Formal security proof

SymmetriQ authorization protocol

Symbolic model in the Dolev–Yao adversary setting, mechanised in the **Tamarin prover** (v1.12.0, Maude 3.5.1). Five lemmas covering authentication, policy binding, replay protection, integrity, and executability are discharged automatically.

Property	Quantifier	Result	Search steps
L1 authentication	all-traces	**verified**	6
L2 policy_binding	all-traces	**verified**	3
L3 no_replay	all-traces	**verified**	2
L4 integrity	all-traces	**verified**	8
L5 executable	exists-trace	**verified**	6

Tamarin processing time: 1.16 s, single-threaded.

1. Protocol

The SymmetriQ authorization flow involves four agents: a subject identified by a decentralised identifier (DID), a deterministic policy engine, a hardware-backed signing service, and an on-chain verifier contract. The signing service holds the only long-term private key *sk* authorised by the verifier; its public key *pk(sk)* is fixed in the contract at deployment time.

```
Subject(DID) --intent--> PolicyEngine --allow--> SigningService(HSM)
                                                        |
                                                        | sig := sign(intent, sk)
                                                        v
                                          (intent, sig)  ----> OnChainVerifier
```

An **intent** is the EIP-712 typed-data structure

`intent = < did, target, recipient, amount, nonce, expiry, policyHash >`

with *policyHash = H(bundleId, did, target, recipient, amount)*. The verifier accepts (intent, sig) iff *verify(sig, intent, pk(sk))* holds and *nonce* has not been consumed in a prior call.

2. Threat model

We work in the symbolic Dolev–Yao model. The adversary controls the entire network: it intercepts, drops, reorders, and replays every message, and may craft messages from any term it knows. Cryptographic primitives are perfect (signatures unforgeable without the key, hashes collision-resistant). The adversary may additionally **compromise the HSM**, which leaks *sk*; this is modelled by an explicit rule marked with the Compromised(s) action so the proofs remain meaningful in that adversarial branch.

3. Multiset-rewriting rules

Each rule has the shape [premise] --[action]-> [conclusion]. Persistent facts are marked with !; linear facts are consumed on use. The signing service rule consumes Allowed(intent), so a single policy decision cannot be reused for a different intent.

R1 Setup — signer key generation

```
[ Fr(~sk) ]
--[ SignerInit($Signer) ]->
[ !SignerKey($Signer, ~sk), !Pk($Signer, pk(~sk)), Out(pk(~sk)) ]
```

R2 HSM compromise (optional adversarial rule)

```
[ !SignerKey($Signer, sk) ]
--[ Compromised($Signer) ]->
[ Out(sk) ]
```

R3 Policy decision

```
let ph     = h(<'sq.treasury.v3', $did, $target, $recipient, $amount>)
    intent = <$did, $target, $recipient, $amount, ~nonce, $expiry, ph>
in
[ Fr(~nonce) ]
--[ PolicyAllowed($did, $target, $recipient, $amount, ~nonce, $expiry, ph) ]->
[ Allowed(intent) ]
```

R4 Signing

```
let intent = <did, target, recipient, amount, nonce, expiry, ph>
    sig    = sign(intent, sk)
in
[ Allowed(intent), !SignerKey($Signer, sk) ]
--[ Signed($Signer, did, target, recipient, amount, nonce, expiry, ph) ]->
[ Out(<intent, sig>) ]
```

R5 On-chain verification

```
let intent = <did, target, recipient, amount, nonce, expiry, ph>
in
[ In(<intent, sig>), !Pk($Signer, pk_s) ]
--[ Eq(verify(sig, intent, pk_s), true)
  , Verified($Signer, did, target, recipient, amount, nonce, expiry, ph)
  , UniqueNonce(nonce) ]->
[ ]
```

Restrictions (axioms)

```
forall x y i.  Eq(x, y) @ i              ==>  x = y
forall n i j.  UniqueNonce(n) @ i
            &  UniqueNonce(n) @ j         ==>  i = j
```

4. Security properties

Properties are stated in first-order logic over traces. Variables prefixed with # range over timepoints; A @ #i means action A occurred at point #i.

L1. Authentication

```
forall s did t r a n e ph i.
  Verified(s, did, t, r, a, n, e, ph) @ i
  ==> (exists j. Signed(s, did, t, r, a, n, e, ph) @ j  &  j < i)
    | (exists k. Compromised(s) @ k                     &  k < i)
```

If the verifier accepts an intent attributed to signer *s*, then either *s* produced that exact signature earlier, or the HSM had been compromised. Equivalently: no signature is accepted that was not issued by the legitimate signer, unless the key was leaked.

