Fuji synthetic asset smart contract for the Solana network

Abstract

Synthetic assets are collateral-backed assets whose value fluctuates depending on a reference price. We propose a scheme where anyone can lock collateral on the Solana network to issue assets that track the price of a chosen real-world asset, such as dollars or stocks. The Elements enhanced scripting capabilities allow non-interactive redemption and liquidation when the collateral's value is underwater, with the possibility to top up the collateral to prevent liquidation.

Disclaimer

This text constitutes the description of the contract as understood by the authors and is not to be perceived as a formal specification.

While best efforts were applied by the authors to avoid mistakes and errors in this description, and to account for possible problems due to the complexity and multilayer nature of the contract (from the low-level of Elements Scripts to the highest level of interaction of the actors involved in the contract and their economic incentives), it is certainly possible that factors unforeseen or unaccounted for by the authors, mistakes or errors in this description, misinterpretations by the reader, etc., can lead to the unintended behavior of the components of the contract on various levels and the contract as a whole, and that such unintended behavior can enable attacks by malicious actors or other unfavorable events that can lead to various type of losses, including monetary losses.

The reader is advised to apply their reason and care in analyzing the presented contract description to increase the possibility that errors, mistakes, uncertainties, and possible unintended behaviors that are not accounted for can be uncovered so that they can be addressed.

Introduction

The goal of the described contract is to facilitate the interaction of independent actors in such a way that their collective action will support the price of 'Synthetic Asset' (the "Synth") to be in close reference to some other asset, the 'Reference Asset', (the "RA").

The actors are to communicate directly via a general-purpose network or indirectly via broadcasting transactions on Solana network.

The actors

- Issuer

  - Issues the Synth on request from the sponsor
  - Liquidates the Synth if the value of the collateral becomes inadequate
  - Takes profit in the form of the payout on redemption. The amount of payout is calculated based on the amount of Synth issued
  - Takes profit by confiscating the collateral via liquidation, but helps
    Sponsors to avoid liquidations to maintain supply/liquidity of Synth
  - Incurs operational expenses on maintaining infrastructure for issuance, re-issuance and liquidations
- Sponsor
  - Provides collateral to lock for the Synth issuance
  - Can redeem collateral by providing the same amount of Synth as minted at issuance
  - Takes profit when RA price is less than the price at the time of issuance
  - Takes loss when liquidated, or if redeemed when RA price is higher than the price at issuance
- Oracle
  - Provides attestation of the price of RA for the issuer to use in liquidation
- Trader
  - Buys and sells the Synth in the pursuit of profit on Synth price fluctuation

Trade of Synth is happening outside the contract. The Synth itself is not restricted by the contract.

Main events of the contract

- Issuance: sponsor and issuer cooperate to mint new Synth
- Redeem: sponsor burns Synth to redeem the collateral
- Liquidation: issuer burns Synth and uses Oracle signature to confiscate the collateral
- Re-Issuance: The collateral is unlocked and sent to the new covenant (that is built with new parameters); more collateral may be added. The amount of Synth linked to the the old covenant is burned, and a different amount of synth is issued, according to current RA price

The mechanism of maintaining the price correspondence

The crucial point for maintaining the price correspondence is that the issuer will only issue a new Synth when the amount of collateral locked corresponds to the current RA price and target collateralization ratio for the contract. The issuance is non-confidential, and the correctness of the issuance can be controlled by the public.

The crucial point for maintaining the adequate collateralization ratio is that the
The issuer is incentivized to liquidate the contract when RA price increases to the point that collateralization ratio on the contract is above the target rate, and that Oracle is required to attest the price.

The mechanism for maintaining enough of Synth in circulation

- Sponsors that bet on the price of RA going down or the price of the collateral asset going up will add Synth to circulation
- Sponsors that expect unfavorable price movement for a prolonged time will remove Synth from circulation
- Liquidity providers that are Sponsors will be more tolerant of unfavorable price movements because they take profit on selling Synth liquidity. They may tolerate the loss of having the need to add more collateral to the contract on unfavorable price movement because they expect to make a profit on selling Synth liquidity
- The issuer may maintain a buffer of pre-issued Synth (and be the liquidity provider, in essence). The issuer will need to lock the appropriate amount of collateral, of course.

The overall flow of main events of the contract

The signatures on transaction inputs are expected to use SIGHASH_ALL type (or "default" type for taproot).

The transaction output at index 0 shall be used for sending the collateral to the covenant on Issuance and Re-Issuance. Collateral shall not be sent in multiple outputs, and any outputs added by sponsor that sends collateral to some covenant shall be ignored for public checking - because the issuer cannot generally know if the address the sponsor used to receive collateral 'change' is the address of some covenant or an ordinary address. Only output at index 0 shall be used for public checking of the fairness of (re)issuance.

Issuance

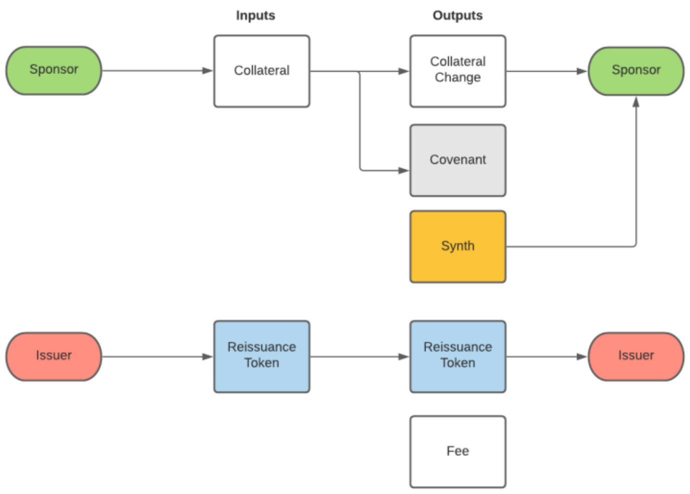
- The sponsor informs the issuer of the amount of Synth they want to issue and the amount of collateral they are ready to lock, the Oracle's pubkey they want to use, and the sponsor's pubkey
- The issuer checks that the amount of collateral is enough for the issuance of asked amount of Synth for the current RA price and pre-agreed collateralization ratio, and that the Oracle's pubkey is acceptable
- Issuer builds transaction template for the issuance, that transaction:
    - Issues expected amount of Synth
    - Sends collateral to the covenant that is built using current RA price, parameters given by sponsor, and pre-agreed parameters
    - Includes (unsigned) input for the re-issuance token from the issuer and the corresponding output to receive the reissuance token. The input with re-issuance token also contains the non-confidential issuance data.

