalize the signature, or it could encrypt it using a secret key and send it to the front-end (in that case, the wallet retrieves the nonce by decrypting the ciphertext that the front-end will send back with the request). However, to set up this partial signature rationale, several precautions should be taken:

1. Both the sending of $\alpha \cdot G$ and of the partial signature should be encrypted to avoid the recovery of the index π by sniffing the communication between the front-end and the wallet. Even if $\alpha \cdot G$ does not explicitly contain the signer index, the latter can be deduced using the signature since $\alpha \cdot G$ is equal to $r_\pi \cdot G + c_\pi \cdot K_\pi$.

2. To avoid any nonce reuse, the wallet should integrate security checks to not sign twice a message with the same nonce (and different challenges). A nonce reuse would totally reveal the private key as shown just after the observation.

If there is no real need for partial signature, we highly recommend to implement the entire computation of the ring signatures in the wallet. As explained above, having a secure design with partial signatures is possible, but it drastically increases the attack surface.

Fix from CYPHER LAB:

This has been fixed by CYPHER LAB in commit 5f90a9a: the partial signature related elements have been completely removed, and the wallet now fully performs the ring signature. The `PartialSignature` interface no more exists, and the related methods `partialSign`, `combine`, `partialSigToBase64` and `base64ToPartialSig` have been properly deleted. All the elements related to encryption between the front-end and the wallet (in the file `encryption/encryption.ts`) have been removed as they are not needed anymore: the front-end now sends all the elements to the wallet to perform the full ring signature σ.

The random value α is also trivially susceptible to nonce reuse. For two ring signatures $\sigma_1(m_1)$ and $\sigma_2(m_2)$ under the same private key and using the same nonce, we have:

$$\begin{aligned} \alpha &= r_{1,\pi_1} + c_{1,\pi_1} \times k_\pi \pmod{l} \\ &= r_{2,\pi_2} + c_{2,\pi_2} \times k_\pi \pmod{l} \\ \Rightarrow k_\pi &= (r_{2,\pi_2} - r_{1,\pi_1}) \times (c_{1,\pi_1} - c_{2,\pi_2})^{-1} \pmod{l} \end{aligned}$$

Finally, the nonce α must not leak any of its bits since lattice based attacks could exploit these leaks to recover the private key by solving the hidden number problem [2]: gathering a set of signatures under the same private key and various nonces allows to recover it by solving a system of equations – the number of necessary signatures depends on the leaked bits. All these attack vectors on α must be checked as they can result in the same critical failure of leaking the user's private key.

Random generation of the signer index π. Before launching the signing process, the front-end retrieves a random list of public keys which satisfy the considered assessment. This list is used as an anonymity set (i.e. as a ring) for the SAG signature scheme. The position π of the signer's public key is sampled uniformly at random at the beginning of the signing operation. This randomness aims to obfuscate the signer index π, assuming that the order of the other keys has been chosen randomly. The index π is then used to build the complete ring $\mathcal{R}$ from the incomplete ring $\hat{\mathcal{R}}$ by inserting the signer's public key in the right position.

Observation 2: Random generation of the signer index

The signer's index π is generated uniformly at random.

Recommendation:

We recommend to sort all these keys in a public order (for example, in the lexicographic order). In that case, instead of chosing randomly the position of the signer's public key in the ring, the front-end just needs to insert it in its right position. It reduces the attack surface of the scheme since the algorithm does not rely on secure random generation for π. That has also the advantage to enable people to decrease the cost of sending the rings associated to some signatures in some contexts. For example, if we know that two signatures rely on the same ring, we do not need to send it twice.

Fix from Cypher Lab:

This has been fixed in commit d65fe72: before signing, the scheme appends the signer's public key to the incomplete ring and then sorts all the public keys (the keys are sorted in the ascending order according to their x coordinates). The index π is deduced by finding the position of the signer's key after the sorting.

```
// add the signer public key to the ring
ring = ring.concat([signerPubKey]);

// order ring by x coordinate ascending
ring.sort((a, b) => {
  if (a.x < b.x) return -1;
  if (a.x > b.x) return 1;
  return 0;
});

// set the signer position in the ring from its position in the ordered ring
const signerIndex = ring.findIndex((point) => point.equals(signerPubKey));
```

Key reuse between AEAD and ring signature Although the code handling the `x25519` AEAD private key $k_{\texttt{x25519}}$ origin is not part of the audited repository (it is part of the wallet code), discussions between CryptoExperts and Cypher Lab revealed that the same user private key k_π is reused for the authenticated encryption between the front-end and the wallet: $k_\pi = k_{\texttt{x25519}}$[7]. The reasons are simplicity (since the front-end will use the same user public key) and a similar key format (random 32 bytes values).

Observation 3: Key reuse between AEAD and ring signature –

While no immediate attack can be used to exploit this key reuse, this is dangerous as this mixes two assets with different sensitivity levels (see Table 1): breaking $k_{\texttt{x25519}}$ immediately breaks the user private key k_π. This can be the consequence, for instance, of a side-channel leaking $k_{\texttt{x25519}}$, or a vulnerability in `@metamask/eth-sig-util`, etc.

Recommendation:

Using the same key for different cryptographic purposes is not sound, specifically when the cryptographic assets do not share the same sensitivity. Ideally, two completely different random private keys must be used. A one-way key derivation algorithm could also be used to derive $k_{\texttt{x25519}}$ from k_π if necessary, ensuring that the derivation implementation does not leak k_π. In any case, the front-end will have to get the `x25519` public key (that is now different from K_π) in some way that must also be protected.

Fix from Cypher Lab:

This has been fixed with partial signature removal – see Observation 1 (●) – as no more encryption is needed between the front-end and the wallet.

3.1.2 White paper

The white paper [5] of the *Alice's Ring* protocol aims to describe the *Alice's Ring* technological solution that uses ring signatures for group membership proofs. After an introduction and a short explanation of all the different concepts, the white paper presents the protocol mechanisms. It mainly focuses on the indexer, the ring signature process and the badge issuance. The white paper then studies several use cases.

In practice, the goal of such a white paper dealing with a security technology is that readers can understand the complete solution and be more confident about its security. The current white paper only partially achieves this goal for the following reasons:

[7] This is actually not exactly the case because of the padding presented in Observation 72 (●), but the two values are equivalent.

1. The only part of the *Alice's Ring* protocol which has been formalized is the ring signature process. However, the proposed technology is a complete protocol, so each part of it should be formally described. This includes at least the high-level API of each protocol actors (indexer, wallet, front-end, ...), together with a description of the content of each interaction. The overall security of the protocol cannot be evaluated only by analyzing the ring signature, it is important to understand the context and how this signature scheme is used in the protocol.

2. The white paper only briefly describes the authors' motivations of considering the current design for their technology. For example, the choice and the motivation of considering partial signatures are not documented. Moreover, the considered threat and attack models are not presented.

3. The authors assumed that the readers are familiar with the blockchain technologies. However, to enable a larger adoption of the proposed solution, it would be better to decrease the reader requirements about blockchain knowledge. In the current version of the white paper, the explanations related to the badge issuance could be hard to understand without such a knowledge.

4. There are no technical details about the badges and their signature (e.g. the fields they encapsulate, how the signature private key is handled, etc.). Since badges play a crucial role in the solution, the white paper must shed light on how they are designed in detail.

The scope of this audit is the ring signature process, not the entire *Alice's Ring* protocol. However, we searched to understand how the signature scheme is used. To proceed, we completed the informations of the white paper with the informations provided during discussions between CryptoExperts and Cypher Lab.

Observation 4: Incomplete white paper

The current version of the white paper does not enable the readers to precisely understand the *Alice's Ring* protocol.

Recommendation:

Complete the white paper with the points listed just before the observation.

Feedback from Cypher Lab:

The last version of the white paper reviewed by CryptoExperts brought the following improvements regarding the points listed above:

- Point 1 is partially addressed in section 3.1 of the white paper where more information are provided regarding the indexer and the data flow between the components of *Alice's Ring* protocol. However, more formal descriptions of the APIs (or references to design documents describing them) are still missing.

- Point 2 is not fully addressed: although the partial signature explanation is no more required thanks to the fix in Observation 1 (●), other key concepts related to security such as the threat analysis and attacker model are still not properly formalized.

- Point 3 is addressed in section 3.3 of the white paper by adding more explanations and references about the underlying blockchain technologies used in *Alice's Ring* (EIP-4671 for Soulbound Tokens, explanations about IPFS, etc.).

- Point 4 is addressed in section 3.3 of the white paper by describing the metadata and fields of the badges, and the motivation behind them.

Observation 5: Mistakes in signature description

The SAG signature scheme used in *Alice's Ring* protocol is described in Section 3.2 of the white paper [5]. However, there are several mistakes:

- there is no response r_0 (the reponses' indexes are in $\{1, \ldots, n\}$);

- since both the message and its digest are manipulated in the protocol, precisions about what m represents should be explicit (in this case, notation m in the white paper seems to correspond to the digest $H(m)$ in the notations of Section 2.2);

- the challenge c_{i+1} should be computed as $\mathcal{H}(R, m, [r_iG + c_iK_i])$ in the signing algorithm, instead of $\mathcal{H}(R, m, [r_iG - c_iK_i])$;

- the nonce α should satisfy $r_\pi + c_\pi k \pmod{\ell}$, instead of $r_\pi - c_\pi k \pmod{\ell}$;

- there is no challenge c_0 (the challenges' indexes are in $\{1, \ldots, n\}$);

- the terminology "seed" for c_i is unconventional ("challenge" would be better).

- the computation of c'_i in the verification algorithm is mistaken (error in the signs, inconsistency in the indexes);

- the final checks should be $c'_1 = c_1$.

Recommendation:

Fix all the listed mistakes.

Feedback from Cypher Lab:

The last version of the white paper reviewed by CryptoExperts fixes most the listed point above (except issues in the signature verification description: the "seed" terminology is still used, the message m should be the digest, and there is a small typo in the c_i' notation), but did not update the π handling fix following Observation 2 (●).

3.1.3 The `README.md` file

A **`README.md`** file is provided with the source code. It presents the source code in few words, proposes an example code using the library, lists all the dependencies, describes the implemented ring signature, and partially details the library API.

■ Observation 6: Mistakes in signature description (`README.md`)

The description of the ring signature scheme provided in the **`README.md`** file inherits the mistakes of the description in the white paper, described in Observation 5 (■).

Recommendation:

Fix all the mistakes.

Feedback from Cypher Lab:

The authors have addressed this Observation after [commit b5ef0b9]() but the fix was not audited by CryptoExperts.

■ Observation 7: Incomplete import in example code of `README.md`

The example code of the **`README.md`** file fails when executing it as is, because of the import instruction.

Recommendation:

Replace the current line

```
import { RingSignature } from '@cypherlab/types-ring-signature';
```

by

```
import { RingSignature, Curve, CurveName, Point } from
       '@cypherlab/types-ring-signature';
```

Feedback from Cypher Lab:

The authors have addressed this Observation after commit b5ef0b9 but the fix was not audited by CryptoExperts.

3.2 Review of the signature related files

3.2.1 Ring signature: `ringSignature.ts` and `signature/piSignature.ts`

The `ringSignature.ts` file contains the main functions of the high level logic of *Alice's Ring*. The `RingSignature` class is the main part of this file and represents $\sigma(m)$ from Section 2.1 notations: an overview is presented on Figure 4. It is made of the following `private` fields:

- The message `message` (i.e. m) to be signed that is represented as a `string`.
- The challenge, c_1 from Section 2.1 notations, which is a `TypeScript bigint`. Only c_1 is needed as the other challenges are computed from it.
- The responses, $\{r_i\}_{i\in[1,n]}$ from Section 2.1 notations, which is an array of `TypeScript bigint`.
- The ring $\mathcal{R}$ that is an array of `Point` objects from the point abstraction (see Section 3.3.1 for the definition of this class).
- The curve C that is a `Curve` object from the curve abstraction (see Section 3.3.2 for the definition of this class).
- An optional configuration `config` of type `SignatureConfig` (more details on this hereafter).

Objects are also exported through the `TypeScript` interfaces in the `interfaces/` folder:

- The `PartialSignature`, corresponding to $\hat{\sigma}(m)$ using Section 2.2 notations, is defined as an interface in `interfaces/partialSignature.ts`:

```
/**
 * Partial ring signature interface
 *
 * @see message - Clear message
 * @see ring - Ring of public keys
 * @see pi - The signer index
 * @see c - The first c value
 * @see cpi - The c value of the signer
 * @see alpha - The alpha value
 * @see responses - The generated responses
 * @see curve - The elliptic curve to use
 * @see config - The config params to use (optional)
 */
export interface PartialSignature {
  message: string;
  ring: Point[];
  pi: number;
  c: bigint;
  cpi: bigint;
  alpha: bigint;
  responses: bigint[];
  curve: Curve;
  config?: SignatureConfig;
}
```

ringsignature.ts
```
class RingSignature {
    -message:  string;
    -c:  bigint;
    -responses:  bigint[];
    -ring:  Point[];
    -curve:  Curve;
    -config?:  SignatureConfig;
    ------------------
    constructor
    accessors (ring, challenge, responses, curve,
    config, message, messageDigest)
    (from/to)JsonString
    (from/to)Base64
    sign
    partialSign
    combine
    verify
    signature
    computeC
    partialSigToBase64
    base64ToPartialSig
}
checkRing
checkpoint
```

Figure 4: `ringSignature.ts` methods and functions.

