Implementing Threshold Signatures for FIDO Authentication*

Javier Garcia[1], Magnus Ringerud[2], and Tjerand Silde[1,2]

[1] Norwegian University of Science and Technology, Trondheim, Norway
[2] PONE Biometrics, Oslo, Norway, ts@ponebiometrics.com,
https://ponebiometrics.com

Abstract. Threshold signatures split a signing key across N devices so that any T of them can jointly sign, while any fewer learn nothing about the key. For authentication this removes the single point of failure a lone hardware token represents, invisibly to the relying party, which stores one public key and verifies one ordinary signature.

We implement the `FROST(secp256k1, SHA-256)` ciphersuite of RFC 9591 on the OFFPAD biometric authenticator and an STM32 Nucleo-L476RG under Zephyr RTOS, in a 2-out-of-3 configuration. The devices cannot reach each other, so a computer relays all traffic; we encrypt every secret share end-to-end so the relay learns nothing, authenticate every message under a device key so the relay cannot equivocate, and add a blaming protocol that names the party at fault in a dispute.

We measure computation, memory, and per-transport cost over UART and USB HID, state the threat model and the residual gaps, and show that threshold FIDO authentication is within reach. Schnorr signatures over `secp256k1`, the scheme FROST produces, have no COSE identifier, but our experiments open a path to threshold Ed25519 that requires few changes. The work is also a starting point for threshold ECDSA for FIDO authentication in high-security environments.

* This report builds on the master's thesis of Javier Garcia [Gar25] at NTNU, carried out in collaboration between the NTNU Applied Cryptology Lab and PONE Biometrics. A condensed version appears in the proceedings of the Norwegian Information Security Conference, NISK 2026 [GS26]. The work was funded by the Research Council of Norway under the OFFPAD project: https://prosjektbanken.forskningsradet.no/en/project/FORISS/321619. This version: October 2026.

Table of Contents

1 Introduction . . . 4
1.1 Objectives and Achievements . . . 4
1.2 Contributions . . . 5
1.3 Scope and Limitations . . . 6
1.4 Related Work . . . 6
2 Background . . . 9
2.1 Notation . . . 9
2.2 Elliptic Curves . . . 9
2.3 Elliptic Curve Cryptography . . . 11
2.4 Schnorr Signatures . . . 12
2.5 Zero-Knowledge Proofs . . . 13
2.6 Secret Sharing . . . 14
2.7 Commitments . . . 15
2.8 Public Key Encryption of Secret Shares . . . 15
3 FIDO Protocol . . . 17
3.1 Architecture . . . 17
3.2 Client to Authenticator Protocol . . . 18
3.3 Specification Versions . . . 19
3.4 Web Authentication . . . 19
3.5 Signature Algorithms and How WebAuthn Encodes Them . . . 19
3.6 Registration . . . 20
3.7 Authentication . . . 22
4 FROST Implementation . . . 23
4.1 Threshold Signatures . . . 24
4.2 The FROST Protocol and Our Ciphersuite . . . 25
4.3 Message Authentication . . . 26
4.4 Distributed Key Generation . . . 27
4.5 Signing . . . 32
4.6 A Walkthrough of One Complete Run . . . 34
4.7 Registration and Authentication with FROST Signatures . . . 36
4.8 Registering a FROST Key with WebAuthn . . . 39
4.9 The FIDO2 Compliance Boundary . . . 41
4.10 Evaluation . . . 42
4.11 Reproducing the Setup . . . 46
5 Communication . . . 54
5.1 Communication Type . . . 54
5.2 Communication Model . . . 59
5.3 Broadcast Communication . . . 60
5.4 Summary of Design Decisions . . . 61
6 Security Considerations . . . 62
6.1 Threat Model . . . 62

6.2 The Coordinator and the Provisioned Key List 63
6.3 Information Revealed by the Blaming Protocol 64
6.4 Nonce Handling in Flash Memory 65
6.5 Session Binding and Concurrency 66
6.6 The WebSocket Bridge 66
6.7 Summary of Residual Gaps 66
7 Conclusions 66
7.1 Threshold Scheme and Communication Model 67
7.2 Lessons Learned 68
7.3 Software and Environment Setup 69
7.4 Resource Management and Configuration 69
7.5 Hardware Integration and Connections 69
8 Future Work 70
8.1 Environment and RTOS Upgrades 70
8.2 Dynamic Resource Allocation 70
8.3 Communication over Bluetooth 70
8.4 Threshold Parameter Exploration 71
8.5 Porting to Ed25519: A Concrete Work Breakdown 71
8.6 Threshold ECDSA for Universal Compatibility 75
8.7 Device Loss, Replacement, and Share Refresh 75
8.8 Open Design Questions for a Threshold Authenticator 76
8.9 Post-Quantum Considerations 77
8.10 Becoming an Authenticator 78
8.11 Closing the Security Gaps 78

1 Introduction

Users increasingly interact with online services that require authentication. Since maintaining many credentials is burdensome, users tend to fall back on simple, easy-to-remember passwords, which increases their exposure to attacks. The FIDO Authentication standards aim to eliminate this problem by replacing passwords with public key cryptography.

The OFFPAD, a biometric authentication device developed by PONE Biometrics, implements these standards. The device verifies its user locally, through fingerprint recognition or a PIN code, and is FIDO2 compatible, so it gives secure and convenient access to online services without a secret ever leaving the device. This work proposes to further develop this system by incorporating threshold signatures into the authentication process of the OFFPAD.

In a threshold signature scheme, the signing process is distributed across multiple devices: instead of storing the complete key on a single device, the key is split among several of them. Security improves because an attacker must compromise multiple devices to forge a signature. Distributing the key also removes the single point of failure that a lone authenticator represents, and increases availability, since authentication remains possible even if a device is lost, provided enough devices remain available. The price is additional protocol complexity and communication. The system is parameterised by a threshold T and the total number of devices N: any T of the N devices can jointly produce a signature, and raising T increases the number of devices an attacker must compromise, at the cost of requiring more devices to be present for every legitimate signature.

1.1 Objectives and Achievements

The main goal of this project is to explore the integration of threshold signatures into the OFFPAD's authentication mechanism, and to create the theoretical and technical foundation for a future production implementation. The focus is on how threshold cryptography can be incorporated into this device without compromising its simplicity and ease of use, and on the design decisions that make this possible on resource-constrained hardware.

To achieve this, the objectives of the project are the following:

FIDO2 fundamentals: To review and understand the fundamentals of the FIDO2 authentication standard, including its architecture, registration, and authentication flow, to identify where and how threshold cryptography can be introduced.

Cryptographic background: To study the mathematical and cryptographic background required for threshold signatures, including elliptic curve cryptography and secret sharing.

Threshold signature schemes: To review the state of the art in threshold signature schemes, with particular attention to Schnorr-based schemes such

as FROST, and threshold variants of ECDSA, evaluating their suitability for embedded, resource-constrained hardware like the OFFPAD.

Communication models: To analyse different communication models that could coordinate multiple devices during a threshold signing process, taking into account the trade-offs between decentralisation, security, and the limitations of the OFFPAD.

1.2 Contributions

Beyond these exploratory objectives, the project delivered a working prototype. A condensed account of it has been peer reviewed and accepted for publication at NISK 2026 [GS26]; this report is the full record. Concretely, it contributes the following.

A FROST implementation for embedded authenticator hardware. We implement the ciphersuite `FROST(secp256k1, SHA-256)` of RFC 9591 [CKGW24] on the OFFPAD development board and on an STM32 Nucleo-L476RG board, both running Zephyr RTOS 3.7.0, and evaluate it in a 2-out-of-3 configuration. To the best of our knowledge, this is the first reported implementation of FROST distributed key generation and signing on embedded FIDO authenticator hardware.

A distributed key generation hardened for a relaying coordinator. The devices cannot reach each other directly, so every message passes through a coordinating computer. We therefore encrypt every secret share end-to-end under a hashed-ElGamal scheme, so that the relaying coordinator learns nothing; sign every message under a long-term device key, verified against a list of device public keys provisioned out of band, so that the coordinator can neither equivocate nor substitute an encryption key of its own; and add a blaming protocol that resolves a disputed share cryptographically and identifies which party deviated (Sections 4.3 and 4.4).

A transport layer for heterogeneous constrained devices. A single header-plus-payload message format carries both key generation and signing traffic over two different physical transports, UART on the Nucleo and USB HID on the OFFPAD, with chunking and reassembly for the latter. The coordinator broadcasts to all connected devices concurrently and proceeds as soon as the first T valid responses arrive (Section 5).

An end-to-end passwordless demonstrator. A proof-of-concept web server performs registration and authentication against the group public key, and we report on an experiment that substitutes the group key into a WebAuthn attestation object, which makes explicit exactly where the FIDO2 compliance boundary lies (Sections 4.7 and 4.8).

A measurement study and a security discussion. We report computation, memory, and communication costs separately per transport (Section 4.10), and we state the threat model explicitly, what the coordinator can and cannot

do once every message is authenticated, the information published by a blame proof, nonce persistence in flash, and the residual gaps of the deployment (Section 6).

A compliance analysis for FIDO2. We state precisely what blocks a threshold FIDO2 authenticator, which is narrower than the usual simplification, and identify three concrete paths forward, of which threshold Ed25519 is the nearest term (Section 4.9).

1.3 Scope and Limitations

The prototype is not a deployable FIDO2 authenticator. Three things separate it from one, and it is worth naming them early so that the rest of the report can be read with the right expectations.

First, the signatures it produces are Schnorr signatures over `secp256k1`. WebAuthn identifies signature algorithms by COSE identifier, and no registered COSE identifier denotes Schnorr over `secp256k1`; the registered `ES256K` (`-47`) denotes *ECDSA* over that curve [Jon20]. A relying party therefore has no way to describe, let alone verify, the credential this prototype produces. However, WebAuthn registers `EdDSA` (`-8`) and Ed25519 is itself a Schnorr signature. Section 4.9 works through the consequences.

Second, the coordinator is a general-purpose computer and the browser integration goes through an unauthenticated local WebSocket bridge, which is a testing convenience rather than a transport (Section 6.6).

Third, we implement identifiable abort rather than robustness: a participant that deviates during signing is detected and named, but the signing session fails and must be restarted with a different signing set. Achieving a guaranteed output requires a wrapper such as ROAST [RRJ+22].

1.4 Related Work

Threshold Schnorr signatures. Our starting point is the line of work on threshold Schnorr signatures. FROST, introduced by Komlo and Goldberg [KG21], reduced signing to two rounds while allowing unrestricted concurrent signing sessions, and has since been standardised as RFC 9591 [CKGW24], the specification our implementation follows. Subsequent work has both refined and extended the protocol. Bellare et al. [BCK+22] gave a systematic security treatment of non-interactive threshold signatures and introduced FROST2, a variant that reduces the number of exponentiations per signer. ROAST [RRJ+22] wraps FROST-like schemes to obtain robust, asynchronous signing, guaranteeing that a signature is produced whenever T honest signers are reachable even if other participants actively disrupt the protocol. Crites, Komlo, and Maller [CKM23] gave Sparkle, a three-round threshold Schnorr scheme with security against an *adaptive* adversary, one that chooses whom to corrupt during the protocol rather than in advance; this is worth noting here because the analysis we rely on, and

hence the guarantee our system inherits, assumes static corruption (Section 6.1), and the extra round is the price Sparkle pays for removing that assumption. On the key generation side, the DKG we use is Pedersen's distributed key generation, as analysed by Gennaro et al. [GJKR99], with a proof of knowledge added against rogue-key attacks, and ChillDKG [RNM+24] is a recent DKG designed specifically for FROST deployments without a broadcast channel; Section 4.4 relates both to our construction.

Threshold signatures are also moving into standardisation. NIST's first call for multi-party threshold schemes [BP26] was finalised in January 2026, and a companion draft specifically treats threshold EdDSA and Schnorr signatures [BD22]; the latter is directly relevant to the migration path we recommend in Section 4.9. All of these works target the protocol layer rather than the devices that run it.

Threshold signing on constrained hardware. Running a threshold scheme on small devices is not new in itself. Mavroudis et al. [MCS+17] built a signing appliance out of a board of commodity smart-card integrated circuits, so that no single chip has to be trusted, and reported the engineering this demands of the host that drives them. More recently, FROST has been implemented on microcontroller-class signing devices for cryptocurrency custody [Fro25]. Both establish that the arithmetic fits hardware of the class we target, and neither settles the question this report is about. Our participants are FIDO authenticators: each has a single link to a host computer that the user's security model would rather not trust, no way to address its peers, and an output that an unmodified relying party has to accept. It is that combination of a constrained device, an untrusted relay and an unchanged verifier that makes the engineering here specific, and it is the gap this report addresses.

Threshold ECDSA. A parallel line of work studies threshold variants of ECDSA, which matters for FIDO because `ES256`, ECDSA over the NIST P-256 curve, is the only signature algorithm that every relying party can be assumed to accept. Lindell [Lin17] gave a fast two-party ECDSA signing protocol, and Gennaro and Goldfeder [GG18] extended fast threshold ECDSA to the general multiparty setting with distributed key generation. More recently, Doerner, Kondi, Lee, and shelat [DKLs24] presented a three-round threshold ECDSA protocol that replaces expensive zero-knowledge proofs with lightweight statistical consistency checks. As discussed in Section 8, this protocol is our candidate for the universally compatible version of the present system.

Removing the single point of failure in FIDO2. The problem this project addresses, that a FIDO2 credential lives on one device, has been attacked from two directions in the literature, and it is useful to separate them.

The first is *key derivation for backup and delegation*. Asynchronous Remote Key Generation (ARKG), introduced by Frymann et al. [FGK+20] in an analysis of

a Yubico proposal for WebAuthn, lets a sender generate a fresh public key pk' on behalf of a receiver such that only that receiver can later derive the matching sk'. This allows a backup authenticator to be registered with a relying party without ever being present, which solves the recovery half of the problem; the primitive is now the subject of an IETF CFRG draft. Manulis and Nartz [MN25] recently extended this to the distributed setting, where a sender generates pk' for a group of receivers such that any threshold of them can jointly derive sk', giving threshold-controlled access to a WebAuthn credential.

Distributed ARKG (DARKG) is the closest published work to ours in intent, and the difference is worth stating precisely. Its threshold structure governs who is able to *derive* the credential's private key; ours governs who is able to *use* it, and the private key is never assembled at any point or place in our system, including at signing time. The two are complementary rather than competing: DARKG addresses how a distributed credential comes to exist and be registered, which is exactly the part our prototype handles crudely through a coordinator-driven registration, while a threshold signing protocol addresses how it is exercised without reconstruction. Combining the two is a natural direction and, to our knowledge, has not been done. The other difference is one of kind: ARKG and DARKG are protocol-design papers with security proofs and no embedded implementation, whereas this report contributes an implementation and measurements and no new protocol.

The second direction is *splitting the signing operation*, which is what we do. Larch [DLCGM23] splits the ability to produce FIDO2 signatures between the user's device and a log server using two-party ECDSA, so that every authentication necessarily leaves an audit record, while remaining compatible with unmodified relying parties. Larch distributes trust between the user and a service to achieve accountability; in contrast, our system distributes the signing key across multiple hardware devices held by the user, with no server-side participant, to remove the single point of failure that a lone authenticator represents. Like Larch, our design is transparent to the relying party, which only ever sees an ordinary public key and signatures.

Formal analysis of FIDO2. A third line of work analyses FIDO2 itself. Barbosa et al. [BBCW21] gave the first provable-security treatment of FIDO2 as a composition of WebAuthn and CTAP2, Bindel, Cremers, and Zhao [BCZ23] extended this to CTAP 2.1 and WebAuthn 2 and studied post-quantum instantiations, and Hanzlik, Loss, and Wagner [HLW23] formalised privacy and revocation. These analyses are relevant to us in a specific way: all of them model the authenticator as a single party holding a single key. A threshold authenticator does not fit that model, and extending any of these frameworks to an authenticator whose signing capability is distributed across devices, with a relay in between, is open work that would put a system like ours on the same formal footing as the standard it aims to implement.

2 Background

Cryptography relies on mathematical structures to provide secure communication. A fundamental ingredient of many cryptographic techniques is a mathematical problem that is easy to compute in one direction but computationally hard to invert. One example is integer multiplication: computing the product of two large prime numbers is easy, whereas factoring the product is a much harder task.

Elliptic Curve Cryptography (ECC) builds on such an asymmetry: it exploits the mathematical properties of elliptic curves over finite fields to construct cryptographic protocols that provide strong security with comparatively small key sizes.

2.1 Notation

Table 1 collects the notation used throughout the report. Three conventions deserve emphasis, because confusing them is a common source of implementation bugs. First, we work with two different finite fields: the *base field* $\mathbb{F}_p$ over which the curve is defined, whose elements are the coordinates of curve points, and the *scalar field* $\mathbb{Z}_q$, where q is the order of the group $\mathbb{G}$ generated by G, whose elements are secret keys, shares, nonces, and challenges. For `secp256k1` both are 256-bit but they are distinct primes, and every scalar arithmetic operation in this report is modulo q, never modulo p. Second, we write the group additively, as is customary for elliptic curves: $[a]P$ denotes scalar multiplication of the point P by the scalar a, and $P + Q$ denotes point addition. Sources written multiplicatively, including parts of the literature we cite, write these as P^a and $P \cdot Q$.

Third, single uppercase Roman letters denote group elements and single lowercase letters denote scalars or byte strings; multi-letter identifiers such as `sk`, `pk`, `dk` and `ek` follow the usual cryptographic convention instead. The square brackets in $[a]P$ therefore always enclose a scalar and are always applied to a point. This convention resolves a reading hazard that is easy to fall into: a commitment $C_{ij} = [a_{ij}]G$ is a curve point, not a field element, even though the challenge c of a Schnorr signature is a scalar and looks similar. Where the literature we follow writes commitments in lowercase, we capitalise them.

2.2 Elliptic Curves

For this project, we use elliptic curves over a finite field $\mathbb{F}_p$ (where p is a prime and operations are performed modulo p), defined by the equation [Sil09, Ch. 3]:

$$E_{a,b} = \{(x, y) \in \mathbb{F}_p \times \mathbb{F}_p \mid y^2 = x^3 + ax + b\} \cup \{\mathcal{O}\},$$

where $4a^3 + 27b^2 \neq 0$, which is the condition (up to the constant factor -16 appearing in the definition of the discriminant $\Delta = -16(4a^3 + 27b^2)$) that the curve be nonsingular. The points of $E_{a,b}$ form a group in which points can be added and scaled efficiently, with group operations given by algebraic rules over the finite field.

Table 1. Notation used throughout this report.

Symbol	Meaning
p, $\mathbb{F}_p$	Prime characteristic and base field of the curve
q, $\mathbb{Z}_q$	Prime order of $\mathbb{G}$ and the scalar field; all scalars live here
$\mathbb{G}$, G, $\mathcal{O}$	Group of curve points of order q, its generator, the point at infinity
$[a]P$	Scalar multiplication of the point P by the scalar a
N, T	Total number of participants and the signing threshold
P_i	Participant with identifier $i \in \{1, \ldots, N\}$
$\mathcal{J}$	The signing coalition, a subset of $\{1, \ldots, N\}$ with $\lvert\mathcal{J}\rvert = T$
f_i	The degree-$(T-1)$ polynomial sampled by P_i during key generation
C_{ij}	The commitment $[a_{ij}]G$ to the j-th coefficient of f_i
sk, pk	Signing key and public key $[\mathsf{sk}]G$; in the threshold setting pk is the group key, the only one a relying party sees
sk_i, pk_i	Signing share of P_i and its public verification share $[\mathsf{sk}_i]G$
dk_i, ek_i	Key pair of P_i used to encrypt shares in transit
U_{ij}, v_{ij}	Encrypted share from P_i to P_j: ephemeral point and masked bytes
d_i, e_i	Hiding and binding signing nonces of P_i, with commitments $D_i = [d_i]G$ and $E_i = [e_i]G$
λ_i	Lagrange coefficient of P_i at 0 for the coalition $\mathcal{J}$
$\mathsf{H}_1, \ldots, \mathsf{H}_5$	The domain-separated hash functions of the ciphersuite
$\xleftarrow{\$}$	Uniform sampling from a set
$\overset{?}{=}$	An equality that an algorithm is required to check

2.2.1 Point Addition Given $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ on $E_{a,b}$, group operations satisfy [HMV04, Ch. 3]:

Additive identity: $P + \mathcal{O} = P$.

Additive inverse: if $x_1 = x_2$ and $y_1 = -y_2$, then $P + Q = \mathcal{O}$.

Point addition:

$$P + Q = (x_3, y_3), \text{ where}$$

$$x_3 = \lambda^2 - x_1 - x_2, \qquad y_3 = \lambda(x_1 - x_3) - y_1,$$

$$\lambda = \begin{cases} \frac{3{x_1}^2 + a}{2y_1} & \text{if } P = Q \\ \frac{y_1 - y_2}{x_1 - x_2} & \text{if } P \neq Q \end{cases}$$

Scalar multiplication of a point is repeated point addition and is denoted by $Q = [x]P$, so that, for instance, $[2]P = P + P$. Implementations must compute $[x]P$ in time independent of x; a naive double-and-add leaks the bits of the scalar through timing and power consumption, which is a particularly relevant concern on an embedded authenticator that an attacker may hold in their hand.

The curve used throughout this project is `secp256k1` [Cer10], defined by $a = 0$, $b = 7$ over the prime field with

$$p = 2^{256} - 2^{32} - 977,$$

whose group of points has prime order

```
q = 0xFFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE
   BAAEDCE6 AF48A03B BFD25E8C D0364141
```

and cofactor 1. The cofactor being 1 matters: every point other than $\mathcal{O}$ generates the full group, so the small-subgroup checks required on curves such as edwards25519 reduce here to a check that a received point is on the curve and is not the identity. The choice of `secp256k1` was driven by library availability rather than by the application; Section 4.9 discusses the consequences.

2.3 Elliptic Curve Cryptography

Elliptic Curve Cryptography (ECC) [Mil86] leverages the structure of elliptic curves over finite fields to implement secure cryptographic protocols like Elliptic Curve Diffie-Hellman (ECDH) or Elliptic Curve Digital Signatures (ECDSA) [JMV01].