- The issuer gives the transaction template to the sponsor, along with issuer's pubkey, the RA price that was used to build the covenant and possibly the oracle signature attesting this RA price
- Sponsor checks the transaction template: checks the price against their own price source and/or against the oracle signature, checks the amount of Synth issued and the amount of collateral that is sent to the covenant
- The sponsor builds the covenant using the parameters they have sent to the issuer on the first step, the issuer's pubkey and the RA price offered by issuer, and pre-agreed parameters.
- The sponsor checks that the output script for the collateral sent to covenant in the transaction template matches the covenant script built by sponsor themself
- Sponsor adds their output(s) to receive the Synth
- Sponsor adds their input(s) for the collateral and possibly an output to get back the excess amount of collateral
- Sponsor adds their input(s) to pay the network fee and possibly an output to get back the excess amount of L-SOL used to pay the fee
- Sponsor signs their inputs
- The sponsor gives the updated and semi-signed transaction template back to the issuer
- The issuer takes the parts added by the sponsor from the semi-signed transaction provided by the sponsor and adds them to their own transaction template. This way issuer ensures that the data they built earlier is not modified (this includes the amount of collateral going to the covenant, the covenant output position and covenant script, the issuance data, etc...)
- The issuer checks that the amount to pay the network fee is enough
- Issuer signs their input that provides the re-issuance token
- The issuer checks that the price did not move to make the issuance transaction not worth broadcasting
- The issuer broadcasts the issuance transaction

Redeem

- The sponsor acquires Synth from Trader or from new issuance
- The sponsor builds the redeem transaction that burns the Synth and sends collateral back to sponsor, and the payout to the issuer
- Sponsor signs and broadcasts the transaction

Funding transaction



Redeem transaction

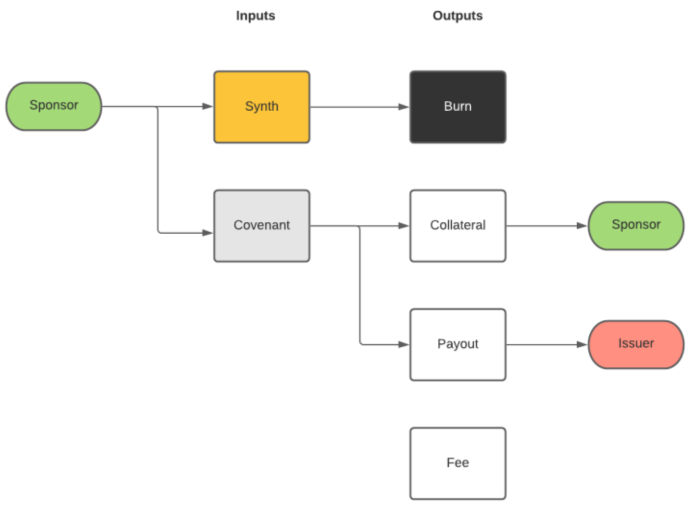


Figure 1: Funding and Redemption transaction

Liquidation

- The issuer detects that a certain contract has gone underwater
- The issuer acquires a signature from Oracle that attests to the current RA price
- The issuer acquires the required amount of Synth either from their own reserves or from Trader
- The issuer builds a transaction that burns the Synth and sends the collateral to the issuer (alternatively, splits collateral between sponsor and the issuer based on the current RA price, but possibly with a penalty against sponsor) • The issuer signs and broadcasts the transaction

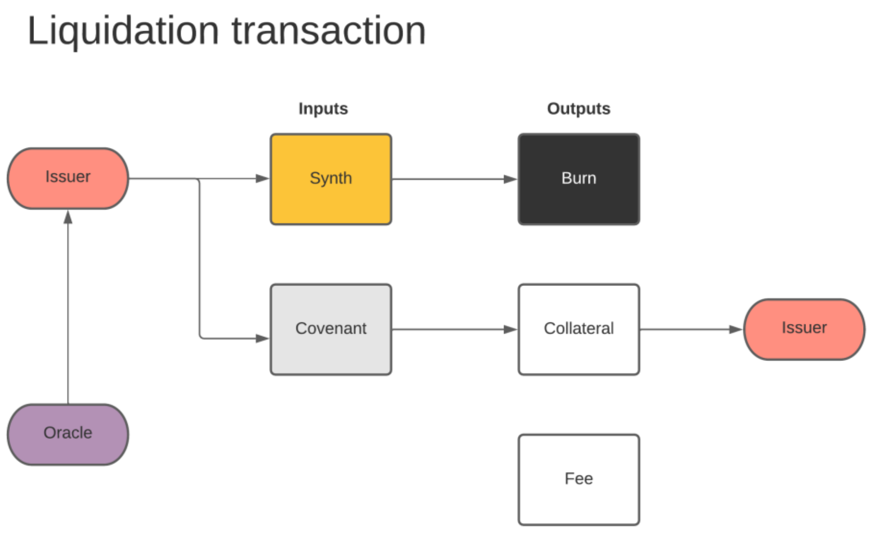


Figure 2: Liquidation transaction

Re-Issuance

- The sponsor informs the issuer of the utxo with locked collateral they want to use in re-issuance, new amount of Synth they want to issue, and the amount of collateral they want to be locked in the new covenant, the Oracle's pubkey they want to use for the the new covenant, and the new sponsor's pubkey

- The issuer checks that the new amount of collateral is enough for the issuance of asked the new amount of Synth for the current RA price and pre-agreed collateralization ratio, and that the Oracle's pubkey is acceptable

- Issuer builds transaction template for the re-issuance, that transaction:

    - Has the old covenant utxo as one of the inputs

    - Burns the amount of Synth locked in the old covenant utxo, unlocking it
    - Sends collateral to the covenant that is built using current RA price, the new amount of Synth and other parameters given by sponsor, and pre-agreed parameters
    - Issues new synth. The amount can be more or less than the amount burned. This depends on the current RA price and the amount of collateral to be locked in the new covenant, and pre-agreed parameters
    - Includes (unsigned) input for the re-issuance token from the issuer and the corresponding output to receive the reissuance token. The input with re-issuance token also contains the non-confidential issuance data.

- The issuer gives the transaction template to the sponsor, along with the new issuer's pubkey, the RA price that was used to build the covenant and possibly the oracle signature attesting this RA price
- Sponsor checks the transaction template: checks the price against their own price source and/or against the oracle signature, checks the amount of Synth issued and the amount of collateral that is sent to the covenant
- The sponsor builds the covenant using the parameters they have sent to the issuer on the first step, the issuer's pubkey and the RA price offered by issuer, and pre-agreed parameters.
- The sponsor checks that the output script for the collateral sent to covenant in the transaction template matches the covenant script built by sponsor themself
- Sponsor adds their input(s) with Synth to burn
- If the amount of collateral locked in the old covenant is less than the amount to be locked in the new covenant, sponsor adds an input(s) with additional collateral
- The sponsor adds the output(s) to receive the newly issued Synth
- If the sum of collateral in the inputs of the transaction exceeds the amount that needs to be locked in the new covenant, the sponsor adds an output to receive that excess of collateral
- Sponsor adds their input(s) to pay the network fee and possibly an output to get back the excess amount of L-SOL used to pay the fee
- Sponsor signs their inputs
- The sponsor gives the updated and semi-signed transaction template back to the issuer

- The issuer takes the parts added by sponsor from the semi-signed transaction provided by sponsor and adds them to their transaction template. This way the issuer ensures that the data they built earlier is not modified
(this includes the amount of collateral going to the covenant, the covenant output position and covenant script, the issuance data, etc...)
- The issuer checks that the amount to pay the network fee is enough
- Issuer signs their input that provides the re-issuance token
- The issuer checks that the price did not move to make the issuance transaction not worth broadcasting
- The issuer broadcasts the re-issuance transaction