As we can see, it is made of the message m, the ring $\mathcal{R}$, the index π, the first challenge c_1, the challenge at the index c_π, the nonce α, the curve C, and an optional configuration `config` of type `SignatureConfig`.

- This configuration type is defined in `interfaces/signatureConfig.ts`:

```
/**
 * Signature config interface
 *
 * @see derivationConfig - The config to use for the key derivation
 * @see evmCompatibility - If true, the signature will be compatible with our
       EVM verifier contract
 * @see safeMode - If true, check if all the points are on the same curve
 * @see hash - The hash function to use (input: string, output: Uint8Array)
 */
export interface SignatureConfig {
  evmCompatibility?: boolean;
  safeMode?: boolean;
  hash?: hashFunction;
}
```

Observation 8: Discrepancy in docstring fields for `derivationConfig`

The docstring comment of `SignatureConfig` describes `derivationConfig`, but the field is missing.

Recommendation:

Remove the field from the docstring comment.

Fix from Cypher Lab:

This has been fixed in commit 683dfa8.

Observation 9: Some configuration fields are not used

The `evmCompatibility` field is a boolean expressing the compatibility of the ring signature with Cypher Lab's EVM contract. Although it is parsed by the `fromJsonString` method of `RingSignature`, it is not used in the audited code.

This is also the case for `safeMode` that is parsed and sanity checked to be a boolean, but the parsed value is never used.

Recommendation:

Explicitly use these values, remove them or explain their usage in other places. If the `safeMode` corresponds to a security level (e.g. dealing with ECC operations for private and public operations), explicitly describe it.

Fix from Cypher Lab:

This has been fixed in commit 683dfa8 and commit 5024d6b.

We will now review all the methods and functions related to the ring signature in `ringSignature.ts`.

The constructor of the `RingSignature` class This is the main constructor of the class, taking parameters as input and instantiating an object containing the ring signature.

Observation 10: Bad docstring comment in `RingSignature` constructor

The comment for `constructor` is the following:

```
/**
 * Ring signature class constructor
 *
 * @param message - Clear message to sign
 * @param ring - Ring of public keys
 * @param cees - c values
 * @param responses - Responses for each public key in the ring
 * @param curve - Curve used for the signature
 * @param safeMode - If true, check if all the points are on the same curve
 * @param config - The config params to use (optional)
 */
```

First, the `cees` should be a single `c` since only one `bigint` challenge c_1 is provided. Secondly, the `safeMode` parameter is not in the arguments of the constructor, but rather a field of the optional `config` configuration object.

Recommendation:

Fix the docstring comment appropriately.

Fix from CYPHER LAB:

This should be fixed in commit 26809f0.

The constructor performs some sanity checks, throwing exceptions if there is an issue. The input ring of public keys $\hat{\mathcal{R}}$ is checked to be not empty, and the message is checked to be of size non-zero. The responses length is checked to be equal to the ring length (i.e. equal to n). If the input optional configuration is present, its type is checked to be a valid `TypeScript` object.

Then, the constructor calls `checkRing` for sanity checks on the ring (see below for the details on this function).

Observation 11: Empty message not accepted

The empty message `""` is not allowed, but there is no obvious reason to filter it.

```
...
    if (!message || message === "") throw err.noEmptyMsg;
...
```

Recommendation:

Accept the empty message or provide argument why it is filtered out.

Fix from CYPHER LAB:

This should be fixed in commit c3af520.

Observation 12: Challenge $c_1 = 0$ not accepted, challenge $c_1 \geq l$ accepted

The challenge value $c_1 = 0$ is not accepted as we can see below:

```
...
    if (c === 0n) throw err.invalidParams("c");
...
```

c_1 is the result of a hash function (reduced modulo l), and with very small probability can be equal to 0. This check seems to be actually an artifact related to the avoidance of the point at infinity in all the operations in *Alice's Ring* high level layers: we will come back to this issue in the section dedicated to the ECC abstraction Section 3.3, but the point at infinity can appear in various ways and must be properly handled.

Also, c_1 is not checked to be reduced modulo l (while the responses are checked): this check must be performed.

Recommendation:

Accept possible $c_1 = 0$ and properly handle the point at infinity in the ECC high level abstraction. Check for reduced $c_1 < l$.

Fix from CYPHER LAB:

This should be fixed in commit 0df4498 by adding the proper check $c_1 < l$ and removing the check $c_1 = 0$.

Observation 13: Bad arguments for `checkRing` in `RingSignature` constructor

When calling `checkRing`, the empty ring is allowed while empty ring explicitly throws an exception in `constructor`:

```
...
    if (ring.length === 0) throw err.noEmptyRing;
...
    // check ring, c and responses validity
    checkRing(ring, curve, true);
```

Recommendation:

Call `checkRing` with `false` as a third argument.

Fix from Cypher Lab:

This should be fixed in commit 9228160.

The accessors of the `RingSignature` class Many accessors to the private fields of `RingSignature` are implemented: `getRing()`, `getChallenge()`, `getResponses()`, `getCurve()`, `getConfig()`, `getMessage()`. These accessors are straightforward.

Observation 14: Bad docstring comment for `getRing()`

The `getRing()` accessor has a bad comment:

```
  /**
   * Get the message
   *
   * @returns The message
   */
  getRing(): Point[] {
    return this.ring;
  }
```

Recommendation:

Fix the docstring comment to:

```
  /**
   * Get the ring
   *
   * @returns The ring
   */
  getRing(): Point[] {
    return this.ring;
  }
```

Fix from Cypher Lab:

This should be fixed in commit 5892996.

Another accessor like function is `messageDigest()` that is defined as a `getter`. This computes the hash of the message field using the hash function H from the configuration, and transforms the resulting string $H(m)$ to a returned big endian `bigint`.

Observation 15: Unused getter `messageDigest()`

The `messageDigest()` getter is not used, while some places could use it such as in the non-static `verify` method:

```
    // hash the message
    const messageDigest = BigInt("0x" + hash(this.message, this.config?.hash));
```

Recommendation:

Use the `messageDigest()` getter instead of duplicating it where necessary.

Fix from Cypher Lab:

This should be fixed in commit a3ea484.

The (to/from)JsonString methods The `RingSignature` class has two dedicated methods for exporting and importing it to and from a `json` string. The `toJsonString` is straightforward as the current object is supposed to be inherently well-formed.

The `fromJsonString` on the other hand performs sanity checks as the input might be corrupted: the string is parsed using `JSON.parse` if it is a `string` or considered to be directly a parsed `json` object. Then each field is checked against the expected types, throwing an error if anything is wrong.

Observation 16: Incomplete sanity checks in `fromJsonString`

For `parsedJson.message` and `parsedJson.c` the objects are checked to be instances of `Array` or `object`, and an error is thrown if this is the case since they are expected to be of type `string` or `bigint`:

```
...
    // check if message is stored as array or object. If so, throw an error
    if (
```

```
      parsedJson.message instanceof Array ||
      typeof parsedJson.message === "object"
    )
      throw err.invalidJson("Message must be a string or a number");
    // check if c is stored as array or object. If so, throw an error
    if (parsedJson.c instanceof Array || typeof parsedJson.c === "object")
      throw err.invalidJson("c must be a string or a number");
...
```

This check is insufficient. For example, boolean values pass the check while it should not. Instead of checking a black list of types, a white list (of `string` and `bigint`) should be used.

Recommendation:

Use white list checks for `parsedJson.message` and `parsedJson.c`.

Fix from Cypher Lab:

This should be fixed in commit ffba716.

The (to/from)Base64 methods These methods import or export a `RingSignature` from and to a `base64` string (encoding a `json` string). The `toBase64` is straightforward and directly calls the `toJsonString` with a conversion to `base64`. The `fromBase64` performs some `json` decoding, but misses sanity checks.

Observation 17: Missing sanity checks in `fromBase64`

The `fromBase64` method does not perform the necessary sanity checks, while it should do as the `fromJsonString` does (actually it should internally use it after decoding `base64`.).

```
  /**
   * Transforms a Base64 string to a ring signature
   *
   * @param base64 - The base64 encoded signature
   *
   * @returns The ring signature
   */
  static fromBase64(base64: string): RingSignature {
    // check if the base64 string is valid
    if (!base64Regex.test(base64)) throw err.invalidBase64();

    const decoded = Buffer.from(base64, "base64").toString("ascii");
    const json = JSON.parse(decoded);
    const ring = json.ring.map((point: string) => Point.fromString(point));

    return new RingSignature(
```

```
      json.message,
      ring,
      BigInt(json.c),
      json.responses.map((response: string) => BigInt(response)),
      Curve.fromString(json.curve),
      json.config,
    );
  }
```

Recommendation:

Use the `fromJsonString` that includes proper sanity checks in `fromBase64`.

Feedback from Cypher Lab:

The authors have addressed this Observation after commit b5ef0b9 but the fix was not audited by CryptoExperts.

The `static sign` method This is a `sign` method of the class performing a full (i.e. not partial) ring signature: it takes the incomplete ring $\hat{\mathcal{R}} = \{\hat{K}_1, \ldots, \hat{K}_{n-1}\}$ as input, the signer private key k_π, the message m, the curve C, and an optional configuration `config`. It outputs an instance of the `RingSignature` class representing $\sigma(m)$. It internally uses two `static` and `private` methods (of the same `RingSignature` class) `computeC` and `signature`, and the external function `piSignature` from `signature/piSignature.ts`.

This method performs the following:

- Some sanity checks on the input, and compute the message digest $H(m)$. Empty input rings are accepted by the function.
- Generate a random nonce $\alpha \in [1, l-1]$ using `randomBigint` (see Section 3.6), as well as a random index $\pi \in [0, n-1]$ using `getRandomSecuredNumber` (see Section 3.6).
- Derive the signer public key $K_\pi = k_\pi \cdot G$ using the `derivePubKey` function from the curves abstraction (see Section 3.3.1).
- Introduce K_π in the incomplete ring $\hat{\mathcal{R}}$ at index π to produce the final ring $\mathcal{R} = \{K_1, \ldots, K_\pi, \ldots, K_n\}$ (this is performed by slicing and concatenating `TypeScript` arrays).
- Compute $c_{\pi+1}$ using Equation 1 by calling the `computeC` private static method. This method takes as inputs the ring $\mathcal{R}$, the message digest $H(m)$, the nonce α, the curve C and the optional `config` to produce a `bigint` representing $c_{\pi+1}$: it uses the ECC abstraction to compute the scalar multiplication $\alpha \cdot G$. Actually `computeC` is versatile and is used to compute other c_i depending on the context: we will describe it in more details later.

- Compute the incomplete ring signature with almost everything but the missing response r_π from the signer: $\sigma'(m) = (\mathcal{R}, c_1, \ldots, c_n, \pi, r_1, \ldots, r_{\pi-1}, r_{\pi+1}, \ldots, r_n)$. The `private` and `static` method `signature` is used to produce this.
- Compute the signer response r_π from α, c_π extracted from $\sigma'(m)$, and the private key k_π with Equation 3 by calling the `piSignature` function.
- Finally, produce and return the ring signature $\sigma(m) = (m, \mathcal{R}, c_1, r_1, \ldots, r_n)$ with all the previously computed elements.

Observation 18: Empty message – Observation 11

Duplicate of Observation 11, please refer to it.

Fix from Cypher Lab:

Same fix as in Observation 11.

Observation 19: Bad comment in `sign`

The following comment `ring.length = n+1` is wrong:

```
...
  static sign(
    ring: Point[], // ring.length = n+1
...
```

Recommendation:

Fix the comment.

Fix from Cypher Lab:

This should be fixed in commit 70b5160.

Observation 20: Missing check on `signerPrivateKey` k_π in `sign`

The `sign` function checks that $k_\pi > 0$:

```
...
```

```
    if (signerPrivateKey === 0n)
      throw err.invalidParams("Signer private key cannot be 0");
...
```

However it does not check that $k_\pi < l$, the prime order of the generator G. This has no practical security impact as the provided k_π is reduced modulo l in the ECC abstraction scalar multiplication when calling `derivePubKey`, but this could bring API confusion on values $k_\pi \geq l$.

Recommendation:

Add the proper check on k_π.

Fix from Cypher Lab:

This should be fixed in commit 37d6747.

Observation 21: Perform input sanity checks before any computation

The message digest is computed before the sanity check on the input ring. As a general rule, one should avoid to perform any computation before performing all the (doable) sanity checks in order to limit useless computations:

```
...
    const messageDigest = BigInt("0x" + hash(message, config?.hash));

    // check if ring is valid
    try {
      checkRing(ring, curve, true);
    } catch (e) {
      throw err.invalidRing(e as string);
...
```

Recommendation:

Compute the hash after checking the input ring. As a general rule, perform all the possible sanity checks before doing any computation.

