Discreet Log Contracts

Thaddeus Dryja

Abstract

Smart contracts [1] are an often touted feature of cryptographic currency systems such as Bitcoin, but they have yet to see widespread financial use. Two of the biggest hurdles to their implementation and adoption have been scalability of the smart contracts, and the difficulty in getting data external to the currency system into the smart contract. Privacy of the contract has been another issue to date. Discreet Log Contracts are a system which addresses the scalability and privacy concerns and seeks to minimize the trust required in the oracle which provides external data. The contracts are discreet in that external observers cannot detect the presence of the contract in the transaction log. They also hinge on knowledge of a *discrete* logarithm.

Model

There are 3 parties involved in the contract process: Alice, Bob, and Olivia. Alice and Bob are contract counterparties, while Olivia is the oracle. Alice and Bob do not trust each other and do not need to know any legal identifying information about each other, but they must be able to communicate over an authenticated channel, and they must be able to persistently recognize each other. Alice and Bob also must be able to receive signed broadcast messages from Olivia. Olivia does not need to be aware of Alice and Bob, and ideally she has no contact other than broadcasting information. The information is compact enough that broadcast could take place over the Bitcoin network itself, though this should not be necessary.

The DLC protocol can be used for a wide variety of contracts, covering most cases where payouts between parties depend on a publicly known number in the future. In this example, Alice and Bob make and execute a currency future contract. They start on Wednesday, when the Japanese Yen is worth 1000 satoshis. The contract closes on Friday, at which time the Yen is worth 1050 satoshis.

Committed R-point Signatures

Discreet Log Contracts use the well-known Schnorr signature[2], but in an unusual way. Normally users create a private scalar a, compute $A = aG$ by repeated addition of the group generator G, and publish A as their public key. When signing a message, they create another random private scalar k, compute $R = kG$, and then compute

$$s = k - h(m, R)a$$

where $h()$ is a hash function, and m is the message to be signed. The signature is (R, s).

A verifier, given (A, m, R, s) can compute

$$sG \stackrel{?}{=} R - h(m, R)A$$

$$\stackrel{?}{=} kG - h(m, R)aG$$

In Discreet Log Contracts, we instead call the pair (A, R) the public key, leaving only s as the signature. The equation is the same, but R has been re-classified as part of the public key instead of as part of the signature. Use of a similar construction in an attempt to prevent double-spends was recently published in [3].

Other similar recent work along these lines includes Adaptor Signatures [4], and a use of them in Scriptless Scripts for cross chain atomic swaps [5]. While these techniques alter the signature equation by introducing new components, Discrete Log Contracts use the same signing and verification equations, unmodified, and only alter how the components are used; specifically, the s scalar component of the signature is now treated as a private key.

The signing oracle, Olivia publishes point V, which is her public key, and equal to vG. Olivia will use this public key multiple times and should keep v safe. v can be made up of different keys held by different parties. Olivia also publishes another point, $R = kG$. k is a random nonce, and R is a one-time-use signing key. This is the same k and R used in normal Schnorr signatures, but the R value is committed in advance of the signature being calculated; the message to be signed is not yet known, but a nonce has been chosen and an R point published. (This prevents the use of some deterministic nonce schemes such as RFC6979[6]).

Olivia also publishes the description metadata associated with R. Every R has an asset type and closing time associated with it. In this example, R is associated with the price of the Japanese Yen at Friday market close. As R is 33 bytes in length, the metadata will likely be larger than R itself.

Because R is known, sG can be computed for any given m before s is computed by the signer.

Contract creation

Contracts exist as a single output on the blockchain with all the funds to be paid out upon contract execution held for the duration of the contract. (This output can be optimized away in many cases; see Optimizations below) Before contract setup can begin, Alice and Bob must first find each other and agree on the terms of the contract. Alice is "buying" Yen for delivery Friday, while Bob is "selling" Yen.

In order to establish a contract, both parties must have funds in a shared multisignature address. The establishment of this fund output does not differ substantially from the process of channel funding in the Lightning Network. Alice and Bob initially agree on the funding txid and index, so that they may create child transactions spending from this output before it is broadcast to the network and confirmed in a block.

Alice and Bob then proceed to create the contract, which consists of a large number of transactions spending the funding output. The output can only be spent once, so only one of the transactions which make up the contract will ever appear on the blockchain; Alice and Bob do not yet know which and so multiple transactions need to be signed and stored by both of them.

Each closing transaction is based on a different possible price at closing time. In this example, the price is that of the Japanese Yen, expressed in satoshis. Alice and Bob will create thousands of transactions, each for a different closing price.

