From Papers to Production

Cryptography Engineering that sucks less

July 2026

Alireza Rafiei

Alireza Rafiei

Mateusz Kramarczyk

Alberto Ibarrondo
Now at Arcium

Andrei Pshenkin

Paul Germouty
Now at Geminos

Hoang Ong

Alexandre Adomnicai
Now at DV Labs

BRON IS DESIGNED FOR HIGH-NET-WORTH INDIVIDUALS USING CRYPTO

MPC SOLUTION

Institutional-grade security delivered in sleek UX to HNWI

SIMPLE TRANSFERS

Simple transfers

STAKING

All blockchains through P2P.org

SWAPS

Anything into anything swaps with low fees

HIDDEN VAULTS

Stay protected from the street attacks

SEEDLESS RECOVERY

Secure recovery, no seed phrase

POLICY ENGINE

Stay protected from the street attacks

BATCH PAYMENTS

Make crypto payouts to multiple addresses at once

INHERITANCE

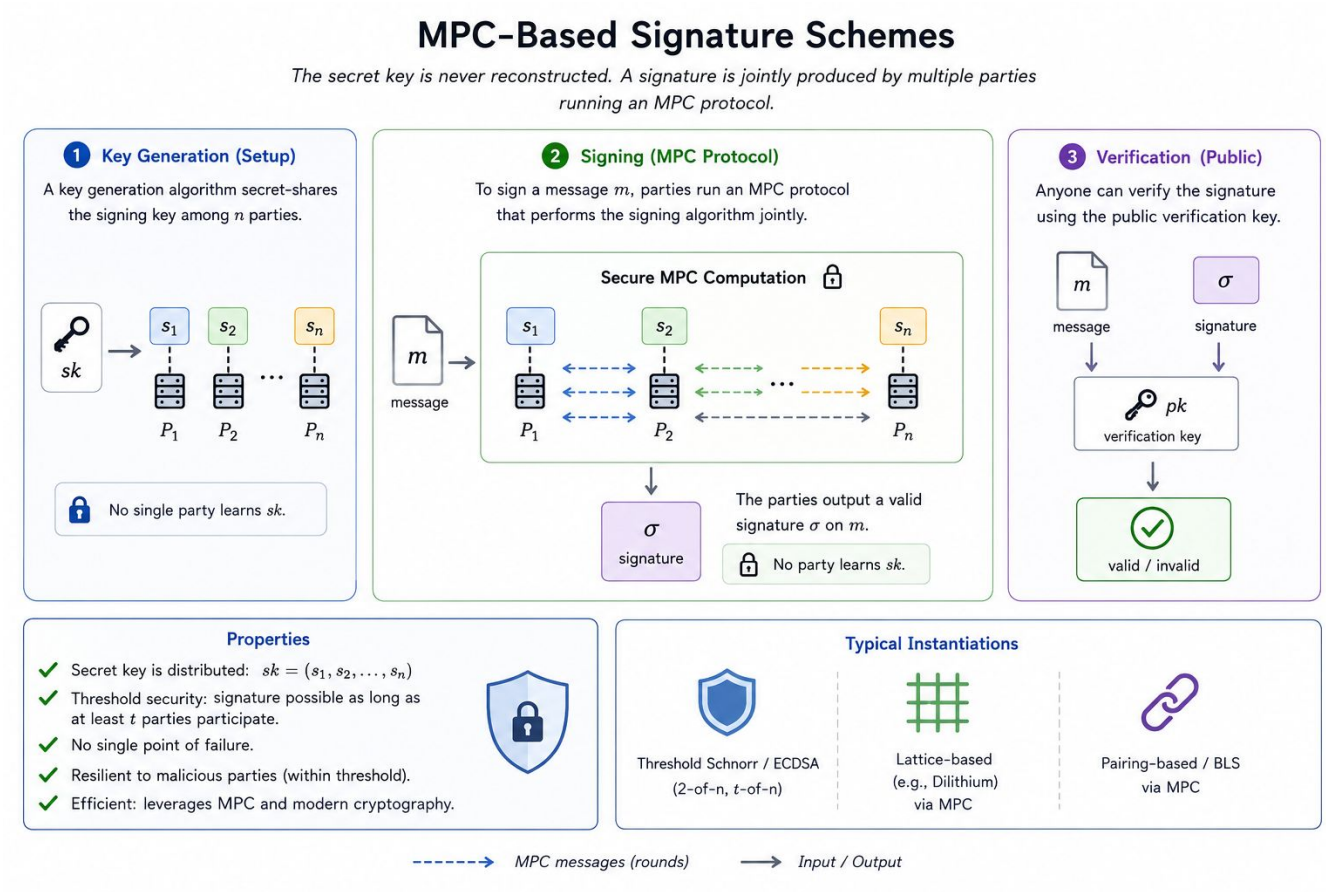
Securely pass your crypto assets to your family

GUARDIANS

Extra layer of security with trusted counterparties

Bron-Crypto

Mainly focusing on MPC-based signing and the related machinery.



This Talk

Doing cryptography engineering nowadays is:

01 Tricky

02 Changed a lot in the past decade

03 Requires a different mentality than the Academia

This talk has four parts:

01 Reusability and abstraction, as a guiding principle.

02 Challenges of porting papers to production

03 Example of productionizing a protocol: MSP-based signing

04 Future work: On Specifications

We will describe real life issues, lessons learnt, and the design decisions behind our open-source cryptography library: https://github.com/bronlabs/bron-crypto

1
Reusability

A Distinct Lack of Abstraction

Reusability: The case for abstraction

*“**Hard**** Real-world catastrophic bugs” in applied cryptography are typically **NOT** a consequence of poor usage of a programming language eg. nil dereference, memory leaks etc.

*“**Hard ** Real-world catastrophic bugs” in applied cryptography are mostly **STRUCTURAL** and **ALGORITHMS** related.

CVE-2022-21449: Psychic Signatures in Java

Paper 2016/771

How not to Prove Yourself: Pitfalls of the Fiat-Shamir Heuristic and Applications to Helios

David Bernhard, Olivier Pereira, and Bogdan Warinschi

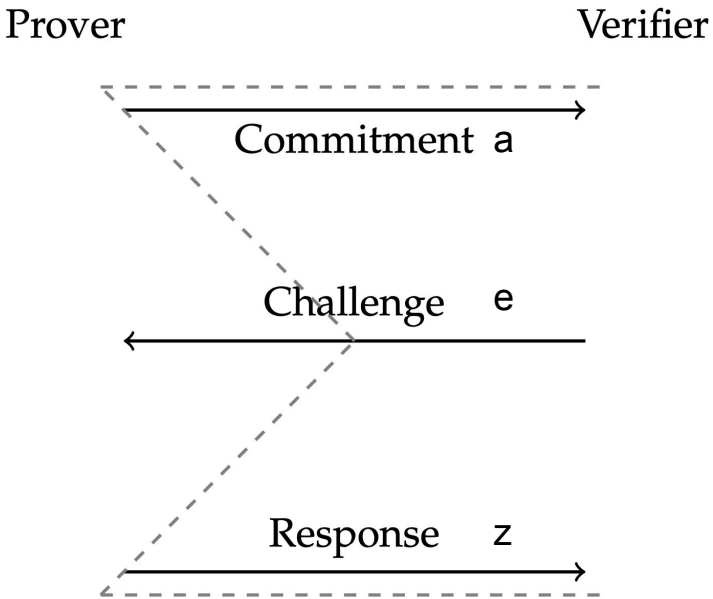
Memory leaks, nil dereference and alike are easier to catch than structural/algorithmic issues.

Abstractions helps with minimizing structural/algorithmic issues by limiting them to a single reference point.

Cryptography is often written in a self-contained way, to be written once, audited once, then deployed and never to be touched again. This is no longer a sustainable strategy.

Reusability: Background

What is a sigma protocol?



Reusability: Background

What is Fiat-Shamir?

The Fiat–Shamir transformation is a fundamental cryptographic technique widely used to convert public-coin interactive protocols into non-interactive ones.

It effectively reads challenge from the transcript.

challenge e=H(x || a)
challenge e=H(a) (**WEAK** Fiat-Shamir)

Reusability: Sigma protocol Interface

Stateless interfaces

```
type Protocol[X Statement, W Witness, A Commitment, S State, Z Response] interface {
    Name() Name
    ComputeProverCommitment(statement X, witness W) (A, S, error)
    ComputeProverResponse(statement X, witness W, commitment A, state S, challenge ChallengeBytes) (Z, error)
    Verify(statement X, commitment A, challenge ChallengeBytes, response Z) error

    // RunSimulator produces a transcript that's statistically identical to (or indistinguishable from)
    // the output of the real Prover.
    // In the context of Honest-Verifier Zero-Knowledge Proofs of Knowledge,
    // the simulator is an algorithm that is able to fake a commitment and a convincing proof
    // without the knowledge of the witness.
    // To fake it, the simulator "rewinds" (aka does things in reverse order):
    // first create a response, and then compute the commitment intelligently so that the full transcript (a,
    // would be valid if played in the right order.
    RunSimulator(statement X, challenge ChallengeBytes) (A, Z, error)

    // SpecialSoundness returns n for which protocol has n-special soundness.
    // In other words, it returns a minimal number of how many distinct, valid
    // sigma protocol transcripts T_i = (x, e_i, z_i) for i = 1, 2, ..., n
    // are required for the existence of polynomial-time extractor of witness.
    SpecialSoundness() uint

    // SoundnessError returns the statistical soundness error `s` of the protocol,
    // i.e., the probability that a cheating prover can succeed is ≤ 2^(-s).
    // For interactive proofs it must be at least base.StatisticalSecurity,
    // for non-interactive proofs it must be at least base.ComputationalSecurity.
    SoundnessError() uint
    GetChallengeBytesLength() int

    ValidateStatement(statement X, witness W) error
}
```

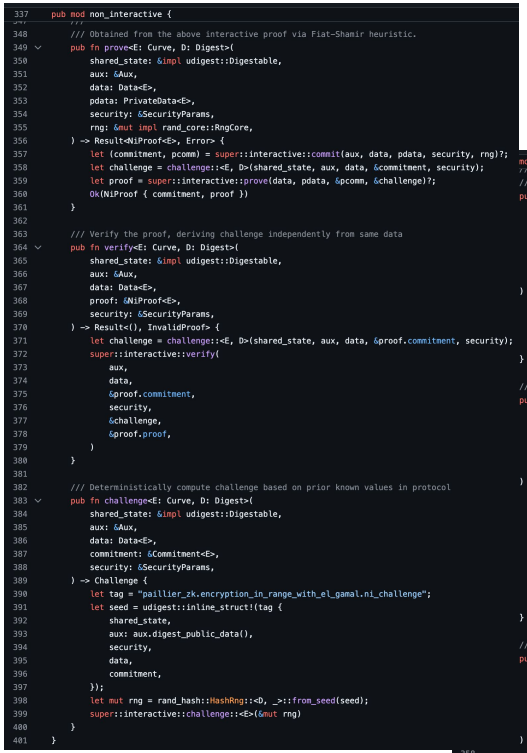
```
type MaurerProtocol[
    I algebra.GroupElement[I],
    P algebra.GroupElement[P],
    X MaurerStatement[I],
    W MaurerWitness[P],
    A MaurerCommitment[I],
    S MaurerState[P],
    Z MaurerResponse[P],
] interface {
    Protocol[X, W, A, S, Z]

    PreImageGroup() FiniteGroup[P]
    ImageGroup() FiniteGroup[I]
    Phi() OneWayHomomorphism[I, P]
    Anchor() Anchor[I, P]
}
```

```
type NonInteractiveProtocol[X sigma.Statement, W sigma.Witness] interface {
    Name() Name
    SigmaProtocolName() sigma.Name
    NewProver(sessionId network.SID, transcript transcripts.Transcript) (NIProver[X, W], error)
    NewVerifier(sessionId network.SID, transcript transcripts.Transcript) (NIVerifier[X], error)
}
```

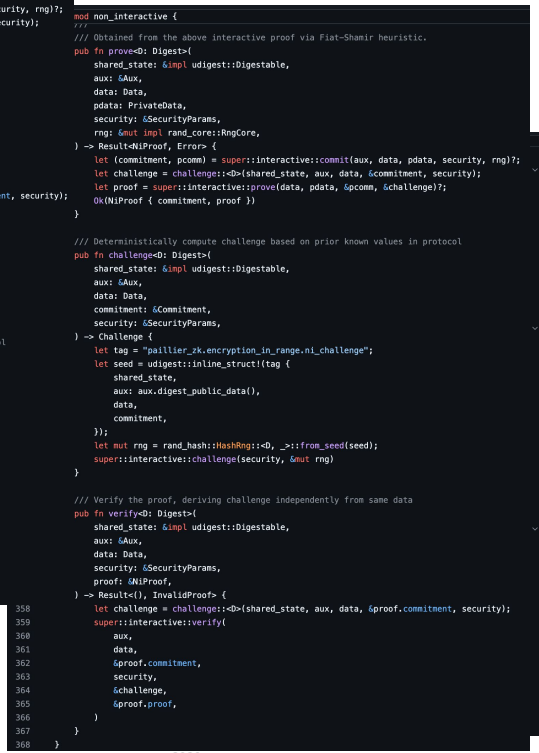
Reusability: Sigma protocol Compilation, Them

