Taproot

(und Schnorr)

Coinfinity, 17.02.2021

Warum Taproot?

- **Scaling**
  - MultiSigs und Contracts brauchen viel weniger Platz
  - Block Validation wird schneller
- **Privacy (Fungibility)**
  - Alle utxo’s (und die meisten spent outputs) werden immer gleich aussehen!
    (egal ob single key, lightning channel opening, close, etc..)
- **Neue Script Features**
  - Sehr große m von n MultiSig Setups möglich
  - Größere und komplexere Scripts möglich
  - bessere Upgradability

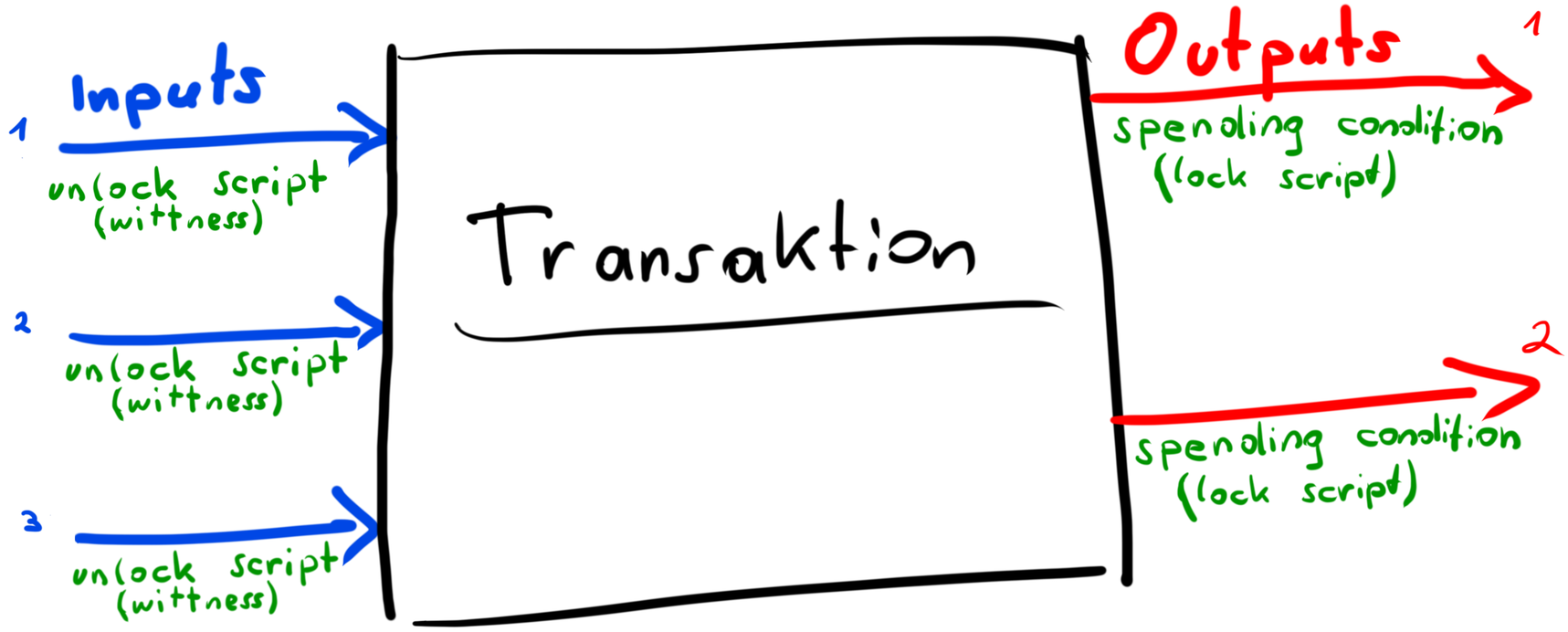
Was ist Taproot?

