Monero CLSAG review

Supported by OSTIF

JP Aumasson, Antony Vennard

July 29, 2020

Contents

Contents 2

1 Introduction 3

2 Paper review 5
- 2.1 General assessment . . . 5
- 2.2 Specific comments . . . 6
  - 2.2.1 In 2 Preliminaries . . . 6
  - 2.2.2 In 3.1 Hardness Assumptions . . . 6
  - 2.2.3 In 3.2 Linkable Ring Signatures . . . 7
  - 2.2.4 In 3.3 Linkability . . . 8
  - 2.2.5 In 3.4 Unforgeability and Non-Slanderability . . . 8
  - 2.2.6 In 3.5 Linkable Anonymity . . . 9
  - 2.2.7 In 4 Construction . . . 9
  - 2.2.8 In 5 Proofs of Security . . . 9
  - 2.2.9 Comments on notation . . . 11

3 Code review 13
- 3.1 Summary . . . 13
- 3.2 Methodology . . . 14
- 3.3 Implementation approach . . . 15
- 3.4 Findings . . . 16
  - 3.4.1 key struct misuse prevention . . . 16
  - 3.4.2 Unnecessary code duplication . . . 17

1

Introduction

The Monero community solicited our help, via OSTIF, to review the security of the new signature mechanism CLSAG (concise linkable ring signatures), a variant of Monero's MLSAG (multilayered linkable spontaneous anonymous group signatures, which is actually a ring signature rather than a group signature scheme).

The CLSAG paper establishes a new general security result linking unforgeability and non-slanderability (a variant of unforgeability that intuitively looks harder to achieve).

The draft paper we reviewed was version 20200330:160235, available at `https://eprint.iacr.org/2019/654/20200330:160235`. In response to feedback in this report and subsequent disccusions for clarification, the paper has since been updated. The latest version is available at `https://ia.cr/2019/654`

This engagement consists in two parts:

1. A thorough review of the theoretical security of the scheme as specified in the paper, with a focus on the correctness of the main results and proofs thereof.
2. A security audit of the C++ source code implementing CLSAG, available on GitHub (commit `66bcde2`, May 12).

We would like to thank the Monero Community, for trusting us with this project, and we would like to thank OSTIF for making this work happen and for their organizational assistance.

Report feedback and impact

After sharing a first version of this report, Monero stakeholders brought to our attention inaccuracies in our observations (which we have corrected). They also described their

response to each of the issues we raised, and shared with us new versions of Theorem 1's proof, Definition 2, and Theorem 5, which adequately address our comments. A revised version of the paper has been created and shared on ePrint, see https://ia.cr/2019/654.

We thank Monero for their timely and clear follow-up to our report.

2

Paper review

We reviewed the construction and related security results from the paper *Concise Linkable Ring Signatures and Forgery Against Adversarial Keys*, version 20200330:160235 available at https://eprint.iacr.org/2019/654/20200330:160235 (the latest version, incorporating feedback and changes since this review, is available at https://ia.cr/2019/654).

In order to good a good understanding of the scheme, we reviewed all of the paper's technical content, and report our observations below, regarding technical correctness, completeness, as well as editorial quality.

2.1 General assessment

The paper describes CLSAG signatures as an extension of previous MLSAG signatures, following similar design principles. The new security results regarding the strength of the proposed CLSAG construction appear correct to us. We have a good confidence that the construction proposed satisfies the intended security goals, yet a moderate confidence that the proofs are entirely valid.

Indeed, the article sometimes lacks rigor and clarity, be it in notations, definitions, or proofs.

The main general result, in Theorem 1, is relatively straightforward and appears to us to be correct, and adequately demonstrated by its proof.

The most important result is Theorem 2, about the security of the proposed CLSAG construction. Although we could no find flaws in the result nor its proof, we point out potential gaps that may affect the security level of the final scheme, as described in the rest of this report. We believe that the security model considered is strong enough to capture realistic adversaries.

We also report that the proof of Theorem 5, about linkable anonymity, is a bit "handwaved", and seems to omit details to be complete and verifiable.