Similar to the currently developing Lightning Network software [7], parties agree on the contract state but hold variations of the same transaction. Alice holds transactions signed by Bob, which have two outputs: one which pays to Bob directly, and the other which pays to either Alice or Bob. Bob holds the reverse: transactions signed by Alice, which pay directly to Alice, and pay to a script which either Bob or Alice may redeem.

While in the Lightning Network these scripts are used to maintain consistency in a payment channel such that either party broadcasting an old state allows the other party to take all the funds from both outputs, in DLC the same output script is used for a different purpose. Alice and Bob do not reveal secrets to each other; rather it is Olivia who reveals a secret to everyone.

Transactions within the contract

Alice and Bob have thousands of signed Contract Execution Transactions, which they store on their computers. Each of these CETs spends from the contract funding outpoint, and sends to two outputs: the counterparty's pubkey hash, and one's own script hash. The script is the same sequence of opcodes as used in Lightning Network channels. Alice holds

$$Pub_{A_i} \vee (Pub_B \wedge TimeDelay)$$

while Bob holds

$$Pub_{B_i} \vee (Pub_A \wedge TimeDelay)$$

The latter allows a user with key $Priv_{B_i}$ to spend the output immediately, and allowing a user with key $Priv_A$ to spend the output after some time has elapsed. In the Lightning Network, this is

used to enforce revocation of a previously valid channel state, while here it enforces that users only broadcast the transaction which the oracle has indirectly endorsed as the correct state.

Alice holds transactions $TX_1...TX_n$ which send to

$$Pub_{Ai} = Pub_{Alice} + s_iG$$

where

$$s_iG = R - h(i, R)V$$

and Pub_B is just Bob's public key.

Bob holds the inverse: transactions which send to Pub_A, Alice's public key, with a time delay and

$$Pub_{Bi} = Pub_{Bob} + s_iG$$

without the time delay.

While verifiers can compute the point s_iG, they do not know what s_i will be for any given i. When Olivia signs, she reveals the discrete log of a point that Alice and Bob have already computed.

To prevent key-substitution attacks, both Alice and Bob must sign messages proving that they know the private key associated with their public key. As they are interactively constructing the contract, they can sign the messages they exchange from Pub_A and Pub_B. A safer way to avoid key-substitution attacks without the need for interactivity is the key-delinearization technique described in [8]. Using this delinearization, Bob's public key would no longer be

$$Pub_{Bi} = Pub_{Bob} + s_iG$$

, but have coefficients which would still allow him to sign given knowledge of the private key $Priv_B$ and s_i.

Note that while Olivia was using a variant of Schnorr signatures, which as of 2019 is not supported in Bitcoin, Alice and Bob can use ECDSA signatures for their public keys for the Contract Execution Transactions. As Olivia exists entirely off-chain, the entire contract mechanism is compatible with Bitcoin as it exists now, using the same transaction scripts as currently wide-spread Lightning Network channel transactions.

Oracle Signing

Olivia's job is straightforward: wait until Friday market close, observe the closing price of the Yen, and sign that number with the pre-committed nonce.

The oracle sets m to be the observed price, in this example, 1050 yen per satoshi. The oracle computes

$$s = k - h(1050, R)v$$

Here, the price of Yen rose to 1050 satoshis per Yen, so $m = 1050$. (In practice the message m is a hash output but for clarity here it's left as the raw number being signed)

This reveals s_{1050} which Alice and Bob have previously used to derive keys for their transactions.

$$s_{1050}G = R - h(1050, R)V$$

Once the oracle reveals s_{1050}, the private scalar for $Pub_{B_{1050}}$ is fully known to Bob as it is $b + s_{1050}$. Alice similarly knows the scalar for $Pub_{A_{1050}}$.

Contract Execution

Both Alice and Bob now have the option of unilaterally closing the contract at the correct state by broadcasting their TX_{1050}. Immediately upon broadcast, they spend the sighash output, sending that to an address they fully control. As this results in two on-chain transactions, it would be more efficient if the parties agree to create a new transaction, TX_{gg}, sending directly to pubkey hash outputs for both parties with the same amounts as in the contract execution transaction. They also

must be careful to spend the script hash output immediately; if they wait longer than the delay period, before spending it, their counterparty could claim those funds.

If either party prematurely broadcasts an execution transaction, or broadcasts the wrong transaction (for example, broadcasting TX_{950}), they will not be able to spend the script hash output. Their counterparty, after the delay time has elapsed, will be able to claim the script hash output. Since they have sole access to the the pubkey hash output this allows them to claim the entire value of the contract. In this way, violating the rules of the contract forfeits all value in the contract to the opposite party.