Its security is based on the Elliptic Curve Discrete Logarithm Problem (ECDLP) [Sil09, Ch. 5]: given two points $P, Q \in E(\mathbb{F}_p)$ on an elliptic curve over a finite field, find an integer x satisfying $[x]P = Q$.

Even the best known algorithms for the ECDLP, such as Pollard's ρ method [Pol78], require $O(\sqrt{q})$ group operations, where q is the order of the group generated by P. Solving the ECDLP in $E(\mathbb{F}_p)$ is therefore infeasible for sufficiently large p.

This gives ECC two important practical advantages [Mag16]: a 256-bit ECC key offers security comparable to a 3072-bit RSA key, and the smaller keys translate into lower computational overhead and faster processing.

2.4 Schnorr Signatures

For this project, we decided to work with the Schnorr digital signature scheme [Sch91], which is based on the discrete logarithm problem in finite groups. The scheme is valued for its efficiency and security, as it produces simpler and faster signatures than ECDSA [Won21]. It works as follows.

Key Generation:

1. Sample the secret key sk uniformly at random from a finite field $\mathbb{Z}_q$.
2. Compute public key as $\mathsf{pk} = [\mathsf{sk}]G$, where G is the generator point.

Signing:

1. Sample a random value r from the finite field $\mathbb{Z}_q$ and compute $R = [r]G$.
2. Compute challenge $c = \mathsf{H}(R, \mathsf{pk}, m)$, where H is a hash function that behaves as a random oracle and must be collision-resistant, which can be instantiated with a Secure Hash Algorithm such as SHA-2.
3. Compute response $z = r + c \cdot \mathsf{sk}$.
4. The signature is $\sigma = (R, z)$.

Verification:

1. Compute the challenge $c = \mathsf{H}(R, \mathsf{pk}, m)$.
2. The signature is valid if $[z]G = R + [c]\mathsf{pk}$.

Correctness follows by substitution: $[z]G = [r+c\cdot\mathsf{sk}]G = [r]G+[c][\mathsf{sk}]G = R+[c]\mathsf{pk}$. The argument order (R, pk, m) matches both RFC 9591 and Ed25519, and we keep it throughout. Three details of this scheme matter for the implementation and are easy to gloss over. The challenge c is an element of $\mathbb{Z}_q$, so the hash output must be reduced modulo q; when the hash output length matches the size of q this reduction is slightly biased, and specifications such as RFC 9591 avoid the bias by hashing to a wider value before reducing. The public key pk is included in the challenge, which binds the signature to it and prevents related-key attacks; variants that omit it are not equivalent. Finally, the nonce r must be sampled

freshly for every signature and never revealed: two signatures (R, z) and (R, z') on distinct messages $m \neq m'$ under the same R immediately yield

$$\mathsf{sk} = \frac{z - z'}{c - c'} \bmod q,$$

which is the single most damaging implementation failure in Schnorr-type schemes and which Section 6.4 returns to in the threshold setting.

2.5 Zero-Knowledge Proofs

A zero-knowledge proof is a cryptographic protocol that allows a party to prove that a certain statement is true, without revealing any information other than the statement being true itself [GMR85]. This makes zero-knowledge proofs very useful in threshold cryptography, where participants must demonstrate knowledge of a secret, such as their contribution to a jointly generated key, without revealing it. In the key generation of Section 4.4, each participant proves knowledge of its secret contribution a_{i0}, which prevents rogue-key attacks in which a malicious participant chooses its contribution as a function of the others' to bias or control the group key.

A zero-knowledge proof system must satisfy three fundamental properties:

Completeness: a prover can convince a verifier that a statement is true if it is true.

Soundness: no prover can convince a verifier that a statement is true if it is false; the probability of this happening is negligible.

Zero-knowledge: no information other than the fact that the statement is true is revealed.

A good example of a zero-knowledge proof of knowledge is the Schnorr identification protocol [Sch91], which allows a prover to demonstrate knowledge of a discrete logarithm without revealing it. For a cyclic group $\mathbb{G}$ of prime order q with generator G, and a secret value sk with corresponding public value $\mathsf{pk} = [\mathsf{sk}]G$, the protocol works as follows:

Commitment: the prover samples a random value r from $\mathbb{Z}_q$, computes $R = [r]G$, and sends R to the verifier.

Challenge: the verifier samples a random challenge c from $\mathbb{Z}_q$ and sends it to the prover.

Response: the prover computes $z = r + c \cdot \mathsf{sk}$ and sends z to the verifier.

Verification: the verifier accepts if $[z]G = R + [c]\mathsf{pk}$.

This protocol is interactive; it requires the verifier to generate a challenge. However, thanks to the Fiat-Shamir heuristic [FS87], an interactive proof can be transformed into a non-interactive one by replacing the verifier's random

challenge with the output of a cryptographic hash function applied to the prover's commitment and other contextual information, that is, $c = \mathsf{H}(R, \dots)$, removing the need for direct interaction between the prover and verifier.

Zero-knowledge proofs of this kind are a core building block of the FROST protocol, where each participant proves knowledge of their secret share during the distributed key generation phase without revealing it, preventing malicious participants from claiming to hold a share they never correctly generated.

2.6 Secret Sharing

Secret sharing is a central building block of this project: a secret is split into multiple shares such that any threshold T of them suffice to recover it, while fewer reveal nothing. Shamir's scheme [Sha79] uses polynomial interpolation over a finite field $\mathbb{Z}_q$ (with prime $q > N$). Since a polynomial of degree $T-1$ is uniquely determined by T points, having fewer than T points yields zero information about the constant term.

To share a secret $\alpha \in \mathbb{Z}_q$ [BS23, §22.1]:

Generation: Choose $T-1$ random coefficients $(a_1, \dots, a_{T-1}) \in \mathbb{Z}_q$ and define $f(x) = a_{T-1}x^{T-1} + \cdots + a_1 x + \alpha$, so that $f(0) = \alpha$. Distribute $\alpha_i = f(i)$ to participant i.

Combination: Collect T shares from a subset $\mathcal{J}$ and reconstruct $f(x)$ using Lagrange interpolation:

$$f(x) = \sum_{j=1}^{T} \alpha_{i_j} \cdot \lambda_{i_j}(x) \pmod q,$$

$$\text{where} \quad \lambda_{i_j}(x) = \prod_{\substack{k=1 \\ k \neq j}}^{T} \frac{x - i_k}{i_j - i_k} \pmod q.$$

Evaluate $f(0) = \sum_{j=1}^{T} \alpha_{i_j} \cdot \lambda_{i_j}(0) \pmod q = \alpha$ to recover the secret.

Only the evaluation at 0 is ever needed, so we abbreviate $\lambda_i = \lambda_i(0)$ and note that this coefficient depends on the whole coalition $\mathcal{J}$, not only on i:

$$\lambda_i = \prod_{k \in \mathcal{J},\, k \neq i} \frac{k}{k - i} \pmod q.$$

Two consequences are relevant to the threshold signing protocol of Section 4.5. The participant identifiers must be nonzero and distinct modulo q, since $i = 0$ would hand out the secret itself and a collision would make the denominator vanish; we use $i \in \{1, \dots, N\}$. A participant also cannot precompute λ_i before knowing who else is signing, which is why the coalition $\mathcal{J}$ is part of the message the coordinator sends in the second round.

Although Shamir's scheme ensures confidentiality, it does not guarantee that the distributed shares are consistent or correctly generated. Commitments, described in the following section, address this.

2.7 Commitments

A commitment scheme is a cryptographic primitive that allows a party to commit to a chosen value while keeping it hidden from others, with the ability to later reveal it in a way that proves it was not changed. Commitment schemes satisfy two main properties [Sch05]: *hiding*, meaning that the commitment reveals no information about the committed value, and *binding*, meaning that a committed value cannot later be opened as a different value.

In the context of secret sharing, commitments are used to construct verifiable secret sharing (VSS) schemes, extending Shamir's secret sharing so that participants are able to verify the validity of their shares [Sch05].

A well-known example of a verifiable secret sharing scheme is Feldman's scheme [Fel87], which uses commitments based on the discrete logarithm problem. In this scheme, for a secret $\alpha = a_0 \in \mathbb{Z}_q$, a dealer broadcasts commitments $C_j = [a_j]G$ for all coefficients of the polynomial $f(x) = \sum_{j=0}^{T-1} a_j x^j$. Each party receiving a private share $\alpha_i = f(i)$ validates it by verifying that:

$$[\alpha_i]G = \sum_{j=0}^{T-1} [i^j]C_j.$$

If this equality does not hold, the participant can conclude that the received share is invalid and reject it.

Feldman's commitments are the ones used in the distributed key generation of FROST (Section 4.4): each participant acts as a dealer, broadcasts $[a_{ij}]G$ for the coefficients of its own polynomial, and every receiver checks its share against them without learning the shares of the others. Because $C_0 = [a_0]G$ is revealed, the commitments are only computationally hiding; Pedersen's commitment scheme [Ped92], which commits with two generators of the group instead of one, makes them perfectly hiding, and Pedersen's VSS uses it for that purpose, but FROST does not need it, since the group public key $\sum_i [a_{i0}]G$ is published in any case. Secret sharing and commitments together form the basis of the verifiable secret sharing used throughout this project.

2.8 Public Key Encryption of Secret Shares

During distributed key generation, each participant must send a secret share to each other participant over a channel relayed by an untrusted coordinator. This calls for public key encryption. We first recall textbook ElGamal, then describe the hashed variant that the implementation actually uses and explain why the change is necessary.

2.8.1 Textbook ElGamal The ElGamal encryption scheme [ElG85] is a public key encryption scheme whose security relies on the difficulty of the decisional Diffie-Hellman problem in a cyclic group. Most presentations write the ciphertext as (c_1, c_2); we write it (U, v) so that the ephemeral group element is uppercase, and so that it is never mistaken for a commitment C_j or for the Schnorr challenge c. Written additively over a group $\mathbb{G}$ of prime order q with generator G, it works as follows.

Key generation: Sample $\mathsf{dk} \xleftarrow{\$} \mathbb{Z}_q$ and publish $\mathsf{ek} = [\mathsf{dk}]G$.

Encryption: To encrypt a message m *represented as an element of the group* $\mathbb{G}$, sample $r \xleftarrow{\$} \mathbb{Z}_q$ and output the ciphertext $(U, v) = \big([r]G,\ m + [r]\mathsf{ek}\big)$.

Decryption: Given (U, v) and dk, compute $m = v - [\mathsf{dk}]U$.

Decryption is correct because $[\mathsf{dk}]U = [\mathsf{dk}][r]G = [r][\mathsf{dk}]G = [r]\mathsf{ek}$, which cancels the masking term applied during encryption.

2.8.2 Hashed ElGamal, and Why We Use It Textbook ElGamal cannot be applied directly here, because the plaintexts we need to encrypt are secret shares, which are *scalars* in $\mathbb{Z}_q$, not points of $\mathbb{G}$. Encoding a scalar as a group element and recovering it afterwards requires an invertible encoding, which for elliptic curves is awkward and not constant-time. The standard remedy, and the one we use, is *hashed ElGamal*, the key-encapsulation form of the scheme that underlies ECIES and DHIES [ABR01]. Instead of masking the message with the group element $[r]\mathsf{ek}$, the sender hashes that element into a symmetric key and masks the message with it:

Key generation: Sample $\mathsf{dk} \xleftarrow{\$} \mathbb{Z}_q$ and publish $\mathsf{ek} = [\mathsf{dk}]G$.

Encryption: To encrypt a scalar $m \in \mathbb{Z}_q$, sample $r \xleftarrow{\$} \mathbb{Z}_q$, set $U = [r]G$, derive $k = \mathsf{KDF}\big(\mathsf{SerializeElement}([r]\mathsf{ek})\big)$, and output $(U, v) = (U,\ \mathsf{SerializeScalar}(m) \oplus k)$.

Decryption: Given (U, v) and dk, derive $k = \mathsf{KDF}\big(\mathsf{SerializeElement}([\mathsf{dk}]U)\big)$ and recover m from $\mathsf{SerializeScalar}(m) = v \oplus k$.

Correctness follows from $[\mathsf{dk}]U = [\mathsf{dk}][r]G = [r][\mathsf{dk}]G = [r]\mathsf{ek}$, so both parties derive the same k. Two implementation points are essential and are stated here once so that Section 4.4.2 can refer to them.

Serialise the point canonically before hashing. $\mathsf{SerializeElement}$ denotes the fixed-length encoding of a curve point, in our case the 33-byte compressed SEC1 encoding [Cer09]. Hashing an in-memory point representation instead is a portability bug waiting to happen, since two implementations may store the same point differently.

The derived key is a one-time pad. It must never be reused. Because KDF output is XORed with the plaintext, encrypting two different shares under

the same ephemeral r towards the same recipient would reveal the XOR of the two shares. Each share therefore gets its own freshly sampled r_{ij}.

We instantiate KDF with SHA-256, whose 256-bit output matches the 32-byte encoding of a scalar in $\mathbb{Z}_q$, so a single hash call suffices and no stream cipher is needed. In a revision of the scheme, the KDF input should additionally be domain-separated and bound to the context of the transmission, that is,

$$k = \mathsf{SHA256}\big(\text{label} \,\|\, \mathsf{SerializeElement}(U) \,\|\, \mathsf{SerializeElement}(\mathsf{ek}) \\ \|\, \mathsf{SerializeElement}([r]\mathsf{ek}) \,\|\, i \,\|\, j\big);$$

binding the recipient's public key and the participant indices into the derivation prevents a relayed ciphertext from being replayed towards a different recipient or reinterpreted as a share for a different index. The current implementation hashes only the shared point, which is sufficient in the analysed setting because the VSS check of Section 4.4 rejects any share that does not match the sender's published commitments at the recipient's own index, but the stronger binding is cheap and we recommend it.

2.8.3 Why the Ciphertext Alone Is Not Enough Hashed ElGamal as described provides confidentiality against a passive relaying coordinator, but it is malleable: flipping a bit of v flips the corresponding bit of the recovered share. It therefore does not, on its own, guarantee that the share a participant decrypts is the share the sender intended. What rules that out is not the encryption but the verifiable secret sharing check performed afterwards, which ties the decrypted value to commitments the sender published to everyone. Confidentiality comes from the encryption, the integrity of an individual share from the VSS check, and the integrity of the transcript as a whole from the device signatures of Section 4.3. Section 6 makes this division of labour explicit, since it is exactly what determines how much the coordinator has to be trusted.

3 FIDO Protocol

FIDO2 (Fast Identity Online 2) [FID22] consists of a group of protocols and standards, such as Web Authentication (WebAuthn) and Client-to-Authenticator Protocol 2 (CTAP2), developed by the FIDO Alliance in collaboration with the World Wide Web Consortium (W3C).

FIDO uses standard public key cryptography techniques to provide safe and passwordless authentication based on the use of passkeys. It supports second-factor and multi-factor user experience with platform authenticators built into a device and unlocked with a biometric or a PIN, or with roaming authenticators such as FIDO security keys, mobile devices and wearables.

3.1 Architecture

There are different parties involved in the FIDO2 protocol [W3C21]:

Relying Party (RP): Online platform or service that a user aims to access. The RP stores the public key corresponding to each user registered, which allows it to authenticate the user by issuing a challenge, requiring them to sign it with the private key, and verifying that the signature is correct with the public key.

Client: Bridge between the RP and the Authenticator. It handles the communication between both parties for the authentication process. This can be a device such as a phone, a laptop, etc.

Authenticator: Device in charge of validating the user's identity by using the private key to sign a challenge issued by an RP. There are two different categories:

Roaming Authenticators: Portable external devices that can work across different platforms.

Platform Authenticators: Embedded within a specific device permanently, acting as a trusted component for authentication.

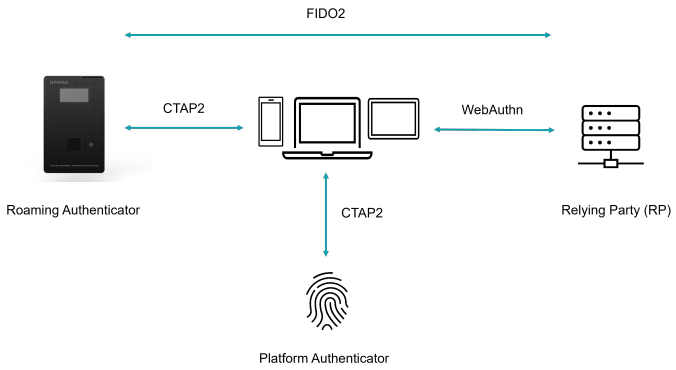


Fig. 1. FIDO2 Ecosystem.

3.2 Client to Authenticator Protocol

The Client to Authenticator Protocol (CTAP) is a protocol that establishes a standardised method for the interaction between a client and authenticator [FID26], in which the client and authenticator need to establish a confidential and mutually authenticated data transport channel beforehand. Moreover, the authenticator is required to have a mechanism to obtain a user gesture, such as a consent button, password, PIN, or biometric.

The protocol consists of three parts:

Authenticator API: At this level, operations are defined similarly to API calls, accepting input parameters and returning either an output or an error code. This API is conceptual; each platform provides actual APIs.

Message encoding: To invoke a method in the authenticator API, the host needs to construct and encode a request and send it to the authenticator, which then processes it and returns an encoded response.

Transport-specific binding: For each transport technology, bindings are specified for this protocol.

3.3 Specification Versions

FIDO2 is a moving target, and it is worth recording which versions this report describes. WebAuthn Level 3 [W3C26] became a W3C Recommendation on 25 August 2026, superseding Level 2 [W3C21], which the prototype was developed against; Level 3 adds, among other things, credential-management and hints machinery that does not affect the algorithm negotiation this report depends on. On the CTAP side, version 2.3 [FID26] is the current Proposed Standard; the material described below is stable across 2.0 through 2.3. Readers implementing against these specifications should check the current versions, but the compliance analysis of Section 4.9 turns on the COSE algorithm registry rather than on the specification revision, and is therefore stable.

3.4 Web Authentication

Web Authentication API, also known as WebAuthn, is a specification by W3C and FIDO. It allows servers to register and authenticate users by means of public key cryptography instead of a password [Duo19]. Instead of a password, a private-public key pair, known as a credential, is created, with the private key safely stored at the user's device and the public key sent and stored at the server, which can then use that public key to verify the user's identity.

3.5 Signature Algorithms and How WebAuthn Encodes Them

For this project the decisive part of the WebAuthn specification is not the message flow but the way it names and encodes signature algorithms, since that is what ultimately decides whether a threshold-produced credential can be used at all. This subsection collects those details in one place; Section 4.9 draws the conclusions.

When a relying party asks for a credential, the `PublicKeyCredentialCreationOptions` object carries a field `pubKeyCredParams`, an ordered list of the algorithms the relying party is willing to accept, each identified by a COSE algorithm identifier, a small negative integer from the IANA COSE Algorithms registry [IAN]. The authenticator picks the first entry it supports and, if it supports none, aborts with `CTAP2_ERR_UNSUPPORTED_ALGORITHM`. The public

key it returns is not a bare point but a *COSE_Key* structure, CBOR-encoded inside `authenticatorData`, whose fields state the key type (`kty`), the algorithm (`alg`), the curve (`crv`), and the coordinates. Verification at the relying party is then driven entirely by the `alg` value.

Table 2 lists the identifiers relevant to this work.

Table 2. COSE algorithm identifiers relevant to a threshold FIDO2 authenticator.

Name	ID	Signature scheme	Practical status in WebAuthn
`ES256`	`-7`	ECDSA over P-256	Universally supported; the de facto default
`RS256`	`-257`	RSASSA-PKCS1-v1_5	Widely supported, mainly by platform authenticators
`EdDSA`	`-8`	Ed25519 (a Schnorr signature)	Registered; implemented by some authenticators, accepted by a minority of relying parties
`ES256K`	`-47`	ECDSA over `secp256k1`	Registered by RFC 8812 [Jon20]; we know of no relying-party support
—	—	Schnorr over `secp256k1`	**No registered identifier exists**

The last row is the one that matters. The prototype described in this report produces Schnorr signatures over `secp256k1`, and there is no COSE identifier for that combination: `ES256K` is ECDSA over the same curve, not Schnorr, so reusing it would produce a credential that a relying party would attempt to verify with the wrong algorithm and reject.

Two consequences follow. Schnorr signatures are not excluded from WebAuthn in principle, since `EdDSA` is registered and Ed25519 is itself a Schnorr signature; the problem is the specific pairing of scheme and curve that this prototype produces. But being registered is not the same as being accepted: a relying party that lists only `ES256` in `pubKeyCredParams` will never negotiate anything else, so the reach of any `EdDSA`-based route is a question about deployed services rather than about standards. Section 4.9 works through both.

3.6 Registration

The FIDO registration process allows users to create and link a new passkey credential to their account with a Relying Party. The process involves the user, a device equipped with a FIDO authenticator, and the RP's server, and includes verifying the user's identity through an authentication method such as a fingerprint or a PIN code.

The registration flow is the following [Duo19]:

0. The user starts the registration on the RP's website or application, providing a username, after which the client sends a request to register an authenticator.
1. The RP's server generates a `PublicKeyCredentialCreationOptions` object containing information about the user's account, the RP's details, and the properties of the new credential, and sends it to the client.
2. The client communicates with the available FIDO authenticator and forwards relevant information from `PublicKeyCredentialCreationOptions`, and a hash of the serialised client data `clientDataHash`, containing the challenge, to this authenticator.
3. The authenticator prompts the user for authentication, and if it is successful, then the authenticator is authorised to generate a new passkey, credential ID, and attestation data, which proves the type and characteristics of the authenticator.
4. The authenticator returns an `attestationObject` to the client, which is an object that contains the generated public key, the credential ID, and attestation data including an attestation signature that ties the public key to the authenticator and the relying party [W3C21].
5. The client sends the `attestationObject`, whose authenticator data contains the new credential's identifier and public key, to the RP's server together with `clientDataJSON`, the serialised client data containing the operation type, the challenge and the origin.
6. The RP's server receives the data and performs some validation steps. This includes checking that `clientDataJSON` carries the expected challenge and origin, verifying the attestation signature and, if the RP chooses to, checking the attestation certificate against the authenticator model's entry in the FIDO Metadata Service. If the validation is successful, the RP securely stores the public key and associates it with the user's account and authenticator characteristics. This enables future authentication using the corresponding secret key held by the authenticator.

