Fine penetration tests for fine websites

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Audit-Report Silence Laboratories ECDSA lib 10.2022

Cure53, Dr.-Ing. M. Heiderich, Dr. N. Kobeissi

Index

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Introduction

"The only library you need for Proofs supported MPC. Enabling enterprises to adopt proofs-assisted Threshold Signatures (MPC) and MFA through unique fusion of cryptography and signal processing."

From https://silencelaboratories.com/

This report describes the results of a thorough cryptography review and source code audit performed by Cure53 against the ECDSA secp256k1 TSS(2,2) JS library, which is maintained by Silence Laboratories Pte. Ltd. As for the context and timeline, the work was requested by Silence Laboratories Pte. Ltd. in September 2022 and carried out by Cure53 in early-to-mid October 2022, namely in CW40 and CW41.

Note that this was the first time Cure53 looked at this library but some of its dependencies have been subject to Cure53 audits before, for example *noble/secp256k1*[1]. Regarding resources, a total of ten days were invested to reach the coverage expected for this project. It should also be noted that a team consisting of two senior testers has been created and assigned to this project's preparation, execution and finalization.

The work was structured under one work package (WP):

- **WP1**: Cryptography reviews & Audits against ECDSA secp256k1 TSS(2,2) JS library

In preparation for this assessment, Cure53 was given access to sources as well as any other means of information required to perform this audit, the methodology chosen here was crystal-box. All work necessary from the maintainers' side was completed in late September 2022, namely in CW39, so that the Cure53 team could have a smooth start into the auditing stage in the following week.

Communications during the test were done using a dedicated shared Slack channel, with which the two teams were connected and could exchange test-specific info, status reports and Q&As. Test-related discussions were very smooth and not many questions had to be asked. The scope was well prepared and clear, contributing to no noteworthy roadblocks being encountered during this audit and review.

[1] https://cure53.de/pentest-report_noble-lib.pdf

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Cure53 gave frequent status updates about their progress and the related findings; live reporting of major issues was not necessary, since none such findings were made.

On that note, the Cure53 team managed to get very good coverage over the WP1 scope items. Neither vulnerabilities nor weaknesses of any kind were observed by the auditors as negatively affecting the security premise of the secp256k1 TSS(2,2) JS library. This means that the library presented itself in a very good light with regard to cryptographic features and guarantees it makes. As this report will describe in later chapters, all relevant scope areas and key focus areas were inspected in depth, but none yielded findings. Hence, the overall impression gained is very positive.

The report will now delineate the scope and test setup, as well as present the available material for testing. After that, the report will detail the steps undertaken during the cryptography audit and source code review, offering detailed descriptions of the adopted approaches and corresponding outcomes.

The report will then close with a conclusion in which Cure53 will elaborate on the general impressions gained throughout this test. The responsible test team will share some words about the perceived security posture of the ECDSA secp256k1 TSS(2,2) JS library.

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Scope

- **Cryptography reviews & Code audits against ECDSA secp256k1 TSS(2,2) JS library**
  - **WP1**: Cryptography reviews & Audits of ECDSA secp256k1 TSS(2,2) JS library
    - **Library in scope:**
      - *ecdsa-tss-js*
    - **Repository URL:**
      - https://github.com/silence-laboratories/ecdsa-tss-js
    - **Commit ID:**
      - 9eefe4ec90dab75904542df5fbfa50a535cb3ef6
    - **Key focus areas:**
      - **Paillier cryptosystem**
        - External dependency:
          - *paillier-bigint* (not in scope)
      - **ECDSA implementation**
        - External dependency:
          - *noble/secp256k1* (not in scope, audited before[2])
      - **Key-generation**
        - *ecdsa/keygen/P1KeyGen.ts*
        - *ecdsa/keygen/P2KeyGen.ts*
      - **Establishing a threshold signature**
        - *ecdsa/signature/P1Signature.ts*
        - *ecdsa/signature/P2Signature.ts*
      - **Zero-knowledge proof logic**
        - Proofs of knowledge regarding discrete log:
          - *zkProofs/pDLProof*
        - Hash commitments:
          - *zkProofs/hashCommitments*
  - **Test-supporting material was shared with Cure53**
  - **All relevant sources were shared with Cure53**

[2] https://cure53.de/pentest-report_noble-lib.pdf

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Cryptography Review

ECDSA-TSS is a clean-room implementation of Yehuda Lindell's *Fast Secure Two-Party ECDSA Signing* scheme[3], first introduced in 2017 as foundational work for Lindell's startup company, Unbound Security Ltd., recently acquired by Coinbase. The scheme proposes a faster, simpler protocol to tackle the restricted "2-of-2" threshold signature case, which is especially useful in custodial cryptocurrency wallet scenarios.

