Secure protocols on BIP-taproot
Disclaimer

It’s not at all certain that a BIP-taproot softfork activates in its current form or at all. This depends on community consensus.
BIP-taproot address generation (witness version 1)

Policy: single key

<1> <publicicke>
BIP-taproot address generation (witness version 1)

Policy: single key OR script1 OR script2 OR script3
BIP-taproot spending

output

Key spend

Script spend (Script 2)

(BIP-schnorr) signature

Script 2 inputs

Script 2
bitcoin-core/secp256k1

- “Difficult to use insecurely”
  - Well reviewed and tested
  - Fast and portable
  - Free of timing sidechannels
- rust-bitcoin/rust-secp256k1 type-safe rust bindings (no_std)
- Will provide cryptographic primitives for bip-taproot
  - minimum required: schnorrsig module
- elementsproject/libsecp-zkp
  - fork of secp256k1 with rangeproofs, surjectionproofs, schnorrsig, musig, …
  - just released: rust-secp256k1-zkp beta (schnorrsig, optional no_std)
- HOWTO
  - read the docs before using it (include/secp256k1_*.h)
schnorrsig module

- Deterministic nonce derivation as per BIP-schnorr
  - Picking a specific nonce is unnecessary
- Batch verification
  - 400 sigs can be verified in half the time
  - Don’t know which exact sig was invalid
  - May not reduce worst case cost

64 bytes

"public nonce"

nonce = Number used ONCE
Covert nonce channel

- Problem: malicious HWW can exfiltrate secret key through nonce
Covert nonce channel protection

Solution: enforce putting host-supplied randomness in nonce with sign-to-contract

Alternative: MuSig key aggregation but that’s currently difficult for hardware wallets
Tweak Add

- Create taproot commitment if there’s a script path

```
secp256k1
```

```
<1> <publickey>
```

```
merkle root
```

```
internal publickey
```

```
OR
```

```
script leaf 1
```

```
script leaf 2
```

```
script leaf 3
```

```
output
```
Tweak Add

- Create taproot commitment if there’s a script path

```
P            + hash(prefix, P, root)
G      =      Q
```

```
secp256k1
```

```
internal publickey
OR
merkle root

script leaf 1
script leaf 2
script leaf 3
```

```
<1> <publickey>
```

```
P + hash(prefix, P, root)G = Q
```
Tweak Add

- Create taproot commitment if there’s a script path

\[
\text{int secp256k1\_ec\_pubkey\_tweak\_add(}
\text{    const secp256k1\_context* ctx,}
\text{    secp256k1\_pubkey *pubkey,}
\text{    const unsigned char *tweak)}
\]

\[
P + \text{hash(prefix, P, root)}G = Q
\]
Tweak Add Fungibility

- Try avoiding the script path
  - in multi-party contracts use “happy” case
- Don’t reuse keys
  - internal keys and leaf keys
- Using script path basically leaks wallet
  - Depth of tree, script, ...
- Ensure sufficient leaf entropy
Multisignature Options with BIP-taproot

1. **use** CHECKMULTISIG replacement opcode CHECKDLSADD
   - uses BIP-schnorr and is batch verifiable

2. **Key aggregation**
   - Encode n-of-n signing policy in single public key and single BIP-schnorr signature
   - more fungible, cheaper
   - interactive protocol
Key Aggregation Options

1. **“Legacy”:** p2wpkh key aggregation
   - complicated and 80 bits security

2. **BIP-taproot:** MuSig key aggregation
   - $P = \text{hash}(P_1, P_2, 1)P_1 + \text{hash}(P_1, P_2, 2)P_2$

3. **BIP-taproot: Non-MuSig key aggregation**
   - $P = P_1 + P_2$, and proof of knowledge to avoid key cancellation
   - But one party can add taproot tweak!
   - $P_1 = P_1' + \text{hash}(\text{prefix}, P, \text{root})G$
nonce_commitment = session_initialize(session_id)

nonce_commitment

nonce = get_public_nonce(nonce_commitments)

nonce

set_nonce(nonce)
combine_nonces
partial_sig = partial_sign

partial_sig

sig = partial_sig_combine(partial_sigs)
MuSig Implementation using libsecp-zkp is safe if you

1. Never reuse a session id
   ○ need randomness or atomic counter

2. Never copy the state
   ○ otherwise: Nonce reuse and active attacks
MuSig: Reducing Communication

- Can attach the nonce (commitment) to already existing messages in protocol
  - old message: ClientHello
  - new message: (ClientHello, nonce_commitment)
- Can run multiple sessions in parallel ("pre-sharing nonces")
- Three parallel sessions get one sig per round
  - (partial_sig_i, nonce_i+1, nonce_commitment_i+2)
MuSig with Offline/Hardware Wallets is hard

- Storing state on persistent medium is a copy (dangerous)
- Therefore, serializing state not supported right now in our implementation
- Just have a “single” session?
  - Need to travel to your HWW vault for every single signature
- Hope: deterministic nonce derivation
  - No randomness, no state, two rounds
  - But must be efficient
  - Adds code complexity
Hash locks

\[ \text{OP HASH} < \text{Y} > \]\n\[ \text{OP EQUAL} \]

\[ y \text{ such that } \text{hash}(y) = Y \]
MuSig Adaptor Signatures

- **DANGER**: partial verification required
- **Bonus**: works with n-of-n, where n>=2
Blind Schnorr Signatures

- Interactive protocol between client and signing server
- Signer does not know the message being signed
- Result is a BIP-schnorr signature
Blind Schnorr Signatures Problems

- Vulnerable to Wagner’s attack
  - 65536 parallel signing sessions can forge a signature with only $O(2^{32})$ work
- Moreover, they can’t be proven secure in the Random Oracle Model
Blind Schnorr Signatures

1. If you just need blind signatures (f.e. ecash)
   ○ Don’t use blind Schnorr signatures

2. If you need blind signatures for Bitcoin transactions
   ○ Need to use blind Schnorr signatures
   ○ Idea to prevent Wagner’s attack
     i. Client blinds message with 128 different blinding factors and sends them to server
     ii. Server picks only one of those to blindly sign
Conclusion

- BIP-taproot is a substantial efficiency & fungibility improvement
- Simple sending remains simple
- Can use libsecp256k1 ecosystem for cryptography
- DL assumption is nice (fast, studied)
  - but requires interactive protocols, creates new challenges
- TODO: k-of-n threshold signatures
- Please try to break it!
- Slides at nickler.ninja/slides/2019-breaking.pdf

GPG: 36C7 1A37 C9D9 88BD E825 08D9 B1A7 0E4F 8DCD 0366