Non-Interactive Sigma protocol compilation

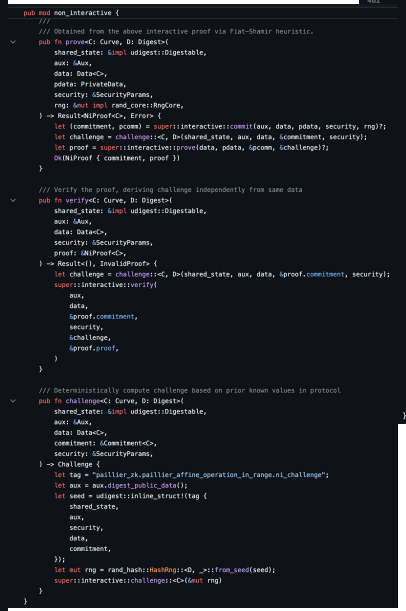
Is it weak-FS or strong-FS?



```rust
pub mod non_interactive {
    /// Obtained from the above interactive proof via Fiat-Shamir heuristic.
    pub fn prove<E: Curve, D: Digest>(
        shared_state: &impl udigest::Digestable,
        aux: &Aux,
        data: Data<E>,
        pdata: PrivateData<E>,
        security: &SecurityParams,
        rng: &mut impl rand_core::RngCore,
    ) -> Result<NiProof<E>, Error> {
        let (commitment, pcomm) = super::interactive::commit(aux, data, pdata, security, rng)?;
        let challenge = challenge::<E, D>(shared_state, aux, data, &commitment, security);
        let proof = super::interactive::prove(data, pdata, &pcomm, &challenge)?;
        Ok(NiProof { commitment, proof })
    }

    /// Verify the proof, deriving challenge independently from same data
    pub fn verify<E: Curve, D: Digest>(
        shared_state: &impl udigest::Digestable,
        aux: &Aux,
        data: Data<E>,
        proof: &NiProof<E>,
        security: &SecurityParams,
    ) -> Result<(), InvalidProof> {
        let challenge = challenge::<E, D>(shared_state, aux, data, &proof.commitment, security);
        super::interactive::verify(
            aux,
            data,
            &proof.commitment,
            security,
            &challenge,
            &proof.proof,
        )
    }

    /// Deterministically compute challenge based on prior known values in protocol
    pub fn challenge<E: Curve, D: Digest>(
        shared_state: &impl udigest::Digestable,
        aux: &Aux,
        data: Data<E>,
        commitment: &Commitment<E>,
        security: &SecurityParams,
    ) -> Challenge {
        let tag = "paillier_zk.encryption_in_range_with_el_gamal.ni_challenge";
        let seed = udigest::inline_struct!(tag {
            shared_state,
            aux: aux.digest_public_data(),
            security,
            data,
            commitment,
        });
        let mut rng = rand_hash::HashRng::<D, _>::from_seed(seed);
        super::interactive::challenge::<E>(&mut rng)
    }
}
```



```rust
mod non_interactive {
    /// Obtained from the above interactive proof via Fiat-Shamir heuristic.
    pub fn prove<D: Digest>(
        shared_state: &impl udigest::Digestable,
        aux: &Aux,
        data: Data,
        pdata: PrivateData,
        security: &SecurityParams,
        rng: &mut impl rand_core::RngCore,
    ) -> Result<NiProof, Error> {
        let (commitment, pcomm) = super::interactive::commit(aux, data, pdata, security, rng)?;
        let challenge = challenge::<D>(shared_state, aux, data, &commitment, security);
        let proof = super::interactive::prove(data, pdata, &pcomm, &challenge)?;
        Ok(NiProof { commitment, proof })
    }

    /// Deterministically compute challenge based on prior known values in protocol
    pub fn challenge<D: Digest>(
        shared_state: &impl udigest::Digestable,
        aux: &Aux,
        data: Data,
        commitment: &Commitment,
        security: &SecurityParams,
    ) -> Challenge {
        let tag = "paillier_zk.encryption_in_range.ni_challenge";
        let seed = udigest::inline_struct!(tag {
            shared_state,
            aux: aux.digest_public_data(),
            data,
            commitment,
        });
        let mut rng = rand_hash::HashRng::<D, _>::from_seed(seed);
        super::interactive::challenge(security, &mut rng)
    }

    /// Verify the proof, deriving challenge independently from same data
    pub fn verify<D: Digest>(
        shared_state: &impl udigest::Digestable,
        aux: &Aux,
        data: Data,
        security: &SecurityParams,
        proof: &NiProof,
    ) -> Result<(), InvalidProof> {
        let challenge = challenge::<D>(shared_state, aux, data, &proof.commitment, security);
        super::interactive::verify(
            aux,
            data,
            &proof.commitment,
            security,
            &challenge,
            &proof.proof,
        )
    }
}
```



```rust
pub mod non_interactive {
    /// Obtained from the above interactive proof via Fiat-Shamir heuristic.
    pub fn prove<C: Curve, D: Digest>(
        shared_state: &impl udigest::Digestable,
        aux: &Aux,
        data: Data<C>,
        pdata: PrivateData,
        security: &SecurityParams,
        rng: &mut impl rand_core::RngCore,
    ) -> Result<NiProof<C>, Error> {
        let (commitment, pcomm) = super::interactive::commit(aux, data, pdata, security, rng)?;
        let challenge = challenge::<C, D>(shared_state, aux, data, &commitment, security);
        let proof = super::interactive::prove(data, pdata, &pcomm, &challenge)?;
        Ok(NiProof { commitment, proof })
    }

    /// Verify the proof, deriving challenge independently from same data
    pub fn verify<C: Curve, D: Digest>(
        shared_state: &impl udigest::Digestable,
        aux: &Aux,
        data: Data<C>,
        proof: &NiProof<C>,
        security: &SecurityParams,
    ) -> Result<(), InvalidProof> {
        let challenge = challenge::<C, D>(shared_state, aux, data, &proof.commitment, security);
        super::interactive::verify(
            aux,
            data,
            &proof.commitment,
            security,
            &challenge,
            &proof.proof,
        )
    }

    /// Deterministically compute challenge based on prior known values in protocol
    pub fn challenge<C: Curve, D: Digest>(
        shared_state: &impl udigest::Digestable,
        aux: &Aux,
        data: Data<C>,
        commitment: &Commitment<C>,
        security: &SecurityParams,
    ) -> Challenge {
        let tag = "paillier_zk.paillier_affine_operation_in_range.ni_challenge";
        let aux = aux.digest_public_data();
        let seed = udigest::inline_struct!(tag {
            shared_state,
            aux,
            security,
            data,
            commitment,
        });
        let mut rng = rand_hash::HashRng::<D, _>::from_seed(seed);
        super::interactive::challenge::<C>(&mut rng)
    }
}
```

Reusability: Sigma protocol Compilation, Us

Compilation, without redefinition

```
func SchnorrTestHelper[
    P curves.Point[P, F, S],
    F algebra.FiniteFieldElement[F],
    S algebra.PrimeFieldElement[S],
](t testing.TB, curve curves.Curve[P, F, S], w S, x P) {
    witness := schnorr.NewWitness(w)
    statement := schnorr.NewStatement(x)
    basePoint := curve.Generator()

    tape := hagrid.NewTranscript("testing")
    sessionId, err := network.NewSID([]byte("something random and unique"))
    require.NoError(t, err)

    protocol, err := schnorr.NewSigmaProtocol(basePoint, rand.Reader)
    require.NoError(t, err)

    nonInteractiveProtocol, err := rfischlin.NewCompiler(protocol)
    require.NoError(t, err)

    prover, err := nonInteractiveProtocol.NewProver(sessionId, tape)
    require.NoError(t, err)

    proof, err := prover.Prove(statement, witness)
    require.NoError(t, err)

    verifier, err := nonInteractiveProtocol.NewVerifier(sessionId, tape)
    require.NoError(t, err)

    err = verifier.Verify(statement, proof)
    require.NoError(t, err)
}
```

Reusability: Background

Abstractions goes much further.

A sigma protocol for “one-way Group homomorphisms”.

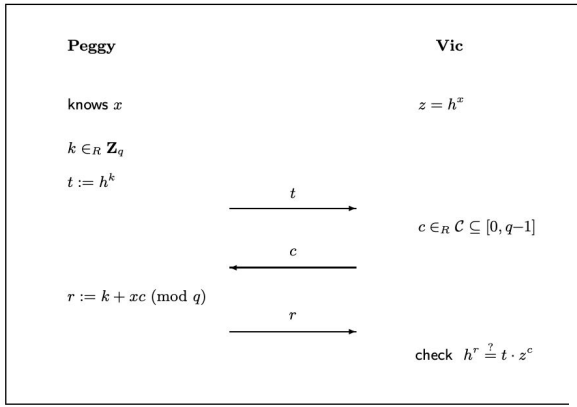


Fig. 1. The Schnorr protocol

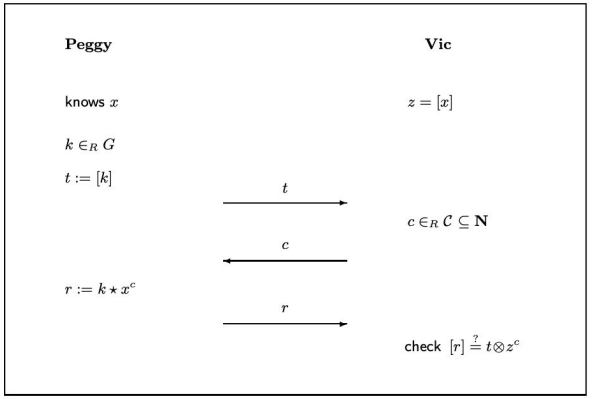


Fig. 3. Main protocol: Proof of knowledge, for a given value z, of a value x such that $z = [x]$, where $x \mapsto [x]$ is a (one-way) group homomorphism

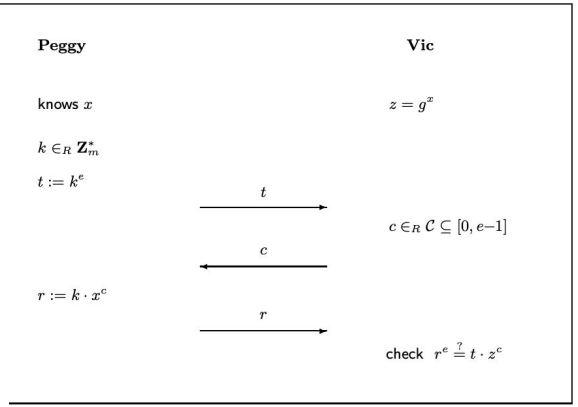


Fig. 2. The Guillou-Quisquater (GQ) protocol

Reusability: Maurer Proofs

1. Schnorr Discrete Logarithm proof
2. PoK of Opening of ElGamal Commitments
3. Okamoto's PoK of Representation
4. PoK of N'th root mod N^2

Single reference points:

1. Is the simulate function implemented correctly?
2. Are challenge bytes read correctly?
3. etc.

```
// NewProtocol creates a new Schnorr protocol instance with the given generator and randomness source.
func NewProtocol[G algebra.PrimeGroupElement[G, S], S algebra.PrimeFieldElement[S]](generator G, prng io.Reader) (*Protocol[G, S], error) {
    group := algebra.StructureMustBeAs[algebra.PrimeGroup[G, S]](generator.Structure())
    scalarField := algebra.StructureMustBeAs[algebra.PrimeField[S]](group.ScalarStructure())
    challengeByteLen := base.ComputationalSecurityBytesCeil // To make it non interactive, it has to be at least equal to computational security parameter.
    soundnessError := uint(challengeByteLen * 8)
    homomorphism := func(s S) (G, error) {
        if utils.IsNil(s) {
            return *new(G), proofs.ErrInvalidArgument.WithMessage("homomorphism input cannot be nil")
        }
        return generator.ScalarOp(s), nil
    }
    l, err := num.N().FromBytes(group.Order().Bytes())
    if err != nil {
        return nil, errs.Wrap(err).WithMessage("cannot create anchor")
    }
    anc := &anchor[G, S]{l, scalarField.Zero()}

    maurerProtocol, err := maurer09.NewProtocol(
        challengeByteLen,
        soundnessError,
        Name,
        group,
        scalarField,
        homomorphism,
        anc,
        prng,
    )
    if err != nil {
        return nil, errs.Wrap(err)
    }

    return &Protocol[G, S]{*maurerProtocol}, ni
}
```

```
homomorphism := func(w *constructions.FiniteDirectProductGroupElement[G, S]) (*constructions.FiniteDirectPowerModuleElement[G, S], error) {
    messageValue, nonceValue := w.Components()
    nonce, err := elgamal.NewNonce(nonceValue)
    if err != nil {
        return nil, errs.Wrap(err).WithMessage("failed to create nonce from homomorphism input")
    }
    witness, err := indcpacom.NewWitness(nonce)
    if err != nil {
        return nil, errs.Wrap(err).WithMessage("failed to create commitment witness from homomorphism input")
    }
    plaintext, err := elgamal.NewPlaintext(messageValue)
    if err != nil {
        return nil, errs.Wrap(err).WithMessage("failed to create plaintext from homomorphism input")
    }
    message, err := indcpacom.NewMessage(plaintext)
    if err != nil {
        return nil, errs.Wrap(err).WithMessage("failed to create commitment message from homomorphism input")
    }
    commitment, err := key.CommitWithWitness(message, witness)
    if err != nil {
        return nil, errs.Wrap(err).WithMessage("failed to compute commitment from homomorphism input")
    }
```

```
homomorphism := func(s *constructions.FiniteDirectPowerRingElement[S]) (G, error) {
    if s == nil {
        return *new(G), proofs.ErrInvalidArgument.WithMessage("homomorphism input cannot be nil")
    }
    return generatorsVector.ScalarDiagonal(
}
```

```
oneWayHomomorphism := func(x *znstar.PaillierGroupElement[A]) (*znstar.PaillierGroupElement[A], error) {
    if x == nil {
        return nil, proofs.ErrInvalidArgument.WithMessage("homomorphism input cannot be nil")
    }
    y, err := group.NthResidue(x)
    if err != nil {
        return nil, errs.Wrap(err).WithMessage("cannot compute nth residue")
    }
    return y, nil
}
```

Reusability: Yet another example

In-lined CRT

```rust
pub fn rsa_decrypt<R: TryCryptoRng + ?Sized>(

            let p = &priv_key.primes()[0];
            let q = &priv_key.primes()[1];

            // precomputed: dP = (1/e) mod (p-1) = d mod (p-1)
            // precomputed: dQ = (1/e) mod (q-1) = d mod (q-1)

            // TODO: it may be faster to convert to and from Montgomery with prepared parameters
            // (modulo `p` and `q`) rather than calculating the remainder directly.

            // m1 = c^dP mod p
            let p_wide = p_params.modulus().resize_unchecked(c.bits_precision());
            let c_mod_dp = (&c % p_wide.as_nz_ref()).resize_unchecked(dp.bits_precision());
            let cp = BoxedMontyForm::new(c_mod_dp, p_params.clone());
            let mut m1 = cp.pow(dp);
            // m2 = c^dQ mod q
            let q_wide = q_params.modulus().resize_unchecked(c.bits_precision());
            let c_mod_dq = (&c % q_wide.as_nz_ref()).resize_unchecked(dq.bits_precision());
            let cq = BoxedMontyForm::new(c_mod_dq, q_params.clone());
            let m2 = cq.pow(dq).retrieve();

            // Note that since `p` and `q` may have different `bits_precision`,
            // it may be different for `m1` and `m2` as well.

            // (m1 - m2) mod p = (m1 mod p) - (m2 mod p) mod p
            let m2_mod_p = match p_params.bits_precision().cmp(&q_params.bits_precision()) {
                Ordering::Less => {
                    let p_wide = NonZero::new(p.clone())
                        .expect("`p` is non-zero")
                        .resize_unchecked(q_params.bits_precision());
                    (&m2 % p_wide).resize_unchecked(p_params.bits_precision())
                }
                Ordering::Greater => (&m2).resize_unchecked(p_params.bits_precision()),
                Ordering::Equal => m2.clone(),
            };
            let m2r = BoxedMontyForm::new(m2_mod_p, p_params.clone());
            m1 -= &m2r;

            // precomputed: qInv = (1/q) mod p

            // h = qInv.(m1 - m2) mod p
            let h = (qinv * m1).retrieve();

            // m = m2 + h.q
            let m2 = m2.try_resize(n.bits_precision()).ok_or(Error::Internal)?;
            let hq = (h * q)
                .try_resize(n.bits_precision())
                .ok_or(Error::Internal)?;
            m2.wrapping_add(&hq)
        }
        _ => {
            // c^d (mod n)
            pow_mod_params(&c, d, n_params)
        }
    };
```