Cure53 was tasked with a comprehensive cryptographic review of *ECDSA-TSS*. This review was split into three major work components:

- Zero-knowledge logic library (*zkProofs*)
- Key generation protocol logic for parties 1 and 2 (*ecdsa/keygen*)
- Threshold signature establishment logic for parties 1 and 2 (*ecdsa/signature*)

For each of the above, the following review methodologies were performed:

- Code analysis according to the TSS protocol's flow logic and comparison to the original protocol. as specified in Lindell's paper.
- Comparison to a third-party implementation of Lindell paper's contribution in Rust.[4]

Review methodologies failed to identify any outstanding security issues or vulnerabilities in *ECDSA-TSS*. As such, this documentation of the cryptography review aims to outline the process adopted during the project, as it has been followed for each of the above-mentioned methodologies.

It is crucial to note that Lindell (2017) does not provide an IETF-style protocol implementation spec, but rather sticks to high-level descriptions, leaving much of the implementation details unspecified. This results in protocol engineers needing to fill in the gaps. Hence, it fosters a certain lack of consistency among independent implementations.

While *ECDSA-TSS* does fill these gaps with how it applies and implements low-level primitives, the high-level specification is still sufficiently close to the 2017 description by Lindell. In essence, the paper can be used as a benchmark for evaluating the design and security of the resulting software library.

[3] https://eprint.iacr.org/2017/552
[4] https://github.com/ZenGo-X/multi-party-ecdsa/tree/master/src/protocols/two_party_ecdsa/lindell_2017

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Code analysis according to TSS protocol flow logic

ECDSA-TSS's *README.md* provides a sample description of the top-level protocol flow that is expected to be executed by both parties for session setup and signature. Session ID generation is handled outside of the scope of the library, with private keys being optionally provided from a third-party source or generated internally as a random curve scalar using secure pseudorandomness.

Key generation subprotocol

In accordance with the original protocol specification laid out by Lindell, Party 1 (P1) is tasked with generating the Paillier cryptosystem parameters and communicating the initial session commitment message to Party 2 (P2).

PROTOCOL 3.1 (Key Generation Subprotocol $\mathsf{KeyGen}(\mathbb{G}, g, q)$**)**

Given joint input $(\mathbb{G}, G, q)$ and security parameter 1^n, work as follows:

