Demystifying Discreet Log Contracts (DLCs)

The Firefish Team

Abstract

Discreet Log Contracts (DLCs), sometimes referred to as smart contracts on Bitcoin, are a technique that allows Bitcoin payments to be conditioned on real-world data. Most resources on DLCs are either high-level overviews or technical specifications. We bridge this gap by providing a detailed yet comprehensible explanation of DLCs, including their motivation and the underlying cryptographic foundation of adaptor signatures.

Introduction

Discreet Log Contracts (DLCs), first proposed in 2017 by Thaddeus Dryja [Dry17] and sometimes referred to as smart contracts on Bitcoin, are a technique that allows Bitcoin payments to be conditioned on real-world data. Typical use cases include sports betting (conditioned on the result of a match) or futures contracts (conditioned on the price of a certain asset). This real-world data is provided by so-called oracles. Unlike on Ethereum and similar blockchains, oracles and real-world data in DLCs are off-chain. This comes from the fact that Bitcoin scripting capabilities are limited. Concretely, it is not possible to directly bring real-world data to the Bitcoin blockchain and work with it in the Bitcoin script. Thus, in DLCs, the execution of the contract is not handled by the blockchain itself but requires taking the cryptographically confirmed data from the oracle and using it to finalize and broadcast the transaction that settles the contract.

Why DLCs

When conditioning Bitcoin payments on external data, one might consider using a 2-of-3 multisig, with the oracle holding one key and working as an arbitrator between the two contracting parties holding the other two keys. Such a setup works, but the oracle (i) must interact with every contract, (ii) knows details about each contract, and (iii) can equivocate, i.e., decide the outcome of each contract individually (even when conditioned on the same event), making potential fraud more feasible.

That brings us to why DLCs are so interesting—they also allow conditional Bitcoin payments, but they solve the above drawbacks. Concretely, DLCs provide:

Scalability: The oracle does not interact with any contract. It only attests to the outcome of an event and publishes the attestation in some public feed.

Privacy: The oracle does not know any details about contracts conditioned on it. In fact, the oracle does not even know that the contract exists at all.

Accountability: Any number of contracts can be conditioned on one event. The oracle cannot attest to two different outcomes of one event without compromising its private key.

DLCs lifecycle

At a high level, the whole process works as follows:

1) Oracle announcement: The oracle first defines an event, including the date of the event, its possible outcomes, and the event's public key. As mentioned above, oracles are off-chain, so an oracle may publish this information, for instance, on its website. In addition to the event's public key, which must be unique for each event, there is also the oracle's public key that can be reused across multiple events. The definition of an event is called an *oracle announcement*.

2) Funding the contract: Anyone can then condition a Bitcoin payment on an outcome of this event. Typically, two parties (Alice and Bob) first lock their Bitcoin into a 2-of-2 multisig address, each party holding one key. This setup ensures that (i) no one can back out of the contract once it has been set up and (ii) Bitcoin cannot be stolen by any single party. The transaction locking Bitcoin in this 2-of-2 multisig address is called a *funding transaction*.

3) Contract Execution Transactions: For each possible outcome of the event, using the event's and oracle's public keys, Alice and Bob compute a special value called an *anticipation point* (sometimes also called a *signature point*). These anticipation points are then used to construct special Bitcoin transactions, called *Contract Execution Transactions* (CETs). They spend Bitcoin from the 2-of-2 multisig address to Alice and Bob in an agreed-upon way, according to the event's outcome. So, each CET corresponds, through the anticipation point, to one possible outcome of the event.

What makes CETs special is that they are not valid transactions yet—a given transaction will only be valid if the oracle confirms the outcome corresponding to this particular transaction. In the end, unless the oracle misbehaves, only one CET will be a valid transaction and will close the contract.