Fix from Cypher Lab:

This should be fixed in commit 32a7d9c.

Observation 22: Partial leak of the signer index

The signer index π is chosen as follows:

```
    // set the signer position in the ring
    const signerIndex = // pi
      ring.length === 0 ? 0 : getRandomSecuredNumber(0, ring.length - 1);
...
    // add the signer public key to the ring
    ring = ring
      .slice(0, signerIndex)
      .concat([signerPubKey], ring.slice(signerIndex)) as Point[];
```

It implies that the signer index cannot be n, where n is the size of the (complete) ring. Indeed, the maximal value for π is `ring.length - 1`, which corresponds to $n - 1^{\text{th}}$ position since the `ring` is *incomplete* when choosing the signer index. So, it leaks information about the signer index (the attacker knows that the last public key does not correpond to the signer key).

Recommendation:

Compute the signer index as

```
    const signerIndex = getRandomSecuredNumber(0, ring.length);
```

Fix from Cypher Lab:

As recommended in Observation 2 (●), Cypher Lab no longer samples the signer's index π randomly, but derives it from the sorted list of public keys of the complete ring. Therefore, this observation is not relevant anymore.

Remark 2: Local side-channel leakage of π

No particular effort has been spent to prevent π leakage across the `ringSignature` layer: most of the algorithms do not mask its usage and are probably leaking some information about it that could be gathered from local side-channels perspective (cache, etc.), potentially allowing attackers to recover it partially or fully.

This is not rated as an observation since this leakage is considered as too “small” to be exploited in practice (especially in the context of `TypeScript`), however this potential leakage should be kept in mind for future improvements of the *Alice's Ring* library implementation.

Here is a non-exhaustive list of potential leakage sources of π:

- When slicing the ring to insert the signer's public key, the `slice` and `concat` methods are not constant time and can leak π through the size of the two manipulated sets.
- In the loop handling the $\{c_i\}_{i\in[1,n]}$ computation in `signature`, the indexing of the `cees` table depending on the index could leak the position π:

```
...
if (index === (signerIndex + 1) % ring.length)
  cees[index] = ceePiPlusOne; // params = { alpha: alpha };
else {
...
  // compute the c value
  cees[index] = RingSignature.computeC(
...
```

The static `partialSign` method The `partialSign` method computes and returns the partial signature $\hat{\sigma}(m) = (m, \mathcal{R}, c_1, c_\pi, r_1, \dots, r_{\pi-1}, r_{\pi+1}, \dots, r_n, \pi, \alpha)$. We will not detail this method as it is the same as the `sign` previously described, except that the response r_π is not computed as only the partial signature is returned. This method is expected to be used in the front-end to produce the encrypted partial signature to be sent to the wallet.

Observation 23: Code redundancy between `sign` and `partialSign`

The `sign` function duplicates the logics of `partialSign`: from a coding perspective, code factorization should be used. The main issue being that `partialSign` returns encrypted data, an argument for optional encryption should be used for code factorization.

Recommendation:

Adapt `partialSign` to optionally return unencrypted data and call it in `sign` instead of code duplication.

Fix from Cypher Lab:

Fixed by partial signature removal: see Observation 1 (●).

Observation 24: Missing sanity checks in `partialSign`

The sanity checks on the inputs (ring and so on) are not performed, contrary to what is done in the `sign` signature. Since `partialSign` is not `private`, it should perform sanity checks.

Recommendation:

Perform sanity checks on the inputs.

Fix from Cypher Lab:

Fixed by partial signature removal: see Observation 1 (●).

■ Observation 25: Missing arguments in docstring comment of `partialSign`

The docstring comment for `partialSign` is missing the `curve` and `encryptionPubKey` arguments.

```
/**
 * Sign a message using ring signatures
 * Allow the user to use its private key from an external software (external
     software/hardware wallet)
 *
 * @param ring - Ring of public keys (does not contain the signer public key)
 * @param message - Clear message to sign
 * @param signerPubKey - Public key of the signer
 * @param config - The config params to use
 *
 * @returns An encrypted PartialSignature
 */
static partialSign(
  ring: Point[], // ring.length = n + 1
  message: string,
  signerPubKey: Point,
  curve: Curve,
  encryptionPubKey: string,
  config?: SignatureConfig,
): EthEncryptedData {
```

Recommendation:

Add the missing parameters in the docstring comment of `partialSign`.

Fix from Cypher Lab:

Fixed by partial signature removal: see Observation 1 (●).

Observation 26: Bad comment in `partialSign` – Observation 19

Duplicate of Observation 19, please refer to it:

```
static partialSign(
  ring: Point[], // ring.length = n + 1
...
```

Fix from Cypher Lab:

Fixed by partial signature removal: see Observation 1.

Observation 27: Partial leak of the signer index – Observation 22

Duplicate of Observation 22, please refer to it.

Fix from Cypher Lab:

Fixed by partial signature removal: see Observation 1.

The static method `combine` This method takes as inputs a partial signature $\hat{\sigma}(m) = (m, \mathcal{R}, c_1, c_\pi, r_1, \ldots, r_{\pi-1}, r_{\pi+1}, \ldots, r_n, \pi, \alpha)$ and a signer response, i.e. r_π (computed in the wallet), and combines them to produce the final ring signature $\sigma(m)$. After some sanity checks on the inputs, the function uses the `RingSignature constructor` to instantiate the object. The `combine` method is used in the wallet to finalize the partial signature after the computation of the response involving the private key.

Observation 28: Missing check on r_π in `combine`

The `combine` method does not check that $r_\pi < l$ (i.e. reduced modulo l), while it should be as the result of Equation 3.

As a side note, the other sanity checks on the parameters are performed in the `constructor` of `RingSignature`.

Recommendation:

Add the sanity check on r_π.

Fix from Cypher Lab:

Fixed by partial signature removal: see Observation 1 (●).

● Observation 29: Code redundancy between `sign` and `combine`

The `sign` function could call `combine` in the final return value for code factorization (after calling `partialSign` as proposed in Observation 23 (●)).

Recommendation:

Use `combine` in `sign`.

Fix from Cypher Lab:

Fixed by partial signature removal: see Observation 1 (●).

The `verify` method This method has as only argument the current `RingSignature` object instance representing $\sigma(m) = (m, \mathcal{R}, c_1, r_1, \ldots, r_n)$. Here is the sketch of the function:

- Perform some sanity checks on the elements of $\sigma(m)$ (no empty ring, check the number of responses against the ring size), and compute the message hash $H(m)$.
- Compute the $\{c'_i\}_{i\in[1,n]}$ (replacing $n+1$ by 1) following Equation 4, using the `computeC` method.
- Check if $c'_1 = c_1$ and return `true` if this is the case, `false` if not.

● Observation 30: Redundant sanity checks in `verify`

The `verify` method checks that the ring is not empty and that the number of responses is consistent with the ring size:

```
if (this.ring.length === 0) throw err.noEmptyRing;

if (this.ring.length !== this.responses.length) {
  throw err.lengthMismatch("ring", "responses");
}
```

However, these checks are redundant since they are performed in the `RingSignature` constructor.

Recommendation:

Remove these sanity checks.

Fix from Cypher Lab:

This should be fixed in commit e575a09.

Observation 31: Simplify the code of `verify`

The code and loop computing $\{c'_i\}_{i\in[1,n]}$ can be simplified:

```
...
    // computes the cees
    let lastComputedCp = RingSignature.computeC(
      // c1'
      this.ring,
      messageDigest,
      {
        previousR: this.responses[0],
        previousC: this.c,
        previousPubKey: this.ring[0],
      },
      this.curve,
      this.config,
    );

    // compute the c values: c2', c3', ..., cn', c0'
    for (let i = 1; i < this.ring.length; i++) {
      lastComputedCp = RingSignature.computeC(
        this.ring,
        messageDigest,
        {
          previousR: this.responses[i],
          previousC: lastComputedCp,
          previousPubKey: this.ring[i],
        },
        this.curve,
        this.config,
      );
    }
...
```

Recommendation:

Simplify this code with the more readable:

```
...
    // NOTE: the loop has at least one iteration since the ring
    // is ensured to be not empty
    lastComputedCp = this.c;
    // compute the c values: c1', c2', ..., cn', c0'
    for (let i = 0; i < this.ring.length; i++) {
      lastComputedCp = RingSignature.computeC(
        this.ring,
        messageDigest,
        {
          previousR: this.responses[i],
          previousC: lastComputedCp,
          previousPubKey: this.ring[i],
        },
        this.curve,
        this.config,
      );
    }
...
```

Fix from CYPHER LAB:

This should be fixed in commit 00a952d.

Observation 32: Check that the public keys are not of low order or hybrid

For curves having a cofactor $h \neq 1$, so-called small subgroup confinement attacks exist: all the points of the protocol must be checked to be in the subgroup generated by G, and hence not of low order and not hybrid. See [4] for a thorough discussion on the implications of cofactor. For curves with $h = 1$, only the point at infinity is an issue.

In the case of *Alice's Ring*, low order concerns `ed25519` and would allow to forge a fake signature without knowing a private key with high probability if no check is performed.

For the sake of simplicity, let us consider a ring signature with one public key (which actually boils down to a simple Schnorr signature). In order to pass the verification test, one must find a way to accept:

$$c_1 = \mathcal{H}(\mathcal{R}, H(m), r_1 \cdot G + c_1 \cdot K_1). \tag{5}$$

Let the attacker take a point $\widetilde{K_1}$ on a curve with $h \neq 1$ in the small subgroup of order h, meaning that $h \cdot \widetilde{K_1} = \mathcal{O}$ (where $\mathcal{O}$ is the point at infinity). This means that the attacker will be able to satisfy Equation 5 with a probability $\frac{1}{h}$ for random c_1 (corresponding to c_1 being multiples of h). Hence, by carefully choosing $\widetilde{r_1}$ so that $\widetilde{c_1} = \mathcal{H}(\mathcal{R}, H(m), \widetilde{r_1} \cdot G)$ is a multiple of h, the attacker can satisfy:

$$\begin{aligned}\widetilde{c_1} &= \mathcal{H}(\mathcal{R}, H(m), \widetilde{r_1} \cdot G + \widetilde{c_1} \cdot \widetilde{K_1}) \\ &= \mathcal{H}(\mathcal{R}, H(m), \widetilde{r_1} \cdot G + \mathcal{O}) \\ &= \mathcal{H}(\mathcal{R}, H(m), \widetilde{r_1} \cdot G).\end{aligned}$$

The signature $\widetilde{\sigma}(m) = (m, \{\widetilde{K_1}\}, \widetilde{r_1}, \widetilde{c_1})$ successfully passes the verification algorithm. For the `ed25519` curve where $h = 8$, a successful forged signature (with no associated private key) will be found with roughly a $\frac{1}{8}$ test rate.

The risk is rated as ● because in the context of *Alice's Ring*, other elements prevent such an attack (although these safeguards **are not sound**, as discussed in Section 3.3).

Recommendation:

Add the check that the public keys points are not of low order or hybrid, ideally through a helper in the `Point` abstraction. A simple way to check this for any point P is to check that its order is l: $l \cdot P = \mathcal{O}$ and $P \neq \mathcal{O}$. Since scalar multiplication by l is not allowed in `noble`, an equivalent test could consist in checking that $(l-1) \cdot P = -P$. Checking low order and hybrid points should only be performed in `verify` as it is the only function where attackers can inject bad points.

Feedback from CYPHER LAB:

The authors have addressed this Observation after commit b5ef0b9 but the fix was not audited by CRYPTOEXPERTS.

A `static` variant of the `verify` method is also implemented: it takes as argument a `string` representing either a `json` encoding of a ring $\mathcal{R}$, or a `base64` encoding of this `json`. The `fromJsonString` is used to decode and instantiate a `RingSignature` object, and then the non `static` `verify` method is called on this object.

Observation 33: Use `fromBase64` in `verify`

The following code can make use of the existing `fromBase64`:

```
...
    // check if the signature is a json or a base64 string
    if (base64Regex.test(signature)) {
      signature = Buffer.from(signature, "base64").toString("ascii");
    }
...
```

Recommendation:

Use `fromBase64` for the `base64` conversion to avoid code duplication.

Fix from Cypher Lab:

This should be fixed in commit 786579a.

The `static` and `private signature` method This `signature` method computes an incomplete ring signature with almost everything but the missing response r_π from the signer: $\sigma'(m) = (\mathcal{R}, c_1, \ldots, c_n, \pi, r_1, \ldots, r_{\pi-1}, r_{\pi+1}, \ldots, r_n)$. It takes as inputs the curve C, the ring including the user public key $\mathcal{R}$, the challenge $c_{\pi+1}$, the index π, the message digest $H(m)$ and the optional configuration `config`. Here is the sketch of this function:

- Some sanity checks on the inputs are performed: no empty message and no zero value message digest, the ring length, the ring validity.
- The $\{r_i\}_{i\in[1,n]}$ values are randomly generated using `randomBigint` (see Section 3.6).
- The $\{c_i\}_{i\in[1,n]}$ are computed using the circular wrap up from index π, following Equation 1.
- All the elements are computed to produce $\sigma'(m)$ that is returned.