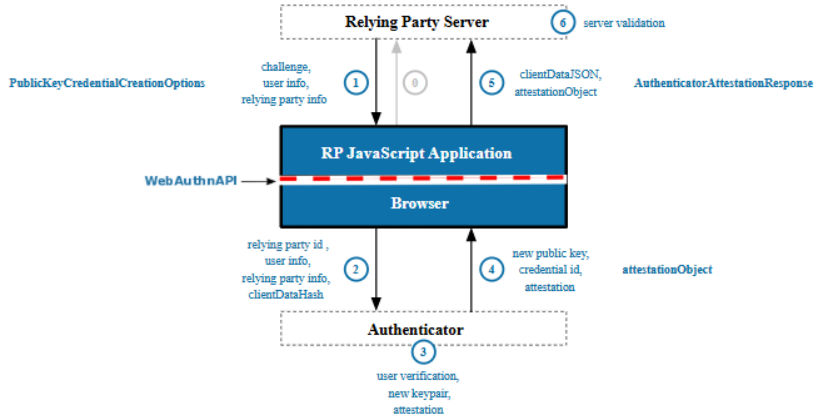


Fig. 2. Registration Flow.

3.7 Authentication

The FIDO authentication process allows users to prove their identity to a Relying Party after they have performed the registration procedure. As in registration, the process involves the user, a device equipped with a FIDO authenticator, and the RP's server. The user provides their username and completes authentication on their authenticator to demonstrate possession of the secret key associated with the registered public key.

The authentication flow is the following [Duo19]:

0. The user starts the authentication on the RP's website or application, providing a username, after which the client sends a request to start the authentication procedure.
1. The RP's server generates a `PublicKeyCredentialRequestOptions` object containing a challenge generated by the RP that will be signed and a list `allowCredentials` of the registered credentials previously registered by the user, and sends it to the client.
2. The client calculates a hash with the client data `clientDataHash` and together with the RP's identifier `RP ID` sends it to the user's authenticator.
3. The authenticator prompts the user to perform user verification, and only then looks for a credential matching the `RP ID`, so that it reveals nothing about which credentials it holds before the user has been verified. If verification is successful, then the authenticator creates an assertion by signing `clientDataHash` and `authenticatorData`, which contains information about the authentication event, with the private key associated with the credential.

4. The authenticator returns the `authenticatorData` and the generated digital signature to the client.
5. The client forwards the `authenticatorData` and the signature back to the RP's server, together with the `clientDataJSON`, which contains the client data that was originally passed to the authenticator, serialised in JSON format.
6. The RP's server receives the data and performs some validation steps. This includes the verification of the signature, the `authenticatorData`, and the `clientDataJSON`, ensuring the challenge and all parameters are as expected. If all these checks are correct, the RP confirms the user's identity and grants them access.

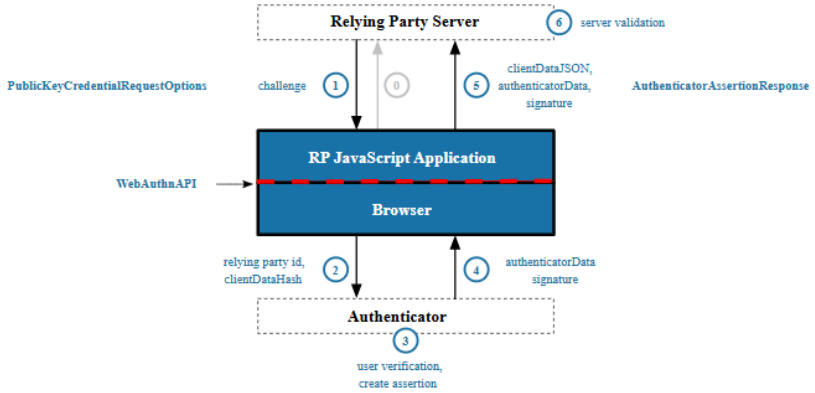


Fig. 3. Authentication Flow.

4 FROST Implementation

This section presents the core of the project: the implementation of the FROST threshold signature scheme. Figure 4 shows the system it builds up to. It first recalls the general structure of a threshold signature scheme and fixes the FROST ciphersuite we use, then describes the distributed key generation, its hardening with end-to-end encrypted shares and a blaming protocol, and the two-round signing procedure. It then covers the proof-of-concept integration with a web server, the WebAuthn experiment and what it teaches about the FIDO2 compliance boundary, and finally the measurements and the information needed to reproduce the setup.

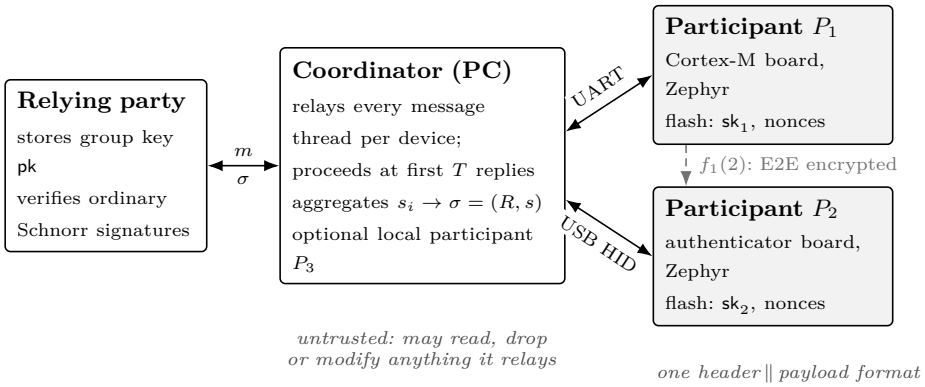


Fig. 4. Overview of the prototype. The coordinator relays all traffic between participants over two transports and aggregates signature shares. Secret shares cross it end-to-end encrypted and every message is signed under the sender's device key, so the coordinator can neither read shares nor alter the transcript. The relying party sees one ordinary public key and signature and is unaware of the threshold structure. Participant P_1 is the STM32 Nucleo-L476RG and P_2 the OFFPAD development board.

4.1 Threshold Signatures

In a threshold signature scheme [BS23, §22.2], the secret signing key sk is divided into N shares distributed among different devices. This avoids a single point of failure, as the full key is never reconstructed in a single place. A valid signature requires cooperation from a subset of at least T out of N devices.

Formally, a threshold signature scheme consists of four algorithms, ($\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Combine}, \mathsf{Verify}$):

Share Generation (KeyGen): A protocol that produces a group public key pk and gives each device i a signing share sk_i, together with the public verification shares $\mathsf{pk}_i = [\mathsf{sk}_i]G$ of all participants. It may be run by a trusted dealer or distributed among the participants.

Share Signing (Sign): For a message m, each device produces a signature share $\sigma_i = \mathsf{Sign}(\mathsf{sk}_i, m, \cdot)$. In an interactive scheme such as FROST, Sign additionally consumes state established in an earlier round.

Share Combination ($\mathsf{Combine}$): The combiner $\mathsf{Combine}(\mathsf{pk}, m, \mathcal{J}, \{\sigma_j\}_{j\in\mathcal{J}})$ collects T signature shares from a coalition $\mathcal{J}$ and combines them into a single signature σ.

Verification (Verify): A deterministic algorithm $\mathsf{Verify}(\mathsf{pk}, m, \sigma)$ that accepts or rejects. Crucially, in FROST this is *ordinary* Schnorr verification against the group public key: a verifier neither knows nor cares that σ was produced by a coalition rather than by a single signer.

The last point is what makes the construction attractive for authentication: the relying party is unchanged and unaware.

4.1.1 Unforgeability, Availability and Robustness

Two guarantees are often conflated, and the distinction determines what a deployment can promise its users.

Unforgeability is the security property: an adversary that corrupts up to $T - 1$ participants, sees their entire state, and interacts with the honest participants in arbitrarily many concurrent signing sessions cannot produce a valid signature on a message the honest participants did not agree to sign. FROST achieves this, and Bellare et al. [BCK+22] showed that it does so in a stronger sense than originally claimed.

Robustness is an availability property: the guarantee that the protocol *terminates with a signature* even if some participants actively misbehave. Plain FROST does **not** provide this. If a participant in the chosen coalition $\mathcal{J}$ submits an invalid signature share, the aggregator's check detects it and identifies the culprit, but the session aborts and must be restarted with a different coalition. This weaker guarantee is called *identifiable abort*. ROAST [RRJ+22] is a wrapper that turns FROST into a genuinely robust protocol by running overlapping coalitions, at the cost of additional sessions.

What our system therefore provides is: signing succeeds whenever T *honest and available* participants can be reached, and a participant that deviates is named rather than merely detected. It does not provide a guaranteed signature in the presence of an actively malicious participant that keeps volunteering for coalitions. The distinction matters in practice: a 2-out-of-3 deployment in which one device is compromised and hostile can be prevented from signing, even though it can never forge.

4.2 The FROST Protocol and Our Ciphersuite

Flexible Round-Optimized Schnorr Threshold, or FROST, is a threshold signature scheme producing ordinary Schnorr signatures [Sch91,KG21]. Its distinguishing property is that signing takes two rounds and that concurrent signing sessions are unrestricted: earlier two-round schemes were vulnerable to the ROS attack [BLL+21] when many sessions ran in parallel, and FROST avoids this by binding each participant's nonce pair to the specific session through a *binding factor* derived from the message and the full commitment list.

Our implementation follows the specification of Connolly, Komlo, Goldberg, and Wood [CKGW24], RFC 9591, using the ciphersuite

`FROST(secp256k1, SHA-256)`,

one of the five ciphersuites the RFC defines, alongside `FROST(Ed25519, SHA-512)`, `FROST(ristretto255, SHA-512)`, `FROST(Ed448, SHAKE256)`, and

`FROST(P-256, SHA-256)`. The choice of `secp256k1` was driven by the availability of a C library suitable for porting to Zephyr (Section 4.11) rather than by the application; Section 4.9 examines what the other ciphersuites would have bought us.

A ciphersuite fixes the group, the point and scalar encodings, and a family of domain-separated hash functions built from the underlying hash. All of them are SHA-256 in our case, prefixed with a context string Φ and a per-function label, so that a digest computed for one purpose can never be reinterpreted as a digest computed for another:

H_1 derives the binding factor of the second signing round;

H_2 derives the Schnorr challenge c;

H_3 derives signing nonces from fresh randomness and the signing share;

H_4 hashes the message;

H_5 hashes the encoded commitment list.

Domain separation is not a formality. If H_1 and H_2 were the same function, an adversary that controls the message could aim for a collision between a binding factor and a challenge, and the security proof would no longer apply. An implementation that instantiates all five as bare `SHA256` calls is not implementing RFC 9591, even if it interoperates with itself.

FROST can be run with every participant broadcasting to all others, or with a designated *signature aggregator*, a semi-trusted role that any participant or an external party can assume. We adopt the aggregator variant: the aggregator orchestrates communication, identifies misbehaving participants, and publishes the final group signature. Even if the aggregator deviates from the protocol, it cannot learn the private key or forge signatures. It can, however, affect availability, and Section 6 analyses what else it can and cannot do once every message is authenticated. The complete implementation is publicly available at `https://github.com/JavierGarcia22/FROST_Threshold`.

4.3 Message Authentication

Both protocol phases run through the coordinator, which is not a trusted channel: it can drop, delay, reorder or modify anything it relays. A distributed key generation is analysed under the assumption of an authenticated broadcast channel, and a single relaying computer provides nothing of the sort by itself, so we authenticate the traffic rather than the channel.

Each device holds a long-term signing key pair, separate both from its FROST signing share and from the encryption key pair that protects shares in transit, and the list of the N device public keys is provisioned to every device out of band before the first protocol run. Every message a device sends, in either phase, carries an ordinary Schnorr signature over `secp256k1` under that key, computed

over the header and the payload. The signature is 64 bytes, a 32-byte nonce point and a 32-byte scalar, and the receiver verifies it against the provisioned list before the payload is parsed. A message that fails verification is discarded and the session aborts.

Three consequences follow, and the security analysis of Section 6 rests on them. The coordinator cannot equivocate, because presenting different commitment sets to different participants requires messages that no device key covers. It cannot substitute an encryption key of its own for ek_j and read the shares addressed to P_j, because the encryption keys are broadcast signed like everything else. Nor can it fabricate a complaint or tamper with a ciphertext in transit without the tampering surfacing as an invalid signature rather than as an invalid share, which is what makes the blaming protocol below sound rather than merely public. What the coordinator retains is the ability to stop the protocol.

The provisioned list is therefore the root of trust of the whole system, and it is worth being explicit that the prototype establishes it out of band and by hand: whoever controls that step controls which devices are in the group, and a device whose list has been tampered with will accept messages signed by the tamperer's keys. A product needs an enrolment procedure that establishes the list under the user's control and then fixes it on the device; this is the counterpart of credential enrolment and attestation in FIDO2, and it is the largest piece of design this report leaves open (Section 6.2).

4.4 Distributed Key Generation

Key generation can be achieved via two approaches:

Trusted dealer: A single device generates the key and all shares and distributes them using Shamir's secret sharing together with a VSS commitment vector. This is simpler and cheaper for the participants, but it creates a single point of failure: an attacker who corrupts the dealer at the moment of generation obtains the full secret key, and the dealer must be trusted to erase it afterwards.

Distributed key generation (DKG): Participants collaboratively generate the key so that no single party ever holds it, at the cost of an additional round and per-participant verification work.

The trusted dealer contradicts the very property that motivates the system, so we implement a distributed key generation. RFC 9591 leaves key generation out of scope and specifies only a trusted dealer (its Appendix C), so we use the two-round DKG of the original FROST paper [KG21], which is Pedersen's distributed key generation, as analysed by Gennaro et al. [GJKR99], with a proof of knowledge added against rogue-key attacks: every participant acts as a dealer of its own polynomial and the group key is the sum of everyone's constant terms. ChillDKG [RNM+24], a DKG designed specifically for FROST, removes the need for a broadcast channel, the same problem Section 4.3 addresses, by having

participants certify a common transcript at the end of the protocol; we chose instead to sign every message under a device key, because the same signatures also make the blaming protocol of Section 4.4.3 sound.

4.4.1 The Protocol Each participant P_i executes the following steps.

1. Sample T random coefficients $(a_{i0}, a_{i1}, \ldots, a_{i(T-1)}) \xleftarrow{\$} \mathbb{Z}_q$, where a_{i0} is P_i's secret contribution to the group key.
2. Define the polynomial of degree $T-1$
$$f_i(x) = a_{i0} + a_{i1}x + \cdots + a_{i(T-1)}x^{T-1}.$$
3. Compute a proof of knowledge $\sigma_i = (K_i, \mu_i)$ of a_{i0}, where $K_i = [k]G$ for a fresh $k \xleftarrow{\$} \mathbb{Z}_q$ and $\mu_i = k + a_{i0} \cdot h_i$, with $h_i = \mathsf{H}(i, \Phi, [a_{i0}]G, K_i)$ for the context string Φ.
4. Compute public commitments $C_{ij} = [a_{ij}]G$ for $j = 0, 1, \ldots, T-1$.
5. Broadcast $(i, C_{i0}, \ldots, C_{i(T-1)}, K_i, \mu_i)$.
6. Verify every other participant's proof by checking $[\mu_j]G \stackrel{?}{=} K_j + [h_j]C_{j0}$, where h_j is recomputed from the received values. Abort on failure; otherwise, discard the proofs.
7. Send the secret share $f_i(j)$ to each participant P_j, encrypted as described in Section 4.4.2, erase the plaintext after sending, and retain $(i, f_i(i))$.
8. Verify every received share against the sender's commitments: $[f_j(i)]G \stackrel{?}{=} \sum_{k=0}^{T-1} [i^k]C_{jk}$, raising a complaint (Section 4.4.3) if the check fails.
9. Compute the signing share $\mathsf{sk}_i = \sum_{j=1}^{N} f_j(i)$ and erase the individual summands $f_j(i)$.
10. Compute the group public key and every participant's public verification share from the commitments alone:
$$\mathsf{pk} = \sum_{j=1}^{N} C_{j0}, \qquad \mathsf{pk}_\ell = \sum_{j=1}^{N} \sum_{k=0}^{T-1} [\ell^k] C_{jk},$$
the latter for every $\ell = 1, \ldots, N$.

The final step is worth spelling out, because it is easy to compute $\mathsf{pk}_i = [\mathsf{sk}_i]G$ for oneself and then be unable to check anyone else's signature share. The identity $\mathsf{pk}_\ell = [\mathsf{sk}_\ell]G$ holds because $\mathsf{sk}_\ell = \sum_j f_j(\ell)$ and each $[f_j(\ell)]G$ equals $\sum_k [\ell^k]C_{jk}$ by construction. Every party that saw the broadcast commitments can therefore derive the verification share of every participant without any additional communication, which is exactly what the aggregator needs in step 7 of signing.

This key generation still has an important weakness in our setting. Having one device orchestrate the exchange means that all broadcast material and all shares pass through the coordinating computer, and the DKG as specified assumes an authenticated broadcast channel that a single relaying device does not provide. Two separate problems follow: the coordinator could read the shares it relays, and it could deliver inconsistent broadcasts to different participants. The first is solved below by encrypting the shares end-to-end. The second is not solved by encryption at all; it is solved by the message authentication of Section 4.3, and Section 6.2 analyses what remains once both are in place.

4.4.2 End-to-End Encryption of Shares To prevent the coordinator from reading the shares it relays, every share is encrypted under the hashed-ElGamal scheme of Section 2.8.2 before it leaves the sending device. Each participant P_i generates an encryption key pair, private $\mathsf{dk}_i \xleftarrow{\$} \mathbb{Z}_q$ and public $\mathsf{ek}_i = [\mathsf{dk}_i]G$, and the public keys are broadcast through the coordinator during the first round. Like every other message, this broadcast is signed under the sender's device key and verified against the provisioned key list (Section 4.3), which is what prevents the coordinator from substituting an encryption key of its own and reading the shares addressed to P_j. The construction differs from DHIES in one detail: the masking key is derived from the shared point alone, without U_{ij} in the hash input. Since every U_{ij} is fresh nothing is lost, but an interoperating implementation must match the choice.

Encryption (sender P_i to receiver P_j):

1. For the share $s_{ij} = f_i(j)$, sample a fresh ephemeral scalar $r_{ij} \xleftarrow{\$} \mathbb{Z}_q$.
2. Compute the ephemeral public key $U_{ij} = [r_{ij}]G$.
3. Derive the masking key from the Diffie-Hellman point:

$$\begin{aligned}\mathsf{shared}_{ij} &= \mathsf{SHA256}\big(\mathsf{SerializeElement}([r_{ij}]\mathsf{ek}_j)\big)\\ &= \mathsf{SHA256}\big(\mathsf{SerializeElement}([r_{ij}][\mathsf{dk}_j]G)\big),\end{aligned}$$

   where $\mathsf{SerializeElement}$ is the 33-byte compressed SEC1 encoding and ek_j is the encryption key of P_j.
4. Encrypt the share: $v_{ij} = \mathsf{SerializeScalar}(s_{ij}) \oplus \mathsf{shared}_{ij}$, a 32-byte value.
5. Send the ciphertext (U_{ij}, v_{ij}) to the coordinator, store r_{ij} until the key generation is confirmed, and erase the plaintext s_{ij} from memory.

Decryption (receiver P_j):

1. Receive the ciphertext (U_{ij}, v_{ij}) attributed to P_i, and check that U_{ij} is a valid point of $\mathbb{G}^*$, rejecting the identity.

2. Recompute the masking key

$$\begin{aligned}\mathsf{shared}_{ij} &= \mathsf{SHA256}\big(\mathsf{SerializeElement}([\mathsf{dk}_j]U_{ij})\big)\\ &= \mathsf{SHA256}\big(\mathsf{SerializeElement}([\mathsf{dk}_j][r_{ij}]G)\big)\\ &= \mathsf{SHA256}\big(\mathsf{SerializeElement}([r_{ij}]\mathsf{ek}_j)\big).\end{aligned}$$

3. Decrypt the share: $\mathsf{SerializeScalar}(s_{ij}) = v_{ij} \oplus \mathsf{shared}_{ij}$, and reject if the result does not decode to a scalar in $\mathbb{Z}_q$ or is zero.

4. Verify the share against the sender's VSS commitments:

$$[s_{ij}]G \stackrel{?}{=} \sum_{k=0}^{T-1} [j^k]C_{ik}.$$

   If the check fails, raise a complaint against P_i; otherwise accept s_{ij}.

Note that r_{ij} must be retained by the sender until the key generation completes, because it is the sender's only way to answer a complaint (Section 4.4.3); it must be erased once the DKG is confirmed, since together with the ciphertext it reveals the share.

The XOR step deserves one remark. For an honestly formed ciphertext the XOR recovers exactly $\mathsf{SerializeScalar}(s_{ij})$, which always decodes. But a corrupted or forged ciphertext unmasks to a uniformly random 32-byte string, and since q is slightly smaller than 2^{256} such a string fails to decode to an element of $\mathbb{Z}_q$ with probability about 2^{-128}. The recipient must therefore treat "does not decode" as a possible outcome rather than as an impossible one, which is the reason for the explicit check in step 3 of decryption; the probability is of no cryptographic consequence, but an implementation that omits the check fails far from the cause when it does occur.

4.4.3 Blaming The VSS check tells a participant that a received share is wrong, but not who is at fault: the sender may have sent a bad share, or the receiver may be fabricating the accusation to have an honest participant excluded. Since the share travelled encrypted, no third party can adjudicate by inspection. We therefore implement a blaming protocol that resolves the dispute cryptographically, distinguishing a false accusation (**scenario 1**) from a legitimate complaint (**scenario 2**).

Phase 1 (Complaint): Accuser P_j claims that the share s_{ij} received from P_i is invalid, publishing the indices (i, j) and the ciphertext (U_{ij}, v_{ij}) it received.