4) Oracle attestation: When an event happens, the oracle determines its outcome and confirms this outcome using the oracle's and event's secret keys, known only to the oracle. The oracle publishes the confirmation, called

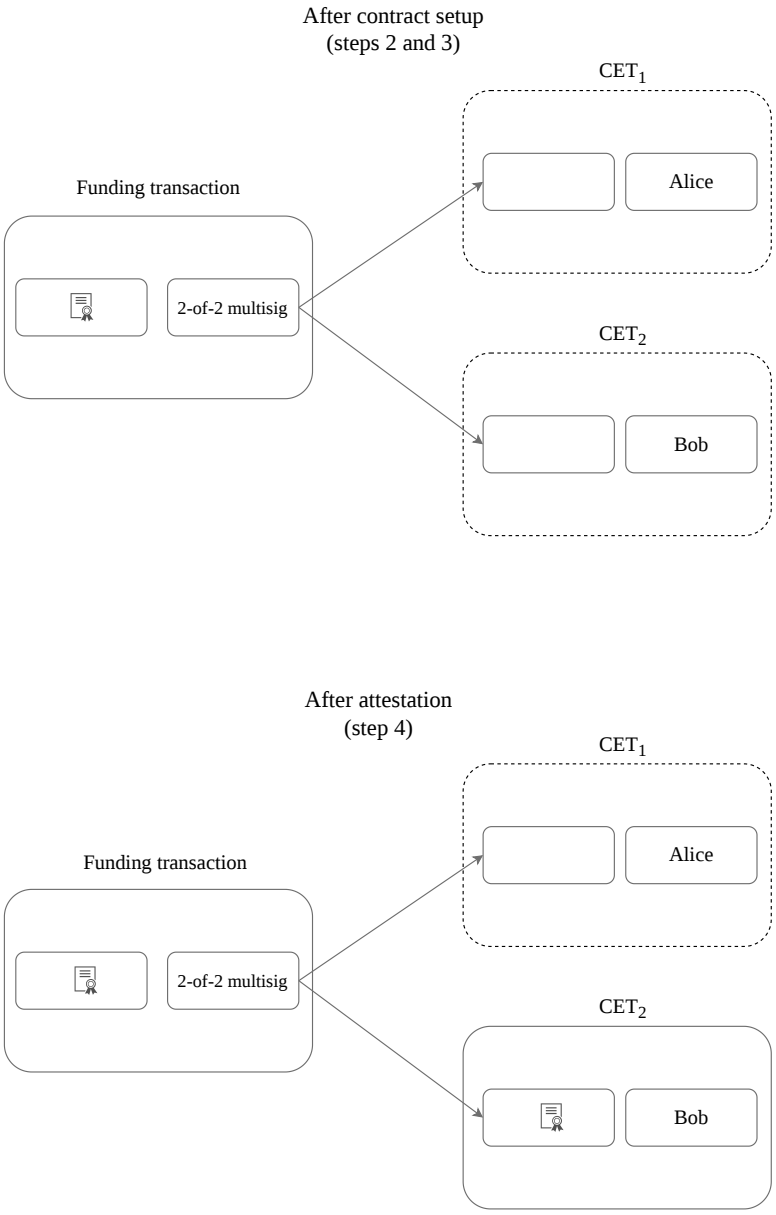

Figure 1: Visualization of CETs after contract setup and after attestation for a simple contract between Alice and Bob consisting only of two CETs. Solid lines correspond to on-chain transactions, and dashed lines to off-chain transactions.

oracle attestation, and Alice and Bob can use it to "unlock" precisely one CET to close the contract.

The funding transaction and Contract Execution Transactions (CETs) are visualized in figure 1.

Evolution of DLCs

In Dryja's original proposal from 2017, anticipation points were applied directly in the CETs locking scripts together with timelocks. The construction worked but introduced an asymmetry in CETs held by Alice and Bob and required two transactions to close the contract unilaterally.

The current DLC specification [Spe] and implementation [Imp] moved from the original proposal and removed the above-mentioned asymmetry by making use of adaptor signatures, as suggested by Fournier [Fou19] and extended by Le Guilly et al [GKK22]. We focus on the current variant using adaptor signatures. Before diving deeper into the construction of DLCs, we show how adaptor signatures work.