1. ***P*$_1$'s first message:**
   (a) P_1 chooses a random $x_1 \leftarrow \left\{\frac{q}{3}, \ldots, \frac{2q}{3}\right\}$, and computes $Q_1 = x_1 \cdot G$.
   (b) P_1 sends $(\mathsf{com\text{-}prove}, 1, Q_1, x_1)$ to $\mathcal{F}_{\mathsf{com\text{-}zk}}^{R_{DL}}$ (i.e., P_1 sends a commitment to Q_1 and a proof of knowledge of its discrete log).
2. ***P*$_2$'s first message:**
   (a) P_2 receives $(\mathsf{proof\text{-}receipt}, 1)$ from $\mathcal{F}_{\mathsf{com\text{-}zk}}^{R_{DL}}$.
   (b) P_2 chooses a random $x_2 \leftarrow \mathbb{Z}_q$ and computes $Q_2 = x_2 \cdot G$.
   (c) P_2 sends $(\mathsf{prove}, 2, Q_2, x_2)$ to $\mathcal{F}_{\mathsf{zk}}^{R_{DL}}$.
3. ***P*$_1$'s second message:**
   (a) P_1 receives $(\mathsf{proof}, 2, Q_2)$ from $\mathcal{F}_{\mathsf{zk}}^{R_{DL}}$. If not, it aborts.
   (b) P_1 sends $(\mathsf{decom\text{-}proof}, 1)$ to $\mathcal{F}_{\mathsf{com\text{-}zk}}^{R_{DL}}$.
   (c) P_1 generates a Paillier key-pair (pk, sk) of length $\max(3 \log |q| + 1, n)$ and computes $c_{key} = \mathsf{Enc}_{pk}(x_1)$. Denote $N = pk$. (Note that n denotes the minimum length of N for Paillier to be secure.)
   (d) P_1 sends $pk = N$ and c_{key} to P_2.
4. **ZK proofs:** P_1 proves to P_2 in zero knowledge that $N \in L_P$ and that $(c_{key}, pk, Q_1) \in L_{PDL}$.
5. ***P*$_2$'s verification:** P_2 aborts unless all the following hold: **(a)** it received $(\mathsf{decom\text{-}proof}, 1, Q_1)$ from $\mathcal{F}_{\mathsf{zk}}^{R_{DL}}$, **(b)** it holds that $c_{key} \in \mathbb{Z}_{N^2}^*$, **(c)** it accepted the proofs that $N \in L_P$ and $(c_{key}, pk, Q_1) \in L_{PDL}$, and **(d)** the key $pk = N$ is of length at least $\max(3 \log |q| + 1, n)$.
6. **Output:**
   (a) P_1 computes $Q = x_1 \cdot Q_2$ and stores (x_1, Q).
   (b) P_2 computes $Q = x_2 \cdot Q_1$ and stores (x_2, Q, c_{key}).

Figure 1. Key generation subprotocol description, from Lindell (2017).

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

In *P1KeyGen.getKeyGenMessage1()*, P1:

1. generates *q1* in accordance with the spec, with a random *x1*[5] (line 102): `this.q1 = P1KeyGen.G.multiply(this.x1)`
2. generates *com-prove* and creates a message containing two discrete log proof commitments, which are then communicated to P2.

In *P2KeyGen._processKeyGenMessage1()*, P2:

1. validates the state to ensure protocol context and correctness of session ID
2. generates *q2* in accordance with the spec, whereas a random *x2* (line 77) is `q2 = P2KeyGen.G.multiply(this.x2)`
3. generates *proof* and creates a message containing two discrete log proof commitments, which are then communicated to P2.

In *P1KeyGen._processKeyGenMessage2()*, P1:

1. validates the state to ensure protocol context and correctness of the session ID
2. verifies *dLogProof1* shared in the previous message by P2.
3. *Step 3*(c) of *Figure 1* is pre-calculated by *P1* before protocol messages occur, meaning that the Paillier cryptosystem keys are pre-generated. This is likely done in order to optimize performance.
4. generates a Paillier cryptosystem commitment under the session ID.
5. generates and encrypts *cKey* in accordance with *Step 3(*c) of Figure 1.
6. calculates the *proofs* described in *Step 4* in *Figure 1*.

In *P2KeyGen._processKeyGenMessage3()*, P2:

1. validates the state to ensure protocol context and correctness of the session ID
2. validates all of the necessary zero-knowledge *proofs* described in *Step 4* of *Figure 1* and is able to authenticate and derive the shared signing public key.

[5] Note that Lindell (2017) describes *x1* as being within the range *{q/3, …, 2q/2}*. *ECDSA-TSS* does not enforce this restriction, allowing *x1* to be generated and provided fully outside of the scope of the implementation.

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Signing subprotocol

PROTOCOL 3.2 (Signing Subprotocol Sign(sid, m))

A graphical representation of the protocol appears in Figure 1.

Inputs:

1. Party P_1 has (x_1, Q) as output from Protocol 3.1, the message m, and a unique session id sid.
2. Party P_2 has (x_2, Q, c_{key}) as output from Protocol 3.1, the message m and the session id sid.
3. P_1 and P_2 both locally compute $m' \leftarrow H_q(m)$ and verify that sid has not been used before (if it has been, the protocol is not executed).