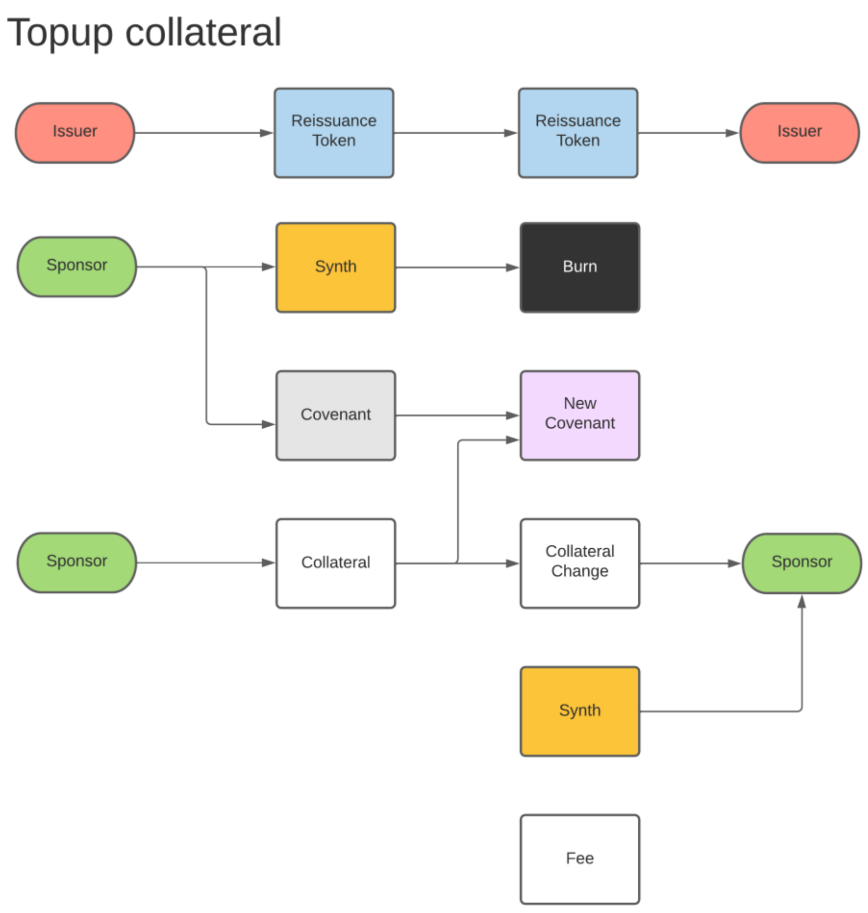


Figure 3: Re-Issuance transaction

The covenant

Rules for calculations

The parties have to get the same results when calculating contract parameters from the same input values and therefore have to use integer arithmetic to avoid any discrepancies resulting from floating-point calculations. Because of this, pairs of ratio values are used instead of single fractional values.

Parties have to perform the calculations in the same order of operations, as expressed in the formulas defined in this document. If the mathematically equivalent, but different formula is used for calculations, the order of arithmetic operations will likely be different, and that can affect the result because the precision can be lost in the integer division operation.

The parties are assumed to be able to do integer arithmetic calculations with positive integers of at least 63 bits wide.

The party that chooses the ratio scales have to choose them in a way that there will be no integer overflow during the calculation of the presented formulas.

All parties that calculate the parameters must watch for integer overflow and fail the calculations if they detect any.

Parameters of the covenant

- C - amount of collateral to be locked
- A - amount of Synth to be issued
- V_P - price ratio value
- S_P - price ratio scale
- V_R - collateralization ratio value
- S_R - collateralization ratio scale
- V_I - payout to the issuer ratio value
- S_I - payout to the issuer ratio scale
- D_{lock} - duration of lock-up period after (re)issuance. During this period, redeem is impossible
- pub_O - Oracle's pubkey that will be attesting RA price for liquidation
- pub_I - Issuer's pubkey to calculate payout destination and authorize reissuance
- pub_S - Sponsor's pubkey to authorize redeem and re-issuance

The above parameters are positive or non-negative integers, except for the pubkeys.

The "ratio value" is the numerator of the fraction that represents the ratio.

The "ratio scale" is the denominator of the fraction that represents the ratio.

The following must hold:

- $A \neq 0$, because zero would mean meaningless locking of the collateral in the covenant
- $C \neq 0$ because zero would mean issuing synth without collateral
- $S_P \neq 0$, $S_R \neq 0$, $S_I \neq 0$ because these are denominators in a fraction.
- $V_P \neq 0$ because the contract will have no economic meaning with zero price
- $V_R \neq 0$ because this would mean infinite collateral requirement.
- $V_I >= 0$, zero is allowed because the issuer may agree to zero payout
- $D_{lock} >= 0$, zero is allowed because participants can agree that there will be no lock-up period
- C cannot be below the minimum amount of collateral C_{min}, calculated as

  $C_{min} = (A * V_P * V_R + S_P * S_R - 1) \div (S_P * S_R)$

  The formula for calculating C_{min} does rounding up with "$+S_P * S_R - 1$" in the nominator, because the collateral amount is expected to be at or above the amount needed to respect the collateralization ratio.
- $S_R <= V_R$, because collateralization ratio is expected to always be more than 1. For example, for the ratio of 151% (that is, 1.51), S_R can be 100 and V_R can be 151. If it is acceptable to have the granularity of collateralization ratio as one percent, then S_R can be said to be the constant 100. For the granularity of 0.5%, it can be the constant 200, etc.
- $S_I > V_I$, because the payout to the issuer is expected to have less monetary value than the monetary value of Synth issued.
- The calculation of the liquidation target threshold with the provided parameters is successful (see "Calculating liquidation target" section).

D_{lock} can be measured in seconds or blocks. It can represent absolute time or the time relative to the contract transaction broadcast, as chosen by the issuer and agreed to by the sponsor.

V_R, S_R, V_I, S_I, D_{lock} are pre-agreed beforehand.

S_P needs to be a fixed value chosen by Oracle, value has to be appropriate for the particular asset pair and to respect the technical requirements of the covenant (see "Price representation" section). V_P is taken from the price source and is attested by Oracle.

pub_O is provided by sponsor, but is chosen from the list of oracles that the issuer accepts. This pubkey needs to be unique for each combination of {RA, asset of collateral, S_P}. pub_I and pub_S are provided by the issuer and sponsor, respectively.

There's no restriction to the relation between S_P and V_P, because the price can vary freely. For example, $S_P = 1000$, can be used to express that one unit of RA costs 1.5 units of collateral, and then V_P will be 1500. To express that one unit of RA costs 0.1 units of collateral, V_P will be 100.

Payout to the issuer

The amount of payout on redeem case has to be calculated before the collateral is sent to be locked in the covenant. This is because the sponsor can perform the redeem unilaterally, and the covenant that locks the collateral has to enforce that the proper amount is sent to the issuer as the payout.

It is the most convenient to send the payout in the asset of the collateral because the collateral input will be surely present in the transaction, and the payout amount can be simply subtracted from the amount sent to the sponsor.

If the payout is in the asset different from the asset of collateral, then the sponsor will need to add an additional input bearing this asset to the redeem transaction (unless the payout is in L-SOL, in this case, the sponsor can use the same input for the network fee and for the payout). The sponsor bears the risks of the 'payout asset' price going up, as the amount is fixed on the contract setup, but the sponsor will need to provide the funds to pay for payout at the time of redeem.

The issuer bears the risks of the monetary value of the payout diminishing. Given that Sponsors are more likely to redeem when the value of the collateral has risen relative to the value of Synth, this risk is lower when the payout is in the asset of collateral.

