# tBTC

## *A Decentralized Redeemable BTC-backed ERC-20 Token*

2019-08-15

# Table of Contents

**Abstract**

# Overview

# Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC 2119](https://tools.ietf.org/html/rfc2119).

# A note on naming

The system, in its entirety, is called "tBTC". In this document and throughout the project, the fungible Bitcoin-backed token is called "TBTC" to distinguish it from the rest of the project, and strongly suggest an eventual ticker.

Further discussion can be found on Github.

# Prior Work

Prior efforts toward a cross-chain Bitcoin peg are well-known. A Bitcoin peg is desirable for sidechains- functionality and scalability extensions to the conservatively upgraded main Bitcoin blockchain. Due to early interest in sidechains, a number of pegged Bitcoin approaches predate Ethereum.

## Centralized, Provable, Redeemable

Two solutions in the wild today provide centralized pegs that rely on trusted third parties based on variants of the "federated peg" model.

In a federated peg, a multi-sig wallet is used to lock up bitcoins. Another blockchain then issues tokens representing those bitcoin. The signers of the multisig on the Bitcoin chain are expected to validate the sidechain, and only allow issued tokes to be burned in exchange for bitcoin withdrawals following the rules of the sidechain.

Liquid, a sidechain developed by Blockstream, is an inter-exchange settlement network based on a federated peg sidechain. Bitcoin is locked in a 15-signer multi-sig wallet comprised of exchanges and Liquid participants pre-vetted by Blockstream. These signers validate the sidechain in an approach the team calls "strong federation", where a majority vote to sign blocks, and agree to approve exits to the main chain.

WBTC is a Bitcoin-backed ERC-20 token using a similar approach. The token is part of a greater effort called "Wrapped Tokens".

> Wrapped tokens follow the centralized model, but instead of relying entirely on one institution, they rely on a consortium of institutions performing different roles in the network.
>
> — the Wrapped Tokens whitepaper

The WBTC consortium votes on adding and removing custodians that manage the token’s Bitcoin reserves. Each custodian operates a multi-sig Bitcoin wallet, with control of all keys. Custodians are able to move custodied bitcoin at will, and are responsible for minting WBTC on Ethereum.

Together, the custodians act similarly to a traditional federated peg. Instead of requiring a majority to sign Bitcoin withdrawals, however, a single member can withdraw their share of the Bitcoin reserves at any time.

### Trade-offs

These approaches have a few clear benefits

- They each effectively peg Bitcoin on other blockchains.
- Backing reserves are easily audited on-chain at any time.
- Both are simple mechanisms, lowering the chance of operational failure as well as the total cost to operate.

However, there are downsides. The most obvious is introducing a trust model incompatible with Bitcoin.

Custodians need to be fully trusted, either as a group, as in Liquid’s model, or individually, following the Wrapped Tokens model. A malicious custodian can block withdrawals and in some cases collude to abscond with funds. Custodians may also be compelled by governments, hackers, or other forces to tamper with reserves, despite their good intentions.

## Decentralized, Synthetic, Irredeemable

An alternative approach to a centralized peg is to create a decentralized synthetic asset.

One approach that’s been popular on Ethereum is Maker’s Dai stablecoin.

Dai is a token synthetically pegged to the US dollar. Ether is locked up in reserves, which, coupled with a robust price feed and a number of stability mechanisms, allow for maintenance of the peg under adverse conditions.

While Maker hasn’t launched a Bitcoin synthetic, the same network maintaining Dai’s peg could easily be applied to maintain a similar Bitcoin product.

### Trade-offs

The biggest benefit of a synthetic Bitcoin peg is its flexibility. A synthetic doesn’t need to follow the rules governing the pegged asset, for better or worse.

For example, a synthetic might effectively "inflate" the supply of the underlying asset, which might be desirable for some financial systems- and directly fly in the face of the purpose of a currency aspiring to be hard money.

Despite the potential reuse of Maker’s network, launching such a synthetic has other risks. A synthetic peg to a volatile asset like Bitcoin, backed by a volatile, under-diversified reserve entirely of Ether, is a dangerous combination.

# Design Goals

The goal of tBTC is the creation an ERC-20 token that maintains the most important property of Bitcoin- its status as "hard money".

To maintain the "hard money" status of its backing BTC deposits, tBTC must remain

- Censorship and seizure resistant, across friendly and unfriendly jurisdictions
- Inflation-resistant. TBTC may only be minted after proof is provided of a backing BTC deposit.
- Leverage-resistant. The existence of tBTC shouldn’t allow cross-chain "printing" of additional synthetic Bitcoin. We can’t stop someone from launching a synthetic, but artificially expanding the Bitcoin supply is not a goal of the project.
- Without middlemen, in the same sense as Bitcoin. The only rent extraction should be from the minimal participation of signers required to secure the network, similar to miners on the Bitcoin network.
- Redeemable. The ability to trade scrip for its backing deposit freely is what distinguishes a backed currency from fiat money. The supply of tBTC is always backed by an equal number of reserved BTC. This means for every token in circulation, 1 BTC has been removed from circulation.

Together, these properties ensure a strong supply peg across chains, and the closest to "hard money" status that a Bitcoin-pegged asset can achieve.

Notably, these properties don’t require an artificial price peg as is common in stable coin projects — they instead require a supply peg across chains.

## Developing Intuition: A simple single-signer protocol

To understand how we might develop a protocol and token that satisfies those requirements, it's useful to consider a simple, under-specified variant that could theoretically do the job.

Imagine an off-chain actor, which we'll call `Signer`, an Ethereum smart contract that implements the ERC-20 interface, `PeggedBitcoin`, with ticker `PBTC`, and another contract with the permission to mint and burn `PBTC` called `PeggedBitcoinReserve`.

Another off-chain actor, `Depositor`, wants to mint a token on the `PeggedBitcoin` contract. `Depositor` requests the `PeggedBitcoinReserve` accept a 1 BTC deposit. `PeggedBitcoinReserve` waits for `Signer` to acknowledge and return a new BTC address, as well as depositing 150% collateral of the deposit's value in ETH into the `PeggedBitcoinReserve`. `Depositor` deposits 1 BTC into the new BTC address, and provides proof to `PeggedBitcoinReserve` - which in turn mints 1 `PBTC`, sending 0.99 to `Depositor` and .01 to `Signer` for the convenience.

Withdrawals happen in reverse- any participant can send 1 `PBTC` to `PeggedBitcoinReserve` with a Bitcoin address. `Signer` pays that Bitcoin address 1 BTC minus any transaction fees, and provides proof of payment to `PeggedBitcoinReserve`, which burns the remaining 1 `PBTC`, maintaining a 1:1 backing of `PBTC`. `Signer` is now free to withdraw the corresponding collateral from `PeggedBitcoinReserve`.