The Protocol:

1. **P_1's first message:**
   (a) P_1 chooses a random $k_1 \leftarrow \mathbb{Z}_q$ and computes $R_1 = k_1 \cdot G$.
   (b) P_1 sends $(\mathsf{com\text{-}prove}, sid\|1, R_1, k_1)$ to $\mathcal{F}_{\mathsf{com\text{-}zk}}^{R_{DL}}$.
2. **P_2's first message:**
   (a) P_2 receives $(\mathsf{proof\text{-}receipt}, sid\|1)$ from $\mathcal{F}_{\mathsf{com\text{-}zk}}^{R_{DL}}$.
   (b) P_2 chooses a random $k_2 \leftarrow \mathbb{Z}_q$ and computes $R_2 = k_2 \cdot G$.
   (c) P_2 sends $(\mathsf{prove}, sid\|2, R_2, k_2)$ to $\mathcal{F}_{\mathsf{zk}}^{R_{DL}}$.
3. **P_1's second message:**
   (a) P_1 receives $(\mathsf{proof}, sid\|2, R_2)$ from $\mathcal{F}_{\mathsf{zk}}^{R_{DL}}$; if not, it aborts.
   (b) P_1 sends $(\mathsf{decom\text{-}proof}, sid\|1)$ to $\mathcal{F}_{\mathsf{com\text{-}zk}}$.
4. **P_2's second message:**
   (a) P_2 receives $(\mathsf{decom\text{-}proof}, sid\|1, R_1)$ from $\mathcal{F}_{\mathsf{com\text{-}zk}}^{R_{DL}}$; if not, it aborts.
   (b) P_2 computes $R = k_2 \cdot R_1$. Denote $R = (r_x, r_y)$. Then, P_2 computes $r = r_x \bmod q$.
   (c) P_2 chooses a random $\rho \leftarrow \mathbb{Z}_{q^2}$ and random $\tilde{r} \in \mathbb{Z}_N^*$ (verifying explicitly that $\gcd(\tilde{r}, N) = 1$), and computes
   $$c_1 = \mathsf{Enc}_{pk}\left(\rho \cdot q + \left[k_2^{-1} \cdot m' \bmod q; \tilde{r}\right]\right).$$
   Then, P_2 computes $v = k_2^{-1} \cdot r \cdot x_2 \bmod q$, $c_2 = v \odot c_{key}$ and $c_3 = c_1 \oplus c_2$.
   (d) P_2 sends c_3 to P_1.
5. **P_1 generates output:**
   (a) P_1 computes $R = k_1 \cdot R_2$. Denote $R = (r_x, r_y)$. Then, P_1 computes $r = r_x \bmod q$.
   (b) P_1 computes $s' = \mathsf{Dec}_{sk}(c_3)$ and $s'' = k_1^{-1} \cdot s' \bmod q$. P_1 sets $s = \min\{s'', q - s''\}$ (this ensures that the signature is always the smaller of the two possible values).
   (c) P_1 verifies that (r, s) is a valid signature with public key Q. If yes it outputs the signature (r, s); otherwise, it aborts.

If a party aborts at any point, then all Sign(sid, m) executions are halted.[2]

Figure 2: Signing subprotocol description, based on Lindell (2017).

In *P1Signature._getSignMessage1()*, P1:

1. generates *k1* and *r1* in accordance with *Step 1*(a) from *Figure 2*.
2. generates a discrete log *proof* over *k1 and r1* for the session ID as the context,
3. creates a commitment which is shared with *P2*, in accordance with *Step 1*(b) of *Figure 2*.

In *P2Signature._processSignMessage1()*, *P2*:

1. validates the state to ensure protocol context and correctness of session ID.

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

2. generates *k2* and *r2* in accordance with *Step 2*(b) of *Figure 2*.
3. generates a discrete log *proof* over *k2 and r2* for the session ID as the context,
4. creates a commitment which is shared with *P1*, in accordance with *Step 2*(c) of *Figure 2*.

In *P1Signature._processSignMessage2()*, P1:

1. validates the state to ensure protocol context and correctness of session ID.
2. validates the discrete log *proof* received from *P2* (*Step 3(*a) of *Figure 2*).
3. performs *Step 5*(a) of *Figure 2* in advance.