Adaptor signatures

Adaptor signatures, popularized by Poelstra as a way to execute scriptless scripts [Poe], are a specific application of verifiably encrypted signatures [BGLS03]. They allow us to create so-called pre-signatures (sometimes also called adaptor signatures) instead of signatures. The pre-signature is not a valid signature on a message—it can rather be seen as an encryption of a valid signature. Now you may ask, encryption using what? Apart from a secret key and a message, a pre-signature is created using a so-called *adaptor point*. The adaptor point can therefore be seen as a public encryption key.

One can transform (or adapt, as the name suggests) a pre-signature into a valid signature only with knowledge of a so-called *adaptor secret* corresponding to the adaptor point used to create the pre-signature. The secret key is not necessary for this transformation. An adaptor point Y is an elliptic curve point, and the underlying adaptor secret y is its discrete logarithm, i.e., y such that $Y = yG$, where G is the group generator. A pre-signature and its transformation into a valid signature are visualized in figure 2.

It is also possible to verify that a pre-signature can be transformed into a valid signature once the adaptor secret corresponding to the adaptor point is revealed. This property will be crucial in the construction of oracles using adaptor signatures.

Adaptor signatures can be constructed for both signature schemes used in Bitcoin—Schnorr signatures and ECDSA. The construction for Schnorr signatures is more straightforward and is shown in the last section.

Linearity of adaptor points and adaptor secrets. Notice that due to its simple structure, adaptor points and adaptor secrets have a very useful

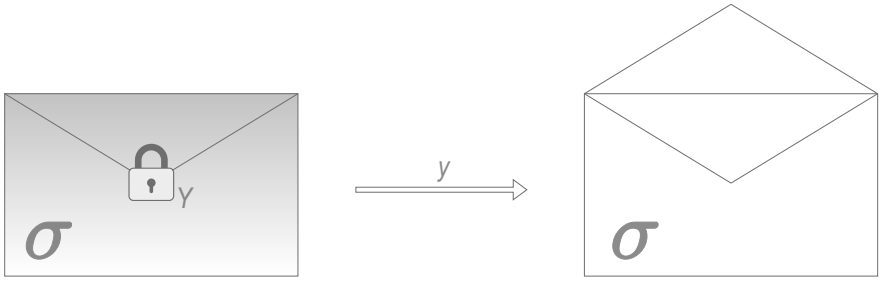

Figure 2: Visualization of a pre-signature and its transformation into a valid signature.

property—linearity. Concretely, consider two adaptor points, $Y_1 = y_1 G$ and $Y_2 = y_2 G$, with adaptor secrets y_1 and y_2, respectively. Then, $y = y_1 + y_2$ works as an adaptor secret for an adaptor point $Y = Y_1 + Y_2$. This property will play a crucial role later in the multiple oracle setup and numerical optimizations.

Construction of DLCs

Now, we show how to construct DLCs using adaptor signatures. We stick with the example of Alice and Bob. The idea is fairly straightforward. Alice and Bob lock Bitcoin to the 2-of-2 multisig address via the funding transaction, compute the anticipation points corresponding to possible outcomes, and prepare the CETs. Then, instead of signing the CETs, Alice and Bob create pre-signatures on the CETs using anticipation points as adaptor points.

Then Alice sends her pre-signatures to Bob and vice versa. They verify that the pre-signatures are valid. Later, when an event happens, the oracle's attestation is interpreted as an adaptor secret and is used to adapt Alice's or Bob's pre-signature into a valid signature for the CET corresponding to the event's actual outcome. Either Alice adapts Bob's pre-signature, adds her own signature, and broadcasts the CET, or vice versa. Note that the resulting CET is the same, no matter whether Alice or Bob finalizes and broadcasts the transaction.