Observation 34: Bad check in `signature`

The following check is useless:

```
...
    // check ring and responses validity
    if (ring.length !== ring.length)
      throw err.lengthMismatch("ring", "responses");
...
```

It seems that this should be a check of `ring.length` against `responses.length`, but this is also useless as `responses` is initialized like this from the ring length:

```
...
    // generate random responses for every public key in the ring
    const responses: bigint[] = [];
    for (let i = 0; i < ring.length; i++) {
      responses.push(randomBigint(curve.N));
    }
...
```

Recommendation:

Remove the useless check.

Fix from Cypher Lab:

This should be fixed in commit 76115fd.

Observation 35: Bad checks on the input message digest $H(m)$

The following check is not really sound:

```
...
    if (
      messageDigest === 0n ||
      messageDigest === BigInt("0x" + hash("", config?.hash))
    )
      throw err.noEmptyMsg;
...
```

The first predicate checks for $H(m) \neq 0$, which could occur with very low probability. The second predicate is for empty messages, i.e. $H(m) \neq H("")$, related to Observation 11 (●).

Recommendation:

Remove these checks or provide arguments of their presence.

Fix from Cypher Lab:

This should be fixed in commit c3af520.

Observation 36: Bad arguments for `checkRing` in signature method

When calling `checkRing`, the empty ring is allowed in the context where the ring should contain at least the signer key.

```
    // check if ring is valid
    try {
      checkRing(ring, curve, true);
```

```
    } catch (e) {
      throw err.invalidRing(e as string);
    }
```

Recommendation:

Call `checkRing` with `false` as a third argument.

Fix from Cypher Lab:

This should be fixed in commit 9228160.

Observation 37: Bad docstring comment for `signature`

The comment for `signature` states that the message should be given as input, while it is the message digest in the function argument.

Recommendation:

Fix the docstring comment appropriately.

Fix from Cypher Lab:

This should be fixed in commit 6f57d84.

The `static` and `private` `computeC` method This `computeC` method is used to compute the challenges $\{c_i\}_{i\in[1,n]}$ for the two different contexts where either $i = \pi+1$ and Equation 1 is used, or $i \neq \pi + 1$ and Equation 2 is used.

In order to deal with these two contexts, `computeC` takes as input an optional parameter `params` that either contains α for Equation 1, or contains $(r_{i-1}, c_{i-1}, K_{i-1})$ for Equation 2. The other inputs of the method are the ring $\mathcal{R}$, the message digest $H(m)$, the curve C, and the optional configuration `config`.

Here is the sketch of the `computeC` method:

- G and l are extracted from the curve C.
- If `params` contains α, then perform Equation 1 using the ECC abstraction for point multiplication by G. Return the hashed result modulo l.

- Else, perform Equation 2 using the ECC abstraction for point multiplication and point addition (as two scalar multiplications and one point addition are required). Return the hashed result modulo l.

Observation 38: Ambiguous serialized data hashed in `computeC`

The `computeC` uses the `formatPoint` and `formatRing` methods from the utilities (see Section 3.8.3 and Section 3.8.4) as inputs of the hash function H to compute $\mathcal{H}(\mathcal{R}, H(m), \ldots)$ in the different cases, either using the private α or using the public data:

```
...
      return modulo(
        BigInt(
          "0x" +
            hash(
              formatRing(ring) +
                messageDigest +
                formatPoint(G.mult(params.alpha)),
              hashFct,
            ),
        ),
        N,
      );
...
```

The serialization for points and ring is not sound as detailed in Observation 75 (●) and Observation 78 (●): the methods suffer from ambiguity in the representation of the objects (the coordinates split during deserialization has multiple possibilities), and in the size of the produced string (since the size of the string will inherently depend on the size of the serialized `bigint` composing the fields). Also, `bigint` are serialized as decimal numbers, which is not very compact.

Although this does not bring concrete exploitable flaws, we usually want big numbers to be of fixed size with padding for a clean and reproducible serialization (and possible compatibility with external APIs that properly handle serialization). Also when serializing, binary data (instead of decimal or string hexadecimal) is more compact and more suited.

Recommendation:

Apply the recommendations in Observation 75 (●) and Observation 78 (●) for padded binary representations that would remove the current ambiguity.

Fix from Cypher Lab:

This should be fixed through Observation 75 (●) and Observation 78 (●) fixing (please refer to them for more details on the fixes).

Observation 39: Useless parameter `previousPubKey` in `computeC`

The ring $\mathcal{R}$ is passed as a parameter of `computeC` (which is mandatory for the hash computation $\mathcal{H}(\mathcal{R}, H(m), \ldots)$: this means that `previousPubKey` can be recovered from the index i if it is passed (which uses less memory).

Recommendation:

Use the index i as input of `computeC` instead of `previousPubKey`.

Fix from Cypher Lab:

This should be fixed in commit c85500e.

Observation 40: Useless hash function fixing in `computeC`

The following code for default hash function (set to `KECCAK256`) is useless, as the hash abstraction (see Section 3.5) deals with this:

```
...
    let hashFct = hashFunction.KECCAK256;
    if (config?.hash) hashFct = config.hash;
...
```

Recommendation:

Remove this code. If this is related to a way of providing a default value for `hashFct` that might be different than the one in the hash abstraction, then all the places where the hash function is used in `RingSignature` must handle this in an homogeneous fashion (e.g. in `signature` and so on).

Fix from Cypher Lab:

This should be fixed in commit dcb6779.

Observation 41: Missing sanity check for `params` in `computeC`

The `computeC` function does not check for `params` to be exclusively containing either α or $(r_{i-1}, c_{i-1}, K_{i-1})$.

Recommendation:

Add a check for exclusive α or $(r_{i-1}, c_{i-1}, K_{i-1})$ in `params`. This check should be performed in the beginning of the function (not in the end) to avoid useless computations.

Fix from Cypher Lab:

This should be fixed in commit 0753fea.

Observation 42: Split the usages for `computeC`

The `computeC` function suffers from readability because of the optional `params` used for the two usages.

Recommendation:

Splitting the two cases α and $(r_{i-1}, c_{i-1}, K_{i-1})$ for `computeC` in two dedicated functions would greatly benefit code readability.

Feedback from Cypher Lab:

This has not been fixed willingly (not being a priority).

Remark 3: Local side-channel leakage of `computeC`

The `computeC` method is not constant time at all: the branch handling the α case will take a lot less time (around 10 times less) than the other branch handling $(r_{i-1}, c_{i-1}, K_{i-1})$. This is because the $(r_{i-1}, c_{i-1}, K_{i-1})$ case uses an additional scalar multiplication with a public point that is not G, that does not make use of low level ECC wNAF optimisation (see Section 3.4 for more details).

This increases the opportunity for a local attacker to monitor the calls to `computeC` and distinguish between the two cases (e.g. observing the cache usage).

It is to be noted that this cannot be exploited in this context, since the upper loop using `computeC` always starts at $\pi + 1$ (see the `signature` method), hence avoiding the leakage of π.

The current remark is mostly here as a warning of `computeC` usage in other contexts, and more generally of cases (in potential future developments) where multiplication by G and multiplications by non-G points are considered. Ideally, a constant time version of `computeC` should be envisaged.

The static `partialSigToBase64` and `base64ToPartialSig` methods These methods are used to convert a `PartialSignature` to and from a `json` string encoded in `base64`. These conversions are straightforward, but no sanity check is performed here. While this is fine for `partialSigToBase64` since we can assume that an instantiated `PartialSignature` has been produced by legitimate APIs, this is not right when considering `base64ToPartialSig` that takes an external string malleable by attackers.

● Observation 43: Add sanity checks for `base64ToPartialSig`

The `base64ToPartialSig` method does not perform any sanity check on the parsed elements of the `PartialSignature`.

Recommendation:

Perform sanity checks in `base64ToPartialSig` on the ring, the challenges, the responses, α, π (that must be in the range of the ring length), on the configuration, etc.

Fix from CYPHER LAB:

Fixed by partial signature removal: see Observation 1 (●).

The `checkRing` function This function checks the validity of a ring of public keys $\mathcal{R} = \{K_1, \ldots, K_n\}$, which is an array of objects of type `Point`: it mainly checks that there is no duplicate public key in the ring using the `Set TypeScript` helper, and then calls `checkPoint` (see below) function on each element to check that every public key is on the same curve. An exception is thrown if any sanity check fails.

● Observation 44: Invalid test to detect duplicated public keys

The function `checkRing` checks that there is no duplicate public key in the ring using the instruction

```
// check for duplicates using a set
if (new Set(ring).size !== ring.length) throw err.noDuplicates("ring");
```

However, this test will not detect when an affine point (x, y) is twice in `ring`, it only detects when the corresponding `Point` *objects* are the same. For example, the following code

```
let curve = new Curve(CurveName.ED25519);
let A = new Point(curve, [1n, 1n], false);
let B = new Point(curve, [1n, 1n], false);
let ring = [A, B];
console.log(new Set(ring).size, ring.length, new Set(ring).size !==
        ring.length);
```

prints `2 2 false`, showing that it does not detect that the point $(1, 1)$ is twice in `ring`. It only detects it when both points are represented by the same `Point` object: the following code

```
let curve = new Curve(CurveName.ED25519);
let A = new Point(curve, [1n, 1n], false);
let ring = [A, A];
console.log(new Set(ring).size, ring.length, new Set(ring).size !==
        ring.length);
```

prints `1 2 true`.

Recommendation:

Fix the check that detects if there are duplicated public keys in the ring.

Fix from Cypher Lab:

This should be fixed in commit 8b29486. The fix consists in serializing to strings x and y for each point and using the same `Set` method to check the set size against the ring size.

The `checkPoint` function This function checks that a `Point` is on the curve it is supposed to belong to.

Observation 45: Perform sanity check at the beginning of `checkPoint`

In order to avoid useless computation, this check:

```
if (curve && !curve.equals(point.curve)) {
  throw err.curveMismatch();
}
```

should be performed before this one:

```
if (!point.curve.isOnCurve(point)) {
  throw err.notOnCurve();
}
```

Recommendation:

Change the sanity checks order in `checkPoint`.

Fix from CYPHER LAB:

This should be fixed in commit 00f5cf4.

The `piSignature` function (in `signature/piSignature.ts`) This function is executed in the wallet to compute the response r_π using the private key, the nonce α from the partial signature and Equation 3. The function is straightforward as it implements some sanity checks and then computes the equation to return r_π.

Observation 46: Incomplete or inconsistent sanity checks in `piSignature`

The sanity checks in `piSignature`:

```
if (
  alpha === BigInt(0) ||
  c === BigInt(0) ||
  signerPrivKey === BigInt(0) ||
  curve.N === BigInt(0)
)
  throw invalidParams();
```

are:

- Check that $\alpha \neq 0$, $c_\pi \neq 0$, and $k_\pi \neq 0$. This is not sufficient, they must be completed with a test that α, c_π and k_π elements are $< l$ (i.e. ensure they are modulo l, `curve.N` in the code). Also, the check that $c_\pi \neq 0$ is not right as this can happen with very small probability, related to Observation 12 ( ).
- The check that $l \neq 0$, albeit appropriate, is not consistent at all with the rest of *Alice's Ring* that does not perform any check on this curve order (which comes from trusted constants anyways). Also, why checking the order and not the other parameters of the curve (the prime, etc.)?

Recommendation:

Add the missing sanity checks for α, c_π and k_π modulo l, remove the check that $c_\pi \neq 0$ when the point at infinity is properly handled, and remove the useless check on $l \neq 0$.

Fix from Cypher Lab:

This should be fixed in commit ec652fd.

Code architecture. As presented above, the high-level structure of the code is composed of the class `RingSignature`, in which there are a constructor with sanity checks, getters, methods for exports, etc. We could imagine such a structure for the other "objects" of the library. For example, there could be a class for ring objects, centralizing all the sanity checks (about length, duplicates, valid points, etc.), the formatting methods (`formatRing`, see Section 3.8.4), the export methods, etc. We could also imagine a dedicated class for partial signatures `partialSignature`.

Observation 47: No dedicated class for ring objects

All the code related to the ring objects are spread in the files.

Recommendation:

We recommend to create a class dedicated to ring objects to improve code readability and limit the coding errors.

Feedback from Cypher Lab:

This has not been fixed willingly (not being a priority).

3.2.2 Schnorr signature: `signature/schnorrSignature.ts`

This file contains two functions: `schnorrSignature` and `verifySchnorrSignature` to produce and verify a Schnorr signature. These two functions are not used in the audited code of *Alice's Ring*, and are mainly here for sanity checks: Schnorr signatures indeed correspond to SAG ring signatures with a ring containing one public key[8].

[8] This is a variant of Schnorr with dedicated public key prefixing during hashing, and applied to the digest of the message $H(m)$.

Observation 48: Missing sanity checks in `schnorrSignature` and `verifySchnorrSignature`

The `schnorrSignature` misses the check that the private key is in $[1, l-1]$.