Finally, we note that although the paper reports benchmarks of an implementation of the proposed scheme, it does not elaborate on the actual instantiation of the proposed construction, and notably does not quantify its security with respect to the bounds established in Theorem 2. Although we have not made such a detailed analysis, our expectation is that the instance implemented should provide a high enough security.

The following section provides more specific comments, including reports of inaccuracies, errors, and suggestions of improvement.

2.2 Specific comments

For clarity, this section is split in subsections, corresponding to sections or subsections of the article.

2.2.1 In 2 Preliminaries

The random oracles $\mathcal{H}^s$ and $\mathcal{H}^p$ are defined as functions sampled from an infinite domain, which is incorrect, as it cannot be mathematically defined. Instead, a random oracle is usually explained as a system that picks and registers a fresh random value for each new input. Furthermore, either these are random oracles, or hash functions, but the current write-up uses both terms.

2.2.2 In 3.1 Hardness Assumptions

Undefined collision resistance

The function μ is defined as collision resistant, but the notion of collision resistance is not clearly defined in this context. First, are the μ_i coefficients known to an attacker? Our understanding is that they are secret, and fixed per instance of μ, but remain the same over repeated queries to the function.

However, under this assumption an attacker choosing $\mathbf{x}$ inputs can trivially recover all μ_i coefficients in order to determine collisions. The paper should therefore clarify what notion of collision resistance is considered.

Incomplete Definition 1

It might help to note that $\kappa < q$. The term "challenge keys" is used once but was not introduced. The notation $\mathbb{F}^*$ for $\mathbb{F} \setminus \{0\}$ was introduced in section 2 but is not used here.

Forking lemma: superfluous formalism?

The forking lemma is expressed in terms of an algorithm equipped with a random tape, highlighting the underlying Turing machine model. However this model is not strictly

necessary, furthermore the "random tape" should be defined more precisely (since a tape, or initial state of the machine, can be of arbitrary length). Instead, one can just talk of a random string as input. The forking lemma can then be applied in terms of state of the algorithm's variable, instead of a Turing machine's tapes and index.

The notation $[q]^2$ should be defined, as many readers will not be familiar with it.

Inaccuracies in Definition 2

Definition 2 defines the "Random Oracle Decisional Diffie–Hellman" problem, although it does not have much to do with Diffie–Hellman operations: no secret key (scalar) is applied to a public key (group), instead multiplication happens between a scalar and the hash of its own public key. The problem is therefore more about distinguishing group elements than Diffie–Hellman results. In particular, given an oracle solving the Diffie–Hellman problem, the current DDH variant does not seem breakable.

In the text preceding the definition, "as hard" should be "at least as hard". Simple rationale behind this assumption should probably appear too.

The use of advantage here is also confusing. Advantage is usually defined as:

$$Adv = |P[\text{attacker succeeds}] - P[\text{idealized case}]|$$

We would normally expect that, for example if the idealized case has probability exactly $1/2$, and the attacker's success is denoted as A, then:

$$Adv = |A - \frac{1}{2}| \leq \epsilon$$

where ϵ is negligible. This may be what is meant, but the term ϵ has been used for the attacker's outcome, implying that it is negligible and consequently that the advantage is $1/2$.

2.2.3 In 3.2 Linkable Ring Signatures

Definition 3 improvements

For completeness, it might be added that the algorithms are PPT, and which ones are randomized (at least KEYGEN).

In the post-definition comments, ϕ might be called a "one-way map" instead of just a "map".

Hash function notations

Hash functions $\mathcal{H}^s$ and $\mathcal{H}^p$ are cited but these do not appear in the definition.

The example of elliptic curves uses the notation H_s for a hash function, whereas $\mathcal{H}^s$ was introduced (same for p instead of s).

Definition 4 improvements

The events E_i would be easier to parse for a reader if they had more explicit names. For example,

1. "IsKeyPair" for E_1,
2. "ValidSecretKey" for E_2,
3. "VerifiedSig" for E_3,
4. "LinkCommute" for E_4,
5. "TwoSigsFrom1key" for E_5,
6. "TwoSigsFrom2keys" for E_6.

