Asset-Based Lending smart contract for Liquid network

Author: Dmitry Petukhov (dp@ruggedbytes.com) (C) 2020

With review and help from Russell O'Connor

Introduction

Challenges in 'smart contracts'

Contracts are specified with constraints and alternatives for the behavior of the parties entering the contract. The specification is created in a way to encode the mutual agreement between the parties for a subset of their future behavior - the 'spirit' of the contract.

Such encoding (the 'letter' of the contract) is good if the interpretation of this encoding in all relevant states of the 'universe' the contract is interpreted in allows for all the behavior parties agreed to, and disallows the behavior parties agreed that it is not acceptable.

Often the number of possible states and transitions in a contract is so big that it is cumbersome to encode all of them in the contract. Hence, a lot of cases get deferred to broader rules such as laws of the country. For cases that are not covered by the contract or when the law is ambiguous, the resolution is deferred to the third party, which can be a court of law.

For contracts encoded for execution in blockchain-based distributed consensus systems ("blockchains"), the final arbiter is the network of independent agents and the rules of consensus that governs this network. Deferral to the court of law may be possible if the parties to the contract are mutually identified, but this might be undesirable or uneconomical. In extreme cases, there can be calls to make exceptions to consensus rules to rectify unintended consequences of some contract execution. This may pose danger to the most valuable property of such networks - the consensus.

There may be different answers to the problems outlined above for blockchains: centralization ('the blockchain governance'), real-world contractual agreements in addition to on-blockchain contracts, or ensuring that totality of the contract states and transitions can be understood by the parties to the contract.

The later may actually be achievable if the rules of the contract and the relevant rules of the blockchain the contract is encoded for are sufficiently simple. The reduction in complexity may mean that such contracts cannot express rules complex enough to encode all the conditions that parties to the contract desire to encode. Still, we believe that with sufficient research, encodings can be found that would capture the essence of the popular contracts, and that these encodings can be widely examined and sufficiently understood.

UTXO-based[1] blockchain systems are most suitable to achieve this goal, because the rules governing the spending of the units of value in these systems are relatively simple and understandable. When the rules for the spending of an UTXO restrict not only how the current UTXO can be spent, but also putting restrictions its descendant UTXO, this is usually referred to as "covenant"[2]. By combining covenants, it is possible to build constructions that can have significant expressive power for defining the rules of the contract.

The innovations

We will show how features such as the possibility to create covenants that restrict spending of UTXO by the hash of transaction outputs, and how the ability to issue and transact new assets on Liquid network[3] allow to build an Asset-Based Lending contract that can be examined and understood in its entirety without extraordinary effort, from the high-level specification to the low-level (the transactions and the covenant scripts).

One distinguishing feature of Elements platform and Liquid network that is built on this platform is the confidential values and confidential assets in transactions, where only those who possess special "blinding" keys can the view actual values and asset types in the transaction outputs protected by this feature. We will show that it is possible to use this feature to protect the financial privacy of the parties in the contract, and still enforce the rules of the contract.

The already mentioned ability to issue and transact new assets is used to enable the transfer of the contract rights and

obligations to the third party without spending the UTXO locked by the covenant. This enables a secondary market for such contracts to exist.

Our approach to contract specification and development

Holistic analysis

In blockchain systems, the behavior of the contracts is defined through layers of successive encodings from the high-level specification of the contract down to on-chain behavior.

The layers can include: the network of deterministic agents interpreting the encoded behavior; the semantics of the low-level language such agents implement to interpret the encoded behavior; the compilers from the high-level languages used to encode behavior to the low-level language; translation of the contract specifications and concrete contract parameters to the code and data ready for deployment.

Each layer introduces a degree of complexity, and complexity increases the effort required to check that the behavior allowed by the aggregate encoding of the contract represents the contract specification fairly.

The simpler the layers are, and the fewer layers exist, the less is the probability of deviations. Note that formal verification employed at particular level can somewhat alleviate the difficulties introduced by complexity, but these gains do not usually transfer to the other layers.

This means that the examination of the contract needs to be done holistically over all the layers of the system.

Reasoning about the contract behavior

On each layer, reasoning about the contract behavior require specific considerations. When the layer-specific reasoning is localized, it is also becomes easier to reason about the whole contract and inter-layer interactions.

On the layer of network agents ("network layer"), we need to reason about the rules of enforcing the timelocks, the rules of transaction sorting in the mempool, and inclusion of the transactions in the blockchain.

Inclusion of a transaction in the blockchain results in spending one set of UTXO and creating another set of UTXO, which represents state transition in the contract.

The contract construction requires the use of absolute timelocks, and allows to set the fee for the transaction at the spend time. Reasoning about absolute timelocks is simpler, and setting the fee at spend time gives the spender an ability to control the transaction position in the mempool directly, by replacing the transaction with the new one that pays increased fee, but still adheres to the rules of the contract for particular state transition.

There are two available timelock mechanisms: blockheight-based and median-time-past based[4]. The former are more precise in definition, but is sensitive to the rate with which the new blocks are produced. The later does not have the easily predicted cut-off moment where the timelock expires, but is based on the wall-clock time of the block producers.

On the layer of Elements Script ("script layer") Where the UTXO spending conditions are enforced by the low-level opcode interpreter, we need to reason about the constraints and alternatives the script allows with expected arguments (witness data), and ensure that successful execution of the script is impossible with unexpected arguments.

Our covenant scripts are quite easy to understand, with the extensive comments provided in the Appendix section. The scripts are either linear or can be considered as such, because OP_IF conditionals are branching on the flag directly provided by the spender, so it can be said that they just choose between the two linear scripts to execute.

Because the covenant scripts are small, their goal is simple and the data is manipulated in a straightforward way, it is easy to trace the script execution and show that successful completion is impossible with unexpected arguments and that the script enforces the needed constraints.

The layer of high-level language compilation ("compilation layer") we do not need to consider, because we work with Elements Script directly.

It is possible that this layer will need to be considered when Simplicity[5] bit machine will be introduced in Elements. Clear and concise semantics makes programs compiled for Simplicity bit machine well-suited to apply formal methods to. High-level languages that target Simplicity will allow to express complex spending rules more conveniently, and hopefully some of the complexity can be counterbalanced by the formal verification.

The miniscript encoding developed by Blockstream and the Policy Language[6] that is compiled to miniscript provide the

tools to easily define spending rules using a number of standardized components that can be combined together in predefined ways. The advantage of the Policy Language is at the same time its weakness. It provides great clarity when your spending conditions can be defined using its restricted set of expressions. It is not applicable otherwise, and cannot be used to define our custom covenants.

On the layer of translating the contract specifications ("translation layer") and concrete contract parameters to the encoded constraints, we need to reason about data generation.

The transaction templates are constructed for each possible stage of the contract execution. Arrays of outputs from the constructed transaction templates are serialized, and the hashes of this data are used to encode the constraints for the stage. Timelock constraints are calculated from the contract parameters. These hashes and the timelock constraints are then embedded in a predefined Elements Script code templates.

This is the most sophisticated part of the whole contract implementation, but on this layer we are also the most prepared to deal with complexity.

We are able to use well established general-purpose programming languages, which makes it possible to use a wide arsenal of tools to ensure the correctness of the code, and also facilitates code review.

On the inter-layer interactions, we need to reason about how the generated data is embedded in the low-level script, how the timelocks express the intended contract terms, how the timelock restrictions interact with mempool ordering, etc.

At this layer we only have our models, judgement and the knowledge of the underlying levels as the tools of reasoning.

We will need to use tools available at the respective layers to ensure that our models are fairly encoded.

This is the layer most suited for wide review, and publishing the detailed explanations is one of the tools available to ensure consistency of the models and judgement, via critical public review.

High-level specification

The specification of the contract terms is presented in this section.

An example algorithm for the calculation of differentiated repayments is specified.

The contract can accomodate other algorithms as long as they can produce the results representable by the fixed amount of covenant-locked steps.

The terms for the contract variant with partial repayments has also been specified using TLA$^+$ formal specification language at https://github.com/dgpv/ABL_contract_TLAplus_spec/

Contract premise

Alice possesses certain quantity of asset *Principal_Asset* that she does not currently utilize, but wants to extract some value from.

Bob possesses asset *Collateral_Asset*, but needs some P amount of asset *Principal_Asset* temporarily.

Bob does not want to sell *Collateral_Asset*, because he predicts that its value or utility will be higher in the future than what the current market value, for tax reasons, etc.

Bob is willing to pay the for the temporary use of P while offering C amount of *Collateral_Asset* as a guarantee of repayment to the creditor.

Contract can be terminated at any time by any settlement that is mutually agreed by the two parties.

Basic Asset-Based Loan contract

t_0 is the point in time when contract begins

t_1 is the point in time when the pre-agreed duration has passed since t_0

Interest rate R is pre-agreed

Alice is willing to give out P to Bob, provided that:

- Before t_1, she will receive $P + P * R$

- Otherwise, at or after t_1, she will be able to claim C

Bob is willing to freeze C for certain period, provided that:

- He will receive P immediately
- If $P + I$ is repaid before t_1, he can recieive C back

Bob agrees that if $P + I$ is not repaid before t_1, Alice can claim C for herself.

Contract start: To enter the contract, Alice and Bob create and cooperatively sign a transaction that:

- Sends P, provided by Alice, to Bob's address
- Sends C, provided by Bob, to the address of a script that enforces the terms of the contract above

This contract is simple, but limited. It requires for principal to be repaid in one lump sum, it is often preferable for the principal to be repaid in portions over time.

Asset-Based Loan contract with partial repayments

The repayment is split into N installments.

M consecutive missed payments lead to collateral forfeiture.

The contract ends in maximum $S \in [\max\{N, M\} + 1, N + M]$ number of steps.

The concrete value of S within this range is pre-agreed.

The rates used for calculation of interest or surcharge are pre-agreed:

- R_D is the rate for regular repayments *due*
- R_E is the rate for surcharge on *early* repayments
- R_C is the rate for penalty on the part of collateral returned in the event of default
- $R_{L(1)} \dots R_{L(M-1)}$ are the rates for surcharge on *late* repayment: $R_{L(1)}$ is applied when one payment is missed, $R_{L(2)}$ is applied when two consecutive payments are missed, and so on

n is the number of partial repayments, $n \in [0, N]$

m is the number of missed payments, $m \in [0, M]$

B is the outstanding principal balance

$F_P = \frac{P}{N}$ is installment size (the "Fraction of P")

$D = \min\{F_P * (m + 1), B\}$ is the portion of the balance currently *due* to be repaid[7]

$L = \min\{F_P * m, B\}$ is the amount the repayment is *late* on

C_{uncond} is the amount of collateral that is unconditionally forfeited in the event of default

$A_{reg} = D + B * R_D + L * R_{L(m)}$ is the regular repayment amount

$A_{early} = B + B * R_D + (B - D) * R_E + L * R_{L(m)}$ is the early repayment amount

$A_{penalty} = \max\{B, A_{reg}\} + \max\{B, A_{reg}\} * R_C$ is used for calculating collateral distribution in the event of default

The contract can progress over total S time periods, and $t_0 \dots t_{S-1}$ are the points in time at the beginning of each period.