## Flaws

While this simple design is attractive, it's skipped over some of the more difficult issues- efficient Bitcoin proof of payment validation on the EVM and a reliable price feed implementation, for example.

It's also based on a deeply insecure custody solution.

First, the protocol relies on a single signer. If the value of deposits ever exceeds the value of the collateral `Signer` has put down, there's nothing stopping `Signer` from walking with the BTC. `Signer` can also decide or be coerced to censor particular withdrawals, removing any hope of censorship or seizure resistance.

Second, the protocol relies on a single **hot wallet**. As the market cap of `PBTC` conceivably grows, the risk due to hacking that wallet increases tremendously.

Finally, the protocol does nothing to localize failure. If there's an issue with a single deposit or withdrawal, it could impact the entire `PeggedBitcoin` supply, blocking all further deposits and withdrawals.

# System Architecture: Designing a robust multi-wallet multi-signer protocol

The rest of this document is devoted to specifying a protocol that addresses those flaws, providing a robust BTC-backed bearer asset on Ethereum.

At a high level, that means the protocol described must

- have a multi-wallet architecture
- with many geographically distributed signers
- that removes single points of failure

This protocol must also counter the secondary effects of these requirements and the details we skipped in the single signer example, including multi-signer payment, a more complex bonding system, an approach for detecting and dealing with undercollateralized signers, a Bitcoin proof system, and robust handling of failures on both chains.

Some components necessary to this protocol are described outside this document and will be assumed. In particular, we assume the existence of

- a well-distributed work token for signer selection
- a random beacon for signer selection
- an efficient distributed key generation protocol on the secp256k1 curve
- an efficient multi-party threshold ECDSA protocol on the secp256k1 curve

all of which are implemented by the Keep network.

The architecture is broken down into

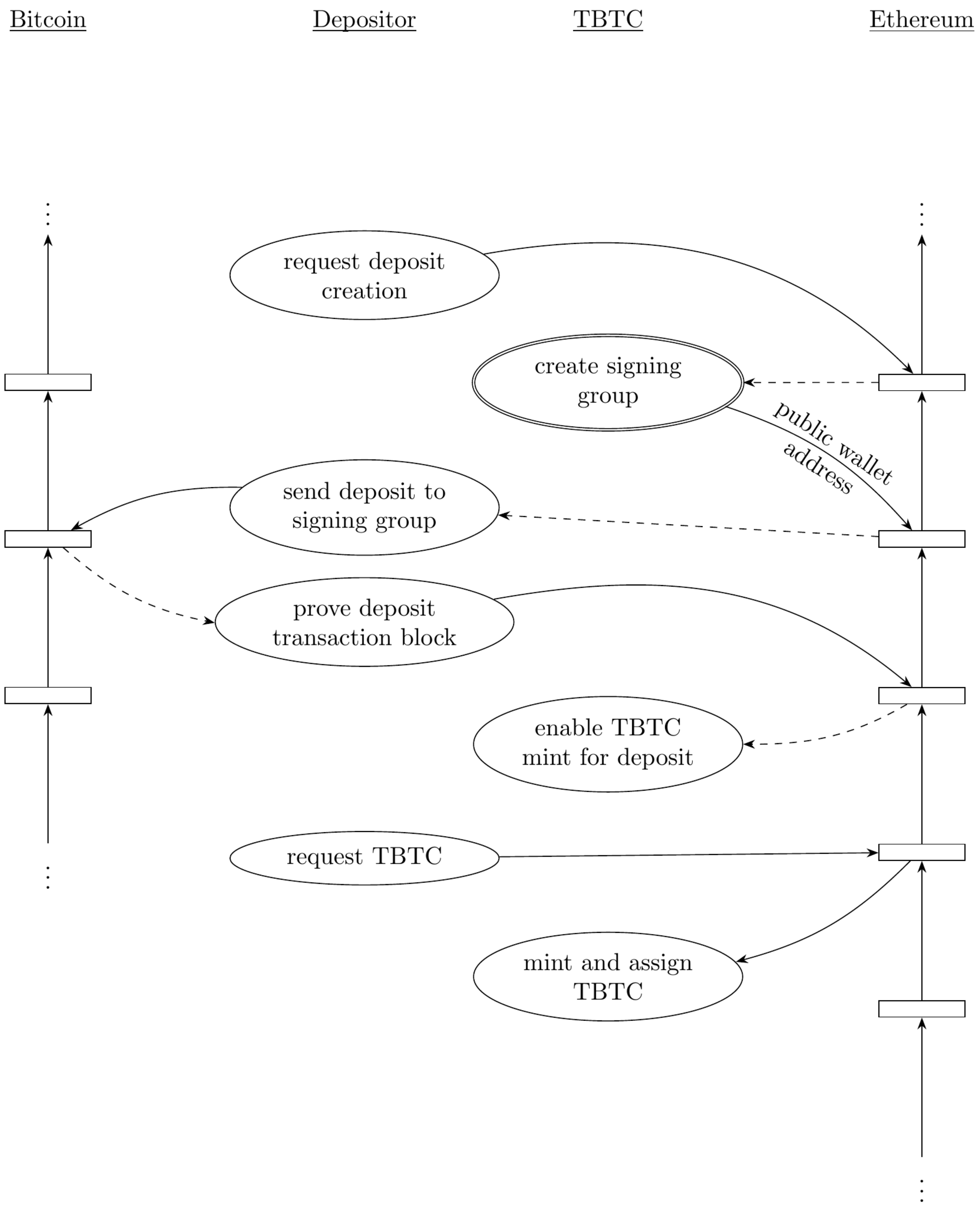
- Deposits and signer selection
- Bonding and price feeds
- Custodial fees
- Signing
- Wallet failure
- Redemption

# Deposits

## Overview

tBTC provides a mechanism for creating a token, TBTC, on a non-Bitcoin *host chain*, that is 1-to-1 backed by bitcoin. Parties interested in minting TBTC request that the tBTC system provide them with a Bitcoin wallet address. The system selects a set of *signers*, which are tasked with generating a private/public keypair and furnishing it to the system. The interested party then becomes a *depositor* by sending bitcoin to the wallet (the amount of required bitcoin is discussed separately in the section on lots). The bitcoin that is sent to the wallet is in two parts: one is eligible for 1-to-1 minting into TBTC, while the other is reserved as incentive and collateral for the wallet signers.

Each of these steps is shown in the diagram below and discussed in subsequent sections.