2.2.4 In 3.3 Linkability

We recommend to add more precise references to the original definitions (Definition 5 in the ACST paper and Definition 8 in the DBHKS paper), and to explain why Definition 5 in the present paper is called ACST, and why the second is called pigeonhole (at term not used in the original paper).

Furthermore, Definition 5 may more explicitly note that anonymity sets are not necessarily subsets of S (as noted in original definition).

A comment may be added to clarify that in Definition 6, the attacker does not have access to a signing or corruption oracle.

2.2.5 In 3.4 Unforgeability and Non-Slanderability

It would be interesting to discuss how and why Definition 7 differs from unforgeability as defined in BDHKS (page 15-16, where none of the keys in the anonymity set must be corrupted) and in ACTS (Definition 1, same difference among others). The current Definition 7 is thus stronger in the sense that it gives way more power to an attacker.

Point 3. of Definition 7 should say that i is in $0 \ldots q-1$, in addition to the fact that pk_i is in S and was not corrupted.

Theorem 1

The result is fairly straightforward, and the proof, although only informal, convincingly shows that any attacker for Definition 7 can be used to construct an attacker for Definition 8, and the other way. A slightly more formal proof would be good to have though.

2.2.6 In 3.5 Linkable Anonymity

Inaccuracies in Definition 9

"with probability at least ϵ" should be "with advantage at least ϵ", consistently with the later definition of linkable anonymity.

It might clarify to add that $c' = c \oplus b$ (xor). It may also be made clearer that the signing oracle (SO) substantially differs from the one used in previous definitions.

It would be useful to explain how and why this definition differs from the original one.

Furthermore, this linkable anonymity refers to the anonymity of individual keys, but it may be necessary to have another definition regarding linkability of subsets of signers (against attackers that cannot deanonymized linked signatures, but reduce the set of possible signers to a subset of the intersection of the anonymity sets).

2.2.7 In 4 Construction

The random oracle notation $\mathcal{H}$ is used to denote hash functions, which can be confusing. The definition of the scheme should be in terms of hash functions, and the proof, if needed, based on the assumption that said hash functions are random oracles.

The notation $\circ$ is used for multiplication but does not seem defined otherwise.

The definition seems to be general to arbitrary fields and groups, but is clearly written with elliptic curve structures in mind, and sometimes used elliptic curve terms (as in "compute the points"). It might be clearer to express everything in terms of elliptic curve groups, otherwise elliptic curve-specific terms should probably be avoided.

For completeness, it might be added that α is a uniformly random field element, rather than just a sampled element.

Although similar to MLSAG and similar constructions, the signature construction where c_0 is computed after wrapping around n indices should be clarified. Currently, only "by identifying index n with index 0" suggests this behavior. It would be clearer to specify that the index is incremented modulo n.

The equation defining $c_{\ell+1}$ is identical to that defining c_{i+1}, for $i = \ell$, and is thus not necessary.

Point 4 can be shortened to, for example, "Sign returns the signature $\sigma = \ldots$."

The algorithms Sign, Verify, and so on are typeset using a different style than in Definition 3, is there a reason for this?

2.2.8 In 5 Proofs of Security

Theorem 2

Theorem 2 is the main security result establishing the security of the proposed construction.

In Theorem 2, the value η used in the probability expression is not defined (it comes from the forking lemma, but this should be specified).

The constants t_0 and t_1 should be estimated, because concrete bounds are considered here, as opposed to asymptotic results.

The proof of Theorem 2 leverages Theorem 1's equivalence result to work in the simpler non-slanderability setting, and leverages the chaining between all c_i values and their dependency on the aggregated public keys and oracled version of it to demonstrate that any forgery would ultimately required the solving of a discrete logarithm of a linear combination of challenges.

The proof is detailed, and relatively clearly explained. It leverages the forking lemma reasoning presented in an introductory section. Below we report some observations regarding potential gaps between the proof and the result, as well as between the specified scheme and its actual instantiation.

The proof makes the assumption that any successful forgery must have all c_i computations valid. Although this seems intuitively natural, it seems necessary to quantify the probability that this does not hold, namely that a forgery be created without having all valid c_i's (and associated oracle queries).