Reusability: Yet another example

Reusable and efficient CRT

```go
func (m *OddPrimeSquareFactors) ModExp(out, base, exp *numct.Nat) {
    // Compute base^ep mod p and base^eq mod q in parallel.
    var mp, mq numct.Nat
    var wg sync.WaitGroup
    wg.Add(2)
    fmt.Println(m.Q)
    go func() {
        defer wg.Done()
        var ep numct.Nat
        m.P.PhiFactor.Mod(&ep, exp)
        ep.Select(base.Coprime(m.P.Factor.Nat()), exp, &ep)
        (m.CrtModN2.P).ModExp(&mp, base, &ep)
    }()
    go func() {
        defer wg.Done()
        var eq numct.Nat
        m.Q.PhiFactor.Mod(&eq, exp)
        eq.Select(base.Coprime(m.Q.Factor.Nat()), exp, &eq)
        m.CrtModN2.Q.ModExp(&mq, base, &eq)
    }()
    wg.Wait()

    // CRT recombine into modulo n = p*q.
    .Set(m.CrtModN2.Recombine(&mp, &mq))
```

```go
// EncryptWithNonce encrypts under the embedded public parameters; identical in
// result to PublicKey.EncryptWithNonce, with the same fresh-secret-nonce
// requirement.
func (sk *SecretKey) EncryptWithNonce(p *Plaintext, n *Nonce) (*Ciphertext, error) {
    if p == nil || n == nil {
        return nil, encryption.ErrIsNil.WithMessage("plaintext and nonce must not be nil")
    }
    if !sk.PlaintextGroup().Contains(p.Value()) {
        return nil, encryption.ErrSubGroupMembership.WithMessage("plaintext must be in plaintext group")
    }
    if !sk.NonceGroup().Contains(n.Value()) {
        return nil, encryption.ErrSubGroupMembership.WithMessage("nonce must be in nonce group")
    }
    out, err := gift.Encrypt(sk, p, n)
    if err != nil {
        return nil, errs.Wrap(err).WithMessage("could not encrypt message with nonce")
    }
    return out, nil
}
```

```go
func (m *OddPrimeSquareFactors) ExpToN(out, a *numct.Nat) {
    // Direct CRT: y_p = a^{Ep2} mod p^2, y_q = a^{Eq2} mod q^2
    var yp, yq numct.Nat
    var wg sync.WaitGroup
    wg.Add(2)
    go func() {
        defer wg.Done()
        m.P.Squared.ModExp(&yp, a, m.NExpP2) //  Ep2 = p * (N mod (p-1))

    }()
    go func() {
        defer wg.Done()      Alireza Rafiei, 3 weeks ago • [WIP] Ints + Paillier (
        m.Q.Squared.ModExp(&yq, a, m.NExpQ2) // Eq2 = q * (N mod (q-1))
    }()
    wg.Wait()

    // One-multiply CRT using precomputed (q^2)^{-1} mod p^2 inside m.CrtModN2
    out.Set(m.CrtModN2.Recombine(&yp, &yq))
}
```

2
Challenges and Tips

Pitfalls of implementing papers

Challenge 1: The Folklore

Many important protocols either don’t have a paper, or aren’t constructed anywhere.

On Span Programs

M. Karchmer*

Department of Mathematics
Massachusetts Inst. of Technology
Cambridge, MA 02138

A. Wigderson

Department of Computer Science
Hebrew University
Jerusalem, Israel 91904

Secure Schemes for
Secret Sharing and Key Distribution

Research Thesis

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Science

Amos Beimel

Submitted to the Senate of the Technion – Israel Institute of Technology
Sivan, 5756 Haifa June 1996

Challenge 1: The Folklore

What to do?

Image from https://www.youtube.com/watch?v=9alhSquF2-w

Challenge 2: Leaky Boundaries

Black-boxes are often leaky: ***Schnorr != EdDSA***

A note on EdDSA. The EdDSA signing scheme [BDL+12] is a Schnorr variant where the randomness used to generate the signature is derived deterministically from the key, using a pseudorandom function. This design is intended to prevent biased and reused nonces which are catastrophic in Schnorr (and ECDSA) signatures. Our protocol can be used for EdDSA, but the result is not actually compliant with the standard since it is possible to generate two different signatures on the same message since it is not deterministic. As long as only one signature is generated, this is indistinguishable. However, if a higher-level protocol relies on the signature being deterministic, then this is a problem. (Having said that, if a higher-level protocol relies on it being deterministic, it by generating two different signatures, so higher-level prot

7.7 EdDSA Signature Verification

Inputs:

1. Message M
2. Signature $R \| S$ where R and S are octet strings
3. Purported signature verification key Q that is valid for domain parameters D
4. For Ed448, a string *context* set by the signer and verifier with a maximum length of 255 octets; by default, *context* is the empty string

Output: Accept or reject the signature over M as originating from the owner of public key Q.

Process:

1. Decode the first half of the signature as a point R and the second half of the signature as an integer t. Verify that the integer t is in the range of $0 \leq t < n$. Decode the public key Q into a point Q'. If any of the decodings fail, output "reject".
2. Using the established hash function or XOF,

   2.1 For Ed25519, compute $digest = SHA\text{-}512(R \| Q \| M)$.
   2.2 For Ed448, compute $digest = SHAKE256(\text{dom4}(0, context) \| R \| Q \| M, 912)$

   Interpret *digest* as a little-endian integer u.
3. Check that the verification equation $[2^c t]G = [2^c]R + [2^c u]Q'$ holds. It's sufficient, but not required, to instead check $[t]G = R + [u]Q'$. Output "reject" if verification fails; output "accept" otherwise.

Baby Sharks

Injecting small order points to threshold EdDSA

Omer Shlomovits Follow 6 min read · Nov 13, 2020

Taming the many EdDSAs

Konstantinos Chalkias, François Garillot, and Valeria Nikolaenko

Novi/Facebook
kostascrypto@fb.com, fga@fb.com, valerini@fb.com

Abstract. This paper analyses security of concrete instantiations of EdDSA by identifying exploitable inconsistencies between standardization recommendations and Ed25519 implementations. We mainly focus on current ambiguity regarding signature verification equations, binding and malleability guarantees, and incompatibilities between randomized batch and single verification. We give a formulation of Ed25519 signature scheme that achieves the highest level of security, explaining how each step of the algorithm links with the formal security properties. We develop optimizations to allow for more efficient secure implementations. Finally, we designed a set of edge-case test-vectors and run them by some of the most popular Ed25519 libraries. The results allowed to understand the security level of those implementations and showed that most libraries do not comply with the latest standardization recommendations. The methodology allows to test compatibility of different Ed25519 implementations which is of practical importance for consensus-driven applications.

Keywords: EdDSA · ed25519 · malleability · blockchain · cofactor

What to do?

1. Resist the temptation of "black-box"ing a protocol, even if it's supposed to be a black-box.
2. Read original literature. This is not just for culture!

Challenge 3: Concreteness

Papers are often too “concrete” for your abstractions.

FIGURE 23 (Dlog with El-Gamal Commitment—Π^{elog})

- **Inputs:** Common input is $(\mathbb{G}, g, L, M, X, Y, h)$.
  The Prover has secret input (y, λ) such that $L = g^{\lambda}$, $M = g^{y}X^{\lambda}$ and $Y = h^{y}$.

1. Prover samples
$$\alpha, m \leftarrow \mathbb{F}_q \text{ and computes } \begin{cases} A = g^{\alpha}, N = g^{m}X^{\alpha} \\ B = h^{m} \end{cases}$$
and sends (A, N, B) to the verifier.
2. Verifier replies with $e \leftarrow \pm q$.
3. Prover Prover sends (z, u) to the verifier where $z = \alpha + e\lambda \mod q$ and $u = m + ey \mod q$.

- **Equality Checks**: $g^{z} = A \cdot L^{e}$ and $g^{u}X^{z} = N \cdot M^{e}$ and $h^{u} = B \cdot Y^{e} \in \mathbb{G}$.

Figure 23: Dlog with El-Gamal Commitment—Π^{elog}

IND-CPA Commitments (commitment from encryption)

A commitment to a message `m` with witness `r` is the ciphertext

```
C = Enc_ek(m; r)
```

Proof of knowledge of opening of ElGamal Commitment

This package implements a sigma protocol for proving knowledge of an opening of an ElGa homomorphism based framework.

An ElGamal commitment to a message $M \in G$ with nonce $\lambda \in F_q$ under public key $X \in G$ is t

$$(\Gamma, \Delta) = (g^{\lambda}, ; M \cdot X^{\lambda}).$$

Th

Dlog with ElGamal Commitment

This package implements Figure 23 of CGGMP21, titled Dlog with Elgamal Commitments. It is a sigma protocol for the following NP-relation:

$$R_{\text{elog}} = \left\{ ((G, g, L, M, X, Y, h), (y, \lambda)) \mid L = g^{\lambda},\ M = g^{y}X^{\lambda},\ Y = h^{y} \right\}.$$

Maurer09 Sigma Protocol

Generic sigma protocol implementation based on Maurer's unifying framework for zero-knowledge proofs of knowledge [1].

GIFT (Generic shIFt-Type) group-homomorphic encryption

This internal package implements some methods used in implementations of various group homomorphic encryption schemes.

Generalised ElGamal Cryptosystem

This package implements the Generalized ElGamal cryptosystem over any finite abelian cyclic group G, following section 8.4 of the Handbook of Applied Cryptography (Menezes, van Oorschot, Vanstone). Internally, we adapt the implementation to match ElGamal instantiation of GIFT.

Challenge 3: Concreteness

2.4 Pedersen Commitments

We utilize *Pedersen* commitments. A Pedersen commitment, of a value $m \in \mathbb{Z}_q$ with public key (parameter) $D \in \mathbb{G}$ and randomness $r \in \mathbb{Z}_q$, is defined by

$$\mathsf{PedCom}_D(m; r) = r \cdot D + m \cdot G,$$

and $\mathsf{PedCom}_D(m)$ stands for $\mathsf{PedCom}_D(m; r)$, where $r \xleftarrow{R} \mathbb{Z}_q$. We typically denote elements of $\mathbb{G}$ that are outputs of PedCom using the $\widehat{\ }$ symbol, e.g., $\widehat{A} = \mathsf{PedCom}_D(m)$.

Fact 2.7 ([Ped91]). PedCom *is perfectly hiding. Assuming discrete log is hard over* $\mathbb{G}$*, it is computationally binding.*

We recall three secret sharing protocols which we use throughout the paper.

Shamir's Secret Sharing

In Shamir's secret sharing protocol [S2] a dealer shares a secret $\sigma \in Z_q$ among the parties $P_1, \ldots, P_n$ in the following way. The dealer chooses at random a polynomial $f(z)$ over Z_q of degree t, such that $f(0) = \sigma$. He then secretly transmits to each party P_i a share $s_i = f(i) \bmod q$. It is clear that t or less parties have no information about the secret while $t + 1$ can easily reconstruct it by polynomial interpolation.

It is well known however that in the presence of a malicious adversary, Shamir's secret sharing protocol could fail. Indeed, it is possible for a dealer to share values which do not lie on a polynomial of degree t. Also dishonest parties may contribute incorrect shares at reconstruction time. *Verifiable secret sharing* (*VSS*) protocols [CGMA], described next, are intended to prevent this possibility.

Shamir Secret Sharing. Many threshold schemes build upon Shamir secret sharing [28], a (t, n)-threshold scheme that relies on Lagrange interpolation to recover a secret. In Shamir secret sharing, a trusted central dealer distributes a secret s to n participants in such a way that any cooperating subset of t participants can recover the secret. To distribute this secret, the dealer first selects $t-1$ coefficients $a_1, \ldots, a_{t-1}$ at random, and uses the randomly selected values as coefficients to define a polynomial $f(x) = s + \sum_{i=1}^{t-1} a_i x^i$ of degree $t-1$ where $f(0) = s$. The secret shares for each participant P_i are subsequently $(i, f(i))$, which the dealer is trusted to distribute honestly to each participant $P_1, \ldots, P_n$. To reconstruct the secret, at least t participants perform Lagrange interpolation to reconstruct the polynomial and thus find the value $s = f(0)$.

What to do? <As a researcher>

Resist the temptation of writing "self-contained" Papers. Modularize. Abstract.

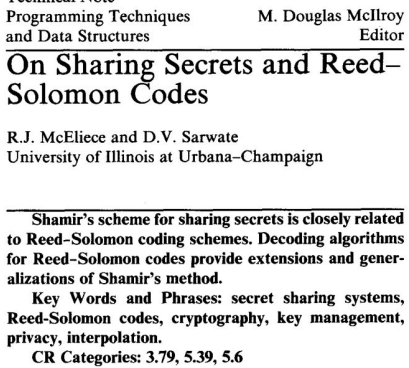
Challenge 3: Concreteness

What to do? <As a practitioner>

Image from https://www.youtube.com/watch?v=lNUYTUGMMxI

Challenge 4: Abstracting the right way

It’s not always obvious what to abstract and how.



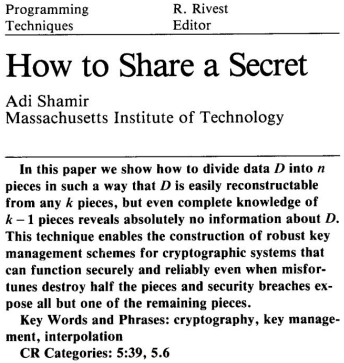
Technical Note
Programming Techniques and Data Structures — M. Douglas McIlroy Editor

On Sharing Secrets and Reed–Solomon Codes

R.J. McEliece and D.V. Sarwate
University of Illinois at Urbana–Champaign

Shamir's scheme for sharing secrets is closely related to Reed–Solomon coding schemes. Decoding algorithms for Reed–Solomon codes provide extensions and generalizations of Shamir's method.

Key Words and Phrases: secret sharing systems, Reed-Solomon codes, cryptography, key management, privacy, interpolation.
CR Categories: 3.79, 5.39, 5.6



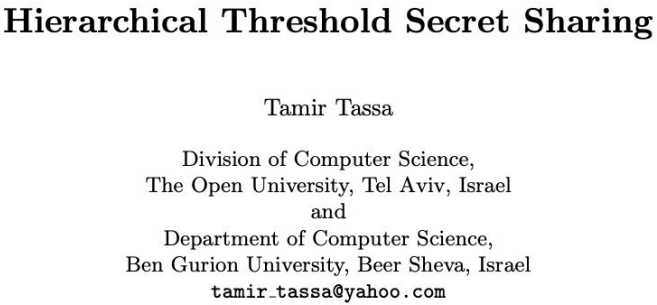
Programming Techniques — R. Rivest Editor

How to Share a Secret

Adi Shamir
Massachusetts Institute of Technology

In this paper we show how to divide data D into n pieces in such a way that D is easily reconstructable from any k pieces, but even complete knowledge of $k-1$ pieces reveals absolutely no information about D. This technique enables the construction of robust key management schemes for cryptographic systems that can function securely and reliably even when misfortunes destroy half the pieces and security breaches expose all but one of the remaining pieces.

Key Words and Phrases: cryptography, key management, interpolation
CR Categories: 5:39, 5.6



Hierarchical Threshold Secret Sharing

Tamir Tassa

Division of Computer Science,
The Open University, Tel Aviv, Israel
and
Department of Computer Science,
Ben Gurion University, Beer Sheva, Israel
tamir_tassa@yahoo.com

Challenge 4: Abstracting the right way

What to do?

Gain experience

Challenge 5: Real vs. Ideal

Your ideal view as a researcher may not be compatible with the real world.