The missing piece in the puzzle is how to compute anticipation points from the oracle's and event's public keys, so that (i) anticipation points can be used as adaptor points, and (ii) later attestations can be used as underlying adaptor secrets.

Anticipation points and attestations

The event's public key, sometimes also called a *nonce*, is a point $R := rG$, where r is the event's secret key. Similarly, the oracle's public key is a point $P := pG$, where p is the oracle's secret key.

Therefore, for a given event, the event's public key R and the oracle's public key P are available to compute the anticipation points. The secret keys r and p are known only to the oracle and the oracle will use them later to compute the attestation when the event occurs.

For outcome o, the anticipation point T_o is computed as

$$T_o = R + P \cdot \mathsf{H}(P \parallel R \parallel o), \tag{1}$$

where H is a hash function. When the event occurs with the actual outcome o^*, the oracle publishes its attestation t_{o^*} computed as

$$t_{o^*} = r + p \cdot \mathsf{H}(P \parallel R \parallel o^*), \tag{2}$$

which is used as an adaptor secret by the contracting parties to adapt the pre-signature into a valid signature for the contract execution transaction tx_{o^*} corresponding to the actual outcome o^*.

From (1) and (2), it is easy to verify that $t_{o^*} \cdot G = T_{o^*}$, meaning that the attestation t_{o^*} really works as an adaptor secret for the anticipation point T_{o^*} being the corresponding adaptor point.

Summary

In summary, the whole adaptor-based DLC construction consists of two main building blocks, visualized in figure 3:

1. Computing anticipation points and attestations.
2. Computing pre-signatures for CETs using anticipation points as adaptor points and adapting the pre-signatures into signatures using attestations as adaptor secrets.

Optimizations and multiple oracles

Optimization for numerical outcomes

Recall that the computation of one anticipation point consists of one point addition and one scalar multiplication, with the scalar multiplication being the much more demanding operation. For events with a lot of possible outcomes, creating all the anticipation points starts to be computationally demanding. Fortunately, for events with numerical outcomes (like Bitcoin price on a specific date), there exists an optimization. Instead of working with outcome numbers directly, these numbers are first decomposed using an agreed-upon base. When the event occurs, the oracle then signs each digit of its numerical outcome separately. For this to work, instead of using one nonce value R as an event's public key, multiple nonce values R_i are used, one for each possible digit. The whole construction works due to the linearity of adaptor points and adaptor secrets.

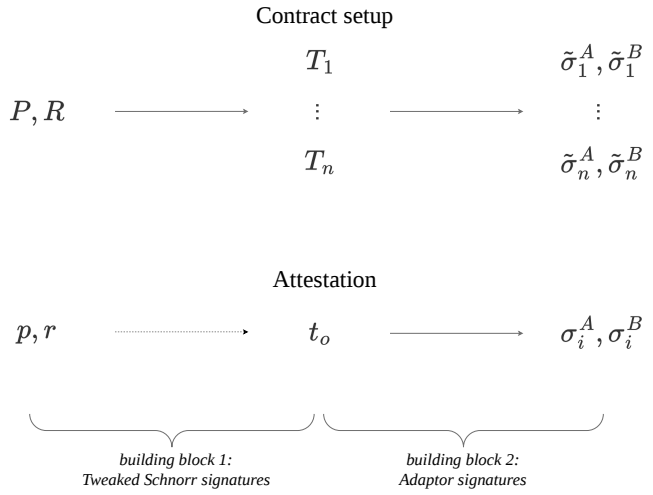

Figure 3: Visualization of adaptor-based DLC construction. Solid lines correspond to actions done by Alice and Bob, and dashed lines to actions done by the oracle.

Multiple oracles

The security and practicality of DLCs heavily depend on the trustworthiness of the oracle. Unfortunately, one cannot completely avoid the risk that the oracle becomes (i) unresponsive or (ii) corrupted. The first leads to the inability to adapt any pre-signature and so to close the contract using a CET. The second leads to the closure of the contract using a CET not reflecting the real outcome. A natural mitigation of this risk is to rely on multiple independent oracles. Fortunately, such construction of DLCs with multiple oracles is possible.