The `verifySchnorrSignature` verification function misses sanity checks on the inputs, namely that the challenges are in $[0, l-1]$, the response are in $[1, l-1]$, and that the public key is on the provided curve `Curve`, and that it is not of low order or hybrid for curves with a cofactor $h \neq 1$ (see Observation 32 (●) for this last issue).

These missing checks are rated as ● because the functions do not seem to be used in *Alice's Ring*.

Recommendation:

Add the missing sanity checks in `schnorrSignature` and `verifySchnorrSignature`.

Feedback from CYPHER LAB:

The authors have addressed this Observation after commit b5ef0b9 but the fix was not audited by CRYPTOEXPERTS.

Observation 49: Useless returned `ring` value

According to the signature of the function `schnorrSignature`, a value `ring` can be returned:

```
export function schnorrSignature(...): { messageDigest: bigint; c: bigint; r:
        bigint; ring?: Point[] }
```

However in practice, such a variable is never returned. Moreover, `ring` is not documented in the docstring.

Recommendation:

Remove the variable `ring` from the function signature.

Fix from CYPHER LAB:

This should be fixed in commit 409d2c9.

■ Observation 50: Strange comment in `verifySchnorrSignature` docstring

In the docstring of the function `verifySchnorrSignature`, the input `message` is documented as follows:

```
@param message - The message (as bigint) (= c[pi] in our ring signature
    scheme)
```

The comment "`= c[pi] in our ring signature scheme`" seems inaccurate.

Recommendation:

Remove the comment or improve it.

Fix from Cypher Lab:

This should be fixed in commit 409d2c9.

3.3 Review of the ECC point and curve abstractions

In this section, we present the two main ECC abstractions of *Alice's Ring*: the curve abstraction in `curves.ts`, and the points abstraction in `points.ts`.

3.3.1 Curves: `curves.ts`

```
curves.ts
class Curve {
    -name:  CurveName;
    -N: bigint;
    -G: [bigint, bigint];
    -P: bigint;
    ------------------
    constructor
    GtoPoint
    (from/to)String
    isOnCurve
    equals
}
derivePubKey
```

Figure 5: `curve.ts` methods and functions.

An overview of the content of `curves.ts` is provided on Figure 5. It contains the main class `Curve` representing an elliptic curve C, with the fields:

- `name`: this is the name of the curve, of type `CurveName` which is an enumeration of only two possible curves:

```
/**
 * List of supported curves
 */
export enum CurveName {
  SECP256K1 = "SECP256K1",
  ED25519 = "ED25519",
}
```

- `N` which is a big integer representing the order l of the generator of the curve.
- `G` which is the generator point, represented with its two $G = (G_x, G_y)$ affine coordinates as an array of two `bigint`.
- `P` which is the prime number q of the finite field $\mathbb{F}_q$ of the curve.

The `constructor` of the `Curve` class The constructor is straightforward as it simply instantiates the fields from constants depending on the `CurveName` provided as input argument:

```
// SECP256K1 curve constants
const SECP256K1 = {
  P: 2n ** 256n - 2n ** 32n - 977n,
  N: 2n ** 256n - 0x14551231950b75fc4402da1732fc9bebfn,
  G: [
    55066263022277343669578718895168534326250603453777594175500187360389116729240n,
    32670510020758816978083085130507043184471273380659243275938904335757337482424n,
  ] as [bigint, bigint],
};

// ED25519 curve constants
const GED25519 = new ExtendedPoint(Gx, Gy, 1n, mod(Gx * Gy));
const ED25519 = {
  P: 2n ** 255n - 19n,
  N: 2n ** 252n + 27742317777372353535851937790883648493n, // curve's (group) order
  G: [GED25519.toAffine().x, GED25519.toAffine().y] as [bigint, bigint],
};
...
  /**
   * Creates a curve instance.
   *
   * @param curve - The curve name
   * @param params - The curve parameters (optional if curve is SECP256K1 or
       ED25519)
   */
  constructor(curve: CurveName) {
    this.name = curve;

    switch (this.name) {
      case CurveName.SECP256K1:
        this.G = SECP256K1.G;
        this.N = SECP256K1.N;
        this.P = SECP256K1.P;
        break;
      case CurveName.ED25519:
        this.G = ED25519.G;
        this.N = ED25519.N;
        this.P = ED25519.P;
        break;
      default: {
        throw unknownCurve(curve);
      }
    }
  }
```

Observation 51: Redefinition of curves constants in `curves.ts`

The constants for the two possible curves `secp256k1` and `ed25519` are defined in `curves.ts` while they are already defined in the underlying `@noble-secp256k1` and `@noble-ed25519` libraries import:

```
const B256 = 2n ** 256n; // secp256k1 is short weierstrass curve
const P = B256 - 0x1000003d1n; // curve's field prime
const N = B256 - 0x14551231950b75fc4402da1732fc9bebfn; // curve (group) order
const Gx =
        0x79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798n;
        // base point x
```

```
const Gy =
        0x483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8n;
        // base point y
const CURVE = { p: P, n: N, a: 0n, b: 7n, Gx, Gy }; // exported variables
        incl. a, b
...
```

Why not reuse the constants?

Recommendation:

Reuse the constants from `noble` in order to avoid their redefinitions.

Feedback from Cypher Lab:

The authors have addressed this Observation after commit b5ef0b9 but the fix was not audited by CryptoExperts.

The `GtoPoint` method This method simply instantiates G as a `Point` abstraction (see Section 3.3.2) from the constant (G_x, G_y).

The `(to/from)String` methods These methods serialize and deserialize a curve to and from a `json string`. The `toString` is straightforward, transforming all the fields in `json` fields:

```
/**
 * Returns the curve as a json string.
 */
toString(): string {
  return JSON.stringify({
    curve: this.name,
    Gx: this.G[0].toString(),
    Gy: this.G[1].toString(),
    N: this.N.toString(),
    P: this.P.toString(),
  });
}
```

The `fromString` instantiates a curve object `Curve` from a `json` string, but only extracting the `CurveName` field:

```
/**
 * Returns a curve instance from a json string.
 *
 * @param curveData - the curve as a json string
 * @returns the curve instance
 */
static fromString(curveData: string): Curve {
  const data = JSON.parse(curveData) as {
    curve: CurveName;
  };
```

```
    return new Curve(data.curve);
}
```

● Observation 52: Rationale in `(to/from)String` operations

The `fromString` instantiates a curve only based on its `CurveName`: what is the purpose of serializing the other fields? Also, since `fromString` does not check fields other than `CurveName`, it is possible to have inconsistent `json` representations imported without any exception raised.

Recommendation:

Either remove useless fields from the serialization and deserialization, or add proper sanity checks on them.

Fix from Cypher Lab:

This should be fixed in commit 55fd6ff.

The `isOnCurve` method This method checks if an input `Point` (see Section 3.3.2 abstraction), or optionally an array of two `bigint` x and y, is on the curve or not. The affine coordinates (x, y) are checked to satisfy the curve equation in $\mathbb{F}_q$: $x^3 + 7 - y^2 = 0$ for `secp256k1`, and $y^2 - x^2 - 1 - d \times x^2 \times y^2 = 0$ with:

$$d = -\texttt{0x98412dfc9311d490018c7338bf86888617} \\ \texttt{67ff8ff5b2bebe27548a14b235ec8feda4} \pmod q$$

for `ed25519`. The method begins to check if $x = 0$ or $y = 0$ and returns an error if this is the case.

● Observation 53: Bad sanity check in `isOnCurve`

The following check is not appropriate:

```
...
    if (x === 0n || y === 0n)
      throw invalidParams("Point is not on curve: " + point);
...
```

This is true for `secp256k1` that does not have any point with $x = 0$ or $y = 0$ (since 7 has no square root and -7 has no cubic root in $\mathbb{F}_q$), and hence the test is useless as checking the curve equation is enough.

On the other hand, this is false for `ed25519`: $(0, 1)$ and

$$P := (\texttt{0x1c842602fd4d2a8b89e796e11c6a6488c3781305c4a610339f49e1228399}, 0)$$

are valid points on the curve, with $(0, 1) = \mathcal{O}$ being the point at infinity, which can be represented in affine coordinates contrary to the case of `secp256k1`. This check has however the side effect of avoiding to deal with the point at infinity for $(0, 1)$, and avoiding to deal with low order points as P is of low order. This erroneous check somehow prevents the small subgroups confinement attacks described in Observation 32 (●), explaining the ● rating in the current observation.

Recommendation:

On curves with $h \neq 1$ cofactor (`ed25519` in *Alice's Ring*), properly deal with low order points, and on curves with $h = 1$ only the point at infinity must be checked: these checks are only to be performed during **signature verification** where bad points could be injected. Also for all curves, properly deal with the point at infinity at low-level abstraction (in both cases where it satisfies the curve equation and does not satisfy it).

Feedback from CYPHER LAB:

The authors have addressed this Observation after commit b5ef0b9 but the fix was not audited by CRYPTOEXPERTS.

● Observation 54: Missing check in `isOnCurve`

The method `isOnCurve` does not check if the input coordinates x and y are in $\mathbb{F}_q$, so non-valid points (with unreduced coordinates) might satisfy the equation.

This is not an issue per se (and hence rated ●) because the ECC low-level layers will systematically reduce x and y to map their representatives in $\mathbb{F}_q$.

Recommendation:

Add a sanity check that $x, y \in \mathbb{F}_q$ at the beginning of `isOnCurve`, and return an error if this is not the case.

Feedback from CYPHER LAB:

> The authors have addressed this Observation after commit b5ef0b9 but the fix was not audited by CRYPTOEXPERTS.

The `equals` method This method simply checks the equality of two `Curve` objects, by checking if all their fields are equal:

```
equals(curve: Curve): boolean {
  return (
    this.name === curve.name &&
    this.G[0] === curve.G[0] &&
    this.G[1] === curve.G[1] &&
    this.N === curve.N &&
    this.P === curve.P
  );
}
```

The `derivePubKey` function This function derives the public key from the private key using a simple scalar multiplication of the private scalar with the base point G.

3.3.2 Points: `point.ts`

```
point.ts
class Point {
    -curve:  Curve;
    -x:  bigint;
    -y:  bigint;
    ------------------
    constructor
    mul
    add
    equals
    negate
    toAffine
    (to/from)String
    (to/from)Base64
    isValid
}
```

Figure 6: `point.ts` methods and functions.

An overview of the content of `point.ts` is provided on Figure 6. It contains the main class `Point` representing a point with affine coordinates (x, y) as two big integers on a curve `curve`: these are the three fields of the class. Although the low-level `noble` ECC layer deals with projective coordinates, the *Alice's Ring* abstraction only deals with affine coordinates: each time it calls the low-level abstraction, it uses `noble fromAffine` method to transform affine to projective, and then the `.x` and `.y` accessors to go back to affine coordinates for the result.

Observation 55: Point at infinity not handled

The point at infinity is not handled in the *Alice's Ring* abstraction, which forces to perform useless checks to ensure that it never appears. This seems to come from the fact that for `secp256k1` this point cannot be represented in affine coordinates, but for `ed25519` this point is $(0, 1)$.

Recommendation:

In order to have a clean API dealing with this point in all the cases, we advise to integrate its representation.

The `@noble-secp256k1` library exposes this point as $(0, 0)$ in affine coordinates:

```
toAffine(): AffinePoint {
  // Convert point to 2d xy affine point.
  const { px: x, py: y, pz: z } = this; // (x, y, z) in (x=x/z, y=y/z)
  if (this.equals(I)) return { x: 0n, y: 0n }; // fast-path for zero point
  if (z === 1n) return { x, y }; // if z is 1, pass affine coordinates as-is
  const iz = inv(z); // z^-1: invert z
  if (mod(z * iz) !== 1n) err("invalid inverse"); // (z * z^-1) must be 1,
      otherwise bad math
  return { x: mod(x * iz), y: mod(y * iz) }; // x = x*z^-1; y = y*z^-1
}
```

This means that this representation can be used at high level to represent and manipulate this point.

The `@noble-ed25519` uses the following:

```
toAffine(): AffinePoint {
  // converts point to 2d xy affine point
  const { ex: x, ey: y, ez: z } = this; // (x, y, z, t) in (x=x/z, y=y/z,
      t=xy)
  if (this.is0()) return { x: 0n, y: 0n }; // fast-path for zero point
  const iz = invert(z); // z^-1: invert z
  if (mod(z * iz) !== 1n) err("invalid inverse"); // (z * z^-1) must be 1,
      otherwise bad math
  return { x: mod(x * iz), y: mod(y * iz) }; // x = x*z^-1; y = y*z^-1
}
```

This is wrong, and some issues regarding `noble` dealing with the point at infinity will be discussed in Observation 66 (●). **Once this issue has been patched in `noble`** according to the recommendation, the point at infinity should be transparently handled at higher level in `point.ts`. This point at infinity and the low order and hybrid points should be then checked during the signature verification where the attacker can inject bad points.

Fix from Cypher Lab:

This has been fixed through proper handling of the point at infinity at various levels: at low level (see Observation 66 (●)), and at high level by properly checking it in signature verification functions.

