Alice Hierarchical Threshold Signature Security Audit

Final Report, 2020-05-19

FOR PUBLIC RELEASE

Contents

1 Summary

AMIS is building Alice, a hierarchical threshold signature scheme (HTSS) wallet for cryptocurrency written in Golang. This solution combines Tassa's hierarchical secret sharing with an optimized version of the well-known Gennaro-Goldfeder multiparty threshold ECDSA scheme. In this optimization, range proofs (a major bottleneck in the original scheme) can be avoided thanks to the replacement of the factorization-based Paillier encryption scheme with the discrete logarithm-based Castagnos-Laguillauime (CL) encryption scheme, therefore avoiding the mismatch between the ECDSA modulus and the Paillier modulus. Compared to other threshold signature scheme solutions, different levels of authorization are possible in Alice, where holders of the secret shares can have different "weights" in respect to the ability to generate a valid signature.

AMIS hired Kudelski Security to perform a security assessment of Alice, providing access to source code and documentation.
The repository concerned is: `https://github.com/getamis/alice`
we specifically audited commit `a648212059ef56f12d8c5579ce80879ef5c713e8`.

This document reports the security issues identified and our mitigation recommendations, as well as some observations regarding the code base and general code safety. A "Status" section reports the feedback from AMIS's developers, and includes a reference to the patches related to the reported issues. All changes have been reviewed by our team according to our usual audit methodology.

We report:

- 2 security issues of medium severity
- 8 security issues of low severity
- 13 observations related to general code safety

The audit was performed by Dr. Tommaso Gagliardoni, Cryptography Expert, with support of Dr. Jean-Philippe Aumasson, VP of Technology.

2 Methodology

In this code audit, we performed four main tasks:

1. informal security analysis of the original protocol;
2. actual code review with code safety issues in mind;
3. assessment of the cryptographic primitives used;
4. compliance of the code with the protocol description.

This was done in a static way and no dynamic analysis has been performed on the codebase. We discuss more in detail our methodology in the following sections.

2.1 Protocol Security

We analyzed the protocol in view of the claimed goals and use cases, and we inspected the original protocol description, looking for possible attack scenarios. We focused on the following aspects:

- possible threat scenarios;
- necessary trust assumptions between involved parties;
- edge cases and resistance to protocol misuse.

2.2 Code Safety

We analyzed the provided code, and we checked for things such as:

- general code safety and susceptibility to known vulnerabilities;
- poor coding practices and unsafe behavior;

- leakage of secrets or other sensitive data through memory mismanagement;
- susceptibility to misuse and system errors;
- error management and logging;
- constant-timeness when relevant;
- safety against malformed or malicious input from other network participants.

2.3 Cryptography

We analyzed the cryptographic primitives and subprotocols used, with particular emphasis on randomness and hash generation, signatures, key management, and encryption. We checked in particular:

- matching of the proper cryptographic primitives to the desired cryptographic functionality needed;
- security level of cryptographic primitives and of their respective parameters (key lengths, etc.);
- proper implementation of the cryptographic primitives, and compliance with their standard specification;
- safety of the randomness generation in the general case and in case of failure;
- safety of key management;
- assessment of proper security definitions and compliance to the use cases;
- checking for known vulnerabilities in the primitives used.

2.4 Protocol Specification Matching

We analyzed the provided documentation, and checked that the code matches the specification. We checked for things such as:

- proper implementation of the protocol phases;
- proper error handling;
- adherence to the protocol logical description.

2.5 Notes

It is important to notice that, although we did our best in our analysis, no code audit assessment is per se guarantee of absence of vulnerabilities. Our effort was constrained by resource and time limits, and in the scope of the agreement between AMIS and Kudelski Security.

In assessing the severity of some of the findings we identified, we kept in mind both the ease of exploitability and the potential damage caused by an exploit. Since Alice is a library, we ranked some of these vulnerabilities potentially higher than usual, as we expect the code to be reused across different applications with different input sanitization and parameters.

Regarding constant-timeness of code: according to AMIS that is an "ideal goal". We acknowledge that achieving constant-time cryptographic code in a distributed protocol like Alice and using a language such as Golang is a difficult task. We reported the most obvious code locations that perform non-constant-time cryptographic operations as observations, but we only reported them as findings where we believe there is an actual potential for abuse.

Correct memory management is left to Golang and was therefore not in scope, although we checked that all the run tests passed. Zeroization of secret values from memory is also not enforceable at a low level in a language such as Golang.