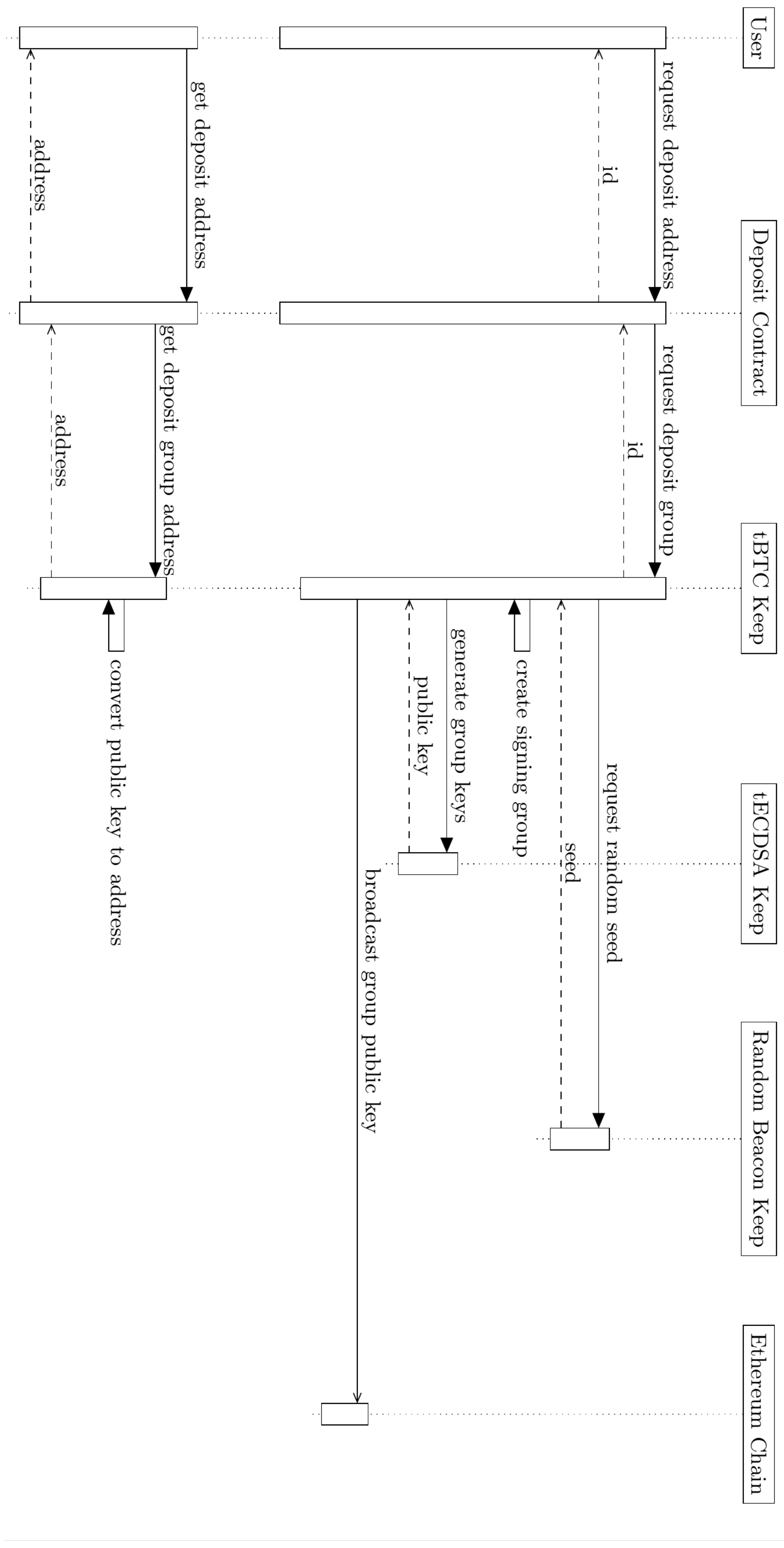


## Deposit request

The starting point for acquiring TBTC is generating a *deposit request*. This is a transaction to a smart contract on tBTC’s host chain that informs the tBTC system that the sender account is interested in creating a TBTC deposit. The transaction is a signal that the sender is interested in a signing group to back a wallet that will receive bitcoin from a wallet the sender controls, in order to produce TBTC. Because signing groups are not free to create, deposit requests include a small bond in the host chain’s native token to cover the creation of the signing group. The bond is refunded when a successful deposit is made to the generated wallet.

### Signer selection

Once the deposit request is received, the signing group is created by randomly selecting a set of *signers* to back a Bitcoin wallet. This is a multi-part process described in the diagram below.[1]


User
Deposit Contract
tBTC Keep
tECDSA Keep
Random Beacon Keep
Ethereum Chain
request deposit address
id
request deposit group
id
request random seed
seed
create signing group
generate group keys
public key
broadcast group public key
get deposit address
address
get deposit group address
address
convert public key to address

When a request comes in to create a signing group, the tBTC system requests a random seed from a secure decentralized random beacon.[2] The resulting random seed is used to randomly select signing group members from the eligible pool of signers. Finally, these signers coordinate a distributed key generation protocol that results in a public ECDSA key for the group, which is used to produce a wallet address that is then published to the host chain. This completes the signer selection phase.

#### Bonding

Before the selected members of a signing group can perform distributed key generation, they must agree to become members of the signing group by putting up a bond in the native token of the host chain. This bond is used to penalize the members of the signing group if an unauthorized piece of data is signed by the signing group once distributed key generation is complete; it is also used to penalize a given member if the distributed key generation fails due to an attributed misbehavior of that member.

Bonding is described in more detail in its own section.

#### Distributed key generation

Some small notes about the distributed key generation a signing group undergoes. The distributed key generation protocol should result in three properties:

1. The signing group as a whole should have an *ECDSA public key*, which will be shared on the host chain and will correspond to the Bitcoin wallet owned by that signing group.
2. Each member of the signing group should have a *threshold ECDSA secret key share*, which can be used to create a *threshold ECDSA signature share* for any transactions involving the signing group's wallet.
3. Each member of the signing group should be able to combine a threshold number of signature shares from itself and other members of the group to produce a signed version of a given transaction to be performed on behalf of the signing group.

### Proof of deposit

Once the tBTC system has a wallet address available for a given deposit request, the *depositor* can issue a Bitcoin transaction sending BTC from a wallet they control to the wallet address for the signing group. Once the transaction has been sufficiently confirmed by the Bitcoin chain, the depositor has to issue a transaction to the host chain proving that the *Deposit* has been funded.

