# Schnorr Signatures

## Zero-Knowledge Proofs & Non-Interactive Variants

Kaspar Etter, November 2018



SHIFT CRYPTOSECURITY

# Motivation

Schnorr is simpler than ECDSA and thus easier to explain and understand. Also security proof exists.

Related to most common zero-knowledge proofs.

Schnorr can easily be extended to ring signatures.

**Linearity Property**:

- Batch validation
- Signature aggregation
- Easier scriptless scripts
- Easier threshold schemes

# Finally Proposed for Bitcoin

https://github.com/sipa/bips/blob/bip-schnorr/bip-schnorr.mediawiki by Pieter Wuille

Why ECDSA instead of Schnorr in the first place?

- Schnorr was patented until February 2008.
- ECDSA was standardized, Schnorr was not.
- For these reasons, ECDSA was more common.

# Digital Signature Scheme

Digital signature schemes consist of 3 algorithms:

- **KeyGeneration**(entropy) –> private k, public K (called key because you can unlock things like coins; k –> K typically easy, K –> k always hard)
- **Signing**(message, k) –> signature (can only be produced by the person who knows the key k)
- **Verifying**(message, K, signature) –> true/false (anyone who knows the public key K can verify)

# Zero-Knowledge Proofs

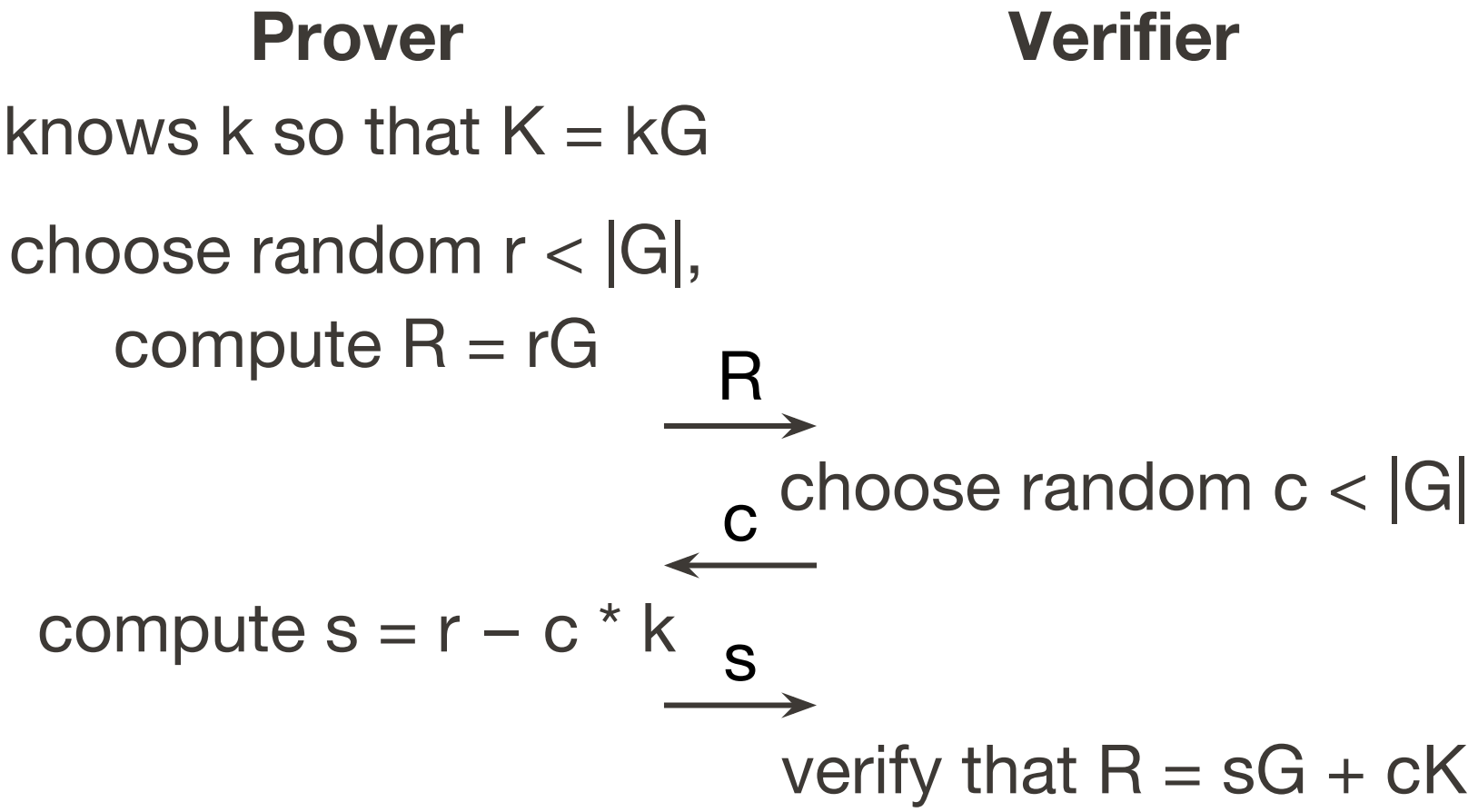
**Goal**: Convince another party of one's knowledge without revealing any information or leaving proof.

**Parties**: Prover convinces verifier of knowing k.

Trivial by revealing k but then not zero-knowledge.