What to do? Exercise your risk management muscles. This is not a purely cryptographic skill. Alternatively, try telling your boss you don’t want to support BLS or some ZK system, because no UC security or AGM and alike and watch natural selection do its magic.

Challenge 6: Provable Security

Cryptography literature is littered with redacted/modified proofs of security. What good is a proof if you can't trust it?

The fundamental goal of
"provable security"

D. J. Bernstein

University of Illinois at Chicago &
Technische Universiteit Eindhoven

CRITICAL PERSPECTIVES ON PROVABLE SECURITY: FIFTEEN YEARS OF "ANOTHER LOOK" PAPERS

NEAL KOBLITZ AND ALFRED MENEZES

ABSTRACT. We give an overview of our critiques of "proofs" of security and a guide to our papers on the subject that have appeared over the past decade and a half. We also provide numerous additional examples and a few updates and errata.

Challenge 6: Provable Security, Recent Example

You are looking at a specific version 20221012:012203 of this paper. See the latest version.

Paper 2022/1371

On the Security of KOS

Benjamin E. Diamond, Coinbase

Abstract

We present a full proof of security of the original random oblivious transfer extension protocol of Keller, Orsini, and Scholl (CRYPTO '15), without altering that protocol as written. Our result circumvents a recent negative result of Roy (CRYPTO '22), which shows that a key lemma in the original proof of KOS is false. Our proof leverages a new simulation strategy, and a careful analysis of that protocol's "correlation check". We thus reestablish evidence of security for this important, widely used protocol.

Paper 2022/1371

On the Security of KOS

Benjamin E. Diamond, Irreducible

Abstract

We study the security of the random oblivious transfer extension protocol of Keller, Orsini, and Scholl (CRYPTO '15), whose security proof was recently invalidated by Roy (CRYPTO '22). We show that KOS is asymptotically secure. Our proof involves a subtle analysis of the protocol's "correlation check", and introduces several new techniques. We also study the protocol's concrete security. We establish concrete security for security parameter values on the order of 5,000. We present evidence that a stronger result than ours—if possible—is likely to require radically new ideas.

Note: Various minor fixes and improvements.

Paper 2022/192

SoftSpokenOT: Communication--Computation Tradeoffs in OT Extension

Lawrence Roy

Abstract

Given a small number of base oblivious transfers (OTs), how does one generate a large number of extended OTs as efficiently as possible? The answer has long been the seminal work of IKNP (Ishai et al., Crypto 2003) and the family of protocols it inspired, which only use Minicrypt assumptions. Recently, Boyle et al. (Crypto 2019) proposed the Silent-OT technique that improves on IKNP, but at the cost of a much stronger, non-Minicrypt assumption: the learning parity with noise (LPN) assumption. We present SoftSpokenOT, the first OT extension to improve on IKNP's communication cost in the Minicrypt model. While IKNP requires security parameter λ bits of communication for each OT, SoftSpokenOT only needs λ/k bits, for any k, at the expense of requiring $2^{k-1}/k$ times the computation. For small values of k, this tradeoff is favorable since IKNP-style protocols are network-bound. We implemented SoftSpokenOT and found that our protocol gives almost a $5\times$ speedup over IKNP in the LAN setting.

Our technique is based on a novel silent protocol for vector oblivious linear evaluation (VOLE) over polynomial-sized fields. We created a framework to build maliciously secure 1-of-N OT extension from this VOLE, revisiting the existing work for each step. Along the way, we found several flaws in the existing work, including a practical attack against the consistency check of Patra et al. (NDSS 2017), while also making some improvements.

Challenge 6: Provable Security

What to do?

DO NOT ASSUME IF A PAPER IS PUBLISHED, ITS PROOF IS CORRECT.

Prefer proof formalism whose artifacts prevent yet-unknown attacks.

Paper 2024/031

Feldman's Verifiable Secret Sharing for a Dishonest Majority

Yi-Hsiu Chen, Coinbase
Yehuda Lindell, Coinbase

Abstract

Verifiable secret sharing (VSS) protocols enable parties to share secrets while guaranteeing security (in particular, that all parties hold valid and consistent shares) even if the dealer or some of the participants are malicious. Most work on VSS focuses on the honest majority case, primarily since it enables one to guarantee output delivery (e.g., a corrupted recipient cannot prevent an honest dealer from sharing their value). Feldman's VSS is a well known and popular protocol for this task and relies on the discrete log hardness assumption. In this paper, we present a variant of Feldman's VSS for the dishonest majority setting and formally prove its security. Beyond the basic VSS protocol, we present a publicly-verifiable version, as well as show how to securely add participants to the sharing and how to refresh an existing sharing (all secure in the presence of a dishonest majority). We prove that our protocols are UC secure, for appropriately defined ideal functionalities.

To generate long-lived key shares in our scheme's key generation protocol, FROST builds upon Pedersen's DKG for key generation; we detail these protocol steps in Figure 1. Note that Pedersen's DKG is simply where each participant executes Feldman's VSS as the dealer in parallel, and derives their secret share as the sum of the shares received from each of the n VSS executions. In addition to the base Pedersen DKG protocol, FROST additionally requires each participant to demonstrate knowledge of their secret a_{i0} by providing other participants with proof in zero knowledge, instantiated as a Schnorr signature, to protect against rogue-key attacks [2] in the setting where $t \geq n/2$.

Know how the formalism is used.

Although we prove security in the stand-alone model that guarantees security under sequential composition only, we are really interested in UC security [Can01]; i.e., security under concurrent general composition. This is achieved by all our protocols, since they are all *perfectly secure* with *straight-line simulation* (i.e., no rewinding). As shown in [KLR10], this implies UC security. We remark that the actual VSS protocol is only perfectly secure in the ideal zero-knowledge hybrid model. However, this suffices since it means that as long as the zero-knowledge proof of knowledge is instantiated with a UC-secure protocol, then everything is UC secure.

Paper 2014/553

A Simpler Variant of Universally Composable Security for Standard Multiparty Computation

Ran Canetti, Asaf Cohen, and Yehuda Lindell

Abstract

In this paper, we present a simpler and more restricted variant of the universally composable security (UC) framework that is suitable for ``standard'' two-party and multiparty computation tasks. Many of the complications of the UC framework exist in order to enable more general tasks than classic secure computation. This generality may be a barrier to entry for those who are used to the stand-alone model of secure computation and wish to work with universally composable security but are overwhelmed by the differences. The variant presented here (called simplified universally composable security, or just SUC) is closer to the definition of security for multiparty computation in the stand-alone setting. The main difference is that a protocol in the SUC framework runs with a \emph{fixed set of parties} who know each other's identities ahead of time, and machines \emph{cannot be added dynamically} to the execution. As a result, the definitions of polynomial time and protocol composition are much simpler. In addition, the SUC framework has authenticated channels built in, as is standard in previous definitions of security, and all communication is done via the adversary in order to enable arbitrary scheduling of messages. Due to these differences, not all cryptographic tasks can be expressed in the SUC framework. Nevertheless, standard secure computation tasks (like secure function evaluation) can be expressed. Importantly, we show a natural security-preserving transformation from protocols in the SUC model to protocols in the full-fledged UC model. Consequently, the UC composition theorem holds in the SUC model, and any protocol that is proven secure under SUC can be transformed to a protocol that is secure in the UC model.

Paper 2025/934

Diving Deep Into UC: Uncovering and Resolving Issues in Universal Composability

Céline Chevalier, CRED, Paris-Panthéon-Assas University, Paris, France, DIENS, École normale supérieure, Paris, France, PSL University, France, CNRS, France, Inria, France
Éric Sageloli, Thales, Gennevilliers, France, DIENS, École normale supérieure, Paris, France, PSL University, France, CNRS, France, Inria, France

Abstract

Introduced by Canetti in 2001, Universal Composability (UC) is a widely adopted security model that enables the specification and proof of security for a broad range of protocols, offering strong security guarantees. At its core lies the universal composition theorem (UC theorem), which ensures that protocols proven secure within the framework remain secure even when deployed in real-world environments with multiple instances of them.

In this work, we present two key contributions. First, we identify several problems with the UC framework, in particular the UC Theorem. They include counterexamples, limitations that make it unusable for important classes of protocols, and weaknesses in its proof. These problems reveal flaws in nearly all the fundamental concepts of UC.

Second, we propose a revised formulation of the main concepts of UC to address these issues. Although the resulting modifications are nontrivial, our updated definitions are designed to remain as faithful as possible to the structure and intent of the original model.

Note: This is an extended version of the article accepted and published in Volume~3, Issue~1 of IACR Communications in Cryptology. The main change in this version is the addition of Appendices~C, D, and~E, which provide missing proofs, constructions, and descriptions of issues. Note relative to previous ePrint versions. While most technical details were already present in the previous version, the present version includes numerous editorial improvements, as well as a change of template.

Challenge 6: Provable Security, Example

Backup plans, and backup plans for the backup plans

Threshold ECDSA in Production

01 DKLs23 v1 (With OT-Extension based RVOLE and Base OT creation in Signing phase)

02 DKLs23 v2 (With Base OTs based RVOLE)

03 Lindell17

04 CGGMP21

Paper 2022/192

SoftSpokenOT: Communication--Computation Tradeoffs in OT Extension

Lawrence Roy

Abstract

Given a small number of base oblivious transfers (OTs), how does one generate a large number of extended OTs as efficiently as possible? The answer has long been the seminal work of IKNP (Ishai et al., Crypto 2003) and the family of protocols it inspired, which only use Minicrypt assumptions. Recently, Boyle et al. (Crypto 2019) proposed the Silent-OT technique that improves on IKNP, but at the cost of a much stronger, non-Minicrypt assumption: the learning parity with noise (LPN) assumption. We present SoftSpokenOT, the first OT extension to improve on IKNP's communication cost in the Minicrypt model. While IKNP requires security parameter λ bits of communication for each OT, SoftSpokenOT only needs λ/k bits, for any k, at the expense of requiring $2^{k-1}/k$ times the computation. For small values of k, this tradeoff is favorable since IKNP-style
…nplemented SoftSpokenOT and found that our protocol gives almost a $5\times$
…ng.

…ilent protocol for vector oblivious linear evaluation (VOLE) over polynomial-
…k to build maliciously secure 1-of-N OT extension from this VOLE, revisiting
…ng the way, we found several flaws in the existing work, including a practical
…k of Patra et al. (NDSS 2017), while also making some improvements.

You are looking at a specific version 20221012:012203 of this paper. See the latest version.

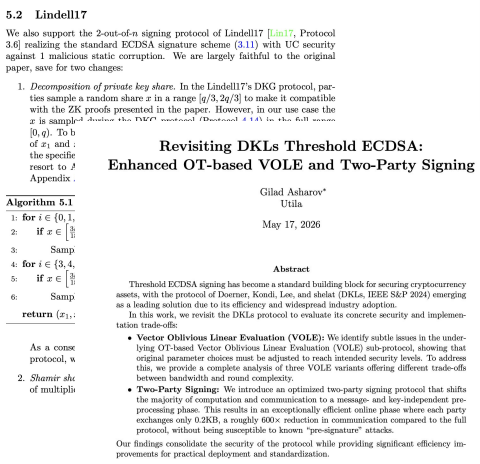
Paper 2022/1371

On the Security of KOS

Benjamin E. Diamond, Coinbase

Abstract

…resent a full proof of security of the original random oblivious transfer extension protocol of Kell…
Scholl (CRYPTO '15), without altering that protocol as written. Our result circumvents a recent n…
lt of Roy (CRYPTO '22), which shows that a key lemma in the original proof of KOS is false. Our p…
rages a new simulation strategy, and a careful analysis of that protocol's "correlation check". We…
tablish evidence of security for this important, widely used protocol.



5.2 Lindell17

We also support the 2-out-of-n signing protocol of Lindell17 [Lin17, Protocol 3.6] realizing the standard ECDSA signature scheme (3.11) with UC security against 1 malicious static corruption. We are largely faithful to the original paper, save for two changes:

1. *Decomposition of private key share.* In the Lindell17's DKG protocol, parties sample a random share x in a range $[q/3, 2q/3]$ to make it compatible with the ZK proofs presented in the paper. However, in our use case the x is sampled during the DKG protocol (Protocol 4.14) in the full range $[0, q)$. To b… of x_1 and … the specifie… resort to … Appendix …

Algorithm 5.1

1: **for** $i \in \{0, 1, \ldots$
2: **if** $x \in \left[\frac{2q}{3}, \ldots$
3: Samp…
4: **for** $i \in \{3, 4, \ldots$
5: **if** $x \in \left[\frac{2q}{3}, \ldots$
6: Samp…
return $(x_1, \ldots$

As a conse… protocol, w…

2. *Shamir sha…* of multipli…

Revisiting DKLs Threshold ECDSA: Enhanced OT-based VOLE and Two-Party Signing

Gilad Asharov*
Utila

May 17, 2026

Abstract

Threshold ECDSA signing has become a standard building block for securing cryptocurrency assets, with the protocol of Doerner, Kondi, Lee, and shelat (DKLs, IEEE S&P 2024) emerging as a leading solution due to its efficiency and widespread industry adoption.

In this work, we revisit the DKLs protocol to evaluate its concrete security and implementation trade-offs:

- **Vector Oblivious Linear Evaluation (VOLE):** We identify subtle issues in the underlying OT-based Vector Oblivious Linear Evaluation (VOLE) sub-protocol, showing that original parameter choices must be adjusted to reach intended security levels. To address this, we provide a complete analysis of three VOLE variants offering different trade-offs between bandwidth and round complexity.
- **Two-Party Signing:** We introduce an optimized two-party signing protocol that shifts the majority of computation and communication to a message- and key-independent preprocessing phase. This results in an exceptionally efficient online phase where each party exchanges only 0.2KB, a roughly 600× reduction in communication compared to the full protocol, without being susceptible to known "pre-signature" attacks.

Our findings consolidate the security of the protocol while providing significant efficiency improvements for practical deployment and standardization.

Paper 2022/1371

On the Security of KOS

Benjamin E. Diamond, Irreducible

Abstract

We study the security of the random oblivious transfer extension protocol of Keller, Orsini, and Scholl (CRYPTO '15), whose security proof was recently invalidated by Roy (CRYPTO '22). We show that KOS is asymptotically secure. Our proof involves a subtle analysis of the protocol's "correlation check", and introduces several new techniques. We also study the protocol's concrete security. We establish concrete security for security parameter values on the order of 5,000. We present evidence that a stronger result than ours—if possible—is likely to require radically new ideas.

Note: Various minor fixes and improvements.

Challenge 6: Provable Security, Example

We now proceed to define security for a distributed signing protocol. We define an experiment that captures concurrent signing executions under the assumption that if an abort occurs, then all executions immediately abort. We call this concurrent executions with instantaneous global abort. This trivially implies security when the signing protocol is run sequentially between two parties, since any abort will imply no later executions.

