Trout++

Robust Asynchronous Two-Round ECDSA for Arbitrary Thresholds

Hila Dahari-Garbian [1] Ariel Nof [2] Luke Parker[3]

[1]Reichman University @ Herzliya, Israel
hiladahari41@gmail.com

[2]Bar-Ilan University @ Ramat-Gan, Israel
ariel.nof@biu.ac.il

[3]Serai DEX @ Florida, United States of America
lukeparker@serai.exchange

Presented at NIST TCPT2, 2026-July-7

What's in a standard? (Theoretically)

- Formal Description and Security Proofs
- Technical Specification
- Reference Implementation

What's in a standard? (Practically)

- Formal Description and Security Proofs
  - Presentation of background
  - Justifications for the cryptographic assumptions
  - Exceptionally clear to review security proofs
- Technical Specification
  - Underlying cryptographic primitives
  - Algorithms composing the protocol itself
  - Exact wire specification and test vectors
  - Commentary on alternative algorithms and practical considerations, implementation advice
- Reference Implementation
  - A correct implementation of everything specified
  - Easy to read and compare against
  - Follows best practices for safety and could be deployed
  - Demonstrates the protocol's performance and practicality

What's in a standard? (Practically)

- Methods of Deployment
  - Does the protocol guarantee delivery of the output (is it robust)?
  - Does the protocol require a broadcast channel? Does it detail how to implement one?
  - Does the protocol require agreement on an Asynchronous Common Subset (ACS)? Is such a specification available? What is the complexity of the combined protocol?
  - If the protocol aborts, is any information from the setup revealed to the adversary?

Trout++ & ROAST

- A complete threshold ECDSA signing service which aims to respond to all of the above
- Requires no setup other than the output of a distributed key generation (DKG) protocol
- Supports arbitrary thresholds and does not require an honest majority
- Guarantees output of a signature so long as a threshold is honest and online
- Works over an asynchronous network, with nothing other than authenticated peer-to-peer channels

Trout++

- A two-round threshold ECDSA signing protocol for arbitrary thresholds
- Supports identifying faulty participants upon abort
- Supports performing the first round without any context (presigning)
- Constant per-participant upload in the bulletin board model
- Constant per-participant prover complexity
- Linear per-participant verifier complexity (not vulnerable to timing analysis and can be done in variable-time without a loss in security)
- Constant-time reference implementation in Rust
- Composable with ROAST

ROAST

- Straight-forward transform for eligible synchronous two-round threshold signing protocols into robust asynchronous threshold signing protocols
- Intended for use with FROST, a leading threshold Schnorr signature scheme
- Only requires authenticated peer-to-peer channels

Trout++ is the first threshold ECDSA signing protocol compatible with ROAST, and their composition is the first robust asynchronous threshold ECDSA signing scheme without an honest majority assumption.

Practicality

- Complete presentation comprehensive to all considerations
- Requires the Decisional Diffie-Hellman, Hard Subgroup Membership[1], and Adaptive Root assumptions over a class group of unknown order, with a known-order subgroup where the discrete-log problem is easy[2]
- Planning full specification of all class group algorithms
- ≈1.8-4.6 KB per-participant upload per Trout++ invocation[3]
- ≈4.5-14 seconds to run the prover in constant-time[4]
- ≈160-450 milliseconds to run the prover in variable-time[5]

Benchmarks performed on a single-thread of an M2 Pro CPU.

---

[1] https://eprint.iacr.org/2018/791

[2] https://eprint.iacr.org/2015/047

[3] Without Identifiable Aborts and With a Prior One-Round Setup - With Identifiable Aborts and Without Any Prior Additional Setup

[4] Portable code compiled with target-specific optimizations

[5] BICYCL (premised on GMP) compiled with target-specific optimizations

Publications

- Trout++ was accepted to ESORICS, 2026
- The full paper, with security proofs, will be published to the IACR ePrint server shortly
- The current implementation, and conference paper, is publicly available now @ `https://github.com/kayabaNerve/trout`

Ongoing Work

- Accumulating decades of miscellaneous literature into a complete reference for engineers (not academics) to implement
- Continuing specification of each underlying algorithm, ideally including the complete proofs on their memory usage and termination bounds
- Further optimizations to the constant-time prover, which does demonstrate a severe gap between the average-case and worst-case termination bounds
- If time allows, specifying not just Trout++, as composable with ROAST, but also providing a reference implementation and exact specification for the ROAST transformation

Optimized Scaled Decryption

- Optimized Scaled Decryption is a two-round protocol to simultaneously scale and decrypt a ciphertext, without revealing the scalar or original plaintext.
- We work with a class group (whose setup is transparent) where G is a uniformly-sampled generators of unknown-order, H is a generator of order q where the discrete-log problem is easy, and $\gcd(\text{ord}(G),\ q) = 1$.
- We commit to the scalar, and create the ciphertext to the value, in round one.
  - Sample $\alpha,\ \beta \in \mathbb{Z},\ a,\ b \in \mathbb{F}_q$
  - Publish $A = \alpha \cdot G + a \cdot H$
  - Publish $B = \beta \cdot (q \cdot G) + b \cdot G$
- We scale and decrypt in round two.
  - Publish $F = ((\beta * q) + b) \cdot A - \alpha \cdot B = (a * b) \cdot H$

Trout++

1.
   - Sample $k,\ u \in \mathbb{F}_q$
   - Publish $R' = k \cdot E$
   - Publish Optimized Scaled Decryption's commitment B for the scalar u
   - Create ciphertexts $N',\ D'$ for $x,\ k$ respectively
2.
   - Sample randomizers $\rho,\ \mu \in \mathbb{F}_q$ from the transcript
   - Set $R = \rho \cdot R' + \mu \cdot E$[6]
   - Derive ciphertexts $N,\ D$ for $m + R_x * x,\ \rho * k + \mu$ respectively
   - Perform Optimized Scaled Decryption's second round with $N,\ B$ to receive n, then with $D,\ B$ to receive d
   - Output $R_x,\ n * d^{-1}$ as the ECDSA signature

[6] Technique from `https://eprint.iacr.org/2024/1950`