Phase 2 (Blame proof construction): P_i reveals the ephemeral scalar r_{ij} it used for that ciphertext by broadcasting the blame proof $\pi_{ij} = (i, j, r_{ij}, U_{ij}, v_{ij})$. This exposes only this one share.

Phase 3 (Verification): Any party, including the coordinator, can now check:

1. If $[r_{ij}]G \neq U_{ij}$, the revealed ephemeral key does not match the ciphertext; P_i **is dishonest**.
2. Otherwise compute $\mathsf{shared}'_{ij} = \mathsf{SHA256}(\mathsf{SerializeElement}([r_{ij}]\mathsf{ek}_j))$ and s'_{ij} from $\mathsf{SerializeScalar}(s'_{ij}) = v_{ij} \oplus \mathsf{shared}'_{ij}$.
3. Check that s'_{ij} decodes to a nonzero element of $\mathbb{Z}_q$ and matches the VSS commitments P_i broadcast: $[s'_{ij}]G \stackrel{?}{=} \sum_{k=0}^{T-1} [j^k]C_{ik}$.
4. If all checks pass, the share was valid: P_i **is honest** and P_j lied. If any check fails, P_i **is dishonest**.

The adjudication is public and verifiable by anyone holding the broadcast commitments, so it does not require trusting the coordinator to judge correctly; the coordinator merely relays. The procedure assumes that the ciphertext presented in phase 1 is the one P_i actually sent, and this is what the message authentication of Section 4.3 secures: every ciphertext carries the sender's device signature, so an accuser cannot present a ciphertext its sender did not produce, and a coordinator that tampers with one in transit produces a message that fails signature verification rather than a false accusation. A party that does not answer a complaint at all is treated as being at fault. Every outcome is therefore an identifiable abort: the key generation is discarded with the responsible party named, which is a denial of service rather than a compromise. Section 6.3 analyses what a published blame proof reveals.

4.4.4 Local Participants The implementation allows not only embedded devices to participate, but also the coordinating computer, which can hold a secret share and take part in the protocol. We call such a participant a *local participant*, in contrast to the board participants.

This decouples N from the number of boards physically available. Rather than being limited to N equal to the number of connected embedded devices, any number of local participants can be added on the coordinating computer to experiment with different (T, N) without additional hardware. The tests and evaluation reported here use $N = 3$, $T = 2$, with one local participant and two board participants, the Nucleo and the OFFPAD.

A local participant behaves exactly like a board participant: it samples its polynomial, encrypts and verifies shares, generates nonces, and produces a signature share. It is, however, a testing device and not a security feature: a local participant's share lives in the memory of the same machine that relays all traffic, so a coordinator holding a share is a participant that is trivially co-located with the relay. Threshold configurations whose security argument depends on the independence of the participants should count only the boards.

4.4.5 State Each Device Retains At the end of a successful key generation, each board persists exactly the following to flash, and nothing else:

the participant index i, which is needed to compute Lagrange coefficients and to identify itself in the message header;

the signing share sk_i, a single 32-byte scalar, which is the only long-term secret;

the group public key pk and the commitment vectors, from which all verification shares pk_ℓ can be recomputed on demand rather than stored.

Everything else, the polynomial coefficients a_{ij}, the individual received shares $f_j(i)$, the ephemeral scalars r_{ij}, and the encryption key dk_i once the DKG is confirmed, is transient and must be erased. Keeping any of the $f_j(i)$ would be particularly damaging: a device holding both its own polynomial and the individual contributions of others gives an attacker who compromises it strictly more than the threshold model assumes.

4.5 Signing

FROST signs in two rounds: a commitment round in which each participant publishes a pair of nonce commitments, and a signing round in which the coalition and the message become known and each participant returns a single scalar.

4.5.1 Round 1: Nonce Generation and Commitments Each participant P_i prepares $\pi \geq 1$ nonce/commitment pairs, indexed by j, and keeps a list L_i:

1. Generate single-use nonces $(d_{ij}, e_{ij}) \in \mathbb{Z}_q^* \times \mathbb{Z}_q^*$.
2. Derive the commitment shares $(D_{ij}, E_{ij}) = ([d_{ij}]G, [e_{ij}]G)$.
3. Append (D_{ij}, E_{ij}) to L_i and store (d_{ij}, D_{ij}), (e_{ij}, E_{ij}) for later use.
4. Publish (i, L_i).

Remark on nonce generation. The formulation above follows the original FROST presentation [KG21], in which nonces are sampled uniformly and π pairs may be precomputed in a batch. RFC 9591 differs in two ways that an implementation targeting the RFC must follow. First, it specifies exactly one nonce pair per signing session and drops the batched preprocessing phase. Second, it does not sample nonces uniformly but derives them as

$$d_i \leftarrow \mathsf{H}_3\big(rnd \,\|\, \mathsf{SerializeScalar}(\mathsf{sk}_i)\big), \qquad rnd \xleftarrow{\$} \{0,1\}^{256},$$

and likewise for e_i, with fresh randomness rnd each time. (We write rnd rather than the RFC's `random_bytes` to avoid a clash with the binding factor ρ_i below.) Mixing the signing share into the derivation is a cheap hedge against *biased or short* randomness: the nonce then still depends on a secret the attacker lacks and differs between devices. It is not a defence against randomness that *repeats exactly*, since the same rnd and the same sk_i give the same nonce, so the hedge complements a working entropy source rather than replacing one. Note also that

this is emphatically *not* the same as deriving nonces deterministically from the message, as Ed25519 does; RFC 9591 warns that fully deterministic derivation enables a complete key-recovery attack in the multi-party setting, because a malicious coordinator can replay the same session with different commitment lists. Our prototype samples nonces uniformly from the hardware entropy source; adopting the RFC's derivation on top is the hardening step listed in Table 14.

Given the flash persistence discussed in Section 6.4, we recommend $\pi = 1$ together with the RFC's derivation: batching multiplies the number of unused nonce pairs sitting in flash, and every one of them is a value whose reuse would expose sk_i.

4.5.2 Round 2: Signature Shares and Aggregation Once the preparation stage is finished, a coalition $\mathcal{J}$ with $|\mathcal{J}| = T$ signs a message m:

1. The aggregator fetches the next unused commitment pair (D_i, E_i) for each $P_i \in \mathcal{J}$ from L_i, dropping the pair index j from here on since each pair is used only once, constructs the commitment list $B = \langle (i, D_i, E_i) \rangle_{i \in \mathcal{J}}$ sorted by identifier, and sends each P_i the tuple (m, B).
2. Each participant validates the message m against its own policy, checks that B contains exactly T entries with distinct identifiers, that its own entry in B matches a pair it holds and has not yet used, and that $D_j, E_j \in \mathbb{G}^*$ for every entry, aborting otherwise.
3. Each participant computes the binding factor ρ_j of every $j \in \mathcal{J}$, the group commitment, and the challenge:

$$\begin{aligned} \rho_j &= \mathsf{H}_1\big(\mathsf{SerializeElement}(\mathsf{pk}) \,\|\, \mathsf{H}_4(m) \,\|\, \mathsf{H}_5(B) \,\|\, \mathsf{SerializeScalar}(j)\big), \\ R &= \sum_{j \in \mathcal{J}} \big(D_j + [\rho_j] E_j\big), \\ c &= \mathsf{H}_2\big(\mathsf{SerializeElement}(R) \,\|\, \mathsf{SerializeElement}(\mathsf{pk}) \,\|\, m\big). \end{aligned}$$

4. Each participant P_i computes its signature share

$$s_i = d_i + (e_i \cdot \rho_i) + \lambda_i \cdot \mathsf{sk}_i \cdot c \pmod{q},$$

   where λ_i is the Lagrange coefficient of P_i at 0 for the coalition $\mathcal{J}$, as defined in Section 2.6.
5. Each participant erases the used nonce pair (d_i, e_i) *before* returning s_i to the aggregator; the ordering is a security requirement, not a matter of style, and Section 6.4 explains why.
6. The aggregator recomputes $\rho_i = \mathsf{H}_1(\cdot)$, $R_i = D_i + [\rho_i] E_i$, $R = \sum_{i \in \mathcal{J}} R_i$ and $c = \mathsf{H}_2(\cdot)$ exactly as the participants did.
7. The aggregator checks each share against the corresponding verification share, $[s_i]G \stackrel{?}{=} R_i + [c \cdot \lambda_i]\mathsf{pk}_i$, and identifies as faulty every P_i for which the check fails.

8. If all checks pass, the aggregator computes $s = \sum_{i \in \mathcal{J}} s_i \bmod q$ and outputs $\sigma = (R, s)$ together with m.

Correctness is a short computation. Writing $\mathsf{sk} = \sum_{i \in \mathcal{J}} \lambda_i \mathsf{sk}_i$ for the group secret, which is what Lagrange interpolation guarantees for any coalition of size T, and $r = \sum_{i \in \mathcal{J}} (d_i + e_i \rho_i)$ for the aggregate nonce with $R = [r]G$, we get

$$s = \sum_{i \in \mathcal{J}} \big(d_i + e_i \rho_i + \lambda_i \mathsf{sk}_i c\big) = r + c\mathsf{sk},$$

so (R, s) satisfies the Schnorr verification equation $[s]G = R + [c]\mathsf{pk}$ of Section 2.4. The verifier performs exactly that check and needs to know nothing about T, N, or $\mathcal{J}$.

4.5.3 Abort Semantics Each check above has a defined consequence, and an implementation that treats them uniformly as "return error" loses the identifiability that motivates them.

A malformed or unused-nonce mismatch in step 2 means the aggregator sent something inconsistent with what this participant published. The participant refuses and reports the aggregator; it must *not* sign with a different pair, since signing twice against overlapping commitment lists is the situation the binding factor exists to control.

A failed share check in step 7 identifies a specific P_i. The session aborts; the aggregator restarts with a coalition excluding P_i, if T other participants remain. All nonce pairs used in the aborted session are consumed and must not be reused, including those of honest participants.

A timeout is indistinguishable from a participant being unavailable and is handled by choosing a different coalition, which is precisely the situation the threshold is for.

4.6 A Walkthrough of One Complete Run

The preceding subsections specify each phase in isolation. This one puts them in order, because the question a reader implementing the system actually has is what happens, in what sequence, on which device. Figure 4 shows the participants; Table 3 shows the traffic between them for a 2-out-of-3 configuration with two boards and one local participant.

Three properties of this sequence are worth drawing out, because they are what an implementation has to preserve.

Key generation is a two-round barrier; signing is not. Every participant must publish its commitments before any share can be verified, so the coordinator cannot begin round two of the DKG until all N participants have completed round one. Signing has no such barrier: the coordinator

Table 3. One complete run, from an unprovisioned device to a verified signature. "C" is the coordinator, P_1 and P_2 the boards. Message counts are for $N = 3$, $T = 2$.

Phase	**Direction**	**Content**
Key generation, once per credential		
DKG round 1	$P_i \to$ C	Encryption key ek_i, commitments $C_{i0..i(T-1)}$, proof of knowledge (K_i, μ_i)
DKG round 1	C $\to P_j$	The same, relayed to every other participant
(each device)	—	Verify $N-1$ proofs of knowledge; abort on failure
DKG round 2	$P_i \to$ C	$N-1$ encrypted shares (U_{ij}, v_{ij}), one per peer
DKG round 2	C $\to P_j$	Each ciphertext relayed to its intended recipient only
(each device)	—	Decrypt, check each share against the sender's commitments, complain on failure (Section 4.4.3)
(each device)	—	Compute sk_i, derive pk and all pk_ℓ, write to flash, erase everything else
Registration, once		
—	C $\to$ server	Username and the group public key pk
Signing, once per authentication		
Challenge	server $\to$ C	32-byte challenge
Round 1	$P_i \to$ C	Nonce commitments (D_i, E_i) and the public key structure
Round 2	C $\to P_i$	Message m and the commitment list B of the chosen coalition
Round 2	$P_i \to$ C	Signature share s_i, after invalidating the used nonce pair in flash
Aggregate	—	C checks each s_i, sums them, outputs $\sigma = (R, s)$
Verify	C $\to$ server	σ, verified as an ordinary Schnorr signature against pk

proceeds as soon as T commitments have arrived (Section 5), which is why the concurrency strategy pays off in signing and not in key generation.

The device is stateless between phases only in one direction. After key generation a board holds exactly the state of Section 4.4.5 and nothing else. After signing round one it additionally holds an unused nonce pair in flash, and that pair is the one piece of state whose mishandling is catastrophic (Section 6.4).

Only the last row leaves the user's side. Everything above the final line happens between devices the user controls. The server sees a public key at registration and a signature at authentication, and nothing else.

4.7 Registration and Authentication with FROST Signatures

Building on the working FROST implementation, this section describes a Python-based test server for user registration and authentication.

This test environment consists of:

Flask web server: Handles HTTP requests, user registration, challenge generation, and signature verification.

Web frontend: Provides the user interface and manages real-time communication.

WebSocket bridge: Coordinates communication between the browser and the native executables.

Executables: Native programs that perform the threshold cryptographic operations (DKG and signing).

4.7.1 Registration For registration, the frontend presents a form for registration with the FROST protocol. The user provides only a username and a display name; since the scheme is based on threshold signatures, no password is required. Upon confirmation, a terminal interface guides the user through the DKG, selecting the participating devices, extracting the group public key, and registering it with the server together with the username and display name.

The corresponding backend workflow is the following:

1. The server runs at `localhost:5000`; users access the website at `https://localhost:5000`.
2. After clicking the button for registration with FROST, the server checks if the WebSocket is running.
3. If the WebSocket bridge is running (at `localhost:9999`), the server connects to it.

Algorithm 1 Create WebSocket connection.

```
CONNECT to WebSocket at ws://localhost:9999

if receive 'output' from WebSocket then
    DISPLAY terminal line in browser
end if

if receive 'success' from WebSocket then
    PROCEED to registration
end if
```

4. The server sends the command `run_dkg` to the WebSocket.
5. The WebSocket bridge launches a batch file in a new terminal window to perform the key generation.

Algorithm 2 Launch of the batch file for the DKG.

```
CREATE temporary output file

CREATE batch script:
  CHANGE directory to project folder
  EXECUTE presign_pc.exe (key generation)
  CAPTURE output to temporary file

LAUNCH batch script in a NEW terminal window
```

6. The WebSocket bridge monitors the output, extracts the group public key, and sends it to the server.
7. The server registers the user's data, including the group public key obtained from the key generation.

Algorithm 3 User registration.

```
if DKG success then
    EXTRACT group_public_key from payload

    SEND POST to /api/frost/register WITH:
       username, display_name, group_public_key

    if registration is successful then
        DISPLAY "User registered successfully!"
    end if
end if
```

With this implementation, users do not need a password; instead, their secret key is stored in shares on the devices they selected during the distributed key generation. Note that the server is agnostic to the signature being produced by a threshold scheme, as it only receives an ordinary public key. The threshold protocol runs entirely on the user's side. We considered two implementations for this, a browser extension and a WebSocket bridge, and chose the WebSocket approach for ease of implementation; Section 6.6 explains why this choice must not survive into a deployment.

4.7.2 Authentication The authentication process closely mirrors registration.

1. The server runs at `localhost:5000`; users access the website at `https://localhost:5000`.
2. After the user initiates authentication with FROST, the server processes the request and creates a challenge to be signed.

Algorithm 4 Authentication request.

```
RECEIVE POST to /api/frost/authenticate/begin WITH username

if user DOES NOT exist then
    RETURN error 404
end if

GENERATE 32-byte random cryptographic challenge
CONVERT challenge to hex format

STORE challenge and username in a temporary session
RETURN challenge to frontend
```

3. If the WebSocket bridge is running (at `localhost:9999`), the server connects to it and forwards the challenge to be signed.

Algorithm 5 WebSocket connection for authentication.

```
STORE server challenge
CONNECT to WebSocket at ws://localhost:9999

if receive 'output' from WebSocket then
    DISPLAY terminal line in browser
end if

if receive 'success' from WebSocket then
    EXTRACT signature from payload
end if
```

4. The server sends the command `run_signing` to the WebSocket.
5. The WebSocket bridge launches a batch file in a new terminal window to perform the signing, passing the challenge as an argument to the program.
6. The WebSocket bridge monitors the output, extracts the final signature, and sends it to the server.
7. The server verifies the signature against the generated challenge; if it is valid, the user is authenticated.

Algorithm 6 Signature verification.

```
RECEIVE POST to /api/frost/authenticate/complete WITH signature

RETRIEVE challenge and username from session
if no active challenge then
    RETURN error 400
end if

FETCH user group_public_key from database

COMPUTE hash of the challenge
VERIFY Schnorr signature using:
  signature, challenge hash, and group_public_key

if signature is valid then
    CLEAR session challenge (to prevent replay attacks)
    RETURN "Authentication successful"
else
    RETURN error 401
end if
```

This demonstrator establishes that the cryptography works end to end: a user with two of three devices present can log in to a server that stores nothing but a public key. What it does not establish is FIDO2 compliance, since the server is one we wrote and the verification routine is one we chose. The next two subsections examine what happens when the server is a standard WebAuthn relying party instead.

4.8 Registering a FROST Key with WebAuthn

Although the prototype is not FIDO2-compliant, we experimented with driving a standard WebAuthn registration and substituting the group public key into the resulting credential, in order to locate precisely where the incompatibility bites. The frontend remains unchanged.

The backend workflow is the following:

1. Steps 1–6 are as in Section 4.7: the bridge runs the DKG and returns the group public key.
2. The server drives a normal WebAuthn registration against the platform authenticator of the coordinating computer, obtaining a genuine credential and its attestation object.

Algorithm 7 Credential creation.

```
REQUEST platform authenticator to create new credential
EXTRACT attestation object from the generated credential
```

3. The WebAuthn-generated public key inside the attestation object is replaced with the FROST group public key. Since JavaScript's CBOR handling does not treat the negative integer labels of a COSE_Key correctly, the CBOR bytes for the key are constructed manually.

Algorithm 8 Key replacement.

```
DECODE attestation object from CBOR format
SKIP relying party hash, flags, and counters
EXTRACT AAGUID and Credential ID

SPLIT 64-byte FROST public_key into:
    X_Coordinate (first 32 bytes)
    Y_Coordinate (last 32 bytes)

CONSTRUCT new COSE key manually (CBOR format)
```

4. The new COSE key is packed together with rebuilt authenticator data into the attestation object and sent to the server to complete the registration.

Algorithm 9 Registration of new attestation object.

```
REBUILD authenticator data using the new FROST COSE key
PACK data back into the attestation object

SEND POST to /api/frost/register/complete WITH:
    username
    group_public_key
    MODIFIED attestation_object
    client_data
```

The experiment fails, and it is instructive to be precise about how, because there are three independent failure points and only the first is usually mentioned.

1. **The COSE key cannot be labelled.** The reconstructed COSE_Key needs an `alg` field, and no registered value denotes Schnorr over `secp256k1` (Section 3.5). Labelling it `ES256K` makes the relying party run ECDSA verification on a Schnorr signature, which fails; labelling it `ES256` additionally misstates the curve.
2. **The attestation no longer attests.** The attestation statement is a signature by the authenticator over the authenticator data and the client data hash. Replacing the credential public key inside the authenticator data invalidates that signature. A relying party that requests attestation and checks it will reject the registration outright, and the substitution only survives against relying parties that use `attestation:"none"`.
3. **The private key stays where it was.** WebAuthn generates the credential key pair inside the authenticator and never exports it. The attestation object can be edited on its way to the server, but the authenticator's own copy of the credential cannot; the platform authenticator continues to hold a key unrelated to the group key, so any subsequent assertion request routed through the standard `navigator.credentials.get()` path is answered with the wrong key.

The conclusion is that the group public key cannot be injected into an existing authenticator's credential from the outside. A threshold FIDO2 authenticator has to be an authenticator: the device, or the set of devices, must be what the client talks to over CTAP, producing an assertion in an algorithm the relying party negotiated. This is a firmware and protocol problem, not a data-format one.

4.9 The FIDO2 Compliance Boundary

Combining Section 3.5 with the failed experiment above yields a clearer statement of what stands between this prototype and a deployable threshold authenticator, and of the options for closing the gap. Table 4 summarises them.

The first route is the cheapest, and it follows directly from Section 3.5. Ed25519 is a Schnorr signature, `EdDSA` is a registered WebAuthn algorithm, and RFC 9591's `FROST(Ed25519, SHA-512)` ciphersuite is specified to produce signatures that verify under RFC 8032 [JL17]. Switching ciphersuites changes the group, the encodings, and the hash function, but not a single protocol step of the DKG or of the two-round signing described above; the message format of Section 5 carries larger fields and is otherwise unaffected. One caveat is real, and it is not fatal: a relying party that advertises only `ES256` will never select `EdDSA`, so support is a matter of deployment reach rather than of standards. Verification itself is not a concern. RFC 9591 mandates the *cofactored* group equation $[8][s]G = [8]R + [8][c]\mathsf{pk}$ for this ciphersuite, but an honestly produced FROST signature

Table 4. Three routes to a relying-party-compatible threshold authenticator.

Route	COSE alg	Protocol work	Deployment obstacle
FROST(Ed25519, SHA-512)	`EdDSA` (`-8`)	Ciphersuite swap; the protocol is unchanged	Relying parties must list `-8`
Threshold ECDSA on P-256	`ES256` (`-7`)	New protocol; three rounds, VOLE-based multiplication	Substantially heavier on the boards; no Zephyr-ready library
FROST(P-256, SHA-256)	none	Ciphersuite swap only	Would need a new COSE registration; not viable near term

has R and `pk` in the prime-order subgroup and a reduced s, so it satisfies the cofactorless equation $[s]G = R + [c]\mathsf{pk}$ as well; every RFC 8032 verifier, strict or not, accepts it. The cofactored form matters only for what a verifier *rejects*, never for whether it accepts an honest signature.

The second route is the universally compatible one and the reason threshold ECDSA remains on the roadmap: `ES256` is the algorithm every relying party accepts. Section 8 discusses what implementing it on this hardware would involve.

The third route is included because it looks attractive but is not: staying with FROST and merely moving to P-256 aligns the curve with `ES256` but not the signature algorithm, and there is no identifier for Schnorr over P-256.

