Axelar

Audit of Threshold Signature Library

Adrian Hamelink, Lúcás Meier

20211103

Contents

1

Introduction

Axelar requested that we audit their implementation of the [GG20] protocol for threshold ECDSA signatures, in their open-source `tofn` library. They also requested we look over their `tofnd` library, providing a gRPC wrapper over this protocol, and their fork of the `paillier-rs` library, providing an implementation of the Paillier encryption system for the protocol.

We audited the following commits of these libraries:

- axelarnetwork/tofn 8dabe9354c67ca3a9753c83719e95231908eb4e9
- axelarnetwork/tofnd c9846f20386364cd77dc87df748856aa56a3783a
- axelarnetwork/paillier-rs 2d965b16d89de6f5d15b054fd5874d0c017c4747

We reviewed the correctness and safety of their implementation, paying particular attention to any deviations from the original protocol as specified in [GG20], and that the wrapper in `tofnd` did not introduce any security flaws. Additional focus on zeroizing secrets in memory was also requested. Side-channel vulnerabilities, such as timing-leaks, were considered out of scope for this audit.

We only found a handful of minor issues related to zeroizing secrets, as well as an issue related to potential replayability of Zero-Knowledge proofs. We also found a handful of technical deviations from the protocol, with negligible impact on security.

As a whole, we noticed that the implementation was of high quality. In particular, the implementation goes out of its way to test failure paths of the protocol, and shows a deep understanding of the [GG20] protocol by the authors.

We thank Axelar for trusting us with this review.

2

Library Overview

In this section, we provide some background on what the [GG20] protocol aims to achieve, and how this collection of libraries implements the protocol. In particular, we focus on what assumptions underpin this implementation, and how ambiguities in the original protocol have been resolved.

2.1 Distributed Key Generation

The distributed key generation protocol allows a consortium of parties to generate shares of a private signing key, so that a quorum of parties can sign future messages, without any individual party knowing this private key. In the [GG20] protocol, this phase is also used to generate auxiliary information needed in the subsequent signing phase: namely, a Paillier encryption key-pair and trusted setup for integer commitments.

A key difference between the implementation and the [GG20] protocol is that a Paillier key (and associated ZK proofs of integrity) are passed into an instance of the key generation phase. A single Paillier key can be used for different key generation phases. The motivation is to avoid repeating the expensive key generation step. This has a negligible impact on security, aside from a potential weakening of the key through many uses.

Unfortunately, the protocol [GG20] does not provide many implementation details about this phase of the protocol. In particular, the reader is left to augment their implementation with the following:

- A zero-knowledge proof for the correctness of Paillier keys.
- A proof of validity for the commitment scheme setup.
- A mechanism for detecting misbehaving parties during the last phase.

The first is a proof that the factorizations of the Paillier modulus does not contain any square factor, and was implemented as specified in [Gol+18]. Due to a recent attack [Vel21] on the correctness the commitment setup, we were asked to focus some of our effort on this specific aspect of our protocol and provide an analysis in Section 2.5. We note that these additional steps are also present in other implementations of the protocol, such as ZenGo/multi-party-ecdsa

Augmenting the key-generation with identifiable aborts is rather straightforward since all failures are attributable (a failing zero-knowledge proof, decommitment or secret sharing verification). We provide an overview of how these cases might be handled in Section 2.4. A final communication round was included in this implementation to allows the parties to confirm that all parties successfully completed the protocol, thwarting a *Forget-And-Forgive* type of attack from [AS20].

These various failure paths are also tested in the implementation, by inserting the possibility of corrupting certain variables at different points in the protocol.

2.2 Share Recovery

The implementation provides two ways of storing and recovering the shares.

The first is to store the shares inside of an encrypted key-value store. This store is protected through a password, using adequate password hashing (scrypt) and authenticated encryption (XChaCha20Poly1305).

