1 Signature Aggregation

Notation

- $x, x_1, x_{2...}$ are private keys with corresponding public keys $X, X_1, X_2 ... X_i = x_i G$, with G the generator
- The message being signed is m
- $H()$ is a cryptographic hash function

Schnorr Signatures

- Signatures are $(R, s) = (rG, r + H(X, R, m)x)$ where r is a random nonce chosen by the signer
- Verification requires $sG = R + H(X, R, m)X$

Naive Schnorr multi-signatures

- Call X the sum of the X_i points
- Each signer chooses a random nonce r_i and shares $R_i = r_i G$ with the other signers
- Call R the sum of the R_i points
- Each signer computes $s_i = r_i + H(X, R, m)x_i$
- The final signature is (R, s) where s is the sum of the s_i values
- Verification requires $sG = R + H(X, R, m)X$, where X is the sum of the individual public keys

It is interesting to note that this satisfies the definition of a *key aggregation scheme*, as multiple parties can jointly produce a signature that is a valid single-key signature for the sum of the keys.

The issue arises in that this scheme is not secure. Consider the following scenario:

- Alice and Bob want to produce a multi-signature together.
- Alice has a key pair (x_A, X_A) and Bob has (x_B, X_B). However, nothing prevents Bob from claiming that his public key is $X'_B = X_B - X_A$.
- If he does so, others will assume that $X_A + X'_B$ is the aggregated key that Alice and Bob need to cooperate in order to sign for
- Unfortunately, that is equal to X_B, thus Bob can clearly sign for this by himself
- This is called a rogue-key attack

- One way to avoid this is requiring that Alice and Bob prove first that they actually possess the private keys corresponding to their claimed public keys; however this is not always possible
- Ideally a scheme needs to be constructed whose security does not rely on out-of-band verification of the keys.

2 Simple Schnorr Multi-Signatures

Here we consider a new Schnorr-based multi-signature scheme called MuSig, which is provably secure in the *plain public-model*. This means that signers are only required to have a public key, but they do not have to prove knowledge of the private key corresponding to their public key to some certification authority or to other signers prior to engaging the protocol.

This new scheme provides improvements to Bellare and Neven [**?**] and its variants by Bagherzandi *et al.* [**?**] and Ma *et al.* [**?**] in two respects:

1. It is simple and efficient, as it has the same key and signature size as standard Schnorr signatures;
2. It allows *key aggregation*, where the joint signature can be verified just as a standard Schnorr signature with respect to a single "aggregated" public key which can be computed from the individual public keys of the signers. [**?**]

3 The Discrete Logarithm Problem

Definition 1 (DL problem)[**?**]

- Let $\mathbb{G}$, p, g be group parameters- it is fixed, but the bit length k and $p$can be regarded as a security parameter if necessary.
- An algorithm $\mathcal{A}$ is said to $(t, \varepsilon solve$the DL problem with with respect to $\mathbb{G}, p, g$ if on input a random group element X, it runs in time at most t and returns $x \in \{0, ..., p-1\}$ such that

4 Elliptic Curve Digital Signature Algorithm

Currently in Bitcoin ECDSA is implemented. To sign a message m we hash it and treat this hash as a number: $z = hash(m)$. We also need a random or random-looking number k. We prefer not to trust random number generators (too many failures and vulnerabilities are related to bad random number generators) so we usually use RFC6979 to calculate deterministic k based on our secret and the message we are signing.

Using a private key pk we can generate a signature for message m consisting of two numbers: $r(x$coordinate of the random point $R = kG$ and $s = (z+rpk)/k$.