First, we describe the case where all oracles must agree on the outcome in order to close the contract (called n-of-n case). To create a CET for a given outcome, Alice and Bob first compute the outcome's anticipation points for each oracle. Then, all these particular anticipation points are added together, and the resulting sum is used as an adaptor point (called *aggregate adaptor point*) to create the pre-signature for this CET. Similarly, when the event occurs, Alice and Bob compute the sum of all particular attestations for the event (one for each oracle) and use it as an adaptor secret to adapt the corresponding pre-signature. They succeed only if all oracles attest to the same outcome. The whole construction works due to the linearity of adaptor points and adaptor secrets explained earlier. The computational complexity of the n-of-n multiple oracles setup is the following. The complexity of the first building block (computation of anticipation points) is linear in the number of oracles, while the complexity of the second building block (pre-signing CETs) does not increase with more ora-

cles. The setup in which all oracles must agree significantly mitigates the risk of possible corruption but does not help with the potential unresponsiveness of oracles; quite the opposite, as only one unresponsive oracle leads to the inability to close the contract.

In practice, only a partial agreement by the oracles may be sufficient to declare an outcome as valid and to close a contract. Such an approach still eliminates the single point of failure and additionally allows a few oracles to go idle or attest to wrong outcomes. Suppose there are three oracles O_1, O_2 and O_3 and two of these oracles are required to agree to close the contract. Instead of creating one pre-signature for a given CET, Alice and Bob create three pre-signatures—one for each possible pair of oracles (O_1O_2, O_1O_3 and O_2O_3), applying the "all oracles must agree" approach explained above. Then, when the event occurs, say that oracles O_2 and O_3 attest to the same outcome o while O_1 attests to a distinct one. Alice and Bob can then use the attestations from oracles O_2 and O_3, sum them up to create an adaptor secret and use it to adapt the pre-signature for the outcome o corresponding to the pair O_2O_3.

For the general multiple oracle setup in which at least t oracles out of n must agree (t-of-n case), the t-of-t case is applied to each possible combination of t oracles. As there are $\binom{n}{t}$ such combinations, the whole computational complexity of the t-of-n setup is $\binom{n}{t}$ times the complexity of the t-of-t setup.

Schnorr signatures

Schnorr signature scheme

Before showing how to construct adaptor signatures for the Schnorr signature scheme, the standard Schnorr signature scheme is recalled first in figure 4, where H is a hash function. It consists of three algorithms KeyGen, Sign and Vrfy. KeyGen is a key generation algorithm. Sign is the signing algorithm that takes as input a message m and a secret key sk, and outputs a signature σ. Vrfy is the verification algorithm that takes as input a message m, a signature σ and a public key pk, and verifies whether σ is a valid signature on m under pk. Note that the same randomness k cannot be reused for signing two different messages, as it would lead to the compromise of the secret key (two linear equations of two unknowns).

KeyGen	$\mathsf{Sign}(m, \mathsf{sk})$	$\mathsf{Vrfy}(m, \sigma, \mathsf{pk})$
$x \leftarrow \mathbb{F}_q$	$(X, x) := \mathsf{sk}$	$X := \mathsf{pk}$
$X := xG$	$k \leftarrow \mathbb{F}_q$	$(s, e) := \sigma$
$\mathsf{sk} := (X, x)$	$e := \mathsf{H}(X \parallel kG \parallel m)$	$e' := \mathsf{H}(X \parallel sG - eX \parallel m)$
$\mathsf{pk} := X$	$s := k + xe$	**return** $(e = e')$
return sk, pk	$\sigma := (s, e)$	
	return σ	

Figure 4: Schnorr signature scheme.

Schnorr adaptor signature scheme

Now, we describe how to construct adaptor signatures for the Schnorr signature scheme. The full construction is shown in figure 5.