- Neue Version von „Output Scripts”

- Was sind Output Scripts? —> „Spending Conditions”

Bitcoin Transaktionen

- Inputs (die Outputs einer früheren Tx ausgeben)
- Outputs (die neue Spending Conditions definieren)



Recap: Spending Conditions

- Früher Bitcoin Script (jetzt *witness program*)
- Im Output: Konditionen, um eine Coin ausgeben zu können (`scriptPubKey` “locking script”)
- Und im Input: Daten, um zu beweisen dass diese Konditionen erfüllt sind (`scriptSig / txinwitness` „unlocking script”)
- Beim Validieren landet beides gemeinsam am Stack und wird als ein Skript ausgeführt. Das Ergebnis muss positiv exekutieren:

Recap: Spending Conditions

- Wie sehen so Conditions in der Regel aus? Beispiele:

```
04a9d6840fdd1497b3067b8066db783acf90bf42071a38fe2
cf6d2d8a04835d0b5c45716d8d6012ab5d56c7824c39718f7
bc7486d389cd0047f53785f9a63c0c9d OP_CHECKSIG
```

Pay to public key (P2PK)

```
OP_HASH160 10b0ed2d1698ff0f0ea151cf41988d05b728746a OP_EQUAL
```

Pay to script hash (P2SH)

```
OP_DUP OP_HASH160 fde0a08625e327ba400644ad62d5c571d2eec3de OP_EQUALVERIFY OP_CHECKSIG
```

Pay to public key hash (P2PKH)

```
OP_1 0378ee11c3fb97054877a809ce083db292b16d971bcdc6aa4c8f92087133729d8b 1283b5fbf5
cc62d4399dfa1025c3e306295264494722c5085ceadadf1291f68125 a31752c9f17c628edc4c69c4c
0846f8d814b21e046eabe06f9968a037ce0741c74 OP_3 OP_CHECKMULTISIG
```

Pay to multisig

```
OP_RETURN 68656c6c6f20776f726c64
```

Null data (Unspendable)

Wie wird Taproot aussehen?

Was ist ein Taproot Output?
(in der Transaktion)

- Ein (native) SegWit Output
- Allerding „witness program” mit einer neuen **Version 1** (aktuell ist 0)
- Das Script (witness Programm) eines solchen **P2TR** - „Pay to Taproot” Outputs wird on-chain so aussehen:

Output Script: `01 <32-bytes pubkey>`

SegWit program
Version Byte

Thats’ it! ALLE Taproot Outputs sehen IMMER so aus, EGAL welch komplexen Scripts dahinter stehen mögen.

Magie? Nein, Taproot! :)

Was ist ein Taproot Output?
(aus User-Sicht)

- Taproot Outputs sind SegWit Outputs
- BIP 173 (bc1.. Adressformat) beinhaltet schon das witness script Version byte
- Taproot Outputs sind version 1 SegWit Programs (aktuell ist version 0)
- —> Dh: für den End-User sind das einfach etwas längere bc1… Adressen

- (nur native SegWit Outputs, keine P2SH wrapped segwit, also KEINE 3er SegWit Adressen)

Kurzer Vergleich SegWit v0 vs. v1 Adressen

- SegWit v0 Script (für bisherige P2WPKH Outputs)

```
v0 Script:     00  <20-byte pub key hash>
v0 witness:    <DER-encoded ECDSA signature> <33 bytes public key>

v1 Script:     01  <32-byte public key>    --> bc1.. Adresse wird länger
v1 witness:    <64 byte signature>   (key spending path)
```

- Notizen:
  - DER-encoded ECDSA signatures können bis zu 72 bytes lang sein (variabel)
  - wittness Blockspace zahlt der Empfänger (der den Coin später ausgeben will)
  - Script Blockspace zahlt der Sendende (der die locking conditions in den Output schreibt)
  - Signatur ist 64 Byte weil: 32B x-coordinate von „R” (man lässt das even/odd byte weg und nimmt immer even (02) an), und 32B scalar „s”
  - Signatur wird optional 65 byte lang, wenn ein Sighash-Flag verwendet wird (SIGHASH_ALL ist jetzt implizit)

Bausteine und Grundideen

- SegWit Script versionierung
- MAST (merkelized abstract syntax tree)
- Schnorr Signaturen

Baustein 1: SegWit

Recap: SegWit

- Locking script bei SegWit „**witness program**”