The second is to encrypt the share using the party's Paillier key. This key, in turn, can be recovered from a mnemonic phrase, similar to how cryptocurrency wallets are implemented. This mnemonic maps directly to a seed. This seed is then feed into a secure PRG (HMAC-SHA256) along with context information identifying the party, and the key, and used as the randomness in the Paillier key generation algorithm. This will repeat the prime number sampling process, but with deterministic decisions guided by this seed.

This allows recovering a share using the public recovery information, and the private mnemonic.

2.3 Threshold Signing

After performing the key-generation protocol among n different parties with threshold parameter t, a consortium of $t + 1$ of these participants can collectively generate a signature for the key returned by the previous protocol. In this specific implementation, any node may be given multiple shares of the signing key, essentially giving it more votes when deciding what messages to sign. Under the hood, the node will run multiple instances of the protocol for each *share* it controls.

The [GG20] protocol improves upon [GG18] by reducing the number of rounds down to 7, and adds the ability to identify misbehaving parties. The relatively high round complexity is mainly due to the non-linearity of ECDSA signatures, which require the inversion of the nonce k. This is in part handled by the Multiplicative-to-Additive share conversion protocol (see Section 2.5 for more details), which relies on the additively homomorphic properties of the Paillier cryptosystem [Pai99]. Moreover, the *dishonest majority* assumption, whereby it is assumed that up to t participants are malicious, enforces the use zero-knowledge proofs in most rounds. Overall, this provides an attacker with ample opportunities to cause the protocol to fail during a signature generation.

The protocol implementation is structured in a way that clearly separates each different possible state the parties can reach, therefore making is easier to handle all possible failure scenarios.

We also note that the authors of the library discovered a new attack vector (Issue #154) with the [GG20] protocol, whereby a fatal crash can occur when an adversary reveals a malicious input δ_i that would cause the other parties to perform a field inversion of 0.

We found that the library closely followed the specifications detailed in [GG20], and did not find any issues relating to the protocol logic or the handling of fault attribution.

2.4 Identifiable Aborts

Identifiable aborts is an important feature in the [GG20] protocol, as its enables more robust deployments in decentralized environments.

When the execution of a protocol fails due to possibly malicious behavior from certain users, honest participants will all agree on which users are responsible for this failure. An external mechanism can then either punish misbehavers, or decide to exclude them from future protocol executions. If these decisions are automated, the fault attribution must be carefully implemented to ensure that no honest participant is wrongfully accused, and that the decision is unanimous.

During this security review, we payed careful attention to makes sure this mechanism respected both of these properties, and provide a high-level overview of how it is implemented across `tofn` and `tofnd`.

An important requirement for identifiable aborts is that all messages sent by parties during the protocol execution be se sent over a *reliable broadcast channel* [CL14]. More specifically, it ensures that all parties agree on the set of all messages sent, including P2P. This assumption is described in the README of `tofn`, since without it, a malicious party would be able to cause false accusals, or prevent selected parties from delivering the final result. As described in White Paper [Axe21], each node in the network has access to a consensus layer over which all messages are posted, and thus the messages delivered to `tofnd` will all be the same. Moreover, it also ensure the authenticity of the message sender.

In order to guarantee agreement of the set of faulters, it is necessary that all parties perform the same checks in the same order. In some cases, this means that some faulters will not be blamed, but this is a lot more desirable than risking a false accusation.

Whenever a new message is delivered, the protocol first ensures its basic correctness (sender ID, proper deserialization, duplicate, etc.). If any of these checks do not pass, the failure is noted and the process continues until messages from all parties have been delivered, or the client aborts (possibly indicating a timeout from one of the peers). Since it is assumed that all parties will have received the same set of messages, they would all detect the same set of invalid ones, and can thus abort the protocol and return the list of faulters to the user. Note that in case of a timeout, missing messages will be detected later on at the start of each round.

At this point, the protocol executes the current round by feeding it all the messages it has received. All rounds check that the messages are of the correct type, and will abort if it is not the case. This includes checking whether other parties may have erroneously advanced to an incorrect state, perhaps due to an incorrect verification process in the previous round.