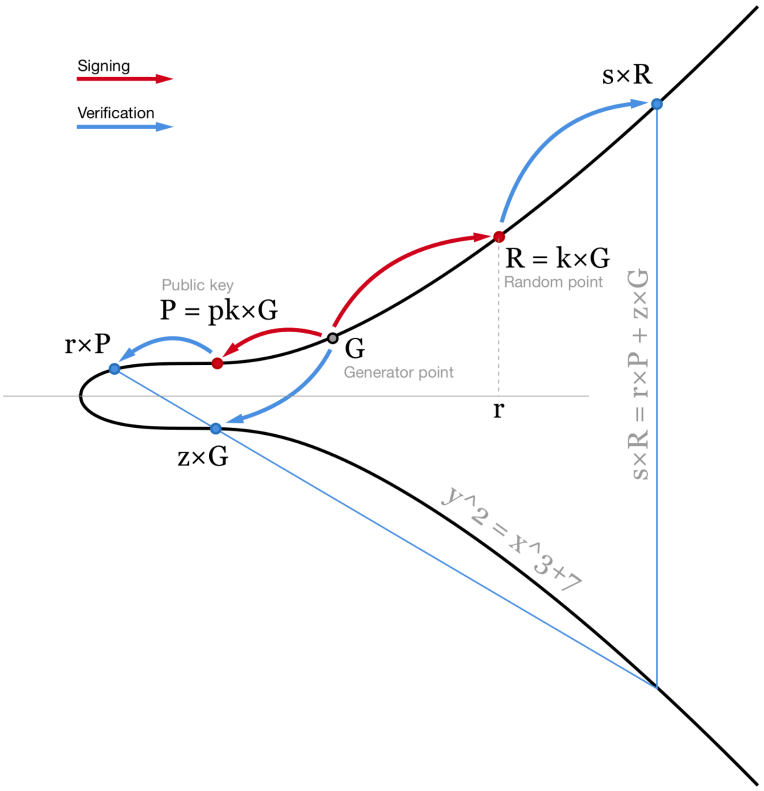

Figure 1: Visualization of the ECDSA algorithm. Elliptic curve is plotted over real number for illustrative purposes

Then, using our public key $P = pkG$ anyone can verify our signature by checking that point $(\frac{z}{s})G + (\frac{r}{s})P$ has $x$coordinate equal to r. This algorithm is very common, however it can be improved. Firstly, signature verification includes inversion $(1/s)$ and two points multiplications an these operations are very computationally heavy. In Bitcoin every node has to verify all the transactions. This means that when you broadcast a transaction, thousands of computers will have to verify your signature. Making verification process simpler will be very beneficial even if signing is more difficult.

Secondly, every node has to verify every signature individually. In the case of m-of-n multisig transaction node may even have to verify the same signature several times. For example, transaction of 7-of-11 multisig input will contain 7 signatures and require from 7 to 11 signature verifications on every node in the network. Also such transaction will take a huge amount of space in the block and you will have to pay large fees for that.[?]