Part of the code is adapted from a C library for performing algebra on binary quadratic forms (https://github.com/maxwellsayles/libqform/blob/master/mpz_qform.c) We checked this part of the code for unsafe behavior and common programming pitfalls, but not for mathematical correctness. We believe the risk for exploitability in this part of the code to be low, as in Alice all the input values are sanitized before being processed, and it is a task of the unit tests to catch any misbehavior in the math.

Finally, we notice that the whole Alice protocol is based on the combination of two academic works: Tassa's hierarchical secret sharing and Gennaro-Goldfeder threshold ECDSA. The Alice protocol itself does not come together with a formal cryptographic security proof. AMIS has provided us with a preprint of an academic paper that is currently pending peer review in this respect, and which was not part of our analysis.

3 Findings

This section reports security issues found during the audit.

The “Status” section includes feedback from the developers received after delivering our draft report.

KS-AMIS-F-01: Hash of Multiple Inputs Without Domain Separation

Severity: Medium

Description

In `hash.go` the function `getDigest` computes the (keyed) hash function Blake2 of a byte array `originData` using another byte array `salt` as cryptographic salt.

```
114         checkData := append(originData, salt...)
115         digest := blake2b256.Sum(checkData)
```

However, there is no check on the size of `salt` and no domain separation between salt and data. This would allow to call the function maliciously and generate collisions, because, e.g., the hash of `(foo , bar)` and `(fo , obar)` would be the same.

Recommendation

Either enforce salt of a fixed size (e.g., 128 bits) or use a domain separator between salt and data.

Status

This has been fixed in commit `d1e3b590fdff6690b05620cdb49f6fb24152309a`.

KS-AMIS-F-02: Modulo Bias in `HashProtos`

Severity: Low

Description

In `utils.go` the function `HashProtos` takes as input an array (slice) of arbitrary messages, packs them into a single string, feeds the string to Blake2 to get a hash, and then reduces this hash modulo the field order to obtain a field element. This introduces a modulo bias. Notice that the field order of secp256k1 is roughly 2^{32} smaller than 2^{256}. This difference is small, but by performing a modulo reduction the resulting distribution is *not* a uniform distribution with values in $[0, \texttt{fieldOrder} - 1]$ because the $\approx 2^{32}$ low-end elements appear with double frequency.

```
187 // HashProtos hashes a slice of message to a field.
188 func HashProtos(blake2bKey []byte, fieldOrder *big.Int, msgs ...proto.Message)
    ↪  (*big.Int, error) {
189         blake2b256, err := blake2b.New256(blake2bKey)
190         if err != nil {
191                 return nil, err
192         }
193
194         // hash message
195         hMsg := &Hash{
196                 Msgs: make([]*any.Any, len(msgs)),
197         }
198         for i, m := range msgs {
199                 anyMsg, err := ptypes.MarshalAny(m)
200                 if err != nil {
201                         return nil, err
202                 }
203                 hMsg.Msgs[i] = anyMsg
204         }
205         inputData, err := proto.Marshal(hMsg)
206         if err != nil {
207                 return nil, err
208         }
209         c := new(big.Int).SetBytes(blake2b256.Sum(inputData))
210         c = new(big.Int).Mod(c, fieldOrder)
211         return c, nil
212 }
```

This function per se does not enforce uniformly distributed outputs, so this is not a vulnerability per se here. However, as a result `HashProtos` is not uniform because of the modulo reduction, and this is used in many other places in the code with this

purpose, for example as a random oracle implementation in the Fiat-Shamir euristics. Although we believe this is a minor concern in terms of exploitability, at a very minimum it breaks the cryptographic security proof.

Recommendation

We recommend performing rejection sampling on `blake2bKey` when `HashProtos` is used to generate a uniform randomness.

Status

This has been fixed in commit `4782e7ed9e9efb1ed345e2d6bd251ff13ba9edb5`.

KS-AMIS-F-03: Timing Leakage in `ComputeLinearCombinationPoint`

Severity: Low

Description

In `ecgrouplaw.go` the function `ComputeLinearCombinationPoint` at line 54 can skip cycles if an element is zero.

```
52          for i := 0; i < len(scalar); i++ {
53                  if scalar[i].Sign() == 0 {
54                          continue
55                  }
```

This is done to save on performance but it can leak information on the number of non-zero coefficients of the linear combination.

