Politecnico di Torino

Dipartimento di Scienze Matematiche "G. L. Lagrange"

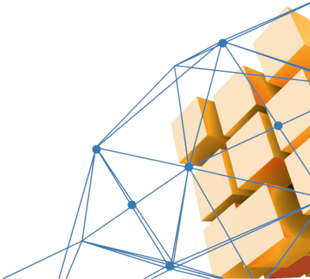
DISMA ECCELLENZA 2018·2022

Schnorr and Taproot: an overview of the new Bitcoin update

Andrea Gangemi, Alessandro Guggino

CrypTO, BIT PoliTO

1 December 2021



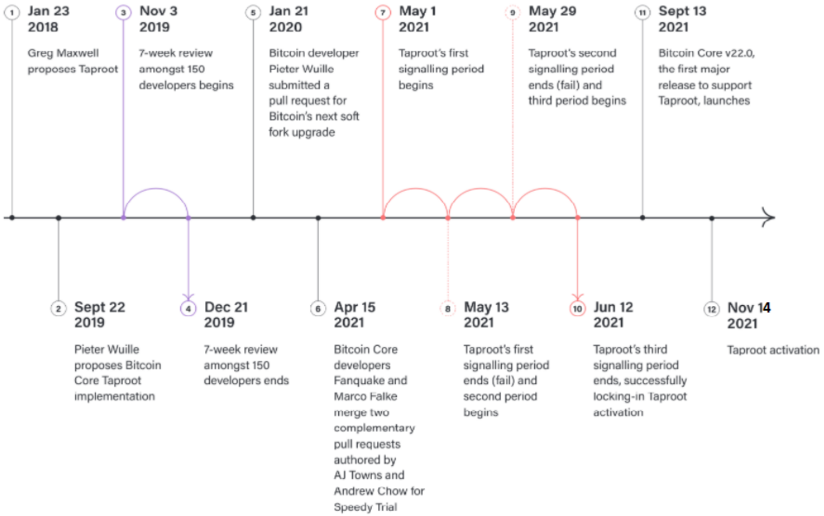
An Overview of the Update

The Taproot update consists of three different *Bitcoin Improvement Proposals* (BIPs):

- ***BIP340*** - ***Schnorr***: a new digital signature.
- ***BIP341*** - ***Taproot***: more privacy, efficiency, and flexibility of Bitcoin's scripting capabilities.
- ***BIP342*** - ***Tapscript***: an upgraded scripting language that complements Schnorr and Taproot.

Taproot has been deployed as a backward-compatible soft fork.

Taproot Timeline



1. **Jan 23 2018** Greg Maxwell proposes Taproot
2. **Sept 22 2019** Pieter Wuille proposes Bitcoin Core Taproot implementation
3. **Nov 3 2019** 7-week review amongst 150 developers begins
4. **Dec 21 2019** 7-week review amongst 150 developers ends
5. **Jan 21 2020** Bitcoin developer Pieter Wuille submitted a pull request for Bitcoin's next soft fork upgrade
6. **Apr 15 2021** Bitcoin Core developers Fanquake and Marco Falke merge two complementary pull requests authored by AJ Towns and Andrew Chow for Speedy Trial
7. **May 1 2021** Taproot's first signalling period begins
8. **May 13 2021** Taproot's first signalling period ends (fail) and second period begins
9. **May 29 2021** Taproot's second signalling period ends (fail) and third period begins
10. **Jun 12 2021** Taproot's third signalling period ends, successfully locking-in Taproot activation
11. **Sept 13 2021** Bitcoin Core v22.0, the first major release to support Taproot, launches
12. **Nov 14 2021** Taproot activation

Source: Kraken Intelligence, GitHub, CoinDesk

A digression: how was the upgrade locked in?

The miners are the people which must signal support for any Bitcoin protocol update: they are saying that they are prepared to run a certain version of the software which implements the updated code.

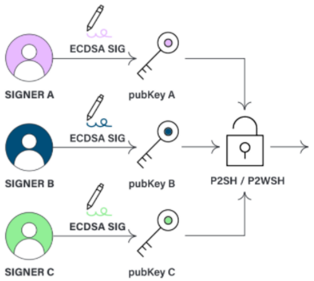
The upgrade has been locked in using a process called ***speedy trial***.