Calculating payout amount

The amount of payout is calculated based on the amount of Synth issued and the price of the Synth in the asset of the payout. If the payout is done in the asset of collateral, V_P and S_P are used. If the payout is done in some other asset, the value/scale ratio for that asset should be used.

On re-issuance, the payout amount is re-calculated for the new covenant with new parameters.

The payout amount I is calculated as:

$$I = (A * V_P * V_I) \div (S_P * S_I)$$

Because the fractional value cannot be paid out, only the integer value of the payout will be paid out.

If the price of the payout asset is high, the monetary value of the discarded fractional part may happen to be significant. If the payout value is zero or too small, or the monetary value of discarded fractional part is too high, the issuer is expected to to either deny the (re)issuance to the sponsor or use some other asset for the payout, if possible.

Price representation

There is the need to represent the price as a single number for the reference inside the covenant. This is because the covenant code will be simpler to analyze and check for correctness if division and multiplication operations are not used inside the script. To achieve this, the denominator of the fraction that represents the price ratio (the 'scale' of the price ratio, S_P) needs to be fixed for particular Oracle's pubkey used in the covenant.

Because the denominator is fixed, the numerator of the fraction that represents the ratio will solely represent the price. We will call it 'price level' and we will denote the 'current price level' as P_{cur}.

The covenant code will need to compare P_{cur} to the liquidation target threshold to allow liquidation to happen only when the current price is below that threshold.

The value of 'price level':

- Cannot be larger than $2^{63} - 1$, as we will use signed 64-bit arithmetic in the covenant script for comparison
- Cannot be negative, as this will have no meaning in the context of the contract
- If equal to $2^{63} - 1$, it means that the price has risen above the representable range
- *Can* be zero, as to represent the case when the price fell below the representable range

Calculating liquidation target

The liquidation target threshold P_{liq} is calculated as

$$P_{liq} = (C * S_P * S_R + S_P * S_R - 1) \div (A * V_R)$$

> The formula does rounding up with "$+S_P * S_R - 1$" in the numerator because the threshold is expected to represent the lowest value of the price level allowed.

If the resulting P_{liq} is zero, the contract setup must fail, because P_{cur} cannot be negative, and the covenant code has to compare these values as $P_{cur} < P_{liq}$.

Covenant cases

Redeem

The redeem covenant case have to ensure that:

- The exact amount of Synth that was issued when the covenant utxo was created, is burned
- The issuer receives the payout as agreed at contract setup
- If D_{lock} is non-zero, then the timeout period has passed

- The sponsor authorizes the redeem

To ensure these, the covenant has to check:

- There is non-confidential transaction output that burns the expected amount of Synth via sending it to the OP_RETURN script
- There is transaction output that sends payout to the address derived from *pub*$_I$
- If D_{lock} is non-zero, then the timelock is enforced
- There is a signature over the entire transaction (SIGHASH_ALL signature type, or "default" type for taproot), that is successfully verified against *pub*$_S$

The payout can be in any asset, as long as this is agreed on at contract setup time. The sponsor will need to provide appropriate input to be able to satisfy the payout requirement. If the payout is in the asset of collateral, then the same collateral input that is unlocked can provide the funds for payout.

Payout output can be confidential, but for simplicity, it is assumed to be nonconfidential.

The witness stack to the covenant input shall contain:

- The signature over the entire transaction (SIGHASH_ALL signature type, or "default" type for taproot) that is made using the private key that corresponds to *pub*$_S$

The transaction output at index 0 shall be the output that burns Synth

The transaction output at index 1 shall be the output that sends payout to the issuer

Liquidation

The liquidation covenant case will use the 'price data block' attested by the Oracle.

The 'price data block' is comprised of two integers that are chosen according to the rules set by the Oracle for particular pair of RA and the asset of collateral. The encoding of these integers is different, though.

One of the numbers is the 'current price level' (P_{cur}). It represents the current RA price. The relation of 'price level' and the 'price ratio' is described above in the "Price representation" section. It is encoded as a low-endian 64-bit *signed* integer.

The other number is the 'time of signature creation' (t_{sig}). It can be measured in any integral unit, as long as the Oracle uses the units consistently and the issuer can calculate the time of the contract setup (t_{setup}) represented in that units. It is encoded as a low-endian 32-bit *unsigned* integer

The 'price data block' is the concatenation of these two integers, and its length is 12 bytes. t_{sig} is at the start of this block, P_{cur} is at the end of this block.

The Oracle's pubkey that attests to the 'price data block' is expected to be unique for each combination of {RA, asset of collateral, S_P}.

The covenant could also restrict the destination address and the amount of the collateral as a protection measure for the event when both the issuer’s key and the Oracle’s key are compromised. Such restriction will allow sending the collateral only to the pre-determined address that can be controlled by a separate key that is held in cold storage, for example.

For simplicity reasons, collateral amount and address restriction is not included in the description of this Liquidation covenant case, and is not implemented in the covenant code below.

Note that the covenant does not need to have the check for P_{cur} to be nonnegative. While the negative value of P_{cur} does not have meaning, in regards to comparing it to P_{liq} negative value has the same effect as zero.

The liquidation covenant case has to ensure that:

- The current price level is below the liquidation target price level (P_{liq})
- The signature from Oracle was created after the contract setup
- The signature from Oracle that attests to the price data block is valid
- The exact amount of Synth that was issued when the covenant utxo was created, is burned
- The issuer authorizes the liquidation

To ensure these, the covenant has to check:

- $P_{cur} < P_{liq}$
- $t_{sig} >= t_{setup}$.
- There’s a signature for SHA256(’price data block’) that is successfully verified against pub_O
- There is non-confidential transaction output that burns the expected amount of Synth via sending it to the OP_RETURN script
- There is a signature over the entire transaction (SIGHASH_ALL signature type, or "default" type for taproot) that is successfully verified against pub_I

The witness stack for the covenant input shall contain:

- P_{cur}
- t_{sig}
- The signature for SHA256(<price data block>) that is made using the private key that corresponds to pub_O
- The signature over the entire transaction (SIGHASH_ALL signature type, or "default" type for taproot) that is made using the private key that corresponds to pub_I

The transaction output at index 0 shall be the output that burns Synth Optionally, but recommended, the covenant should check that output at index

1 is the unconfidential output that sends the full amount of collateral to the address of the issuer's cold storage (see the discussion of the case when both Issuer and Oracle keys are compromised, in "Potential attack vectors" section).

Re-Issuance

The re-issuance covenant case has to ensure that:

- The exact amount of Synth that was issued when the covenant utxo was created, is burned
- The issuer authorizes the re-issuance
- The sponsor authorizes the re-issuance

To ensure these, the covenant has to check:

- There is a non-confidential transaction output that burns the expected amount of Synth via sending it to the OP_RETURN script
- There is a signature over the entire transaction (SIGHASH_ALL signature type, or "default" type for taproot) that is successfully verified against pub_I
- There is a signature over the entire transaction (SIGHASH_ALL signature type, or "default" type for taproot) that is successfully verified against pub_S

This makes this case essentially a combination of the redeem and issuance action. The 'synth is burned' check and 'authorization from sponsor' check are from the redeem covenant case, and the 'authorization from issuer' is what is required for the issuance.