Recommendation

Given that this function is potentially used in scenarios where user identities and/or capabilities are mapped to certain monomials, and given the arguably negligible saving in terms of performance, we recommend avoiding this check in order to not risk the leakage of private information.

Status

This has been fixed in commit `a8b30006df1a0e263604201c0c357f2f47044a60`.

KS-AMIS-F-04: Missing Bounds in Creation of Matrix Transpose

Severity: Low

Description

In `matrix.go` the function `Transpose` does not perform checks on the number of rows and columns.

```
182 func (m *Matrix) Transpose() *Matrix {
183         transposeMatrix := make([][]*big.Int, m.numberColumn)
184         for i := uint64(0); i < m.numberColumn; i++ {
185                 tempSlice := make([]*big.Int, m.numberRow)
```

This might cause a huge memory allocation if called with large inputs.

Recommendation

It would be better to check that both `numberColumn` and `numberRow` are positive integers below a certain limit before allocating memory for the transpose.

Status

This has been fixed in commit `8aa042a8a39b59a6e6ae85007cc5854730e6d997`.

KS-AMIS-F-05: Missing Checks in CL Parameters Generation

Severity: Low

Description

In `cl.go` the function `NewCL` generates new CL encryption parameters.

```
91 func NewCL(c *big.Int, d uint32, p *big.Int, safeParameter int, distributionDistance
   ↪  uint) (*CL, error) {
92         // 1. Ensure lambda >= mu + 2
93         lambda := safeParameter / 2
94         mu := p.BitLen()
95         if lambda < mu+2 {
96                 return nil, ErrSmallSafeParameter
97         }
```

Algorithm 24 in the protocol specs document does not take a prime p as input as in the code above, but generates it internally instead. This is fine, however `NewCL` should perform a sanity check on p (primality and size) to avoid misbehavior. Moreover, there is no check enforced on the size of `safeParameter` nor derived values.

Recommendation

We recommend adding the security checks described above.

Status

This has been fixed in commit `918ddc0a808642fbdd5da65cf9696d69b8c24ff8`.

KS-AMIS-F-06: Paillier KeyGen Accepts Any Security Parameter

Severity: Low

Description

In `paillier.go` the function `NewPaillier` implements Algorithm 11 of the protocol specs document.

```
115 func NewPaillier(keySize int) (*Paillier, error) {
116         p, q, n, lambda, err := getNAndLambda(keySize)
117         if err != nil {
118                 return nil, err}
119         g, mu, err := getGAndMu(lambda, n)
120         if err != nil {
121                 return nil, err
122         }
```

However, `keySize` is not checked. Extremely low and insecure values of keysize will thus be accepted.

Recommendation

We recommend enforcing a minimum keysize.

Status

This has been fixed in commit `c9668e4b81de00573c791142fb960c0ca73f4d2f`.

KS-AMIS-F-07: Missing Checks CL `MulConst`

Severity: Low

Description

In `cl.go` the function `MulConst` does not perform a validation of the correct form of the ciphertexts as from Algorithm 23 of the protocol specs document.

Recommendation

We recommend to check correctness of the CL ciphertext as from protocol specs.

Status

This check is actually implemented in the function `getBQs` in `crypto/homo.cl/message.go`.

KS-AMIS-F-08: Missing Checks in Factorization ZK Proof for Paillier

Severity: Medium

Description

In `integerfactorization.go` the function `Verify` does not perform important security checks to validate the zero-knowledge proof of knowledge of the ephemeral Paillier key. This follows the logic of Algorithm 27 in the protocol specifications document, where the error is also present.

The first missing check is about the minimal size of the modulus N. In `Verify()` it is correctly checked that the value of the proof is between 0 and $N-1$. However, the value of the public key modulus N is read directly from the proof sent by the prover, which might be maliciously chosen very small.

```
73 func (msg *IntegerFactorizationProofMessage) Verify() error {
74         publicKey := new(big.Int).SetBytes(msg.GetPublicKey())
75         proof := new(big.Int).SetBytes(msg.GetProof())
76         err := utils.InRange(proof, big0, publicKey)
```

A small modulus could be easily factorized by a malicious prover, so technically speaking this would *still* be a honest-verifier zero-knowledge proof of knowledge. But,

as a consequence, the scope of the proof is rather inconclusive, as the proof itself does not provide any security guarantee.