- A *signaling period* (or *epoch*) is a window of 2016 blocks.
- To signal support for Taproot, miners put a "4" at the end of the *version bits* field in the block that they mined, during the signaling period.
- If at least the 90% of the blocks (i.e., 1815 blocks) during the epoch had a version bits field which ended with "4", then Taproot was successfully locked in.
- If this strategy had failed for six consecutive epochs, Taproot would not have been added to the protocol.

Recap: ECDSA

The original digital signature used by the Bitcoin protocol is **ECDSA**. However, it has some limitations:

- The best known results for the provable security of ECDSA rely on stronger assumptions with respect to other digital signature schemes (e.g., Schnorr).
- It does not (easily) allow the aggregation of signatures: P2SH and P2WSH transactions take a lot of space to save every signature separately.



BIP340 - Schnorr

Schnorr Signatures for secp256k1

- The ***Schnorr signature*** is a digital signature algorithm which was described by Claus Schnorr in 1989.
- This signature scheme was patented until February 2008.
- The scheme is known for its simplicity, since it is *linear*, and its security is based only on the supposed intractability of the Discrete Logarithm problem.

The Schnorr's signature can be computed on a Short Weierstrass elliptic curve: in particular, it can be computed on the Bitcoin curve *secp256k1*.

Schnorr Signatures

- *KEY GENERATION*
  - Fix an elliptic curve E on a finite field $\mathbb{F}_q$. Let N be the order of the curve.
  - Fix a generator G and a Hash Function h.
  - Every user chooses his *secret key* d, $0 < d < N$, and computes his *public key* P_x as the x-coordinate of the point $P = dG$.
- *SIGNING*
  - Let M be the message.
  - The signer (A) chooses an integer k, $1 < k < N$ and computes $R = kG = (R_x, R_y)$.
  - A computes $c = h(R_x || (P_x)_A || M)$.
  - A computes $s = k + d_A c \mod N$.
  - The signature is the couple (R_x, s).

Schnorr Signatures

- *VERIFICATION*
  - The recipient (B) computes the point sG.
  - B computes the hash $c = h(R_x||(P_x)_A||M)$.
  - If $sG = R + cP_A$, the signature is valid.

This works because

$$sG = (k + d_A c)G = kG + c(d_A G) = R + cP_A.$$

This signature has a length of 64 bytes (32 bytes for the x-coordinate of R, 32 bytes for the number s). The signature is about 10% shorter than a ECDSA one.

To recover the y coordinate of the point, we implicitly choose the one that is even.

Schnorr Applications

The Schnorr signature allows several interesting applications:

- **Multisignatures**.
- Batch verification of signatures.
- Adaptor signatures.
- Blind signatures.

Multisignature Schemes

The Schnorr signature allows the aggregation of public keys into a single public key which users can jointly sign for. This enables n-of-n multisignatures which, from a verifier's perspective, are no different from ordinary signatures.

Multisignature schemes which are compatible with Schnorr are:

- **MuSig**.
- **Musig2**.
- MusigDN.
- FROST (threshold signature scheme t-of-n, with $t < n$).

MuSig

The **MuSig** scheme utilizes the same key generation algorithm used by the Schnorr scheme. Suppose there are n users involved, then:

- *SIGNING*
  - Let $L = h(P_1 || \dots || P_n)$. Every user computes the quantity $a_i = h(L || P_i)$.
  - Let $\tilde{X} = \sum_{i=1}^{n} a_i P_i$: $\tilde{X}$ is a public parameter.
  - Every user chooses r_i and computes $R_i = r_i G$. Then, he sends to everyone else the commit $t_i = h(R_i)$.
  - Upon reception of the other $n-1$ commitments, every user sends R_i to the other users.
  - Upon reception of the other $n-1$ R-points, every user checks if effectively $t_i = h(R_i)$ holds for $i \in \{1, \dots, n\}$.

MuSig

- Every user computes the point $R = R_1 + \ldots + R_n = (R_x, R_y)$.
- Every user computes $c = h(\tilde{X}_x || R_x || M)$.
- Every user computes $s_i = r_i + c d_i a_i \mod N$, then the aggregate is $s = s_1 + \ldots + s_n \mod N$.
- The signature is the couple (R_x, s).