At t_0:

- $n = 0$
- $m = 0$
- $B = P$

Alice is willing to give out P to Bob, provided that:

- Before each $t_s, s \in [1, S - 1]$ she will receive A_{reg}, and then:
    - n will be incremented
    - m will be reset to 0
    - B will be decreased by D

- Otherwise, m will be incremented
- If $m \geq M$, or after t_s, $s \geq S - 1$, she will be able to claim certain portion of C

Alice agrees that before t_{N-1}, B can be set to 0 if Bob repays A_{early}

Bob is willing to freeze C for certain period, provided that:

- He will receive P immediately
- When the condition $B = 0$ is reached during contract execution, he can receive C back

Bob agrees that Alice can claim a portion C for herself if the condition $m \geq M$ is reached during contract execution, or after the point in time t_s, $s \geq S - 1$ is reached.

A portion of C that Alice can claim in this case will be dependent on the amount of principal that was repaid previously, and will equal to $C_{forfeit} = \max\{C_{uncond}, \min\{C, C * A_{penalty} \div P\}\}$, and Bob will receive $C - C_{forfeit}$ portion of the collateral back[8]

Contract start: To enter the contract, Alice and Bob create and cooperatively sign a transaction that:

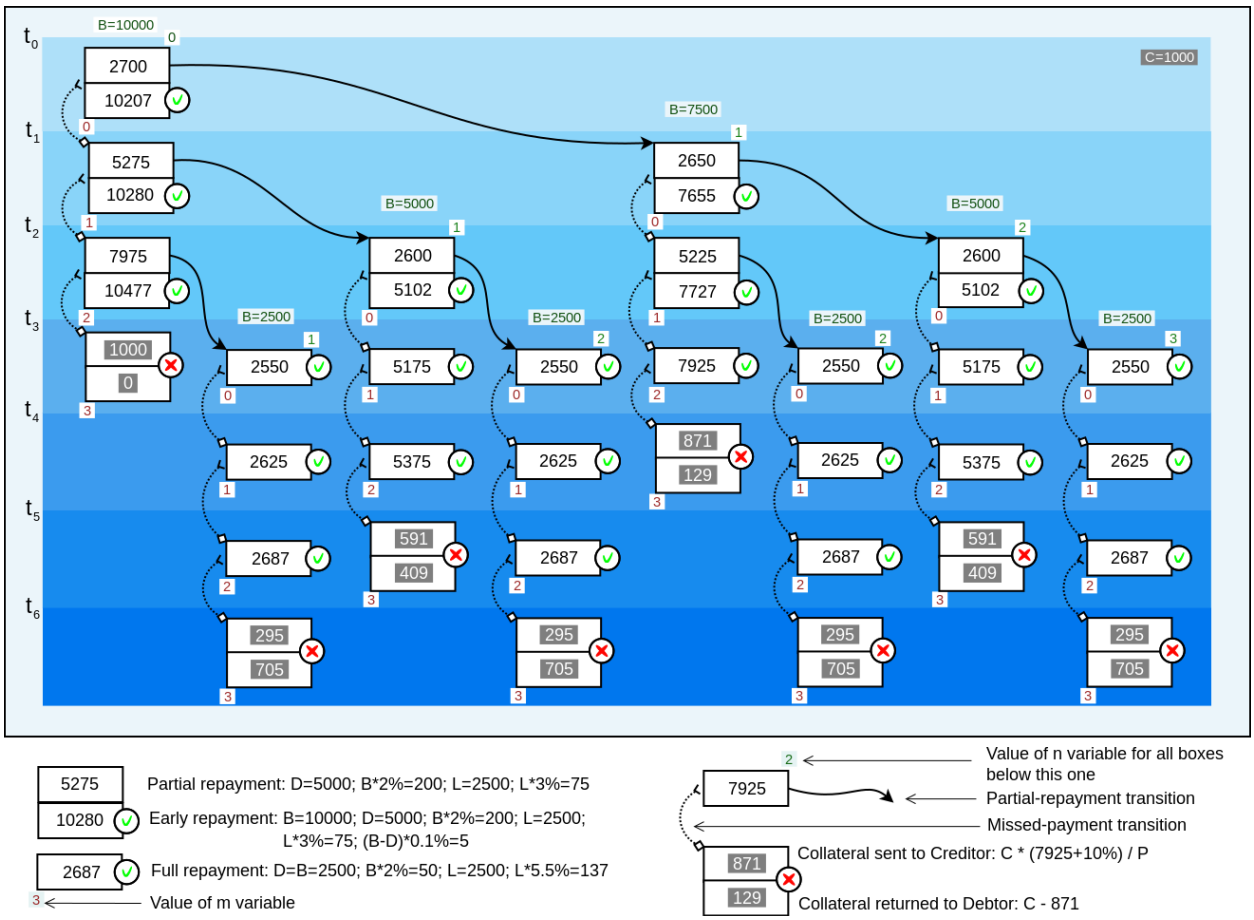
- Sends P, provided by Alice, to Bob's address
- Sends C, provided by Bob, to the address of a script that enforces the terms of the contract above

Examples

Calculated amounts on the presented schemes are rounded down.

Example scheme 1 illustrates the contract with:

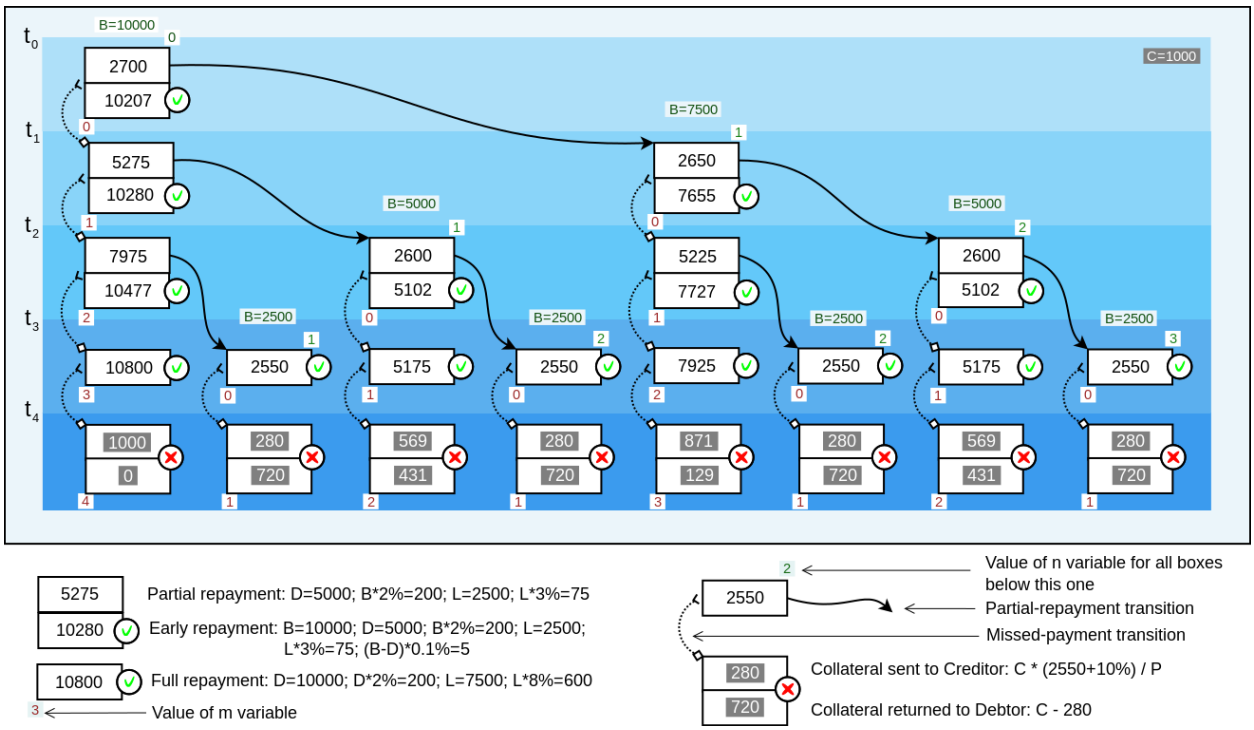
- $P = 10000$, $C = 1000$
- $N = 4$, $M = 3$, $S = 7$
- $R_D = 0.02$, $R_E = 0.001$, $R_C = 0.1$, $R_L = (0.03, 0.055)$, corresponts to 2%, 0.1%, 10%, (3%, 5.5%)



Example scheme 2 illustrates the contract with:

- $P = 10000$, $C = 1000$
- $N = 4$, $M = 4$, $S = 5$
- $R_D = 0.02$, $R_E = 0.001$, $R_C = 0.1$, $R_L = (0.03, 0.055, 0.08)$, corresponts to 2%, 0.1%, 10%, (3%, 5.5%, 8%)

The layout with $N = M = (S - 1)$ allows to have the collateral forfeiture event to always happen in one particular period.



Representing state in the contract

As shown in the specification, there are three main components of the state: n, m and B. On the graphical scheme, we can see that change of these variables is orthogonal: n increases as the contract progresses laterally, and B decreases. m increases as the contract progresses vertically.

On the network layer, the state progression is represented as spending the UTXO that represents the current state to create a new UTXO that represents the new state (enforces the constraints that should be applied in the new state).

It can be said that actions of Debtor move the state laterally, and actions of Creditor move the state vertically.

Lateral state progression

This is pretty straightforward. With repayment of debt, each new state in lateral progression represents reduced obligations of Debtor. When the debt is repaid in full, the collateral is returned to Debtor. The Debtor is not required to wait for the end of the time period to perform the partial repayment, but there is no incentive for them to pay too early. The contract allows the Debtor to repay early in full, but there is no reduction of interest if they decide to combine the repayments of adjacent periods. Combining partial repayments to reduce interest is a complication for the non-common case. Such cases can be dealt with via mutual agreement clause in the contract.

Vertical state progression

The vertical progression is more interesting. Timelocks can prevent the spending of UTXO before a certain point in time. But after that point in time, restriction is lifted forever.