The second missing check is about binding the value of the challenge to the public key. In an (interactive) Σ-protocol the challenge is chosen at random by the verifier. In a non-interactive version it is necessary to make sure that the prover does not have control on the value of the challenge, otherwise he can easily forge a proof for a statement for which he does not possess a witness, by first generating a random proof, and then postselecting a challenge that makes that proof verify. This is why in heuristics such as Fiat-Shamir the challenge is computed as a hash of a bunch of values including the statement. This is done correctly in the Schnorr proof for the CL encryption public key, but not for the Paillier key. In fact, here the challenge is simply embedded in the proof sent by the prover, and the verifier blindly reads this value from the proof without additional checks. It is therefore easy for a malicious prover to forge a valid proof for any `publicKey`: just generate a random `proof` and compute `challenge` as `new(big.Int).Exp(proof, publicKey, publicKey)`.

```
80          challenge := new(big.Int).SetBytes(msg.GetChallenge())
81          err = utils.InRange(challenge, big2, publicKey)
82          if err != nil {
83                  return err
84          }
85          expected := new(big.Int).Exp(proof, publicKey, publicKey)
86          if expected.Cmp(challenge) != 0 {
87                  return ErrVerifyFailure
88          }
```

Proofs of knowledge of Paillier modulo factorization are used in the standard GG18 version of the distributed signature algorithm (Algorithms 35 and 36 of the architecture document) to authenticate each participant's commitment of the ephemeral key. We notice that Alice uses by default the CL homomorphic encryption scheme rather than Paillier, and in the case of CL the implementation of the zero-knowledge Schnorr proof does not suffer from this vulnerability. Moreover, even in the case of Paillier, it is not clear to us how this vulnerability might lead to an exploitable attack except maybe for a denial-of-service, but at a very minimum it breaks the cryptographic security proof, and might open up the scheme to further attacks where a malicious adversary causes honest parties to leak information about their private shares by initiating a signing round using a malformed ephemeral public key.

For these reasons we rate the above vulnerability as Medium severity.

Recommendation

We recommend adding the following security checks:

1. after line 74: check that `publicKey` has a suitable bitsize.
2. In both `NewIntegerFactorizationProofMessage` and `Verify`, compute `challenge` as a hash of the public key and a random value, and include the random value in the proof message rather than the challenge itself.
3. Rejection-sample the above random value until the challenge so computed lies in the interval $[1, N-1]$.

Status

This has been fixed in commit `903b2d421a7105614825136d9ca544723535e2a4`.

KS-AMIS-F-09: CL Encryption Can Create Invalid Ciphertexts

Severity: Low

Description

In `cl.go`, CL encryption is implemented as per Algorithm 20, but without checking the size of the input plaintext.

```
180 // Encrypt is used to encrypt message
181 func (publicKey *PublicKey) Encrypt(data []byte) ([]byte, error) {
182         // Pick r in {0, ..., A-1} randomly
183         r, err := utils.RandomInt(publicKey.a)
184         if err != nil {
185                 return nil, err
186         }
187         // Compute c1 = g^r
188         c1, err := publicKey.g.Exp(r)
189         if err != nil {
190                 return nil, err
191         }
192
193         // Compute c2 = f^m*h^r
194         message := new(big.Int).SetBytes(data)
195         messageMod := new(big.Int).Mod(message, publicKey.p)
```

```
196         c2, err := publicKey.f.Exp(messageMod)
197         if err != nil {
198                 return nil, err
199         }
200
201         // h^r
202         hPower, err := publicKey.h.Exp(r)
203         if err != nil {
204                 return nil, err
205         }
```

Notice how, if `data` encodes a too large plaintext, decryption of the produced ciphertext will fail.

Recommendation

As a sanity check it would be better to enforce that `data` is an array of suitable size to not be reduced modulo p otherwise decryption would fail in general.

Status

This has been fixed in commit `5bbf4d2b908f3335fe52d765a05db0ce42ca956d`.

KS-AMIS-F-10: Minimal Threshold Not Enforced

Severity: Low

Description

The `EnsureThreshold()` defined in `crypto/utils/utils.go` validates a threshold value as long as it is less than the number of participants:

```
117 // EnsureThreshold ensures the threshold should be smaller than or equal to n.
118 func EnsureThreshold(threshold uint32, n uint32) error {
119         if threshold > n {
120                 return ErrLargeThreshold
121         }
122         return nil
123 }
```

But in this way, if the threshold is controlled by a malicious party, then it can be effectively removed by setting it to one. Moreover, the invalid threshold value zero is not filtered by this test.