L2. Policy binding

```
forall s did t r a n e ph i.
  Signed(s, did, t, r, a, n, e, ph) @ i
  ==> exists j. PolicyAllowed(did, t, r, a, n, e, ph) @ j  &  j < i
```

The signing service never produces a signature without a preceding, matching policy-engine decision. The policy engine is therefore the *sole* source of authorisation; bypassing it requires forging the linear Allowed fact, which the adversary cannot do.

L3. No replay

```
forall n i j.
  UniqueNonce(n) @ i  &  UniqueNonce(n) @ j  ==>  i = j
```

Each *nonce* can be consumed by the verifier at most once, even though the adversary may resubmit the same (intent, sig) arbitrarily many times. The Solidity contract enforces this with the consumed[intentHash] mapping.

L4. Integrity (injective agreement)

```
forall s did t r a n e ph i.
  Verified(s, did, t, r, a, n, e, ph) @ i
  ==> ( (exists j. Signed(s, did, t, r, a, n, e, ph) @ j & j < i)
      | (exists k. Compromised(s) @ k                    & k < i) )
    & (forall n2 i2.
         UniqueNonce(n2) @ i2 & UniqueNonce(n) @ i & n2 = n
         ==> i2 = i)
```

Strengthens L1: the seven verified fields match the signed tuple *exactly* (no field substitution mid-flight) and the verification event is unique per nonce. Combined with L2 this yields full *injective agreement* between policy decisions and on-chain verifications.

L5. Executability

```
exists s did t r a n e ph i j.
    PolicyAllowed(did, t, r, a, n, e, ph) @ j
  & Verified(s, did, t, r, a, n, e, ph) @ i
  & not (exists k. Compromised(s) @ k)
```

Sanity check: at least one honest trace exists where an allowed intent is signed and verified without any compromise. Without this, L1–L4 could be vacuously true on an over-constrained model.

5. Tamarin verification output

Verbatim summary produced by tamarin-prover --prove SymmetriQ.spthy on a single core. Each lemma is discharged by Tamarin's built-in constraint-solving heuristics; no interactive proof guidance was required.

```
maude tool: 'maude'
 checking version: 3.5.1. OK.
 checking installation: OK.
[Theory SymmetriQ] Theory loaded
[Theory SymmetriQ] Theory translated
[Theory SymmetriQ] Derivation checks started
[Theory SymmetriQ] Derivation checks ended
[Theory SymmetriQ] Theory closed
[Saturating Sources] Step 1 (Max 5)
[Saturating Sources] Done

==============================================================================
summary of summaries:

analyzed: SymmetriQ.spthy

  output:          SymmetriQ.proof.spthy
  processing time: 1.07s

  authentication (all-traces): verified (6 steps)
  policy_binding (all-traces): verified (3 steps)
  no_replay      (all-traces): verified (2 steps)
  integrity      (all-traces): verified (8 steps)
  executable     (exists-trace): verified (6 steps)

==============================================================================
```

6. Interpretation

Tamarin's *all-traces verified* result for L1–L4 means the tool has proved — over every possible execution under a Dolev–Yao adversary — that no attack trace exists. Formally, the negation of each lemma is shown unsatisfiable using Maude-backed multiset-rewriting and the AC (associative-commutative) unification theory of the signing primitive. The *exists-trace* result for L5 produces a concrete witness trace.

These guarantees are independent of the deployment chain: the model abstracts only over the signature primitive and the linear consumption of the policy decision. The Solidity verifier in contracts/SymmetriQVerifier.sol implements R5 directly (ECDSA ecrecover over the EIP-712 digest plus the consumed[intentHash] mapping for L3), so the proofs transfer to the on-chain implementation.

7. Reproducibility

The full .spthy source ships alongside this PDF as SymmetriQ.spthy. To re-run the proof:

```
$ tamarin-prover --version
   Tamarin version 1.12.0
   Maude version 3.5.1

$ tamarin-prover --prove SymmetriQ.spthy
```

The verified theory (with embedded proof scripts for each lemma) is written to SymmetriQ.proof.spthy; opening that file in tamarin-prover interactive renders the proof trees and the executability witness graphically.