The only link between the Bitcoin chain and the host chain is the tBTC system, which runs as a set of smart contracts on the host chain. As such, the Bitcoin transaction issued by the depositor has to be proven to the tBTC system before the tBTC system allows the depositor to behave as if they have successfully deposited their BTC into the custodial wallet. When a deposit proof is accepted by the system, the deposit bond is refunded to the depositor. If a deposit proof is not received within a given timeout window, the signing group will disband and the system will seize the bond's value, making it available to the signing group members to reclaim.

To prove deposit, the depositor constructs a transaction for the host chain that provides proof that the transaction was accepted on the Bitcoin chain and has had sufficient work built on top of the block that included the deposit transaction. These proofs are verified by an on-chain simple payment verification (SPV) system. A more complete discussion of cross-chain SPV systems and their security properties is included in the appendix.

#### Overpayment & Underpayment

The system is designed to function with a predefined lot size for all *Deposits* which is given as a system parameter. **Depositors should send the exact lot amount of BTC in the funding transaction, or expect loss of funds.** Since it is not possible for the system to force users into sending any specific amount, the system must gracefully handle overpayment and underpayment. The primary impact of overpayment and underpayment is on the `Deposit's collateralization ratio. We treat overpayment and underpayment as faulty depositor behavior, and pass on the associated costs to the depositor.

### Underpayment

Allowing underpayment on `Deposit` would result in over-bonded signers. To prevent this, the system will not accept funding proofs of less than the lot size (`1.0 BTC` initially). This implies that the a user that sends less than `1.0 BTC` in the funding transaction does not receive any TBTC, and forfeits the BTC locked in the funding transactions to the Signers. The Signers can unlock and evenly split the funds in the transaction after the *Deposit* is resolved on-chain (see Multiple UTXOs for a full discussion).

### Overpayment

Overpayment, in contrast, would result in under-bonded signers. When overfunding occurs, the system accepts the funding proof, but mints TBTC according to the regular lot size.

In an efficient market, this *Deposit* will be immediately redeemed, as the redeemer expects to take the overfunded amount as arbitrage.

Example: A user providing a funding proof for `1.6 BTC` in a system with lot size of `1 BTC` mints only `1.0 TBTC`. Any user that burns `1.0 TBTC` is able to claim the `1.6 TBTC` *Deposit*.

A depositor should notice this and immediately try to burn their TBTC to reclaim the amount. If not, the depositor effectively forfeits the overfunded value to other participants.

#### Multiple UTXOs

A faulty depositor may send multiple UTXOs to the signer group address. This may be the result of human or software error. Unfortunately, returning the funds to the depositor would impose require significant on-chain complexity and gas fees, as each UTXO would need to be proven via SPV, and a signature on it explicitly authorized. In addition, we would have to develop mechanisms to economically compel signers to sign each transaction despite the fact that the total value of the UTXOs is not known. As such, the system accepts only the first UTXO greater than the deposit lot size. All other BTC sent to the signing group is forfeit. Therefore it is imperative that depositors send only a single UTXO of an appropriate size.

As a particularly damaging example, consider a naive human depositor. If she mistakenly sends half the lot size in one transaction and half in another, both UTXOs would be forfeit. **This**

**represents a serious pitfall for depositors that must be carefully addressed by the user interface since significant loss of funds can occur.**

The signers, while they may collude to move the BTC to other UTXOs, may not do so during the life of the *Deposit* contract as the production of the required signature would constitute ECDSA fraud. As such, the most likely outcome is that the signers collectively wait to take personal possession of that BTC until the *Deposit* concludes naturally. This allows the signers to make regular signing fees and keep the additional UTXOs without penalty.

#### Light Relays

Light relays are a new variant of on-chain SPV developed for tBTC. They seek to take advantage of the compact and efficient stateless SPV proofs while relaying enough information to provide each stateless proof with some recency guarantee. We achieve this by taking advantage of the difficulty adjustment feature of Bitcoin's protocol. Bitcoin adjusts difficulty every 2016 blocks, based on timestamps of the first and last block in that period (due to an off-by-one error in the Satoshi client, one interblock period is exlcuded from the difficulty calculation). The change is deterministic and within some tolerance may be set by the miner of the last block.

A light relay does not store every block header. Instead it stores only a slice of headers around the difficulty adjustment event and records the difficulty for the current 2016-block epoch. This slice is validated by its objective proof of work, as well as verifying that its first headers' difficulty matches the current epoch difficulty, that the change occurs at an expected index in the slice, and that the new difficulty conforms to Bitcoin's adjustment algorithm. In other words, our light relay tracks only Bitcoin's current difficulty, and no other information about its state.

Knowing the current difficulty gives stateless SPV proofs stronger recency assurances. Any newly-generated stateless SPV must include that difficulty in its header chain. And that difficulty is not known to any party in advance. Miners with an `n`-fraction (as usual, `n >= 2` due to the 51% assumption) of the hashrate have a `1/n` chance of being allowed to set the difficulty, and thus have a `1/n` chance of being able to successfully predict it 2 weeks in advance (by generating fake proofs, and then setting the difficulty such that they appear valid). Generalized, this is a $1/n^t$ chance of successfully predicting difficulty `t` adjustment periods (`2t` weeks) in advance. Therefore the use of the light relay provides stronger security properties to stateless SPV proofs when used as an additional validation step, as even entities with significant mining resources have a greatly reduced chance of creating fake proofs.

## Lots

Deposits will be managed in fixed-size *lots*. Each deposit therefore will have to be of the same amount: 1.0 BTC. Thus, a depositor submitting their proof of deposit must prove that they deposited 1.0 into the deposit's signing group wallet. If a depositor wants to deposit more than the lot size, they will need to create multiple deposit requests and fund multiple deposits. This allows each deposit to be backed by a different signing group, both simplifying the bonding of signing groups and improving the resilience of the system to signing group failure.

## Deposit Economics

Once a deposit has been made and funded, the corresponding TBTC is minted. Minted TBTC is immediately issued to the funder, who is now the beneficiary of a funded *Deposit*. To prevent denial of service attacks 0.005 TBTC is withheld on minting. This will be returned to the beneficiary when the *Deposit* is closed. This ensures that DoS attacks based on repeatedly creating signing groups (e.g. Attacker creates signing group, locks 1 BTC, creates 1 TBTC via a Deposit, trades for 1 BTC in an exchange and repeats the Deposit process multiple times) have high economic cost.

Beneficary status is transferable (in Ethereum this is implemented as an ERC721-compatible non-fungible token). When the *Deposit* resolves, the withheld TBTC (or equivalent value) will be returned to the current beneficiary along with a small additional payment. In this way the beneficiary NFT functions as a zero-coupon bond issued by the signing group upon funding. If the signing group performs its obligations, the beneficiary will eventually receive the bond payout. It can be expected that there will be service providers willing to trade {signer-fee-witheld} TBTC for 1 TBTC-coupon-bond along with some fee, for providing liquidity to holders of the otherwise illiquid for the duration of a Deposit coupon.