- **witness program** ist fast dasselbe wie Bitcoin Script
- wichtiger Unterschied: prefixed mit **VERSIONSNUMMER**
  - —> das ermöglicht überhaupt erst das Ausrollen neuer Features

Baustein / Idee 2 : MAST

(Merkelized Abstract Syntax Tree)

Recap: Spending Conditions

- Es gibt ja auch komplexere Conditions mit mehreren möglichen Pfaden:

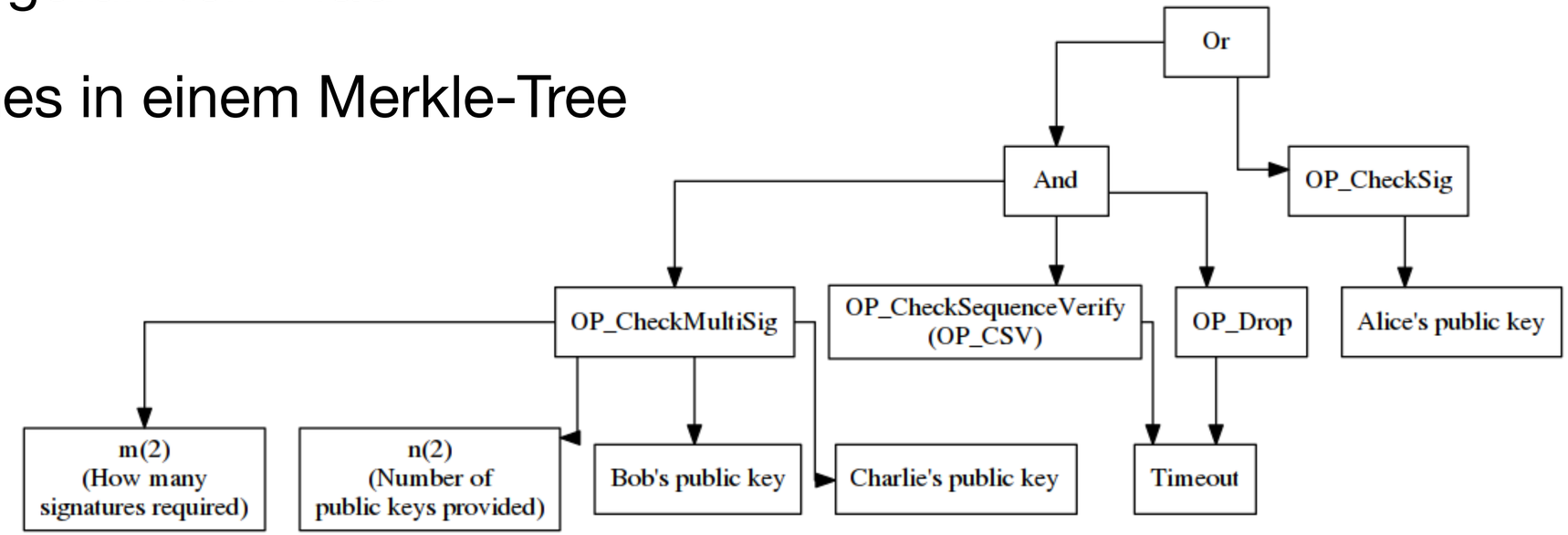
```
OP_HASH160 <revocationHash> OP_EQUAL
OP_IF
    <bobPubKey>
OP_ELSE
    <timeDelay> OP_CHECKSEQUENCEVERIFY OP_DROP
    <alicePubKey>
OP_ENDIF
OP_CHECKSIG
```

- Wenn der Output unlocked wird, landet nur die Erfüllung einer dieser Conditions on-chain, aber trotzdem sehen alle Nodes das ganze Script, müssen es speichern, verarbeiten, etc.