The amount of interest that Creditor is entitled to receive increases with m. The Creditor would want to restrict the ability of a Debtor to spend UTXO that represents the state with m after point in time t_{m+1} has passed, so only the transactions that pay the increased interest can be valid.

The direct mechanism to restrict the validity of transactions after certain point in time has some fundamental problems related to resolving chain reorganizations[9], and is unlikely to ever be available in Bitcoin-based cryptocurrency networks.

Since we cannot create a contract that would automatically revoke the ability of the Debtor to use earlier states in a vertical progression, the Creditor has to perform an action to do this revocation. And the timelock can be used to prevent the Creditor from performing this action too early.

After the timelock representing the end of period expires, the Creditor can spend the UTXO with locked collateral, to be locked in the new covenant. This new covenant will represent the next state in the vertical progression. The last covenant in the vertical progression will allow the Creditor to claim the collateral (in full or in part, depending on the terms and the state of the contract).

Transferable rights and obligations

A distinguishing feature of Elements/Liquid is the natively issued assets, that is present in the transaction data and therefore can be committed to by covenants. This enables another interesting feature, the on-chain spending delegation.

It is possible to cheaply issue a new *unique asset* (with total issued amount equal to 1), and then construct the spending condition in a covenant in such a way that spending requires that this particular unique asset to be present in the transaction. The UTXO with that asset (the "control UTXO") is different from the UTXO that locks the collateral (the "main UTXO").

The spending condition in the main UTXO covenant is tied to the asset id of this unique asset, and not the keys of the participants. It is possible to transact the control UTXO independently of the main UTXO. This allows to transfer the ownership of the rights or obligations in the contract without changing the contract's state[10].

This on-chain delegation of spending enables secondary market to exist for the contracts that are implemented via Elements/Liquid covenants.

Control assets

Both Debtor and Creditor will receive their own unique issued assets on contract deployment. These assets will be used to control the respective sides othe contract.

"Debtor's control asset" will be required to be present in the transaction outputs whenever lateral state transition or final debt repayment happens.

"Creditor's control asset" will be required to be present in the transaction outputs whenever vertical transition or the forfeiture of collateral happens.

The mechanism for controlling the spending via a control asset is simple. As the covenant script "fixes" certain transaction outputs, one of these fixed outputs can be the output that bears the asset id of the control asset, and the amount of 1. Since the control asset is issued with total amount of 1, there can be only one UTXO in existence with that asset. Therefore, to spend a covenant controlled via this mechanism, the spender must also control that UTXO, and include it as one of the inputs to the transaction.

As that UTXO with the control asset can be independently spent, it is possible to transfer the control of the covenant. This "control UTXO" can be sold, atomically swapped, can even be bound by its own covenants. This makes it possible for the secondary market for the contracts controlled via this mechanism to exist.

Because OP_CHECKSIGFROMSTACK-based[11] covenants allow to commit not only to all outputs as a whole, but to the part of the outputs, and the boundary of the committed/non-committed part is not necessary on the boundary of the data representing a single output, it is possible to enforce the requirement that certain output bears particular asset and amount, but allow any scriptpubkey in this output. Thus the covenant that requires certain control asset to be present can also allow that control asset to be sent to any address.

It might be possible to devise more elaborate control schemes with assets that have total issued amounts of more than one. We did not explore this in too much detail. In trivial, easy to devise cases this strategy does not seem to have an advantage over traditional multisig policies.

Another thing to note is that transferring the control asset not only gives the control over the corresponding side of the contract, but also the control of all the repayments made by the Debtor to date, if these repayments are not yet spent. If

the current owner of the Creditor's control asset do not wish to also gift the already made repayments to the new owner of the control asset, they need to spend these repayments to their own addresses, first.

Keeping data confidential

Parties to the contracts often want the terms or even the existence of the contract to be confidential, due to commercial and privacy considerations.

On-chain versus off-chain

In contrast with off-chain contracts, which usually reveal some of this information only when the parties disagree and one party decides to settle on-chain, smart contracts that are executed on-chain will always make some of the information about the contract public.

The on-chain contract deployment transaction can have distinguishing features that can reveal what kind of the contract it likely is, and show that certain pseudonymous entities are entering the contract, by attributing the source UTXO that was spent for contract creation. Spending the UTXO locked in the contract necessarily reveal the scripts that enforce the terms of spending this particular UTXO. While these terms may not reveal all the terms of the contract, they may reveal some part of these terms, and also allow an outside observer to narrow down the identification of what type of the contract this is.

Off-chain contracts can have disadvantages like interactivity requirements, limited expressiveness or increased complexity of zero-knowledge cryptographic constructions.

For the type of the contract described here, where the state transitions are likely to be infrequent, the on-chain contract is a good choice, especially when the underlying blockchain network has additional privacy features.

Confidential transactions

Confidential transactions do not reveal the amounts in transaction outputs, or the type of assets transferred. Using confidential amounts and assets in a covenant is a challenging, but feasible task. Such confidential covenants make preserving privacy for on-chain contracts a bit easier.

The covenants used in the contract described here require that those outputs of the transaction that are critical to enforce the contract terms exactly match the outputs of the template transaction that was created according to these terms before the contract is deployed on-chain.

One problem with this is that the process that is applied to the transaction outputs to make the amounts and assets confidential[12] (the "blinding" process) uses random data in the calculations. The random data is necessary for the cryptographic construction to work as intended. An outside observer that can know or predict that data can reverse the "blinding" process and reveal the assets and amounts.

The requirement that an outside observer is unable to predict the data does not mean that the data has to be truly random. Its distribution has to be uniform, but it can be deterministic. An outside observer has to be unable to know or predict the source values from which this deterministic random data is generated (the "random seed").

It makes sense to choose a shared extended BIP32[13] key to be used to generate both the "blinding keys" for the outputs the participants are willing to share details about, as well as to generate a random seed for blinding these outputs from the same shared extended key. It also makes sense to use different BIP32 derivation paths for the blinding keys and random data.

One complication is that when a transaction with confidential outputs also has confidential inputs, the data of the confidential fields in the outputs of this transaction will depend on the data in the inputs. The so-called "blinding descriptors" that one has to supply to the blinding function for each input contain the "blinding factors" for amounts and for asset identifiers.

When the tree of transaction templates are prepared, the blinding factors of the "parent" transactions in the contract are already deterministic. But when there are "free" inputs, such as the inputs used to repay the debt, that are not known when the contract is created and deployed. The UTXO used for these inputs need to be specially prepared, they have to use contract-specific deterministic random data to make the blinding factors the same as the contract's covenant expect.

This means that, for example, the Debtor that is going to partially repay the debt will need to make an extra transaction to create this special UTXO with deterministic blinding factors. The small cost of this extra transaction is justified by

the increase in privacy.

Often this can be worked around, because in the blinding process, only the last confidential output receives the 'balancing' blinders and has its data changed. If the covenant can avoid committing to the last confidential output, arbitrary confidential inputs can be used.

Before contract deployment

Verification of the covenant scripts

Before the contract is committed on-chain, participants can verify the functioning of all the covenants in the contract. The covenants do not commit to the inputs of the transactions, only to the outputs. This makes creating the mock transactions to test even easier. Script verification in Bitcoin-derived systems does not require access to the blockchain. The timelock values are assessed based on nLockTime field of the transaction. While transaction with particular nLockTime may not be accepted by the blockchain at the moment of testing, it is fine to use such transaction for the purpose of the verification of script execution.

Number of possible states in the described contract grow with larger N and M parameters. Still, for many practical combinations it is feasible to generate all the transactions and witness data to exhaustively test the possible states in the contract and check that spending scripts for all inputs finish successfully. Where an input needs to be signed by the counterparty, the tester can just do the signing themselves with some keys generated specifically for this test. It is the covenant scripts that are tested, not the ability of the counterparty to sign.

The possibility to do these tests in isolation allow the participants to gain increased confidence in the correctness of execution of the contract before it is deployed on-chain.

Privacy of UTXO ownership

For the initial contract transaction to be created, at least one party have to know all the UTXOs that go into that transaction. This is because someone needs to broadcast the transaction, the transaction has to be valid and thus contain all the inputs. The party that received the information of the UTXO of the counterparty, can just stop any communication, but at this point there is an information asymmetry. One party knows more than the other. The amounts for the contract are negotiated beforehand, so not only the UTXO is known, but the amount and the asset of that UTXO as well, even though it may be protected by the confidential transaction features of Elements/Liquid.

One way to solve this problem is to use a trusted third party. The trust participants need to extend towards this third party is only in that it will respect their privacy, and will not share the information about their UTXO with anyone. It will not take the custody of the funds at any moment. Such third party, let's name it "Facilitator", would know the terms of the contract and all the UTXOs that participate in the contract.

The participants will receive the semi-signed transaction from the Facilitator with the UTXO of the counterparty removed. They will then sign their inputs with sighash flags SIGHASH_ALL and SIGHASH_ANYONECANPAY, so their signature would commit to all outputs of the transaction, but only one input, their input. The Creditor's control asset and the Debtor's control asset would be issued using the Creditor's and Debtor's inputs, accordingly. This way they can ensure that the total amount of the control asset is exactly one, there's no reissuance, and that control asset is sent to their address.

Contract deployment

Interactive

In the basic scheme of the contract deployment, the Debtor's control asset is issued in the input that bears the collateral asset, provided by the Debtor, and the Creditor's control asset is issued in the input that bears the principal asset, provided by the Creditor. This means that the parties must know the asset ids before they can sign their inputs. They can prepare the deployment transaction together and mutually sign it.

When the parties do not want to reveal their UTXO to the other party, they can use the services of the Facilitator, as discussed earlier. The Facilitator can relay the asset ids between the counterparties without revealing their UTXO. This would still be an interactive process:

- The first party that initiates the contract chooses the contract terms and gives their UTXO, that also bears the issuance of the first control asset, to the Facilitator
- Facilitator gives the contract terms and the first asset id to the second party
- The second party accepts the terms and gives the Facilitator their signed UTXO that also bears the issuance of the second asset id
- The Facilitator gives the second asset id to the first party
- First party gives their signed UTXO to the Facilitator
- Facilitator completes and broadcasts the contract deployment transaction

Non-interactive

To make the deployment non-interactive, the first party need to know the asset id of the second party beforehand. That asset can be issued separately by the first party or the Facilitator. The second party needs to check that this asset is issued with total amount of one and no reissuance allowed. Because the address of the second party is not known beforehand, the deployment transaction has to allow the second asset to be sent to any address. This means that additional covenant will be involved.