Signer fees are described in more detail in their own section.

## Bonding

Because signers are able to collude to censor withdrawals or abscond with funds, a bond is required per deposit from each backing signer.

Unlike the staked work tokens used to choose signers, signer bonds need to be a liquid asset with a large market cap. This restriction increases the cost of market-based attacks, where the price of bonded collateral can be pushed up or down by market manipulation.

Bonded signers offer depositors recourse in the case of colluding signers interfering with operation. A signing group that doesn’t sign within a timeout forfeits their bond; a signing group that provably signs unauthorized material forfeits their bond, and risks their work token stake.

### Acceptable collateral

Two tokens present themselves as obvious choices for signing bond collateral-- TBTC and the underlying work token. During the bootstrap phase of the network, neither is an appropriate candidate due to low liquidity.

Since signer bonds need to be denominated in a widely traded asset to avoid market manipulation, the next most obvious pick for bonding is the host chain’s native token. For the initial release of tBTC, that means ETH. As the ecosystem matures, other bond collateral options might become feasible at the expense of a more complex price feed implementation.

### Measuring security

Clearly, security concerns require signing bonds that are proportional to the size of a *Deposit*. To maintain a negative expected value from signers colluding, the amount forfeited by misbehaving signers must be strictly greater than the amount they have to gain. Assuming a lot size of 1 BTC, constant exchange rate between BTC and the bonded asset, and a M-of-N group of signers backing a *Deposit*, the minimum collateral for each signer is `(1 BTC)/M`, denominated in the asset being bonded, ETH in the base case.

Example: Consider a 1 BTC *Deposit* backed by a 3-of-5 group of Signers. In the worse case, 3 of the signers can be malicious and try to steal the Deposit, which would net them each 1/3 BTC. As a result, all 5 Signers must bond 0.33 BTC each, denominated in ETH.

## Pricing currency fluctuations

The above assumes a constant exchange rate between BTC and ETH, but in truth the two currencies fluctuate relative to each other, sometimes wildly.

### ETH Price Drop relative to BTC

If the value of ETH drops precipitously relative to BTC, then the dollar value of the ETH bonded by the signers can be less than the dollar value of the BTC Deposit they have backed, meaning they have positive expected value if they try to steal the BTC.

In order to avoid that, we require that the bonds are overcollateralized. For each ETH they collateralize, they must put up an additional 50%, for a total of 150% collateralization rate.

**Without overcollateralization:** Let 1 BTC be worth $10000, and 1 ETH be worth $200. Signers have to put up 50 ETH to back a deposit. Due to market conditions, ETH drops 25% to $150, while BTC maintains its value. The 50 ETH is worth $7500, meaning the Signers can make a $2500 profit by stealing the Deposit.

**With overcollateralization:** Let 1 BTC be worth $10,000, and 1 ETH be worth $200. Signers have to put up 75 ETH (150% of 50) to back a deposit. Due to market conditions, ETH drops 25% to $150, while BTC maintains its value. The 75 ETH is worth $11250, which is above the dollar value of BTC meaning the Signers will maintain honest behavior since they have more to lose.

In general, total overcollateralization of 150% (`3/2 * 100%`) keeps Signer incentives aligned with the well-being of the system up to a 33% drop (`(1 - 2/3) * 100%`) in price of the bonded asset against the Deposit’s asset. Increasing this percentage can increase the robustness of the system, at the expense of opportunity cost to the Signers which should be compensated via fees.

If the value of ETH crosses a security threshold, open *Deposit* s will enter pre-liquidation, followed by liquidation if they do not top up their collateral.

### BTC Price Drop relative to ETH

Since Custodial Fees are denominated per BTC in custody (with overcollateralization factored in), a BTC value drop against the bonded asset translates in lower fees for Signers. Note that this does not create any issue for tBTC reserves, but it makes the system less attractive to signers looking to earn interest via fees on their assets.

Signers SHOULD buy TBTC from the markets in anticipation of such overly overcollateralized Deposits and they SHOULD use it to redeem these positions, thus reclaiming their ETH liquidity which can be used to back other Deposits. An alternative would be to provide Signers with the ability to safely rebalance their bonds back to 150%, however that introduces implementation complexities and as a result is not the preferred solution for the initial deployment of the mechanism.

Example: Let 1 BTC be worth $10,000, and 1 ETH be worth $200. Signers have to put up 75 ETH to back a deposit. Signers are expected to make a custodial fee of 5 basis points for $15,000 (150% of $10,000): $7.5. Due to market conditions, ETH soars 25% to $250, while BTC maintains its value. The Signers still get $7.5 per BTC under custody, however the 75 ETH is worth $18750 (hence 187.5% overcollateralized), meaning 5 basis points for its custody would be $9.375. A signer redeems the Deposit by paying 1 TBTC, reclaiming 1 BTC and unlocking the 75 ETH which was locked by all Signers. All significantly overcollateralized Signers now have liquid ETH which they can use to back another deposit to mint new TBTC.

## A resilient price oracle

Unlike popular synthetic stablecoin schemes, the tBTC system design makes no effort to stabilize the value of TBTC relative to BTC — TBTC will be priced by the market. Instead, the goal is to ensure that the TBTC supply is strictly less than its backing BTC reserves.

For this reason, the only price relationship the system needs to understand is between the signing bond collateral and BTC.

In concrete terms, that means the price of ETH to BTC. Due to only needing prices for a single pair of assets, tBTC will initially use a simple price oracle.

## Undercollateralization

### Pre-liquidation: a courtesy call

At the first threshold of 125%, a *Deposit* enters pre-liquidation. Pre-liquidation indicates that the signers should close the *Deposit*_ or face forced liquidation after a pre-liquidation period. If the *Deposit* is not closed within 6 hours, or if the *Deposit* collateral falls below 110% collateralization, liquidation will follow. This gives each signer an incentive to close the position before it becomes severely undercollateralized. Alternatively, if the ETHBTC ratio recovers such that the deposit becomes at least 125% collateralized during the 6 hours the Deposit is safe and is moved away from the pre-liquidation state.

In future versions of the system, more complex pre-liquidation mechanisms could be introduced. For the initial version it seems prudent to choose a simple mechanism with large penalties for ongoing undercollateralization. In addition, by incentivizing redemption of undercollateralized or significantly overcollateralized positions, Signers are protected from being long ETH for long periods of time.