Independently of the route, a deployable authenticator must also implement the CTAP2 side: the boards must appear to the client as an authenticator, produce `authenticatorData` with the correct relying party hash, flags, and signature counter, gate signing behind the user verification gesture that the OFFPAD already provides through its fingerprint sensor, and handle the credential management commands. The threshold protocol then sits underneath the assertion-generation step, where the coordinator role would be played by whichever device the client is connected to.

4.10 Evaluation

The thesis underlying this report [Gar25] evaluated the prototype in the 2-out-of-3 configuration over 50 runs. Table 5 reports computation and memory, and Table 6 reports the communication cost per transport.

Four observations follow, and they are more useful than the raw numbers.

The measured HID column is pacing, not transport. The HID path has no acknowledgement above the transport: the host delivers each 61-byte chunk as a `SET_REPORT` control transfer (`HidD_SetOutputReport`), and the firmware copies it into the reassembly buffer inside the USB stack's `set_report`

Table 5. Computation and memory cost, $T = 2$, $N = 3$; times are the range observed over 50 runs.

Operation	Where	Time	Memory
Key generation	Coordinator	102–144 ms	204 KB (3.3%)
Signature share	Board	503–523 ms	—
Share aggregation	Coordinator	2 ms	728 KB (11.9%)

Table 6. Time per message in milliseconds, UART at 115200 baud versus USB HID with 64-byte reports, as measured and with the fixed pauses of the instrumented applications subtracted (columns *meas.* and *corr.*). The pauses are: in `presign_pc_tests`, 100 ms after every HID chunk; in `signing_pc_tests`, 200 ms after every HID chunk and 500 ms after every message on both links; in `signing_offpad_tests`, 1 s before the share is sent and 50 ms after every chunk.

		UART		USB HID	
Phase	**Message (payload)**	meas.	corr.	meas.	corr.
Key generation	Secret share (36 B)	1	1	102	2
Key generation	Public key structure (136 B)	5	5	324	24
Key generation	VSS commitments (232 B)	11	11	433	33
Signing	Round-2 payload (300 B)	521	21	1788	88
Signing	Signature share collection (36 B)	526	526	1223	173

callback. To keep the next report from arriving before the callback has drained the previous one, the instrumented applications pause for a fixed time after every chunk and after every message, and those pauses are the measured column: one, three and four chunks take 102, 324 and 433 ms against 100 ms of pause per chunk in `presign_pc_tests`. Subtracting the pauses listed in the caption gives the corrected columns of Table 6, in which every message crosses either link in tens of milliseconds; a `SET_REPORT` control transfer costs about 8 ms on the Windows host, against a few milliseconds per message over UART. A full-speed interrupt endpoint carries one 64-byte report per 1 ms frame, so even the corrected HID figures are host-side latency rather than wire time. The pauses can be removed outright: the firmware already waits on the `int_in_ready` callback before every `hid_int_ep_write`, which is the back-pressure the stack provides, and the host should use the interrupt OUT endpoint the firmware exposes (`WriteFile` on the HID handle, currently the fallback path) with a one-report acknowledgement per message instead of a timer.

The UART key generation figures fall below the line rate. At 115200 baud with 8N1 framing the wire carries 11 520 bytes per second, so with the 12-byte header the three key generation messages cannot cross in less than 4.2, 12.8 and 21.2 ms respectively. The measured 1, 5 and 11 ms are all below those floors, which means the timer stops when `WriteFile` returns into the

driver's buffer rather than when the bytes are delivered. That column should be read as an upper bound on host-side call latency. The signing rows wait for the board and do not have this problem.

Half of an authentication is the curve arithmetic. A signature share costs 503–523 ms on the board and aggregation two milliseconds on the coordinator. The corrected share-collection rows show what the boards spend beyond the arithmetic. On UART the host's 500 ms post-send pause overlapped the Nucleo's computation, and the share still arrived 526 ms after the pause ended (of which up to 100 ms is the serial read's interval timeout), so the Nucleo delivers its share roughly half a second after the arithmetic is done. On HID the corrected figure of 173 ms, which still includes the host's 100 ms polling granularity, shows the OFFPAD board delivering within about 0.2 s. The difference between the boards lies in flash writes for nonce invalidation, serialisation and the diagnostic logging of the instrumented firmware, which prints hex dumps of every value it handles; it does not lie in the links.

The key generation measurement is for the dealer. The 102–144 ms figure was measured on `presign_pc`, the trusted-dealer application. The distributed key generation of Section 4.4 performs the same algebraic operations per participant and adds N commitment broadcasts, $N(N-1)$ encrypted share transmissions and the associated verification; the figure should therefore be read as the cost of the dealer-based generation used at that stage of the project, and not as the cost of the distributed one.

4.10.1 The End-to-End Cost of One Operation

We do not report a measured end-to-end figure, because no timer in the implementation spans a complete run. The component measurements do support an estimate, and since the estimate is what a reader actually wants we state it explicitly, together with how it is assembled.

For *authentication*, the operation a user performs repeatedly, the critical path is the round-one commitment exchange, the round-two exchange, and aggregation. The two signing rows of Table 6 are consecutive timers that together cover the request, the board's computation and the return of the share, and the concurrency of Section 5 overlaps the T participants rather than serialising them, so they are summed once rather than per participant. As measured this gives $521 + 526 + 2 \approx 1.05$ s over UART and $1788 + 1223 + 2 \approx 3.01$ s over USB HID. Removing the pauses of the instrumented build changes the two figures very differently. On UART the host's 500 ms pause overlapped the Nucleo's computation, so removing it only moves the wait from the send timer to the receive timer and the total stays at about 1.0 s, which is what the Nucleo needs to receive the request, compute the share, invalidate the nonce in flash and return the share. On HID the pauses lie on the critical path: the six 200 ms pauses delay the last chunk and hence the start of the computation, and the 1 s firmware pause delays the share. With them removed the exchange costs $88 + 173 + 2$ ms of transport and polling plus the roughly 0.5 s of board work that the host's

pause had hidden, just under 0.8 s. Round one is not separately instrumented and adds tens of milliseconds on either link. The estimate is therefore about one second per authentication over either link, with the remaining difference due to the boards rather than the transports.

For *enrolment*, which happens once per credential, the coordinator's dealer-based key generation of 102–144 ms is joined by the three key generation transmissions per participant, giving roughly 0.12 s over UART and, with the pauses removed, 0.17 s over USB HID (1.0 s as measured).

Four qualifications belong with these numbers. The message authentication of Section 4.3 was added after these measurements were taken, so every message on the wire is now 64 bytes larger than when it was timed: one or two additional 61-byte reports over USB HID and some six milliseconds of line time over UART, which is small against the figures above but not nothing. The corrected figures are arithmetic on the measured ones, not new measurements: they subtract the fixed pauses listed in the caption of Table 6 and nothing else, and the polling granularity of the host's HID receive loop (100 ms) and the serial read's interval timeout (up to 100 ms) are left in. Re-measuring after the pauses are replaced by proper flow control would replace the arithmetic with a measurement. Both figures are for $T = 2$, $N = 3$ with the dealer-based key generation; the distributed key generation will add to the enrolment figure and leaves the authentication figure unchanged, since signing is identical in both.

A further caveat is the absence of a single-device baseline: we report what a signature share costs but not what an ordinary Schnorr signature costs on the same board with the same library, so the threshold overhead cannot be stated as a multiplier. The structure of the two computations bounds it: an ordinary signature needs one fixed-base scalar multiplication, whereas a FROST participant needs two in round one and T variable-base multiplications in round two to recombine the commitments into R. A single-device signature should therefore cost a fraction of the 503–523 ms share; on the device, the threshold overhead is chiefly those T variable-base multiplications, together with the second round and its storage and transport. That baseline is the measurement we would make first, since it is the number a reader comparing authenticators wants.

Two more caveats apply to all of the above. Memory percentages are the growth of the coordinator process's own working set over its value at start-up (Section 4.11.9) and say nothing about the boards, whose static `prj.conf` tuning is discussed in Section 4.11 and which are the genuinely constrained side of the system; per-board heap and stack high-water marks are not currently instrumented and should be. Every figure reported here is for $T = 2$, $N = 3$: the per-participant computation in the DKG grows linearly in N and the number of share transmissions grows as $N(N - 1)$, so communication, not computation, will be what limits larger configurations.

4.11 Reproducing the Setup

This section is written for a reader who wants to rebuild the system on their own hardware rather than only to understand it. The coordinator programs, the board firmware, and build instructions are publicly available at

https://github.com/JavierGarcia22/FROST_Threshold

and every concrete value below was read out of that repository. What follows is the information needed to plan the work, to know when it has succeeded, and to know which of our results a given setup can and cannot check.

4.11.1 Contents of the Published Repository The repository holds seventeen application directories and a `west` manifest. Ten are board firmware, built with Zephyr, in matched pairs for the two targets: `receive_usb_nucleo` and `receive_usb_offpad` take delivery of key material from the trusted dealer, `presign_nucleo_dkg_ElGamal` and `presign_offpad_dkg_ElGamal` and their `_blame` variants take part in the distributed key generation, `signing_nucleo` and `signing_offpad` perform the signing rounds, and `signing_nucleo_tests` and `signing_offpad_tests` are the instrumented variants that produced the timings in Section 4.10. Seven are host programs built natively: `presign_pc`, `presign_pc_dkg_ElGamal`, `presign_pc_dkg_ElGamal_blame`, `signing_pc` and `signing_pc_broadcast`, with `presign_pc_tests` and `signing_pc_tests` as the instrumented counterparts of the first and fourth. The vendored cryptographic library appears under each application's `modules` directory and under `external`.

Two points about scope matter to anyone planning to reproduce this work, and we state them here rather than leaving them to be discovered.

The applications come in variants rather than one configurable build, which is the first thing to understand before choosing what to compile. Key generation exists in three forms: `presign_pc` performs the trusted-dealer generation used in the earliest measurements; `presign_pc_dkg_ElGamal`, with its board counterparts `presign_nucleo_dkg_ElGamal` and `presign_offpad_dkg_ElGamal`, performs the distributed key generation of Section 4.4 with the encrypted share exchange of Section 4.4.2; and the `_blame` variants of those three add the blaming protocol of Section 4.4.3. Signing likewise exists as `signing_pc` and as `signing_pc_broadcast`, the latter implementing the concurrent coordination of Section 5. A reader reproducing the protocol as this report specifies it wants the `_dkg_ElGamal_blame` variants and `signing_pc_broadcast`; a reader reproducing the timings in Section 4.10 wants the `_tests` variants, which instrument the dealer-based path. This is why the reported key generation figure is a dealer figure: it was measured on the application that exists for that purpose.

The repository README now covers both paths. As of commit `8682a93` it describes the trusted-dealer applications and the distributed key generation variants side by side, including the ElGamal-encrypted share exchange and the blame variants, and its build instructions cover both.

The web demonstrator of Section 4.7 is not in this repository. Nothing in the reproducibility path below depends on it, and no measurement in Section 4.10 was taken through it.

4.11.2 Platform Requirements *The coordinator currently runs on Windows only.* It is written in C, compiled with GCC, and links against the Windows `setupapi` and `hid` libraries for device enumeration and USB HID transfers, and against Win32 serial calls for UART. This is the first thing to check before planning a reproduction, because on Linux or macOS the coordinator will not build as it stands.

The dependency is confined to the transport layer and is not deep. A port would replace device enumeration and HID transfers with `hidapi`, which offers the same operations across platforms and is what we would use if starting again, and the Win32 serial calls with `termios`. The cryptographic code, the message format, and the protocol logic are platform-independent. We estimate this as a contained piece of work rather than a rewrite, but we have not done it, so a reader on a Unix host should budget for it rather than assume it.

4.11.3 Reproduction with Partial Hardware Not every result requires our exact hardware, and it is worth being explicit about which is which, since the OFFPAD development board is not a product an outside reader can obtain.

Table 7. Reproducibility by available hardware.

You have	You can reproduce
A coordinator only	The full protocol with all N participants local to the coordinator: distributed key generation, signing, blaming, and signature verification. This exercises every cryptographic step and none of the embedded constraints.
One Nucleo-L476RG	All of the above, plus on-board key generation and signing, the flash persistence behaviour, the memory tuning, and the complete UART column of Table 6. This is enough to check our central finding that the curve arithmetic accounts for about half of an authentication while each message crosses the link in tens of milliseconds.
Two or more Nucleos	All of the above with genuine multi-device coordination and the concurrency behaviour of Section 5.
An OFFPAD board	Additionally the USB HID column, and therefore the UART-versus-HID comparison as we report it.

Any board that Zephyr supports and that has enough RAM and a spare flash partition should work in place of the Nucleo; the Nucleo is the more constrained of our two targets, so it is a reasonable lower bound.

4.11.4 Software Versions Embedded builds are sensitive to versions, and this one more than most: the Zephyr pin is load-bearing rather than incidental, because later releases break compatibility with the cryptographic library. Table 8 records the environment the reported results were produced in.

Table 8. Build environment for the reported results.

Component	Version
Zephyr RTOS	`v3.7.0`, pinned in `west-manifest/west.yml`
Toolchain	`arm-none-eabi-gcc` via the Zephyr SDK
Cryptographic library	`secp256k1-frost` [Ban22], upstream commit `8d5328c`, vendored per application
Coordinator compiler	GCC on Windows
Coordinator optimisation	none specified: built `gcc -g`, hence `-O0`
Board optimisation	Zephyr default for the build type

Two entries deserve comment. The coordinator is built with `gcc -g` and no optimisation flag, so it compiles at `-O0`; the aggregation and key generation timings in Table 5 are therefore unoptimised host figures and would improve, probably substantially, at `-O2`. Optimising the host would lower the aggregation and dealer figures and leave the board and transport figures untouched, so it does not affect the conclusions of Section 4.10, but a reader comparing numbers needs to know that we did not compile for speed.

The cryptographic library is not fetched by `west`. Each application directory carries its own copy of the sources under `modules/secp256k1/`, and that copy is what the application's `CMakeLists.txt` compiles, so the code we built against is fixed by the repository itself. The upstream revision the repository records for that library is commit `8d5328c`; a reader who needs byte-level provenance should compare the vendored copies against it.

The underlying library is an extension of the Bitcoin Core `secp256k1` library with FROST support, adapted in our repository to build under Zephyr for the board targets. It supplies constant-time scalar multiplication, which is why we do not implement curve arithmetic ourselves.

4.11.5 Repository Layout and Build Section 4.11.1 lists the application directories. For the trusted-dealer path, the order to build them in is: host key generation, board key reception, then the two signing halves; for the distributed path, the host and board `_dkg_ElGamal_blame` applications take the place of the first two.

Building the firmware follows the standard Zephyr workflow, with `west-manifest/west.yml` supplying the pinned Zephyr version (the cryptographic library is vendored in each application directory, as described above):

```
west update
cd signing_nucleo && west build && west flash
```

`west update` pulls Zephyr into `external/`; each application directory is then a build root. The board target is supplied to `west build` with `-b`, or configured once per build root: for the Nucleo this is Zephyr's `nucleo_l476rg` target, and for the OFFPAD it is the board definition supplied with that hardware. The host programs are built directly:

```
cd presign_pc/src
gcc -g main.c -lsecp256k1 -lsetupapi -lhid -o keygen.exe
```

and likewise in `signing_pc/src`. Note the absence of an optimisation flag, discussed above.

One edit is required before the Nucleo firmware will work on a given board. The UART device is selected by a compile-time node label,

```
#define UART_DEVICE_NODE
    DT_NODELABEL(usart1)
```

and this must be changed to match the port actually wired on the board in use. This is the first thing to check if a Nucleo appears not to respond.

Running the system end to end means starting the coordinator with the boards attached, performing a key generation once, and then signing as often as wanted; Table 3 in Section 4.6 gives the order of operations, and Section 4.11.8 gives the checks that tell you it worked.

4.11.6 Board Configuration Three categories of Zephyr configuration are load-bearing and are the most likely source of difficulty when porting.

Entropy. Every `prj.conf` sets three Kconfig options,

```
CONFIG_ENTROPY_GENERATOR=y
CONFIG_ENTROPY_STM32_RNG=y
CONFIG_ENTROPY_DEVICE_RANDOM_GENERATOR=y
```

and the firmware samples through `sys_csrand_get`. A port to another board must find the equivalent entropy driver and keep these three lines; a project that starts from a Zephyr sample inherits `CONFIG_TEST_RANDOM_GENERATOR`, which is a timer-based generator for testing, and nothing in the protocol's behaviour will reveal the difference (Section 6.4).

Heap and stack. The FROST operations allocate the commitment vectors and the intermediate point arithmetic on the heap, and peak allocation grows with T and N. The heap pool and the stacks of the threads running the cryptographic work must be raised from their defaults; the failure mode when they are not is

an allocation failure or a stack overflow inside the library, surfacing as a fault far from its cause. Table 9 lists the settings that matter and the values we used.

Table 9. Memory settings in the shipped `prj.conf` files, in bytes. "—" means the setting is left at the Zephyr default. Values differ between the key-reception and signing applications because their peak allocations differ.

	Key reception		**Signing**	
Setting	Nucleo	OFFPAD	Nucleo	OFFPAD
`HEAP_MEM_POOL_SIZE`	4 096	65 536	32 768	32 768
`MAIN_STACK_SIZE`	70 000	65 536	16 384	32 768
`SYSTEM_WORKQUEUE_STACK_SIZE`	—	32 768	4 096	16 384
`COMMON_LIBC_MALLOC_ARENA_SIZE`	—	32 768	—	32 768
`ISR_STACK_SIZE`	—	—	—	4 096

Three observations for anyone porting. The settings are not uniform across the four applications, so a reader must take the values for the application they are building rather than a single global figure. The `MAIN_STACK_SIZE` of 70 000 bytes in the Nucleo key-reception application is more than half the 128 KiB of SRAM on an L476RG, which is a warning rather than a recommendation: it is what made that application run, and it leaves little headroom. These values are for $T = 2$, $N = 3$; peak allocation grows with both, so any other configuration needs re-sizing. Doing that by measurement, with Zephyr's thread analyser reading the high-water mark, is strongly preferable to trial and error.

Flash storage. The implementation does not define its own partitions. It opens the board's existing `storage_partition` through Zephyr's flash-map API, using `FIXED_PARTITION_ID(storage_partition)`, and writes a single packed record containing the key material of Section 4.4.5 together with the nonce state and a `used` flag. There are no device tree overlay files in the repository, so the partition's size and offset are whatever the board's own device tree provides.

This is worth knowing for two reasons. It makes porting easier, since any Zephyr board that defines a `storage_partition` will work without an overlay. But it also means the key material and the nonce state share one erase unit, which is precisely the arrangement Section 6.4 argues against: the nonce state needs erase-before-use discipline of its own, and separating the two into distinct partitions is a change we would make before any deployment.

4.11.7 Physical Setup Two points cost us more time than any software problem and belong here rather than in a lessons-learned section.

Flashing the OFFPAD requires a Nucleo board acting as a programming bridge. Supporting this while also allowing simultaneous debugging and data exchange requires careful GPIO mapping, guided by the board datasheet, so that the USART1 and USART2 pins are assigned to their respective roles instead of debugging and communication competing for one interface.

Loose pins repeatedly disrupted both UART and USB communication during testing, producing intermittent failures that are easy to misattribute to the communication code. Before debugging any transport problem, reseat the connections.

4.11.8 Validating a Build A reproduction is only useful if the reader can tell whether it worked, and a threshold signature scheme fails quietly: an implementation with the wrong hash instantiation or the wrong byte order will interoperate perfectly with itself and with nothing else. We therefore recommend the following sequence, in order, before trusting any measurement. It runs from cheapest to most expensive and each step localises a different class of fault.

1. **Check against the published test vectors.** RFC 9591 publishes vectors for `FROST(secp256k1, SHA-256)` covering the hash functions, the binding factors, the signature shares, and the aggregate signature. This is the only step that verifies conformance rather than self-consistency, and it is the one we have not yet performed ourselves; a reader reproducing this work should do it first.
2. **Verify a single signature against an independent implementation.** Run the protocol with all participants local to the coordinator, sign a fixed message, and verify the resulting σ with a library that shares no code with ours, for example the audited Rust reference implementation [Zca24]. A signature that our verifier accepts and an independent one rejects indicates an encoding or domain-separation error.
3. **Exercise the failure paths.** Corrupt one byte of one encrypted share and confirm that the recipient's verification check rejects it and that the blaming protocol names the correct party. An implementation that never rejects anything is not validating anything.
4. **Power-cycle between phases.** Complete signing round one, remove power from a board, restore it, and complete round two. This is the behaviour the flash storage exists for, and it is easy to have working state that does not survive a reset.
5. **Confirm nonce invalidation.** Ask a board to sign twice from one round-one commitment and confirm that it refuses the second request; then power the board off between computing a share and receiving the coordinator's acknowledgement, and confirm on restart that the pair is already marked consumed. Together these check the single property whose failure exposes the signing share.

4.11.9 Reproducing the Measurements Timing numbers are only comparable if they bracket the same work. The instrumentation lives in the four `_tests` applications, not in the plain ones, so a reader wanting to re-measure must build those.

Clocks. The boards time the signature share computation with Zephyr's `k_uptime_get()`, whose millisecond resolution is ample at roughly 500 ms. The transport rows of Table 6 and the 2 ms aggregation figure are timed on the coordinator with `QueryPerformanceCounter`, which has sub-microsecond resolution; their limitation is what the timer brackets rather than its resolution, as Section 4.10 shows for the UART key generation rows, whose timer stops when `WriteFile` returns.

Memory. Host memory is the process working set from `GetProcessMemoryInfo`. The percentages in Table 5 are the *increase over the process's own initial working set*, not a share of any machine-wide budget: key generation grew the working set from 6200 to 6404 KB, an overhead of 204 KB or 3.3%, and signing from 6108 to 6836 KB, an overhead of 728 KB or 11.9%. Board memory is *not* measured: the instrumented firmware contains a memory helper that returns a real figure only when `CONFIG_SYS_HEAP_RUNTIME_STATS` is enabled, and no shipped `prj.conf` enables it, so the fallback branch returns a synthetic value derived from elapsed time. No board memory number appears in this report, and none should be taken from that code path without first enabling that Kconfig option, which is the cheapest way to obtain an on-board memory figure.