Lindell17 Abort Vulnerability [CVE-2023-33242]: Technical Report

The Subtleties of Error Handling Flaws in MPC

By Yehuda Lindell, Jeff Lunglhofer, Chief Information Security Officer Aug 9, 2023

Tl;dr: Recently researchers at Fireblocks found an error-handling flaw in a number of MPC libraries that, under certain conditions, could manifest as a significant vulnerability. While our Coinbase Wallet consumer product was not impacted by this issue in any way, the libraries in question were used in a previous version of our Wallet as a Service (WaaS) solution. However, due to Coinbase's approach of implementing multi-layered security across our software stack, there was no way to practically exploit this issue within any of our products. We acknowledge that other organizations and companies across the DeFi ecosystem were more significantly impacted, including some instances that may have enabled a malicious client to extract the secret key from an impacted service. Coinbase immediately released updated libraries with improved error handling to eliminate this issue. We thank the researchers at Fireblocks for identifying this issue, conducting a responsible disclosure, and helping to improve the security of the ecosystem.

Some Tips

Tip: Fat Cryptography

You will face a choice in designing real-world systems:

1. Glue small cryptographic primitives together at a software level. OR
2. Make a fatter cryptographic primitive.

To make a decision:

1. Do you need “exotic” crypto? If so, has it ever been implemented? How “different” are the underlying assumptions?
2. Know that gluing small primitives together is likely to be brittle but often faster to implement.

Tip: Author responsiveness

multi-party-bls

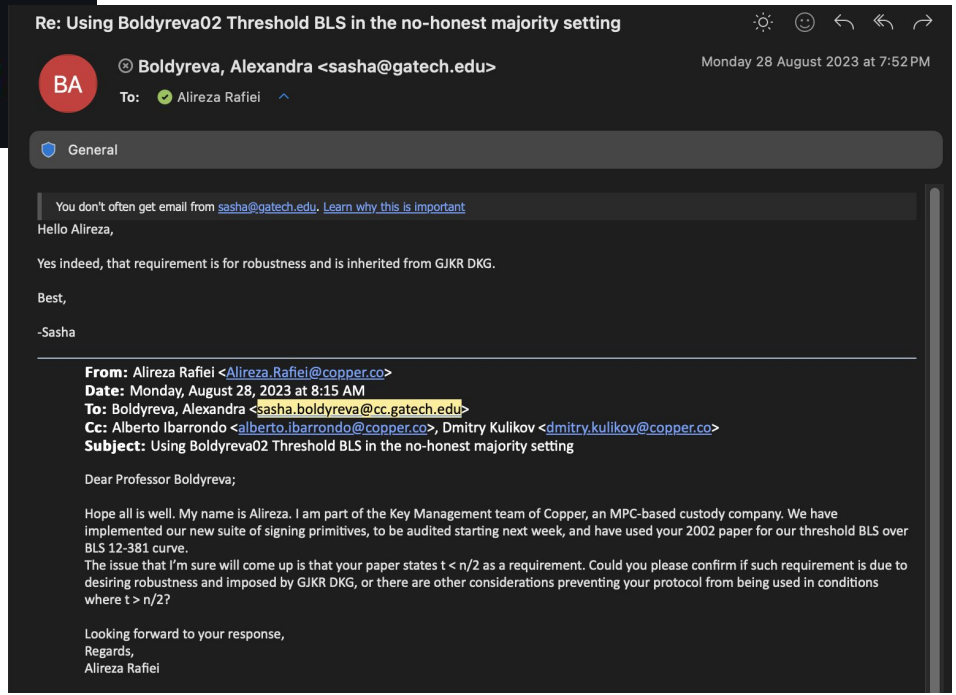
Rust implementation of {t,n}-threshold BLS over BLS12-381 elliptic curve. Currently two protocols are implemented:

- Aggregated BLS. Based on the MSP protocol (BDN18, section 3.1)
- Threshold BLS assuming dishonest majority. Based on Threshold GLOW signatures (GLOW20 version 20200806:135847)

Papers are written with a lot of background Assumptions not explicitly stated.

When in doubt, contact the authors.

Only pick papers where authors respond.



Re: Using Boldyreva02 Threshold BLS in the no-honest majority setting

BA

Boldyreva, Alexandra <sasha@gatech.edu>

To: Alireza Rafiei

Monday 28 August 2023 at 7:52 PM

General

You don't often get email from sasha@gatech.edu. Learn why this is important

Hello Alireza,

Yes indeed, that requirement is for robustness and is inherited from GJKR DKG.

Best,

-Sasha

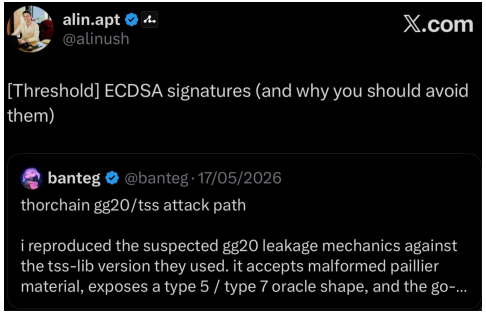
From: Alireza Rafiei <Alireza.Rafiei@copper.co>
Date: Monday, August 28, 2023 at 8:15 AM
To: Boldyreva, Alexandra <sasha.boldyreva@cc.gatech.edu>
Cc: Alberto Ibarrondo <alberto.ibarrondo@copper.co>, Dmitry Kulikov <dmitry.kulikov@copper.co>
Subject: Using Boldyreva02 Threshold BLS in the no-honest majority setting

Dear Professor Boldyreva;

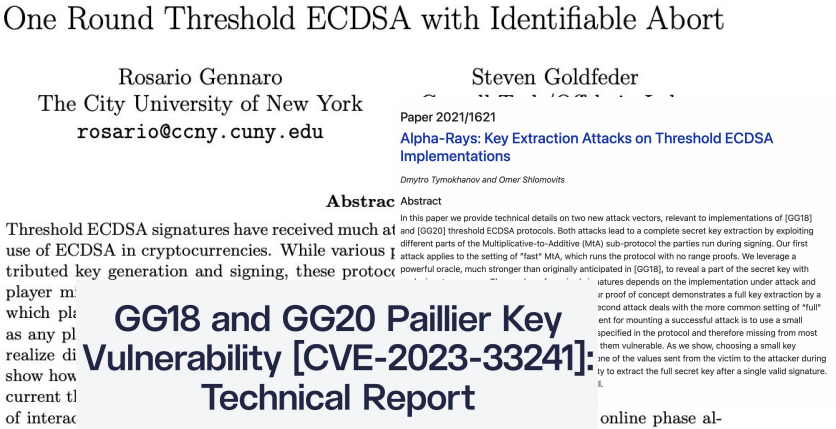
Hope all is well. My name is Alireza. I am part of the Key Management team of Copper, an MPC-based custody company. We have implemented our new suite of signing primitives, to be audited starting next week, and have used your 2002 paper for our threshold BLS over BLS 12-381 curve.
The issue that I'm sure will come up is that your paper states t < n/2 as a requirement. Could you please confirm if such requirement is due to desiring robustness and imposed by GJKR DKG, or there are other considerations preventing your protocol from being used in conditions where t > n/2?

Looking forward to your response,
Regards,
Alireza Rafiei

Tip: Picking Broken Protocols



You may get a broken legacy system, or have to pick it for any reason


One Round Threshold ECDSA with Identifiable Abort

Rosario Gennaro
The City University of New York
rosario@ccny.cuny.edu

Steven Goldfeder

Paper 2021/1621
Alpha-Rays: Key Extraction Attacks on Threshold ECDSA Implementations

Dmytro Tymokhanov and Omer Shlomovits

Abstrac Abstract

In this paper we provide technical details on two new attack vectors, relevant to implementations of [GG18] and [GG20] threshold ECDSA protocols. Both attacks lead to a complete secret key extraction by exploiting different parts of the Multiplicative-to-Additive (MtA) sub-protocol the parties run during signing. Our first attack applies to the setting of "fast" MtA, which runs the protocol with no range proofs. We leverage a powerful oracle, much stronger than originally anticipated in [GG18], to reveal a part of the secret key with

Threshold ECDSA signatures have received much at
use of ECDSA in cryptocurrencies. While various t
tributed key generation and signing, these protoc
player m
which pl
as any pl
realize di
show how
current tl
of interac

GG18 and GG20 Paillier Key Vulnerability [CVE-2023-33241]: Technical Report

online phase al-
lowing for players to asynchronously participate in the protocol without the need to be online simultaneously. We benchmark our protocols and find that our protocol simultaneously reduces the rounds and computations of current protocols, while adding significant functionality: identifiable abort and noninteractivity.

Note: This report is now obsolete and readers should refer to the joint paper [8] which subsumes it. The paper below is a revised version of the previous eprint version which fixes some crucial details in the protocol. The proof of the protocol described in the previous version is not correct, though no attack has been shown that exploits the bug in the proof. More details appear in the Introduction.

What to do?

1. Resist! Don't pick a protocol that's broken.
2. Remember that if something is broken once, its patchwork is more likely to break in the future than something that hasn't been broken at all!

Tip: Consolidate

Papers are not always updated in-time.

Consolidate on what the community converges to and keep a close eye.

OT-based ECDSA protocol implementation flaws

High **veorq** published **GHSA-7f6p-phw2-8253** on Nov 24, 2024

Package	Affected versions	Patched versions
github.com/taurushq-io/multi-party-sig (Go)	<= v0.6.0-alpha-2021-09-21	v0.7.0-alpha-2025-01-28

Description

Coinbase researchers reported 2 security issues in our implementation of the oblivious transfer (OT) based protocol DKLS:

1. Secret share recovery attack

If the base OT setup of the protocol is reused for another execution of the OT extension, then a malicious participant can extract a bit of the secret of another participant. By repeating the execution they can eventually recover the whole secret.

Therefore, unlike our comments suggested, you **must not reuse an OT setup** for multiple protocol executions.

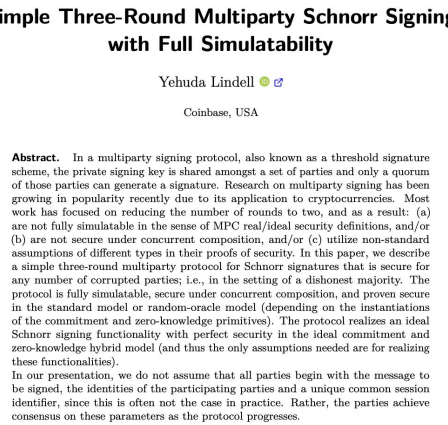
We're adding a warning in the code:

multi-party-sig/internal/ot/extended.go
Line 114 in 9e4400f

```
114     // SECURITY WARNING: A setup must not be reused for multiple invocations of this protocol
```

2. Invalid security proof due to incorrect operator

But know when to stop!!

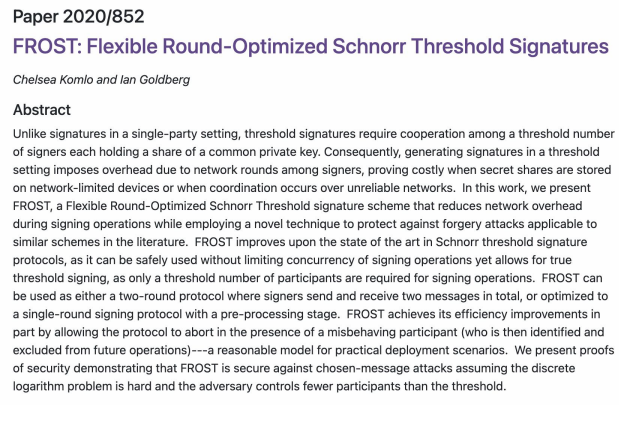


Simple Three-Round Multiparty Schnorr Signing with Full Simulatability

Yehuda Lindell

Coinbase, USA

Abstract. In a multiparty signing protocol, also known as a threshold signature scheme, the private signing key is shared amongst a set of parties and only a quorum of those parties can generate a signature. Research on multiparty signing has been growing in popularity recently due to its application to cryptocurrencies. Most work has focused on reducing the number of rounds to two, and as a result: (a) are not fully simulatable in the sense of MPC real/ideal security definitions, and/or (b) are not secure under concurrent composition, and/or (c) utilize non-standard assumptions of different types in their proofs of security. In this paper, we describe a simple three-round multiparty protocol for Schnorr signatures that is secure for any number of corrupted parties; i.e., in the setting of a dishonest majority. The protocol is fully simulatable, secure under concurrent composition, and proven secure in the standard model or random-oracle model (depending on the instantiations of the commitment and zero-knowledge primitives). The protocol realizes an ideal Schnorr signing functionality with perfect security in the ideal commitment and zero-knowledge hybrid model (and thus the only assumptions needed are for realizing these functionalities).
In our presentation, we do not assume that all parties begin with the message to be signed, the identities of the participating parties and a unique common session identifier, since this is often not the case in practice. Rather, the parties achieve consensus on these parameters as the protocol progresses.



Paper 2020/852

FROST: Flexible Round-Optimized Schnorr Threshold Signatures

Chelsea Komlo and Ian Goldberg

Abstract

Unlike signatures in a single-party setting, threshold signatures require cooperation among a threshold number of signers each holding a share of a common private key. Consequently, generating signatures in a threshold setting imposes overhead due to network rounds among signers, proving costly when secret shares are stored on network-limited devices or when coordination occurs over unreliable networks. In this work, we present FROST, a Flexible Round-Optimized Schnorr Threshold signature scheme that reduces network overhead during signing operations while employing a novel technique to protect against forgery attacks applicable to similar schemes in the literature. FROST improves upon the state of the art in Schnorr threshold signature protocols, as it can be safely used without limiting concurrency of signing operations yet allows for true threshold signing, as only a threshold number of participants are required for signing operations. FROST can be used as either a two-round protocol where signers send and receive two messages in total, or optimized to a single-round signing protocol with a pre-processing stage. FROST achieves its efficiency improvements in part by allowing the protocol to abort in the presence of a misbehaving participant (who is then identified and excluded from future operations)---a reasonable model for practical deployment scenarios. We present proofs of security demonstrating that FROST is secure against chosen-message attacks assuming the discrete logarithm problem is hard and the adversary controls fewer participants than the threshold.

Tip: Choice

If a parameter is security-sensitive, prefer convention over customization.