- **Completeness** (successful proof): An honest verifier will be convinced by an honest prover.
- **Soundness** (proof of knowledge): Prover can fake knowledge only with negligible probability.
- **Zero-Knowledge**: Verifier can fake transcript.

# Knowledge of Discrete Logarithm




$$rG = sG + cK = (r - ck)G + c(kG) = rG - ckG + ckG$$

# Evaluation of Criteria

- **Completeness**: Verifier will be convinced.
- **Soundness**: By sending distinct challenges c and c' after the same R, the verifier can extract the secret k by computing $(s - s')/(c' - c)$ because $sG + cK = R = s'G + c'K$ and thus $sG - s'G = (s - s')G = c'K - cK = (c' - c)K$.
- **Zero-Knowledge**: By choosing the random values c and s first, the verifier can compute $R = sG + cK$, which results in a valid transcript.

# Overly Large Footnote

Since the verifier might choose the challenge c non-randomly, this protocol is only so-called **honest-verifier** zero-knowledge. A dishonest verifier can turn this into a signature scheme.

To be a proof of knowledge, the challenge c has to be chosen **after** the temporary point R has been fixed. This dependency can either be established by a verifier or by a cryptographic hash function.

# Non-Interactive Version (Schnorr)

| Signer | Verifier |
| --- | --- |
| knows k so that K = kG | In case of (c, s), verify |
| choose random r < \|G\|, compute R = rG | c = hash(K, sG + cK, m) |
| c = hash(K, R, m) | In case of (R, s), verify |
| compute s = r – c * k | R = sG + hash(K, R, m)K |
| share (c, s) or (R, s) as signature for message m | |

# Why include public key K in hash c?

Computed challenge c = hash(K, R, m) includes

- **R** to ensure signer has to commit to R first.
- **m** to bind the signature to the message m.
- **K** because otherwise anyone else can forge a signature for a public key K' = K + jG = (k + j)G by adapting s to s' = s – c * j = r – c * (k + j). This may or may not be a problem. Since key derivation according to BIP32 performs jumps like this, anyone who knows your xpub could shift you signature from one key to another.

# Recap: ECDSA (for reference)

**Signing** (slightly simplified):

- Choose random $r < |G|$.
- Compute $R = rG$ (with generator $G$).
- Compute $s = (m + R_x * k) / r$ with message $m$, private key $k$ and the x-coordinate of $R$ as $R_x$.
- Attach $(R_x, s)$ to the message as the signature.

**Verifying**:

- Calculate $R = (m / s)G + (R_x / s)K$.
- Accept if and only if x-coordinate of $R = R_x$.

# Linearity of Schnorr Signature

Given $R_1 = s_1G + c_1K$ and $R_2 = s_2G + c_2K$, then $R_1 + R_2 = (s_1 + s_2)G + (c_1 + c_2)K$ also holds. Since normal verification determines only one challenge c, party 1 with $K_1 = k_1G$ and party 2 with $K_2 = k_2G$ need to collaborate to produce a valid signature.

Multisignature scheme with several **advantages**:

- more private (indistinguishable from singlesig)
- cheaper (because shorter and thus less fee)
- more efficient to verify (1 instead of n checks)

# Key Aggregation; n-out-of-n Multisig

Determine aggregated public key $\mathbf{K} = K_1 + \ldots + K_n$.
(K can be encoded as a P2PKH Schnorr address.)

Each signer first computes $R_i = r_i G$ with random $r_i$.

Share the $R_i$s, add them up to $\mathbf{R} = R_1 + \ldots + R_n$.

Calculate common challenge $\mathbf{c}$ = hash(K, R, m).

Each signer then computes solution $s_i = r_i - c * k_i$.

Share the $s_i$s, add them up to $\mathbf{s} = s_1 + \ldots + s_n$.

(R, s) is signature for public key K and message m:
$R_1 + \ldots + R_n = (s_1 + \ldots + s_n)G + \text{hash}(K, R, m)K$.

# Rogue Key Attack

**Problem**: If not careful, cosigner can return a fake public key with which he/she controls the multisig.

**Example**: Instead of returning $K_2$, the $2^{nd}$ party returns $K_2' = K_2 - K_1$, thus $K = K_1 + K_2' = K_2$.

**Solutions**:

- Share first only commitment to $K_i$ with others.
- Require that each cosigner signs its $K_i$ with $k_i$.
- Break the linearity by introducing multiplication factors that depend on all the public keys $K_i$.

# Exponentiation by Squaring

For additive notation (with EC): **double-and-add**

When verifying signatures, point multiplication is the **most expensive operation** computationally.

Instead of calculating point addition n – 1 times to determine nG, we can repeatedly double and add.

**Example**: 13G = G + G + G + G + G + G + G + G + G + G + G + G + G = 2(2(2(0 + G) + G) + 0) + G
Instead of doing 12 point additions, we did only 5.

# Batch Validation

Instead of validating $R_1 = s_1G + c_1K_1$, $R_2 = s_2G + c_2K_2$, etc. with $c_i = \text{hash}(K_i, R_i, m_i)$ separately, we can add them up to $R_1 + R_2 + \ldots = (s_1 + s_2 + \ldots)G + c_1K_1 + c_2K_2 + \ldots$ and thereby halve the number of point multiplications. The latter equation fails if any of the former fails but we do not learn which.

Unfortunately, this is not secure. A miner could add invalid signatures that cancel other invalid signatures (see www.deadalnix.me/2017/02/17/schnorr-signatures-for-not-so-dummies/ for a fix).