- pSign creates pre-signatures using the adaptor point Y.
- pVrfy verifies whether a pre-signature can be transformed into a valid signature once the adaptor secret is revealed.
- Adapt transforms a pre-signature into a valid signature using the adaptor secret y.

The computation of a pre-signature to a large extent resembles the computation of a standard signature. The only difference is that in the computation of e, the point kG is shifted by the adaptor point Y. As a result, the randomness under e is in fact $k + y$ while $\tilde{s}$ is computed using the original randomness k. Hence, in order to adapt the pre-signature $(\tilde{s}, e)$ into a valid signature, $\tilde{s}$ must be shifted by the adaptor secret y.

$\mathsf{pSign}(m, Y, \mathsf{sk})$	$\mathsf{pVrfy}(m, Y, \tilde{\sigma}, \mathsf{pk})$	$\mathsf{Adapt}(\tilde{\sigma}, y)$
$(X, x) := \mathsf{sk}$	$X := \mathsf{pk}$	$(\tilde{s}, e) := \tilde{\sigma}$
$k \leftarrow \mathbb{F}_q$	$(\tilde{s}, e) := \tilde{\sigma}$	$s := \tilde{s} + y$
$e := \mathsf{H}(X \parallel kG + Y \parallel m)$	$e' := \mathsf{H}(X \parallel \tilde{s}G - eX + Y \parallel m)$	**return** (s, e)
$\tilde{s} := k + xe$	**return** $(e = e')$	
$\tilde{\sigma} := (\tilde{s}, e)$		
return $\tilde{\sigma}$		

Figure 5: Schnorr adaptor signature scheme.

Attestation as a Schnorr signature

Notice that the attestation t_{o^*} is basically a standard Schnorr signature of the outcome o^*, using the event's secret key r as the randomness. That also explains why an anticipation point is sometimes called a signature point. The trick is that by committing to and publishing the point R as part of the oracle announcement, signature points can be computed with only public information and prior to the creation of actual signatures.

Nevertheless, the main purpose of anticipation points and attestations is to be able to compute elliptic curve points (anticipation points) from the oracle's public information such that the oracle is able to provide discrete logarithms for these points (attestations).

It is worth emphasizing that such functionality could be achieved via other constructions, for example, using the non-hardened derivation from BIP 32. However, the construction using Schnorr signatures has the nice side effect that the attestation is actually a signature of the outcome.

Randomness reuse and DLCs

We have shown that attestation is basically a standard Schnorr signature of the outcome, and we know that the same randomness cannot be used to sign two different messages, as it would lead to the compromise of the secret key. In DLC construction, a side effect of this property is that it serves as a security measure against a malicious oracle (the accountability property mentioned in the introduction). If a malicious oracle were to try to cheat by attesting to two outcomes for one event, the oracle's secret key could be computed from the two attestations (signatures). That would lead to the complete compromise of the oracle, as its secret key is reused across all its events.

References

[BGLS03] Dan Boneh, Craig Gentry, Ben Lynn, and Hovav Shacham. Aggregate and verifiably encrypted signatures from bilinear maps. In *International conference on the theory and applications of cryptographic techniques*, pages 416–432. Springer, 2003.

[Dry17] Thaddeus Dryja. Discreet log contracts, 2017.

[Fou19] Lloyd Fournier. One-time verifiably encrypted signatures a.k.a. adaptor signatures, 2019.

[GKK22] Thibaut Le Guilly, Nadav Kohen, and Ichiro Kuwahara. Bitcoin oracle contracts: Discreet log contracts in practice. In *IEEE International Conference on Blockchain and Cryptocurrency, ICBC 2022, Shanghai, China, May 2-5, 2022*, pages 1–8. IEEE, 2022.

[Imp] rust-dlc. `https://github.com/p2pderivatives/rust-dlc`.

[Poe] Andrew Poelstra. Scriptless scripts.

[Spe] dlc-specs. https://github.com/discreetlogcontracts/dlcspecs.