- *VERIFICATION*
    - Check if $sG = R + c\tilde{X}$.

The verification step can be performed without knowing every single public key: we just need the aggregate $\tilde{X}$.

MuSig2

The main limitation of MuSig is the number of communication rounds (three).

- **MuSig2** is a novel scheme proposed in 2020 which removes the preliminary commitment phase, so that signers start right away by sending the points $R_1, \ldots, R_n$.
- To date, this protocol is the only one that is considered secure, supports key aggregation, outputs ordinary Schnorr signatures and needs only two communication rounds.
- The price to pay is a stronger cryptographic assumption.

The key generation and the verification algorithms are unchanged. Suppose there are n signers.

MuSig2

- SIGNING
  - Compute L, $a_i = h(L||P_i)$ and finally $\tilde{X}$ as before.
  - Each user i chooses ν different nonces $r_{i,1}, \dots, r_{i,\nu}$ and computes the points $R_{i,j} = r_{i,j}G, \quad \forall j \in \{1, \dots, \nu\}$.
  - All these R points are public. Compute $R_j = \sum_{i=1}^{n} R_{i,j} \quad \forall j \in \{1, \dots, \nu\}$.
  - Compute the hash $b = h(\tilde{X}_x||R_{1_x}||\dots||R_{\nu_x}||M)$, then compute $R = \sum_{j=1}^{\nu} b^{j-1} R_j$.
  - Compute the hash $c = h(\tilde{X}_x||R_x||M)$, then every user computes $s_i = ca_i d_i + \sum_{j=1}^{\nu} r_{i,j} b^{j-1} \mod N$.
  - Compute $s = s_1 + \dots + s_n \mod N$. The signature is the couple (R_x, s).

For the applications, usually $\nu = 2$ or $\nu = 4$.

Our contribution

- BIT PoliTO's cryptography team has implemented all three digital signatures.
- In particular, the Schnorr signature implementation passes all the vector tests provided in the official BIP-340 page.
- The code is open source and can be downloaded from the following GitHub page.
- Everyone can generate signatures using the Jupyter Notebook provided in the above link! Soon, the same tool will be directly available on the official BIT PoliTO website.

Recap: Bitcoin Script

Bitcoin uses **Script**, a *non-Turing-complete* programming language that sets instructions on how to spend bitcoins.
When sending bitcoins, users must define a locking script on each output in the transaction, while the recipient must then execute an unlocking script to spend the bitcoins.

Elements:

- **Data**: information needed for the transaction to occur (e.g. digital signature and public key).
- **Opcodes**: commands that operate on the data, allowing to create complex spending conditions (or smart contracts).

Example: locking script = *3 OP_ADD 5 OP_EQUAL*

$\rightarrow x + 3 = 5$

unlocking script = x = 2

Recap: Bitcoin Outputs

Although there are many different locking scripts combining various opcodes, most outputs relay on standard scripts.

- **Pay-to-Public-Key Hash (P2PKH)**:
  It contains an address derived from the hash of a public key and it may be unlocked by a signature and a public key (to verify the address and check the signature).
- **Pay-to-Script-Hash (P2SH)**:
  It contains the digest of a script (with different spending conditions than a key and a signature). To unlock it, the data needed to satisfy the conditions and the actual script must be provided.

There are also the SegWit versions of these output types: **P2WPKH** and **P2WSH**.

BIP 341 - Taproot

SegWit version 1 spending rules

The idea behind Taproot is to combine the traditionally separate **pay-to-public-key** and **pay-to-script** output types into one type of output called ***pay-to-taproot***.