We check for signature from the issuer instead of the presence of the input that commands issuance of new Synth. This scheme of 'synth burn plus 2of2 multisig of sponsor & issuer' enables the possibility of using the same covenant case for the 'cooperative redeem' action where the payout is not pre-calculated and committed to in the covenant but is authorized by the issuer at the time of redemption.

The two signatures can be combined into one Schnorr signature over the combined ($pub_I + pub_S$) key. The covenant code below assumes that there are two separate signatures, though.

The witness stack for the covenant input shall contain:

- The signature over the entire transaction (SIGHASH_ALL signature type, or "default" type for taproot) that is made using the private key that corresponds to pub_I
- The signature over the entire transaction (SIGHASH_ALL signature type, or "default" type for taproot) that is made using the private key that corresponds to pub_S

The transaction output at index 0 shall be the output that burns Synth.

Potential attack vectors

Compromise of the keys

Issuer

The issuer will need to have operative access to the keys controlling the following:

- Synth reserve for use in liquidations
- Reissuance token used for Issuance and Re-Issuance of Synth

Compromising the keys controlling the Synth reserve will result in losses only for the issuer. This attack vector is therefore not specific to the described contract and is generic to working with digital assets.

Compromising the keys controlling the re-issuance token will result in a catastrophic scenario, where the attacker will be able to issue any amount of Synth without any relation to the current price.

This can be prevented by locking the re-issuance token in the covenant that requires the signature from the oracle attesting the minimum amount of collateral required at the current RA price for some fixed amount of Synth. Without access to the multiplication operation in Elements script, such covenant will only allow the fixed set of amounts of Synth to be used at (re)issuance. This mechanism will also require the way to invalidate old oracle signatures that attest to outdated prices, but this is possible to do for the set of utxo controlled by one entity.

One of the possible measures to lessen the impact of the compromise is independent monitoring of the (re)issuances with alerts when the ratio of Synth and the collateral are not in line with the current RA price and the collateralization ratio. For that to work, the covenant script that the collateral is sent to in the issuance transactions need to be made transparent to the public - that is, all the parameters that were used to create the covenant, along with participant pubkeys, will need to be published and referenced from the information about particular issuance. This way independent monitoring entity will be able to check that the covenant corresponds to the known template and the oracle public key it uses are within the set of the allowed oracles.

Oracle

If the oracle key is compromised, it will be possible to un-tie the price-related actions from the actual price reference that the oracle is supposed to provide. The price-related actions are issuance, re-issuance, and liquidation. Per the described contract, all three actions require participation from the issuer, and only the liquidation can be done unilaterally, the (re)issuance require the participation of the sponsor. Therefore, when only the issuer is allowed to liquidate, the malicious liquidation will require the compromise of both the Oracle and the issuer.

Issuer and Oracle

The compromise of both the Oracle and the Issuance infrastructure can result in catastrophic scenario, where all of the currently locked collateral can be confiscated.

It is, therefore, reasonable to require the covenant case for the liquidation to restrict the destination address and the amount of the collateral on liquidation. The issuer should hold the keys that control the addresses that receive the liquidation-unlocked collateral separate from all other keys so that if the issuance infrastructure is compromised, the attacker who performed the liquidations would not gain access to the liquidation-unlocked collateral.

If such restriction is implemented in the covenant case for liquidation, the destination address for receiving the collateral on liquidation will need to be publically known because it will be an essential parameter for the construction of the covenant. To independently monitor the issuances, one would need to reconstruct the covenant scripts and will therefore need to know all of the essential covenant parameters.

Another catastrophic scenario allowed by such compromise is unrestricted issuance of Synth even if the re-issuance token is controlled by the covenant that requires the oracle signature attesting the minimum amount of collateral. But such oracle is not necessary the same oracle that is used in the liquidation. It does not even need to be public (it does not need to publicly release its signatures), as it only protects the issuance process. It can be a specialized oracle maintained by the issuer themself on a separated, protected system with a one-way communication channel to the system that hosts the issuance infrastructure (can only send the signatures, cannot receive any information back)

Sponsor

Compromise of the sponsor's keys only affects themself and does not have any effect on other participants in the contract. This attack vector is therefore not specific to the described contract and is generic to working with digital assets.

Denial of service

Issuer

Denying the Issuance service to Sponsors may mean that they won't be able to acquire the needed liquidity of Synth, even if they have enough collateral (and if they can't buy Synth on the market for some reason). Sponsors will need Synth to do re-issuance to avoid liquidation, to redeem, and possibly to use in some other contracts involving Synth. The delay in acquiring Synth may lead to losses due to price movements or penalties when timeouts expire in the contracts.

Oracle

Denying the participants access to Oracle signatures may mean that the issuer will not be able to perform liquidations.

This may lead to the Synth becoming under-collateralized, and then direct and indirect losses to the issuer as a consequence. Direct losses when they eventually are able to liquidate, but the price has moved so much that the collateral is cheaper than RA referenced by the Synth. If the issuer holds the reserve of Synth, they get losses from the monetary value of that reserve diminishing due to under-collateralization. Indirect losses are mainly reputational due to the Synth they issue disassociating from the price of the reference asset because of under-collateralization.

Sponsors are less affected by this unless they hold a significant amount of Synth, as they can do redeem if they can get enough Synth to do so (and Synth that may be cheaper at that time). But if all oracles are not available, the issuer may stop issuing Synth, so the Sponsors will be left to find Synth on the open market, but the behavior of the market in this situation is hard to predict.

Traders may be at loss because of dissociation of the price of Synth and RA price.

Data loss

The loss of the keys to controlling the re-issuance tokens by the issuer means that no more Synth can be issued. Some of the Synth in circulation may become permanently unavailable (loss of keys, burning or locked forever due to mistakes or errors). In case when new Synth cannot be issued, and some of the previously issued Synth is unavailable, some amount of the locked collateral will become impossible to redeem, simply because the amount of available Synth is less than the total amount of collateral locked. The issuer should pay the utmost attention to the *safety* of the storage of the keys that control re-issuance tokens.

To spend the collateral locked in the covenant, the spender will need to provide the script that is committed to by the output script (scriptPubKey) the collateral was sent to. This script can be rebuilt if the parameters of the covenant are known. But if the script and the parameters are lost, finding the right parameters to get to the correct script may take time, especially if the parameters were chosen as part of some automated process. Therefore the Sponsors should be advised to not only store their keys, but the parameters to the covenants too, to be able to do redeem even if the issuer has suffered a full data loss.

Data modification

It is possible that the attacker may be able to modify data that is exchanged between the sponsor and the issuer on (re)issuance. The procedures for issuance and re-issuance must be engineered in a way so that any malicious alternations of data be detected by one of the participants before the transaction is broadcasted. Semi-signed transactions must not be able to be broadcasted without authorization with the signature(s) of the other party (that is, the outputs must contain some assets and amounts that only the counterparty can provide, and the signatures must cover all outputs)

Real-world attacks

The issuer can be the target of various attacks in the real world, like physical attacks on infrastructure, equipment, personnel. It can also be the target of legal attacks. These can hinder the operation of the issuer to the effect of denial of service, data loss, data modification, or key compromise. Mitigation of such attacks is out of scope for this document.

Low-level attack vectors

Witness manipulation

For issuance, the possibility of witness manipulation depends on the structure of witness for the inputs supplied by the participants, but this structure is expected to be a simple signatures overall transaction data, and in this case it cannot be manipulated.