For example, the process where the Debtor seeks the loan with the help of the Facilitator would be:

- The Facilitator issues the asset with total amount of one and no reissuance, gives asset id to the Debtor
- The Debtor builds the contract using its collateral UTXO to issue the Debtor's control asset, and the asset id provided by Facilitator as the Creditor's control asset
- The Debtor prepares and signs a transaction that sends the collateral UTXO to the covenant that allows two options for the locked "pre-contract" collateral UTXO:
    - It goes into contract deployment transaction
    - It goes back to the Debtor
- The Debtor gives that signed transaction to the Facilitator, but Facilitator does not broadcast it yet
- The Creditor that wants to take the Debtor's offer checks the issuance transaction of the Creditor's asset, and makes sure that the asset is issued with total amount of one and no reissuance
- The Creditor constructs a transaction that sends the Creditor's asset to their address, and that allows the Debtors' collateral UTXO pre-contract covenant to be satisfied
- The Creditor signs their principal input using SIGHASH_ANYONECANPAY flag, so the signature commits only to the principal input, but also to all the outputs, including the destination address for the Creditor's control asset
- The Creditor gives their signed input to the Facilitator
- The Facilitator completes the deployment transaction by signing their input that bears the Creditor's asset, using SIGHASH_ALL flag
- The Facilitator broadcasts two transactions: First the pre-signed transaction that sends the collateral to the pre-contract covenant, and then the contract deployment transaction

In this process, after the initial agreement of the Debtor to offer their collateral to get the loan on certain terms and giving the Facilitator their pre-signed transaction, further actions from the Debtor is not required, and Facilitator can complete the process with any Creditor.

The Debtor can opt out of the contract at any time before the contract deployment transaction was broadcasted, by spending their collateral UTXO. The deployment transaction will be invalid if the collateral UTXO is spent in another transaction, so the contract deployment will be atomically aborted.

The Debtor that is willing to reveal their UTXO can issue the Creditor's asset themselves, and lock this asset in a similar pre-contract covenant, then publish the details. This way, anyone can fullfill the Creditor's part and enter the contract.

Note that because the asset id does not commit to the issued amount, the issuance transaction that the Creditor checks has to be already confirmed at the time of the check.

Of course, the similar process can be perfomed when the Creditor is the party that offers the principal on their chosen terms and Debtor enters the contract non-interactively.

Blockchain network considerations

Timelocks

The scripts in the presented contract use lock-by-blockheight timelocks. Bitcoin-based blockchains also have an option to use the lock-by-blocktime (median-time-past) timelocks[4].

The blockheight-based timelocks are more convenient to use, because they are more exact. At the same time, the inter-block interval is not fixed by consensus in Liquid Network, and in theory the Liquid federation can decide to change the inter-block interval.

If this happens, and the blockheight-based timelocks were used for the implementation of the contract, the assumptions about real-world time for the periods defined by the contract will not be correct anymore.

This can be dealt with by using median-time-past timelocks, or by trusting the federation to not change the inter-block interval at least for the period when the contract can be active. Such drastic change is unlikely to be done suddenly, and it is likely to be announced with significant leeway. Still, the long-running contracts has to take this into account.

Identifying contract transactions

When the contract is deployed and operated, new transactions are created by the participants. These transactions correspond to the allowed transitions in the contract state, and therefore it is possible to reconstruct the contract state by retrieving data from blockchain, when you know the initial, 'seed' data for the contract.

The seed data for the presented asset-based lending contract with partial repayments include:

- The terms of the contract, that consist of the variables defined in the contract specification: $P, C, M, N, S, R_D, R_E, R_C, R_{L(1)} \dots R_{L(M-1)}$, and the length of the time interval $t_s \dots t_{s+1}$.
- The txid of the contract deployment transaction
- Shared blinding extended key that is used in producing blinding keys and deterministic random data for the contract transactions

Since the deterministic algorithm is used in generating all the transaction templates for the contract before the contract deployment, and that the covenants only allow the transactions that match these transaction templates, it is possible to walk the transaction tree in the blockchain to find the actual state of the contract.

Participant that wants to sell their rights and obligations under contract to the third party can provide the seed data to them, and that third party can independently analyze the terms of the contract and its current state, to assess the potential purchase.

Mempool considerations

Fees

The covenant construction used for this contract allows to set the transaction fees dynamically at the time of the broadcast of the transaction, rather than at the time of creation of the contract, even though transactions in Elements are required to state their fees explicitly via special "fee output". This is because OP_CHECKSIGFROMSTACK-based covenant allows to commit only to the part of the outputs data, and thus the fee output can be 'free-standing'.

Transaction pinning

There is an issue known as "transaction pinning"[14] that arises from the conflict between the need to bump the transaction fee and the need to protect nodes from wasting bandwidth due to mempool transaction spam.

Transaction pinning may enable an attack that prevent certain transaction in the contract to be confirmed after it was initially broadcasted with the fee rate inadequate to the current mempool contention. The attack works by preventing the originator of the transaction to use available mechanisms to increase the fee rate for the transaction.

Bitcoin Core version v0.19 introduced a special exception in the rules governing the mempool, the "CPFP carve out"[15]. These changes are yet to be integrated into Elements source code base at the moment of writing, but it can be expected that this will happen soon enough.

The effect of this special exception is that if each participant can directly spend one output in a transaction, they can always use that output to increase the fee rate of the transaction.

Care must be taken in adding extra outputs to the transactions that could be directly controlled by the party other than the initiator of the transaction. For example, there might be third-party service that provides fee inputs to the participants. Such a service might want to add a 'change' output to return excess fee to themself, otherwise they will need to maintain a bunch of exact-value UTXO for this task. This change output would be an additional output that is directly controlled by the party other than the initiator of the transaction, The initiator of the transaction would need to trust this service to not use that change output to deny the use of CPFP carve out exemption to the initiator.

Financial considerations

Non-recourse loan

The contract is essentially a non-recourse loan contract. The amounts of the assets are fixed before the deployment, and changing them requires the amendment of the contract. In the case of Elements/Liquid blockchain network, this means deployment of the new contract that receives the assets that were locked in the old contract. To allow this transition, each state of the contract should allow mutual-agreement case. If participants mutually agree, the contract can have any outcome, including the transfer of assets to the new contract.

The loan being non-recourse may limit its applicability, as the Creditor has to accept that there is no rules of the on-chain contract that would require the Debtor to increase the amount of collateral if the price of the collateral asset falls. Similarly, these rules won't allow the Debtor to take back some of the collateral in case the price of the collateral asset raises.

Unique assets as collateral

The assets that have total issued value of one, the "unique assets" can be used as a collateral in the described loan contract. Such a loan would have a property that the collateral cannot be split. Thus, in the event of default, the Creditor would receive the control of the whole collateral, regardless of how much of the principal was already repaid by the Debtor.

Unique assets may be used in the contexts where they represent some concrete thing that has value as a unit, like a house or a unique game item. In such contexts there is likely to be a custodian who can restrict the transfer of the item, with real-world contractual terms applied to the custody. These real-world contractual terms can include the conditions on how the on-chain control of the unique asset translate to the real-world (or game-world) possession. The arbitration in the case when the collateral was forfeited while part of the principal was already paid can also be done by that custodian.

Secondary market

If the secondary market for these contracts exist, both Debtor and Creditor can realize the new gains or limit the losses by selling their contract to a third party. It might be more difficult for Creditor to sell the contract that has the collateral price diminishing, as the buyer needs to be optimistic on the prospects of that price. Still, in a big enough market, selling such contract might be a viable prospect.

Oracles

It is possible to include a condition in the contract covenant script that would allow to release the collateral to the Creditor if a valid signature from the third party is presented. Such third party would play the role of the "oracle" that would release the said signature only in the event of the price of the collateral asset being lower than certain threshold. Both Debtor and Creditor have to trust in this "oracle". The users of the Liquid network already trust the Liquid Federation to some extent. Having the federation, or some subset of the federation members to be the collective "oracle" might be the option that requires less additional trust from participants. Still, there is enough other difficulties in this approach - how to handle swift price movements, how to reconcile the price between members of the "collective oracle", etc. We believe that the approaches that require less trust from participants are more valuable, and we did not explore the "oracle" approach further at this time.

Options contracts

Using the same techniques and mechanisms as the loan contract described here, it is possible to create options contracts. This means that these options contracts can also have the properties of amounts and assets being confidential, and the possibility of transferring rights and obligations in the contract to the third party without consent of the counterparty.

The option contract that is deployed at time t_0, would lock asset A until time t_1 and will allow the option buyer to exchange A for a fixed amount of asset B, until the point in time t_2 is reached. After that, the put option seller can reclaim A. The option seller gets paid commission at time t_0 for the risk taken on locking A for the duration of the contract.

Whether the contract implements a put option or a call option depends on which asset is used in place of A and B. For example, contract that locks X L-BTC with a strike price of Y L-USDT can be viewed as a call option to buy X L-BTC at price Y. A contract that locks X L-USDT with strike price of Y L-BTC can be viewed as a put option to sell Y L-BTC at price X.

Options contracts can be used to hedge the risk in loan contracts. The contract illustrated at example graphical scheme 2 (the one illustrating the contract with $N = M = S$) is the most suitable for this, because the collateral forfeiture event happens always in the same time period for any contract branch. This way, the Creditor can buy a put option for the collateral asset at this period, and pay the option seller to accept the risk of the asset price fluctuation.

It may be tempting to create a joint contract that would enable the option to be realizable only in the event of the default on the loan. This might make the risk presumably lower for the option seller. But it also opens the possibility of collusion between Debtor and Creditor against the option seller to exploit this presumption of lower risk.

Another thing to mention is that if the option is to sell the collateral asset for the same asset that is given as a loan, it might make sense for the option seller to just become the Creditor in a simpler contract rather than be an option seller in a joint contract. The dynamics of an option contract is different from a loan contract, and some market participants may prefer one type of the contract to another.

Thanks:

- To Russell O'Connor, for inspiration of the idea for this contract and for helping to figure out how to combine it with confidential transactions, for reviewing this article and pointing out bugs and inefficiencies in the early drafts.
- To Jonas Nick, for sharing the idea of using unique assets for delegation and the ideas on implementation of option contracts
- To them both, for reviewing the materials used to prepare this article and for helpful discussions on the related ideas and topics.

