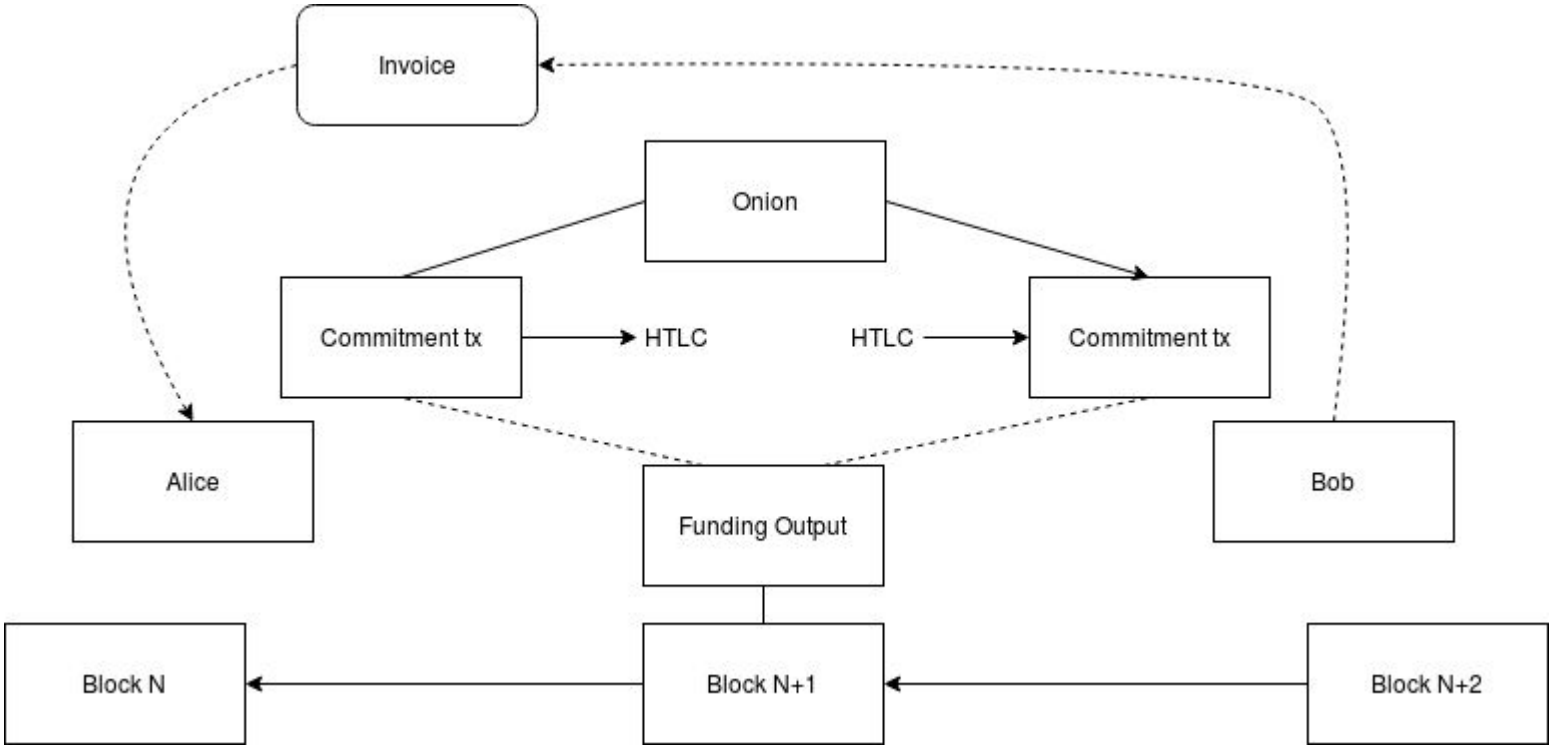
chaincode

A Schnorr-Taproot'ed LN

Antoine Riard, Advancing Bitcoin 2020, London

Lightning architecture



What we should design Lightning for ?

Scalability.

Instant finality.

Cheap fees.

Privacy.

What's the privacy on the base layer ?

A transaction is broadcast on the *p2p layer*, sending an *amount* to an *address* with a given amount by linking *inputs* and *outputs*.

"How much who is paying whom and when ?"

Transactions origin may be inferred on the public network, amounts are unencrypted, addresses may be reused and TXOs form a DAG...

What's the privacy on Lightning ?