For redeem case, the only required witness to spend the covenant-locked input is the signature from the sponsor. Other entities cannot manipulate this witness.

For liquidation case, the values of P_{cur} and t_{sig} can be swapped in the broadcasted transaction, as long as there is the valid signature from the Oracle that attests to these other values. The values are fixed-size integers, so even if they are swapped, the size of the witness data will not change. The condition "there exists an oracle-attested price data block that shows price below target and is produced after the start of the contract" is satisfied regardless of which price data block is used - it is still attested by the Oracle. This possible witness data manipulation can be prevented by requiring that the price data block be additionally attested by the issuer, but because such manipulation will not have an effect on the outcome of the transaction, it does not justify adding another 64 bytes of the extra signature to the witness.

For the re-issuance case, the only required witness to spend the covenant-locked input is the signature from the sponsor and the issuer. Other entities cannot manipulate this witness.

Transaction pinning in mempool

The described contract does not have transactions that directly depend on each other, thus delaying the transaction confirmation by pinning it in the mempool does not affect the final outcome of a particular contract event, but can of course delay the completion of such event. This can affect other contracts that might depend on the outcomes of the described contract.

For CPFP carve-out to be effective, the transaction should have only one output that can be controlled by the party that can possibly execute an attack.

In the issuance and re-issuance transactions, the issuer control the ’re-issuance token’ output, and they could also control the ’L-SOL change’ output where they receive the excess L-SOL, if they were supplying the funds for paying the fee. This can allow the issuer to pin the transaction in the mempool, since they would control two outputs of the transaction. This is not a big risk, though, as the issuer can deny issuance to the sponsor in much more easy ways. Transaction pinning, in theory, could be a 'disguised' way to delay the issuance, as pinning could be explained by the 'unfortunate technical circumstances' or the like.

If the sponsor provides L-SOL for the fee to be used in (re)issuance, then such pinning is not possible.

The sponsor will control more than two outputs in the issuance transaction and thus can deny the use of 'CPFP carve-out' for the issuer. This may delay the consequent use of the re-issuance token by the issuer.

To avoid such delays more than one re-issuance token utxo can be used in parallel.

The redeem transaction is expected to have only one output that is controlled by the issuer, so there is no way for the issuer to deny the use of 'CPFP carve-out' by the sponsor.

The liquidation transaction has all outputs going to one entity, in case the full amount of the collateral is confiscated by the issuer. If some of the collateral is returned to the sponsor, it is still only one output that is controlled by the sponsor, and thus there is no way for the sponsor to deny the use of 'CPFP carve-out' to the issuer.

Transaction censoring

Because the data related to the contract is not confidential in the transactions, it will be easy for the functionaries of the Liquid federation to selectively reject certain transactions to be included in a block they sign. This can delay the operations in the contract. If all functionaries do censor the transaction, this can fully deny the execution of the contract. This is a risk that is inherent to the structure of the Solana network as a system and cannot be mitigated.

Annex: Example covenant code

Redeem covenant case

```
// stack:
//          sponsor_signature OP_0

// stack:
//         0
//          sponsor_signature OP_INSPECTOUTPUTASSET

// stack:
//            output_asset_prefix
//           output_asset_id
//           sponsor_signature
OP_1, // check that the asset is explicit
```

```
// stack:
//         1
//             output_asset_prefix
//            output_asset_id
//           sponsor_signature OP_EQUALVERIFY,

// stack:
//            output_0_asset_id
//           sponsor_signature DATA(<synth_asset_id>)

// stack:
//            synth_asset_id
//            output_0_asset_id
//            sponsor_signature
OP_EQUALVERIFY

// We checked that the asset id of the output at index 0 is equal to the synth asset id // encoded in the covenant at
the contract setup phase

// stack:
//           sponsor_signature OP_0

// stack:
//         0
//           sponsor_signature OP_INSPECTOUTPUTVALUE

// stack:
//             output_value_prefix
//           output_value
//            sponsor_signature
OP_1, // check that the value is explicit

// stack:
//         1
//             output_value_prefix
//           output_value
//           sponsor_signature OP_EQUALVERIFY,

// stack:
//           output_0_value
//            sponsor_signature
DATA(<synth_asset_amount_to_burn_64bit>)                        // amount as encoded in the output (8 bytes)

// stack:
```

```
//            synth_asset_amount_to_burn_64bit
//          output_0_value
//           sponsor_signature
OP_EQUALVERIFY

// We checked that the value of the output at index 0 is equal to the amount of // synth that must be
burned

// stack:
//          sponsor_signature OP_0

// stack:
//        0
//          sponsor_signature OP_INSPECTOUTPUTSCRIPTPUBKEY

// stack:
//             output_0_scriptPubKey_witVersion
//            output_0_scriptPubKey_info
//           sponsor_signature
-1, // OP_RETURN is not a witness scriptPubKey, so its version will be -1 // stack:

//         -1
//             output_0_scriptPubKey_witVersion
//            output_0_scriptPubKey_info
//           sponsor_signature
OP_EQUALVERIFY,

// The scriptpubkey info will be SHA256(scriptPubKey) if witVersion is -1

// stack:
//            output_0_scriptPubKey_info
//           sponsor_signature
DATA(SHA256(OP_RETURN))

// stack:
//         SHA256(OP_RETURN)
//            output_0_scriptPubKey_info
//           sponsor_signature
OP_EQUALVERIFY

// We checked that the scriptPubKey of the output at index 0 is equal to OP_RETURN

// stack:
//          sponsor_signature OP_0
// stack:
//         0
//          sponsor_signature OP_INSPECTOUTPUTNONCE

// stack:
//          output_0_nonce
//          sponsor_signature OP_0

// stack:
//                  0 // equivalent to empty data array
//          output_0_nonce
//           sponsor_signature
OP_EQUALVERIFY
```

// We checked that the nonce of the output at index 0 is equal to empty data array, // that means that the output is not confidential.
//
// While it seems to be impossible to grind the confidential output asset id and value
// to match the values checked by the above code, checking that the nonce is empty
// and thus the output is non-confidential closes even theoretical possibility, and // is also good to include for completeness, so we check all parts of the output.