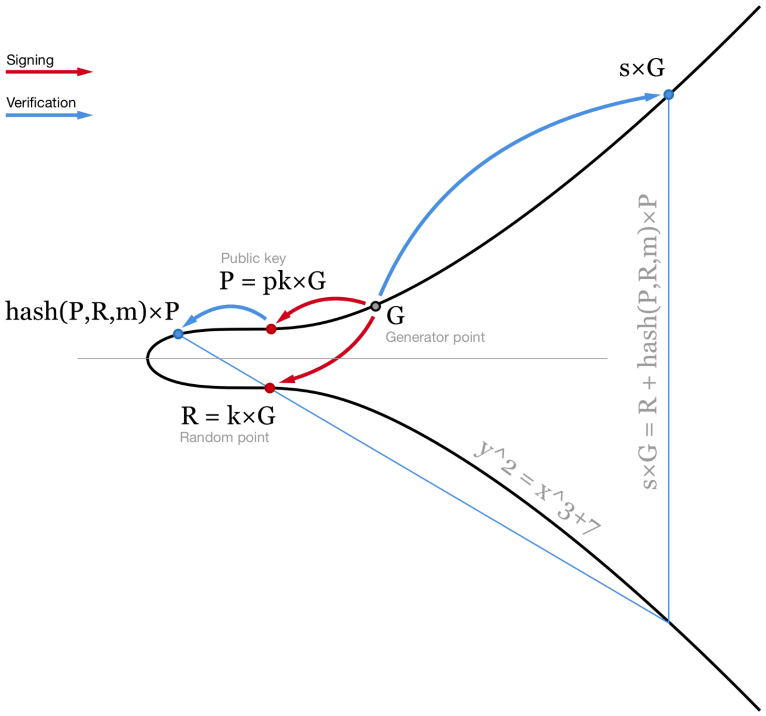

Figure 2: Visualisation of the Schnorr signature verification

5 Schnorr signatures

Schnorr signatures are generated slightly differently. Instead of two scalars (r, s) we use a point R and a scalar . Like ECDSA, R is considered a random point on the elliptic curve ($R = kG$). Second part of the signature is calculated slightly differently: $s = k + hash(P, R, m)pk$. Here pk is your private key, $P = pkG$ is your public key, m is the message. Then once can verify this signature by checking that $sG = R + hash(P, R, m)P$

This equation is linear, so equations can be added and subtracted with each other and still stay valid. This leads to a nice feature of Schnorr signatures.[**?**]

5.1 Batch validation

To verify a block in Bitcoin blockchain we need to make sure that all signatures in the block are valid. If one of them is not valid we don't care which one- we just reject eh whole block and that's it.

With ECDSA every signature has to be verified separately. Meaning that if we have 1000 signatures in the block we will need to compute 1000 inversions and 2000 point multiplication. In total approximately 3000 heavy operations.

With Schnorr signatures we can add up all the signature verification equa-

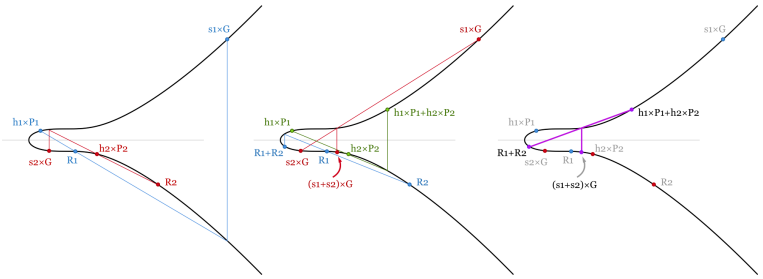

Figure 3: Batch validation of two signatures. As verification equation is linear the sum of several equations is valid as soon as all signatures are valid. We save some computational power as scalar and point additions are much easier than point multiplication.

tions and save some computational power. In total for a block with 1000 transactions we need to verify that:

$(s1 + s2 + ... + s1000)G = (R1 + ... + R1000) + (hash(P1, R1, m1)P1 + hash(P2, R2, m2)P2 + ... + hash(P1000, R1000, m1000)P1000$

Here we have a bunch of point additions (almost free in sense of computational power) and 1001 point multiplication. This is already a factor of 3 improvement- we meed to compute roughly one heavy operation per signature. [**?**]

5.2 Key aggregation

There is a need to keep one's bitcoin safe, so we might want to use at least two different private keys to control bitcoins. Once we will use on a laptop or a phone and another one- on a hardware wallet/cold wallet. So when one of them is compromised we still have control over our bitcoins.

Currently it is implemented via 2-of-2 multi-signature script. This requires two separate signatures to be included in the transaction.

With Schnorr signatures we can use a pair of private keys $(pk1, pk2)$ and generate a shared signature corresponding to a shared public key $P = P1+P2 = pk1G + pk2G$. To generate this signature we need to choose a random number on every device $(k1, k2)$, generate a random point $Ri = kiG$, add them up to calculate a common $hash(P, R1 + R2, m)$ and then get $s1$ and $s2$ from every device $(si = ki + hash(P, R, m)pki)$. Then we can add up these signatures and use a pair $(R, s) = (R1 + R2, s1 + s2)$ as our signature for shared public key P. Everyone else won't be able to say if it is an aggregated signature or not, it looks exactly the same as a normal Schnorr signature. [**?**]