A *HLTC* is send through a *payment path* composed of *public channels* to an *identified peer*.

"How much who is paying whom and when ?"

HTLC inherently leak the amount transferred, all locks of the path reuse the same hash, funding outputs are easily recognizable...

Why should we focus on privacy ?

“Cryptography rearranges power, it configures who can who can do what, from what”

(The Moral Character of Cryptographic Work, Philip Rogaway)

EC-Schnorr: efficient signature scheme

Keypair := (x,P) with P = xG and Ephemeral keypair := (k,R) with R = kG

Message hash := e = hash(R || m) and Signature := (R,s) with s = k + ex

Verification := sG = R + eP

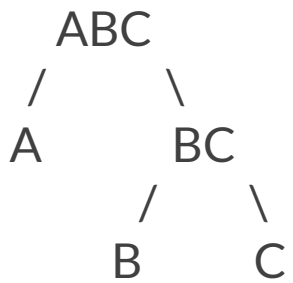
Contrary to ECDSA, the verification equation is *linear*, i. e you can sum up *components* (public key, signatures, nonces)

Taproot: privacy-preserving Script Tree

Taproot pubkey := Q = P + tG with Q and P curve points.

t is the *root* of a Merkle Tree whose each *leaves* is a hash of a Bitcoin script.

Spending witness provides *Merkle proof* and *script*.



New consensus properties

Schnorr:

- non-malleable
- *linear*
- efficient

Taproot:

- *everyone-agree branch assumption* (via Musig)
- *cheaper complex scripts*

More Schnorr-Taproot resources

BIP-340: Schnorr Signatures for secp256k1

BIP-341: Taproot: SegWit version 1 spending rules

BIP-342: Validation of Taproot

Schnorr/Taproot/Tapscript review resources

Channel : “Plaintext” closing

P2WSH output : 0 <32-byte-hash>

Witness script : 2 <pubkey1> <pubkey2> 2 OP_CHECKMULTISIG

Chain analysis heuristic : “A 2-of-2 is *likely* a channel closing, Alice and Bob were using Lightning”

ST*-Channel : “Discreet” closing

Taproot output : 1 <32-pubkey>

Witness script : <MuSig-sig>

Mutual closing can’t be dissociated from a “1-of-1” taproot spending.

Assuming wide deployment of Taproot, *best-case* off-chain contract stays *discreet*, external blockchain observers cannot learn about the channel existence.

*: Schnorr-Taproot’ed ofc

Channel: Worst-case closing

Unreliable/malicious peers may lead to an unilateral close.

Third-party learn about channel state, i.e Alice and Bob balances + pending HTLCs in both directions.

Blockchain-as-a-judge model, concern is about *faithful* execution of the contract, but ideally *content* should stay confidential.

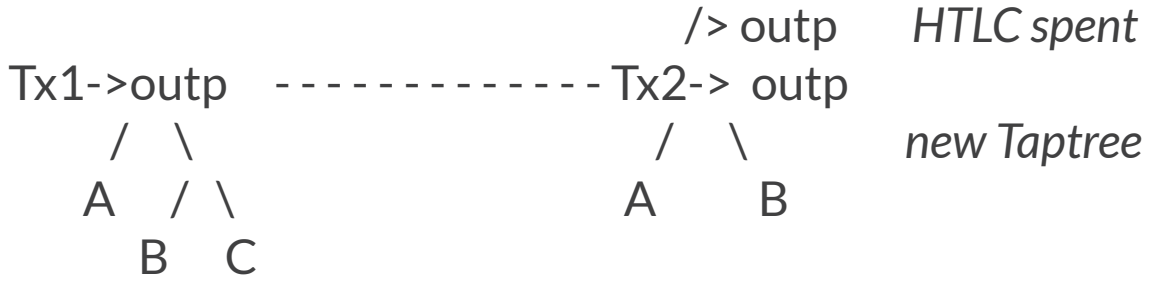
Can we make also unilateral/revoked close *discreet ?*

ST*-Channel: Pooled commitment

For each HTLC + party balances, add a Tapscript to the tree.

Every Tapscript contains a multisig spending to a 2nd-stage transaction.

This is a 2-outputs tx, one is the HTLC being spent, one the Taptree minus the HTLC spent.



HTLC: Payment hash correlation

Every HTLC part of the payment path reuse the same Script hash-lock, i.e:

OP_HASH160 <RIPEMD160(payment_hash)> OP_EQUALVERIFY

If a observer operates multiple hops on the network, it can make assumptions about *graph nearness* of payer-payee.

