Scaling Bitcoin 2018
Signatures Workshop

Andrew Poelstra
Director of Research,
Blockstream

Scriptless Scripts

Taproot

Schnorr

Adaptor Signatures

MuSig

Elliptic Curve Cryptography

- EC crypto consists of geometric points with an addition operator.
- Such a set is a group.
- Any group element G can be "multiplied" by an integer n by adding G to itself repeatedly.

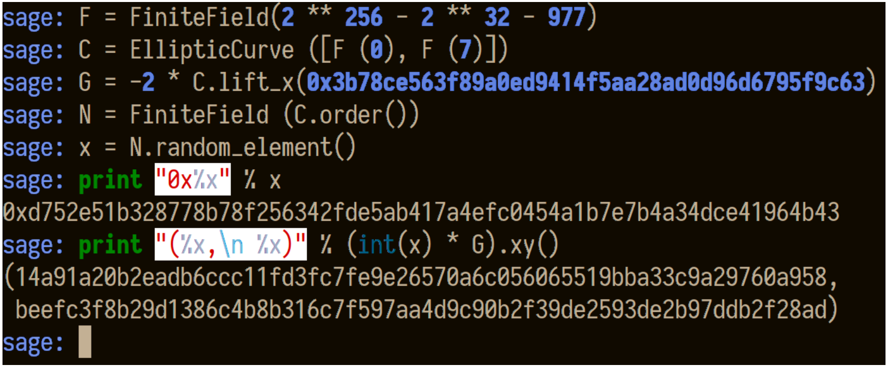
$$\underbrace{G + G + \cdots + G}_{n \text{ times}}$$

- The multiples of G form a cyclic group that we say is generated by G.

Elliptic Curve Cryptography

- Suppose we have a group with $n \sim 2^{256}$ elements generated by some point G.
- We can efficiently map integers into the group by $x \mapsto xG$.
- This is a homomorphism: $xG + yG = (x + y)G$.
- In theory we can map $xG \mapsto x$, but in practice it is an open problem to do this efficiently with only a classical computer. This is the discrete logarithm problem.
- We can think of x as a secret key and xG as a public key.

Elliptic Curve Cryptography



```
sage: F = FiniteField(2 ** 256 - 2 ** 32 - 977)
sage: C = EllipticCurve ([F (0), F (7)])
sage: G = -2 * C.lift_x(0x3b78ce563f89a0ed9414f5aa28ad0d96d6795f9c63)
sage: N = FiniteField (C.order())
sage: x = N.random_element()
sage: print "0x%x" % x
0xd752e51b328778b78f256342fde5ab417a4efc0454a1b7e7b4a34dce41964b43
sage: print "(%x,\n %x)" % (int(x) * G).xy()
(14a91a20b2eadb6ccc11fd3fc7fe9e26570a6c056065519bba33c9a29760a958,
 beefc3f8b29d1386c4b8b316c7f597aa4d9c90b2f39de2593de2b97ddb2f28ad)
sage: 
```

Elliptic Curve Commitments

- Let H be some cryptographic hash function, c some message, and (x', P') a keypair.
- Then $P = P' + H(P' \| c)G$ is a commitment to c.
- Further, if $x = x' + H(P' \| c)$, then (x, P) is a keypair and somebody who knows x' definitely knows x.
- Further, somebody who knows (P', c) can verify the commitment, while somebody who knows neither cannot learn anything about them from P.

Taproot and Pay-to-Contract

- Consider a hypothetical Bitcoin output labelled by a key P, such that it may be spent by a signature with this key.
- Validators who see this output cannot determine how P was generated, and an ordinary transaction does not reveal anything about it.
- Suppose, however, that there exists some script c and alternate point P' such that $P = P' + H(P' \| c)G$.

Taproot and Pay-to-Contract

- This construction is called *pay-to-contract* and has been floating around in some form or another since 2014 at least. (A 2012 paper by Gerhardt and Hanke described a similar but broken scheme.)
- This is used in Liquid and Elements: the script c is actually the scriptPubkey of an output on the sidechain.
- But on Bitcoin itself, we could allow a second way to spend these coins: reveal (P, c) and satisfy the script c. Validators would check the commitment and check the witness for c.
- This is Taproot.