Footnotes:

1 https://en.wikipedia.org/wiki/Unspent_transaction_output

2 See https://bitcoinops.org/en/topics/covenants/ and https://arxiv.org/abs/2006.16714

3 https://blockstream.com/liquid/

4 (1,2) https://github.com/bitcoin/bips/blob/master/bip-0113.mediawiki

5 https://blockstream.com/simplicity.pdf

6 https://medium.com/blockstream/miniscript-bitcoin-scripting-3aeff3853620

7 With presented simple formula, D for the last repayment equals $P \bmod N$.

In most cases P will likely be much larger than N, and last repayment will be very small in this case. Simpler formula is easier for understanding, but for real application, it makes sense to just make the last repayment slightly bigger than others, and the more complex formula should be used:

$$D = \begin{cases} F_P * (m+1) & \text{if } (F_P * (m+1) + P \bmod N) \geq B \\ B & \text{otherwise} \end{cases}$$

[8] There can be a variant of the contract where the portions of the collateral are returned to the Debtor as the partial repayments are made, rather than at the end of the contract. This variant is not included in this particular specification.

[9] Historical discussion: https://bitcointalk.org/index.php?topic=1786.msg22119#msg22119

[10] Credit goes to Jonas Nick (https://twitter.com/n1ckler) for the idea of using unique assets in this way

[11] https://bitcoinops.org/en/topics/op_checksigfromstack/

[12] Described in Blocksream's "Confidential Assets" paper (https://blockstream.com/bitcoin17-final41.pdf)

[13] https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki

[14] https://bitcoinops.org/en/topics/transaction-pinning/

[15] https://bitcoinops.org/en/topics/cpfp-carve-out/

Appendix

Transaction schematics

In the diagrams, the "free-floating" part of outputs data (the explicit fee and the scriptPubKey of the last output) is sometimes shown at the top, and sometimes at the bottom. This is done only to improve the readability of the diagrams. The Debtor entity is always shown at the top, the Creditor at the bottom, and the outputs are oriented in a way to minimize visual clutter.

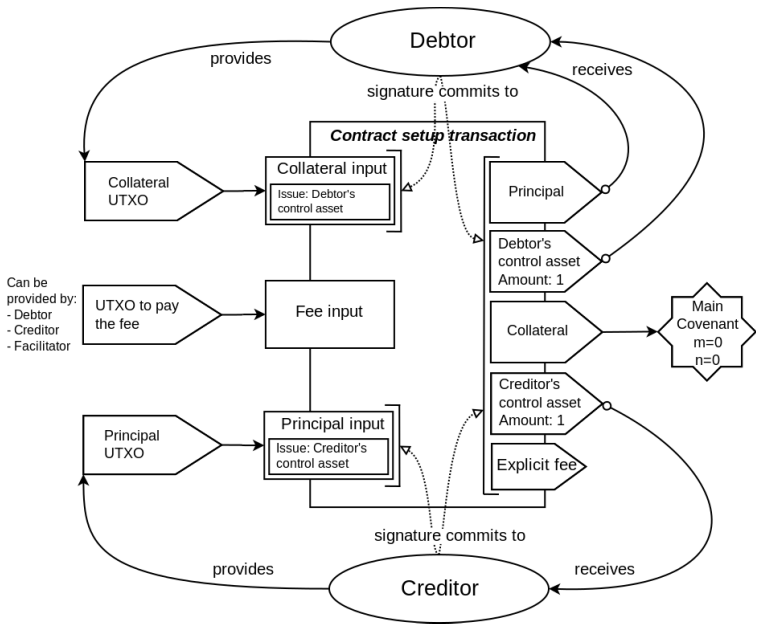
As discussed below in the "Security of the scripts" section, "committed" part of the outputs data in the actual transaction should always come first, and the "free-floating" part should always be the "suffix".

The input to pay the transaction fee can be provided by one of the participants, or by Facilitator as a convenience.

If the collateral asseet or the principal asset is suitable to pay the fee, the dedicated fee input can be omitted, and the fee can be deducted from the input that bears the suitable asset.

Contract deployment

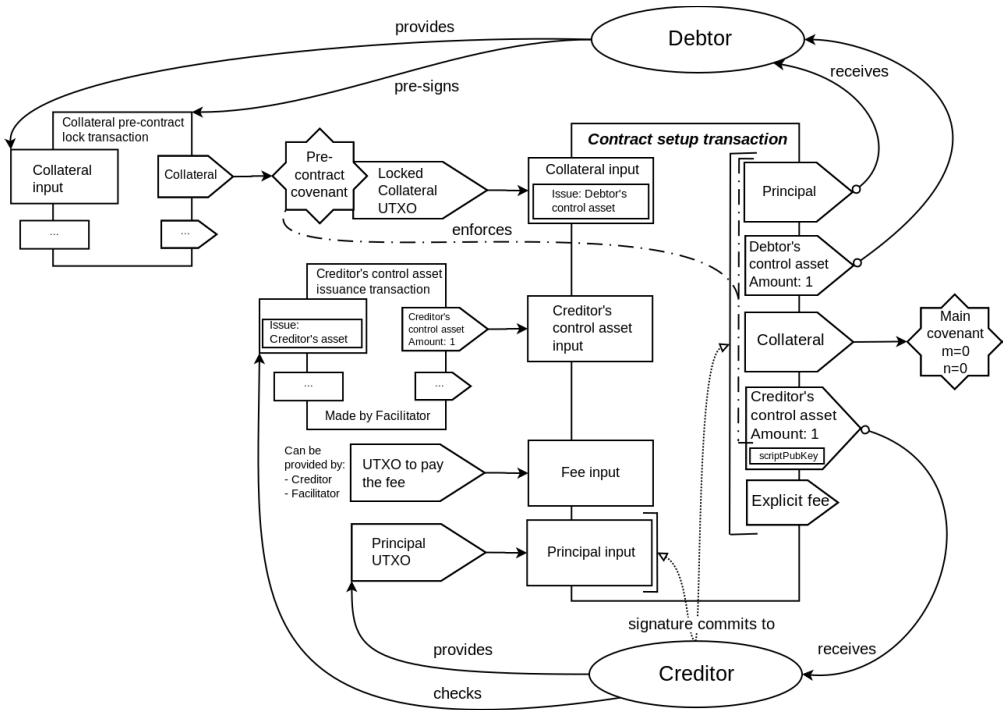
Interactive



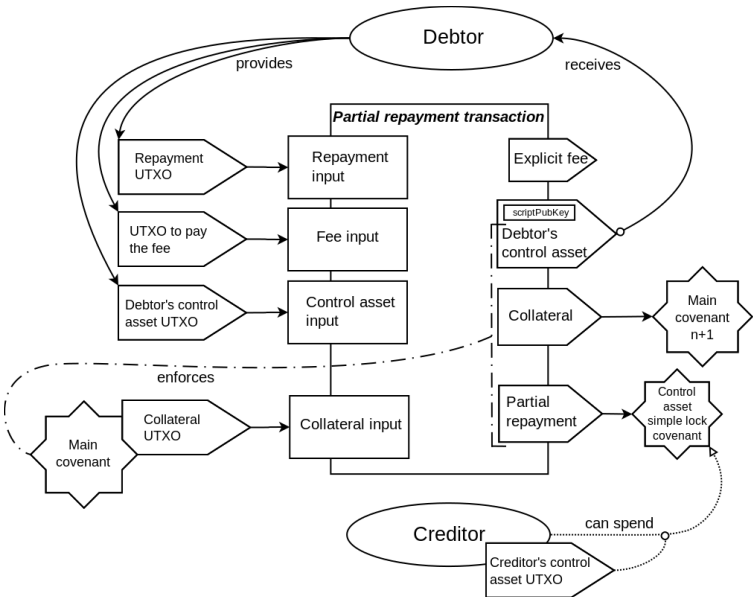
With a Facilitator, the Debtor and Creditor will need to use SIGHASH_ANYONECANPAY sighash type when signing their inputs, so that the Facilitator can combine their inputs independently. If the Facilitator is not involved, participants can sign using SIGHASH_ALL, so the signatures would cover all the inputs.

When SIGHASH_ANYONECANPAY is used for Debtor's and Creditor's inputs, and the fee input is present in the transaction, the transaction can be malleated by the block producer. Because the explicit fee output is committed to by the signatures, ordinary network nodes cannot replace the transaction using replace-by-fee mechanism. But the block producer can substitute fee input with its own UTXO with the same amount, and the transaction id will change. This is not an issue for the contract described here, because its behavior does not depend on the txid of the contract deployment transaction.

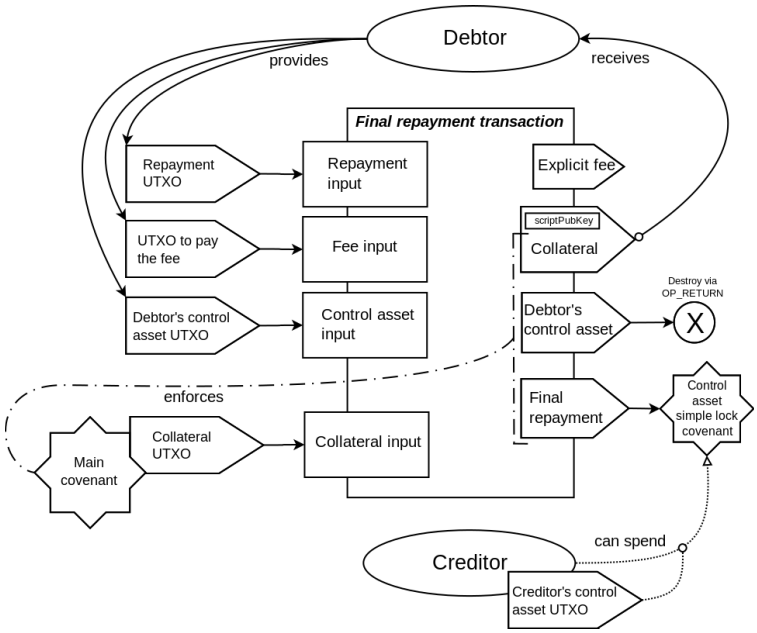
Non-interactive with Facilitator



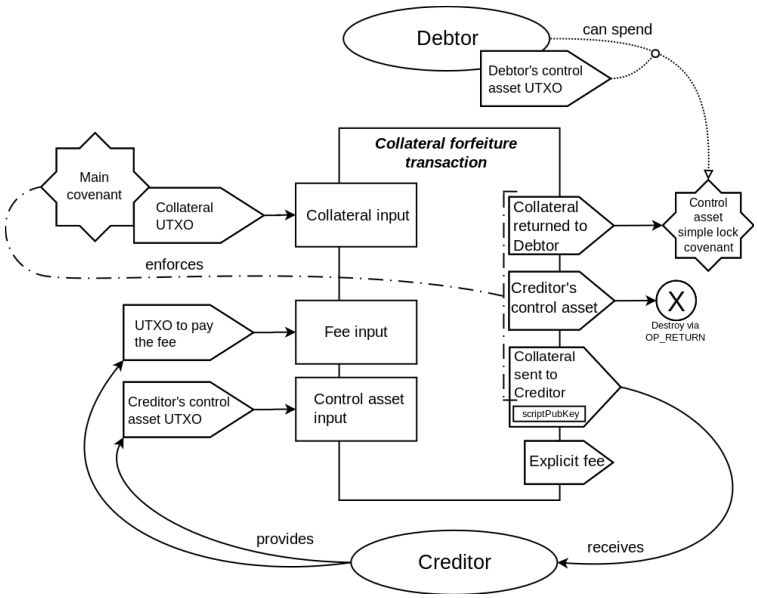
Partial repayment



Final repayment



Collateral forfeiture



In the case when the Debtor does not receive any part of collateral back, the corresponding output will not be included in the transaction.