Alice → Bob → Caroll → Dave → Eve

Network graph is public and efficient payment paths are short.

ST-HTLC: Point-Time-Locked-Contract

Using Scriptless Scripts, lock would use a different value at every hop.

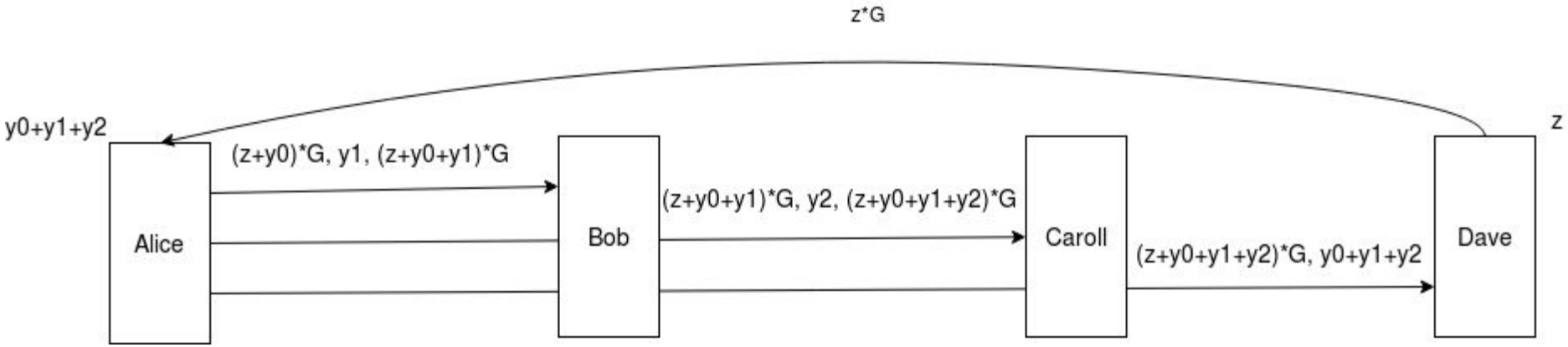
partial_sig := *sG == R + H(P || R || m)P*

adaptor_sig := *s'G == T+ R + H(P || R || m)P* with T, the *nonce tweak*

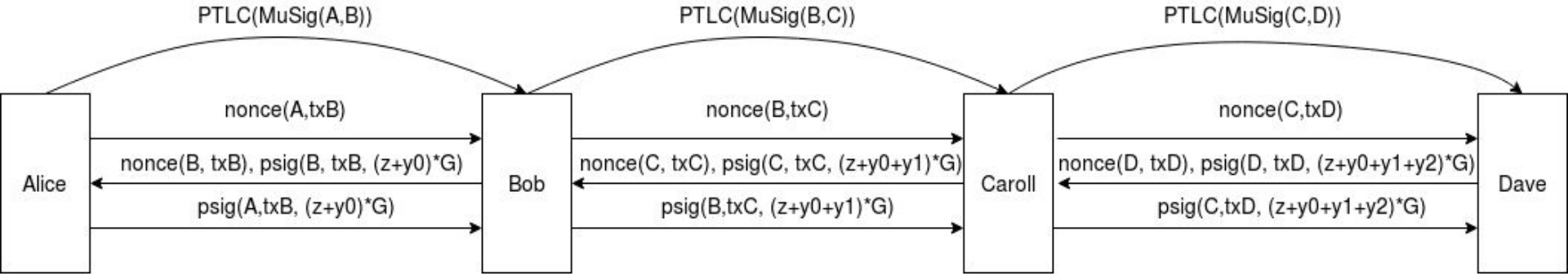
secret t := *adaptor_sig - partial_sig*

Alice and Bob use a multisignature Schnorr where to *redeem coins* Bob combine her partial sig with his own adaptor sig, and so *revealing secret t*.