In some rounds, each peer receives a “personalized” zero-knowledge proof (this is the case for rounds which include the MtA protocol described below). While the client still receives all proofs addressed to all all other peers, it only checks its own proofs for efficiency reasons. If one of these personalized proofs fails, then it knows that the sender is malicious, but that peer could have sent valid proofs to all other parties. The victim must therefore alert the other parties of this failure, by sending this peer’s ID to everyone. In the next round, if any party detects an accusations, it must also switch to the sad path and confirms all accusations, by

verifying the relevant messages exchanged between peers in the previous round. If any false accusation was detected, the accuser is blamed instead.

In the signature generation protocol, some rounds force the protocol to abort, but fault attribution requires additional information. Depending on which stage of the protocol this error occurs in, it is safe for parties to reveal ephemeral private values. All honest parties will therefore broadcast the necessary data which will allow them to reveal the possible many malicious parties.

Some rounds can fail for different reasons for different users. In order to guarantee agreement on *a single* set of faulters, the implementation specifies that the parties should verify claims from parties for which the failure would be detected first. Indeed, if a party receives an incorrect personalized message from a peer which it cannot interpret, then it has no way of advancing beyond that point. Other peers take this into account in order to not falsely accuse a victim, and will therefore check accusations for that specific type of error only. Other claims are ignored, as the victim may not be able to verify them and this would lead to non-agreement of the faulters.

During our review, we found that the library authors had carefully considered and implemented all of the failure paths. The authors also included tests specifically stressing these failure paths.

2.5 Paillier Range Proofs and Zero-Knowledge Setup

The additively-homomorphic property of the Paillier cryptosystem [Pai99] enables efficient multiplication of two additively shared secrets, which is commonly referred to as a *Multiplicative-to-Additive* (MtA) protocol. More specifically, two parties Alice and Bob who respectively know secrets $a, b \in \mathbb{F}_q$ can run this protocol to obtain two new secret α, β such that $a \cdot b = \alpha + \beta \bmod q$.

In `tofn`, this process is handled by src/crypto_tools/mta.rs.

- In the first round, Alice sends to Bob an encryption of a under her own public key $C_a \leftarrow \mathrm{Enc}_A(a)$, along with a zero-knowledge *range proof* π_a which proves that the plaintext a is in the range $[-q^3, q^3]$.
- After verifying the proof π_a, Bob samples $\beta' \in \mathbb{Z}_{N_A}$ and computes $C_\alpha \leftarrow \mathrm{Enc}_A(\beta') \oplus (b \odot C_a)$ which should be equal the an encryption of $\alpha = a \cdot b + \beta' \bmod N_A$. Bob also include a zero-knowledge proof stating that the computation was of C_α is consistent with his secrets b, β' and Alice's ciphertext C_a. He send (C_α, π_b) to Alice and set his share $\beta \leftarrow (-\beta' \bmod N_A) \bmod q$.
- Alice now verifies the proof π_b and if it passes, decrypts $\alpha \leftarrow \mathrm{Dec}_A(C_\alpha) \bmod q$.

The range proofs π_a, π_b are essential for guaranteeing that both the confidentiality of the secret shares, as well as ensuring the correct computation of signatures. While the original iteration [GG18] of the protocol describes that these proofs may be omitted, recent attacks [TS21] have shown that it is in fact possible to extract the secrets. We therefore spent considerable time verifying the correctness of these protocols, including input validation and comparing the the code with the original specifications.

A distinctive feature of both of these range proofs is the dependency on a *trusted setup* used for Fujisaki-Okamoto [FO97] commitments over the integers $\mathbb{Z}$. The trusted setup is defined as a triple $(\tilde{N}, h_1, h_2)$ such that $\tilde{N}$ is the product of two safe primes p, q, and $h_1 = h_2^s \bmod \tilde{N}$,

for some secret exponent s, both h_1, h_2 are generators of a multiplicative subgroup of order $\phi(\tilde{N}) = (p-1)(q-1)$. A commitment to $x \in \mathbb{Z}$ with randomness $r \in \mathbb{Z}$ is then defined as $c \leftarrow h_1^x h_2^r \bmod \tilde{N}$.