Important point: the Creditor's control asset is destroyed in this transaction. Any partial repayment UTXO that is yet unspent will be rendered inaccessible after this transaction was broadcasted. It is crucual that all the partial repayment UTXO are spent before that (and sufficiently confirmed, so the ordering of the transactions cannot change).

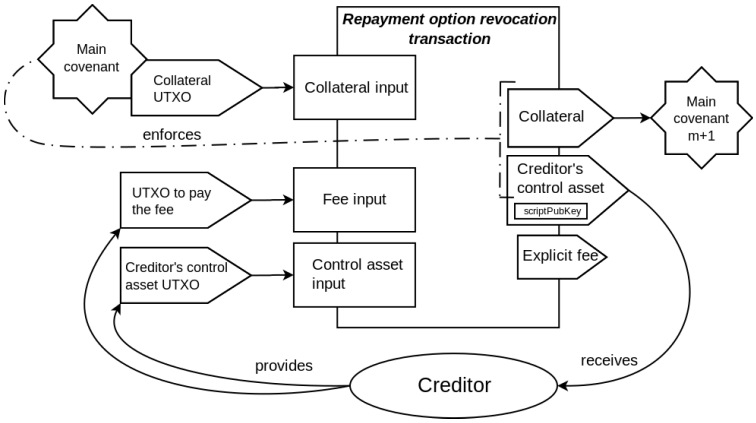
The alternative approach is to not destroy the Creditor's control asset in this transaction, but to send the collateral to the "Control asset simple lock covenant" controlled by the Creditor's control asset. The Creditor will then spend the collateral in a separate transaction.

An implementation will need to make a decision on which approach to use.

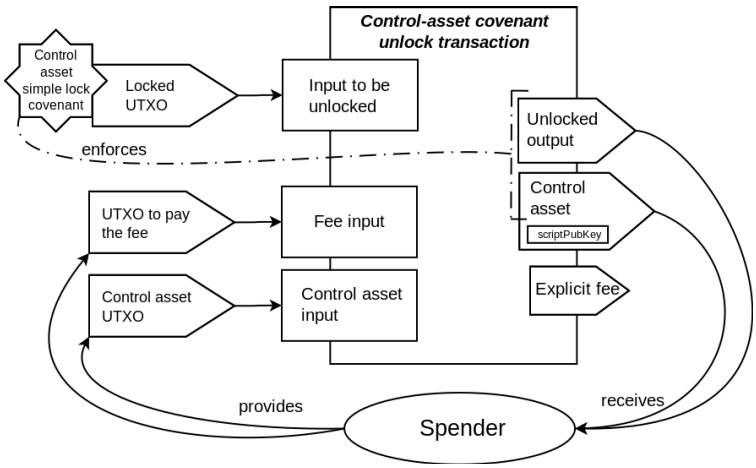
The second approach is more conservative. There might be a bug in the software where it fails to detect that there is

some partial repayment UTXO that is yet unspent. Keeping the control asset for longer might make the consequences of such bug not so severe. But at the same time, it costs one extra transaction. In addition to that, if the Creditor forgets to destroy the control asset afterwards, it will waste space in thie UTXO set. The size of the UTXO set is a limited shared resource of all the users of the blockchain and wasting it should be discouraged.

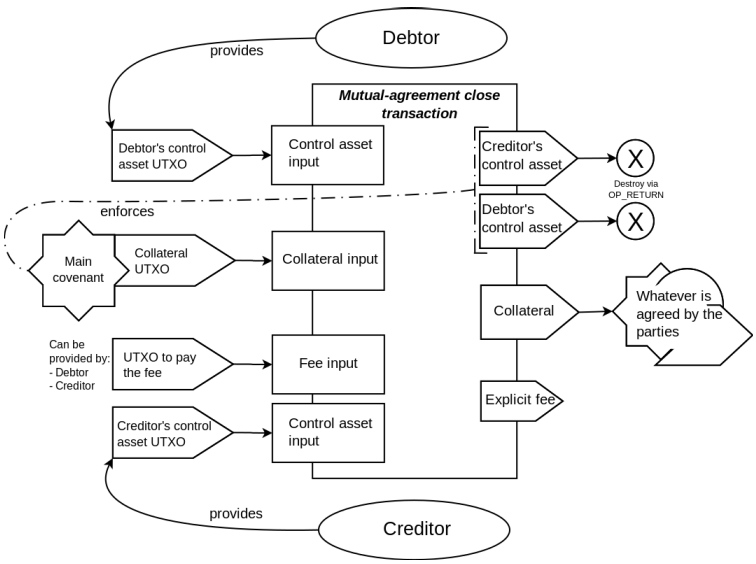
Repayment option revocation



Control asset simple lock covenant spending



Mutual-agreement contract close



The Debtor could also provide additional inputs, for example for the repayment of the remaining debt according to the new mutually-agreed terms. Then, there would also be additional output that would send that repayment to the Creditor.

Important point: the Creditor's control asset is destroyed in this transaction. Any partial repayment UTXO that is yet unspent will be rendered inaccessible after this transaction was broadcasted. It is crucual that all the partial repayment UTXO are spent before that (and sufficiently confirmed, so the ordering of the transactions cannot change). The notes on the "Collateral forfeiture" transaction diagram expand on this issue in more detail, and offer an alternative approach.

Covenant scripts

General description

As shown in the transaction schematics, most of the covenants in the described contract enforce the matching only on the part of the outputs data. One of the outputs is only partially committed to by these covenants. In particular, the scriptPubKey field is not committed to. This makes it possible to define the destination for that output at the spending time, not at the commitment time. The explicit fee output that is present in Elements transactions (but not in Bitcoin transactions) are not covered by the covenant commitment at all. This makes it possible to set the fee for the transaction at the spending time, and removes the difficulties related to trying to estimate the needed fee ahead of time, that is present in other covenant constructions that fix the fee at the commitment time.

When the covenant script is created, the transaction template is constructed, and then the outputs of the transaction template are blinded using a deterministic random source that is derived according to BIP32 from the extended key shared between the participants (the "shared blinding xkey"). This extended key is used to both derive the individual blinding keys for the outputs, and to derive the deterministic random data used in the blinding process itself. Different derivation paths are used for different purposes.

Not all outputs may be blinded. For example, it may be not worth the effort to blind the outputs of the control assets, given that these outputs only reveal the structure of the contract, that might also be revealed by another properties of the contract transactions.

The limitation of Elements script (that also exist in Bitcoin script) is that it is forbidden to construct the data chunk of longer than 520 bytes on the stack. The blinded outputs take much more space than non-blinded ones. With 520 byte limit on the data, only a few outputs in the transaction bounded by the described covenant can be blinded. Streaming SHA256 opcodes that are proposed to be included[16] in Elements can be a way to workaround this limitation.

During the covenant script construction, the blinded outputs are serialized as they would be serialized when building a transaction to broadcast to the network. The chunk of that serialized data is used as a template of the committed part of the outputs for the covenant. This chunk of data consist of all the whole outputs that are committed to, and the partial output that has the scriptPubKey excluded from the commitment. This data is then hashed with SHA256 hash algorithm. The resulting hash is used to ensure that the data supplied to the script at the spending time correspond to this "committed part" of the outputs data. Along with the "committed part", the "free-floating" chunk of outputs data is supplied in the input witness. This chunk have to contain the scriptPubKey for the partially-committed output, and any other outputs that the spender needs to add, such as explicit fee output.

The script then combines the "committed part" with the "free-floating part" to create the final image of the outputs data on the stack at run-time (or rather, spending time). This final image is SHA256-hashed, and then combined with other data supplied in the input witness. These additional data chunks are what the 'signature hash' algorithm uses to create the final 'signature hash', that the signatures in the transaction inputs commit to.

The routines

The script might need to commit to several possible variants of the outputs. For example, when the mutual-agreement case is allowed in the covenant, at least two variants for the committed outputs will be present, the primary covenant case, and the mutual-agreement case. This might be dealt with via the conditional execution using OP_IF/OP_ELSE /OP_ENDIF. But a more succinct way is to just include all the enabled hash images (each 32byte in length) in the script as a continuous data array (the length of which in bytes would be 32*n, where n is the number of options). The spender then supplies a position in this data array, and the script takes the 32bytes of the hash from this array at the supplied position using the OP_SUBSTR opcode (which is enabled in Elements).

```
OUTPUTS_HASH_LOOKUP_OPS defined as:

    # stack on entry:
    #   hashes_array (to be defined by the script)
    #   hash_position (to be supplied by the spender)
    OP_SWAP  # because the hashes_array comes from the script,
             # and the offset comes from the witness data,
             # they will be in reverse order, and we need to swap them first

    # stack:
    #   hash_position
    #   hashes_array
    5

    # stack:
    #   5
    #   hash_position
    #   hashes_array
    OP_LSHIFT # left-shift to 5 binary places, same as multiplying by 32

    # stack:
    #   offset
    #   hashes_array
    32

    # stack:
    #   32
    #   offset
    #   hashes_array
    OP_SUBSTR

    # stack:
    #   chosen_hash
```

When the hash is chosen from the hashes_array (or directly given in the script, if there is only one option), the data chunk that represents the outputs data portion that is committed to by the covenant has to be matched against this hash, and then combined with the "free-floating" part of the outputs data.

```
OUTPUTS_HASHCHECK_THEN_COMBINE_OPS defined as:

    # stack on entry:
    #    chosen_hash
    #    committed_outs_data_chunk
    #    free_floating_outs_data_chunk
    OP_OVER  # get the committed_outs_data_chunk to the top

    # stack:
    #    committed_outs_data_chunk
    #    chosen_hash
    #    committed_outs_data_chunk
    #    free_floating_outs_data_chunk
    OP_SHA256  # take SHA256 of committed_outs_data_chunk

    # stack:
    #    SHA256(committed_outs_data_chunk)
    #    chosen_hash
    #    committed_outs_data_chunk
    #    free_floating_outs_data_chunk
    OP_EQUALVERIFY  # check that the resulting hash matches the expected, fail otherwise

    # stack:
    #    committed_outs_data_chunk
    #    free_floating_outs_data_chunk
    OP_SWAP  # swap the data chunks to be in correct order for OP_CAT

    # stack:
    #    free_floating_outs_data_chunk
    #    committed_outs_data_chunk
    OP_CAT  # combine the data chunks to form the complete outputs data

    # stack:
    #    outputs_data
```

The covenant script dynamically constructs the signature hash from the data supplied in the input witness, and ensures that particular chunk of that data is the same as it was in the transaction template when the covenant script was constructed.