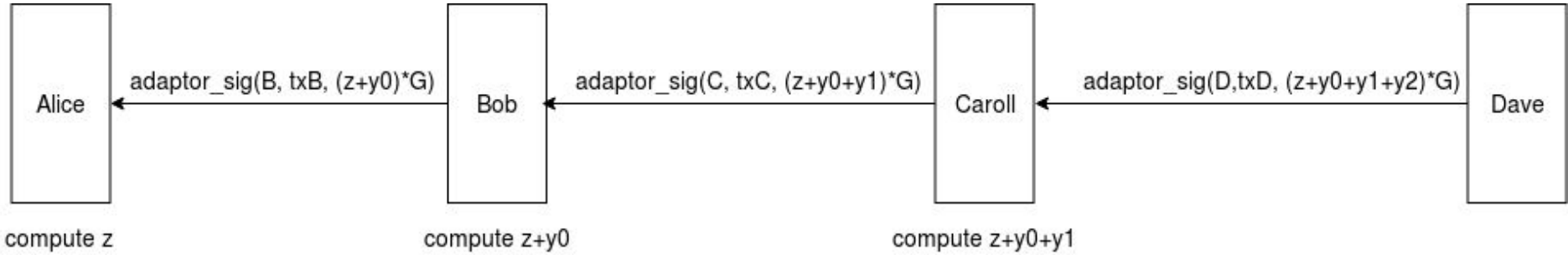
PTLC protocol : setup phase



PTLC protocol : update phase



PTLC protocol : settlement phase



Invoices: proof-of-payment

Invoices are signed by author, such as to recover pubkey from it and allow authentication of requesting node.

It has been designed such to serve as a proof-of-payment in a later dispute after a payment success.

However what happen if invoice leak to an intermediary hop who have seen the corresponding preimage ? Who is the real payer ?

ST-Invoices: proof-of-payer

Following PTLC protocol, the *z* value is only obtained by the sender.

If z^*G has been signed by the receiver, the scalar serves as a proof to uniquely authenticate the sender.

This secret could serve as a primitive to safely trigger a *consumer protection* off-chain contract between a seller, a buyer and an escrow.

Onion-packet: simple payment or MPP

Today Lightning can *fragment* its packets similar to IP.

The general method is called Atomic-Multi-path Payment, a first version is already deployed today, *Multi-Part Payments*.

Such AMP reuse the same payment hash for all paths. Using different hashes means loosing proof-of-payment.

It's huge for liquidity issues, but maybe stand-off for privacy due to *payment paths intersection*.

ST-onion-packet: Discreet Log AMP

Sender picks a random q = q1 + … + qn, with n the number of paths.

With a PTLC protocol, the sender adds $q*G$ to the recipient's payment point ($z*G$) on every path.

For every path i with 0 <= i <= n, sender sends qi along to the recipient inside the onion packet.

After paths are established, recipient can compute q and claim the payment.

HTLC: stuck payments

HTLC may be stuck through the payment path, i.e an intermediary hop is offline and doesn't lock forward or claim/timeout backward.

Ping-before-commitment-signed is implemented but not strong protection.

Make the UX really ugly or worst-case loose funds by paying invoices twice.

ST-HTLC: cancellable payments

With a PTLC mode, Alice secret is *y0+y1+y2* for path A -> B -> C -> D.

Alice send *(z+y0+y1+y2)*G* to Dave as his left lock. Dave knows *z* and *(y0+y1+y2)*G* at the end of update phase.

Dave request *y0+y1+y2* from Alice and present *(y0+y1+y2)*G* to prove her that he received PTLC.

Alice can then safely send *y0+y1+y2* and settlement phase begins.

Can be used to implement some *Forward-Error-Correction* packets.

HTLC: simple hash-time-locked-contract

It’s a simple yet powerful primitive but may not support every real-world payment case, e.g adding an escrow pubkey or revealing multiple preimages.

The whole network would need to upgrade and some common standard spec supported (like *Miniscript’ed-HTLC*).

ST-HTLC: end-to-end payment point contracts

Going further than simple Scriptless Scripts, combine few primitives:

AND := $z*G == (v+w)*G$

OR := $z*G == ECDH((v*G), (w*G))$

Timelock := *MuSig(Tx)*, with Tx a timelocked transaction

Oracle := *DLC(v)*

You now have *confidential* contracts like multiple-party escrows, options, futures, swaps, ...

Protocol-side, no silver bullet, a lot of tricks

Constant-value AMPs ?

Random-CLTV-delta routing algorithms ?

Constant-time onion packet processing ?

Coinjoin-style funding/closing ?

Padded payment paths ?

Application-side, building private-first apps.

Privacy-Preserving light client (no such yet) ?

Tor default integration ?

Identity-less logins (LSAT or blind tokens ideas) ?

Minimizing and encrypting user datas ?

Thanks to Chaincode

Thanks to Advancing Bitcoin

chaincode

Questions?