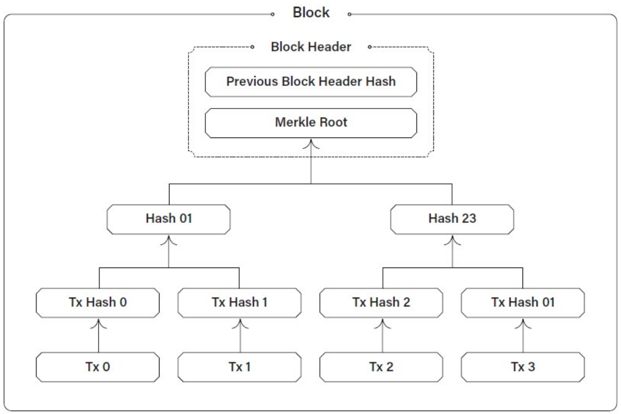
Coins protected by Taproot may be spent either by satisfying one of the committed scripts or by simply providing a signature that verifies against the public key.

Taproot is intended for use with:

- Schnorr signatures that simplify multiparty construction (e.g. using *MuSig*).
- MAST to allow committing to more than one script, any one of which may be used at spend time.

Merkle Tree

Merkle trees are a *hash-based data structure* where each node contains the hash of its children hashes, and the leaf nodes contain the hashes of the actual data being committed to it.



Source: Kraken Intelligence

MAST

Merklized Alternative Script Tree

MAST is a method of using a Merkle tree to store various **user-selected spending conditions** organized into the leaves of a binary tree, that can be a balanced tree if each condition is equally likely.
Otherwise, a Huffman tree can be constructed.
This allows the spender to select which one of the conditions they will fulfill without having to reveal the details of other conditions to the blockchain.

Advantages:

- Larger contracts.
- Lower fees.
- Improved privacy.
- Improved fungibility.

Tweaked Public Key

It is possible to combine the *pay-to-public-key* and *pay-to-script* output types by using a **tweaked public key**:

$$Q = P + H(P,m)G$$

- ***P*** is a *inner public key* which can spend the output.
- ***m*** is a *Merkle root of a MAST* which contains script conditions under which the output can be spent.
- ***H(P,m)G*** is a *public key* generated from the hash and the generator point of the Bitcoin curve (secp256k1).

If no single (nor aggregated) public key is permitted to spend an output, then a provably unknown public key is used.
If no scripts are permitted to spend an output, then the hash is calculated without m instead of omitting it.

Tweaked Public Key

Keypath or Scriptpath

$$Q = P + H(P,m)G$$

- When spending a Taproot output using the **keypath** (the *pay-to-public-key* condition):

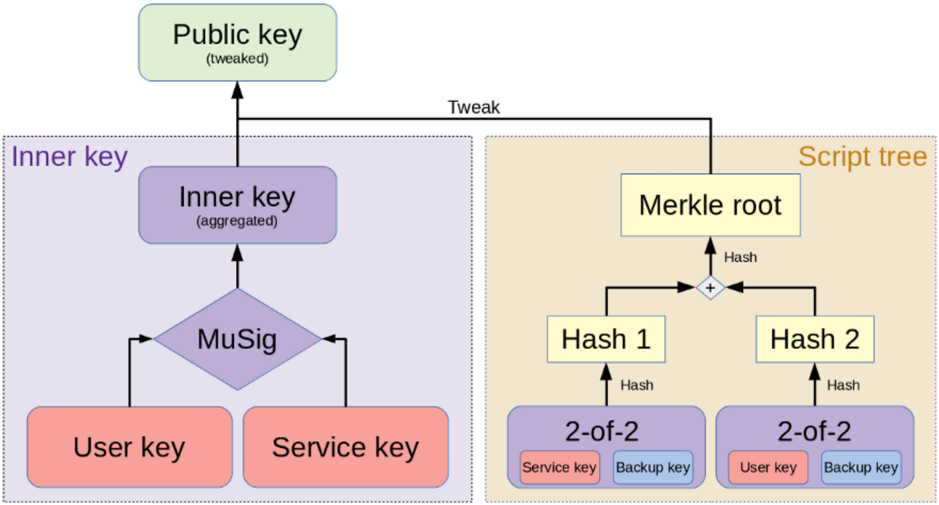
  You simply provide a digital signature for the public key ***Q***, since you know ***H(P,m)*** and the private key of ***P***. So you can sign using the sum of these two private keys.

- When spending using the **scriptpath** (the *pay-to-script* condition):

  You reveal ***P*** and ***m*** allowing validators to see that ***Q*** does indeed commit to ***m***, and then you provide a Merkle proof that a specific script is committed to by ***m***.