The general reasoning of the proof seems sound to us, and we did not identify flaws in the complexity, query, and probability estimates. However, greater accuracy seems required to have a better estimate of the security of concrete instances. In particular, the bit-level security of proposed instances should be estimated, taking into account the discrete log variant best attack's complexity, and estimates for t_0 and t_1.

Furthermore, the result is given in the random oracle model: it would be interesting to at least informally relate collision and preimage resistance to the CLSAG construction security. Our gut feeling is that arbitrary collisions are hard to exploit, but we may be wrong.

Theorem 3

This result seems to follow as an observation from Theorem 2's proof. It might be better to appear as a corollary than as a standalone theorem.

Theorem 4

This result relies on the collision resistance assumption, whose generality we questioned in an earlier section of this report. The collision resistance function should be described more specifically, in a way that relates it to the collision and/or preimage resistance of the underlying hash functions.

Theorem 5

Reference to Definition 9 may be added, e.g. as “is linkably anonymous under Definition 9”. The proof of Theorem 5 is very short and did not appear convincing too us, as it lacks details about how it relates the proposed scheme to the “oracle DDH”, and how distinguishing between signatures leads to distinguishing elements of the oracle DDH problem.

Typo “piar”.

2.2.9 Comments on notation

Some of the notation in use in the paper is overcomplicated or non-standard and may be simplified in order to make the paper clearer. This section will list a non-exhaustive set of issues that may need to be considered.

Firstly and perhaps most egregiously, there is excessive use of complicated typefaces for symbols, which is unnecessary. This includes the use of `mathcal`, `mathbb` and `mathfrak` for terms defined by the paper where simple letters would do. As examples, in probability, $\mathbb{P}$ is rarely used. The main use of the blackboard font-face is for what we might term well-known sets, and $\mathbb{P}$ often refers to projective space over which the group law for elliptic curves holds.

Sequence notation is typical in real and complex analysis. For example, one might write:

$$\left\{\frac{1}{in}\right\} \in \mathbb{C}$$

by which we mean a sequence of elements all of which are contained in $\mathbb{C}$ and are characterized by n increasing from 1. Notation such as $\{p_i\} \subset \mathbb{F}$ makes little sense. The subset notation might be taken to mean subfield, but it is also unnecessary to invoke sequence notation here. All we care is that the p_i are in the field in question. One might instead prefer set construction notation:

$$p_i = \{f(a,b) : a, b \in \mathbb{F}\}$$

if there is some meaningful f that should be known. If not, then it is far clearer simply to say that there is a set of elements p_i chosen from the finite field.

Indeed, when talking about fields themselves, if a general field is desired we typically say let K be a field. There are good reasons for the use of $\mathbb{F}_p$ – finite fields have properties not inherent to all fields generally: any finite field is perfect and so Galois in the sense of Galois theory, and any finite field of a given order is isomorphic to any other finite field of that order. For that reason, any $\mathbb{F}_p^n$ is isomorphic to any other, regardless of the choice of characteristic polynomial. We thus use blackboard notation because of the generality involved.

The paper also switches between field notation for a generic field, and that of a finite field. As is normal in cryptography, we require a group over which the discrete logarithm problem is hard, but that the field is also finite. One should be sure when asserting general statements that these are generally true. It is perfectly valid to restrict the definition of a field to an appropriate one.

It does not make sense to explicitly define order, or secret keys, as the multiplicative group associated with a field (the $\mathbb{F}^*$ notation) because this does not make sense in all settings. For example, the curve FourQ is defined by the equation:

$$E/\mathbb{F}_{p^2} : -x^2 + y^2 = 1 + dx^2y^2$$

The base field here can be defined as follows:

$$\mathbb{F}_{p^2} = \{a + bx : a, b \in \mathbb{F}_p\}$$

Clearly, it is not generally true that elements of the multiplicative group associated with this field, i.e. $\mathbb{F}_{p^2}$, are integers. This is, in fact, the definition of Gaussian integers with coefficients modulo p (x should be taken as i). It is therefore not true that these can be taken in general to represent the order of a given group element. More clearly stated: scalar multiplication in these curves still uses integers less than 2^{256}, not Gaussian integers.

This type of construction (over extension fields) is common in pairing-based settings.