Nachteile: Onchain-Platzverbrauch, Privacy

Idee: MAST

- MAST (merkelized abstract syntax tree)
- IDEE:
  - Zeige nur den ausgeführten Pfad
  - Hashe die Branches in einem Merkle-Tree



Hint: Das hier ist nicht Taproot. Taproot macht das ein bißchen anders. (dazu später)

Hintergrund: MAST

- Idee gibt’s schon lange
  2013 bitcointalk: https://bitcointalk.org/index.php?topic=255145.msg2757327#msg2757327
- Umsetzungsvorschlag BIP 114 („Merkelized Abstract Syntax Tree”) (rejected)
- **Vorteile**:
  - Privacy (nur die erfüllte Bedingung wird publik)
  - Onchain Platzverbrauch
  - Komplexere Scripts werden möglich (Script-Size und OP-Code Limits)
- **Nachteil**:
  - Macht für Privacy nur Sinn wenn alle Scripts so aussehen (anonymity set sonst nicht sehr groß)

Baustein 3 : Schnorr-Signaturen

Schnorr

die BESSERE ecdsa Signatur :)

- Neues Signaturschema (statt ECDSA)
- Nutzt dieselbe elliptic curve und gleiche private keys wie ECDSA
  (also kompatibel mit existierenden Keys)
- **In JEDER Hinsicht besser als ECDSA !!**
- Kleinere Signaturen, schnellere Validierung, klar definiertes Format, coole Features (weil lineare Validierung)
- Selbe Sicherheitsannahmen wie ECDSA
  (wenn das sicher ist, ist Schnorr auch sicher, Schnorr ist sogar simpler)

Mathe Details Schnorr

Signaturen

G	EC generator point
H()	Hash-Funktion
k	random nonce
d	mein private Key
R	random nonce point (k*G)
P	mein public key (d*G)
m	Message

Kleinbuchstaben sind Skalare („Zahlen”)
Großbuchstaben sind Punkte auf der Kurve (x und y Koordinaten)
** ist Punkt-Multiplikation auf der Kurve*
| ist Konkatenierung von Byte-Arrays

Schnorr-Signatur erstellen:

```
e = H(R|P|m)
s = k + ed
```

```
Sig:   (s , k*G)
bzw.: (s , R)
```

Schnorr-Signatur validieren:

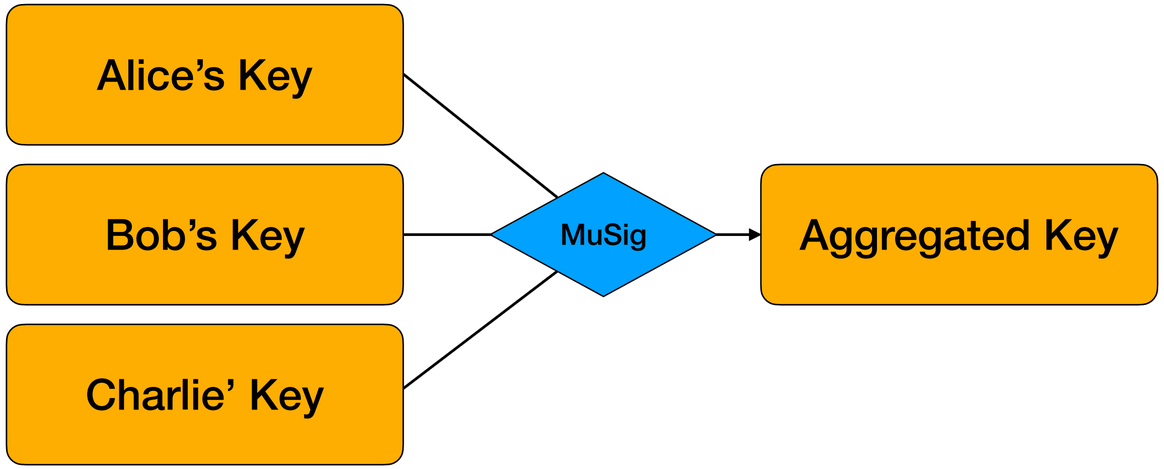
```
s*G  =  k*G + ed*G
s*G  =  R   + e*P
```

```
—> wenn == dann valide!
```

Schnorr Features

Key Aggregation

- Man kann mehrere Keys aggegrieren -> 1 pubkey
- man kann die Signaturen aggregieren (m von m)
- Ergebnis ist 1 Pubkey und 1 Signatur, nicht unterscheidbar von single-key Signatur