ECDSA and EC-Schnorr

- Suppose we have a keypair (x, P) with $P = xG$.
- To sign a message m, first generate an ephemeral keypair or nonce, (k, R).
- Compute $r = R_x$ and $e = H(m)$ (ECDSA) or $e = H(P \| R \| m)$ (Schnorr).
- Compute s such that

$$sk = e + rx \qquad \text{(ECDSA)}$$
$$s = k + ex \qquad \text{(Schnorr)}$$

- The signature is (s, R).

Sign-to-Contract

- By the way, we can do the pay-to-contract/Taproot construction on R.
- The result is called sign-to-contract and allows committing things to the blockchain in ordinary transactions with no additional space.

ECDSA and EC-Schnorr

- Verification is basically the same as signing

$$\begin{aligned} sk &= e &+ rx \qquad && \text{(ECDSA)} \\ s &= k &+ ex \qquad && \text{(Schnorr)} \end{aligned}$$

ECDSA and EC-Schnorr

- Verification is basically the same as signing

$$skG = eG + rxG \qquad \text{(ECDSA)}$$
$$sG = kG + exG \qquad \text{(Schnorr)}$$

ECDSA and EC-Schnorr

- Verification is basically the same as signing

$$\begin{aligned} sR &= eG + r\ \ P && \text{(ECDSA)} \\ sG &= \ \ R + e\ \ P && \text{(Schnorr)} \end{aligned}$$

ECDSA and EC-Schnorr

- From this equation we can see that Schnorr signatures are *linear in the components of the signature* (s, R) and can therefore be added:

$$\begin{aligned} s_1 G &= R_1 + eP_1 \\ s_2 G &= R_2 + eP_2 \\ (s_1 + s_2)G &= (R_1 + R_2) + e(P_1 + P_2) \end{aligned}$$

  Notice same e in both equations — this requires interaction to achieve.

- vs ECDSA

$$\begin{aligned} s_1 R_1 &= eG + rP_1 \\ s_2 R_2 &= eG + rP_2 \end{aligned}$$

Multisignatures

- Suppose n parties with keys $\{P_1, \ldots, P_n\}$ want to jointly sign a message m.

- They could create n nonces $\{R_1, \ldots, R_n\}$ and compute

$$P = P_1 + \cdots + P_n$$
$$R = R_1 + \cdots + R_n$$
$$e = H(P \| R \| m)$$

then each compute $s_i = k_i + x_i e$. Writing $s_i = \sum s_i$, we then have

$$sG = \left(\sum s_i\right) G = \left(\sum R_i\right) + e \sum P_i = R + eP$$

Multisignatures

- This is a valid signature with P, but it does **not** prove that each party P_i participated in the production of the signature.
- Suppose that the nth party chose $P_n = P - \sum_{i<n} P_i$ for some key P that xe knows the secret key to.
- Then $P = \sum P_i$ is actually entirely controlled by the nth party, who can produce signatures without anyone else's involvement.

Multisignatures

- We could prevent this by using knowledge-of-secret-key (KOSK), i.e. having the participants refuse to participate unless every other participant had proven control over their individual key.
- This is a bad fit for Bitcoin where the signers don't necessarily choose the public key (the sender does).
- But also, it doesn't work with Taproot! Suppose our attacker chooses P_n by first choosing some P'_n and a script c which gives xir all the coins. Xe computes $P = \sum_{i<n} P_i + P'_n$ and publishes $P_n = P'_n + H(P\|c)G$.

Multisignatures

- Instead, we use a key-combining technique called MuSig[1].
- In MuSig, rather than taking $P = P_1 + \cdots + P_n$, we first compute

$$\begin{aligned} C &= H(P_1 \| \cdots \| P_n) \\ \mu_i &= H(C \| i) && \text{for each } i \\ P &= \sum_i \mu_i P_i \end{aligned}$$

- Now every party must choose their public key P_i before learning any μ_i or P, so it is impossible to cancel others' contributions or add Taproot commitments.

[1] Well, technically MuSig refers to the full signing scheme. But I spent a lot of time mixing keys and basically no time signing anything, so I've started using "MuSig" to refer to just the key-combination part.