Recommendation

We recommend to enforce in addition a minimal threshold value.Such filter could for example be enforced by new DKG and new commit handlers.

Status

This has been fixed in commit `8bb0e204c27c9853ac3bb778f84f4ccccbf7e7cb`.

4 Observations

This section reports various observations that are not security issues to be fixed, such as improvement or defense-in-depth suggestions.

KS-AMIS-O-01: Salt Size Unnecessary Large in Hash Commitments

In `hash.go` it is enforced that the length of the salt is at least as that of the data to be hashed. This seems excessive, a minimum bound on the salt size should be enough.

KS-AMIS-O-02: Potentially Redundant GCD Check in Paillier Key Generation

In the function `getNAndLambda` (Algorithm 16) two large primes p and q are generated and then it is checked that $GCD(pq, (p-1)(q-1)) = 1$. This is done also in `paillier.go` line 221. This check is redundant in general: if p and q are two large primes of equal bitsize then it will always be $GCD(pq, (p-1)(q-1)) = 1$.

The issue is that the prime generation is actually probabilistic: it might happen that composite numbers are mistakenly checked as primes. In this sense the GCD test above could be intended as an additional sanity check. However, notice the following:

1. the probability of a randomly generated large composite number to pass the primality test in `util.go` is less than 2^{-40} (20 rounds of Miller-Rabin and Lucas checks as from Golang's `math/big/prime.go` .)
2. If a composite number p or q (or both) is found that mistakenly passes the above test, we do not see a valid reason why necessarily $GCD(pq, (p-1)(q-1)) \neq 1$, i.e., it is not clear that the GCD check above will catch the error.

3. Either checks do not offer any protection against adversarially generated numbers, as from `math/big/prime.go` :

```
21 // ProbablyPrime is not suitable for judging primes that an adversary may
22 // have crafted to fool the test.
```

Given the above, one might consider the above GCD check redundant. However, the performance impact is arguably low.

KS-AMIS-O-03: Potential Modulo Bias if `NewPolynomial` is Misused

In `polynomial.go` the function `NewPolynomial` takes as input a slice of random elements and then all of them are reduced modulo `fieldOrder`.

```
44         for i, c := range coefficients {
45                 mc[i] = new(big.Int).Mod(c, fieldOrder)
46         }
```

This behavior in this context is OK. However, it might be a good idea to include a comment warning the user that this function can introduce modulo bias if applied directly to slices that have not previously undergone rejection sampling. For example, the function `RandomPolynomial` does this correctly by sampling uniform slices from the correct range.

KS-AMIS-O-04: Paillier Public Key Integrity Not Validated

One might expect that, when available, the Paillier ZK factorization proof be verified when encrypting (as a validation that the sender is not using a malformed public key), but this check is not performed. As a defense-in-depth mechanism to further protect the integrity of the `PublicKey` structure, one could change it to a private struct, so that only the caller can construct the object by the method defined.

KS-AMIS-O-05: Arguments in Schnorr Proof Hashed in Different Order From Specs

In `schnorr.go`, the challenge `c` is computed as $H(G, V, R, \alpha)$ instead of $H(G, \alpha, V, R)$ as from protocol specification (Algorithm 29, pag. 15).

```
116         c, err := utils.HashProtos(blake2bKey, fieldOrder, msgG, msgV, msgR, msgAlpha)
117         if err != nil {
118                 return nil, err
```

This is not a vulnerability per se but we always suggest to closely match the specs. Same inversion happens in `Verify` at line 197.

KS-AMIS-O-06: Misleading Function Name `computeRangeProof`

In `paillier.go`, the function `computeRangeProof(n)` actually computes $(n-1)^2$. Even if the function is private, this might sound misleading, also because range proofs are not used in the current Alice scheme.

KS-AMIS-O-07: Time Leak in `Differentiate`

In `polynomial.go`, the function `Differentiate` first checks if the order of the derivative is greater or equal to the degree of the polynomial, and if so returns the zero polynomial. This is clearly not constant-time as it can leak the degree of the polynomial, which is not a problem in this particular application as the degree is known, but it might be in general.

