Mysten Fastcrypto Pedersen DKG and tBLS Audit

Shresth Agrawal[1,2] Pyrros Chaidos[1,3]
Bernardo David[1,4]

[1] Common Prefix
[2] Technical University of Munich
[3] University of Athens
[4] IT University of Copenhagen

Dec 10, 2023
Last update: May 14, 2024

1 Overview

1.1 Introduction

Mysten Labs commissioned Common Prefix to audit the Pedersen DKG and tBLS implementations within their fastcrypto library. The primary objectives of the audit were to assess security, adherence to the relevant publications, performance optimizations, and code quality. fastcrypto is a Rust-based library that implements selected cryptographic primitives and also serves as a wrapper for several chosen cryptography crates, ensuring optimal performance and security for Mysten Labs' software solutions, including their blockchain network, Sui.

A Distributed Key Generation (DKG) protocol enables a set of users to cooperatively create a cryptographic key. In the case of the Pedersen DKG [Ped91] scheme, the resulting key is unpredictable as long as even a single contributor is honest. At the same time, the scheme distributes "shares" of the secret key across all users, so that only a large enough coalition is able to recover it. The corresponding public key on the other hand can be calculated with only public data. In this application, this key pair is used to implement a threshold version [Bol02] of the Boneh–Lynn–Shacham (BLS) signature scheme [BLS01]. This enables any set of users to produce a local signature using their share of the secret. A large enough set of local signatures can be used to compute a signature that verifies against the combined public key, without revealing the secret key.

The scope of this audit was limited to the fastcrypto implementation and did not extend to the library's dependencies or any downstream applications.

1.2 Audited Files

Audit start commit: [ea66012]
Latest audited commit: [4e43631]

1. fastcrypto-tbls/src/dkg.rs
2. fastcrypto-tbls/src/dl_verification.rs (except `verify_triplets`, `verify_deg_t_poly` and `verify_equal_exponents`)
3. fastcrypto-tbls/src/ecies.rs
4. fastcrypto-tbls/src/nizk.rs
5. fastcrypto-tbls/src/nodes.rs
6. fastcrypto-tbls/src/polynomial.rs
7. fastcrypto-tbls/src/random_oracle.rs
8. fastcrypto-tbls/src/tbls.rs
9. fastcrypto-tbls/src/types.rs

Supporting documentation:

1. Pedersen_s_DKG_for_tBLS.pdf
(SHA-256: b5c64f7124a74da548ef5c35ec2b024e3f18105945ca50f55433a13e8ad37c46) referred to as the specification document in the rest of the audit report.

1.3 Disclaimer

This audit does not give any warranties on the bug-free status of the given code, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. This audit report is intended to be used for discussion purposes only. We always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of the project.

The scope of the audit was constrained exclusively to the Fastcrypto wrapper code, with no examination conducted on its associated dependencies. Furthermore, the audit does not encompass any reference string generation functionality in terms of code or execution.

1.4 Executive Summary

The code implements Pedersen's DKG and tBLS protocol, employing high-quality cryptographic primitives and prioritizing performance optimizations. Overall, the implementation is of very high quality and follows Rust's best practices.

Pedersen's DKG is a multi-phase protocol requiring interaction with a totally ordered broadcast channel at each phase. The codebase is organized with each protocol phase encapsulated in a distinct function. The codebase has broad comments on the usage of these functions and the requirements assumed for the high-level protocol which integrates the implementation. Despite the specification explicitly mentioning the use of blockchains as the broadcast channel, the codebase lacks implementation or documentation on blockchain integration or interaction with the broadcast channel. This omission places significant dependency on the higher-level protocol to manage these interactions correctly for secure usage of the protocol.

The audit revealed certain assumptions about the usage of the implementation that were not clearly documented, potentially leading to protocol misexecution or susceptibility to denial-of-service attacks.

Another overarching concern is that many values are instantiated externally, which may lead to unsafe use. Similarly, several internal values may overflow if the setting involves a (significantly) higher number of stake and/or participants.

We propose using random oracle calls for randomness calculations (to ensure consistency across nodes) and to increase the length of randomizers. Finally, we recommend potential optimizations, minor refactoring, and highlight areas that need further documentation.

1.5 Findings Severity Breakdown

The findings are classified under the following severity categories according to the impact and the likelihood of an attack.