Risks of a Trusted Oracle

The oracle, Olivia, can mis-report the price. If the oracle does mis-report, all users of the system will be able to identify the error and stop using the oracle. If Olivia attempts to publicly report two different prices (in order to assume the role of a counterparty in a contract and "win" the bet regardless of the true outcome), she will reveal her permanent private key, as well as the k value for the particular contract she attempted to double-report. If Olivia herself is a counterparty to a contract (e.g. Alice is Olivia), she can cause it to execute in an arbitrary fashion without revealing her private key. This is possible by broadcasting her desired outcome TX_2, then computing

$$Priv_{A_2} = Priv_A + s_2$$

and signing the output to Pub_{A_2}. Olivia signs without publicly revealing s_2, keeping her private key v. This is detectable, and the defrauded party Bob can provide a compact proof of the fraud so that all other users can stop using Olivia's commitments and signatures. Olivia has one chance to defraud contracts she participates in, and then she would immediately lose the trust of all users.

Olivia may also disappear after publishing the R values, but before signing to report on prices or events. To deal with this possibility, contracts can have a refund transaction which can be broadcast by either party a few days after the s values are anticipated to arrive. The refund transaction returns would by default return parties their initial deposits.

Multiple oracles can be used by counterparties in a contract. If two oracle signatures are desired, counterparties simply add the oracle sG points together before adding them to their pubkeys. This can substantially reduce the risk of an fraudulent oracle, at the cost of some risk for data which may not have exact consensus; if Oracle1 reports 1050 and Oracle2 reports 1049, no execution transaction may be safely used and the pre-arraged timeout transaction will eventually become valid, resulting in a refund for both parties.

Another risk is that a party who has lost all or substantially all of their position in a contract may broadcast an invalid Contract Execution Transaction in order to delay the rightful owner of the funds. Ultimately the funds can be recovered by the correct party, but they will have to wait a timeout period before gaining access to the funds. This is an annoyance which can be mitigated by over-collateralizing the contract, and will hopefully be rare in practice.

Optimizations

Several optimizations can reduce computation and data requirements. There are undoubtedly many optimizations to be found when implementing the system, but several basic ideas are listed here.

Base and Exponent R values

When Alice and Bob create their contract, they need to send and store signatures for every possible message m_i that Olivia may sign. There may be many possible prices for an asset, and many thousands of signatures to be verified and stored. While transactions with differing payouts to Alice and Bob are useful and provide precision, there may be many different prices $m_1, m_2, ..., m_q$ which result in identical payout amounts that nevertheless must be anticipated. Most likely there will be "knock-in" and "knock-out" prices below which, or above which, a single party receives all the money in the contract. For example, if the price of Yen falls below 10 satoshis, Alice and Bob agree that Bob

gets all the money, regardless of whether the final price is 4 or 5 satoshis. Similarly, Alice gets all the money when the price goes above 5,000, whether it's 6,000 or 77,000.

Since prices can range over many orders of magnitude, Alice and Bob can optimize for these knock-in and knock-out prices by creating fewer transactions in these extreme ranges. This is possible if Olivia commits to two R values instead of one: $R_{mantissa}$ and $R_{exponent}$. Olivia then promises to sign two messages representing the price: instead of signing "1050", she signs ".050" using $R_{mantissa}$, and 3 using $R_{exponent}$. The decimal base and mantissa's leading 1 are implied; $1.050 * 10^3 = 1050$

Alice and Bob can construct transactions the same way by calculating the sum of the $s_{mantissa}G$ and $s_{exponent}G$ points. When Olivia signs the pair of mantissa and base messages, she reveals the s values which can then be added together to get the scalar needed. The optimization lies in the fact that Alice and Bob don't need precision if the exponent is 4; they can ignore $R_{mantissa}$ when $R_{exponent}$ is 4, 5, 6, etc, and only sign a handful of transactions dealing with many possible unlikely outcomes.

This could be extended to 3 or 4 R-points to allow further granularity. Also, 10 is not an optimal base for exponentiation; 2 is probably a better choice. In the extreme case, one can expose each bit of m and have each bit signed individually, giving contract participants flexibility in their precision, at the cost of a larger message size from the oracle as there will be more s values as well as more public keys to initially share.

Contracts within channels

If contracts each have an individual outpoint and p2wsh address, then every contract will take up space on the blockchain. Instead, contracts can be created within an already existing Lightning Network channel. In addition to the direct and HTLC outputs that exist in the commitment transaction spending a channel funding outpoint, there may be multiple, 2 of 2 multisig outputs which are their own channels or in this case, contracts. This way if both parties are online and acknowledge the results of a contract, they may remove the contract output from the channel and update both balances to reflect the changes the contract made. If either party is un-cooperative the other party can close the parent channel, and immediately close the contract as well using the oracle-supplied s values.