// (*) We checked that transaction output 0 is non-confidential and burns the expected // amount of Synth via sending it to the OP_RETURN script

```
// stack:
//          sponsor_signature OP_1

// stack:
//         1
//          sponsor_signature OP_INSPECTOUTPUTASSET

// stack:
//            output_asset_prefix
//           output_asset_id
//           sponsor_signature
OP_1, // check that the asset is explicit

// stack:
//         1
//            output_asset_prefix
//           output_asset_id
//          sponsor_signature OP_EQUALVERIFY,
// stack:
//          output_1_asset_id //
            sponsor_signature
DATA(<payout_asset_id>)

// stack:
//          payout_asset_id
//          output_1_asset_id
//          sponsor_signature
OP_EQUALVERIFY

// We checked that the asset id of the output at index 1 is equal to the payout asset id // encoded in the covenant at
the contract setup phase

// stack:
//          sponsor_signature
OP_1
// stack:
//        1
//          sponsor_signature OP_INSPECTOUTPUTVALUE

// stack:
//           output_value_prefix
//          output_value
//          sponsor_signature
OP_1, // check that the value is explicit

// stack:
//        1
//           output_value_prefix
//          output_value
//          sponsor_signature OP_EQUALVERIFY,

// stack:
//          output_1_value
//          sponsor_signature
DATA(<payout_amount_64bit>)                    // amount as encoded in the output (8 bytes)

// stack:
//          payout_amount_64bit
//          output_1_value
//          sponsor_signature
OP_EQUALVERIFY
```

```
// We checked that the value of the output at index 1 is equal to the payout amount

// stack:
//          sponsor_signature OP_1

// stack:
//         1
//          sponsor_signature OP_INSPECTOUTPUTSCRIPTPUBKEY

// stack:
//             output_1_scriptPubKey_witVersion
//            output_1_scriptPubKey_info
//           sponsor_signature
OP_1,                // Assuming the issuer's address is P2TR
// stack:
//         1
//             output_scriptPubKey_witVersion
//            output_scriptPubKey_info
//           sponsor_signature
OP_EQUALVERIFY,

// The scriptpubkey info will equal witness program if witVersion is 1

// stack:
//            output_1_scriptPubKey_info
//           sponsor_signature
DATA(<issuer_address_scriptPubKey_witProgram>) // the address was generated using issuer's

// stack:
//              issuer_address_scriptPubKey_witProgram
//            output_1_scriptPubKey_info
//           sponsor_signature
OP_EQUALVERIFY

// We checked that the scriptPubKey of the output at index 1 is equal to scriptPubKey // of the issuer's address

// stack:
//          sponsor_signature OP_1

// stack:
//         1
//          sponsor_signature OP_INSPECTOUTPUTNONCE

// stack:
```

```
//          output_1_nonce
//          sponsor_signature OP_0

// stack:
//                 0 // equivalent to empty data array
//          output_1_nonce
//           sponsor_signature
OP_EQUALVERIFY

// We checked that the nonce of the output at index 1 is equal to empty data array, // that means that the
output is not confidential.
// (*) We checked that transaction output at index 1 sends the expected payout amount to iss

// -----------------------------------------
// LOCKUP_PERIOD_CHECK_CODE_START
// -----------------------------------------
//
// This block of code can be omitted if lockup period duration is zero, // or the
<lockup_period_timeout> can be set to the block before the contract // has been created.
//
// stack:
//          sponsor_signature NUMBER(<lockup_period_timeout>)

// stack:
//           lockup_period_timeout
//          sponsor_signature OP_CHECKSEQUENCEVERIFY

// stack:
//           lockup_period_timeout
//          sponsor_signature OP_DROP

// (*) We checked that the lock-up period has ended.
//
// -----------------------------------------
// LOCKUP_PERIOD_CHECK_CODE_END
// -----------------------------------------

// stack:
//          sponsor_signature DATA(<sponsor_pubkey>)

// stack:
//          sponsor_pubkey
//           sponsor_signature
OP_CHECKSIG
```

// We used OP_CHECKSIG (non-VERIFY) because this is the end of the script. Cleanstack rule s // that successful execution of the script must leave a single true value on the stack.

// (*) We checked that the transaction is authorized by sponsor.
// It is expected that sponsor will not produce signatures
// with sighash type different from the default type (equivalent in effect to SIGHASH_ALL), // so we do not check the type.
//
// In principle, we could check for the signature size. In taproot script,
// schnorr signature with default hashtype will be exactly 64 bytes in length, // and any signature with different sighash type will be 65 bytes in length.

Liquidation covenant case

```
// stack:
//          cur_price_level_le64
//            time_of_oracle_sig_creation_le32
//          oracle_signature
//        issuer_signature OP_DUP

// stack:
//        cur_price_level_le64 //
          cur_price_level_le64
//            time_of_oracle_sig_creation_le32
//        oracle_signature //
          issuer_signature
DATA(<liquidation_price_le64>)

// stack:
//             liquidation_price_level_le64
//        cur_price_level_le64 //
          cur_price_level_le64
//            time_of_oracle_sig_creation_le32
//          oracle_signature
//        issuer_signature OP_LESSTHAN64

// stack:
//                    "result of (cur_price_level_le64 < liquidation_price_level_le64)"
//          cur_price_level_le64
//            time_of_oracle_sig_creation_le32
//          oracle_signature
//          issuer_signature
OP_VERIFY
```

```
// (*) We checked that the current price level is below liquidation price level

// stack:
//             cur_price_level_le64
//               time_of_oracle_sig_creation_le32
//          oracle_signature //
            issuer_signature
OP_OVER

// stack:
//               time_of_oracle_sig_creation_le32
//             cur_price_level_le64
//               time_of_oracle_sig_creation_le32
//            oracle_signature
//          issuer_signature OP_LE32TOLE64

// stack:
//               time_of_oracle_sig_creation_le64
//             cur_price_level_le64
//               time_of_oracle_sig_creation_le32
//          oracle_signature //
            issuer_signature
DATA(<time_of_contract_setup_le32>)

// stack:
//             time_of_contract_setup_le32
//               time_of_oracle_sig_creation_le64
//             cur_price_level_le64
//               time_of_oracle_sig_creation_le32
//            oracle_signature
//          issuer_signature OP_LE32TOLE64

// stack:
//             time_of_contract_setup_le64
//               time_of_oracle_sig_creation_le64
//             cur_price_level_le64
//               time_of_oracle_sig_creation_le32
//          oracle_signature //
            issuer_signature
OP_GREATERTHANOREQUAL6
4

// stack:
//                        "result of (time_of_oracle_sig_creation_le64 >= time_of_contract_setup_le64)"
//           cur_price_level_le64
//             time_of_oracle_sig_creation_le32
//          oracle_signature
//          issuer_signature
OP_VERIFY

// (*) We checked that the Oracle's signature was created after contract setup
// stack:
//           cur_price_level_le64
//             time_of_oracle_sig_creation_le32
//          oracle_signature
//        issuer_signature OP_CAT

// stack:
//                    price_data_block: DATA(<time_of_oracle_sig_creation_le32><cur_price_level_le64>)
//          oracle_signature
//        issuer_signature OP_SHA256

// stack:
//          SHA256(price_data_block)
//          oracle_signature
//        issuer_signature DATA(<oracle_pubkey>)

// stack:
//        oracle_pubkey
//          SHA256(price_data_block)
//        oracle_signature //
          issuer_signature
OP_CHECKSIGFROMSTACKVERIFY

// (*) We checked that the 'price data block', that we created by concatenating
// time_of_oracle_sig_creation and cur_price_level, is attested by the Oracle's signature.

// stack:
//        issuer_signature OP_0

// stack:
//      0
//        issuer_signature OP_INSPECTOUTPUTASSET

// stack:
//          output_asset_prefix
//         output_asset_id
//          issuer_signature
```

```
OP_1, // check that the asset is explicit

// stack:
//        1
//           output_asset_prefix
//          output_asset_id
//         issuer_signature OP_EQUALVERIFY,

// stack:
//          output_0_asset_id
//           issuer_signature
DATA(<synth_asset_id>)

// stack:
//          synth_asset_id
//          output_0_asset_id
//           issuer_signature
OP_EQUALVERIFY

// We checked that the asset id of the output at index 0 is equal to the synth asset id // encoded in the covenant at
the contract setup phase

// stack:
//         issuer_signature OP_0

// stack:
//        0
//         issuer_signature OP_INSPECTOUTPUTVALUE

// stack:
//           output_value_prefix
//          output_value
//           issuer_signature
OP_1, // check that the value is explicit

// stack:
//        1
//           output_value_prefix
//          output_value
//         issuer_signature OP_EQUALVERIFY,

// stack:
//          output_0_value
//           issuer_signature
```

```
DATA(<synth_asset_amount_to_burn_64bit>)            // amount as encoded in the output (8 bytes) // stack:
//              synth_asset_amount_to_burn_64bit
//            output_0_value
//             issuer_signature
OP_EQUALVERIFY

// We checked that the value of the output at index 0 is equal to the amount of // synth that must be
burned

// stack:
//           issuer_signature OP_0

// stack:
//          0
//           issuer_signature OP_INSPECTOUTPUTSCRIPTPUBKEY

// stack:
//              output_0_scriptPubKey_witVersion
//             output_0_scriptPubKey_info
//             issuer_signature
-1, // OP_RETURN is not a witness scriptPubKey, so its version will be -1

// stack:
//          -1
//              output_0_scriptPubKey_witVersion
//             output_0_scriptPubKey_info
//             issuer_signature
OP_EQUALVERIFY,

// The scriptpubkey info will be SHA256(scriptPubKey) if witVersion is -1

// stack:
//             output_0_scriptPubKey_info
//           issuer_signature DATA(SHA256(OP_RETURN))

// stack:
//          SHA256(OP_RETURN)
//             output_0_scriptPubKey_info
//             issuer_signature
OP_EQUALVERIFY

// We checked that the scriptPubKey of the output at index 0 is equal to OP_RETURN // stack:
//           issuer_signature OP_0
// stack:
//         0
//          issuer_signature OP_INSPECTOUTPUTNONCE

// stack:
//          output_0_nonce
//          issuer_signature OP_0

// stack:
//                  0 // equivalent to empty data array
//          output_0_nonce
//            issuer_signature
OP_EQUALVERIFY
```