### Liquidation

Forced liquidation should be rare, as rational signers will redeem *Deposits* before liquidation becomes necessary. However, the possibility of extreme punishment via liquidation is necessary to prevent dishonest behavior from signers. Liquidation may occur because because signers didn't produce a valid signature in response a redemption request, because the value of the signing bond dropped below the liquidation threshold, because they did not respond to the courtesy call, or because the signers produced a fraudulent signature.

The primary goal of the liquidation process is to bring the TBTC supply in line with the BTC custodied by *Deposits*. The most valuable asset held by the system is the signers' bonds. Therefore, the liquidation process seizes the signers bonds and attempts to use the bonded value to purchase and burn TBTC.

First, the contract attempts to use on-chain liquidity sources, such as [Uniswap](https://hackmd.io/@477aQ9OrQTCbVR3fq1Qzxg/HJ9jLsfTz).

If the bond is sufficient to cover the outstanding TBTC value on these markets, it is immediately exchanged for TBTC.

Second, the contract starts a falling-price auction. It offers 80% of the signer bond for sale for the outstanding TBTC amount. The amount of bond on sale increases over time until someone chooses to purchase it, or the auction reaches 100% of the bond. The auction will remain open until a buyer is found.

TBTC received during this process is burned to maintain the supply peg. If any bond value is left after liquidation, a small fee is distributed to the account which triggered the liquidation. After that, any remaining value is either distributed to the signers (in case of liquidation due to undercollateralization) or burned (in case of liquidation due to fraud).

What the unresponsive signers do with the BTC outside the tBTC system design is for them to decide-- it might be split up, stolen by a signing majority, or lost permanently.

Example: 1. Signers guard a deposit of 1 BTC, backed by 75 ETH at 0.02 BTC/ETH (1.5 BTC in ETH, 150% collateralization ratio).

1. ETH price drops to 0.01333 BTC/ETH. 75 ETH now only collateralizes 100% of the Deposit (1 BTC / 75 ETH)
2. Liquidation is triggered and the 75 ETH is seized to buy back TBTC.
3. Assuming Uniswap has only 0.8 TBTC available in its reserves, that amount is bought, at market price, for 60 ETH (`0.8 BTC / (1/75) = 60`) and is subsequently burned. Note that there may be slippage here so the contract SHOULD check that it does not purchase TBTC at non-favorable rates
4. The Deposit is left with 15 ETH which must be used to purchase 0.2 TBTC. In an attempt to get a discount, it auctions 80% of its ETH reserves.
5. An arbitrageur burns 0.2 TBTC at 90% of the auction and obtains 13.5 ETH. The liquidation of the Deposit is now over.
6. The remaining 1.4 ETH is distributed to the signers (if they had committed fraud it’d be burned), and 0.1 ETH is given to the account which called the liquidation function on the Ethereum smart contract.
7. The N signers coordinate and agree on how they will distribute the 1 BTC deposit.

## Price Oracle

The price oracle is an integral part of the system, ensuring sufficient collateral backs all tBTC signers. We model the oracle after the USD price feeds from MakerDAO, operated initially by a single trusted actor and later governed by the ecosystem.

The minimal price feed design is specified completely by the interface below:

```
interface PriceFeed {
    function getPrice() external view returns (uint128);
    function updatePrice(uint128 price) public;
}
```

It is principally used for calculating the value of Bitcoin lot deposits, priced in Ether.

### Mechanisms of price feed updates

Price has a built-in expiration of 6 hours. In the unlikely event that the feed is not updated in a timely manner due to miner censorship or other attacks, calls to `getPrice` will revert transactions.

The price feed accepts updates that differ by at least 1% to the previously submitted price. This mitigates unnecessary recomputations by maintainers for price changes below the threshold. If the price is expiring within 1 hour, this check is ignored.

### Price encoding

A bitcoin has 8 decimal places, the smallest unit being a satoshi, meaning 100 000 000 satoshis = 1 bitcoin. An ether by contrast, has 18 decimal places, the smallest unit being a wei, meaning 1 000 000 000 000 000 000 wei = 1 ether.

To express the price of bitcoin relative to ether, we must use a ratio of the number of wei to a satoshi. A simple design is to use `x` wei : 1 satoshi. Hence, for a call to `getPrice` when 32.32 ETH : 1 BTC (Jun 2019), the value 323 200 000 000 wei is returned.

However, if 1 wei is worth more than 1 sat, then the price can no longer be accurately encoded. This scenario of a 'flippening', when 1 ether becomes worth 10,000,000,000x as much as 1 bitcoin, we find unlikely in the very short-term. Rather than prematurely optimize, incorporating a 2 integer ratio of `x` wei to `y` satoshi and changing the call semantics, we leave this as a future exercise for governance.

### Future design

The price oracle is integral to tBTC’s security and in the future, will be principally governed by the tBTC ecosystem. The first upgrades will focus on incorporating a medianizer model from MakerDAO, where multiple price feeds are voted in and the median price is calculated from their reports. Other on-chain price signals like decentralized exchanges (DEX’s) and liquidity pools (Uniswap) are being considered.

## Custodial Fees

Signers put their own funds at risk to assure depositors there will be no foul play. The bonds they put down are capital that could otherwise be productive, and need to earn a return relative to the risk to remain competitve with other opportunities.

## Paying for security

There are a number of pricing models that could cover the opportunity cost of signers' bonds. An adjacent space offers a strongly aligned pricing model.

Today’s centralized cryptocurrency custodians charge 50 to 75 basis points (between 0.5-0.75%) on *assets under custody (AUC)* per year. For each year that a centralized custodian protects a bitcoin deposit, that’s as much as 0.75% lost to the costs of custody.

A decentralized model should eventually allow a lower effective fee on custody by introducing more competition to the space. There’s a caveat, however — a decentralized approach to custodianship makes legal recourse more difficult, requiring additional bonded collateral to ensure recompense in case of failure.

Applying this pricing model to tBTC’s bonding, it’s clear that a Signer would like to make a similar return on the total capital it is locking up.

## Fee parameterization

### Terminology

- `Deposit`: A non-fungible smart contarct construct to which a signing group is assigned. It coordinates the creation and redemption of `LotSize * 1 TBTC`.
- `LotSize`: The exact value of a `Deposit` denominated in `BTC`.
- `OvercollateralizationFactor`: The additional amount which must be deposited as collateral by the Signer
- `BondValue`: The amount a `Signer` must lock in a smart contract as collateral to mint `TBTC`. Initially this will be denominated in `ETH`. `Deposit = OverCollateralizationFactor * LotSize * (ETHBTC conversion rate)`. In the future, `TBTC` may be used to collateralize a deposit. As a result, assuming a 1:1 ratio between `BTC` and `TBTC`, the price conversion can be skipped.
- `N`: The number of Signers authorized to sign on a `Deposit’s withdrawal request.
- `M`: The minimum number of Signers required to sign the authorization of a `Deposit’s withdrawal request.