Finally, the use of $\sigma, \sigma^*, \sigma'$ is somewhat confusing. It may be more clear to use subscripts and numbers, particularly as greater numbers of signatures are in use.

3

Code review

The scope of the code review includes all the code implementing and directly supporting the integration of CLSAGs in Monero. The main file of interest is ringct/rctSigs.cpp, with other files of interest in ringct/ and device/. We reviewed the code base at commit `66bcde2` (May 12).

3.1 Summary

Overall, we believe the code implements the CLSAG scheme as desired accurately and provides a high level security.

Our assessment results can be summarized as follows:

- The code is clean and readable - it avoids the potential complexities sometimes found in large C++ code bases.
- The code reuses well tested and understood components to build on.
- The code contains helpers for error conditions that make it simple to check errors and exit as needed.
- The code accurately implements the CLSAG signature scheme, and we found no major security issues.

As with all C++ code bases, risks remain. We would recommend our client continue the unit testing and fuzzing setup they already have and extend this to CLSAG. We have provided some possible recommendations as part of our feedback to remove redundant code and a possible suggestion for automatic memory erasure using C++'s features, which the client may find useful.

The remainder of this report discusses our methodology for approaching the code review, and our findings and recommendations.

3.2 Methodology

Our reference for the code review was the paper reviewed above. We compared this to the code implementation to find possible discrepancies in the algorithmic logic as well as encoding, bit ordering, etc.

The code contains a hardware device abstraction. We examined the default software device for potential faults. We did not review device-specific code as this was out of scope of the audit.

When auditing the code for pure code safety (as opposed to correctness), we specifically looked for

- Correct and safe usage of previously implemented components.
- Correct and safe usage of cryptographic APIs
- The handling of edge cases that could lead to unsafe behavior (for example, potential ambiguous encodings).
- Software security bugs, such as:
  - Potential memory corruption issues.
  - Memory leaks and unsafe use of dynamic allocators.
  - Arithmetic bugs, such as integer overflow and division by zero.
  - Unhandled or unsafely handled errors.

We did this via a combination of static analysis (reading code, comparing to specifications/expected outcomes) and some dynamic analysis tooling.

As in all C++ audits we do, we ran linters and analyzers to get an idea of the general code hygiene and safety. We specifically employed:

- cppcheck, a static analysis tool.
- clang-tidy, a clang/LLVM-based static analysis tool.
- CodeQL, a source code analysis tool by Semmle and now Github, to explore potential issues in the code.

We reviewed the output of these tools for the code in scope for our audit. These tools did not reveal any security-specific faults in the code.

3.3 Implementation approach

The code reviewed implements the d-CLSAG scheme as described by the reviewed paper.

The code provides a signature generation function that does much of the heavy lifting for signatures:

```
// Generate a CLSAG signature
// See paper by Goodell et al. (https://eprint.iacr.org/2019/654)
//
// The keys are set as follows:
//   P[l] == p*G
//   C[l] == z*G
//   C[i] == C_nonzero[i] - C_offset (for hashing purposes) for all i
clsag CLSAG_Gen(const key &message, const keyV & P, const key & p,
                const keyV & C, const key & z, const keyV & C_nonzero,
                const key & C_offset, const unsigned int l,
                const multisig_kLRki *kLRki,
                key *mscout, key *mspout, hw::device &hwdev)
```

The remaining signature logic is present in the prove function, whose signature is as follows and invokes the above generation function with suitable parameters:

```
clsag proveRctCLSAGSimple(const key &message, const ctkeyV &pubs, const ctkey &inSk,
    const key &a, const key &Cout, const multisig_kLRki *kLRki,
    key *mscout, key *mspout, unsigned int index, hw::device &hwdev)
```

Another focus of our review was the verification function, which has the following signature in the code we audited:

```
bool verRctCLSAGSimple(const key &message,
         const clsag &sig, const ctkeyV & pubs, const key & C_offset)
```

For their elliptic curve operations, these functions appear to use code based on or closely following the techniques from the ed25519 reference implementation, and as such give us high confidence in their correctness.