The secret s can be thought of as a *trapdoor* which can be used to create fake opening proofs. For this reason, no party should trust proofs where the parameters were generated by anyone else but them. In order to circumvent the need for a trusted 3rd party who would not reveal s, each party generates their own set of parameters $(\tilde{N}_i, h_{i,1}, h_{i,2})$. This means the each range proof can only be deemed valid by the party who's parameters were used, though all parties can assert that an incorrect proof fails. Since the protocol assumes that all but one party may be malicious, it should be assumed that all other parties are aware of each other's trapdoors, and are able to create fake range proofs amongst themselves.

When an honest party is creating a range proof for another party however, they want to prevent the receiver from learning any information about their secrets. This can happen in cases where the parameters $(\tilde{N}_i, h_{i,1}, h_{i,2})$ were maliciously generated, as shown in a recent attack [Vel21] on existing protocol implementations.

During the key generation protocol, each party sends their set of public parameters, along with a zero-knowledge proof stating the $h_1 \in \langle h_2 \rangle \wedge h_2 \in \langle h_1 \rangle$. The proof for both statements is taken from [Poi00]. The statistical zero-knowledge property guarantees that the proof does not leak any information about s which would otherwise invalidate all ZK proofs created using these parameters. Thanks to soundness, the verifier in turn is confident that their commitments are statistically hiding. Indeed, if the verifier receives the following commitment $c = \mathrm{Commit}_{(\tilde{N},h_1,h_2)}(x;r) = h_1^x h_2^r \bmod \tilde{N}$ for x, then it could solve for the discrete log over the base h_1 to get $y = x + sr \bmod \phi(\tilde{N})$. Even with knowledge of s, the distribution of y statistically close to the uniform distribution over $\mathbb{Z}_{\phi(\tilde{N})}$, as long as r is sampled from a sufficiently large set.

During our review, we verified that all parameters were correctly sampled in both range proofs of the MtA protocol. After becoming aware of the attack in [**FS21**], we noticed that the random values `tau` and `gamma` were sampled from $\mathbb{Z}_{q\tilde{N}}$ and $\mathbb{Z}_N$, where N is Alice's Paillier modulus. Instead, [**FS21**] recommends sampling these from $\mathbb{Z}_{q^3\tilde{N}}$ and $\mathbb{Z}_{q^2N}$, respectively. Due to proper verification of all other parties' Paillier encryption keys in src/crypto_tools/paillier/zk/paillier_key.rs L157, the described attack is not applicable.

On the necessity of safe-primes for the zero-knowledge setup

It is not completely clear whether the prime factors of $\tilde{N}$ need to be safe, or whether it suffices that they contain a large enough multiplicative subgroup. While the original commitment scheme construction [FO97] relied on safe primes, it was later improved in [DF02] to allow for strong primes as well. However, we cannot assert with full confidence that the security proofs from [GG20] would remain valid when only strong primes are used. For the trusted setup, the security proof [Poi00] only assumes that $\tilde{N}$ is a product of strong primes and would therefore still apply.

We also point out that a similar protocol for threshold signature [CMP20] [Can+20] follows a similar approach to that of [GG20], but takes $h_1, h_2 \in \mathrm{QR}(\tilde{N})$, which slightly affects the security proofs.

3

Security Issues

3.1 Potential Replayability of ZK Proofs

Severity: low

Description

The protocol makes use of Non-Interactive Zero-Knowledge (NIZK) proofs, which hash in inputs of the proof in order to generate challenges. To generate different challenges for different instantiations of the proof, information from the ambient context should be included inside of these hashes. Unfortunately, the contextual information provided inside of this implementation is minimal. Many proofs only include the ID of the party generating the proof as a context string. Because party IDs are merely indices $0..N-1$, it's easy to find a collision between party IDs between different consortiums running the protocol. This would allow reusing a NIZK proof from one execution inside of a different execution. This is merely a realistic scenario of how this could be exploited. Other possibilities for reuse also exist.