### Description

It is assumed that each `Signer` contributes equally to the collateralization of a `Deposit`.

The capital cost per `Signer` is `BondValue / N`. Using `LotSize = 1 BTC` and `OverCollateralizationFactor = 150%`, that is `1.5 BTC / N`.

An initial parameterization of the system will use `15` Signers per lot. In addition, due to the lack of attributability in the aggregate signature mechanism used, we pick `M = N`. This requires a `0.1` BTC value in capital cost for **each** Signer per `1.0 TBTC` minted.

Taking into account the fees from centralized custodians (`0.0025-0.0075 BTC`), we choose to reward signers with a flat `0.005 TBTC` per `1.0 TBTC` minted, meaning the total signing revenue is `0.5%` of the market cap of the minted amount of `TBTC` each year.

## Signing

All of the aforementioned mechanisms require that there is a M-of-N multisignature wallet guarding each Deposit on Bitcoin.

Bitcoin’s consensus rules restrict script size to 520 bytes (10,000 bytes for Segwit outputs), limiting the maximum size of multisignature scripts to about 80 participants (OP_CHECKMULTISIG is limited to 20 public keys, but this can be bypassed by using OP_CHECKSIG ADD and <threshold> OP_GREATERTHAN as shown by Nomic Labs). Future proposals such as MAST would allow implementing larger multisigs, however the activation of new features on Bitcoin has historically been a procedure with unclear timelines.

Finally, large multisignature wallets in Ethereum and Bitcoin both have increasing verification costs as the number of participants increases. Building multisigs on Ethereum is particularly hard. By utilizing aggregate signatures with public key aggregation, we can remove all of the above complexities and replace them by a simple single signature verification.

Intuitively, an aggregate public key is generated from all multisignature participants who communicate via an out of band protocol, a process also known as Distributed Key Generation (DKG). Each participant signs the intended message with their private key and contributes a "share" of the final aggregate signature. Assuming ECDSA, the aggregate signature can then be verified against the aggregate public key with an OP_CHECKSIGVERIFY on Bitcoin, or an ECRECOVER operation on Ethereum. This process is simple and inexpensive, and avoids the path of implementing complex multisignature verification logic which can be upgraded for different M-of-N configurations. If another configuration is required, the script or the smart contract only needs to be configured to use a new aggregate public key after re-executing the DKG.

### Threshold ECDSA

For a private key x, a message M, a hash function H, and a uniformly chosen k, an ECDSA signature is the pair (r, s), where s = k (m + xr), r = R_x, R = g$^{(k}$-1) and m = H(m). Intuitively, this signature can be converted to a threshold signature if k and x are calculated via secret sharing between t of n protocol participants. Gennaro and Goldfeder’s paper describes an efficient mechanism for performing this procedure. Note that a similar mechanism was proposed by Lindell at al in the same year.

Informally, the participants perform the following actions to sign a message:

1. Produce an additive share of k * x, where each participant i holds k_i and x_i.
2. Efficiently calculate R = g^(1/k) using Bar-Ilan and Beaver’s inversion trick, without any participant i revealing k_i, and set r = R_x.
3. Each participant calculates their share of the signature: s_i = m * k_i + r * k_i * x_i.
4. The threshold signature is the sum of all signatures s_i.

A more in-depth description of the protocol can be found in Section 4.1 and 4.2 of the paper.

## Improved Fault Attribution

Currently, when Signers misbehave, all of their security bonds are seized and burned. If the system is parameterized to use `M-of-N` multisigs to back deposits, this means that if `M` parties misbehaved, the bonds of all `N` parties would be slashed. This is a griefing vector which we ideally would like to avoid. Accountable-subgroup multisignatures (described in Section 4 of the related paper) allow distinguishing a signature made by a subgroup `S` in a `M-of-N` multisignature from another subgroup `S'`. This can be leveraged to penalize only the `M` misbehaving signers, removing the risk of punishing honest signers.

The threshold-ECDSA protocol described in the previous section does not support fault attribution to subgroups of signers. We will deploy tBTC without that feature, and enable it in future protocol upgrades.

## Future Signature Schemes

In this section we explore other aggregate signature schemes we may use in the future. The described techniques are secure in the plain public key model, meaning that users do not need to prove ownership of their secret key, making them attractive for usage in blockchains. We briefly describe MuSig and BLS signatures.

### MuSig

*Note: This section is taken from the last section of the official MuSig blogpost by Blockstream*

1. Let `H` be a cryptographic hash function
2. Call `` L = H(X_1,X_2,⋯)` ``
3. Call `X` the sum of all `H(L,X_i) * X_i`
4. Each signer chooses a random nonce `r_i`, and shares `R_i = r_i * G` with the other signers
5. Call `R` the sum of the `R_i` points
6. Each signer computes `s_i = r_i + H(X,R,m) * H(L,X_i) * x_i`
7. The final signature is `(R,s)`, where `s` is the sum of the `s_i` values
8. Verification must satisfy: `sG = R + H(X,R,m) * X`

Contrary to earlier constructions, this signature verification algorithm is secure against rogue key attacks because X is defined as a weighted sum of the signers' public keys, where the weighting factor depends on the hash of all participating public keys.

### Pairing based multisignatures

Building on the work from MuSig and BLS signatures, Boneh, Drijvers and Neven introduce an efficient variant of previous BLS signature constructions which requires only 2 pairing operations for verification and is also secure against rogue key attacks.

This multisignature is shorter than MuSig since it is only 1 group element instead of 2. MuSig also requires an additional round of communication to generate the nonce `R`, which is not present in BLS. All signers can send their signatures to a third party who will aggregate them, removing the need for further interaction and for all parties to be online at the same time.