Tweaked Public Key

An example



Pay-to-Taproot (P2TR)

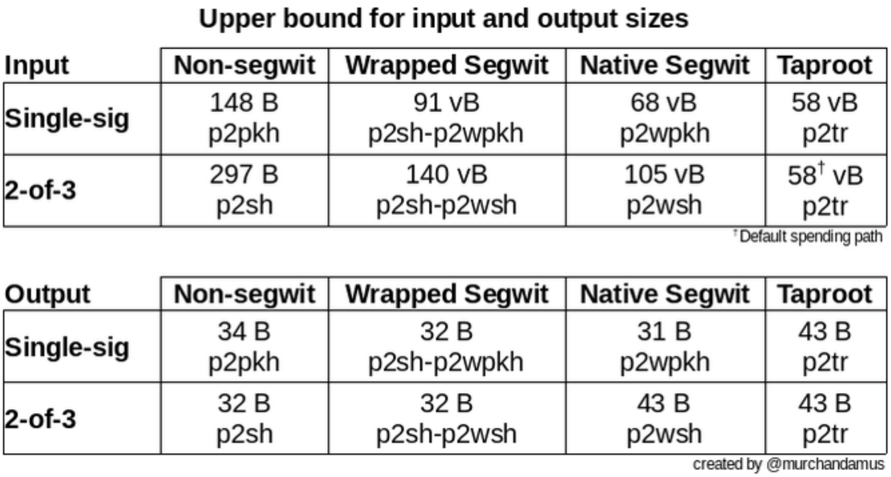
Pay-to-Taproot (**P2TR**) funds are locked to a single public key similarly to Pay-to-Public-Key (**P2PK**) outputs.

P2TR (Native SegWit v1) will be the first output type that uses the *bech32m* address encoding, an updated version of *bech32* which was used for **P2WPKH** and **P2WSH** (Native SegWit v0).

All the outputs look identical on-chain, from the opening of a Lightning Network channel (with HTLC or PTLC) to a transaction with multi-signature or with a complex script.
... and this means **more privacy**!

Pay-to-Taproot (P2TR)

Comparisons



Upper bound for input and output sizes

Input	Non-segwit	Wrapped Segwit	Native Segwit	Taproot
Single-sig	148 B p2pkh	91 vB p2sh-p2wpkh	68 vB p2wpkh	58 vB p2tr
2-of-3	297 B p2sh	140 vB p2sh-p2wsh	105 vB p2wsh	58† vB p2tr

† Default spending path

Output	Non-segwit	Wrapped Segwit	Native Segwit	Taproot
Single-sig	34 B p2pkh	32 B p2sh-p2wpkh	31 B p2wpkh	43 B p2tr
2-of-3	32 B p2sh	32 B p2sh-p2wsh	43 B p2wsh	43 B p2tr

created by @murchandamus

BIP 342 - Tapscript

Validation of Taproot Scripts

Tapscript is the scripting language used for Taproot outputs.

It shares most operations with legacy and SegWit Bitcoin Script but has a few differences:

- Opcodes **OP_CHECKSIG** and **OP_CHECKSIGVERIFY** are modified to verify Schnorr signatures.
- Opcodes **OP_CHECKMULTISIG** and **OP_CHECKMULTISIGVERIFY** are replaced by **OP_CHECKSIGADD**.
- Many previously disabled opcodes are redefined to be **OP_SUCCESS** opcodes. They allow introducing new opcodes more cleanly.

References

- https://en.bitcoin.it/wiki/BIP340
- https://en.bitcoin.it/wiki/BIP341
- https://en.bitcoin.it/wiki/BIP342
- https://kraken.docsend.com/
- https://lists.linuxfoundation.org/pipermail/bitcoin-dev
- https://eprint.iacr.org/2018/068.pdf

References

- https://eprint.iacr.org/2020/1261.pdf
- https://github.com/BITPoliTO/schnorr-sig
- https://bitcoinops.org/en/topics/taproot/
- https://bitcointechtalk.com/
- https://suredbits.com/the-taproot-upgrade/
- https://murchandamus.medium.com/

Grazie per l'attenzione

Politecnico
di Torino