● Observation 56: Sub-optimal projective to affine transformation

In order to transform the projective coordinates result to affine coordinates, the methods in the `Point` class use the `.x` and `.y` accessors from `noble`, but these accessors perform the same operation `this.aff()` twice (below is an extract from `noble`):

```
get x() {
  return this.aff().x;
} // .x, .y will call expensive toAffine:
get y() {
  return this.aff().y;
} // should be used with care.
```

The `'expensive toAffine'` comment here is related to the fact that a modular inversion, considered costly, is used in this method.

Recommendation:

Call the dedicated `toAffine()` method from `noble` that will perform a costly inversion only once.

Fix from Cypher Lab:

This should be fixed in commit 4dcd416.

● Observation 57: Code factorization

Many of the methods in `point.ts` follow the following pattern (example taken from the `mult` method):

```
  switch (this.curve.name) {
    case CurveName.SECP256K1: {
      const result = SECP256K1Point.fromAffine({
        x: this.x,
        y: this.y,
      }).mul(modulo(scalar, this.curve.N));
      return new Point(this.curve, [result.x, result.y]);
    }
```

```
      case CurveName.ED25519: {
        const result = ED25519Point.fromAffine({
          x: this.x,
          y: this.y,
        }).mul(modulo(scalar, this.curve.N));
        return new Point(this.curve, [result.x, result.y]);
      }
      default: {
        throw unknownCurve(this.curve.name);
      }
    }
```

As we can see, the same pattern of code is duplicated for both curves, which brings redundancy.

Recommendation:

Ideally factorize this using an “opaque” class that is either equal to `SECP256K1Point` or `ED25519Point`, while keeping the main code only once:

```
        const result = OpaquePoint.fromAffine({
          x: this.x,
          y: this.y,
        }).mul(modulo(scalar, this.curve.N));
        return new Point(this.curve, [result.x, result.y]);
```

This could be done using optional types.

Feedback from Cypher Lab:

This has not been fixed willingly (not being a priority).

Observation 58: Useless test verifying the validity of the output point

The outputs of many methods of `Point` are points formatted as `Point` objects. However, assuming that the low-level ECC implementation is sound, the `Point` constructor does not need to test if these points are on the curve.

Recommendation:

Set the constructor argument `safeMode` as `false` when building the output point in the methods `mult`, `add` and `negate`.

Fix from Cypher Lab:

This should be fixed in commit b5e6f5e.

The `constructor` method of `Point` This constructor takes a curve and two big integers x and y in the form of an array of two elements as inputs, and returns an instantiated `Point`. An optional `safeMode` boolean parameter (set to `true` by default) allows to ensure that the point is indeed on the provided curve.

Observation 59: Missing check in the `constructor` of `Point` – Observation 54

The `constructor` does not check if the input coordinates x and y are in $\mathbb{F}_q$, so non-valid points (with unreduced coordinates) might generate a new point without triggering an error.

Please refer to Observation 54 for the recommendation.

Fix from CYPHER LAB:

This is fixed through Observation 54 fixing.

The `mult` method This method performs a scalar multiplication of the instantiated `Point` with an input scalar.

Observation 60: Bad sanity checks in `mult`

The following check is only here to avoid point at infinity because it is not properly handled (see Observation 55):

```
...
    if (scalar === BigInt(0)) throw invalidParams("invalid scalar : 0");
...
```

Specific cases regarding the scalar (e.g. the fact that scalars in the ring signature must be non-zero) must be handled in the upper layer calling `mult`, not in the current layer.

Also, `scalar` must be checked to be strictly less than the order l, and return an error otherwise. Since there is a silent reduction, the API is not sound:

```
...
        }).mul(modulo(scalar, this.curve.N));
...
```

Recommendation:

Remove the check to zero when the point at infinity is properly handled. Add the check that the scalar is $< l$.

Feedback from Cypher Lab:

The authors have addressed this Observation after commit b5ef0b9 but the fix was not audited by CryptoExperts.

The `add` method This method adds two points together. It first checks that the two points are on the same curve, and then calls the low-level `noble` addition method to perform the computation on the projective coordinates.

The `equals` method This method checks if two points are equal. It first checks that the two points are on the same curve, and then calls the low-level `noble` equality check method to perform the computation on the projective coordinates.

The `negate` method This method computes the opposite of a point P, i.e. $-P$ so that $-P + P = \mathcal{O}$. This method calls the low-level `noble` negation method on the projective coordinates.

The `toAffine` method This method returns the two coordinates (x, y) as an array of big integers:

```
/**
 * Converts a point to its affine representation.
 *
 * @returns the affine representation of the point
 */
toAffine(): [bigint, bigint] {
  return [this.x, this.y];
}
```

Observation 61: Bad naming for `toAffine`

The name `toAffine` is not well chosen, as the point is already in its affine form in the `Point` class. This method performs coordinate extraction.

Recommendation:

Rename `toAffine` to a more suitable name, e.g. `toCoordinates`.

Fix from Cypher Lab:

This should be fixed in commit 3afafd3.

The `(to/from)String` and `(to/from)Base64` methods These `(to/from)String` methods export and import a `Point` to and from a `json` string: this is straightforward using `JSON.stringify` and `JSON.parse`. The `(to/from)Base64` methods do the same with `base64` encoded `json`.

Observation 62: Reuse `(to/from)String` in `(to/from)Base64`

The `(to/from)Base64` perform exactly the same work as `(to/from)String`, except that `base64` encoding or decoding is added. This means that code can be factorized here.

Recommendation:

Factorize the code and reuse `(to/from)String` in `(to/from)Base64`.

Fix from Cypher Lab:

This should be fixed in commit 9754753.

Observation 63: Duplicate serializations for `Point` and ring

The `toString` method is a serialization method for `Point`, but another serialization `formatPoint` exists in `utils/formatData/formatPoint.ts` (see Section 3.8.3).

The same issue holds for the ring that is a set of points and makes use of their serialization. On one side, we have `formatRing` (see Section 3.8.4) that makes use of `formatPoint`, and on the other side in the `RingSignature` class (see Section 3.2.1) we have the following serialization that uses the `point.toString` method:

```
...
ring: partialSig.ring.map((point: Point) => point.toString()),
...
```

What is the rationale of having two incompatible serialization methods for points and rings? This is error prone.

Recommendation:

Perform proper `Point` serialization in a unique method.

Feedback from CYPHER LAB:

The authors have addressed this Observation after commit b5ef0b9 but the fix was not audited by CRYPTOEXPERTS.

The `isValid` method This method checks if a point is on the curve by directly calling the `checkPoint` function from `ringSignature.ts` (see Section 3.2). This is a way of checking if points are on the curve after their instantiation, which could be useful if the constructor has been called with `safeMode=false`.

Observation 64: Outsourced function to check point validity

The `isValid` method relies on `checkPoint` defined in the file `ringSignature.ts`. According to Figure 3, `checkPoint` is in an upper level of the code architecture. There is no reason that `isValid` relies on a function from an upper level and it prevents this ECC level from being standalone.

Recommendation:

Use the function `isOnCurve` directly in this method, instead of relying on `checkPoint`. Moreover, the test in the constructor could rely on this method instead of reimplementing the test.

Fix from CYPHER LAB:

This should be fixed in commit 7a6a020.

3.4 Review of the ECC low level operations (taken from `@noble-secp256k1` and `@noble-ed25519`)

The two `@noble-secp256k1` and `@noble-ed25519` libraries code have been statically integrated in *Alice's Ring* source code in the `utils/noble-libraries/` folder. The two files `noble-SECP256k1.ts` and `noble-ED25519.ts` are almost untouched (except for comments), and have been extracted from recent versions of the libraries (no major security related commit have been pushed upstream at the time of the report writing). Since many of the algorithms and the observations are common to both libraries, and in order to make this section more readable, we will factorize the descriptions and will explicitly cite the differences when only one of them is impacted.

The `@noble-secp256k1` library implements the deterministic ECDSA signature according to `RFC 6979` with no external dependencies. The ECC operations (addition of points, scalar multiplication, etc.) are implemented over the `secp256k1` Weierstraß curve with equation $y^2 = x^3 + 7$. Projective coordinates are internally used: (x, y, z) where $(x \times z^{-1}, y \times z^{-1}, 1)$ represents the affine point (x, y). Affine to projective and projective to affine helpers are implemented. The point at infinity $\mathcal{O}$ does not have an affine representation satisfying the curve equation, which is the case in projective coordinates where it is represented with $(0, 1, 0)$.

The `@noble-ed25519` library implements the EdDSA signature according to `RFC 8032` with no external dependencies. The ECC operations are implemented over the Twisted Edwards curve `ed25519` with equation $y^2 - x^2 = 1 + d \times x^2 \times y^2$ with:

$$\begin{aligned} d = -&\texttt{0x98412dfc9311d490018c7338bf86888617} \\ &\texttt{67ff8ff5b2bebe27548a14b235ec8feda4} \pmod q. \end{aligned}$$

Extended projective coordintates are used: (x, y, z, t) where $(x \times z^{-1}, y \times z^{-1}, 1, x \times y)$ represents the affine point (x, y). Affine to extended projective and extended projective to affine helpers are implemented. The point at infinity $\mathcal{O}$ has an affine representative $(0, 1)$ that satisfies the curve equation, its representation in extended projective coordinates being $(0, 1, 1, 0)$. A major difference between `secp256k1` and `ed25519` besides the Weierstraß and Twisted Edwards representation is that `ed25519` is not a prime curve: it has a cofactor $h = 8$, implying small subgroups and hence some care when dealing with the signature. Dealing with the cofactor is in fact transparently handled by `RFC 8032` encoding of the points, and `@noble-ed25519` follows the guidelines to avoid issues with it.

Observation 65: The `noble` ECC libraries include huge dead code

Huge parts of `noble-SECP256k1.ts` and `noble-ED25519.ts` are not used by *Alice's Ring*, notably the functions related to signatures and to their encoding and decoding, as well as points (public keys) encoding and decoding.

All in all, only the following ECC methods (and their calling dependencies) are used by *Alice's Ring*:

- Basic operations on the curve: `add` for addition, `mul` for multiplication, `negate` for the opposite point.
- `fromAffine` for (extended) projective coordinates to affine, and `x`, `y` to get the affine x and y from (extended) projective coordinates.

Recommendation:

Remove the unused functions so that `noble-SECP256k1.ts` and `noble-ED25519.ts` remain simple and readable, with the bare minimum for the upper layers.

Feedback from Cypher Lab:

This has not been fixed willingly (not being a priority).

Observation 66: Bad affine representation of the point at infinity

When dealing with affine to/from projective transformations, `noble` is not handling properly the point at infinity with different causes and implications depending on the curve:

- On the `secp256k1` curve: the `toAffine` method encodes the point at infinity $\mathcal{O}$ in affine coordinates as $(0, 0)$:

```
...
    if (this.equals(I)) return { x: 0n, y: 0n }; // fast-path for zero
        point
...
```

  The code of `fromAffine` importing affine to projective is the following:

```
...
  static fromAffine(p: AffinePoint) {
    return new Point(p.x, p.y, 1n);
  }
...
```

  As we can see, `fromAffine` does not check for $(0, 0)$ to properly import $\mathcal{O}$. This means that silent errors in the computation will occur when exporting and then importing $\mathcal{O}$, which can happen with valid operations (such as $(l-1) \cdot P + P$ where P is a point of order l).

- On the `ed25519` curve: the `toAffine` method **wrongly** encodes the point at infinity $\mathcal{O}$ in affine coordinates as $(0, 0)$ as it is done for `secp256k1`, which seems to be a bad copy/paste:

```
static fromAffine(p: AffinePoint) {
  return new Point(p.x, p.y, 1n);
}
```

This is notably wrong as $\mathcal{O}$ must be represented by $(0, 1)$ in affine coordinates, and it satisfies the curve equation. This implies wrong computations when exporting and importing $\mathcal{O}$, which is not correct on Twisted Edwards known for transparently handling it.

This is rated as 🟡 because of spurious checks in *Alice's Ring* preventing to exploit this, see Observation 53 (🟡). When considering only `noble`, this should be rated as 🔴: this issue should be reported to the authors of `noble` for a proper upstream patch.

Recommendation:

Fix the two libraries:

- In `@noble-secp256k1`: simply fix `fromAffine` with the following:

```
static fromAffine(p: AffinePoint) {
  if ((x === 0n) && (y === 0n)) return new Point(0n, 1n, 0n);
  else return new Point(p.x, p.y, 1n);
}
```

- In `@noble-ed25519`: simply fix `toAffine` with the following:

```
...
  if (this.equals(I)) return { x: 0n, y: 1n }; // affine zero point (0, 1)
...
```

Fix from Cypher Lab:

This should be fixed in commit bb3acd7. Pull requests have also been pushed and accepted to the two upstream projects `@noble-secp256k1` (see PR 121) and `@noble-ed25519` (see PR 99).

In the following, we will only focus on the methods used by *Alice's Ring* (see Observation 65 (🟡)) and their dependencies, namely: `add`, `mul`, `negate`, `fromAffine`, `x`, and `y`. Since all the other methods are considered as dead code, they will not be covered in detail.