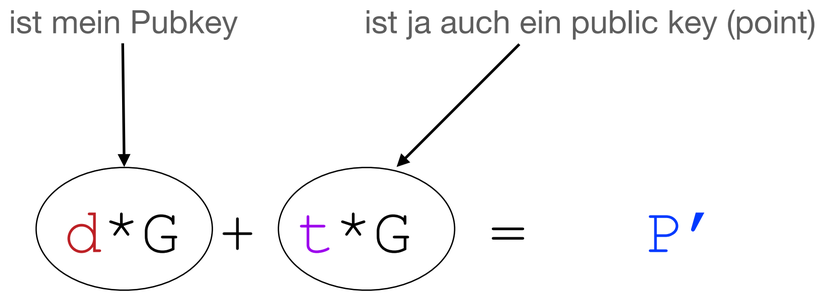
Hinweis: Schnorr ermöglicht zwar direkt das Addieren der Signaturen und von Keys, allerdings gibt’s ein paar Pitfalls, wenn man’s einfach straightforward macht (*Stichwort „key cancellation attack”*), deswegen ein leicht adaptiertes Protokoll. Und das Protokoll nennt sich „MuSig”.

Schnorr Features

Tweak Signatures

- Man kann zu einem Key beliebige Daten („tweak” **t**) addieren und mit diesem Key ganz normal signieren:

- Key: d (private) P (public) $d*G = P$
- Tweaked key (vereinfacht): $(d + t) * G = d*G + t*G = P'$



Dh. man kann **entweder**:

den P’ key nutzen und so tun als wäre das ein normaler Key

Oder: man kann revealen, dass man sich zu „t” committed hatte

indem man den den eigenen „echten” pubkey P und t offenlegt

Dann kann jeder verifizieren:
P + t*G = P’
und es ist klar ersichtlich, dass ich dieses t bereits bei der Erstellung der Signatur gekannt haben musste (die Signatur „commited” auf das t)

Tapscript

Tapscript

- Ist ein upgedatetes Bitcoin Script:
  - Verwendet Schnorr statt ECDSA
  - Ermöglicht auch Updates und zukünftige Versionen *(TapLeaf Version Byte, „upgradeable” bisher non-defined OP_Codes beenden das Script mit SUCCESS nicht fail, anders als NOP bisher)*
  - Multisig OP_CODES entfernt und ersetzt durch CHECKSIG_ADD Op-Codes (ermöglichen Batch-Verifizierung, siehe CHECKSIG_ADD Explainer)
  - Tapscripts werden in einem Key „Tweak” committed (siehe Tweaks von Schnorr weiter oben)

Recap: Was ist ein Taproot Output?

Output Script: `01 <32-bytes pubkey>`

SegWit program
Version Byte

Im Output kein Tapscript weit und breit zu sehen..