Level	Description
High	Logical errors or implementation bugs that are easily exploited. In the case of contracts, such issues can lead to any kind of loss of funds.
Medium	Issues that may break the intended logic, are deviations from the specification, or can lead to DoS attacks.
Low	Issues harder to exploit (exploitable with low probability), can lead to poor performance, clumsy logic, or seriously error-prone implementation.
Informational	Advisory comments and recommendations that could help make the codebase clearer, more readable, and easier to maintain.

2 Findings

2.1 High

None Found.

2.2 Medium

M01: Non unique messages can cause DOS attack in `process_message`.

Affected Code: fastcrypto-tbls/src/dkg.rs (line 306)

Summary: The `process_message` function is implemented such that the processing of multiple messages can be completely parallelized without relying on other messages that have been processed. This leads to a potential DOS attack where a single malicious party can send multiple messages, which will all be processed without checking for the uniqueness of the sender. In fact, the current implementation allows for this attack to be performed by a malicious party with zero weight.

Suggestion:

- Add additional pre-processing step that checks for the uniqueness of the sender or add additional assumption on the higher-level protocol to perform this check.
- Reject messages from parties with zero weight in `process_message`.

Status: Resolved [f658d44c]

M02: Async computation combined with `merge` and `process_confirmations` can lead to incorrect protocol execution.

Affected Code: fastcrypto-tbls/src/dkg.rs (lines 401,463)

Summary: It is unclear when the `merge` and `process_confirmations` message should be called by the higher-level protocol. On the protocol level, both of these functions should be called by all the parties exactly at the same first message when messages with sufficient weight have been accumulated. This has to be implemented carefully by the higher level protocol, specifically if the previous step has async computation (e.g., multiple `process_message` being executed asynchronously before merge). This could lead to race conditions where the messages that join the final set differ for different parties.

Suggestion: Add the above assumption of the higher-level protocol to the comments.

Status: Resolved [f658d44c]

M03: Check for `minimal_threshold` is lax.

Affected Code: fastcrypto-tbls/src/dkg.rs (line L471)

Summary: The check that `minimal_threshold` is at least `t` does not guarantee that signers will be able to form a quorum. The adversary can fill the first `t` slots, and then allow a single honest party with good shares. If we assume all other honest parties have bad shares, the adversary can always break liveness.

Suggestion: As the specification mentions, the threshold should be set to `t+f`.

Status: Resolved [f658d44c]

2.3 Low

L01: Unsafe typecasting from usize to u32.

Affected Code:

- fastcrypto-tbls/src/polynomial.rs (line 36)
- fastcrypto-tbls/src/dl_verification.rs (lines 53,82,115)

Summary: **usize** to **u32** type conversions might be unsafe if the code runs on a 64-bit machine. Max **u32** is not an astronomical number and can be overflowed when indexing a large number of items, even with each item taking up multiple bytes.

Suggestion: Either enforce that the maximum number of items in the array cannot be greater than `u32::MAX` or do not perform the type conversion.

Status: Resolved [3e7b3d88, f658d44c]

L02: Modification of magical constants can lead to overflow.

Affected Code: fastcrypto-tbls/src/nodes.rs (line L154)

Summary: The types are tightly tied to the magical constants in the code: 1000, which is used as an upper limit for the number of nodes, and 40, which is the upper bound for the reduction divisor. The sum in the reduce function is of type **u16** (maximum 65535). Based on the above constants and the code, the sum can be a maximum of 39000 (1000 users, with each, having the maximum possible remainder mod 40). This is safe, but if either of the above-mentioned constants is increased by 1.6x, the sum variable will overflow.

Suggestion: Define both the constants with type **u16** and the sum variable with type **u32**. The types will enforce that any value of the constant will not overflow the sum.

Status: Resolved [3e7b3d88]

L03: Type mismatch for `t` variable.

Affected Code: fastcrypto-tbls/src/nodes.rs (line 151)

Summary: The `t` parameter in the reduce function should be of type **u32**. It should be of the same type as the `total_weight`. Also, in `Party` **struct**, `t` is a **u32**.

Suggestion: Use the same type for `t` as `total_weight` consistently.

Status: Resolved [3e7b3d88]

L04: Inefficient iteration in `share_ids_of`.

Affected Code: fastcrypto-tbls/src/nodes.rs (line 124)

Summary: The `share_ids_of` function currently brute forces over all the potential share IDs and filters that of the given party ID. This operation requires O(`total_weight` * log(nodes)). This function is called over all the nodes in the `create_message` function.