There are three problems with this construction. First one, from UI point of view. To make a transaction we need several communication rounds, to calculate common R, and then to sign. With two private keys it can be done with a single access to cold wallet: we prepare an unsigned transaction. Then we transfer this data to the cold wallet and sign. As we already have R1 wa can sign transaction

on the cold wallet in one run. From the cold wallet we get R2 and s2 which previously chosen (k1, R1), combines both signatures and broadcasts a signed transaction. This is pretty much similar to what we have now, but as soon as you add a third provate key everything becomes more complicated. Imagine

5.3 Merkle multi-signatures

MuSig and key aggregation require all signers to sign a transaction. What happens if you want to make a 2-of-3 multisig? Can one use signature aggregation or will we have to use our usual OP_CHECKMULTISIG and separate signatures?

it is possible, but with a small change in the protocol. A new op-code similar to OP_CHECKMULTISIG can be developed that checks if aggregated signature corresponds to a particular item in the Merkle tree of public keys.

For example, if we use a 2-of-3 multisig with public keys $P1$, $P2$ and $P3$, then we need to construct a Merkle tree of aggregated public keys for all combinations we can use: $(P1, P2)$, $(P2, P3)$, $(P1, P3)$ and put the root in the locking script. To spend bitcoins we provide a signature and a proof that our public keys is in the tree. For 2-of-3 multisig there are only 3 elements in the tree and the proof will consist fo two hashes-the one we want to use and its neighbor. For a 7-of-11 multisig there will be already 11!/7!/4!=300 possible key combinations and the proof will require 8 elements. In general the number of elements in the proof scales almost linear with the number of keys in multisig (its $log2(n!m!/(n-m)!)$.

But with the Merkle tree of public keys we are not limited to $m-of-n$ multisignatures. We can make a tree with any public keys we want. For example , if we have a laptop, a phone, a hardware wallet and a recovery seed, we can construct a structure that would allow us to spend bitcoins with a laptop and a hardware wallet, a phone and a hardware wallet or just with a recovery seed. This is currently not possible just with OP_CHECKMULTISIG- only if you construct much more complicated script with branches.

6 Boneh-Lynn-Shacham Signatures

Boneh-Lynn-Shacham signature scheme is based on the computational Diffie-Hellman assumption on certain and hyper-elliptic curves. The signature length is half the size of a DSA signature for a similar level of security.

Schnorr signatures combine all signatures and public keys in the transaction into a single key and a signature and nobody will find out that they correspond to multiple keys. In addition block validation is faster, as all signatures can be validated at once. However there are a few problems with Schnorr signatures:

- Multisig schemes require several rounds of communication. This can be a hindrance with regards to cold storage
- With signature aggregation we have to rely on random number generator-we can't choose random point R deterministically like we do in ECDSA

- m-of-n multisig scheme is tricky- we need to make a merkle tree of public keys that can get pretty large for large m of n
- We can't combine all signatures in the block to a single signature

BLS signatures can fix all of the above. We don't need random numbers at all, all signatures in the block can be combined to a single signature, m-of-n multisig is very simple and we don't need several communication rounds between signers. In addition to that BLS signatures are 2 times shorter than Schnorr or ECDSA-signature is not a pair, but a single curve point.[**?**]

6.1 Hashing to the curve

Normally with ECDSA and Schnorr we hash the message and use this hash in the signing algorithm as a number. For BLS signatures we need a slightly modified hashing algorithm that hashes directly to the elliptic curve. The easiest way is to hash a message as usual and treat the result as an x-coordinate of a point. Elliptic curves usually have about 2^{256} points and SHA-256 hashing algorithm also gives a 256-bit result. But for every valid x-coordinate there are two points with positive and negative y-coordinate (just because if (x, y) is on the curve $y^2 = x^3 + ax + b$ then $(x, -y)$ is also on the curve). This means that our hash has roughly 50% probability to find two points for some $x$and 50% to find none.