Disclosing Shamir's Secret Sharing vulnerabilities and announcing ZKDocs

Filipe Casal, Jim Miller | December 21, 2021 | cryptography, vulnerability-disclosure, zero-knowledge

Trail of Bits is publicly disclosing two bugs that affect Shamir's Secret Sharing implementation of **Binance's threshold signature scheme library (tss-lib)** and most of its active forks. Here is the full list of affected repositories:

- **Binance's tss-lib**
- **Clover Network's threshold-crypto**
- **Keep Network's keep-ecdsa**
- **Swingby's tss-lib**
- **THORchain's tss-lib**
- **ZenGo X's curv**

Tip: Ignore the Bullshit

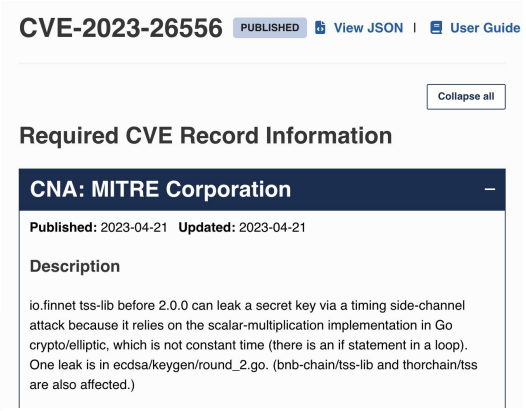
Remote side-channel Free Programming:

01 Not Measurable… Not really.

02 Remote attacks have never happened outside laboratory conditions

03 Impractical

Yet:



Tip: Ignore the bullshit … but have a backup plan

“Best-Effort” Constant Time

Building High level applications by abstracting over constant-time interfaces

```
type monoidElement[E any] interface {
    Set(v E)
    ct.ConditionallySelectable[E]
    ct.Equatable[E]
    Add(lhs, rhs E)
    Double(E)

    SetBytes([]byte) (ok ct.Bool)
    Bytes() []byte
    SetZero()
    IsZero() ct.Bool
    IsNonZero() ct.Bool
}

type MonoidElement[E monoidElement[E]] monoidElement[E]

Alireza Rafiei, 3 months ago | 1 author (Alireza Rafiei)
type MonoidElementPtr[E MonoidElement[E], T any] interface {
    *T
    MonoidElement[E]
}

Alireza Rafiei, 3 months ago | 1 author (Alireza Rafiei)
type groupElement[E any] interface {
    monoidElement[E]
    Sub(lhs, rhs E)
    Neg(E)
}

type GroupElement[E groupElement[E]] groupElement[E]

Alireza Rafiei, 3 months ago | 1 author (Alireza Rafiei)
type GroupElementPtr[E GroupElement[E], T any] interface {…
}

Alireza Rafiei, 3 months ago | 1 author (Alireza Rafiei)
type ringElement[E any] interface {
    groupElement[E]
    SetOne()
    IsOne() ct.Bool
    Mul(lhs, rhs E)
    Square(E)
    Inv(E) (ok ct.Bool)
    Div(lhs, rhs E) (ok ct.Bool)
    Sqrt(E) (ok ct.Bool)
}
```

```
func ScalarMul[PP GroupElementPtr[PP, P], P any](out, pp *P, s []byte) {
    var precomputed [16]P

    PP(&precomputed[0]).SetZero()
    PP(&precomputed[1]).Set(pp)
    for i := 2; i < 16; i += 2 {
        PP(&precomputed[i]).Double(&precomputed[i/2])
        PP(&precomputed[i+1]).Add(&precomputed[i], pp)
    }

    var res P
    PP(&res).SetZero()
    for i := len(s) - 1; i >= 0; i-- {
        PP(&res).Double(&res)
        PP(&res).Double(&res)
        PP(&res).Double(&res)
        PP(&res).Double(&res)
        w := (s[i] >> 4) & 0b1111
        PP(&res).Add(&res, &precomputed[w])

        PP(&res).Double(&res)
        PP(&res).Double(&res)
        PP(&res).Double(&res)
        PP(&res).Double(&res)
        w = s[i] & 0b1111
        PP(&res).Add(&res, &precomputed[w])
    }

    PP(out).Set(&res)
}
```

```
func (m *ModulusOddPrime) ModExp(out, base, exp *Nat) {
    m.ensureMont()
    m.Mod(out, base)

    bBytes := out.Bytes()
    eBytes := exp.Bytes()

    bNum := bnPool.Get().(*boring.BigNum)
    defer bnPool.Put(bNum)
    if _, err := bNum.SetBytes(bBytes); err != nil {
        panic(err)
    }

    eNum := bnPool.Get().(*boring.BigNum)
    defer bnPool.Put(eNum)
    if _, err := eNum.SetBytes(eBytes); err != nil {
        panic(err)
    }

    ctx := bnCtxPool.Get().(*boring.BigNumCtx)
    defer bnCtxPool.Put(ctx)

    rNum := bnPool.Get().(*boring.BigNum)
    defer bnPool.Put(rNum)
    if _, err := rNum.Exp(bNum, eNum, m.mNum, m.mont, ctx); err != nil {
        panic(err)
    }

    rBytes, err := rNum.Bytes()
    if err != nil {
        panic(err)
    }
    out.SetBytes(rBytes)
}
```

```
func (u *Uint) Exp(exponent *Nat) *Uint {
    if exponent == nil {
        panic(errs.NewIsNil("argument is nil"))
    }
    v := new(numct.Nat)
    u.m.ModExp(v, u.v, exponent.v)
    return &Uint{v: v, m: u.m}
}
```

3
An Example

Case-study for productionizing papers

Background: Secret Sharing and Access Structures

Secret Sharing: distributing a secret among a group, in such a way that no individual holds any intelligible information about the secret. A sharing scheme is **ideal** if the size of the secret and each share is the same.

Different kinds of monotone access structures:

1. **Additive** - all parties required to reconstruct the secret)
2. **Boolean (CNF/DNF)** - e.g. (Alice AND Bob) OR (Alice AND Charlie AND Darcy)
3. **Threshold** - any subset of at least t parties required to reconstruct the secret
4. **Hierarchical** - parties split into hierarchy levels; subset of parties of every level required to reconstruct
5. **Threshold Gate** - Threshold + Boolean

Initial Problem

Boss said:

> We want to group some people in the access structure, to ensure presence of certain people during signing.

Fat or Thin?

Boss said:

> We want to group some people in the access structure, to ensure presence of certain people during signing.

Which route to go?

- **Thin:** Build an authorization layer on top of existing threshold scheme.
- **Fat:** Generalize the threshold scheme.

What to pick? First Step

Hierarchical Threshold Secret Sharing

Tamir Tassa
Division of Computer Science, The Open University,
Ra'anana, Israel
tamirta@openu.ac.il

Communicated by Moni Naor

Received 10 June 2003 and revised 4 July 2006
Online publication 7 February 2007

Dynamic and Verifiable Hierarchical Secret Sharing

Giulia Traverso, Denise Demirel, Johannes Buchmann

Technische Universität Darmstadt*

Abstract. In this work we provide a framework for dynamic secret sharing and present the first dynamic and verifiable hierarchical secret sharing scheme based on Birkhoff interpolation. Since the scheme is dynamic it allows, without reconstructing the message distributed, to add and remove shareholders, to renew shares, and to modify the conditions for accessing the message. Furthermore, each shareholder can verify its share received during these algorithms protecting itself against malicious dealers and shareholders. While these algorithms were already available for classical Lagrange interpolation based secret sharing, corresponding techniques for Birkhoff interpolation based schemes were missing. Note that Birkhoff interpolation is currently the only technique available that allows to construct hierarchical secret sharing schemes that are efficient and allow to provide shares of equal size for all shareholder in the hierarchy. Thus, our scheme is an important contribution to hierarchical secret sharing.

Keywords: hierarchical secret sharing, distributed storage, cloud computing, long-term security, Birkhoff interpolation, proactive secret sharing

And it's ideal!

Hierarchical Threshold Signature Scheme

License Apache 2.0 | go report retired | Build Status | codecov 75%

Introduction:

This is Hierarchical Threshold Signature Scheme (HTSS) worked by AMIS. Comparing to Threshold Signature Scheme (TSS), shares in this scheme are allowed to have different ranks.

The main merit of HTSS is vertical access control such that it has "partial accountability". Although TSS achieves joint control to disperse risk among the participants, the level of all shares are equal. It is impossible to distinguish which share getting involved in an unexpected signature. TSS is not like the multi-signature scheme as the signature is signed by distinct private keys in multi-signature scheme. It is because Shamir's secret sharing only supports horizontal access control.

For example, an important contract not only requires enough signatures, but also needs to be signed by a manager. Despite the fact that vertical access control can be realized on the application layer and tracked by an audit log. Once a hack happens, we will have no idea about who to blame for. However, in HTSS framework, through assigning different ranks of each share induces that any valid signature generated includes the share of the manager.

HTSS has been developed by Tassa and other researchers many years ago. In our implementation, we setup up this theory on TSS(i.e. just replace Lagrange Interpolation to Birkhoff Interpolation).

HTSS

Hierarchical Threshold Secret Sharing

Tamir Tassa
Division of Computer Science, The Open University,
Ra'anana, Israel
tamirta@openu.ac.il

Communicated by Moni Naor

Received 10 June 2003 and revised 4 July 2006
Online publication 7 February 2007

Birkhoff instead of Lagrange.

L1 = Alice
L2 = L1 + Bob + Charlie
L3 = L2 + Darcy

T1 = 1
T2 = 2
T3 = 3

At least T1 from L1 AND/OR
At least T2 from L2 AND/OR
At least T3 from L3

HTSS' Deal Breaker

Hierarchical Threshold Secret Sharing

Tamir Tassa
Division of Computer Science, The Open University,
Ra'anana, Israel
tamirta@openu.ac.il

Communicated by Moni Naor

Received 10 June 2003 and revised 4 July 2006
Online publication 7 February 2007

Theorem 3.6 *Under the conditions of Lemma 3.5, the hierarchical threshold secret sharing scheme satisfies conditions (2) and (3) provided that*

$$\alpha(k)N^{(k-1)(k-2)/2} < q = |\mathbb{F}| \quad \textit{where} \quad \alpha(k) := 2^{-k+2} \cdot (k-1)^{(k-1)/2} \cdot (k-1)! \ . \tag{35}$$

Can we salvage HTSS?

Multipartite Secret Sharing by Bivariate Interpolation*

Tamir Tassa† Nira Dyn‡

Abstract

Given a set of participants that is partitioned into distinct compartments, a multipartite access structure is an access structure that does not distinguish between participants that belong to the same compartment. We examine here three types of such access structures: two that were studied before – compartmented access structures and hierarchical threshold access structures, and a new type of compartmented access structures that we present herein. We design ideal perfect secret sharing schemes for these types of access structures that are based on bivariate interpolation. The secret sharing schemes for the two types of compartmented access structures are based on bivariate Lagrange interpolation with data on parallel lines. The secret sharing scheme for the hierarchical threshold access structures is based on bivariate Lagrange interpolation with data on lines in general position. The main novelty of this paper is the introduction of bivariate Lagrange interpolation and its potential power in designing schemes for multipartite settings, as different compartments may be associated with different lines or curves in the plane. In particular, we show that the introduction of a second dimension may create the same hierarchical effect as polynomial derivatives and Birkhoff interpolation were shown to do in [17].

Keywords. Secret sharing, multipartite access structures, compartmented access structures, hierarchical threshold access structures, bivariate interpolation, monotone span programs.

AMS (2000) Subject Classification: Primary: 94A60 Secondary: 65D05

Can we salvage HTSS? No!

Multipartite Secret Sharing by Bivariate Interpolation*

Tamir Tassa† Nira Dyn‡

Abstract

Given a set of participants that is partitioned into distinct compartments, a multipartite access structure is an access structure that does not distinguish between participants that belong to the same compartment. We examine here three types of such access structures: two that were studied before – compartmented access structures and hierarchical threshold access structures, and a new type of compartmented access structures that we present herein. We design ideal perfect secret sharing schemes for these types of access structures that are based on bivariate interpolation. The secret sharing schemes for the two types of compartmented access structures are based on bivariate Lagrange interpolation with data on parallel lines. The secret sharing scheme for the hierarchical threshold access structures is based on bivariate Lagrange interpolation with data on lines in general position. The main novelty of this paper is the introduction of bivariate Lagrange interpolation and its potential power in designing schemes for multipartite settings, as different compartments may be associated with different lines or curves in the plane. In particular, we show that the introduction of a second dimension may create the same hierarchical effect as polynomial derivatives and Birkhoff interpolation were shown to do in [17].

Keywords. Secret sharing, multipartite access structures, compartmented access structures, hierarchical threshold access structures, bivariate interpolation, monotone span programs.

AMS (2000) Subject Classification: Primary: 94A60 Secondary: 65D05

the dealer publishes shares of some dummy participants, in addition to the private shares that are distributed to the real participants. Our schemes are still ideal, since the private shares are drawn from the same field $\mathbb{F}$ from which the secret is drawn. In fact, our proof techniques may be used to show that the dealer may always select the polynomial $P(x, y)$

Secret Sharing Scheme 3:

1. *Each participant from level $\mathcal{C}_i$ will be identified by a unique public point on $L_{k_i} \setminus \left(\bigcup_{\substack{1 \le j \le n \\ j \neq k_i}} L_j \right)$ and his private share will be the value of P at that point.*
2. *In addition, we publish the value of P at:*
   - *k_{i-1} additional points on L_{k_i}, $2 \le i \le m$; and*
   - *j points on L_j for all $j \in \{1, 2, \ldots, n\} \setminus \{k_i : 1 \le i \le m\}$.*

So if not an ideal scheme, then what?

SECRET SHARING SCHEME REALIZING GENERAL ACCESS STRUCTURE

MITSURU ITO, AKIRA SAITO and TAKAO NISHIZEKI

Department of Electrical Communications, Tohoku University
Sendai, Miyagi 980, Japan

Generalized Secret Sharing and Monotone Functions

Josh Benaloh
University of Toronto

Jerry Leichter
Yale University

Finally… MSPs

Definition 4.7 (Monotone Span Programs [111]). *A monotone span program is a triple* MSP $= \langle \mathbb{F}, M, \rho \rangle$, *where* $\mathbb{F}$ *is a field,* M *is an* $a \times b$ *matrix over* $\mathbb{F}$ *for some* $a, b \in \mathbb{N}$, *and* $\rho : \{1, \ldots, a\} \to \{p_1, \ldots, p_n\}$ *labels each row of* M *by a party.*[11] *The size of* MSP *is the number of rows of* M *(i.e.,* a*). For any set* $A \subseteq \{p_1, \ldots, p_n\}$, *let* M_A *denote the sub-matrix obtained by restricting* M *to the rows labeled by parties in* A. *We say that* MSP *accepts* B *if the rows of* M_B *span the vector* $\mathbf{e_1} \stackrel{def}{=} \langle 1, 0, \ldots, 0 \rangle$. *We say that* MSP *accepts an access structure* Γ *if* MSP *accepts a set* B *iff* $B \in \Gamma$.

- Most Generic
- Any access structure induce an MSP
- MSP can be converted to additive
- BUT NO CONCRETE ALGORITHM ANYWHERE!!

Efficient Generation of Linear Secret Sharing Scheme Matrices from Threshold Access Trees

Zhen Liu, Zhenfu Cao, and Duncan S. Wong

Strategy of transforming existing signing protocols

Ideally one would compute share of signature and then aggregate [in the exponent].

Often, it's simpler (and impossible otherwise) to:

Convert share to additive -> Compute partial signature -> Aggregate.

We would represent shares as MSP-compatible, then convert and proceed normally.

DONE!! …. Right?

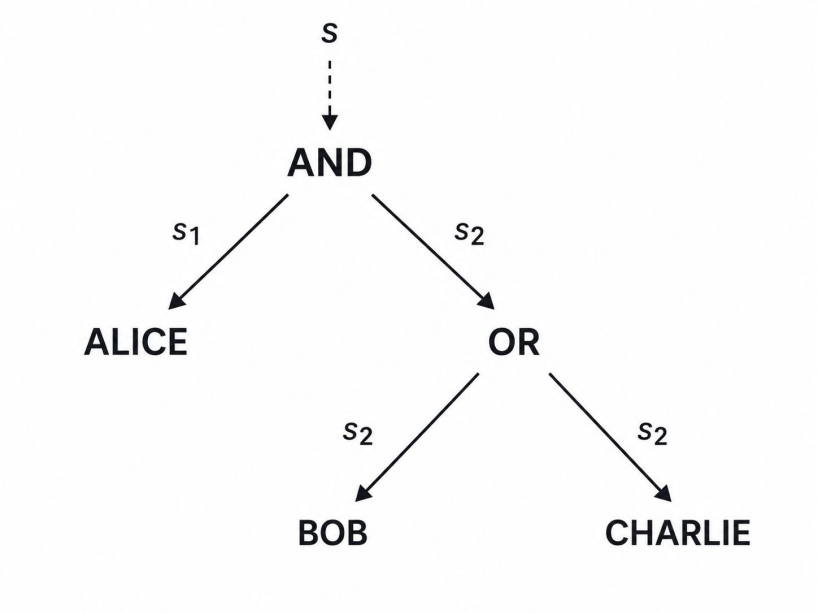
Gotcha #1: Share uniformity

In Shamir, conversion to additive is uniform.