Taproot Commitment

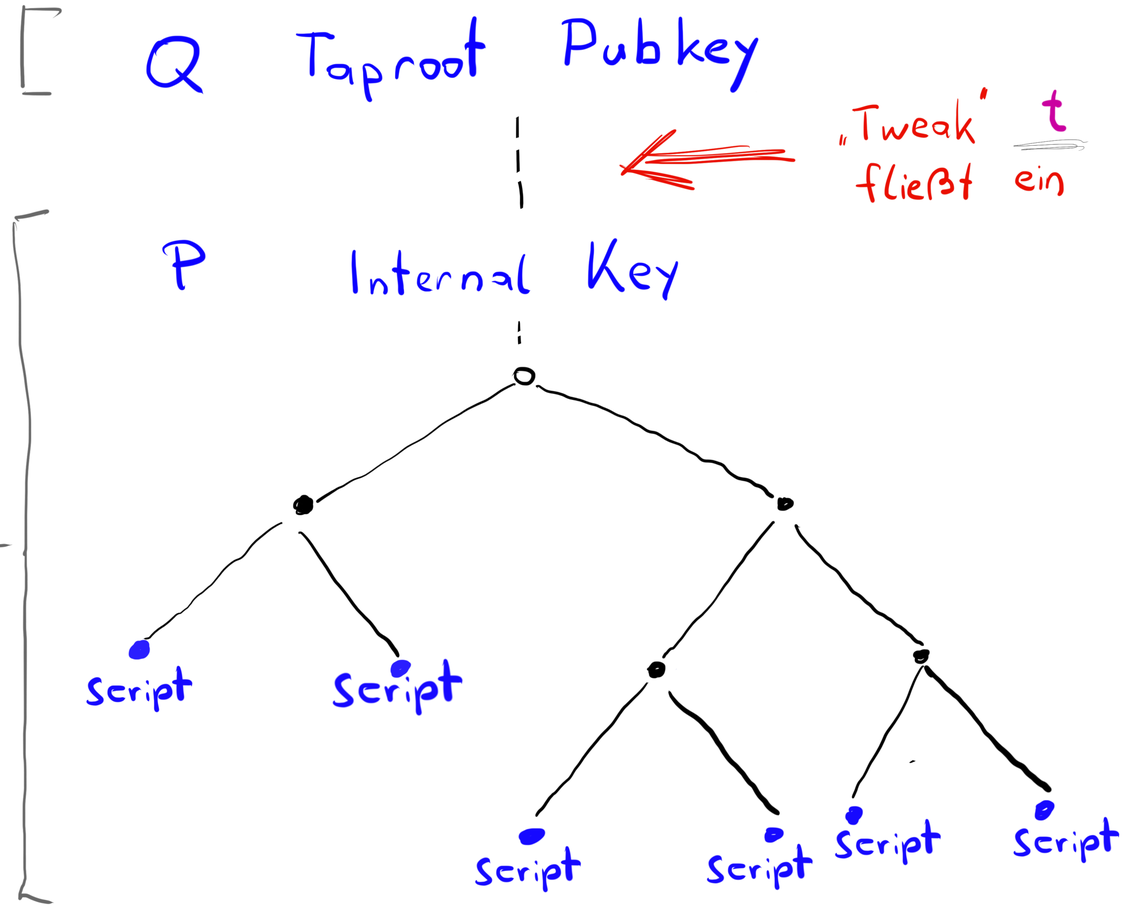
- Q, Taproot Pubkey, ist der public key den man im Output sieht. Das ist ein „tweaked key”
- P ist der interne (ummodifizierte) Pubkey

Q = P + t*G

```
t = TaggedHash(„TapTweak”, P|tapleaf)

tapleaf = ENTWEDER SCRIPT: TaggedHash(
            „TapLeaf”, ver|size|script )

          ODER: Hash des root nodes
                  (siehe nächste Slide)
```