In *P2Signature.processSignMessage3()*, P2:

1. validates the state to ensure protocol context and correctness of session ID.
2. verifies the discrete log *proof* received from *P1* (*Step 4*(a) of *Figure 2*).
3. performs *Step 4(*b) of *Figure 2*; the calculation of *rUpper* and *r* (*r1 * k2* and *rx % q* is done.
4. performs all of the calculations described in *Step 4(*c) of *Figure 2* in the same order, and with the same naming conventions:

```
const m = utils.Uint8ArraytoBigint(this.messageHash);
const ro = utils.randBelow(q ** 2n);
const k2Inv = utils.bigintModInv(this.k2, q);
const c1 = paillierPublicKey.encrypt(
   ro * q + utils.modPositive(k2Inv * m,  q)
);
const v = k2Inv * r * this.p2KeyShare.x2;
const c2 = paillierPublicKey.multiply(cKeyX1, v);
const c3 = paillierPublicKey.addition(c1, c2);
```

Finally, in *P1Signature._processSignMessage4()*, *P1*:

1. validates the state to ensure protocol context and correctness of session ID
2. uses Paillier in order to decrypt the homomorphically calculated value *c3.*
3. performs *Step 5*(b) of *Figure 2* (*Step 5*(a) was performed in advance in *P1Signature._processSignMessage2()*), choosing the smaller signature.
4. performs *Step 5*(c), verifying whether the signature communicated by *P2* is valid.

Overall, *ECDSA-TSS* constitutes an exceptionally readable and to-spec implementation of Lindell 2017. No major deviations were found, except for the potential inclusion of the additional discrete log *proofs* for ephemeral values. These values do not seem to be described in the original paper.

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Comparison to third-party Rust implementation

The cryptographic protocol implemented by *ECDSA-TSS* poses multiple challenges to the security auditor. First, it involves a rarely implemented protocol. Second, the protocol "specification" is limited to a high-level academic description with no engineering specification for setting low-level details in stone. It also fails to provide test vectors. Third, the protocol depends on highly specialized low-level primitives, such as discrete log *proofs* and commitment schemes. On homomorphic encryption, it requires the implementation of the rarely used Paillier cryptosystem.

In order to ascertain a higher level of assurance with regard to the results of this audit, a third-party Rust implementation[6] was chosen as an additional target for the *ECDSA-TSS* code comparison. The results of the comparison are presented next.

Key generation subprotocol

For all key generation messages, the protocol flow was strikingly similar in terms of how low-level primitives were designed. Similarities were also documented in the ordering and separation of operation. They mark the only difference in variable names: both implementations use a hash-based approach in their commitment schemes.

Both implementations calculate a discrete log *proof* over *P1*'s initial secret share, *q1*, etc. The involved parties structure their internal states similarly and communicate the same messages over the wire. The same carries on for the entire key generation subprotocol.

Signing subprotocol

Similarly to the key generation subprotocol, the protocol flow, ordering of messages, ordering of operations and low-level implementation details, were highly similar between *ECDSA-TSS* and the independent Rust implementation. Only minor differences in variable names (with *ECDSA-TSS* being closer to Lindell 2017) were spotted.

[6] https://github.com/ZenGo-X/multi-party-ecdsa/tree/master/src/protocols/two_party_ecdsa/lindell_2017