In MSP, that’s not the case!

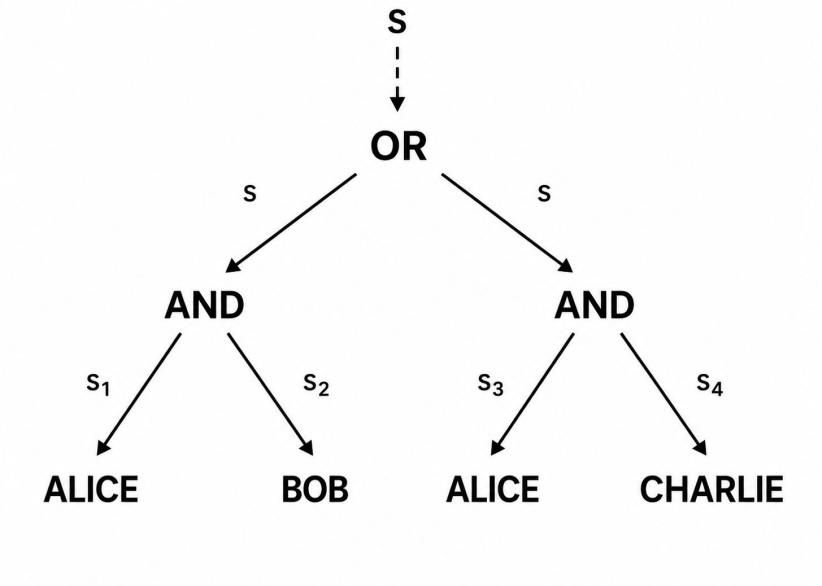
- It may be zero (if quorum is not minimal qualified)
- It may be duplicate (leaves of an OR gate)

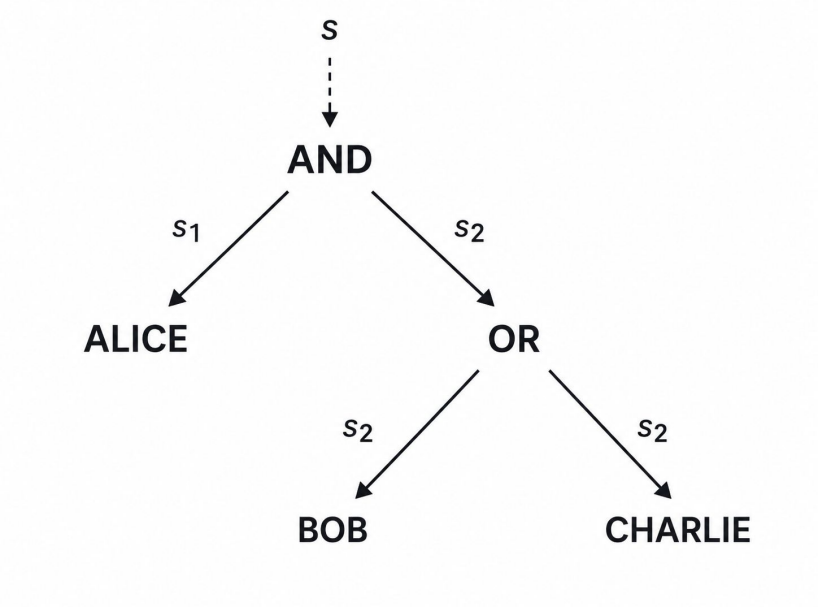
So, rerandomization after additive conversion is required.



Gotcha #2: Output of Equivalent Boolean Formulae

The following two trees are equivalent at a logical level, but result in different share sizes.





Gotcha #3: Share recovery

In Shamir, offline share recovery is always possible.

With MSPs, that's not the case and it must be (i) interactive, and (ii) accompanied with a refresh.

Verifiable Secret Redistribution for Threshold Sharing Schemes

Abstract

We present a new protocol for the verifiable redistribution of secrets from (m,n) to (m',n') access structures for threshold sharing schemes. Our protocol enables the addition or removal of shareholders and also guards against mobile adversaries that cause permanent damage. We observe that existing protocols either cannot be readily extended to allow redistribution between different access structures, or have vulnerabilities that allow faulty old shareholders to corrupt the shares of new shareholders. Our primary contribution is that, in our protocol, new shareholders can verify the validity of their shares after redistribution between different access structures.

Keywords: threshold secret sharing, verification, proactive security.

4
Specifications

Why it matters, and why no one does it.

Spec: Out of Sync Spec

Protocol 2.12 $\mathsf{ROTe}_{\xi,\ell}(\boldsymbol{x}) \to (\boldsymbol{m_0}, \boldsymbol{m_1}, \boldsymbol{m_x})$

Maliciously secure ROT extension protocol from SoftSpokenOT [Roy22] for a batch with ξ messages of $\ell\times\kappa$ bits each, setting $\eta{=}\xi\times\ell\times\kappa$ and $\mu{=}\eta/\sigma$. It uses a pseudo-random generator $\mathsf{PRG}\colon \mathbb{Z}_2^\kappa \mapsto \mathbb{Z}_2^{\eta'}$ for $\eta'{=}\eta{+}\sigma$ (e.g., $\mathsf{TmmoHash}_{\eta'}$), a transcript T (e.g., Scheme 2.4), a base OT (BBOT) and a hash $\mathsf{H}\colon \mathbb{Z}_2^\kappa \mapsto \mathbb{Z}_2^\kappa$

Players: sender $\mathcal{S}$, and receiver $\mathcal{R}$
Inputs: $\mathcal{R}: \boldsymbol{x} \in \mathbb{Z}_2^\xi$, the choice bits
Outputs: $\mathcal{S}: \boldsymbol{m_0}, \boldsymbol{m_1} \in \mathbb{Z}_2^\eta$, pairs of random messages
$\mathcal{R}: \boldsymbol{m_x} \in \mathbb{Z}_2^\eta$, chosen messages such that
$m_{x(i,j)} = m_{0(i,j)} x_{(i)} \oplus m_{1(i,j)}(1 - x_{(i)}) \quad \forall i \in [\xi] \;\; \forall j \in [\ell]$

$\mathcal{S}\&\mathcal{R}.\mathsf{Setup}() \dashrightarrow \mathcal{S}: (\boldsymbol{k_0}, \boldsymbol{k_1});\ \mathcal{R}: \boldsymbol{k_b}$
1: $\mathcal{S}$ samples $\boldsymbol{b} \xleftarrow{\$} \mathbb{Z}_2^\kappa$ as the base OT choice bits
2: $\mathcal{R}$ runs $\mathsf{BBOT}_\kappa()$ as the base OT sender, obtaining $\boldsymbol{k_0}, \boldsymbol{k_1} \in \mathbb{Z}_2^{\kappa\times\kappa}$
3: $\mathcal{S}$ runs $\mathsf{BBOT}_\kappa(\boldsymbol{b})$ as the base OT receiver, receiving $\boldsymbol{k_b} \in \mathbb{Z}_2^{\kappa\times\kappa}$

$\mathcal{R}.\mathsf{Round1}(\boldsymbol{x} \in \mathbb{Z}_2^\xi) \dashrightarrow \boldsymbol{u}, \dot{\boldsymbol{x}}, \dot{\boldsymbol{t}}, \boldsymbol{m_x}$
1: Set $\boldsymbol{x}_{\text{rep}} \leftarrow \{\{x_{(i)}, x_{(i)}, \ldots, x_{(i)}\}_\ell\}_{i\in[\xi]}$ by repeating ℓ times $\boldsymbol{x}$
2: Sample $\boldsymbol{x_\sigma} \xleftarrow{\$} \mathbb{Z}_2^\sigma$ and concatenate $\boldsymbol{x'} \leftarrow \boldsymbol{x}_{\text{rep}} \,||\, \boldsymbol{x_\sigma}$
3: Extend $(\boldsymbol{t_0}, \boldsymbol{t_1}) \leftarrow \{\mathsf{PRG}(k_{0(i)}), \mathsf{PRG}(k_{1(i)})\}_{i\in[\kappa]}$ with $\boldsymbol{t_0}, \boldsymbol{t_1} \in \mathbb{Z}_2^{\kappa\times\eta'}$
4: $\boldsymbol{u} \leftarrow \{\{t_{0(i,j)} \oplus t_{1(i,j)} \oplus x'_{(j)}\}_{j\in[\eta']}\}_{i\in[\kappa]}$ with $\boldsymbol{u} \in \mathbb{Z}_2^{\kappa\times\eta'}$
5: Run $\mathsf{T}.\mathsf{Append}(\boldsymbol{u})$ and $\boldsymbol{\chi} \leftarrow \mathsf{T}.\mathsf{Extract}(\eta)$ to get the challenge $\boldsymbol{\chi} \in \mathbb{Z}_2^{\mu\times\sigma}$
6: Compute the challenge response $(\dot{\boldsymbol{x}}, \dot{\boldsymbol{t}})$ as:
6.a: $\dot{\boldsymbol{x}} \leftarrow \{x_{\sigma(k)} \oplus \bigoplus_{m=1}^{\mu} \chi_{(m,k)} \cdot x_{\text{rep}(\sigma m+k)}\}_{k\in[\sigma]}$ with $\dot{\boldsymbol{x}} \in \mathbb{Z}_2^\sigma$
6.b: $\dot{\boldsymbol{t}} \leftarrow \{\{t_{0(i,\eta+k)} \oplus \bigoplus_{m=1}^{\mu} \chi_{(m,k)} \cdot t_{0(i,\sigma m+k)}\}_{k\in[\sigma]}\}_{i\in[\kappa]}$ with $\dot{\boldsymbol{t}} \in \mathbb{Z}_2^{\kappa\times\sigma}$
7: Transpose $\boldsymbol{t'_0} \leftarrow \{\{t_{0(i,j)}\}_{i\in[\kappa]}\}_{j\in[\eta']}$ with $\boldsymbol{t'_0} \in \mathbb{Z}_2^{\eta'\times\kappa}$
8: $\mathsf{Send}(\boldsymbol{u}, \dot{\boldsymbol{x}}, \dot{\boldsymbol{t}}) \to \mathcal{S}$
9: $\boldsymbol{m_x} \leftarrow \{\{\mathsf{H}(j \,||\, t'_{0(j\ell+l)})\}_{l\in[\ell]}\}_{j\in[\eta]}$
return $\boldsymbol{m_x}$

$\mathcal{S}.\mathsf{Round2}(\boldsymbol{u}, \dot{\boldsymbol{x}}, \dot{\boldsymbol{t}}) \dashrightarrow (\boldsymbol{m_0}, \boldsymbol{m_1})$
1: Extend $\boldsymbol{t_b} \leftarrow \{\mathsf{PRG}(k_{b(i)})\}_{i\in[\kappa]}$ with $\boldsymbol{t_b} \in \mathbb{Z}_2^{\kappa\times\eta'}$
2: $\boldsymbol{q} \leftarrow \{\{b_{(i)} \cdot u_{(i,j)} \oplus t_{b(i,j)}\}_{j\in[\eta']}\}_{i\in[\kappa]}$ with $\boldsymbol{q} \in \mathbb{Z}_2^{\kappa\times\eta'}$
3: Run $\mathsf{T}.\mathsf{Append}(\boldsymbol{u})$ and $\boldsymbol{\chi} \leftarrow \mathsf{T}.\mathsf{Extract}(\eta)$ to get the challenge $\boldsymbol{\chi} \in \mathbb{Z}_2^{\mu\times\sigma}$
4: Verify the challenge response:
4.a: $\dot{\boldsymbol{q}} \leftarrow \{\{q_{(i,\eta+k)} \oplus \bigoplus_{m=1}^{\mu} \chi_{(m,k)} \cdot q_{(i,\sigma m+k)}\}_{k\in[\sigma]}\}_{i\in[\kappa]}$ with $\dot{\boldsymbol{q}} \in \mathbb{Z}_2^{\kappa\times\sigma}$
4.b: Check $q_{(i,k)} \stackrel{?}{=} \dot{q}_{(i,k)} \;\; \forall i \in [\kappa] \;\; \forall k \in [\eta']$; otherwise ABORT
5: Transpose $\boldsymbol{q'} \leftarrow \{\{q_{(i,j)}\}_{i\in[\kappa]}\}_{j\in[\eta']}$ with $\boldsymbol{q'} \in \mathbb{Z}_2^{\eta'\times\kappa}$
6: $(\boldsymbol{m_0}, \boldsymbol{m_1}) \leftarrow \{\{\mathsf{H}(j \,||\, (q_{(j\ell+l)} \oplus b_{(j\ell+l)}))\}_{l\in[\ell]}\}_{j\in[\xi]}$
return $(\boldsymbol{m_0}, \boldsymbol{m_1})$

Protocol 5.15 $\mathsf{RVOLE}_{\ell,q}$

A two-party protocol [DKLs23, Protocol 5.2] realizing the $\mathcal{F}^{RVOLE}_{q,\ell}$ random multiplicative to additive share generation with malicious security in a group $\mathbb{G}(q, G)$, based on a $\mathsf{ROTe}_{\xi,\ell}$ with batch-size $\xi = |q| + 2\sigma$, a hash function $\mathsf{H}_{2\kappa}\colon \{0,1\}^* \mapsto \mathbb{Z}_2^\kappa$ and a hash-to-field function $\mathsf{H}_{\mathbb{Z}_q^{\ell\times\rho}}\colon \{0,1\}^* \mapsto \mathbb{Z}_q^{\ell\times\rho}$ with $\rho = \lceil |q|/\kappa \rceil$

Players: Alice $\mathcal{P}_A$, and Bob $\mathcal{P}_B$
Inputs: $\mathcal{P}_A \to \boldsymbol{a} \in \mathbb{Z}_q^\ell$; $\mathcal{P}_A\&\mathcal{P}_B \to \mathrm{sid} \in \{0,1\}^*$ as session id
Outputs: $\mathcal{P}_A \leftarrow \boldsymbol{c} \in \mathbb{Z}_q^\ell$ $\mathcal{P}_B \leftarrow b \in \mathbb{Z}_q, \boldsymbol{d} \in \mathbb{Z}_q^\ell$ s.t. $a_{(i)} \cdot b = c_{(i)} + d_{(i)} \;\; \forall i \in [\ell]$