Multisignatures — 2-of-2 MuSig Example

1. **Key Setup.** Suppose Alice and Bob choose public keys P_A and P_B, and send these to each other. They can both compute $C = H(P_A \| P_B)$, $\mu_A = H(C\|1)$, $\mu_B = H(C\|2)$ and $P = \mu_A P_A + \mu_B P_B$.

2. **Signing (1).** They agree on a message m. The parties compute nonces (k_A, R_A) and (k_B, R_B) and exchange the public halves. They both compute $R = R_A + R_B$ and $e = H(P\|R\|m)$[2]

3. **Signing (2).** They each compute
$$s_A = k_A + \mu_A x_A e$$
$$s_B = k_B + \mu_B x_B e$$
and add these together.

[2] Edit 2018-10-07 you **must** add a "exchange precommits to R_A, R_B round, cf https://eprint.iacr.org/2018/417

Sign-to-Contract Redox

- Each partial signature should have the form $s_i = k_i + e\mu_i x_i$, where e is a commitment to, among other things, the total nonce R.
- If one party wants the signature to sign some auxillary data — for example, a detailed invoice or audit log in a Lightning payment — they can require that R has the form $R = R' + H(R' \| c)G$, where c is the auxillary data.
- It can be shown that if the underlying signature scheme is secure, then the extended scheme where (R', c) are revealed is also a secure signature on c.

Adaptor Signatures

- Suppose that in the MuSig protocol some party gave an invalid s_i so that the final signature did not validate. Could the guilty party be identified?
- Yes – just as for complete signatures, the “partial signatures” s_i have a verification equation

$$\begin{aligned} s_i &= k_i + e\mu_i x_i \\ s_i G &= k_i G + e\mu_i x_i G \\ s_i G &= R_i + e\mu_i P_i \end{aligned}$$

Adaptor Signatures

- We can extend this verification equation to make these partial signatures more powerful.
- Suppose that party i chooses yet another ephemeral keypair (t_i, T_i), called an adaptor, and offsets the signature as

$$s_i' = k_i + e\mu_i x_i + t_i$$

- This is verifiable; it's equivalent to

$$s_i' G = R_i + e\mu_i P_i + T_i$$

- and anyone who knows s_i' will learn a valid signature s_i on the same hash e **if and only if** they learn t_i.

Adaptor Signatures

Suppose Bob sends coins to an output controlled by the key $P = \mu_A P_A + \mu_B P_B$[3], with the intention of sending these coins to Alice.

1. Like a standard multisignature, they start by exchanging nonces R_A and R_B and computing the messagehash e. However Alice *also* provides a second public key T_A.
2. Alice provides an adaptor signature $s'_A = k_A + \mu_A x_A + t_A$.
3. Bob provides an ordinary partial signature $s_B = k_B + \mu_B x_B$.
4. Alice computes $s_A = k_A + \mu_A x_A$, adds this to s_B to get a complete signature, and uses this to take her coins. In doing so, she reveals t_A to Bob.

[3] And suppose that he adds a Taproot commitment to a timelocked refund.

Adaptor Signatures and ZKCP

- In the previous protocol, Bob "bought" the discrete log t_A of T_A from Alice, in a trustless way.
- The obvious application of this is a zero-knowledge contingent payment.
- Here t_i is the (encryption key to) a witness to some NP problem, and Alice proves beforehand that this is the case.
- Some useful examples exist that are naturally expressed as discrete logs. e.g. Pedersen commitment openings, blind sigs.
- (Jonas Nick has adaptor-signature-based protocols for both.)

Adaptor Signatures and Atomic Swaps

- Finally we get to the point of this workshop: atomic swaps using adaptor signatures.
- On a high level, this works by doing the "sell t_i" protocol twice in parallel.
- More specifically, both Alice and Bob put coins into jointly controlled outputs. They start to sign transactions that send Alice's coins to Bob and Bob's coins to Alice, but once again Alice first provides adaptor signatures in place of real partial signatures.
- Specifically, she provides adaptor signatures with *the same adaptor* T_A. Then when she completes the transaction that gives her coins, Bob learns t_A, which he uses to take his coins.

Thank You

Andrew Poelstra `<scriptless@wpsoftware.net>`