Here are some details about each method:

- `add`: the addition (and doubling) of projective points use complete formulas on both secp256k1[9] and ed25519[10], allowing to deal with $\mathcal{O}$ without any issue, and to also use add for point doubling (with a bit less performance).

- mul: the scalar multiplication of projective points for both curves follow the same algorithm. First of all, sanity checks are performed: if this is not the safe mode and if the scalar is 0, then $\mathcal{O}$ is returned (to optimize this case). If the scalar is zero or greater than the generator order ($\geq l$), an error is raised. Then, two execution paths are used. Either the input point is the base point G, and a wNAF algorithm is used for scalar multiplication. Or the input point is not G and a classical double-and-add ladder is used, with dummy computations in safe mode. The wNAF algorithm uses precomputations of multiples of G in a sliding window fashion, with a decomposition of the scalar in a w-ary NAF (Non-Adjacent Form) bringing a drastic speedup: we will not provide the details of the algorithm, please refer to [3] for background about wNAF. From a security perspective in *Alice's Ring*, only the wNAF algorithm is of interest as all the private scalars are used with the base point G (the private key k_π for the public key computation, the nonce α for the $c_{\pi+1}$ computation).

Observation 67: Possible local side-channels on wNAF scalar multiplication

In order to have a constant time wNAF, noble implements dummy operations whenever the value of the window (i.e. 8 bits of the scalar since the window is of size 8) is 0, which happens with probability $\frac{1}{256}$:

```
...
    if (wbits === 0) {
      f = f.add(neg(cnd1, comp[off1])); // bits are 0: add garbage to
        fake point
    } else {
      //           ^ can't add off2, off2 = I
      p = p.add(neg(cnd2, comp[off2])); // bits are 1: add to result point
    }
...
```

This strategy is efficient to keep constant time from a remote observer perspective, but is not efficient against local timing side-channels (e.g. exploiting the cache or the branch prediction). An example of such local attacks is the FLUSH+RELOAD cache attack on the LLC (Last Level Cache, usually L3) as presented in [3], where the authors exploit the previous condition leakage on the wNAF OpenSSL implementation for ECDSA over secp256k1. Although the implementation in [3] does not use dummy computations, the memory access patterns would still allow to distinguish between the two branches of the condition in the case of noble. There are three potential leaking sources:

[9] Algorithm 1 of [10].

[10] http://hyperelliptic.org/EFD/g1p/auto-twisted-extended-1.html#addition-add-2008-hwcd-3

1. The conditional branching that relies on a secret value: observing which branch is executed leaks the information whenever `wbits` is zero.
2. The memory access to the variables `f` and `p`: distinguishing which variable is used (either `f` or `p`) also leaks the information whenever `wbits` is zero.
3. The memory access to the array `comp`: detecting which case of the array is used leaks the value `off2` (or `off1`), which is highly correlated to `wbits`. This leaking source is inherent in the wNAF usage since this algorithm relies on precomputed look-up tables by design.

The exploitation requires multiple signatures (usually a few hundreds) under the same private key as it makes use of lattice attacks to solve the hidden number problem using few bits leakage of the nonce in each signature – see Section 3.1.1 on the nonce sensitivity. This perfectly fits the context of *Alice's Ring* regarding k_π and the nonce α usage in the wallet.

This is rated as ● because `TypeScript` adds layers of complexity: the code executes in a JIT (Just In Time) interpreter, which might add a lot of noise during synchronization and leakage gathering. This `TypeScript` noise is highlighted by the `noble` authors in their `README` (`https://github.com/paulmillr/noble-secp256k1/` and `https://github.com/paulmillr/noble-ed25519`) and discussed in old audit reports [13, 14] by Cure53.

Recommendation:

Although `TypeScript` makes this more complex to exploit, we consider that classical counter measures such as scalar blinding added to wNAF might be a good option for defense-in-depth. The possible mitigations are discussed in Section 5 of [3]. Other possibilities could involve using another algorithm for scalar multiplication, with the major drawback of rewriting huge parts of `noble`.

Feedback from Cypher Lab:

This has not been fixed for now, but is scheduled to be fixed.

- `negate`: the computation is straightforward:
  - On `secp256k1`, the opposite of a projective point (x, y, z) is $(x, -y, z)$.
  - On `ed25519`, the opposite of an extended projective point (x, y, z, t) is $(-x, y, z, -t)$.
- `fromAffine`, `x` and `y` (calling `toAffine` underneath): these are used for affine to projective and projective to affine conversions. Going from affine to projective is

simple as it consists of keeping x and y with their values and initializing $z = 1$ (and $t = x \times y$ for the extended projective coordinates). Going from (extended) projective to affine requires a division by z, hence an inversion computing z^{-1} to get $(x \times z^{-1}, y \times z^{-1})$ as the affine representation. These methods have some issues with the point at infinity explained in more detail in Observation 66 (●).

Remark 4: Other possible local side-channels in `noble`

There are other sources of side-channel leakage in `noble` beyond the one described in Observation 67 (●): we mention them in a remark rather than in an observation because we consider them hard to exploit, even locally, due to their very small execution timings:

- `noble` uses non-constant time `TypeScript bigint` arithmetic, which can leak information about the size of the operands, and hence leak bits of the nonces leading to key recovery with lattice attacks. Not much can be done here except modifying the underlying `bigint` library with a more robust one.
- The modular inversion is not constant time as it uses the Euclidean `gcd` algorithm. This could be exploited during projective to affine conversions using the z value as exposed in [1] to leak bits of the nonces and lead to private key recovery with lattice attacks. A classical counter measure to this is to use constant time inversion, e.g. using constant time modular exponentiation exploiting Fermat's little theorem in prime fields (which is the case in our context).
- The modular reduction `mod` uses an extra addition when the rest is negative, which is not constant time and could leak bits of sensitive assets when private data is manipulated. As a side note, the same extra addition is used in the upper layers since `utils/modulo.ts` (see Section 3.8.5) is the exact copy of `noble`'s `mod`.

3.5 Review of the hash abstraction

3.5.1 Hash: utils/hashFunction.ts

The `hashFunction.ts` file contains three functions dealing with cryptographic hash functions: `hash`, `keccak256` and `sha_512`.

Both `keccak256` and `sha_512` are simple gateways respectively for the functions `keccak_256` and `sha512` of the `@noble-hashes` library. These functions

1. take data to hash (of type `string`) as input,
2. run the corresponding function of the external library on it,
3. get the hash digest as `Uint8Array`, and
4. convert the latter into a hexadecimal string using the utils function `uint8ArrayToHex` (see Section 3.8).

Let us mention that the input string value is interpreted as a UTF-8 string: in the `@noble-hashes` library, the data are converted in an auxiliary function defined[11] as

```
export function utf8ToBytes(str: string): Uint8Array {
  if (typeof str !== 'string')
    throw new Error(`utf8ToBytes expected string, got ${typeof str}`);
  return new Uint8Array(new TextEncoder().encode(str));
}
```

while `TextEncoder` only supports UTF-8 encoding[12].

The `hash` function is a wrapper for any hash function. It takes the data to hash, together with a label that indicates which hash functions must be used. The available labels are defined in the `enum` structure named `hashFunction`:

```
export enum hashFunction {
  KECCAK256 = "keccak256",
  SHA512 = "sha512",
}
```

Currently, only `keccak256` and `sha_512` are supported as hash functions. The `hash` function just calls the desired hash function according to the given label. By default (if no label is given), it will use `keccak256`.

Observation 68: Inconsistent naming for hash functions

The two implemented hash functions are `keccak_256` and `sha512`. These two names are not consistent.

Recommendation:

[11] https://github.com/paulmillr/noble-hashes/blob/f209f442c0e7be33526e49ea5576a582981987e4/src/utils.ts#L116

[12] https://nodejs.org/api/util.html#class-utiltextencoder

Rename these functions to follow the same naming structure: either `keccak_256` and `sha_512`, or `keccak256` and `sha512`.

Fix from Cypher Lab:

This should be fixed in commit 78fa81f.

3.6 Review of the RNG abstraction

3.6.1 RNG: `utils/randomNumbers.ts`

The `randomNumber.ts` file contains two functions dealing with randomness:

- given `max` as input, the function `randomBigint` returns a random `bigint` value from $\{1, \ldots, \texttt{max}\}$, and
- given `min` and `max` as inputs, the function `getRandomSecuredNumber` returns a random `number` value from $\{\texttt{min}, \ldots, \texttt{max}\}$.

Both functions perform some sanity checks of their inputs, then they extract random bytes using the `randomBytes` function of the `node:crypto` library and convert them into a value in the desired range using sampling rejection to avoid any distribution bias. Let us mention that `getRandomSecuredNumber` can not handle numbers larger than $2^{51} - 1$ due to the inherent limitation of the `number` type, while `randomBigint` has not such a restriction.

According to the `nodejs` documentation, `randomBytes` "generates cryptographically strong pseudorandom data" and thus is suitable for cryptographic usages as in *Alice's Ring* context. This function may be blocking, since the "`crypto.randomBytes()` method will not complete until there is sufficient entropy available". It can be a performance issue in an environment where a lot of randomness is requested.

Remark 5: `crypto.randomBytes` implementation in `nodejs`

Although this goes beyond the scope of the current audit, it should be noted that the `nodejs` implementation of `crypto.randomBytes` is subject to some discussions as we can see in the following issue (still opened at the time of the report writing): https://github.com/nodejs/node/issues/5798

The highlight of this thread is the fact that `crypto.randomBytes` is based on the `OpenSSL` userland library, but the instantiation made by the `nodejs` code has neither been thoroughly reviewed nor audited. On the platforms where `nodejs` is deployed, it would be safer to directly use the kernel random interfaces (such as `/dev/random`) instead of relying on a userland post-processing.

Given the fact that *Alice's Ring* heavily relies on randomness quality, safer sources of random might be investigated.

Observation 69: `randomBigint` implementation not compliant with the docstring

The docstring of the `randomBigint` states that the output value is in $\{1, \ldots, \texttt{max}\}$:

```
/**
 * generate a random bigint in [1,max]
 *
 * @param max the max value of the random number
```

```
 * @returns the random bigint
 */
```

In practice, the function returns uniform values in $\{0, \ldots, \texttt{max} - 1\}$.

Recommendation:

Return the value `randomBig` incremented by 1.

Fix from Cypher Lab:

This should be fixed in commit aa604f1.

Observation 70: Inconsistent function naming between `randomBigint` and `getRandomSecuredNumber`

The name of the function `getRandomSecuredNumber` tends to indicate that the used randomness is more cryptographically secure than in the function `randomBigint`, while it is not the case since both functions rely on the function `randomBytes` and use the same rejection strategy.

Recommendation:

Homogenize the names of these functions.

Fix from Cypher Lab:

This should be fixed in commit aa604f1.

Observation 71: High rejection rate in the randomness functions

The two functions `randomBigint` and `getRandomSecuredNumber` use sampling rejection to avoid bias in the output distribution. With the current rejection strategy, in the worst case (i.e. when the size of the range is a bit larger than a power of 256), the functions reject the random bytes with probability around $255/256 \approx 0.996\%$ and thus need to perform many loop iterations, degrading the overall performance (especially since the pseudo-random generator may be blocking when there is not enough available entropy).

Recommendation:

Refine the rejection strategy to lower the rejection rate. For example, you can compute `n` as

$$\mathtt{n} \leftarrow \left\lfloor \frac{256^{\mathtt{byteSize}}}{\mathtt{range}} \right\rfloor ,$$

then reject only if `value` is larger than or equal to $\mathtt{n} \times \mathtt{range}$ (in the case we do not reject, the desired integer would correspond to `mod(value,range)`, with eventually an offset).

3.7 Review of the AEAD abstraction

3.7.1 AEAD: `encryption/encryption.ts`

The `encryption.ts` file contains functions to deal with public-key encryption using the algorithm `x25519-xsalsa20-poly1305`. It proposes three methods: `encrypt` to encrypt a message with a public key, `decrypt` to decrypt a message with a private key, and `getEncryptionPubKey` to derive the public key from the private key. Those three functions are gateways to the `encryptSafely`, `decryptSafely` and `getEncryptionPublicKey` functions of the library `@metamask/eth-sig-util`. The private key is a `bigint` value between 0 and the order of the `ed25519` generator, the public key is represented by a string value and the ciphertext is represented using the `EthEncryptedData` structure defined[13] in the external library:

```
export type EthEncryptedData = {
  version: string;
  nonce: string;
  ephemPublicKey: string;
  ciphertext: string;
};
```

Since the library `@metamask/eth-sig-util` requires that the given key is represented into a hexadecimal string of length 64 (to represent a 32-byte integer), the functions `decrypt` and `getEncryptionPubKey` convert the private key from a `bigint` value to a hexadecimal string. However, since a straightforward execution of `.toString(16)` does not guarantee to have a length-64 string, some padding/truncation is performed before calling the desired functions of the library.