$\mathcal{P}_A\&\mathcal{P}_B.\mathsf{Setup}() \dashrightarrow$
0: $\mathcal{P}_A\&\mathcal{P}_B$ run $\mathsf{ROTe.Setup}()$ to generate κ Base OT seeds for the $\binom{2}{1}$-ROTe functionality with $\mathcal{P}_A$ as sender $\mathcal{S}$ and $\mathcal{P}_B$ as receiver $\mathcal{R}$
1: Set an arbitrary public gadget vector $\boldsymbol{g} \in \mathbb{Z}_q^\ell$

$\mathcal{P}_B.\mathsf{Round1}() \dashrightarrow (\gamma, b)$
1: Sample $\boldsymbol{\beta} \xleftarrow{\$} \mathbb{Z}_2^\xi$ as bob's ROTe choice bits
2: Run $\boldsymbol{\gamma} \leftarrow \mathsf{ROTe.Round1}(\boldsymbol{\beta})$
3: $b \leftarrow \sum_{j=1}^{\xi} g_{(j)} \cdot \beta_{(i)}$
4: $\mathsf{Send}(\boldsymbol{\gamma}) \to \mathcal{P}_A$
return b as the multiplicative share of $\mathcal{P}_B$

$\mathcal{P}_A.\mathsf{Round2}(\boldsymbol{\gamma}, \boldsymbol{a}) \dashrightarrow (\tilde{\boldsymbol{a}}, \boldsymbol{\eta}, \mu, \boldsymbol{c})$
$\leftarrow \mathsf{ROTe.Round2}(\boldsymbol{\gamma})$, where $\boldsymbol{\alpha_0}, \boldsymbol{\alpha_1} \in \mathbb{Z}_q^{\xi\times(\ell+\rho)}$
$\sum_{j=1}^{\xi} \alpha_{0(j,i)} \cdot g_{(j)}\}_{i\in[\ell]}$
$\mathbb{Z}_q^\rho$
$- \alpha_{1(j,i)} + a_{(i)}\}_{i\in[\ell]} || \{\alpha_{0(j,\ell+k)} - \alpha_{1(j,\ell+k)} + \hat{a}_{(i)}\}_{k\in[\rho]}\}_{j\in[\xi]}$
$\mathrm{i} \,||\, \tilde{\boldsymbol{a}})$
$\sum_{i=1}^{\ell} \theta_{(i,k)} \cdot a_{(i)}\}_{k\in[\rho]}$
$\cdot k) + \sum_{i=1}^{\ell} \theta_{(i,k)} \cdot \alpha_{0(j,i)}\}_{k\in[\rho]}\}_{j\in[\xi]}$
$\mathrm{l} \,||\, \boldsymbol{\mu})$
$\to \mathcal{P}_B$
he output share of $\mathcal{P}_A$

$, \mu) \dashrightarrow (\boldsymbol{z_B})$
$\mathrm{i} \,||\, \tilde{\boldsymbol{a}})$
$\cdot \beta_{(j)} \cdot \tilde{a}_{(j,i)}\}_{i\in[\ell]}\}_{j\in[\xi]}$
$(j)} \cdot \hat{d}_{(j,i)}\}_{i\in[\ell]}$
$) + \beta_{(j)} \cdot \tilde{a}_{(j,\ell+k)}\}_{k\in[\rho]}\}_{j\in[\xi]}$
$+ \sum_{i=1}^{\ell} \theta_{(i,k)} \cdot \hat{d}_{(j,i)} - \beta_{(j)} \cdot \eta_{(k)}\}_{k\in[\rho]}\}_{j\in[\xi]}$
$\mathrm{l} \,||\, \boldsymbol{\mu'})$
μ, ABORT otherwise
he output share of $\mathcal{P}_B$

Protocol 5.16 DKLs23.Sign

t-out-of-n threshold signing protocol from [DKLs23], realizing the standard ECDSA functionality with UC security for a group $\mathbb{G}(q, G)$. The protocol builds on DKG, Przs, $\mathsf{RVOLE}_{2,q}$, a Commitment scheme and a hash function H.

Players: Key share holders: $\{\mathcal{P}_i\}_{i\in[n]}$ holding $\{x_i\}_{i\in[n]}$ and public key Q
Quorum of signers: $\{\mathcal{P}_i\}_{i\in\mathrm{S}}$ for $\mathrm{S}\in[n]^t$ and $\mathrm{S}^*{=}\mathrm{S}\backslash\{i\}$
Inputs: A session identifier sid, and a message $\boldsymbol{m}$
Outputs: A partial signature σ_i per $\mathcal{P}_i$. A signature σ after aggregation

$\mathcal{P}_i.\mathsf{init}() \dashrightarrow (x_i, Q, \zeta_i)$
1: Run $(Q, x_i) \leftarrow \mathsf{DKG}$ to obtain a public and a private key share
2: Run $\mathsf{Przs.Setup1}()$, $\mathsf{Przs.Setup2}()$ and $\mathsf{Przs.Setup3}()$ to setup zero sharing
3: Run $\mathsf{RVOLE.Setup}()$ as Alice with $\mathcal{P}_k$ as Bob $\forall k \in [n]\backslash\{i\}$
4: Run $\mathsf{RVOLE.Setup}()$ as Bob with $\mathcal{P}_k$ as Alice $\forall k \in [n]\backslash\{i\}$

$\mathcal{P}_i.\mathsf{Round1}() \dashrightarrow (R_i, \{c'_{ij}, \gamma_{ij}\}_{j\in\mathrm{S}^*})$
1: Sample $\phi_i \xleftarrow{\$} \mathbb{Z}_q$ as an inversion mask and $r_i \xleftarrow{\$} \mathbb{Z}_q$ as an instance key
2: $R_i \leftarrow r_i \cdot G$ as the public instance key
3: **for** $j \in \mathrm{S}^*$ **do**
4: Run $(c'_{ij}, w_{ij}) \leftarrow \mathsf{Commit}(i \,||\, j \,||\, sid \,||\, R_i)$
5: Run $(\gamma_{ij}, b_{ij}) \leftarrow \mathsf{RVOLE.Round1}()$ as Bob
6: $\mathsf{Send}(c'_{ij}, \gamma_{ij}) \to \mathcal{P}_j$
7: $\mathcal{F}^{Broadcast}(R_i)$

$\mathcal{P}_i.\mathsf{Round2}(\{R_j, c'_{ji}, \gamma_{ji}\}_{j\in\mathrm{S}^*}) \dashrightarrow (\{\mu_{ij}^{ot2}, \Gamma_{ij}^v, \Gamma_{ij}^u, b_{ij}, w_{ij}\}_{j\in\mathrm{S}^*}, R_i, P_i)$
1: Run $\zeta_i \leftarrow \mathsf{Przs.Sample}()$ to get a zero share
2: $a_i \leftarrow \mathsf{ShamirToAdditive}(i, \mathrm{S}, x_i)$
3: $sk_i \leftarrow a_i + \zeta_i$ and $P_i \leftarrow sk_i \cdot G$ as refreshed instance key shares
4: **for** $j \in \mathrm{S}^*$ **do**
5: Run $(\boldsymbol{\mu}^{ot2}, \boldsymbol{c} \equiv \{c^u, c^v\})_{ij} \leftarrow \mathsf{RVOLE.Round2}(\gamma_{ij}, \boldsymbol{a} = \{r_i, sk_i\})$ as Alice
6: $\Gamma_{ij}^u \leftarrow c_{ij}^u \cdot G$ and $\Gamma_{ij}^v \leftarrow c_{ij}^v \cdot G$
7: $\psi_{ij} \leftarrow \phi_i - b_{i,j}$
8: $\mathsf{Send}(\boldsymbol{\mu}_{ij}^{ot2}, \Gamma_{ij}^u, b_{ij}, w_{ij}, R_i) \to \mathcal{P}_j$
9: $\mathcal{F}^{Broadcast}(P_i)$

$\mathcal{P}_i.\mathsf{Round3}(\boldsymbol{m}, \{\tilde{\boldsymbol{a}}_{ji}, \boldsymbol{\eta}_{ji}, \mu_{ji}, \Gamma_{ji}^u, \Gamma_{ji}^v, b_{ji}, w_{ji}, P_j\}_{j\in\mathrm{S}^*}) \dashrightarrow \sigma_i$
1: **for** $j \in \mathrm{S}^*$ **do**
2: Run $\mathsf{Open}(j||i||sid||R_j, c'_{ji}, w_{ji})$, ABORT if it fails
3: Run $(\boldsymbol{d} \equiv \{d_{ij}^u, d_{ij}^v\}) \leftarrow \mathsf{RVOLE.Round3}(\boldsymbol{\mu}_{ij}^{ot2} = \{\tilde{\boldsymbol{a}}, \boldsymbol{\eta}, \mu\})$ as Bob
4: Check if $b_{ji} \cdot R_j - \Gamma_{ji}^u \stackrel{?}{=} d_{ij}^u \cdot G$ otherwise ABORT
5: Check if $b_{ji} \cdot P_i - \Gamma_{ji}^v \stackrel{?}{=} d_{ij}^v \cdot G$ otherwise ABORT
6: Check if $\sum_{j\in\mathrm{S}} P_j = Q$, otherwise ABORT
7: $R \leftarrow \sum_{j\in\mathrm{S}} R_j$
8: $u_i \leftarrow r_i \cdot (\phi_i + \sum_{j\in\mathrm{S}^*} \psi_{ji}) + \sum_{j\in\mathrm{S}^*} (c_{ij}^u + d_{ij}^u)$
9: $v_i \leftarrow sk_i \cdot (\phi_i + \sum_{j\in\mathrm{S}^*} \psi_{ji}) + \sum_{j\in\mathrm{S}^*} (c_{ij}^v + d_{ij}^v)$
10: $w_i \leftarrow \mathsf{SHA2}(\boldsymbol{m}) \cdot \phi_i + \langle R_x \rangle \cdot v_i$
11: **return** $\sigma_i = \{u_i, w_i\}$

Contents

Spec: No Test Vectors

ARTICLE

Security and Practical Considerations When Implementing the Elliptic Curve Integrated Encryption Scheme

Authors: V. Gayoso Martínez, L. Hernández Encinas, A. Queiruga Dios

Cryptologia, Volume 39, Issue 3 • Pages 244 - 269 • https://doi.org/10.1080/016

Published: 01 July 2015 Publication History

3 0

Abstract

The most popular encryption scheme based on elliptic curves is the
Integrated Encryption Scheme ECIES, which is included in ANSI X9
ISO/IEC 18033-2, and SECG SEC 1. These standards offer many E(
compatible, making it difficult to decide what parameters and crypt
in a specific deployment scenario. In this work, the authors show th
implementation of ECIES can only be compatible with two of the fou
standards. They also provide the list of functions and options that m
implementation. Finally, they present the results obtained when testi
implemented as a Java application, which allows them to offer some
performance and feasibility of their proposed solution.

IEEE Std 1363™-2000

IEEE Standard Specifications for Public-Key Cryptography

Sponsor

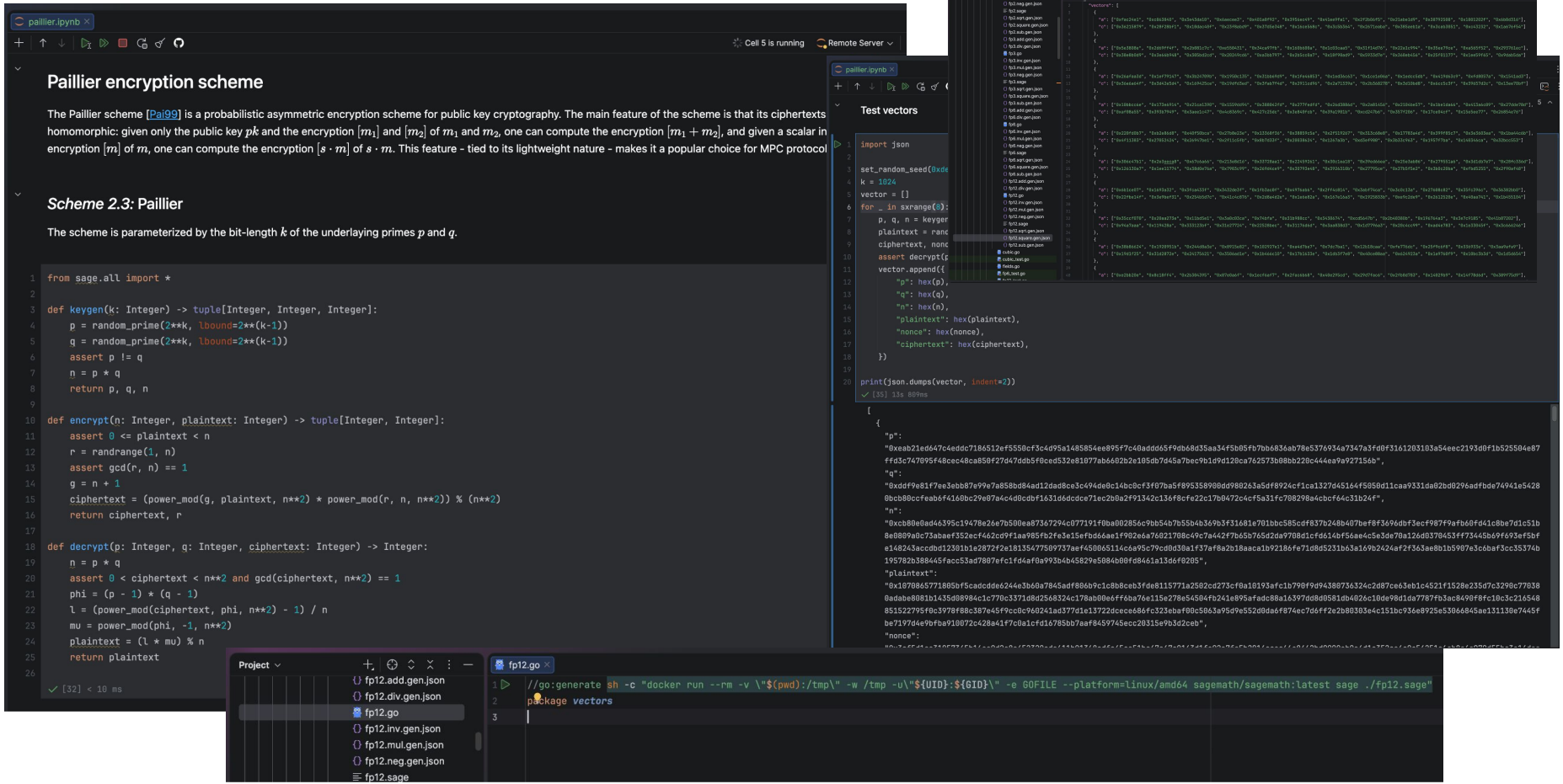
Microprocessor Standards Committee
of the
IEEE Computer Society

Approved 30 January 2000
IEEE-SA Standards Board

Approved 27 July 2000
American National Standards Institute

Abstract: This standard specifies common public-key cryptographic techniques, including mathematical primitives for secret value (key) derivation, public-key encryption, and digital signatures, and cryptographic schemes based on those primitives. It also specifies related cryptographic parameters, public keys, and private keys. The purpose of this standard is to provide a reference for specifications on a variety of techniques from which applications may select.
Keywords: digital signature, encryption, key agreement, public-key cryptography

Spec: Solution



5
Bron-Crypto Is open source!

Thank you!

alireza.rafiei@bron.xyz