To find a point for any message we can try hashing several times by appending a number to the hash and incrementing it on fail. [**?**]

6.2 Key aggregation and $n - of - n$multisignature

If we are using multisignature addresses, we are signing the same transaction with different keys. In this case, we can do key aggregation like in Schnorr, where we combine all signatures and all keys to a single pair of a key and a signature. If we consider a common 3-of-3 multisig scheme:

A simple way to combine them is to add all the signatures and all the keys together. The result will be a signature $S = S1 + S2 + S3$ and a key $P = P1 + P2 + P3$. It is easy to see that the same verification equation still works:

$e(G, S) = e(P, H(m))$

$e((G, S) = e(G, S1 + S2 + S3) = e(G, (pk1 + pk2 + pk3)H(m)) = e((pk1 + pk2 + pk3)GH((m)) = e(P1 + P2 + P3, H(m)) = e(P, H(m))$

Similarly to Schnorr there needs to be protection against rogue key attacks. This can be achieved by asking every co-signer to prove that they have private keys for their public keys (by signing their public keys), or that some nonlinearity to the scheme is added making rogue key attacks impossible. Instead of summing up all the keys and signatures, we multiply them by a certain number and then add:

$S = a1S1 + a2S2 + aS3$

$P = a1P1 + a2P2 + a3P3$

Here coefficients of the signatures and keys are calculated deterministically from the public key of the signer and all other public keys:
$ai = hash(Pi, \{P1, P2, P3\})$ [?]

6.2.1 The benefits of key aggregation

- If a group of $n$signers want to authorize which all of them agree, but do not necessarily wish to reveal their individual public keys
- They can privately compute the aggregated key $\tilde{X}$ corresponding to their multi-set of public keys and publish it as an ordinary (non-aggregated) key.
- Signers are ensured that all of them will need to cooperate to produce a signature which is valid under $\tilde{X}$, whereas verifiers will not even learn that $\tilde{X}$is in fact an aggregated key.
- Moreover, $\tilde{X}$can be computed by a third party just from the list of public keys, without interacting with the signers.
- This property will prove instrumental for obtaining a more compact and privacy-preserving variant of so-called n-of-n multi-signature transactions in Bitcoin.

Two variants of the BN multi-signature scheme have been previously proposed.

Today Bitcoin uses ECDSA signatures [?] over the `secp256k1` curve [?] to authenticate transactions. As Bitcoin nodes fully verify all transactions, signature size and verification time are important design considerations while signing time is much less so. Besides, signatures account for a large part of the size of Bitcoin transactions. Because of this, using multi-signatures seems appealing. However, designing multiparty ECDSA signature schemes is notably cumbersome [?][?][?]due to the modular inversion involved in signing, and moving to Schnorr signatures would definitely help deploy compact multi-signatures.

6.3 Introduction

Bitcoin contains every transaction since the system's inception, resulting in a final state, the set of unspent coins. Each unspent coin has an associated value (expressed as a multiple of the currency unit, 10^{-8} bitcoin) and a programmable public key of the owner. Every transaction consumes one or more coins, providing a signature for each to authorize its spending, and creates one more new coins, with a total value not larger than the value of the consumed coins.

Bitcoins uses a programmable generalization of a digital signature scheme. Instead of a public key, a predicate that determines spend-ability is included in every output (implemented in a concise programming language, called *Bitcoin Script).* When spending, instead of a signature, a witness that satisfies the predicate is provided. In practice, most output predicates effectively correspond to a single ECDSA verification. This is also how Bitcoin supports a naive version

of multi-signatures with a threshold policy: coins can be assigned a predicate that requires valid signatures for multiple public keys. Several use cases for the exist, including low-trust escrow services [**?**] and split-device security. While using the predicate language to implement multi-signatures is very flexible, it is inefficient in terms of size, computational cost, and privacy.

As a global consensus system, kept in check by the ability for every participant to validate all updates to the ledger, the size of signatures and predicates, and the computational cost for verifying them are the primary limiting factors for its scalability. The computational requirements for signing, or the communication overhead between different signers are far less constrained. Bitcoin does not have any central trusted party, so it is not generally possible to introduce new cryptographic schemes that require a trusted setup. Finally, to function as a currency, a high degree of fungibility and privacy is desirable. Among other things, this means that ideally the predicate of coins fo not lead information about the owner. In particular, if several styles of predicates are in use, the choice may reveal what software or service is being used to manage it.