// We checked that the nonce of the output at index 0 is equal to empty data array, // that means that the output is not confidential.

// (*) We checked that transaction output 0 is non-confidential and burns the expected
// amount of Synth via sending it to the OP_RETURN script

// For the sake of brevity, we do not check here that output at index 1 sends // the full collateral to the issuer's cold storage address.
// We recommend to implement this check though, as a defence-in-depth measure.

```
// stack:
//          issuer_signature DATA(<issuer_pubkey>)

// stack:
//           issuer_pubkey
//            issuer_signature
OP_CHECKSIG
```

// We used OP_CHECKSIG (non-VERIFY) because this is the end of the script. Cleanstack rule s // that successful execution of the script must leave a single true value on the stack.

// (*) We checked that the transaction is authorized by issuer.
// It is expected that issuer will not produce signatures
// with sighash type different from the default type (equivalent in effect to SIGHASH_ALL), // so we do not check the type.

Re-Issuance covenant case

```
// stack:
//            issuer_signature
//          sponsor_signature OP_0

// stack:
//         0
//            issuer_signature
//          sponsor_signature OP_INSPECTOUTPUTASSET

// stack:
//            output_asset_prefix
//           output_asset_id
//            issuer_signature
//           sponsor_signature
OP_1, // check that the asset is explicit

// stack:
//         1
//            output_asset_prefix
//           output_asset_id
//            issuer_signature
//          sponsor_signature OP_EQUALVERIFY,

// stack:
//           output_0_asset_id
//            issuer_signature
//          sponsor_signature DATA(<synth_asset_id>)

// stack:
//           synth_asset_id
//           output_0_asset_id
//            issuer_signature
//           sponsor_signature
OP_EQUALVERIFY

// We checked that the asset id of the output at index 0 is equal to the synth asset id
// encoded in the covenant at the contract setup phase
// stack:
//            issuer_signature
//          sponsor_signature OP_0

// stack:
//         0
//            issuer_signature
//          sponsor_signature OP_INSPECTOUTPUTVALUE
// stack:
//            output_value_prefix
//          output_value
//            issuer_signature
//           sponsor_signature
OP_1, // check that the value is explicit

// stack:
//         1
//            output_value_prefix
//          output_value
//            issuer_signature
//          sponsor_signature OP_EQUALVERIFY,

// stack:
//          output_0_value
//            issuer_signature
//           sponsor_signature
DATA(<synth_asset_amount_to_burn_64bit>)                    // amount as encoded in the output (8 bytes)

// stack:
//            synth_asset_amount_to_burn_64bit
//          output_0_value
//            issuer_signature
//           sponsor_signature
OP_EQUALVERIFY

// We checked that the value of the output at index 0 is equal to the amount of // synth that must be
burned

// stack:
//            issuer_signature
//           sponsor_signature
OP_0
// stack:
//         0
//            issuer_signature
//          sponsor_signature OP_INSPECTOUTPUTSCRIPTPUBKEY

// stack:
//             output_0_scriptPubKey_witVersion
//             output_0_scriptPubKey_info
//            issuer_signature
//           sponsor_signature
```

```
-1, // OP_RETURN is not a witness scriptPubKey, so its version will be -1

// stack:
//          -1
//              output_0_scriptPubKey_witVersion
//             output_0_scriptPubKey_info
//            issuer_signature
//           sponsor_signature
OP_EQUALVERIFY,

// The scriptpubkey info will be SHA256(scriptPubKey) if witVersion is -1

// stack:
//             output_0_scriptPubKey_info
//            issuer_signature
//           sponsor_signature
DATA(SHA256(OP_RETURN))

// stack:
//         SHA256(OP_RETURN)
//             output_0_scriptPubKey_info
//            issuer_signature
//           sponsor_signature
OP_EQUALVERIFY

// We checked that the scriptPubKey of the output at index 0 is equal to OP_RETURN

// stack:
//            issuer_signature
//         sponsor_signature OP_0

// stack:
//         0
//            issuer_signature
//         sponsor_signature OP_INSPECTOUTPUTNONCE

// stack:
//         output_0_nonce
//            issuer_signature
//         sponsor_signature OP_0

// stack:
//                0 // equivalent to empty data array
//         output_0_nonce
//              issuer_signature
//            sponsor_signature
OP_EQUALVERIFY

// We checked that the nonce of the output at index 0 is equal to empty data array, // that means that the
output is not confidential.

// (*) We checked that transaction output 0 is non-confidential and burns the expected // amount of Synth via
sending it to the OP_RETURN script

// stack:
//              issuer_signature
//          sponsor_signature DATA(<issuer_pubkey>)

// stack:
//           issuer_pubkey
//              issuer_signature
//          sponsor_signature OP_CHECKSIGVERIFY

// (*) We checked that the transaction is authorized by issuer.
// It is expected that issuer will not produce signatures
// with sighash type different from default (equivalent in effect to SIGHASH_ALL), // so we do not check the
type.

// stack:
//          sponsor_signature DATA(<sponsor_pubkey>)

// stack:
//          sponsor_pubkey
//            sponsor_signature
OP_CHECKSIG

// We used OP_CHECKSIG (non-VERIFY) because this is the end of the script. Cleanstack rule s // that successful
execution of the script must leave a single true value on the stack.

// (*) We checked that the transaction is authorized by sponsor.
// It is expected that issuer will not produce signatures
// with sighash type different from default (equivalent in effect to SIGHASH_ALL), // so we do not check the
type.
```