Observation 72: Unconventional transformation of the `x25519` secret key

To guarantee that `.toString(16)` would lead to an hexadecimal string of length 64, the implementation performs some transformations of the secret key `privateKey`:

1. It converts the `bigint` value into a hexadecimal string which corresponds to its base-16 writing, then it builds an array of `Uint8` (i.e. `Uint8Array`) using a buffer. This operation removes the four least significant bits of the key as soon as $\lceil \log_{16}(\texttt{privateKey}+1) \rceil$ is odd. Precisely, the `Uint8` array represents a value `privateKey'` satisfying

$$\texttt{privateKey'} = \begin{cases} \left\lfloor \frac{\texttt{privateKey}}{16} \right\rfloor & \text{if } \lceil \log_{16}(\texttt{privateKey}+1) \rceil \text{ is odd,} \\ \texttt{privateKey} & \text{otherwise.} \end{cases}$$

2. If the size of the array is strictly larger than 32, the $8 \times n$ least significant bits of

[13] `https://github.com/MetaMask/eth-sig-util/blob/f8e84eaedeb1f9aa7e745d85c743cb8f579459dd/src/encryption.ts#L6`

privateKey' are removed, where n is the minimal integer such that the resulting key is smaller than 2^{256}.

3. If the size of the array is strictly smaller than 32, it pads the value "to the right" with zeros, meaning that it computes

$$\texttt{privateKey''} \leftarrow \texttt{privateKey'} \times 256^n,$$

where n is set such that $256^{31} \leq \texttt{privateKey''} < 256^{32}$.

4. The resulting key corresponds to a value between $16 \cdot 256^{31}$ and 256^{32} (excluded). The key is then transformed in a `bigint` value and reconverted into a hexadecimal string that represents it. Since the final version of the key is in the range $\{16 \cdot 256^{31}, \ldots, 256^{32} - 1\}$, it ensures that the resulting hexadecimal string will be of length 64. The implementation can thus call the desired method of the external library.

While it does not lead to a functional or security issue (it just drops few bits of the secret key), it implies that the implementation is not compliant with the standard practices, since it prevents the `x25519` key to be smaller than $16 \cdot 256^{31}$. Moreover, the current transformation is difficult to understand and prone to errors.

Recommendation:

Simplify the process that transforms the private key and avoid modifying its distribution. For example, the length-64 hexadecimal string of the private key could be computed as

```
privKey.toString(16).padStart(64,'0');
```

This code is simpler and does not drop bits of the key. It also allows all integers smaller than 2^{256}. If the private key is larger than or equal to 2^{256}, we would recommend to raise an error.

It is also to be noted that `@metamask/eth-sig-util` also exports the `padWithZeroes` function that achieves the same goal (see here):

```
/**
 * Pads the front of the given hex string with zeroes until it reaches the
 * target length. If the input string is already longer than or equal to the
 * target length, it is returned unmodified.
 *
 * If the input string is "0x"-prefixed or not a hex string, an error will be
 * thrown.
 *
 * @param hexString - The hexadecimal string to pad with zeroes.
 * @param targetLength - The target length of the hexadecimal string.
 * @returns The input string front-padded with zeroes, or the original string
 * if it was already greater than or equal to to the target length.
```

```
*/
```

Fix from Cypher Lab:

This has been fixed with partial signature removal – see Observation 1 (●) – as no more encryption is needed between the front-end and the wallet (and hence the transformation of the `x25519` secret has been completely removed).

3.8 Review of the various utilities

3.8.1 Converting `Uint8Array`: `utils/convertTypes/uint8ArrayToHex.ts`

The file `uint8ArrayToHex.ts` proposes the function `uint8ArrayToHex` which converts a byte array (`Uint8Array`) into a hexadecimal string.

Observation 73: Homemade `uint8ArrayToHex` conversion function

The function `uint8ArrayToHex` implements itself the conversion while there exists ways to convert using `nodejs` native builtins.

Recommendation:

Simplify the implementation by using `nodejs` builtins. For example, you can use

```
Buffer.from(array).toString("hex")
```

Since `uint8ArrayToHex` is only used twice (in the file `utils/hashFunctions.ts`), we would recommend to directly use the conversion implementation there instead of defining a specific function.

Fix from Cypher Lab:

This should be fixed in commit dc354bd.

3.8.2 Concatenating `Uint8Array`: `utils/formatData/concatUint8Array.ts`

The file `concatUint8Array.ts` proposes the function `concatUint8Array` which concatenates all the given byte arrays (`Uint8Array`) into a single one.

Observation 74: Homemade `concatUint8Array` concatenation function

The `@noble-hashes` external library dependency exports the `concatBytes` function (see here) for `Uint8Array` concatenation:

```
/**
 * Copies several Uint8Arrays into one.
 */
export function concatBytes(...arrays: Uint8Array[]): Uint8Array {
...
```

Recommendation:

Use the already present helpers when necessary: in this case use `concatBytes`.

Fix from CYPHER LAB:

This is fixed in commit c41eec5: the `concatUint8Array` has been removed as it is not used.

3.8.3 Formatting the point: `utils/formatData/formatPoint.ts`

The file `formatPoint.ts` proposes the function `formatPoint` which converts a `Point` object into a string. The output string corresponds to the decimal writing of the coordinate `x`, concatenated to the decimal writing of the coordinate `y`.

Observation 75: Non-unicity of the string representation of `point`

Since the decimal writings of the coordinates are not padded, the string representation of `Point` has a variable length and is not unique (i.e. it does not ensure that a string corresponds to a unique point). Indeed, the string `'121'` could correspond to the two following points:

```
new Point(curve, [12n, 1n])
new Point(curve, [1n, 21n])
```

Recommendation:

Pad the string representation of each coordinate such that its length only depends on the curve itself, and no more on the value.

Fix from CYPHER LAB:

This should be fixed in commit 8a373fd.

Observation 76: Bad docstring for `formatPoint`

The docstring of the function `formatPoint` states that the format of the ouput string depends on some configuration options described by the input `config`, but there is no such an input in the implementation.

Recommendation:

Fix the comment appropriately.

Feedback from Cypher Lab:

The authors have addressed this Observation after commit b5ef0b9 but the fix was not audited by CryptoExperts.

Observation 77: Outsourced formatting method for `Point`

The implementation of the `Point` class (in file `point.ts`) already proposes some methods to format such an object into a string, see Section 3.3.2. The choice of outsourcing this formatting method does not seem to be justified.

Recommendation:

Move this function as a method of the `Point` class or comment the choice of outsourcing it.

Fix from Cypher Lab:

This should be fixed in commit cc82c3e.

3.8.4 Formatting the ring: `utils/formatData/formatRing.ts`

The file `formatRing.ts` proposes the function `formatRing` which converts a list of `Point` objects into a string. The output string corresponds to the concatenation of the string representations of all the points (in the same order than in the input list), while the string representation of each point is computed using the function `formatPoint`.

Observation 78: Non-unicity of the string representation of a ring

Since the string representation of a `Point` object has not a fixed length (c.f. Observation 75 (●)) and since there is no delimiter between the points in the string, the function does not ensure that the output string corresponds to a unique list of points.

Recommendation:

Applying the recommendation of Observation 75 (●) would fix this observation.

Fix from Cypher Lab:

This should be fixed through Observation 75 (●) fixing.

Observation 79: Bad docstring for `formatRing`

The docstring of the function `formatRing` states that the format of the output string depends on some configuration options described by the input `config`, but there is no such an input in the implementation.

Recommendation:

Fix the comment appropriately.

Feedback from Cypher Lab:

The authors have addressed this Observation after commit b5ef0b9 but the fix was not audited by CryptoExperts.

Observation 80: Bad naming for `formatPoint` and `formatRing`

The `formatPoint` and `formatRing` methods (and file names) are not well chosen as they are not really explicit on what they do (these are serialization functions).

Recommendation:

Change the naming of `formatPoint` and `formatRing` to more explicit ones, such as `serialize(Point/Ring)` or `(Point/Ring)ToString`.

Fix from Cypher Lab:

This should be fixed in commit a7f35df.

3.8.5 Modulo operation: `utils/modulo.ts`

The file `modulo.ts` proposes the function `modulo` which computes the remainder of a division: given `n` and `p`, the function outputs $r \in \{0, \ldots, \mathtt{p} - 1\}$ such that $\mathtt{n} = q \cdot \mathtt{p} + r$ for some q. It differs from the native JavaScript implementation of modulo, which returns the *signed* remainder (i.e. the remainder is negative when the dividend is negative).

3.8.6 Utils Index: `utils/index.ts`

The file `index.ts` of the folder `utils` defines a constant `RegExp` object `base64Regex` which can be used to check that a string is a base-64 string.

● Observation 81: Bad placing of `base64Regex`

The `base64Regex` participates to the helpers, and placing it in `utils/index.ts` does not seem appropriate.

Recommendation:

Move `base64Regex` to a more appropriate helper file, e.g. `utils/base64.ts`.

Fix from Cypher Lab:

This should be fixed in commit ff48d50.

3.8.7 Errors: `errors/errors.ts`

The file `errors.ts` gathers all the `Error` objects of the ring signature scheme. They are either simple `Error` objects or functions that return an `Error` object. In the second case, some information can be provided as function input to complete the text message of the output error.

All the objects and functions available in `errors.ts` are listed in Figure 7.

errors.ts

```
noEmptyMsg:  Error
noEmptyRing:  Error
invalidSignature:  Error
computationError(string?):  Error
lengthMismatch(string?, string?):  Error
noDuplicates(string?):  Error
----- Number -----
   tooSmall(string?,number|bigint?):  Error
   tooBig(string?,number|bigint?):  Error
----- Params -----
   invalidParams(string?):  Error
   missingParams(string):  Error
   invalidJson(string|unknown?):  Error
   invalidBase64(string?):  Error
----- Points -----
   invalidPoint(string?):  Error
   notOnCurve(string?):  Error
   invalidCoordinates(string?):  Error
----- Curve -----
   unknownCurve(string?):  Error
   invalidCurve(string?):  Error
   differentCurves(string?):  Error
   curveMismatch(string?):  Error
---- Responses ----
   noEmptyResponses:  Error
   invalidResponses:  Error
----- Ring -----
   invalidRing(string?):  Error
```

Figure 7: `errors.ts` objects and functions.

References

[1] Alejandro Cabrera Aldaya, Cesar Pereida García, and Billy Bob Brumley. From A to Z: Projective coordinates leakage in the wild. Cryptology ePrint Archive, Paper 2020/432, 2020. `https://eprint.iacr.org/2020/432`.

[2] Diego F. Aranha, Felipe Rodrigues Novaes, Akira Takahashi, Mehdi Tibouchi, and Yuval Yarom. LadderLeak: Breaking ECDSA With Less Than One Bit Of Nonce Leakage. Cryptology ePrint Archive, Paper 2020/615, 2020. `https://eprint.iacr.org/2020/615`.

[3] Naomi Benger, Joop van de Pol, Nigel P. Smart, and Yuval Yarom. "Ooh Aah... Just a Little Bit" : A Small Amount of Side Channel Can Go a Long Way. In Lejla Batina and Matthew Robshaw, editors, *Cryptographic Hardware and Embedded Systems - CHES 2014 - 16th International Workshop, Busan, South Korea, September 23-26, 2014. Proceedings*, volume 8731 of *Lecture Notes in Computer Science*, pages 75–92. Springer, 2014.

[4] Konstantinos Chalkias, François Garillot, and Valeria Nikolaenko. Taming the many EdDSAs. Cryptology ePrint Archive, Paper 2020/1244, 2020. `https://eprint.iacr.org/2020/1244`.

[5] Thomas Hussenet, Nathan Hervier, Maxime Dienger, and Adam Dahmoul. Alice's Ring Protocol Whitepaper V 1.0, 2023.

[6] Koe, Kurt M. Alonso, and Sarang Noether. Zero to Monero: Second Edition, 2020. `https://www.getmonero.org/library/Zero-to-Monero-2-0-0.pdf`.

[7] Joseph K. Liu, Victor K. Wei, and Duncan S. Wong. Linkable Spontaneous Anonymous Group Signature for Ad Hoc Groups. Cryptology ePrint Archive, Paper 2004/027, 2004. `https://eprint.iacr.org/2004/027`.

[8] Clémentine Maurice. *Micro-architectural side channels: Studying the attack surface from hardware to browsers.* Habilitation à diriger des recherches, Université de Lille, May 2023.

[9] Amir Naseredini. Exploring the Horizon: A Comprehensive Survey of Rowhammer, 2023.

[10] Joost Renes, Craig Costello, and Lejla Batina. Complete addition formulas for prime order elliptic curves. Cryptology ePrint Archive, Paper 2015/1060, 2015. `https://eprint.iacr.org/2015/1060`.

[11] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. CLKSCREW: Exposing the perils of Security-Oblivious energy management. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1057–1074, Vancouver, BC, August 2017. USENIX Association.

[12] Cure53. Audits-Report TypeScript Hashing Libraries 12.2021, 2021. https://cure53.de/pentest-report_hashing-libs.pdf.

[13] Cure53. Review-Report noble-secp256k1 Library 04.2021, 2021. https://cure53.de/pentest-report_noble-lib.pdf.

[14] Cure53. Audit-Report noble-ed25519 TypeScript Library 02.2022, 2022. https://cure53.de/pentest-report_ed25519.pdf.