Two conventions apply throughout. Board-side computation figures exclude transmission; the transport figures in Table 6 are measured at the coordinator and, for the signing rows, include the time the board spends computing its share, which is why a 36-byte payload appears to take far longer during signing than during key generation. Nothing in this evaluation varies the threshold; every figure is for $T = 2$, $N = 3$.

4.11.10 Troubleshooting Table 10 collects the failures that actually cost us time, with what they turned out to be. Almost every one presented as a problem in a different layer from the one at fault, which is why we record them.

The official Zephyr documentation, particularly its coverage of UART and HID callbacks, flash handling, and interrupt handlers, was the most useful reference during development; the Zephyr community Discord answered several questions the documentation does not cover.

4.11.11 Porting Checklist For a different board we would proceed in this order, based on where the time actually went. Build the library for the target and confirm a single non-threshold Schnorr signature verifies before attempting any threshold operation. Raise heap and stack and confirm a complete key generation with all participants local to one board. Exercise the flash storage, including a deliberate power cycle between phases. Only then bring up the transport, the

Table 10. Symptoms encountered during development and their actual causes.

Symptom	Cause and remedy
Fault or hard reset partway through key generation or signing	Heap exhaustion or stack overflow inside the cryptographic library. Raise the settings in Table 9 and confirm with the thread analyser. The fault address is usually unrelated to the actual cause.
Build fails after updating Zephyr	The version pin is load-bearing. Later releases break compatibility with the cryptographic library; stay on 3.7.0 until the incompatibility is resolved.
Intermittent transport failures that appear to be protocol bugs	Physical connections. Reseat cables and check pin seating before reading any communication code.
Debugging and data exchange interfere on the OFFPAD	USART1 and USART2 roles are competing. Check the GPIO mapping against the board datasheet.
Key material lost across a power cycle	The flash storage is not being written. Confirm that the board defines a `storage_partition` and that writes are flushed before power is removed.
Signature rejected by an external verifier but accepted internally	Encoding or domain separation. The hash functions must be the domain-separated H_1 to H_5 of the ciphersuite, and points must be serialised canonically before hashing (Section 4.4.2). Check against the RFC test vectors.

likeliest source of misleading symptoms: list message fields explicitly rather than serialising library structures, then dump every message type once and confirm no secret appears in any of them.

5 Communication

Threshold signature schemes require devices to exchange information, such as secret shares, commitments, or the message to be signed. Since the OFFPAD's security requires this information to be exchanged correctly and securely, the communication layer is one of the most important parts of this project. This section covers the decisions made for handling communication, including the communication type, the message format that supports it, the communication model used to coordinate the participating devices, and the concurrency strategy adopted to make this coordination efficient.

5.1 Communication Type

Two options were considered for exchanging data between the coordinator and the participant devices: USB and Bluetooth. USB was chosen for this project, since it offers a reliable, fast, and physically secure connection. Its main drawback is practical rather than security-related, because cable and port management can be troublesome as the number of participating devices grows. Bluetooth was considered as an alternative that would solve this scalability issue, but given the added complexity of implementing it and the time constraints of the project, it was left as a possible line of future work rather than adopted here.

Furthermore, in this project, two different development boards were used: an STM32 Nucleo-L476RG board and an OFFPAD development board.

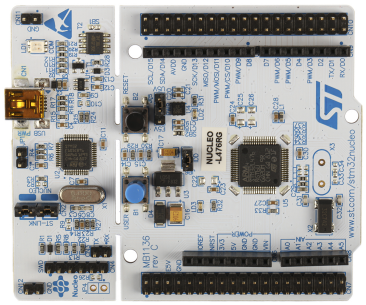


Fig. 5. Nucleo-L476RG board.

The STM32 Nucleo-L476RG board only supports UART communication over its virtual COM port, whereas the OFFPAD development board only supports

USB HID communication. As a result, both protocols had to be implemented and supported, letting the user select, for each board, which of the two methods to use.

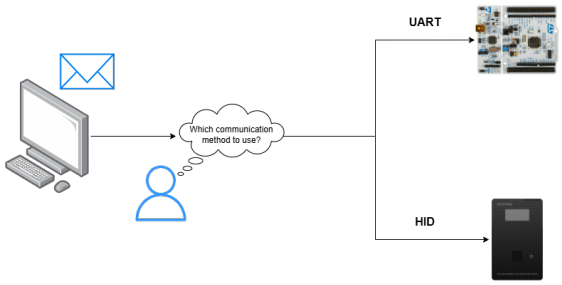


Fig. 6. Communication method selection.

5.1.1 Message Format To keep this implementation manageable despite working with two different communication methods, all messages exchanged during both key generation and signing follow the same structure: a fixed-size header followed by a payload. The header describes what the payload contains, so that the receiver, regardless of whether it arrived over UART or USB HID, can parse it the same way. Table 11 summarises the header fields in the order in which they appear on the wire.

Table 11. Fields of the message header.

Field	Purpose
`MSG_HEADER_MAGIC`	A fixed constant that both sides (sender and receiver) must agree on in advance. Messages that do not start with this value are discarded.
`version`	Format version, currently `0x01`, so that a receiver can reject a message produced by an incompatible build.
`msg_type`	Identifies what kind of payload follows, for example a secret share, a public key, VSS commitments, a signing request or an end-of-transmission marker. This lets the receiver handle the received message correctly depending on the information it contains.
`payload_len`	The length of the payload in bytes. Needed because USB HID transmits data in fixed-size chunks, so the receiver must know when the full message has been reassembled.
`participant`	The ID of the participant the payload belongs to, so that key material for different devices is easily identified.

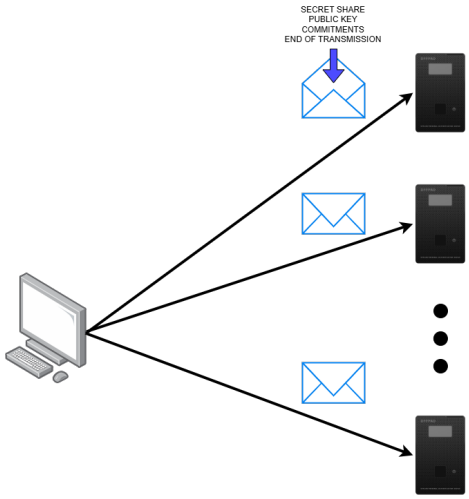


Fig. 7. Message transmission to the boards.

5.1.2 Payload Sizes The payloads are built field by field from the values the protocol defines, using the encodings of the underlying `secp256k1-frost` library, which is what makes them easy to reproduce: an implementer who links the same library gets the same byte counts. Table 12 gives their composition for the $T = 2$, $N = 3$ configuration used in the evaluation, together with the general formula, since two of them grow with the threshold. The key generation rows describe the trusted-dealer applications (`presign_pc` and `receive_usb_*`), whose transmissions Table 6 measures; the distributed key generation applications use their own message types and compressed 33-byte points for the encryption layer. All sizes exclude the 12-byte message header, which is added on top by the transport layer: a 4-byte magic value (`0x46524F53`, the ASCII "FROS"), a 1-byte version, a 1-byte message type, a 2-byte payload length and a 4-byte participant identifier. Every message additionally carries the sender's device signature: an ordinary 64-byte Schnorr signature over `secp256k1`, a 32-byte nonce point followed by a 32-byte scalar, computed over the header and the payload and verified against the provisioned key list before the payload is parsed (Section 4.3). A complete message on the wire is therefore $12 + \ell + 64$ bytes for a payload of ℓ bytes. The message types in use are `SECRET_SHARE` (`0x01`), `PUBLIC_KEY` (`0x02`), `COMMITMENTS` (`0x03`), `NONCE_COMMITMENT` (`0x04`), `READY` (`0x06`), `SIGN` (`0x07`), `SIGNATURE_SHARE` (`0x08`) and `END_TRANSMISSION` (`0xFF`).

Two points are worth drawing out. The VSS commitment payload is the only one that grows with the threshold during key generation, at 64 bytes per additional polynomial degree, and the round-2 request grows at 132 bytes per additional signer; over USB HID, where a payload is split into 61-byte usable chunks, each of those increments costs one to three additional reports. Once the fixed pauses of

Table 12. Payload composition and size, excluding the 12-byte header and the 64-byte device signature that every message carries. Group elements are stored in the library's 64-byte uncompressed internal form; scalars are 32 bytes; indices are 4-byte integers.

Payload	**Composition**	**General**	$T = 2$
Secret share	index + scalar	36	36 B
Public key structure	index + participant count + verification share + group public key	136	136 B
VSS commitments	index + coefficient count + T commitments + proof of knowledge (K_i, μ_i)	$104 + 64T$	232 B
Round-1 response	nonce commitment (index + hiding + binding, 132 B) + public key structure (136 B)	268	268 B
Round-2 request	32-byte message hash + 4-byte count + T nonce commitments (index + hiding + binding)	$36 + 132T$	300 B
Signature share	index + scalar	36	36 B

the instrumented builds are discounted (Table 6) a report costs a few milliseconds rather than the hundreds the measured column suggests, so raising T is not expensive on either transport; it is simply more expensive in transmission than in computation. Note also that the uncompressed 64-byte point encoding used on the wire is the library's internal representation, not the 33-byte compressed SEC1 encoding used as hash input in Section 4.4.2; conflating the two is a portability bug, and an implementation that switched the wire encoding to compressed points would cut the commitment payload roughly in half at the cost of a decompression on receipt.

5.1.3 UART Communication For the STM32 Nucleo board, communication takes place over a UART connection configured with a baud rate of 115200 bits/s, 8 data bits per character, one stop bit, and no parity bit. Since UART naturally supports variable-length streams, no chunking is required, and the header and payload are simply written to the serial port.

On the receiving side, data arrives asynchronously, so reception is interrupt-driven: incoming bytes are placed into a ring buffer by the UART interrupt handler as they arrive, while a separate reception loop consumes that buffer independently. This loop alternates between two states: `WAITING_FOR_HEADER`, where it waits until enough bytes have arrived to validate the magic number so it can then read `msg_type` and `payload_len`, and `WAITING_FOR_PAYLOAD`, where it waits until `payload_len` additional bytes have arrived before processing the complete message and returning to the first state.

5.1.4 USB HID Communication The OFFPAD board instead communicates as a custom USB HID device, identified by Vendor ID `0x2FE3` and Product ID `0x0100`. This removes the need for the user to specify a port, but introduces a different constraint: HID reports have a fixed size (64 bytes, of which the firmware reserves three for a report ID, a chunk length and a framing byte, leaving 61 bytes of usable payload per report), unlike UART's continuous byte stream.

Because a message formed by a header and a payload will often exceed 61 bytes, every message is split into 61-byte chunks before transmission and reassembled on the receiving side. The reassembly process tracks how many bytes have been received so far for the current message (`reassembly_pos`) against the total size it expects to receive (`expected_total_size`, obtained from the header's `payload_len`). When enough bytes have arrived, the message is considered complete and handed off for processing, analogously to the two-state process used for UART, adapted to the chunked transmission required by USB HID.

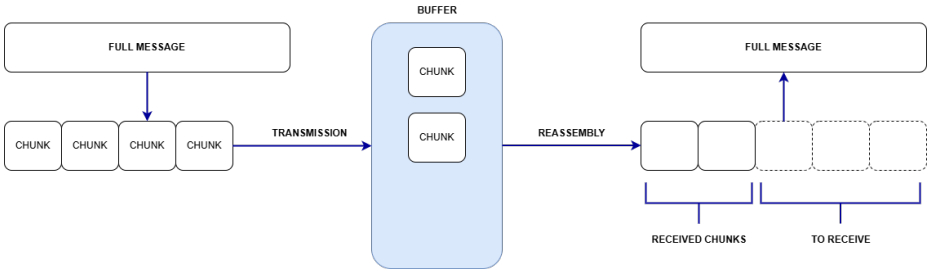


Fig. 8. Message transmission and reception with reassembly buffer.

The device itself is configured as a generic HID interface (using standard HID usage tags), and a status callback tracks the USB connection state, resetting internal buffers on `USB_DC_RESET` and marking the device ready when `USB_DC_CONFIGURED` is reached. Once a full message has been reassembled, it is processed and stored using Zephyr's work-queue system.

Algorithm 10 USB HID callbacks.

```
1: procedure IntOutReadyCb(dev)
2:     Read up to 64 bytes from USB endpoint into buffer
3:     if read successful and bytes_received > 0 then
4:         HandleChunkedData(buffer, bytes_received)
5:     end if
6: end procedure

7: procedure StatusCb(status)
8:     if status == USB_DC_RESET then
9:         configured ← False
10:        ResetAllData
11:    else if status == USB_DC_CONFIGURED then
12:        configured ← True
13:    end if
14: end procedure
```

5.2 Communication Model

Beyond the type of communication for each board, a separate design decision is the communication model. We considered two options.

Direct device communication: Every participating device communicates directly with every other device. This is the more decentralised option, since no single device sees all the communication, but it requires the implementation of a more complex coordination logic and handling a higher communication and computational load, which the OFFPAD's hardware is not prepared for.

Coordinator model: A single device, known as the coordinator, handles all communication: participants send their data only to the coordinator, which then redistributes it to the corresponding participants. This removes the need for participant devices to coordinate with each other directly, at the cost of introducing a single point of failure and potentially requiring the coordinator to be a trusted device.

Given the OFFPAD's hardware limitations, this project adopts the coordinator model, with a computer taking on the role of coordinator. Moreover, so that the coordinator need not be trusted, the exchanged shares are protected with the hashed-ElGamal encryption of Section 4.4.2 and every message is signed under the sender's device key (Section 4.3): although all information passes through a single device, only the intended receiver can decrypt each message, and no message can be altered or fabricated in transit without detection.

It is important to be precise about what this does and does not buy, because it is tempting to read the encryption as making the coordinator harmless. The transport itself provides no authenticity: the header's magic value is a framing marker, not a MAC, and a coordinator or anyone in a position to inject on

the wire can modify, drop, replay, or reorder any message. Authenticity comes from the device signature that every message carries (Section 4.3), which is what gives participants a consistent view of the transcript; the cryptographic checks above the transport then tie each value to what its sender committed to, so that a modified share fails the VSS check, a modified commitment fails the proof-of-knowledge check, and a modified signature share fails the aggregator's verification. What no layer can prevent is the coordinator simply refusing to relay, and Section 6.2 discusses what that leaves.

5.3 Broadcast Communication

The coordination model describes who talks to whom, but not in what order. The project's first implementation handled devices one at a time: the coordinator would complete its exchange with one board before starting with the next. This was simple to implement but suboptimal, as the total time required grew with the number of connected devices, even though only T of the N devices are required to complete a signature.

To address this, the implementation was changed to broadcast communication, where the coordinator opens communication with all connected devices simultaneously and lets them respond independently. This is implemented by using one thread for each connected device, each of which handles the information exchange with a single device without blocking the others.

Furthermore, the coordinator does not need to wait for all N devices to respond, only for the first T devices to return a valid answer, since T is the threshold required to reconstruct a signature. As soon as those T valid responses have arrived, the coordinator proceeds to the next round, without waiting for the remaining devices, which could have been slower to respond or unavailable.

Algorithm 11 Threads creation.

```
1: active_threads ← 0
2: Acquire critical section lock device_lock

3: for i = 0 to num_discovered_devices − 1 do
4:     device ← discovered_devices[i]
5:     if device is active and device.type ≠ LOCAL_COMPUTER then
6:         Store message type and payload for the specific device
7:         Create ready_event and complete_event for synchronisation
8:         Create new thread to execute DeviceSendThread (suspended)
9:         active_threads ← active_threads + 1
10:    end if
11: end for
```

This approach means the total time for a round is determined by the T fastest devices, rather than by the slowest device out of all N.

5.4 Summary of Design Decisions

Table 13 summarises the design decisions covered in this section.

Table 13. Summary of communication architecture and design decisions

Design aspect	Selected approach	Details
Connection	USB	Chosen for reliability, speed, and physical security (Bluetooth relegated to future work).
Transport protocols	UART & USB HID	Necessary to support both the STM32 Nucleo (UART) and OFFPAD (HID) hardware constraints.
Message format	Header + payload	Lets the receiver validate, identify, and reassemble any message the same way regardless of whether it is using UART or USB HID for the communication.
UART settings	115200 baud, 8N1 (8 data bits, no parity, 1 stop bit)	Standardised serial port configuration for reliable UART communication.
USB HID handling	Chunking & reassembly	Communication established via Vendor/Product ID. Data is split into 64-byte reports and reconstructed in a buffer on the device.
Communication model	Coordinator model	A central computer routes messages, protecting OFFPAD devices from heavy computational and communication load. Hashed-ElGamal encryption of the shares ensures the coordinator cannot learn the secret key from the data it relays, and the device signatures of Section 4.3 mean it cannot alter the transcript either.
Concurrency	Threaded broadcast	The computer communicates with all devices simultaneously and proceeds as soon as T (threshold) valid responses arrive.
Data storage	Zephyr work queues	Prevents USB/UART interrupt handlers from blocking while safely writing assembled key material to flash memory.

6 Security Considerations

The implementation described in the previous sections makes several pragmatic design choices, and it is important to be explicit about what each of them does and does not protect against. This section first states the threat model the design assumes, and then discusses the points at which the implementation falls short of it, which is where a deployment building on this work has to do additional work.

6.1 Threat Model

6.1.1 What We Assume

Participants: Up to $T - 1$ of the N participants may be fully corrupted: the adversary sees their long-term shares, their nonces, and their internal state, and controls their messages. Corruption is static, that is, the adversary chooses whom to corrupt before the key generation begins; the adaptive case is not covered by the analysis of FROST that we rely on.

Coordinator: The coordinator is untrusted: it sees every message and may drop, delay, reorder, or modify anything it relays, and may hold a share of its own if configured as a local participant, in which case it counts against the corruption budget. We make no availability claim against it, since a relay that stops relaying halts any protocol trivially; the claims below concern what it can learn and what it can cause to be signed.

Device keys: Each device holds a long-term signing key pair, and the list of the N device public keys is provisioned to every device out of band before the protocol starts (Section 4.3). The adversary cannot alter that list on an honest device. This assumption carries more weight than any other in this list, and Section 6.2 says what follows from it.

Channels: The USB links between the coordinator and the boards provide neither confidentiality nor authenticity in their own right. We treat them as a network under the coordinator's control rather than as trusted wires, and all security comes from the cryptographic layer above.

Devices: The boards are assumed not to be physically compromised beyond the corruption budget: no invasive extraction or rollback of flash contents, and no fault injection. Scalar multiplications are constant-time, since the underlying library provides this, but we make no claim about resistance to power or electromagnetic analysis of the surrounding code.

Randomness: Each honest device is assumed to have a working entropy source. Section 4.5 explains why the RFC's hedged nonce derivation is still worth adopting on this hardware: it degrades gracefully if the randomness is biased or short, though not if it repeats exactly.

6.1.2 What We Claim Under these assumptions, an adversary that corrupts fewer than T participants, and the coordinator besides, cannot produce a signature without the cooperation of an honest participant in that very signing session, learns nothing about the group secret key beyond what its own shares and the public key already determine, and cannot cause an honest participant to accept an inconsistent view of the key generation. The most such an adversary can achieve is to abort the protocol, with the blame assigned to itself. The claim is about *who must participate*, not about *which messages are signed*: what an honest participant agrees to sign is, in the prototype, whatever the coordinator presents (see below), so until signing is gated on a user verification gesture the guarantee protects the key and not the individual authentication. Signing succeeds whenever T honest participants are reachable and the coordinator relays faithfully; it is not guaranteed to succeed otherwise (Section 4.1.1).

6.1.3 What We Explicitly Do Not Claim Robustness in the presence of an actively malicious participant (Section 4.1.1); any protection once T or more devices are compromised, at which point the adversary simply holds the key; resistance to physical attacks on the boards; and any property at all of the browser integration described in Section 4.7, which is a demonstrator. Nor do we claim that a user gesture authorises each signature: the prototype signs whatever the coordinator presents, and binding signing to the OFFPAD's fingerprint verification is required work, not an existing feature of the prototype.

6.2 The Coordinator and the Provisioned Key List

Distributed key generation protocols of the kind described in Section 4.4 are analysed under the assumption of an authenticated broadcast channel: when a participant broadcasts its commitments and proof of knowledge, every other participant is guaranteed to receive the same values. In our coordinator model that broadcast is emulated by a single relaying computer, which provides no such guarantee on its own.

Encryption alone would not supply it either. Against a *passive* coordinator the hashed-ElGamal layer ensures that only the intended receiver can decrypt each share, but encryption gives confidentiality, not consistency, and an *active* coordinator would have had two ways to exploit the difference. It could *equivocate*, presenting different commitment sets to different participants; that forges nothing and extracts no key, but it drives honest participants into inconsistent views of the group key, which shows up as an aborted key generation or, worse, as participants holding shares of a key that no coalition can use. Because the encryption keys ek_j also pass through the coordinator like everything else, it could *substitute* its own ek' for ek_j, decrypt every share sent to P_j and re-encrypt it under the real ek_j before forwarding, with every VSS check passing and no party noticing.

The message authentication of Section 4.3 closes both, and closes them with one mechanism, which is why it belongs to the protocol rather than to a list of

countermeasures. A commitment set or an encryption key that the coordinator has altered or substituted carries no valid signature under any key on the provisioned list, so the receiving device discards it and the session aborts with the coordinator identified as the cause. The same signatures are what make the blame procedure of Section 4.4.3 sound rather than merely public.

What the coordinator retains is denial of service, and no protocol change closes that: a relay that stops relaying halts any protocol. A coordinator that drops or corrupts messages draws the honest participants into the blaming protocol, which terminates in an identifiable abort with the responsible party named rather than in a signature. Robustness against such a coordinator would require the retry structure of ROAST [RRJ+22] over a different coalition, which in a 2-out-of-3 deployment means a different coordinator; Section 4.1.1 explains why plain FROST does not provide it.