Some functions are composites of these APIs for implementing the necessary arithmetic, e.g. `addKeys_aGbBcC` and `hash_to_p3`. These functions make appropriate use of the underlying primitive implementations and appear to be correct.

An important point in reading this code is that a class `hw::device` exists that provides backing cryptographic operations. This is an abstraction to enable potential hardware

devices to exist. A default 'soft-device' is provided by default. We have examined its function:

```
bool device_default::clsag_sign(const rct::key &c, const rct::key &a,
                                const rct::key &p, const rct::key &z, const rct::key &mu_P,
                                const rct::key &mu_C, rct::key &s) {
    rct::key s0_p_mu_P;
    sc_mul(s0_p_mu_P.bytes,mu_P.bytes,p.bytes);
    rct::key s0_add_z_mu_C;
    sc_muladd(s0_add_z_mu_C.bytes,mu_C.bytes,z.bytes,s0_p_mu_P.bytes);
    sc_mulsub(s.bytes,c.bytes,s0_add_z_mu_C.bytes,a.bytes);

    return true;
}
```

which implements the desired final scalar computation.

$$s_l = \alpha - c_l w_l$$

from the CLSAG paper we reviewed earlier.

Random numbers are generated using a DRBG seeded from system-provided CSPRNGs, specifically `/dev/urandom` or `CryptGenRandom` as appropriate.

3.4 Findings

3.4.1 key struct misuse prevention

Severity: low

Description

The `key` struct is is as follows:

```
struct key {
    unsigned char & operator[](int i) {
        return bytes[i];
    }
    unsigned char operator[](int i) const {
        return bytes[i];
    }
    bool operator==(const key &k) const { return !crypto_verify_32(bytes, k.bytes); }
    unsigned char bytes[32];
};
```

In C++11, this will come with default copy and move constructors, which allows developers to manipulate the struct and produce unintentional duplicates when C++ triggers a copy operation. This may prevent `memwipe` from being called everywhere that is intended.

Recommendation

From C++11 onwards, you can delete copy constructors, preventing their default inclusion. In this struct, add:

```
key(const key&) =delete;
key& operator=(const key&) =delete;
```

As a similar, related countermeasure, you may wish to add a destructor as follows:

```
~Key() {
    memwipe(this->bytes, sizeof(this->bytes));
}
```

This can be public without `virtual` provided no types are derived from this type (subclasses) so that deletion does not need to resolve the concrete type - this would be undefined behaviour. If you need to inherit from this base class, declare the destructor virtual. See this rule from Guru-of-the-Week.

This is not an obligatory recommendation and not implementing it does not imply the solution is insecure. It is entirely possible to audit use of the `Key` structure throughout the code and ensure it is used correctly. We have simply provided this recommendation as a potential suggestion for defence-in-depth coding. This is particularly true in that most memory wipe implementations are best-effort, rather than guaranteed.

3.4.2 Unnecessary code duplication

Severity: informational

Description

The following two implementations are almost functionally equivalent:

```
//does a * P where a is a scalar and P is an arbitrary point
void scalarmultKey(key & aP, const key &P, const key &a) {
    ge_p3 A;
    ge_p2 R;
    CHECK_AND_ASSERT_THROW_MES_L1(ge_frombytes_vartime(&A, P.bytes) == 0, ...
    ge_scalarmult(&R, a.bytes, &A);
    ge_tobytes(aP.bytes, &R);
```

```
}

//does a * P where a is a scalar and P is an arbitrary point
key scalarmultKey(const key & P, const key & a) {
    ge_p3 A;
    ge_p2 R;
    CHECK_AND_ASSERT_THROW_MES_L1(ge_frombytes_vartime(&A, P.bytes) == 0, ...
    ge_scalarmult(&R, a.bytes, &A);
    key aP;
    ge_tobytes(aP.bytes, &R);
    return aP;
}
```

Recommendations

C++11 compilers are capable of avoiding any copy/move constructors and can construct returned objects directly in place. See Named Return Value Optimization in the C++ standard.

If you wish to aim for compatibility with compilers that may not offer this optimization, then the former implementation ensures that the object is constructed by referencing source memory.

You may wish to `inline` these implementations as necessary, as well.