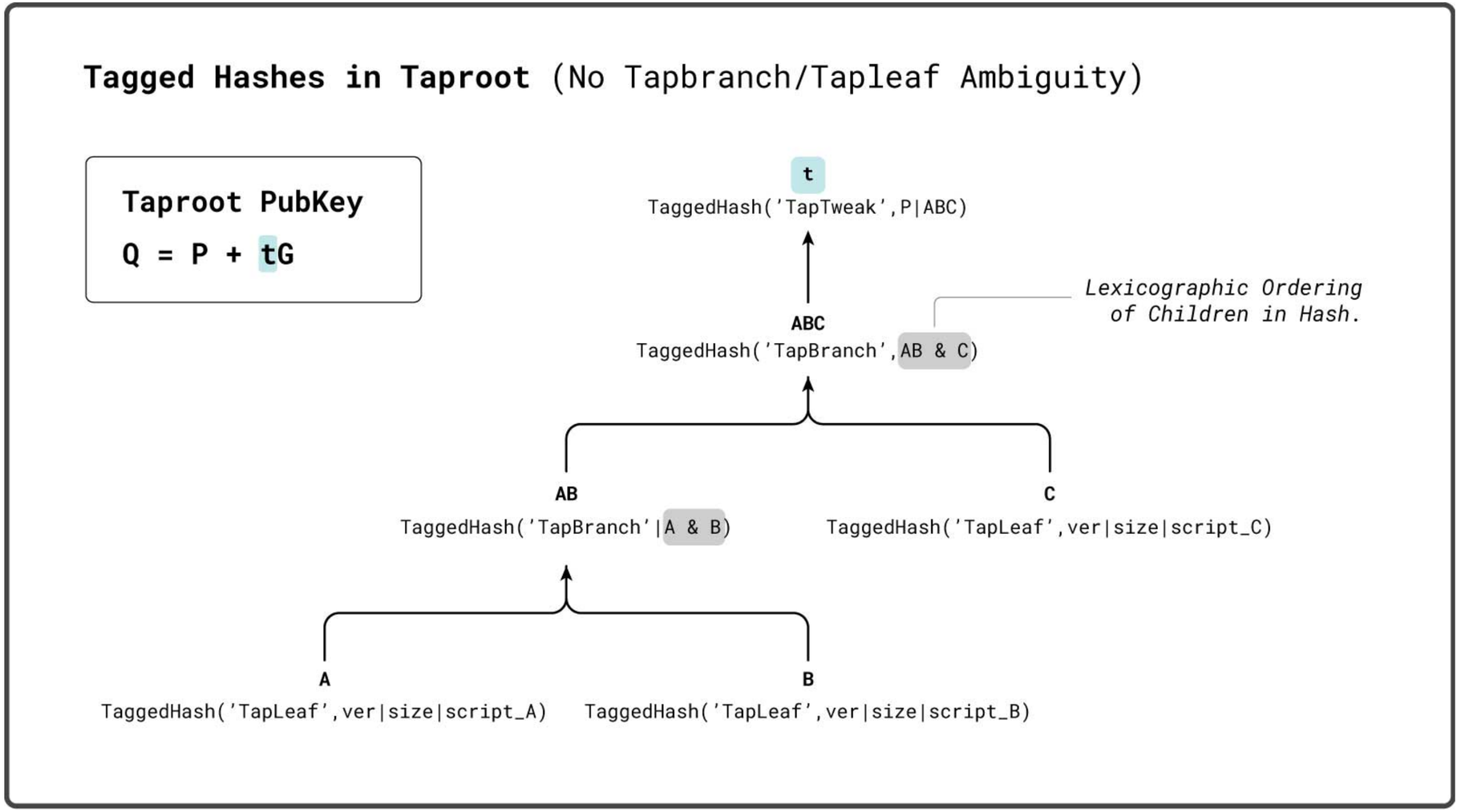


Sidestep: „TaggedHash”

- Auf den folgenden Slides kommt eine neue Hashfunktion vor:
- TaggedHash(….)
- Definiert wie folgt:

```
TaggedHash(tag, data) = SHA256( SHA256(tag) | SHA256(tag) | data )
```

Taptree committent



Details siehe BIP 341

Taptree

- Binärer Baum (muss nicht balanced sein, eher zu erwartende Spending Conditions kann man „näher” an der Spitze platzieren)
- Knoten auf einer Ebene immer lexigrafisch sortiert (damit eindeutig welcher hash rauskommt)
- Branching Knoten und Leaf-Knoten haben unterschiedlichen Hash-„Tag” („TapBranch” vs „TapLeaf”)
- TapScripts (also die eigentlichen Scripts) sind in den Leaf Nodes committed

Auswahl wie man den Output spendet

- Im unlocking script (witness) bestimmt man, ob man:
  - **key path spending** (1 element in witness)
    - 1 element in witness: signature erstellt mit TapRoot Key
  - oder **script path spending** (>1 element in witness)
    - witness elements (unlocking script parts) das Leaf-Script das ich ausführen möchte
    - das konkrete TapScript (Leaf Script) selbst
    - und ein sog. „control block”
      - der unmodifzierte, interne public key (P)
      - merkle inclusion proof

Good case vs. Fallback

Default case vs. fallback

- **Beobachtung**: In komplexen (Multiparty) Scripts (Contracts) gibt es meistens einen good-case, wenn sich alle Beteiligten einig sind
- Die anderen Conditions sind das „Sicherheitsnetz”, wenn sich nicht jemand nicht kooperativ verhält
  - *Beispiel*: Lighning Channel **cooperative vs. non-cooperative** close (forced close)
  - *Beispiel*: Atomic Swap: Gegenpart published auch seine Transaktion auf der anderen Chain vs. ich hole mein Funds über den Timelock Pfad zurück

Angewendet auf Taproot

Erst beim Ausgeben (spenden) eines Taproot Outputs bestimmt man wie man ihn spenden will.