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

```rust
impl Signature {
    pub fn compute(
        party_one_private: &Party1Private,
        partial_sig_c3: &BigInt,
        ephemeral_local_share: &EphEcKeyPair,
        ephemeral_other_public_share: &Point<Secp256k1>,
    ) -> Signature {
        //compute r = k2* R1
        let r = ephemeral_other_public_share * &ephemeral_local_share.
        secret_share;

        let rx = r
            .x_coord()
            .unwrap()
            .mod_floor(Scalar::<Secp256k1>::group_order());

        let k1_inv = ephemeral_local_share.secret_share.invert().unwrap
        ();

        let s_tag = Paillier::decrypt(
            &party_one_private.paillier_priv,
            &RawCiphertext::from(partial_sig_c3),
        )
        .0;
        let s_tag_fe = Scalar::<Secp256k1>::from(s_tag.as_ref());
        let s_tag_tag = s_tag_fe * k1_inv;
        let s_tag_tag_bn = s_tag_tag.to_bigint();

        let s = cmp::min(
            s_tag_tag_bn.clone(),
            Scalar::<Secp256k1>::group_order().clone() - s_tag_tag_bn,
        );

        Signature { s, r: rx }
    }
```

```rust
async _processSignMessage4(signMessage4: SignMessage4):
    Promise<{ signMessage5: SignMessage5, signature: string }> {
    if (this._state !== Party1SignatureState.PROCESS_SIGN_MSG_4) {
        this._state = Party1SignatureState.FAILED;
        throw new SignatureFailed('Invalid state');
    }
    if (this.sessionId !== signMessage4.sessionId) {
        this._state = Party1SignatureState.FAILED;
        throw new SignatureFailed('Invalid sessionId');
    }
    const c3 = signMessage4.c3;
    const s1 = this.p1KeyShare.paillierPrivateKey.decrypt(c3);
    const s2 = utils.modPositive(utils.bigintModInv(this.k1, q) * s1, q);
    const s = s2 < q - s2 ? s2 : q - s2;
    const signature = utils.signatureToHex(this.r, s);
    try {
        const signatureIsCorrect = await utils.verifySignature(
            this.messageHash,
            this.p1KeyShare.publicKey,
            signature
        );
        if (!signatureIsCorrect) {
            throw new SignatureFailed('Invalid signature');
        }
    } catch (e) {
        this._state = Party1SignatureState.FAILED;
        throw e;
    }
    const signMessage5 = new SignMessage5(this.sessionId, signature);
    this._state = Party1SignatureState.COMPLETE;
    return { signMessage5, signature };
}
```

Figure 3: Example of the strong similarity between ECDSA-TSS (right) and the independent Rust implementation (left).

It can be explained that the figure above (*Figure 3)* is showing implementations of *Step 5* from *Figure 2*. The independent Rust implementation does not seem to immediately validate the signature.

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Conclusion

This audit's scope targeted a full and original implementation of the two-party threshold signature scheme first described in 2017 in an academic paper by Yehuda Lindell. This scheme was subsequently deployed within major cryptocurrency wallet infrastructure instances. In autumn 2022, Cure53 was tasked with an audit of the cryptographic correctness of the protocol logic and flow, the protocol operations, as well as the underlying custom-implemented cryptographic primitives. The latter considered items outside of the core signing primitive and the Paillier cryptosystem primitive, both sourced from external libraries.

To assess the ECDSA secp256k1 TSS(2,2) JS library, which is maintained by Silence Laboratories, Cure53 carried out a rigorous analysis of ECDSA-TSS by evaluating its protocol flow logic. The auditors were checking for platform-specific issues that may arise through the engineering of cryptographic protocols in TypeScript/JavaScript, as well as audited the custom-implemented low-level cryptographic primitives. Furthermore, Cure53 performed a complete set of comparisons for the implementation, positioning it both against the original 2017 Lindell paper, and an independent Rust implementation of the same protocol.

For context, it should be noted that some consider the nature of Lindell's 2017 work to be relatively exotic. The project did not have a long or large implementation history and is characterized by unusual dependencies in terms of cryptographic primitives. In addition, this assignment framed the unusual target of TypeScript/JavaScript as a runtime for the implementation of cryptographic protocols involving homomorphic encryption and zero-knowledge proofs. Despite all these possibly adversarial conditions, no outstanding issues could be identified within ECDSA-TSS within the scope of this audit.

After a rigorous comparison to both the original 2017 Lindell paper and to an independent Rust implementation, Cure53 determined that ECDSA-TSS correctly implemented both of the core subprotocols of Lindell's 2017 work, and offered a high-level API that is also relatively hardened against unintentional misuse by the application layer.

Cure53 would like to thank Vlad Khomenko, Jay Prakash and Dr. Andrei Bytes from the Silence Laboratories Pte. Ltd. team for their excellent project coordination, support and assistance, both before and during this assignment.