While this is concerning, the severity of this issue is marked as "low", because turning this issue into an actual exploit seems quite difficult.

Recommendation

More context should be added to the domain used when generating these proofs. For example, the number of parties, which round of the protocol, etc. In particular, if a random nonce was agreed upon by all parties when running the protocol (for either key generation, or signing), this would effectively prevent reusing proofs from one execution of the protocol in another, since that execution would in all likelihood have a difference nonce. Note that using a sequential nonce inside of a consortium may not be sufficient, because proofs may be reused between different consortiums running in parallel. Given the strong assumptions around reliable broadcast, and the byzantine consensus afforded by a blockchain, we believe that agreeing upon a random session nonce would not be an onerous requirement.

3.2 Variables not zeroized in Paillier key-generation

Severity: low

Description

In the `with_safe_primes_unchecked` function (paillier-rs/src/decryptionkey.rs#L74) the information in a Paillier key is created from two prime numbers `p` and `q`. These values are zeroized after calling this function. Unfortunately, intermediate values derived from these primes are created inside this function. Because thes values are not included in the return value, they never get zeroized. These sensitive values are the following: `pm1`, `qm1`, and `tt`. An adversary observing any one of these residual values in memory would be able to factor N, thus breaking the Paillier system. That being said, the difficulty of recovering lingering values in memory makes this only a low severity issue.

Recommendation

Explicitly zeroizing these values, as done for `p` and `q` when calling this function, would prevent this issue.

A more robust approach would be to create a wrapper around the `BigNumber` type, which would automatically be zeroized when dropped.

3.3 Encryption key for KV database is not zeroized

Severity: low

Description

When initializing the encrypted key-value database, a ChaCha20 key is derived from a password. This key is zeroized after creating the `XChaCha20Poly1305` struct: tofnd/src/encrypted_sled/kv.rs#L53. Unfortunately, the struct itself contains a copy of the key, and is not zeroized. This means that the encryption key can potentially linger in memory.

Recommendation

The simplest fix would be to implement zeroization on drop for this struct. Unfortunately, because this struct comes from the `chacha20poly1305` crate, a patch or issue would need to be submitted to that repository. It would also be possible, in theory, to manually zeroize the struct by casting it to a byte pointer, or through other similar bitwise trickery.

4

Other Observations

4.1 Potential modulo bias in Paillier key proof

Description

In tofn/src/crypto_tools/paillier/zk/paillier_key.rs#L117, a challenge is generated modulo N. This is done by sampling l bits, with $N \approx 2^l$, and then reducing modulo N. In theory, this might introduce a certain amount of bias, although this is most likely not exploitable in this context.

Recommendation

That being said, the fix for this issue is so simple that we'd recommend it be applied. To make any bias negligible, it suffices to sample $l + 128$ bits instead (or, replacing 128 with whatever security parameter is desired). This ensures that any modulo bias is $\leq 2^{-128}$.

4.2 Paillier nonces are not checked to be units modulo N

Description

In paillier-rs/src/encryptionkey.rs#L76 a random nonce for encryption is generated in $\mathbb{Z}_N$. In principle, this should be generated in $\mathbb{Z}_N^*$. In practice, because $N = pq$, with p and q primes roughly half the size of N, the probability of a nonce in $\mathbb{Z}_N$ not being invertible is negligible, so an additional check would be superfluous.

That being said, in the `encrypt_with_randomness` method, a nonce is passed to the function as an argument. In this case, it might be a good idea to add a sanity check that `gcd(r, N) == 1`, in case the function is misused.

4.3 Paillier key size check is commented as unnecessary

Description

In tofn/src/crypto_tools/paillier/zk/composite_dlog.rs#L206, the size of the Paillier key is checked to be the expected size (2048 bits). A comment indicates that this check is done only for san-

ity, and isn't necessary. Unfortunately, as a recent attack [**FS21**] has shown, this size check is actually necessary.

The same issue applies to tofn/src/crypto_tools/paillier/zk/paillier_key.rs#L156.