- **„Default Spending Path” (Key Path spend)**
  - Single- oder multiparty public keys (aggregated keys), die unlock conditions (witness) sind on-chain dann alle ununterscheidbar
- **Alternative Spending Path(s) (Script path spend)**
  - Ein oder mehrere alternative Script Paths
  - nur der Path der spent wird, muss revealed werden

- ***Note*:** als Single-User macht es natürlich immer Sinn, den Key Spending Path zu nutzen (da weniger onchain-Platz und somit billiger).

  Aber die **wichtige Erkenntnis** ist, dass **dieser Weg auch für Multi-Party Settings möglich ist** (dank key aggregation), sofern sich alle einig sind, und alle *m von m* bereit sind an der Signatur mitzuwirken.

  Dh. auch bei Multi-Party Setup können alle gemeinsam den „key path spend” wählen und es wir auf der Blockchain ununterscheidbar von einer normalen Single-Key Standard-Transaktion sein! :-)

  —> Beispiele: kooperativer Lightning-Channel Close, kooperatives Finalisieren eines Atomic Swaps, etc..

That’s it..

Geschichte

- Initial vorgeschlagen von:
  - Greg Maxwell
  - Peter Wuille
  - Andrew Poelstra
  - https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2018-January/015614.html

[bitcoin-dev] Taproot: Privacy preserving switchable scripting

Gregory Maxwell greg at xiph.org
Tue Jan 23 00:30:06 UTC 2018

- Previous message: [bitcoin-dev] Blockchain Voluntary Fork (Split) Proposal (Chaofan Li)
- Next message: [bitcoin-dev] Taproot: Privacy preserving switchable scripting
- **Messages sorted by:** [date] [thread] [subject] [author]

```
Interest in merkelized scriptPubKeys (e.g. MAST) is driven by two main
areas: efficiency and privacy. Efficiency because unexecuted forks of
a script can avoid ever hitting the chain, and privacy because hiding
unexecuted code leaves scripts indistinguishable to the extent that
their only differences are in the unexecuted parts.

As Mark Friedenbach and others have pointed out before it is almost
always the case that interesting scripts have a logical top level
branch which allows satisfaction of the contract with nothing other
than a signature by all parties.  Other branches would only be used
where some participant is failing to cooperate. More strongly stated,
I believe that _any_ contract with a fixed finite participant set
upfront can be and should be represented as an OR between an N-of-N
and whatever more complex contract you might want to represent.

One point that comes up while talking about merkelized scripts is can
we go about making fancier contract use cases as indistinguishable as
possible from the most common and boring payments. Otherwise, if the
anonymity set of fancy usage is only other fancy usage it may not be
very large in practice. One suggestion has been that ordinary
checksig-only scripts should include a dummy branch for the rest of
the tree (e.g. a random value hash), making it look like there are
potentially alternative rules when there aren't really.  The negative
side of this is an additional 32-byte overhead for the overwhelmingly
common case which doesn't need it.  I think the privacy gains are
worth doing such a thing, but different people reason differently
about these trade-offs.

It turns out, however, that there is no need to make a trade-off.  The
special case of a top level "threshold-signature OR
arbitrary-conditions" can be made indistinguishable from a normal
one-party signature, with no overhead at all, with a special
delegating CHECKSIG which I call Taproot.

Let's say we want to create a coin that can be redeemed by either
Alice && Bob   or by CSV-timelock && Bob.
```

Relevante BIPs, Ressourcen

- BIP 340: Schnorr
- BIP 341: Taproot SegWit v1 Scripts —> hier die Tap-Trees spezifiziert (also wie der Commit-Hash in der Schnorr Signatur zustande kommt
- BIP 342: Validation of Taproot Scripts —> hier die neue Semantik, also was kann die neue/angepasste Script-Sprache
- BIP 173: Base32 address format for native v0-16 witness outputs (bc1 Adressen)
- BIP141: SegWit: Wie erkennt man SegWit Outputs
- Super Taproot Dev Workshop!!: https://bitcoinops.org/en/schorr-taproot-workshop/
- https://bitcoincore.org/en/2017/03/23/schnorr-signature-aggregation/ —> Ausblick auf Schnorr (aus 2017)