Suggestion: As all the shares of the node are sequential, a cleaner approach would be to find the index of party ID in the `nodes_with_nonzero_weight` and then use that to get the range of share IDs from the `accumulated_weights`. This would be significantly cheaper as the complexity doesn't scale with the `total_weight`.

Status: Acknowledged

L05: Non-deterministic randomizers used for checks.

Affected Code: fastcrypto-tbls/src/dl_verification.rs (line 44)

Summary: As is, `get_random_scalars` may theoretically produce values that fail on one system but pass on others.

Suggestion: `verify_poly_evals` should use the Fiat-Shamir heuristics to make the function deterministically checkable. We recommend hashing the entire set of evals, polynomial and a separate index for each output, and appropriate domain separator strings.

Status: Acknowledged

L06: Short `Scalar` randomizers produced from u64s.

Affected Code: fastcrypto-tbls/src/dl_verification.rs (line 185)

Summary: In the current version, the randomizers are produce by upcasting 64-bit integers into `scalars`. The (relatively) small size of the resulting Scalars may provide insufficient soundness guarantees, as Schwartz–Zippel lemma allows for a failure probability of $\cdot 2^{-64}$. We note that the upcasting is safe to do in terms of bias (if the intention is to produce 64-bit randomizers).

Suggestion: Increase size of rands to ca. 128 bits, to obtain a corresponding reduction to the soundness error. Alternatively, `get_random_scalars` could use the `rand` trait of `Scalar` to generate a random scalar instead of manually deserializing a random **u64**. Ideally, both options should be implemented via RO invocations.

Status: Acknowledged

L07: Non-Standard use of ElGamal CTR Mode.

Affected Code: fastcrypto-tbls/src/ecies.rs (line 126)

Summary: The protocol uses AES CTR Mode in a non-standard way (with fixed zero IV). Current usage ensures that each ElGamal/DH ephemeral key is only used once, so there is no AES key+IV reuse. However, the resulting code may be fragile (e.g., if communications over different shares are not properly batched as they are now). Additionally, the protocol specification should mention using AES in place of the RO construction for ElGamal.

Suggestion: Document that the AES code should not be re-used in other contexts. Alternatively, implement a standard CTR mode implementation (with a non-fixed IV) and appropriate checks to ensure decryption does not fail due to insufficient ciphertext bytes.

Status: Acknowledged

2.4 Informational

I01: Lagrange coefficients can be calculated without evaluation results.

Affected Code: fastcrypto-tbls/src/polynomials.rs (line 95)

Summary: `get_lagrange_coefficients_for_c0` doesn't need the complete `eval` object. The indexes of the evaluations should be sufficient to calculate the coefficients.

Suggestion: It might be worth to alter the interface so that the coefficients can be computed and/or cached based on indexes (without the corresponding evaluations).

Status: Acknowledged

I02: Possible speedup for Lagrange coefficient calculations.

Affected Code: fastcrypto-tbls/src/polynomials.rs (line 143)

Summary: Instead of dividing immediately, the `get_lagrange_coefficients_for_c0` can return the common numerator, and then, inside `recover_c0`, the numerator can be multiplied to produce the final result. This saves `t` scalar multiplications.

Status: Acknowledged

I03: `total_weight` can be removed from `Nodes` struct.

Affected Code: fastcrypto-tbls/src/nodes.rs (line 26)

Summary: It is not required to store and maintain the `total_weight` in the `Nodes` struct as the `total_weight` will always be equal to the last entry of the `accumulated_weight`.

Suggestion: The `total_weight` getter function can be augmented to return the last entry of `accumulated_weight`.

Status: Acknowledged

I04: Name reuse w.r.t. `total_weight`.

Affected Code:

- fastcrypto-tbls/src/nodes.rs (line 50)
- fastcrypto-tbls/src/dkg.rs (lines 408,481)

Summary: The name `total_weight` is used more than twice in different contexts. Initially it is used inside `nodes.rs` to represent the sum of all weights in the protocol. In `dks.rs` it is used in `L408` to refer to the weight of the first message set (I1), and later in `L481` to refer to the size of the second message set (I2).

Suggestion: Use more specific names for the variables in `dks.rs`, e.g. `phase_1_weight`, `phase_2_weight`.

Status: Acknowledged

I05: Potential speedup in `reduce`.