Recommendation

This comment should be amended, so that future contributors or maintainers do not accidentally remove the check.

4.4 Paillier encryption does not accept 0

In paillier-rs/src/encryptionkey.rs#L71, the message m is checked to be in the range $[1..N-1]$. Strictly speaking, the message $m = 0$ should also be allowed. That being said, the difference is unlikely to matter in practice.

Bibliography

[GG20] R. Gennaro and S. Goldfeder. *One Round Threshold ECDSA with Identifiable Abort*. Cryptology ePrint Archive, Report 2020/540. https://ia.cr/2020/540. 2020.

[Gol+18] S. Goldberg et al. *Efficient Noninteractive Certification of RSA Moduli and Beyond*. Cryptology ePrint Archive, Report 2018/057. https://ia.cr/2018/057. 2018.

[Vel21] Velas Blockchain. *Counter-Strike: Threshold Attack*. Medium. https://medium.com/velasblockchain/counter-strike-threshold-attack-87f3b456b1e0. 2021.

[AS20] J.-P. Aumasson and O. Shlomovits. *Attacking Threshold Wallets*. Cryptology ePrint Archive, Report 2020/1052. https://ia.cr/2020/1052. 2020.

[GG18] R. Gennaro and S. Goldfeder. "Fast Multiparty Threshold ECDSA with Fast Trustless Setup". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS '18. Toronto, Canada: Association for Computing Machinery, 2018, pp. 1179–1194. ISBN: 9781450356930. DOI: 10.1145/3243734.3243859. URL: https://doi.org/10.1145/3243734.3243859.

[Pai99] P. Paillier. "Public-Key Cryptosystems Based on Composite Degree Residuosity Classes". In: *Advances in Cryptology — EUROCRYPT '99*. Ed. by J. Stern. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 223–238. ISBN: 978-3-540-48910-8.

[CL14] R. Cohen and Y. Lindell. "Fairness versus Guaranteed Output Delivery in Secure Multiparty Computation". In: *Advances in Cryptology – ASIACRYPT 2014*. Ed. by P. Sarkar and T. Iwata. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 466–485. ISBN: 978-3-662-45608-8.

[Axe21] Axelar. *Axelar Network: Connecting Applications with Blockchain Ecosystems*. White paper. https://axelar.network/wp-content/uploads/2021/07/axelar_whitepaper.pdf. 2021.

[TS21] D. Tymokhanov and O. Shlomovits. *Alpha-Rays: Key Extraction Attacks on Threshold ECDSA Implementations*. Preprint. 2021.

[FO97] E. Fujisaki and T. Okamoto. "Statistical zero knowledge protocols to prove modular polynomial relations". In: *Advances in Cryptology — CRYPTO '97*. Ed. by B. S. Kaliski. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 16–30. ISBN: 978-3-540-69528-8.

[Poi00] D. Pointcheval. "The Composite Discrete Logarithm and Secure Authentication". In: *Public Key Cryptography*. Ed. by H. Imai and Y. Zheng. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 113–128. ISBN: 978-3-540-46588-1.

[DF02] I. Damgård and E. Fujisaki. “A Statistically-Hiding Integer Commitment Scheme Based on Groups with Hidden Order”. In: *Advances in Cryptology — ASIACRYPT 2002*. Ed. by Y. Zheng. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 125–142. ISBN: 978-3-540-36178-7.

[CMP20] R. Canetti, N. Makriyannis, and U. Peled. *UC Non-Interactive, Proactive, Threshold ECDSA*. Cryptology ePrint Archive, Report 2020/492. https://ia.cr/2020/492. 2020.

[Can+20] R. Canetti et al. “UC Non-Interactive, Proactive, Threshold ECDSA with Identifiable Aborts”. In: *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’20. Virtual Event, USA: Association for Computing Machinery, 2020, pp. 1769–1787. ISBN: 9781450370899. DOI: 10.1145/3372297.3423367. URL: https://doi.org/10.1145/3372297.3423367.