Then, OP_CHECKSIGFROMSTACKVERIFY is used to verify a signature over this constructed signature hash, with a signature that is partially supplied in the input witness. Then, the same signature is supplied to OP_CHECKSIG, which checks this signature against the signature hash that was constructed from the transaction currently being processed. This ensures that the current transaction matches the template transaction in the part that the covenant checks.

Note that OP_CHECKSIGFROMSTACKVERIFY do additional SHA256 hashing over the data supplied to it. This is because OP_CHECKSIG checks against a double-SHA256-hashed data. So technically, not a sighash, but a hash preimage of the final sighash is supplied to OP_CHECKSIGFROMSTACKVERIFY.

Signature hash commits to the serialized spending script. This means that when we create the sighash data dynamically, the data would need to include the covenant script itself, and this would make the witness data big, and the length of the data will likely exceed the 520 byte limitation for the stack item, that was discussed earlier. Fortunately, the OP_CODESEPARATOR opcode allows to split the script for the purposes of sighash commitment. Only the opcodes that come after OP_CODESEPARATOR are used in the construction of the signature hash. It makes sense to make this part minimal, and include only the OP_CHECKSIGFROMSTACKVERIFY + OP_CHECKSIG combination. Note that while the sighash does not commit to the whole script, the P2WSH scriptPubKey used for the covenant outputs commits to the whole script. Thus, the spending condition for the input will require the whole script to execute successfully.

```
POST_CODESEP_OPS defined as:

    # stack on entry:
    #   pubkey
    #   sighash_preimage
    #   signature
    #   pubkey (the same one)
    #   signature+SIGHASH_ALL (the same signature, but concatenated with a SIGHASH type byte)
    OP_CHECKSIGFROMSTACKVERIFY

    # stack:
    #   pubkey
    #   signature+SIGHASH_ALL
    OP_CHECKSIG

    # stack:
    #   TRUE if the current transaction matches the template, FALSE otherwise
```

Because the signature checks are used as a way to enforce the match between the transaction template and the actual transaction and not for spending authorization, it is not important what key is used for these signatures. It may be a widely known key. If the signature matches both the sighash constructed by the script and the sighash calculated by OP_CHECKSIG, this means that both hash values are equal.

Since we don't need to keep the key secret, the value of the nonce used in the signing process can also be arbitrary.

It is therefore makes sense to exploit the fact that there is an elliptic point for the secp256k1 curve with an anomalously small x coordinate, yet with a known logarithm[17] (this fact is likely a consequence of how the generator for the curve was chosen[18]). Using this value as both the nonce in the signing process and as a public key for the signing allows to save a bunch of bytes in the witness data.

```
PUSH_SMALL_X defined as:

    # stack on entry: <no arguments required>
    DATA("3b78ce563f89a0ed9414f5aa28ad0d96d6795f9c63")

    # stack:
    #    DATA("3b78ce563f89a0ed9414f5aa28ad0d96d6795f9c63")
```

When used as a public key, the value will be (encoded in hexadecimal): "020000000000000000000003b78ce563f89a0ed9414f5aa28ad0d96d6795f9c63".

The private key for this public key is calculated as $(order + 1)/2$ where $order$ is the order or the secp256k1 curve.

We can save additional 4 bytes in the witness by not pushing the 12-byte prefix "0200...00" on the stack just as a data value, but by constructing it instead.

```
PUSH_SMALLPUB_PREFIX defined as:

    # stack on entry: <no arguments required>
    2

    # stack:
    #    2
    1

    # stack:
    #    1
    #    2
    88  # 11*8

    # stack:
    #    88
    #    1
    #    2
    OP_LSHIFT # take the value "1" and left-shift it to 88 binary places
              # The result is the value 100000000000000000000000
              # That is encoded as little-endian bytestring 000000000000000000000001

    # stack:
    #    DATA("000000000000000000000001")
    #    2
    11

    # stack:
    #    11
    #    DATA("000000000000000000000001")
    #    2
    OP_LEFT  # Take 11 bytes of the constructed long value
             # so that only zeroes are left

    # stack:
    #    DATA("00000000000000000000")
    #    2
    OP_CAT  # Concatenate 2 (represented as the byte "02")
            # with a string of zeroes to construct the needed prefix

    # stack:
    #    DATA("020000000000000000000000")
```

To satisfy the spending conditions of the covenant, the spender needs to sign the sighash of the transaction with the key discussed above, and provide the S component of the signature in the input witness. The covenant script will construct the final signature from the known R component of the signature, the 4 bytes prefix that is also provided in the input witness, and the S component provided by the spender. One of those 4 bytes in the prefix will depend on the length of the S component, and thus it is more convenient to provide the whole 4 bytes as one witness stack item rather than construct it byte by byte.

The other data that the spender needs to provide is two chunks of the outputs data, the "committed part" and the "free-floating" part, and two data chunks representing the data that go into the signature hash before (the 'prefix') and after (the 'suffix') of the hashOuts.

There are also two fields that go into the 'suffix': the nLockTime field and the sighash type field. The later is added by the script itself, and the former is provided in the witness. It is important to check that the size of nLockTime data provided is exactly 4 bytes. Because the 'prefix' is supplied by the spender, allowing variable-sized data in the 'suffix' would enable an attacker to 'hide' the real hashOuts inside one of the fields in the 'prefix', and supply hashOuts of their liking.

The routine that checks the "committed part" against a chosen hash and the routine that combines the "committed part" with the "free-floating part" were presented above.

The following routine takes the outputs data already as a whole field. It prepares everything for the OP_CHECKSIGFROMSTACKVERIFY + OP_CHECKSIG pair in POST_CODESEP_OPS. This is the longest routine in the covenant scripts.

OUTPUTS_FINAL_CHECK_OPS defined as:

```
    # stack on entry:
    #    outputs_data  # data of the outputs, checked and combined with OUTPUTS_HASHCHECK_THEN_COMBINE_OPS
    #    locktime_data  # data representing the nLockTime field of the transaction
    #    sighash_data_prefix  # all the other data that go into sighash before the outputs:
    #                         # from nVersion to the asset issuances
    #    sig_prefix  # first 4 byte of the signature, including the byte that
    #                # depends on the length of the S component
    #    sig_suffix  # the S component of the signature with a couple of service bytes
    OP_HASH256  # The sighash commits to the hash of the outputs data. Do the hashing.
                # Note that HASH256 is SHA256(SHA256(data))

    # stack:
    #    hashOuts  # this is the HASH256(outputs_data)
    #    locktime_data
    #    sighash_data_prefix
    #    sig_prefix
    #    sig_suffix
    OP_SWAP  # Prepare the stack for OP_CAT, so that locktime_data is appended to hashOuts

    # stack:
    #    locktime_data
    #    hashOuts
    #    sighash_data_prefix
    #    sig_prefix
    #    sig_suffix
    OP_SIZE  # Get the size of locktime_data

    # stack:
    #    locktime_data_size
    #    locktime_data
    #    hashOuts
    #    sighash_data_prefix
    #    sig_prefix
    #    sig_suffix
    4  # will check that locktime_data_size is 4

    # stack:
    #    4
    #    locktime_data_size
    #    locktime_data
    #    hashOuts
    #    sighash_data_prefix
    #    sig_prefix
    #    sig_suffix
    OP_EQUALVERIFY  # Fail if locktime_data_size is not 4

    # stack:
    #    locktime_data
    #    hashOuts
    #    sighash_data_prefix
    #    sig_prefix
    #    sig_suffix
    OP_CAT  # Append locktime_data to hashOuts

    # stack:
    #    hashOuts+locktime_data
    #    sighash_data_prefix
    #    sig_prefix
    #    sig_suffix
    DATA('01000000')  # Push the value for SIGHASH_ALL on the stack.
                      # Note that sighash type takes 4 bytes in the sighash data,
                      # even if it is expressible in just 1 byte

    # stack:
    #    SIGHASH_ALL
    #    hashOuts+locktime_data
    #    sighash_data_prefix
    #    sig_prefix
    #    sig_suffix
    OP_CAT  # Append sighash type to the hashOuts+locktime_data chunk.
            # We now completed the suffix of the data that the sighash commits to
```

```
# stack:
#    sighash_data_suffix  # consists of: hashOuts+locktime_data+SIGHASH_ALL
#    sighash_data_prefix
#    sig_prefix
#    sig_suffix
OP_CAT  # Combine the sighash data prefix and suffix, and get the complete
        # sighash data on the stack

# stack:
#    sighash_data
#    sig_prefix
#    sig_suffix
OP_SHA256  # hash the sighash data once (one round of SHA256)
           # OP_CHECKSIGFROMSTACKVERIFY will do another round of SHA256,
           # so we now have the preimage of the sighash

# stack:
#    sighash_preimage
#    sig_prefix
#    sig_suffix
OP_TOALTSTACK   # move the sighash_preimage to alternative stack

# stack:
#    sig_prefix
#    sig_suffix
# altstack:
#    sighash_preimage
PUSH_SMALL_X  # get the x coordinate of the 'special elliptic point'

# stack:
#    SMALL_X
#    sig_prefix
#    sig_suffix
# altstack:
#    sighash_preimage
PUSH_SMALLPUB_PREFIX  # get the prefix for the 'special pubkey' on the stack

# stack:
#    DATA("020000000000000000000000")
#    SMALL_X
#    sig_prefix
#    sig_suffix
# altstack:
#    sighash_preimage
OP_OVER  # get SMALL_X to the top of the stack

# stack:
#    SMALL_X
#    DATA("020000000000000000000000")
#    SMALL_X
#    sig_prefix
#    sig_suffix
# altstack:
#    sighash_preimage
OP_CAT  # construct the 'special pubkey' on the stack

# stack:
#    pubkey
#    SMALL_X
#    sig_prefix
#    sig_suffix
# altstack:
#    sighash_preimage
OP_TOALTSTACK   # move the 'special pubkey' to alternative stack

# stack:
#    SMALL_X
#    sig_prefix
#    sig_suffix
# altstack:
#    pubkey
#    sighash_preimage
OP_CAT  # Construct the first part of the signature
```

```
# stack:
#    sig_prefix+SMALL_X
#    sig_suffix
# altstack:
#    pubkey
#    sighash_preimage
OP_SWAP  # Swap the two values for next OP_CAT

# stack:
#    sig_suffix
#    sig_prefix+SMALL_X
# altstack:
#    pubkey
#    sighash_preimage
OP_CAT  # Combine the first part of the signature to the second part, completing it

# stack:
#    signature
# altstack:
#    pubkey
#    sighash_preimage
OP_DUP  # We will need one signature for OP_CHECKSIGFROMSTACKVERIFY and one for OP_CHECKSIG

# stack:
#    signature
#    signature
# altstack:
#    pubkey
#    sighash_preimage
OP_1  # the sighash type in the signature itself takes only one byte, as opposed
      # to 4 bytes it takes within the sighash data.
      # OP_1 will push byte 01 to the stack, which corresponds to SIGHASH_ALL

# stack:
#    01
#    signature
#    signature
# altstack:
#    pubkey
#    sighash_preimage
OP_CAT  # Append the byte 01 to the signature, so we will have the signature with
        # sighash type for OP_CHECKSIG, and without it, for OP_CHECKSIGFROMSTACKVERIFY

# Now we need to prepare the stack arguments for POST_CODESEP_OPS

# stack:
#    signature+SIGHASH_ALL
#    signature
# altstack:
#    pubkey
#    sighash_preimage
OP_FROMALTSTACK  # get the 'special pubkey' from altstack

# stack:
#    pubkey
#    signature+SIGHASH_ALL
#    signature
# altstack:
#    sighash_preimage
OP_ROT  # Rotate the top 3 items on the stack so that signature+SIGHASH_ALL becomes last

# stack:
#    signature
#    pubkey
#    signature+SIGHASH_ALL
# altstack:
#    sighash_preimage
OP_OVER  # Duplicate the pubkey from the second position on the stack

# stack:
#    pubkey
#    signature
#    pubkey
#    signature+SIGHASH_ALL
```

```
    # altstack:
    #    sighash_preimage
    OP_FROMALTSTACK  # Retrieve sighash_preimage from the stack

    # stack:
    #    sighash_preimage
    #    pubkey
    #    signature
    #    pubkey
    #    signature+SIGHASH_ALL
    OP_SWAP  # Adjust argument positions to match what is expected by POST_CODESEP_OPS

    # stack:
    #    pubkey
    #    sighash_preimage
    #    signature
    #    pubkey
    #    signature+SIGHASH_ALL
    OP_CODESEPARATOR  # Everything after this opcode will be included in the sighash data.
                      # The full script including everything before and after this opcode
                      # will be included in the input witness.

    # stack:
    #    pubkey
    #    sighash_preimage
    #    signature
    #    pubkey
    #    signature+SIGHASH_ALL
    POST_CODESEP_OPS  # Do the checking via OP_CHECKSIGFROMSTACKVERIFY + OP_CHECKSIG
```

