How DLC-bets work?

Leonardo Comandini – leonardocomandini@gmail.com
Satoshi Spritz

March, 2021

Disclaimer

I am not an expert,
this presentation serves as a starting point for a discussion on the topic.

What are DLC-bets?

- DLC: *discreet* log contracts
- discrete log: private key is the discrete log of public key
- DLC-bets: bets settled on the blockchain, in a trust-minimized fashion

The gamblers

Alice and Bob want to bet on the outcome of an event.

They want to do it now, without waiting for Taproot and Schnorr.

For the sake of simplicity consider a binary event, either Alice wins or Bob wins.

Alice and Bob have their keypairs, $(a, A = aG)$, $(b, B = bG)$

Alice and Bob have access to an oracle.

The Oracle

An oracle is a complex service.

It has a keypair for signing $(p, P = pG)$, publishes P.

For each event, it generates a nonce $(k, R = kG)$, publishes R in advance, to allow Alice and Bob to setup their bet.

For each event outcome, it publishes a corresponding message.

When the event occurs, it signs using nonce (k, R) the message corresponding to event occurred, and publishes the signature.

Funding Transaction

Alice and Bob create the bet output, a 2of2 output, e.g. $wsh(multi(2, A, B))$.

Alice and Bob create the Fundining Transation, i.e. a transaction sending some funds to the bet output.

However they wait for signing the Funding Transaction.

Winner will be able to spend the bet output.

Contract Execution Transaction (CET)

Alice constructs CET_a a transaction spending the bet output and sending funds to her desired destination.

CET_a requires some signature hash m_{ta} to be signed for the bet output.

Bob Constructs CET_b, which requires m_{tb} to spend the bet output.

Fetch Oracle Data

Oracle advertises that:

- if the event Alice bet on occurs, oracle will sign m_{ea}
- if the event Bob bet on occurs, oracle will sign m_{eb}

Oracle also publishes its signing public key P and the event nonce R.

Now Alice needs a signature from Bob on m_{ta}, and Bob a signature from Alice on m_{tb}.

Key Idea

They will encrypt the required ECDSA signatures s.t. the Schnorr signature that with oracle will produce will allow the winner to decrypt the signature produced by the other gambler.

But how?
They use adaptor primitives to produce encrypted signatures (called adaptor signatures), and they take advantage of Schnorr linearity to choose the encryption key.

Let's have a look at those primitives.

Schnorr Signatures

```
def schnorr_sign(p, m, k=None):
    if not k:
        k = nonce(x, m)
    R = k*G
    P = p*G
    s = k + schnorr_challenge(P, R, m)*p
    return s, R
```

Call $S = sG$ the "signature point".
If we know P, R, m we can compute the signature point before the actual signature is produced

$$S = R + schnorr\_challenge(P, R, m)P \tag{1}$$

One-Time Verifiably Encrypted Signatures A.K.A. Adaptor Signatures

- **adaptor encrypt**: encrypts with key (y, Y) a valid signature for key (x, X) and message m, produces an adaptor signature
- **adaptor verify**: verifies an adaptor signature (i.e. that fulfills the above promises)
- **adaptor decrypt**: decrypts an adaptor signature using decryption key y, produces a valid signature for key X, message m
- **adaptor recover**: recover the decryption key y from an adaptor signature and a valid signaure using the same nonce

Adaptor signatures on ECDSA:

```
ecdsa_adaptor_encrypt(x, Y, m) -> adaptor_sig
ecdsa_adaptor_verify(X, Y, m, adaptor_sig) -> bool,
ecdsa_adaptor_decrypt(adaptor_sig, y) -> sig
ecdsa_adaptor_recover(Y, adaptor_sig, sig) -> y
```

Signature Encryption

Alice and Bob compute the signature points for the messages that the oracle may sign

$$S_a = R + schnorr\_challenge(P, R, m_{ea})P \tag{2}$$

$$S_b = R + schnorr\_challenge(P, R, m_{eb})P \tag{3}$$

They will use these to encrypt their signatures.
Alice produces an adaptor signature for message m_{tb}, encrypted with signature point S_b

$$adaptor\_sig_a = ecdsa\_adaptor\_encrypt(a, S_b, m_{tb}) \tag{4}$$

Bob produces an adaptor signature for message m_{ta}, encrypted with signature point S_a

$$adaptor\_sig_b = ecdsa\_adaptor\_encrypt(b, S_a, m_{ta}) \tag{5}$$

Setup – Final Step

They exchange the adaptor signatures and verifies them

$$ecdsa\_adaptor\_verify(B, S_a, m_{ta}, adaptor\_sig_b) \tag{6}$$

$$ecdsa\_adaptor\_verify(A, S_b, m_{tb}, adaptor\_sig_a) \tag{7}$$

If verification succeeds, Alice and Bob can sign and broadcast the Funding Transaction and wait for the event to happen.

Bet Result

Suppose that Alice wins.

Oracle signs m_{ea}, using nonce (k, R) and publishes the signature

$$s_a, R = schnorr\_sign(p, m_{ea}, k = k) \tag{8}$$

Alice sees s_a, and uses it to decrypt the adaptor signature produced by Bob

$$sig_b = ecdsa\_adaptor\_decrypt(adaptor\_sig_b, s_a) \tag{9}$$

Finally Alice signs m_{ta} and can spend the bet output of CET_a

$$sig_a = ecdsa\_sign(a, m_{ta}) \tag{10}$$

Conclusions

DLC-bets are possible now on Bitcoin without deploying Schnorr.

Oracles are complex services that must store event data and react to "real world" events.

Oracles may collude with one of the gamblers.

Several mitigations and optimizations are possible.

Discussion (and spritz) time

Cheers!

Resources

- "Discreet Log Contracts", Thaddeus Dryja, 2017, https://adiabat.github.io/dlc.pdf
- Suredbits DLC blog post series, https://suredbits.com/discreet-log-contracts-part-1-what-is-a-discreet-log-contract
- Discreet Log Contract In Progress Specification, https://github.com/discreetlogcontracts/dlcspecs
- One-Time Verifiably Encrypted Signatures A.K.A. Adaptor Signatures, https://github.com/LLFourn/one-time-VES
- ECDSA Adaptor signatures in secp256k1-zkp (PR), https://github.com/ElementsProject/secp256k1-zkp/pull/117
- Pure Python toy implementation, https://github.com/LeoComandini/adaptor-py