KS-AMIS-O-08: Misleading or Misplaced Comments in the Code

In `matrix.go` the following comment seems out of place:

```
171 // get gets the element at (i, j) without checking the index range and ensure [...]
172 func (m *Matrix) modInverse(i, j uint64) *big.Int {
```

In `cl.go` the following comment seems out of place:

```
515 // generateAbsDiscriminantK returns -DeltaK and the root of DeltaK
516 func generateAnotherPrimeQ(p *big.Int, bitsQ int) (*big.Int, error) {
```

In `utils.go` the following comment does not fit the code:

```
62 // EnsureRank ensures the rank should be smaller than threshold.
63 func EnsureRank(rank uint32, threshold uint32) error {
64         if rank+1 >= threshold {
```

KS-AMIS-O-09: Time Leak in Low-Level Algebraic Functions

Although we do not consider them to be vulnerabilities given the scope of the audit, there are many points in the code where the implemenetation of certain functions leaks information about the input through timing side channels. This happens in particular in `matrix.go` and `birkhoffinterpolation.go`. Some examples:

- in `matrix.go`: lines 258, 337, 414, 502, 566.
- In `birkhoffinterpolation.go`: lines 79, 82, 102, 138, 164.

KS-AMIS-O-10: Errors in the Protocol Specs Document

In the document 'Framework of Threshold Signature Scheme' dated March 23 2020 by AMIS, detailing the protocol architecture specs, we have found the following errors.

- in Algorithm 19, line 1. it is referenced "Generate parameters (p, q, A, g, f, G, F) by Algorithm 31." but should be Algorithm 24 instead. Moreover, h is missing in the output.
- In Algorithm 31, step 3. the values A and C are mistakenly not included in the hash computation, although they correctly appear in the code. Moreover, the notation for the plaintext message should be a, not p.
- In Algorithm 20, step 3. it is not immediately clear that $f^m h^r$ refers to composition of quadratic forms. We recommend to expand and clarify the notation.
- General typo: it's not Pederson but Pedersen.

KS-AMIS-O-11: Errors in the Scientific Paper

We also report some mistakes we identified in the scientific preprint "Threshold ECDSA Signature with Partial Accountability" authored by Chih-Yun Chuang and Chang-Wu Chen, describing the protocol.

- Pag. 6, sec. 2.2.: $L(u)$ should be a $\max$ not a $\min$.
- Pag. 7, Algorithm 3: inputs of Alice and Bob should be a and b resp, not α and β.
- Pag. 7, Algorithm 3, step 2: there is ambiguity of notation between operations in the algebraic field and in the Paillier homomorphic sense. In particular, Bob should perform an (integer) multiplication between ciphertexts, not an addition.

KS-AMIS-O-12: DBNS `Mod3` Routine Could Be Improved

In `dbns.go` the two initial loops could be unified (except for the first index case `i=0` that could be unrolled outside) to save on performance:

```
77 // This is a algorithm to get number % 3. The velocity of this function [...]
78 func fastMod3(number *big.Int) int {
79         numberOne, numberTwo := 0, 0
80         for i := 0; i < number.BitLen(); i = i + 2 {
81                 if number.Bit(i) != 0 {
82                         numberOne++
83                 }
84         }
85         for i := 1; i < number.BitLen(); i = i + 2 {
86                 if number.Bit(i) != 0 {
87                         numberTwo++
88                 }
89         }
```

KS-AMIS-O-13: Misleading Notation in MtA `Compute`

In `mta.go` we observe the following comment (that also matches the code):

```
117 // Compute gets the encrypted k with a random beta
118 // alpha = (E(encMessage) * a) + E(beta), where * is [...]
119 func (m *mta) Compute(publicKey homo.Pubkey, encMessage []byte) (*big.Int, *big.Int,
    ↪  error) {
```

Looking at the protocol spec documentation, Algorithms 37 and 38, it seems that the function above should rather compute $c = (E(\texttt{encMessage}) * b) + E(\beta)$. We understand that this is a specular situation from each participant's point of view, but we suggest changing the names of the variables involved to make it easier to follow the specs.

5 About

Kudelski Security is an innovative, independent Swiss provider of tailored cyber and media security solutions to enterprises and public sector institutions. Our team of security experts delivers end-to-end consulting, technology, managed services, and threat intelligence to help organizations build and run successful security programs. Our global reach and cyber solutions focus is reinforced by key international partnerships.

Kudelski Security is a division of Kudelski Group. For more information, please visit https://www.kudelskisecurity.com or https://kudelski-blockchain.com/.

Kudelski Security
route de Genève, 22-24
1033 Cheseaux-sur-Lausanne
Switzerland