Main covenant script

The script has two execution paths - for the lateral state transition, and for the vertical state transition. The execution path are chosen by the top value on the witness stack. If the value is 0, the lateral state progression path is chosen. If the value is 1, the vertical state progression path is chosen. The values larger than 1 will mean the same as 1. Using other values than 1 may result in different size of witness data, which can influence the transaction ordering in the mempool. But this is possible only when the transaction standardness rules are not enforced (the MINIMALIF rule). This is not an issue, because only blocksigners can bypass the standardness rules, and blocksigners already directly determine the transaction ordering.

```
MAIN_COVENANT_OPS defined as:

    # stack on entry:
    #
    #    for lateral state progression:
    #        0
    #        offset_into_hash_array
    #        committed_outs_data_chunk
    #        free_floating_outs_data_chunk
    #        locktime_data
    #        sighash_data_prefix
    #        sig_prefix
    #        sig_suffix
    #
    #    for vertical state progression:
    #        1
    #        committed_outs_data_chunk
    #        free_floating_outs_data_chunk
    #        locktime_data
    #        sighash_data_prefix
    #        sig_prefix
    #        sig_suffix
    OP_IF

    # stack:
    #    committed_outs_data_chunk
    #    free_floating_outs_data_chunk
    #    locktime_data
    #    sighash_data_prefix
    #    sig_prefix
    #    sig_suffix
    <timeout>  # the calculated timeout in blocks for when the vertical
```

```
            # state progression becomes allowed

    # stack:
    #    timeout
    #    committed_outs_data_chunk
    #    free_floating_outs_data_chunk
    #    locktime_data
    #    sighash_data_prefix
    #    sig_prefix
    #    sig_suffix
    OP_CHECKLOCKTIMEVERIFY  # enforce the timelock

    # stack:
    #    timeout
    #    committed_outs_data_chunk
    #    free_floating_outs_data_chunk
    #    locktime_data
    #    sighash_data_prefix
    #    sig_prefix
    #    sig_suffix
    OP_DROP  # timeout was not dropped by OP_CHECKSIGFROMSTACKVERIFY, drop explicitly

    # stack:
    #    committed_outs_data_chunk
    #    free_floating_outs_data_chunk
    #    locktime_data
    #    sighash_data_prefix
    #    sig_prefix
    #    sig_suffix
    DATA(<hash_for_revocation>)  # the hash for the committed part of outputs
                                 # for the payment option revocation case

    OP_ELSE  # The lateral progression case

    # stack:
    #    offset_into_hash_array
    #    committed_outs_data_chunk
    #    free_floating_outs_data_chunk
    #    locktime_data
    #    sighash_data_prefix
    #    sig_prefix
    #    sig_suffix
    DATA(<hashes_array>)  # The hashes_array will actually consist of up to three hashes:
                          # the hash of committed part of outputs for partial repayment,
                          # the hash of committed part of outputs for early full repayment,
                          # and the hash of committed part of outputs for mutual close case.

    # stack:
    #    hashes_array
    #    offset_into_hash_array
    #    committed_outs_data_chunk
    #    free_floating_outs_data_chunk
    #    locktime_data
    #    sighash_data_prefix
    #    sig_prefix
    #    sig_suffix
    OUTPUTS_HASH_LOOKUP_OPS

    OP_ENDIF  # branch finished

    # stack:
    #    outputs_hash
    #    committed_outs_data_chunk
    #    free_floating_outs_data_chunk
    #    locktime_data
    #    sighash_data_prefix
    #    sig_prefix
    #    sig_suffix
    OUTPUTS_HASHCHECK_THEN_COMBINE_OPS

    # stack:
    #    outputs_data
    #    locktime_data
    #    sighash_data_prefix
```

```
    #    sig_prefix
    #    sig_suffix
    OUTPUTS_FINAL_CHECK_OPS
```

Control asset spending covenant script

This is the covenant that allows the delegated spending. The repayments to the Creditor and possible partial collateral return to Debtor are spent by satisfying this script.

```
CONTROL_ASSET_SPEND_OPS defined as:

    # stack on entry:
    #    free_floating_outs_data_chunk
    #    locktime_data
    #    sighash_data_prefix
    #    sig_prefix
    #    sig_suffix
    DATA(<control_asset_output_sans_scriptpubkey>)

    # stack:
    #    committed_outs_data_chunk  # the control asset output data, sans scriptPubKey
    #    free_floating_outs_data_chunk
    #    locktime_data
    #    sighash_data_prefix
    #    sig_prefix
    #    sig_suffix
    OP_SWAP  # Swap the two values for next OP_CAT

    # stack:
    #    free_floating_outs_data_chunk
    #    committed_outs_data_chunk
    #    locktime_data
    #    sighash_data_prefix
    #    sig_prefix
    #    sig_suffix
    OP_CAT  # Combine the outputs data chunks to get complete outputs data

    # stack:
    #    outputs_data
    #    locktime_data
    #    sighash_data_prefix
    #    sig_prefix
    #    sig_suffix
    OUTPUTS_FINAL_CHECK_OPS
```

Security of the scripts

The scripts construct the signature from the supplied S component of the signature and known value for R component. The pubkey is fixed. The risk of the attack via this particular input data is the same as the risk for the attack on the signature verification code in the Elements client. Elements is based on Bitcoin Core codebase, and this particular part of functionality does not deviate from the upstream behavior. Given that the incentives to attack Bitcoin signature functionality is enormously high, the level of risk that such a basic functionality will be broken in Elements is extremely low.

The scripts construct the signature hash from the inputs. Only a part of this data is checked to be as expected. Part of outputs, and most of sighash data are supplied by the spender.

The covenants are built on the notion that only a part of outputs data is committed to by the covenant script. The spender is allowed to attach extra inputs and add extra outputs. This might be useful to extend the contract or combine it with another contracts. The assets, amounts and (except for where scriptPubKey is not committed) the destinations of the committed outputs will be as expected, and the contract terms will be enforced.

It is important that the committed-to outputs will be the first. If the attacker is able to put their arbitrary outputs before the committed-to outputs, they can craft their data such that their last custom output not complete: the data chunk in the witness would contain only OP_RETURN opcode, but the size of scriptPubKey would be much bigger. In the transaction itself, this output will contain OP_RETURN followed by the "committed-to" part of the outputs data. The Attacker's outputs that completely bypass the covenant would go before that custom output.

Adding extra data means that participants can construct the transactions of different sizes. But because it is the spender

that will pay the fee for this extra size, this is not a concern.

The spender can supply the prefix for the sighash data, as well as nLockTime field. The size of nLockTime field is checked by the script, while the size of the prefix data is not.

The attack might be imagined where the additional data influences the final hash of outputs in such a way that the expected, "enforced" outputs do not correspond anymore to this influenced hash, but another set of unrelated outputs are enabled. Would the hash function used for sighash calculation be a weak one, we could talk about the possibility of a chosen prefix collision. But even then, such attack would be hindered by the fact that this spender-supplied data would need to correspond to a valid transaction that the spender can actually construct. There's no known collision attacks on SHA256 hash function at the moment.

Footnotes:

[16] https://github.com/ElementsProject/elements/pull/817

[17] https://crypto.stackexchange.com/questions/60420/what-does-the-special-form-of-the-base-point-of-secp256k1-allow

[18] https://bitcointalk.org/index.php?topic=289795.msg3183975#msg3183975