Affected Code: fastcrypto-tbls/src/nodes.rs (line 150)

Summary: The for loop can be reversed, and a break statement can be used when the if statement is entered for the first time. For all cases where a reduction is possible, we will break earlier and not iterate over the initial factors.

Status: This optimization cannot be applied due to the later changes in the codebase.

I06: Potential speedup in `Partial_sign_batch`.

Affected Code: fastcrypto-tbls/src/tbls.rs (line 39)

Summary: `Partial_sign_batch` can perform precomputation with regards to `h` to improve performance. Since the same `h` value is used for each share we sign for, we can afford to use large amounts of precomputation.

Suggestion: Signing with precomputation can be based on the existing fastcrypto `multiplier/windowed.rs` codebase. Alternatively, BLST also supports setting a window manually via `blst_[p1/p2]s_mult_wbits_precompute`

Status: Acknowledged

I07: Potential Consolidation in NIZK code.

Affected Code: fastcrypto-tbls/src/nizk.rs (lines 17,112)

Summary: The structure of the Schnorr and generalized Pedersen protocols is very similar.

Suggestion: Consider consolidating NIZK code so that Pedersen and Schnorr proofs share the same code, which would accept a vector of inputs (with length=1 for Schnorr and length=2+ for Perdersen and variants).

Status: Acknowledged

I08: Potential speedup on Fiat Shamir verification.

Affected Code: fastcrypto-tbls/src/nizk.rs (line 186)

Summary: The final check for the NIZK can be sped up by refactoring to one side and using multi-multiplication.

Suggestion: Currently we check:

```

let left = *e1 + *e2 * c;
let right = *e3 * z;
left == right
```

Instead, we can rewrite this as

```

let left = *e1 ;
let right = *e3 * z + *e2 * (-c);
left == right
```

And calculate `right` via multi-multiplication.

Status: Acknowledged

I09: Unnecessary cloning of the partials iterator.

Affected Code: fastcrypto-tbls/src/tbls.rs (line 94)

Summary: The `aggregate` function performs unnecessary cloning of the partials iterator. The `unique_by` combinator creates a hashmap internally to perform the uniqueness check, and cloning leads to the work being performed twice.

Suggestion: Here is a cleaner way to do this:

```
let unique_partials = partials
    .unique_by(|p| p.borrow().index)
    .take(threshold as usize)
    .collect_vec();
if unique_partials.len() != threshold as usize {
    return Err(FastCryptoError::NotEnoughInputs);
}
```

Status: Acknowledged

I-10: Speed up in polynomial evaluations for share generation.

Affected Code: fastcrypto-tbls/src/polynomials.rs (line 68)

Summary: Currently, polynomial evaluations are calculated independently. Given that a common use case is to evaluate a polynomial on points $1 \dots k$, where k is larger than the `degree` of the polynomial, using the fixed differences method may be more efficient. To implement the method, the first `degree` number of evaluations are performed normally using the Horne's method. Then, a preprocessing step takes place to calculate a vector of derivatives of size `degree`. Using that vector, the value of subsequent evaluations can be calculated using only additions

Suggestion: Below is a pseudocode illustrating the method.

```
#degree+1 values total, skipping 0
for d in range(1,degree+2):
    p[d]=polyeval(d)

#initial preprocessing
A=[p[i+1]-p[i] for i in range(1,degree+1)]

#preprocessing in place
for j in range(degree-1,0,-1):
    for i in range(j):
        A[i]=A[i+1]-A[i]
```

```python
# update step
def update(A):
    for j in range(len(A)-1):
        A[j+1]+=A[j]

# remaining evaluations
for d in range(degree+2,degree+1+k):
    update(A)
    p[d]=p[d-1]+A[-1]
```

Status: Acknowledged

I11: Potential DOS attack due to unconstrained vectors.

Affected Code: fastcrypto-tbls/src/dkg.rs (lines 67,135)

Summary: The structs `Message` and `Confirmation` represent protocol messages which are transferred between parties over the broadcast channel. These structs contain variable-size vectors. The items of these vectors are processed by the receiving party without any checks on the total size of the vectors. This can enable an adversary to craft large protocol messages that might lead to a denial of service attack to the receiver. This attack becomes infeasible if blockchains are used as the broadcast channel due to the cost of posting large messages and the limit on the maximum block size.

Suggestion: Implement a custom deserialization for the structs that constraints the size of the vectors.