The whole arrangement rests on the list of device public keys, and that list is provisioned out of band. Whoever controls the provisioning step controls group membership, and a device whose list has been tampered with will accept messages signed by the tamperer's keys; no amount of cryptography inside the protocol repairs that. The prototype provisions the list by hand before the first run, which is adequate for a laboratory bench and not for a product. A product needs an enrolment procedure that establishes the list under the user's control, for example by pairing the devices directly through a short exchange that the user confirms on each device, and that fixes the list on the device afterwards. This is the counterpart of credential enrolment and attestation in FIDO2, and it is where the security of a threshold authenticator is actually decided; it is the first row of Table 14 we would close.

6.3 Information Revealed by the Blaming Protocol

The blaming protocol resolves a complaint by having the accused participant P_i reveal the ephemeral scalar r_{ij}, which allows anyone to decrypt and publicly verify the disputed share $s_{ij} = f_i(j)$. This deliberately makes one evaluation of the polynomial f_i public, and it is worth explaining why this is acceptable.

First, the revealed value was already known to both parties in the dispute; the proof only publishes what the accuser claimed to have received. Second, a single evaluation of a degree-$(T-1)$ polynomial leaves P_i's secret contribution a_{i0} undetermined as long as fewer than T evaluations become public in total, counting those held by corrupted participants: with a corruption budget of $T-1$ the adversary already holds $T-1$ evaluations of f_i, so one published blame proof is exactly enough to reach T and determine a_{i0}. Third, and this is what makes the point moot, a complaint means the key generation has failed: either P_i misbehaved or P_j lied, and in both cases the offending party is excluded and the DKG is rerun with freshly sampled polynomials. The revealed evaluation therefore belongs to a discarded execution and never contributes to a key that is actually used.

Implementations must preserve this discipline, and it is a genuine requirement rather than tidiness: shares from an execution in which any blame proof was published must never be combined into a signing key, and the ephemeral scalars r_{ij} must be erased once a key generation completes without complaint, since retaining them turns any later compromise of a device into a compromise of the shares it sent.

6.4 Nonce Handling in Flash Memory

Schnorr-style signatures are unforgiving of nonce reuse: two signature shares produced with the same nonce pair (d_i, e_i) on different messages allow anyone to solve for the secret share sk_i, and from T such shares the group key follows. Because the boards must survive power cycles between the commitment round and signing, nonces are persisted in the board's flash storage partition, which turns nonce freshness from a property of volatile memory into one the firmware must actively enforce.

The ordering of operations around the stored nonce is a security property. The stored pair must be invalidated *before* the signature share leaves the device, so that a crash or power loss between signing and invalidation cannot result in the same nonce being loaded and reused after reboot; the firmware therefore marks the pair consumed in flash, flushes, and only then computes and transmits s_i. The cost is one flash erase-and-write of a few milliseconds in the critical path, small against the half-second share computation, and the benefit is that the single most damaging failure mode of Schnorr signing becomes unreachable. The validation checklist in Section 4.11.8 includes a test for exactly this property.

Randomness deserves the same explicitness, because on Zephyr the choice of generator is a build-time configuration with no run-time symptom. Every board application enables the STM32 hardware entropy source through Kconfig (Section 4.11 lists the options) and samples through `sys_csrand_get`, and the coordinator draws its decryption key, its ephemeral scalars and its own nonce material from the platform CSPRNG (`BCryptGenRandom`). We are explicit about this because Zephyr's default for a new project is `CONFIG_TEST_RANDOM_GENERATOR`, a non-cryptographic timer-based generator intended for testing, and a build that inherits it behaves identically in every observable respect. A deployment should additionally adopt the hedged nonce derivation of Section 4.5, which protects against biased randomness though not against a generator that repeats; it is the one hardening step in this area we have not taken, and it is listed in Table 14.

Deployments should also account for flash-specific behaviour. Wear-levelling layers keep stale copies of supposedly erased pages, so explicit erasure of the flash pages holding the nonce state is preferable to filesystem-level deletion. An attacker with physical access who can roll the partition back to an earlier state can replay a stored nonce regardless of how it was derived, since the nonce itself is what is stored; this attack lies outside the threat model above, and the defence

against it is tamper-resistant storage or a monotonic counter, not a derivation rule.

6.5 Session Binding and Concurrency

FROST's resistance to attacks on concurrent signing sessions rests entirely on the binding factor ρ_i being computed over the message and the *complete* commitment list, so that a nonce pair is bound to one specific session. Two implementation requirements follow, both of which are easy to violate without any visible symptom.

The commitment list must be canonically encoded, sorted by participant identifier, so that all parties and the aggregator derive the same ρ_i from the same set. A participant must also refuse to produce two signature shares from the same nonce pair even if the two requests differ, which requires the used-pair check of step 2 in Section 4.5 to consult persistent state rather than a variable in memory. An adversarial coordinator that can induce a participant to sign twice against overlapping commitment lists is exactly the setting the binding factor is designed to defeat, and the defence only holds if the participant's bookkeeping is correct.

6.6 The WebSocket Bridge

The proof-of-concept web integration of Section 4.7 connects the browser to the native FROST executables through a WebSocket bridge listening on `localhost:9999`. This bridge is unauthenticated: browsers do not restrict cross-origin WebSocket connections to localhost, so any website open in the user's browser, or any local process, could connect to the bridge and trigger key generation or signing. In the present system the impact is limited, since signing requires the physical boards to participate, but an attacker could still initiate protocol runs, consume nonce pairs, or observe status output.

This component is a testing convenience only and must not be carried into production. A deployable version should replace it with a channel that authenticates the caller, for example browser-extension native messaging with an allow-listed extension origin, or, in the FIDO2-compliant version discussed in Section 4.9, drop the bridge entirely in favour of the standard CTAP transport between browser and authenticator, which is what the browser is designed to speak to an authenticator in the first place.

6.7 Summary of Residual Gaps

Table 14 lists the gaps identified above in the order we would close them.

7 Conclusions

This project aimed to determine how threshold signatures could be integrated into the OFFPAD's authentication process, and the solution we reached consists of a 2-out-of-3 threshold signing scheme using the FROST protocol with

Table 14. Residual security gaps, in suggested order of remediation.

Gap	Consequence today	Remedy
Key list provisioned by hand	Enrolment decides group membership and has no tooling	A pairing procedure that establishes the list under the user's control and fixes it on the device
No user-verification binding	Coordinator can sign without a user gesture	Gate signing on the OFFPAD fingerprint check
Nonces sampled directly from the RNG	A biased entropy source weakens nonces	RFC 9591 hedged derivation $\mathsf{H}_3(rnd \,\|\| \mathsf{sk}_i)$
Unauthenticated WebSocket bridge	Any local page can start a protocol run	Replace with CTAP, or authenticated native messaging

Schnorr signatures over the secp256k1 curve, running on two different boards, the OFFPAD development board and the STM32 Nucleo-L476RG, under the Zephyr RTOS.

In this implementation, the participants jointly generate their secret shares through a distributed key generation in which every device contributes randomness to the group key, shares are exchanged end-to-end encrypted with hashed ElGamal so that the coordinator relaying them learns nothing, every message is signed under a long-term device key so that the coordinator can neither equivocate nor alter what it relays, and a blaming protocol identifies misbehaving participants. Any two devices holding a valid share can then collaborate, through a computer acting as a coordinator and signature aggregator, to produce a valid Schnorr signature over a message, without any device ever holding or reconstructing the full private key.

During the implementation, we faced various challenges, from which we drew important conclusions and identified several possible directions for future work, both of which are discussed in this section and the next.

7.1 Threshold Scheme and Communication Model

The project confirmed that Schnorr signatures, employing the FROST protocol, are a good match for embedded authentication hardware. Their mathematical structure, combined with the use of a coordinator that relays the distributed key generation and acts as signature aggregator during signing, meant that the coordination and aggregation burden could be carried by the coordinating computer, leaving the boards responsible only for the operations necessary to generate and protect their own secret material: sampling polynomial coefficients, encrypting and verifying shares, generating nonces, and computing a signature share.

Furthermore, the communication design was another important factor to take into account for the implementation. Making all exchanged data work based on a header and payload message format made it possible to support both UART and USB HID transports, which was required to support the two different boards we used. Using a coordinator instead of direct device-to-device communication also kept the coordination burden off the embedded devices, and lets the coordinator reach all available devices concurrently.

However, this implementation is not FIDO2-compliant. WebAuthn identifies algorithms by COSE identifier, and no identifier denotes Schnorr over `secp256k1`, so the credential this prototype produces cannot be described to a relying party, let alone verified by one (Section 4.9). WebAuthn does register `EdDSA`, and Ed25519 is itself a Schnorr signature, which sets the roadmap: switching to the `FROST(Ed25519, SHA-512)` ciphersuite would leave every protocol step in this report unchanged while producing signatures a WebAuthn relying party can in principle verify, whereas threshold ECDSA on P-256 is the heavier route that buys universal acceptance. This project lays the foundation for how to implement threshold signing on the OFFPAD's hardware; it does not itself provide a fully functional authentication mechanism.

7.2 Lessons Learned

Four lessons stand out, and we record them because each is cheap to act on at the start of a project of this kind and expensive to retrofit.

A relay belongs in the threat model, not in the wiring diagram. When participants cannot address one another, a coordinating computer appears in the architecture whether or not the protocol description mentions one, and it has to be analysed as an adversary rather than as a cable. Encrypting what it carries buys confidentiality; consistency is a separate property, and obtaining it requires every message to be authenticated against keys established outside the relay. We reached that conclusion eventually, and the message authentication of Section 4.3 is the result, but the message format had to be revised to carry a signature it was never designed to hold.

Ephemeral secrets acquire a storage lifecycle. Persisting nonces to flash is unavoidable given the two-round structure and the need to survive power cycles, but that decision converts a memory-safety question into a storage-durability one. The ordering of "invalidate" and "sign", together with the build-time choice of random number generator that feeds the nonces, then becomes a security property rather than an implementation detail. Neither appears in any protocol description, and neither has a run-time symptom.

A wire format should be specified as a wire format. Serialising a library's in-memory structures makes payload sizes reproducible and saves code, but a structure transmits every field it holds, including those the protocol never intended to leave the device. A message definition of a few lines, naming exactly the fields to send, costs nothing and removes the question entirely.

A ciphersuite should be chosen against the target standard rather than against library availability. `secp256k1` was the path of least resistance because a Zephyr-portable FROST library existed for it; the resulting key is the one combination of curve and algorithm that WebAuthn cannot name at all.

7.3 Software and Environment Setup

Running code of this complexity on embedded hardware taught us how much depends on the exact software environment it was built against. In this project, we had to lock the Zephyr RTOS version we were using to 3.7.0, because newer releases introduced changes that broke compatibility with the cryptographic libraries we used. The product has since moved on, the OFFPAD+ running 3.7.1, so closing that incompatibility is the first step in bringing this work to current hardware (Section 8.1).

7.4 Resource Management and Configuration

The limited memory available on both development boards made resource management extremely important in the project. Reaching a stable configuration required careful tuning of the `prj.conf` file, balancing heap and stack sizes for the memory consumed by the operations carried out in FROST, and the misconfiguration of any of these parameters resulted in allocation crashes. This tuning is also important for the flash memory, where the board's storage partition holds the key shares and nonces between the key generation and signing phases, since these values need to survive both the interval between phases and any device power-off or reboot.

The difference in hardware between the two boards was another important factor in the development. The Nucleo board is more constrained compared to the OFFPAD development board; therefore, code was designed first for the Nucleo board, as code that ran correctly on the Nucleo board was almost guaranteed to run on the OFFPAD.

7.5 Hardware Integration and Connections

A significant source of instability came from the physical layer rather than the software. Loose pins repeatedly disrupted both UART and USB communication during testing, producing intermittent failures that were easy to misattribute to the communication code instead of the physical connection. Ensuring good physical connections was as important to the project as the software design, and many debugging sessions throughout the project were resolved by reseating a cable rather than by changing code. The same was true of the OFFPAD's programming setup, which requires a Nucleo board as a programming bridge and careful GPIO mapping so that debugging and data exchange do not compete for the same interface.

The general lesson is that almost every problem that cost us significant time presented in a different layer from the one at fault. Table 10 in Section 4.11 records these symptoms and their actual causes, so that a reader rebuilding the system does not have to rediscover them.

8 Future Work

The challenges we encountered during implementation, together with the current state of the project, leave some clear possibilities for future work. These range from improving the current Schnorr/FROST implementation to adapting the threshold signing scheme to achieve FIDO2 compliance through threshold ECDSA.

8.1 Environment and RTOS Upgrades

Although locking the project to Zephyr 3.7.0 was necessary, it leaves the implementation tied to an outdated environment, and the gap has already opened in practice: the OFFPAD firmware this work was developed against ran Zephyr 3.7.0, but the current OFFPAD+ product moved to 3.7.1 in early 2025, so the prototype no longer builds against the same release as the device it targets. The incompatibility that keeps the project on 3.7.0 lies between the cryptographic library and later Zephyr releases, not in the protocol code, and resolving it is a prerequisite for carrying this work onto the OFFPAD+. A priority for future work is therefore identifying and solving the specific incompatibilities that cause build and runtime failures on newer Zephyr releases. Decoupling the project from the pinned version would also allow it to benefit from the changes and features of newer versions of Zephyr.

8.2 Dynamic Resource Allocation

The memory configuration used throughout this project was tuned for the two boards used, which makes it fragile when applied to different hardware. Future work should focus on replacing the current static `prj.conf` tuning with a configuration that adapts automatically to the resources available on a given board. This would remove the need to manually and repeatedly tune heap, stack, and flash settings for every new target device and would improve how portable the implementation is across different embedded devices.

8.3 Communication over Bluetooth

The current implementation relies exclusively on USB communication between the coordinator and the participant devices. While this provided a reliable channel for development and testing, it limits the practical usability and scalability of the system, since it requires a physical cable per device and constrains how many participants can realistically be connected at once. Future work should investigate adapting this model to operate over Bluetooth Low Energy (BLE)

leveraging the OFFPAD's existing BLE compatibility, which would remove this physical constraint. Because the protocol already treats the links as a network under the coordinator's control (Section 6.1), a wireless link does not change the cryptographic analysis. What it adds is engineering: pairing the devices with the coordinator, coping with a lossy link that a nearby attacker can also jam, and the larger messages that BLE's small packets must carry.

8.4 Threshold Parameter Exploration

The implementation is written for general T and N, and local participants (Section 4.4.4) allow any configuration to be exercised on the coordinator, but every measurement in this report was taken in the 2-out-of-3 configuration with two boards and one local participant. Although this setup already demonstrates the benefit of threshold signatures and enables a practical implementation, being able to sign as long as any two of three participants are available, it does not establish whether this is the best balance between security and efficiency for the OFFPAD. Future work should evaluate configurations with a larger number of total participants N and threshold T, to observe how communication overhead and signing latency increase as more devices are added, and to identify the point at which additional participants stop providing a meaningful security benefit relative to their added coordination cost.

8.5 Porting to Ed25519: A Concrete Work Breakdown

Section 4.9 argues that the cheapest route to a credential a relying party can verify is to keep the protocol and change the ciphersuite, from `FROST(secp256k1, SHA-256)` to `FROST(Ed25519, SHA-512)`. That claim deserves substantiation, because "change the ciphersuite" can hide a great deal of work. This section sets out what the change actually involves. The conclusion is that the protocol logic is genuinely untouched, and that essentially all the effort falls in serialisation, the hash layer, and point validation.

8.5.1 What Does Not Change Not one step of the distributed key generation of Section 4.4 or of the two-round signing of Section 4.5 is altered. The polynomial sampling, the commitments, the proof of knowledge, the share verification equation, the Lagrange interpolation, the binding factors, the challenge, the signature share equation, and the aggregator's per-share check are all written in terms of a prime-order group and its scalar field, and both ciphersuites supply one. The blaming protocol, the coordinator model, the concurrency strategy, and the flash-persistence discipline are likewise unaffected. So is the security discussion of Section 6: the coordinator equivocation gap, the nonce ordering requirement, and the session-binding requirements are all properties of the construction rather than of the group.

Table 15. The two RFC 9591 ciphersuites compared. The protocol is identical; everything below the group is not.

	`FROST(secp256k1, SHA-256)`	`FROST(Ed25519, SHA-512)`
Context string	`FROST-secp256k1-SHA256-v1`	`FROST-ED25519-SHA512-v1`
Group order q	$\approx 2^{256}$, cofactor 1	$2^{252} + 27742\ldots648493$, cofactor 8
$\mathsf{H}_1, \mathsf{H}_2, \mathsf{H}_3$	`hash_to_field` via `expand_message_xmd` with SHA-256	SHA-512, read little-endian, reduced mod q
H_2 domain separation	DST = context $\|$ `"chal"`	**none**: $\mathsf{H}_2(m) = \text{SHA-512}(m)$
Scalar encoding	SEC1, 32 bytes, **big-endian**	32 bytes, **little-endian**, top three bits zero
Point encoding	SEC1 compressed, **33 bytes**	RFC 8032 §5.1.2, **32 bytes**
Point validation	on-curve, in range, not the identity	additionally: canonical encoding, **prime-order subgroup**
Verification	Appendix B of RFC 9591	RFC 8032 §5.1.7 with $[8][s]G = [8]R + [8][c]\mathsf{pk}$

8.5.2 What Does Change, and Where the Effort Goes Table 15 sets the two ciphersuites side by side. Four differences are substantive.

The hash layer is a different construction, not a different hash. For `secp256k1`, H_1, H_2 and H_3 are `hash_to_field` instantiated with `expand_message_xmd` over SHA-256, a variable-length expansion function from the hash-to-curve specification. For Ed25519 they are a plain SHA-512 call over a prefixed input, with the 64-byte digest read as a little-endian integer and reduced modulo the group order. The Ed25519 versions are simpler, but they are not obtained by substituting a hash function into the existing code; the `expand_message_xmd` machinery is removed rather than reparameterised.

H_2 loses its domain separation, and this is the point. For Ed25519 the challenge is computed as $\mathsf{H}_2(m) = \text{SHA-512}(m)$ with *no* context prefix, unlike every other hash in the suite. This looks like an inconsistency and is in fact the mechanism that makes the whole route work: RFC 8032 verification computes the challenge as $\text{SHA-512}(R \,\|\, A \,\|\, M)$, where A is the encoded public key and M the message, so any prefix would make FROST signatures unverifiable by a standard Ed25519 verifier. An implementer who "fixes" this apparent omission by adding domain separation produces a scheme that is self-consistent and useless for the purpose.

Endianness flips for scalars. SEC1 scalars are big-endian; Ed25519 scalars are little-endian with the top three bits cleared. Every scalar serialisation and deserialisation path is affected, including the ones inside the share encryption of Section 4.4.2 and the values written to flash. This is the classic silent-failure change: byte-reversed scalars produce valid-looking values that fail only at verification, and existing persisted state from a `secp256k1` deployment is not merely incompatible but misinterpretable.

Cofactor 8 brings back subgroup checks. Section 2 notes that `secp256k1`'s cofactor of 1 let us reduce point validation to an on-curve check. On edwards25519 the group of points has order $8q$, so every deserialised point must additionally be checked to be canonically encoded and to lie in the prime-order subgroup, and low-order points must be rejected. This affects every received commitment, every ephemeral encryption key, and every nonce commitment. It is also why RFC 9591 specifies verification with the cofactored equation $[8][s]G = [8]R + [8][c]\mathsf{pk}$. This does *not* threaten interoperability: an honestly produced FROST signature has R and `pk` in the prime-order subgroup and satisfies the cofactorless equation too, so both strict and permissive RFC 8032 verifiers accept it. The two equations differ only in which malformed signatures they reject, which is a concern for the verifier's own consistency rather than for this route.

8.5.3 Consequences for the Rest of the System Payloads shrink if the wire format adopts the canonical 32-byte point encoding in place of the library's 64-byte internal form. Recomputing the sizes of Section 5.1.2 with the RFC 8032 encoding gives a public key structure of 72 bytes rather than 136, VSS commitments of $72 + 32T$ rather than $104 + 64T$ (so 136 rather than 232 bytes at $T = 2$), and a round-two request of $36 + 68T$ rather than $36 + 132T$. Two qualifications keep this in proportion. Most of the saving is available today, on secp256k1, by sending compressed 33-byte SEC1 points (Section 5.1.2); the curve change merely makes the canonical encoding the natural one. Since Section 4.10 finds that, once the fixed pauses of the instrumented build are discounted, every message crosses either link in tens of milliseconds against roughly half a second of arithmetic per signature share, the gain is one or two 61-byte chunks per message, worth having but not transformative.

The COSE encoding also simplifies. An Ed25519 credential key is an OKP key rather than an EC2 key: key type `OKP` (`1`), curve `Ed25519` (`6`), algorithm `EdDSA` (`-8`), and a single 32-byte `x` parameter, with no y coordinate. The manual CBOR construction described in Section 4.8 becomes correspondingly simpler, although, as that section explains, the substitution approach is a dead end regardless and the real work is on the CTAP side.

One subtlety deserves recording because it will surprise anyone who knows RFC 8032. An Ed25519 private key in RFC 8032 is a 32-byte seed that is hashed and clamped to produce the actual scalar, and the public key is derived from that clamped scalar. A FROST group key has no such seed: the group secret is a sum

of shares, an ordinary field element, and the group public key is $[\mathtt{sk}]G$ directly. The resulting signatures verify under RFC 8032 verification, which is all that matters, but the key pair does not correspond to any RFC 8032 seed and cannot be exported as one. Any tooling that expects to round-trip an Ed25519 private key through a seed will fail, and any test harness that generates a reference key that way is testing something else.

8.5.4 Libraries The practical obstacle is not the mathematics but finding an implementation that exposes the right level of abstraction under Zephyr. Most Ed25519 libraries offer only `sign` and `verify`, which is useless here; what is needed is scalar multiplication of an arbitrary point, point addition, scalar arithmetic modulo the group order, and canonical encoding and decoding with subgroup validation. Three options are worth evaluating, and the choice deserves a short spike before committing.

libsodium exposes exactly the required primitives through its low-level `crypto_core` and `crypto_scalarmult` interfaces for edwards25519, including canonical-point and subgroup validation. It is mature and constant-time, but is a larger dependency than the current one and its suitability for the flash budget of the target boards needs checking rather than assuming.

Monocypher is designed for embedded use and is very small, and exposes low-level edwards25519 operations. It is the most likely fit for the flash budget, and the most likely to require writing the missing pieces by hand.