Modifications to Bitcoin

Discreet Log Contracts, as described here, can be used today without any changes to the Bitcoin network. Uncooperative contract executions will appear identical to uncooperative Lightning Network channel closures (though non-fraudulent contract executions will look like justice transactions, which are comparatively rare in the Lightning Network). Two potential soft-fork rule changes to Bitcoin may affect Discrete Log Contracts.

The adoption of Schnorr signatures or Taproot [9] in Bitcoin scripts would have minimal effect on the functionality of Discreet Log Contracts. As the Contract Execution Transactions are already usable with ECDSA, the only benefit is that the initial commitment transaction can be to a single joint key rather than a 2 of 2 multisig address. If Alice and Bob cooperatively close the contract without using Contract Execution Transactions, the entire contract will be indistinguishable from a single signature payment. (Without the proposed Schnorr / Taproot BIP, in this case a 2 of 2 signature would be visible, but no further details of the contract).

A more impactful change would be the adoption of "sighash noinput" [10] or a similar method to have signatures not commit to the specific outputs they are spending. This would allow for more scalable contracts within channels. With currently available sighash methods, changing any part of the funding mechanism for a contract would require re-signing all execution transactions. With sighash noinput, multiple contracts could be created and destroyed with no redundant signing.

Further work
Novation

Futures contracts are useful, but typically traders want to enter and exit positions before the closing times. If both parties are online and agree, they may be able to negotiate an end to the contract based

on the current price, rather than the future price. One party may, quite reasonably, not agree to cut the contract short. In the case where Bob wants to leave early, but Alice wants to see the contract through to the end, Bob may be able to swap places with Carol, another user who wants to take the position Bob held. Bob may offer a fee for Alice to change nothing about her contract except for the public keys of her counterparty. This seems to require all 3 parties to be online simultaneously and seems to be an interesting area of research.

Decentralized Matching

Before Alice and Bob can create a contract, they need to find each other. Initially this can be done with centralized matching engines, which would hold no custody of user funds. Fair decentralized peer discovery and matching is a topic of active research and would fit well with execution via Discreet Log Contracts.

Conclusion

Discreet Log Contracts have the potential to enhance the use cases of Bitcoin and other cryptographic currency networks by allowing users to discreetly enter in to futures contracts for a wide variety of assets, trusting oracles only to sign the correct price. As the transactions look the same as Lightning Network transactions, it will remain difficult to estimate the total usage of DLCs across the network, and they should allow extensive complex smart contracts to occur without unduly taxing the global network.

REFERENCES

Nick Szabo. Formalizing and securing relationships on public networks. *First Monday*, 2(9), 1997.

Claus Schnorr. Efficient identification and signatures for smart cards. pages 239–252, 1990.

Cristina Prez-Sol, Sergi Delgado-Segura, Guillermo Navarro-Arribas, and Jordi Herrera-Joancomart. Double-spending prevention for bitcoin zero-confirmation transactions. Cryptology ePrint Archive, Report 2017/394, 2017. http://eprint.iacr.org/2017/394.

Andrew Poelstra. Scriptless scripts. Technical Report (draft), 2017. https://download.wpsoftware.net/bitcoin/wizardry/mw-slides/2017-05-milan-meetup/slides.pdf.

Giulio Malavolta, Pedro Moreno-Sanchez, Clara Schneidewind, Aniket Kate, and Matteo Maffei. Anonymous multi-hop locks for blockchain scalability and interoperability. Cryptology ePrint Archive, Report 2018/472, 2018. https://eprint.iacr.org/2018/472.

T. Pornin. Deterministic usage of the digital signature algorithm (dsa) and elliptic curve digital signature algorithm (ecdsa). IETF Request for Comments, 2013. https://tools.ietf.org/html/rfc6979.

Thaddeus Dryja Joseph Poon. The bitcoin lightning network: Scalable off-chain instant payments. Technical Report (draft), 2015.

Gregory Maxwell, Andrew Poelstra, Yannick Seurin, and Pieter Wuille. Simple schnorr multi-signatures with applications to bitcoin. Cryptology ePrint Archive, Report 2018/068, 2018. https://eprint.iacr.org/2018/068.

Pieter Wuille, Jonas Nick, and Anthony Towns. Taproot: Segwit version 1 spending rules. BIP, 2020. https://github.com/bitcoin/bips/blob/master/bip-0341.mediawiki.

Christian Decker. Sighash noinput. BIP, 2017. https://github.com/bitcoin/bips/blob/master/bip-0118.mediawiki.