Status: Pending

2.5 Model Limitations and concerns on Application integration.

In this section we note a number of limitations that are inherent in the design implemented, or possibilities for inadvertent mis-use of internal functions. While we do not believe the issues to be exploitable in the currently intended application, we suggest that they be considered in the documentation and specification of the codebase.

First, subsequent works [GJKR07], have identified issues with the Pedersen DKG scheme [Ped91] with regard to the distribution of the resulting public key. Specifically, an adversary, in control of at least 2 identities can "trap" their own contributions in an undetectable way, by issuing faulty shares from one adversarial identity to the other. The adversary can now choose whether to reveal this discrepancy (disqualifying the corresponding contribution) or keep it secret (in which case the contribution is valid).

Thus, the adversary can preemptively calculate the resulting public key *with* and *without* the contribution and choose the more beneficial one. In fact, the adversary can choose whether or not to sabotage each of their contributions. Thus, if the adversary controls x parties, they can choose amongst 2^x public keys. This limitation is only relevant if the value of the public key is used in applications beyond signature verification (e.g. as a randomness seed). As such, it does not influence the results of this audit.

Second, we note that the higher-level protocol shouldn't predicate uptime rankings, rewards, or similar metrics based on participation in signing w.r.t. the final aggregate key. This is because the adversary can force an honest user out of the final set of participants by sending them bad shares and delaying their complaint until enough parties have replied for Phase 3 to end. If we wish to increase participation in signing, we might want to extend Phase 3 to allow for complaints from all users.

Additionally, the current protocol allows for situations where the derived keys rely on only 1 honest party (e.g., f out of $f+1$ members of I_1 are corrupted). While this is theoretically fine in the static corruption model, in a real-world situation where this 1 honest party is later corrupted, the secret key would be exposed to the adversary. For example, this could happen if this party no longer has an economic incentive to behave honestly and/or protect its protocol randomness and transcript.

Finally, The weight reduction is not specified outside of the code. The property seems to be that if there exists $2f+\delta+1$ weight out of $3f+\delta+1$ in the hands of honest users before compression, then (1) after compression with d, honest users will hold at least roundup $((2f+1)/d)$ and dishonest ones will hold at most `rounddown`(f/d).

As a trivial example, suppose we have 5 honest participants each with 20 stake, and 1 adversarial with 40. Also suppose we allow a δ of 5. A single application of reduce with that δ is safe (as 100-5 = 95 is greater than 2*40+1). The resulting shares are then 5 honest users with 1 share and 1 dishonest with 2. Reducing again, will allow a reduction with 2 as the lost shares are "only" 5 which is allowable.

In general, repeated applications of reduce should not be possible. First, as delta is expressed as an absolute value, its real measure in terms of initial stake is multiplied by the previous reduction scalar (in the above, the delta of 5 in the second round, actually allows for 100 initial stake to be destroyed). Even with delta expressed as a percentage (e.g. 3%), the (worst case) loss is incremented with each round.

Fortunately, even repeated applications do not break safety: if the adversary does not hold t shares at the start, he will not hold t (reduced) shares at the end. This is due to the shares being rounded down when divided, whereas the threshold value t is rounded up. In the above, a value of $t = 85$ would reduce to 3 in the first instance and to 2 in the second.

References

BLS01. Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. In *International conference on the theory and application of cryptology and information security*, pages 514–532. Springer, 2001.

Bol02. Alexandra Boldyreva. Threshold signatures, multisignatures and blind signatures based on the gap-diffie-hellman-group signature scheme. In *International Workshop on Public Key Cryptography*, pages 31–46. Springer, 2002.

GJKR07. Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk, and Tal Rabin. Secure distributed key generation for discrete-log based cryptosystems. *Journal of Cryptology*, 20:51–83, 2007.

Ped91. Torben Pryds Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *CRYPTO 91 - Annual international cryptology conference*, pages 129–140. Springer, 1991.

About Common Prefix

Common Prefix is a blockchain research, development, and consulting company consisting of a small number of scientists and engineers specializing in many aspects of blockchain science. We work with industry partners who are looking to advance the state-of-the-art in our field to help them analyze and design simple but rigorous protocols from first principles, with provable security in mind.

Our consulting and audits pertain to theoretical cryptographic protocol analyses as well as the pragmatic auditing of implementations in both core consensus technologies and application layer smart contracts.