The Zcash Foundation's Rust reference implementation [Zca24] implements RFC 9591 including a `frost-ed25519` ciphersuite crate, is assessed by an external auditor, and is the natural source of test vectors and a cross-checking oracle even if it is not what ships on the board. Whether it can be built for a Cortex-M target in a `no_std` configuration, and whether mixing Rust into the existing Zephyr build is acceptable, are the open questions; if the answer is yes, adopting it directly would remove the library problem altogether at the cost of a new toolchain in the build.

8.5.5 Suggested Sequence Port and validate against RFC 9591's Ed25519 test vectors on the host first, before any board is involved, since the vectors cover precisely the hash and encoding layer where all the risk in this change lies. Then confirm on the host that a signature produced by the ported implementation verifies under an independent, unmodified Ed25519 verifier. Only then move to the board, where the remaining work is the flash and payload changes and re-tuning the memory configuration for the different field sizes. NIST's draft notes on threshold EdDSA and Schnorr signatures [BD22] are a useful companion for this work, and tracking them is prudent given that the same document is likely to shape how such schemes are eventually validated.

8.6 Threshold ECDSA for Universal Compatibility

The route that removes the deployment caveat entirely is threshold ECDSA over P-256, which yields `ES256` credentials that every relying party accepts. This is the harder path, and it is worth pursuing precisely because it is the only one whose result works everywhere.

Among the threshold ECDSA constructions we discussed in the research phase of this project, the three-round protocol of Doerner, Kondi, Lee, and shelat [DKLs24] seemed like the most promising one. It avoids expensive zero-knowledge proofs, relying on lightweight statistical consistency checks instead, and performs the secure multiplications that ECDSA signing requires with vector oblivious linear evaluation (VOLE), splitting the signing equation into three rounds. The rounds consist of: handling nonce commitment, decommitment with consistency checking, and final signature share aggregation. This structure keeps the protocol's round complexity lower compared to earlier threshold ECDSA constructions, which matters on a system in which every round is relayed through the coordinator.

Adopting this scheme would require:

An additional communication round: The current broadcast model should be extended to have the same communication structure, but with an additional round.

VOLE integration on constrained hardware: Implementing VOLE efficiently on devices with the memory and processing constraints would probably be more demanding than the Schnorr operations used by FROST, and may require its own dedicated evaluation of memory footprint and execution time before being considered viable for the OFFPAD.

A suitable cryptographic library: Unlike the `secp256k1` FROST library used in this project, which is Schnorr-oriented, a threshold ECDSA implementation would need a library specific to the elliptic curve and VOLE this protocol depends on, adapted to Zephyr in the same way the current library was.

While threshold ECDSA is expected to be substantially more computationally demanding than the FROST implementation, achieving `ES256` compliance would allow deployment against the existing relying-party infrastructure without any assumption about what a given service supports. Given that the Ed25519 route of Section 8.5 is far cheaper, we would sequence the two: build the Ed25519 version first to obtain a credential that is in principle verifiable and to exercise the full CTAP path end to end, and treat threshold ECDSA as the subsequent step whose only additional contribution, though a decisive one, is universal acceptance.

8.7 Device Loss, Replacement, and Share Refresh

There is a gap in this work that is easy to miss precisely because threshold signatures are motivated by device loss, and it is arguably the most important omission in the design. The system as built can *tolerate* a lost device, in the sense

that T of the remaining devices can still sign. It has no mechanism for *recovering* from one.

Consider a 2-out-of-3 deployment in which a user loses one device. Two things should happen and neither is possible today. The user should be able to enrol a replacement, restoring the original redundancy, and the share held by the lost device should be rendered useless, since an attacker who finds it now holds one of the two shares needed. Both are solved by the same primitive, *share refresh*: the surviving participants jointly re-randomise the sharing of the same secret, producing a fresh set of shares for a possibly different participant set, in such a way that old shares and new shares cannot be mixed. The group secret, and therefore the group public key, is unchanged.

That last property is what makes this a requirement rather than a nicety in the FIDO setting. The group public key *is* the credential registered with the relying party. If replacing a device forced a new group key, every relying party would have to re-register the credential, which is exactly the painful recovery flow that threshold signatures were supposed to eliminate; the system would tolerate loss no better than a backup authenticator does. With share refresh, device replacement is invisible to every relying party, which is a strong claim to be able to make and one the current implementation cannot support.

The same primitive gives *proactive security*: refreshing shares periodically means an attacker must compromise T devices within a single refresh epoch rather than over the lifetime of the credential, which for a device carried around for years is a substantially different proposition. This is well-trodden ground in the literature on proactive secret sharing, and the construction is closely related to the DKG already implemented: each participant deals a sharing of *zero* rather than of a fresh secret, and the shares are added to the existing ones, which leaves the constant term untouched while re-randomising everything else. Much of the machinery in Section 4.4, the commitments, the verification equation, the encrypted transport, and the blaming protocol, is reusable as it stands.

Three questions need answering for a deployment, and we flag them rather than resolve them. What is the refresh policy, given that the devices are rarely all present at once? How is an old share revoked in a threshold system with no online authority, given that refresh only invalidates a share for participants who complete the refresh, and a device that was offline keeps a share from the old epoch? And how is a refresh authorised, so that an attacker holding $T-1$ devices cannot initiate one that excludes the honest device?

8.8 Open Design Questions for a Threshold Authenticator

Three further questions arise specifically from the interaction between threshold signing and the FIDO2 data model. None is addressed by the current implementation, and none has an obvious answer.

Attestation. A WebAuthn attestation statement is a signature by the authenticator, under a manufacturer-provisioned attestation key, testifying to what kind of device generated the credential. In a threshold system there is no single authenticator. Should each participating device attest separately, leaving the relying party to interpret a set of statements it has no way to combine? Should the coalition produce a threshold attestation, which requires the attestation key itself to be shared and raises the question of who provisions it? Or is the honest position that a threshold credential simply cannot be attested in the current model, and the relying party must accept `attestation:"none"`? This is a specification-level question and the answer bears on whether such an authenticator can ever be certified.

The signature counter. `authenticatorData` carries a signature counter that a relying party may use to detect cloned authenticators. With N devices signing in varying coalitions, there is no single monotonic counter: each device knows only the sessions it took part in. A naive per-device counter will appear to a relying party as exactly the anomaly the counter exists to detect. The options are to have the coalition agree on a counter value as part of the signing round, which adds state that must survive aborts, or to report zero, which WebAuthn treats as the value an authenticator reports when it does not maintain a counter, even though the specification says an authenticator SHOULD implement one [W3C21]. The latter is probably right, but it should be a decision rather than an accident.

Enrolment and everyday usability. The current registration flow requires a terminal window and a user who understands what a distributed key generation is. A threshold authenticator is only better than a single one if the user actually configures more than one device, so the enrolment flow is not a cosmetic layer over the cryptography; it is the mechanism by which the security property comes to exist. Designing and then *evaluating* that flow with real users, including the recovery path of Section 8.7, is necessary work and is closer to the security core than it appears.

8.9 Post-Quantum Considerations

The scheme implemented here rests on the hardness of discrete logarithms and offers no security against a quantum adversary. This is not urgent for an authentication credential, whose compromise window is the credential's lifetime rather than decades, and it is not specific to our work, since FIDO2's own algorithms are in the same position. It is nonetheless worth recording where the difficulty would lie. Bindel, Cremers, and Zhao [BCZ23] have already studied post-quantum instantiations of FIDO2 itself, so the single-device path is charted. The threshold path is not: post-quantum threshold signatures are an active research area, the candidate constructions are considerably more expensive than FROST in both computation and communication, and the cost structure we measure in Section 4.10 suggests where it would hurt: the heavier arithmetic would fall on boards that already spend half a second per signature share, and the

larger signatures and commitments of lattice-based schemes would multiply the number of 61-byte chunks per message. Any future post-quantum variant should therefore be designed against the memory, computation and transport budgets of the boards from the outset rather than ported and measured afterwards.

8.10 Becoming an Authenticator

Independently of the signature scheme, a deployable system must speak CTAP2 to the client rather than going through the WebSocket bridge of Section 6.6. This means implementing the authenticator API on the device: constructing `authenticatorData` with the relying party hash, flags and signature counter, managing discoverable credentials, and gating each signing operation behind the user verification gesture the OFFPAD already provides through its fingerprint sensor. That last point is also a security requirement rather than a convenience, since the current prototype signs whatever the coordinator presents. The threshold protocol then sits underneath the assertion step, and the natural place for the coordinator role is whichever device the client is connected to.

8.11 Closing the Security Gaps

Table 14 in Section 6 lists the residual gaps in the order we would address them. The enrolment procedure that establishes the list of device public keys is the first of them, because it is what the message authentication of Section 4.3 ultimately rests on, and it does not depend on any of the larger directions above.

References

ABR01. Michel Abdalla, Mihir Bellare, and Phillip Rogaway. The Oracle Diffie-Hellman assumptions and an analysis of DHIES. In *Topics in Cryptology — CT-RSA 2001*, volume 2020 of *Lecture Notes in Computer Science*, pages 143–158. Springer, 2001. https://doi.org/10.1007/3-540-45353-9_12

Ban22. Banca d'Italia. secp256k1-frost: Implementation of Flexible Round-Optimized Schnorr Threshold Signatures on secp256k1. https://github.com/bancaditalia/secp256k1-frost, 2022. GitHub repository.

BBCW21. Manuel Barbosa, Alexandra Boldyreva, Shan Chen, and Bogdan Warinschi. Provable security analysis of FIDO2. In *Advances in Cryptology — CRYPTO 2021, Part IV*, volume 12827 of *Lecture Notes in Computer Science*, pages 125–156. Springer, 2021. https://doi.org/10.1007/978-3-030-84252-9_5

BCK+22. Mihir Bellare, Elizabeth Crites, Chelsea Komlo, Mary Maller, Stefano Tessaro, and Chenzhi Zhu. Better than advertised security for non-interactive threshold signatures. In *Advances in Cryptology — CRYPTO 2022, Part IV*, volume 13510 of *Lecture Notes in Computer Science*, pages 517–550. Springer, 2022. https://doi.org/10.1007/978-3-031-15985-5_18

BCZ23. Nina Bindel, Cas Cremers, and Mang Zhao. FIDO2, CTAP 2.1, and WebAuthn 2: Provable security and post-quantum instantiation. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1471–1490. IEEE, 2023. https://doi.org/10.1109/SP46215.2023.10179454

BD22. Luís T. A. N. Brandão and Michael Davidson. Notes on Threshold EdDSA/Schnorr Signatures. NIST Internal Report (Initial Public Draft) NIST IR 8214B ipd, National Institute of Standards and Technology, August 2022. https://doi.org/10.6028/NIST.IR.8214B.ipd

BLL+21. Fabrice Benhamouda, Tancrède Lepoint, Julian Loss, Michele Orrù, and Mariana Raykova. On the (in)security of ROS. In *Advances in Cryptology — EUROCRYPT 2021, Part I*, volume 12696 of *Lecture Notes in Computer Science*, pages 33–53. Springer, 2021. https://doi.org/10.1007/978-3-030-77870-5_2

BP26. Luís T. A. N. Brandão and René Peralta. NIST First Call for Multi-Party Threshold Schemes. NIST Internal Report NIST IR 8214C, National Institute of Standards and Technology, January 2026. https://doi.org/10.6028/NIST.IR.8214C

BS23. Dan Boneh and Victor Shoup. A graduate course in applied cryptography. https://toc.cryptobook.us/, 2023. Version 0.6.

Cer09. Certicom Research. SEC 1: Elliptic Curve Cryptography, Version 2.0. Technical report, Standards for Efficient Cryptography Group, 2009. https://www.secg.org/sec1-v2.pdf.

Cer10. Certicom Research. SEC 2: Recommended Elliptic Curve Domain Parameters, Version 2.0. Technical report, Standards for Efficient Cryptography Group, 2010. https://www.secg.org/sec2-v2.pdf.

CKGW24. Deirdre Connolly, Chelsea Komlo, Ian Goldberg, and Christopher A. Wood. The Flexible Round-Optimized Schnorr Threshold (FROST) Protocol for Two-Round Schnorr Signatures. RFC 9591, RFC Editor, 2024. https://doi.org/10.17487/RFC9591

CKM23. Elizabeth Crites, Chelsea Komlo, and Mary Maller. Fully adaptive Schnorr threshold signatures. In *Advances in Cryptology — CRYPTO 2023, Part I*, volume 14081 of *Lecture Notes in Computer Science*, pages 678–709. Springer, 2023. https://doi.org/10.1007/978-3-031-38557-5_22

DKLs24. Jack Doerner, Yashvanth Kondi, Eysa Lee, and abhi shelat. Threshold ECDSA in three rounds. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 3053–3071. IEEE, 2024. https://doi.org/10.1109/SP54263.2024.00178

DLCGM23. Emma Dauterman, Danny Lin, Henry Corrigan-Gibbs, and David Mazières. Accountable authentication with privacy protection: The Larch system for universal login. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 81–98. USENIX Association, 2023. https://www.usenix.org/conference/osdi23/presentation/dauterman.

Duo19. Duo Security. Guide to Web Authentication. https://webauthn.guide, 2019.

ElG85. Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, 1985. https://doi.org/10.1109/TIT.1985.1057074

Fel87. Paul Feldman. A practical scheme for non-interactive verifiable secret sharing. In *28th Annual Symposium on Foundations of Computer Science (SFCS 1987)*, pages 427–438, Los Angeles, CA, USA, October 1987. IEEE Computer Society. https://doi.org/10.1109/SFCS.1987.4

FGK+20. Nick Frymann, Daniel Gardham, Franziskus Kiefer, Emil Lundberg, Mark Manulis, and Dain Nilsson. Asynchronous remote key generation: An

analysis of Yubico's proposal for W3C WebAuthn. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS '20)*, pages 939–954. Association for Computing Machinery, 2020. https://doi.org/10.1145/3372297.3417292

FID22. FIDO Alliance. FIDO2. https://fidoalliance.org/fido2/, May 2022.

FID26. FIDO Alliance. Client to Authenticator Protocol (CTAP). Proposed Standard, FIDO Alliance, February 2026. Version 2.3, https://fidoalliance.org/specs/fido-v2.3-ps-20260226/fido-client-to-authenticator-protocol-v2.3-ps-20260226.html.

Fro25. Frostsnap. Open-source FROST signing devices. https://frostsnap.com, 2025. Source code at https://github.com/frostsnap/frostsnap, accessed September 2026.

FS87. Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In Andrew M. Odlyzko, editor, *Advances in Cryptology — CRYPTO '86*, volume 263 of *Lecture Notes in Computer Science*, pages 186–194, Berlin, Heidelberg, 1987. Springer. https://doi.org/10.1007/3-540-47721-7_12

Gar25. Javier Garcia. Threshold Signatures for FIDO Authentication. Master's thesis, Norwegian University of Science and Technology, Trondheim, Norway, 2025. https://nva.sikt.no/registration/019a0aba4573-1868fcdf-2d4d-481a-98a8-f9c492cc0dd8.

GG18. Rosario Gennaro and Steven Goldfeder. Fast multiparty threshold ECDSA with fast trustless setup. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*, pages 1179–1194. Association for Computing Machinery, 2018. https://doi.org/10.1145/3243734.3243859

GJKR99. Rosario Gennaro, Stanisław Jarecki, Hugo Krawczyk, and Tal Rabin. Secure distributed key generation for discrete-log based cryptosystems. In *Advances in Cryptology — EUROCRYPT '99*, volume 1592 of *Lecture Notes in Computer Science*, pages 295–310. Springer, 1999. Journal version: J. Cryptology 20(1):51–83, 2007, https://doi.org/10.1007/s00145-006-0347-3. https://doi.org/10.1007/3-540-48910-X_21

GMR85. Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof-systems. In *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, STOC '85, pages 291–304, New York, NY, USA, 1985. Association for Computing Machinery. https://doi.org/10.1145/22145.22178

GS26. Javier Garcia and Tjerand Silde. Implementing threshold signatures for FIDO authentication. In *Proceedings of the Norwegian Information Security Conference (NISK 2026)*, Norsk IKT-konferanse for forskning og utdanning (NIKT), 2026. To appear.

HLW23. Lucjan Hanzlik, Julian Loss, and Benedikt Wagner. Token meets wallet: Formalizing privacy and revocation for FIDO2. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1491–1508. IEEE, 2023. https://doi.org/10.1109/SP46215.2023.10179373

HMV04. Darrel Hankerson, Alfred J. Menezes, and Scott Vanstone. *Guide to Elliptic Curve Cryptography*. Springer, 2004. https://doi.org/10.1007/b97644

IAN. IANA. CBOR Object Signing and Encryption (COSE) Algorithms Registry. https://www.iana.org/assignments/cose/cose.xhtml#algorithms. Accessed September 2026.

JL17. Simon Josefsson and Ilari Liusvaara. Edwards-Curve Digital Signature Algorithm (EdDSA). RFC 8032, RFC Editor, 2017. https://doi.org/10.17487/RFC8032

JMV01. Don Johnson, Alfred Menezes, and Scott Vanstone. The Elliptic Curve Digital Signature Algorithm (ECDSA). *International Journal of Information Security*, 1(1):36–63, August 2001. https://doi.org/10.1007/s102070100002

Jon20. Michael B. Jones. CBOR Object Signing and Encryption (COSE) and JSON Object Signing and Encryption (JOSE) Registrations for Web Authentication (WebAuthn) Algorithms. RFC 8812, RFC Editor, 2020. https://doi.org/10.17487/RFC8812

KG21. Chelsea Komlo and Ian Goldberg. FROST: Flexible round-optimized Schnorr threshold signatures. In *Selected Areas in Cryptography — SAC 2020*, volume 12804 of *Lecture Notes in Computer Science*, pages 34–65. Springer, 2021. https://doi.org/10.1007/978-3-030-81652-0_2

Lin17. Yehuda Lindell. Fast secure two-party ECDSA signing. In *Advances in Cryptology — CRYPTO 2017, Part II*, volume 10402 of *Lecture Notes in Computer Science*, pages 613–644. Springer, 2017. https://doi.org/10.1007/978-3-319-63715-0_21

Mag16. Krists Magons. Applications and Benefits of Elliptic Curve Cryptography. In *Proceedings of the Student Research Forum Papers and Posters at SOFSEM 2016*, volume 1548 of *CEUR Workshop Proceedings*, pages 32–42. CEUR-WS.org, 2016. https://ceur-ws.org/Vol-1548/032-Magons.pdf.

MCS+17. Vasilios Mavroudis, Andrea Cerulli, Petr Svenda, Dan Cvrcek, Dusan Klinec, and George Danezis. A touch of evil: High-assurance cryptographic hardware from untrusted components. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS 2017)*, pages 1583–1600. ACM, 2017. https://doi.org/10.1145/3133956.3133961

Mil86. Victor S. Miller. Use of elliptic curves in cryptography. In Hugh C. Williams, editor, *Advances in Cryptology — CRYPTO '85 Proceedings*, volume 218 of *Lecture Notes in Computer Science*, pages 417–426, Berlin, Heidelberg, 1986. Springer. https://doi.org/10.1007/3-540-39799-X_31

MN25. Mark Manulis and Hugo Nartz. Distributed asynchronous remote key generation. In *Applied Cryptography and Network Security — ACNS 2025*, volume 15825 of *Lecture Notes in Computer Science*, pages 486–515. Springer, 2025. https://doi.org/10.1007/978-3-031-95761-1_17

Ped92. Torben Pryds Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In Joan Feigenbaum, editor, *Advances in Cryptology — CRYPTO '91*, volume 576 of *Lecture Notes in Computer Science*, pages 129–140, Berlin, Heidelberg, 1992. Springer. https://doi.org/10.1007/3-540-46766-1_9

Pol78. J. M. Pollard. Monte Carlo methods for index computation (mod p). *Mathematics of Computation*, 32(143):918–924, 1978. https://doi.org/10.1090/S0025-5718-1978-0491431-9

RNM+24. Tim Ruffing, Jonas Nick, Illia Melnyk, Mariia Zhvanko, and Sivaram Dhakshinamoorthy. ChillDKG: Distributed key generation for FROST. Bitcoin Improvement Proposal draft, https://github.com/BlockstreamResearch/bip-frost-dkg, 2024. Accessed October 2026.

RRJ+22. Tim Ruffing, Viktoria Ronge, Elliott Jin, Jonas Schneider-Bensch, and Dominique Schröder. ROAST: Robust asynchronous Schnorr threshold signatures. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22)*, pages 2551–2564. Association for Computing Machinery, 2022. https://doi.org/10.1145/3548606.3560583

Sch91. Claus-Peter Schnorr. Efficient signature generation by smart cards. *Journal of Cryptology*, 4(3):161–174, January 1991. https://doi.org/10.1007/BF00196725

Sch05. Berry Schoenmakers. Verifiable secret sharing. In Henk C. A. van Tilborg, editor, *Encyclopedia of Cryptography and Security*, pages 645–647. Springer US, Boston, MA, 2005. https://doi.org/10.1007/0-387-23483-7_452

Sha79. Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, November 1979. https://doi.org/10.1145/359168.359176

Sil09. Joseph H. Silverman. *The Arithmetic of Elliptic Curves*, volume 106 of *Graduate Texts in Mathematics*. Springer, New York, NY, 2 edition, 2009. https://doi.org/10.1007/978-0-387-09494-6

W3C21. W3C. Web Authentication: An API for accessing Public Key Credentials — Level 2. W3C Recommendation, World Wide Web Consortium, April 2021. https://www.w3.org/TR/webauthn-2/.

W3C26. W3C. Web Authentication: An API for accessing Public Key Credentials — Level 3. W3C Recommendation, World Wide Web Consortium, August 2026. https://www.w3.org/TR/webauthn-3/.

Won21. David Wong. *Real-World Cryptography*. Manning Publications, Shelter Island, NY, 2021. https://www.manning.com/books/real-world-cryptography.

Zca24. Zcash Foundation. FROST: Rust implementation of Flexible Round-Optimized Schnorr Threshold signatures. https://github.com/ZcashFoundation/frost, 2024. Reference implementation of RFC 9591.