*Note: This section is taken from the Section 1 of the official related post by Dan Boneh et al*

1. Call `e: G_0 x G_1 → G_T` a bilinear non-degenerate pairing that’s efficient o compute, `g_0` and `g_1` generators of `G_0` and `G_1` respectively.
2. Call `sk` the user’s secret key, and `g_1^{sk}` their public key.
3. Call `H_0` a cryptographic hash function from the message space to `G_0`
4. Call `H_1` a cryptographic hash function from `G_1^n` to `R^n`
5. A signature on `m` is `s_i = H_0(m)^sk_i`
6. To aggregate N signatures for the same message from public keys `(pk_1, …, pk_n)`:
    1. Compute: `(t_1, …, t_n) = H_1(pk_1, …, pk_n)`
    2. Aggregated signature: `s = s_1^t_1 * … * s_n^t_n`
7. To verify the aggregated signature against the same public keys:
    1. Compute: `(t_1, …, t_n) = H_1(pk_1, …, pk_n)`
    2. Compute the aggregate public key: `pk = pk_1 ^ t_1 * … * pk_n ^ t_n` (independent of the message being signed))
    3. Verify the signature: `e(g_1, s) = e(pk, H_0(m))` (requires 2 pairings since the same message is being signed):

# Handling Failure

## Aborts / Liveness

The system requires that critical actions like funding and redemption occur within a fixed time after request. Failure to do so is treated as an "abort." Where fraud indicates proof positive of forbidden behavior, an abort typically represents a liveness failure from some participant. As such, while aborts are still punished, and may still result in liquidation, they are not punished as severely as fraud. For example, should the signers fail to produce a redemption signature in a timely manner, their bonds are liquidated to protect the supply peg, but any remainder is returned to them.

### Fraud

The system recognizes two redundant fraud proofs: ECDSA, in which the signing group produces a signature on a message which was not explicitly requested, and SPV, in which the UTXO custodied by the signing group moves for an unknown reason. Intuitively, the UTXO should not be able to move without ECDSA fraud, however, the system accepts either proof (and may require both proofs in certain cases). Each proof its own limitations and security parameters. When fraud is detected, the system penalizes the signers by seizing their bonds and starting the Liquidation process.

#### ECDSA Fraud Proofs

The signers collectively control an ECDSA keypair. By cooperating, they can produce signatures under the public key. Signers are charged with producing certain signatures (e.g. on a redemption transaction during the redemption process). Any valid signature under the signers' public key, but not specifically requested by the system is considered fraud.

An ECDSA fraud proof is simply a signature under the signers' public key, the signed message digest, and the preimage of that digest. From there we perform regular ECDSA verification. If the preimage matches the digest and the signature on the digest is valid but the digest was not explicitly requested by the system, then we can be sure that the signer set is no longer reliable. It is worth noting here, that verification of the preimage-digest relationship may not be skipped. Given any public key, it is possible to construct a signature under that public key and select a digest that matches it. Which is to say, anyone can produce an apparently valid signature on any unknown message. Only direct verification of the preimage's existence (via checking its relationship to the signed digest) prevents this attack as the attacker would have to invert the hash function to forge this relationship.

Notionally, the system can verify any signature the signers produce. However, the capabilities of the host chain set practical limitations. For instance, on Ethereum, only certain digest functions are available, so we cannot verify signatures on digests produced by unsupported hash functions. As a practical example, this precludes verification of Decred signatures, which use blake256. Signers in an Ethereum-hosted system can produce signatures on Decred transactions with no possibility of punishment.

All host chain impose costs on argument size, Therefore cost of verification scales with the length of the preimage. This means that it may not be economically feasible to verify signatures on very long pre-images, or that attempting to do so will exceed resource-use limitations (e.g. Ethereum's block gas limit). Fortunately, Bitcoin's signature hash algorithm uses double-sha256. This means that the preimage to the signed digest is always 32 bytes. As such, verification costs never scale with transaction size, and even very large transactions do not evade ECDSA fraud verification.

#### SPV Fraud Proofs

The signers custody a single Bitcoin UTXO. If that UTXO moves, except at the direction of the system then the signers have failed to perform their duties. SPV Proofs of Bitcoin inclusion (as documented here) suffice to prove signer fault. If the coins move, and its movement was not specifically requested by the system, then the signers have failed in their custodial duties. Compared to ECDSA Fraud proofs, SPV Fraud Proofs are more expensive to verify and have a weaker security model. The system expects SPV Fraud Proofs only rarely, and subjects them to much higher work requirements than SPV funding and redemption proofs.

# Redemption

## Overview

Deposits represents real Bitcoin unspent transaction outputs ("UTXOs") and are redeemable for the BTC held there. The tBTC redemption system aims to provide access to those BTC via a publicly-verifiable process. To support this goal, the redemption flow has been designed such that any actor may perform its critical actions (with the exception of producing signatures).

So long as the deposit is maintained in good standing, anyone may request redemption. To do so, the requester must repay outstanding TBTC (plus accrued custodial fees) and provide their Bitcoin payment details. At this point, the redemption process may not be cancelled. Once redemption has been requested, the signers must produce a valid Bitcoin signature sending the underlying BTC to the requested address. After a signature has been published, any actor may build and submit a *redemption transaction* to the Bitcoin blockchain using that signature.

## Redemption Requests

If the deposit is in good standing (has not been accused of fraud, or entered signer liquidation), and 6 months has elapsed since the deposit was created, anyone may request redemption. To do so that person makes a *redemption request* transaction to the smart contract on the host chain. The *redemption request* includes the following:

1. A fee amount
   - must be >=2345 satoshi (~20 satoshi/vbyte)
2. A public key hash (PKH) for BTC delivery
   - the 20-byte hash160 of a public key belonging to the requester
   - for security and privacy, this should be a new keypair
3. *Deposit* size plus fees (see Repayment Amount)

Upon receipt of the *redemption request*, the smart contract burns TBTC equal to the *Deposit* size, distributes signer fees and the beneficiary bond, and records the receipt of the request, and notifies the signers that a signature is required.

Once notified of the redemption request, the signers must wait for confirmation on the host chain. If they do not wait for confirmation, the redemption request may be dropped from the chain via a reorg, in which case any signature they produced could be used to both redeem the BTC and submit a signer fraud proof. A fraud proof created this way would appear valid to the host chain smart contract because it no longer has a record of the redemption request.

## Repayment Amount

The repayment amount is the *Deposit* size plus the custodial fee of 0.005 TBTC (50 basis points) and a payment of 0.0005 TBTC to the deposit beneficiary. This ensures that the signers are paid upon providing a signature and that the beneficiary is compensated for opening the deposit.

## Redemption Transaction Format

A redemption transaction has a perfectly canonical format which is embedded in the smart contracts running on the tBTC host chain. This prevents a number of complex attacks on the tBTC supply peg, as well as simplifying contract logic. The requester may specify only 2 aspects of the transaction: its fee and its destination. All other deposit-specific information (e.g. the outpoint and the UTXO size) is known to the deposit contract in advance.

The *redemption transaction* has 1 input (the deposit UTXO) and 1 output (the redemption output). It does not have change outputs, or additional inputs, as none are needed. It simply transfers the underlying BTC to the sole custody of the requester. Its timelock and sequence numbers are set to 0 and its version is set to 1. Full documentation of the format and the construction of its sighash can be found in the appendix

Because the format is simple and canonical, any observer may use publicly available information to build it. Once a signature has been published, it is simple to add a witness to the transaction and broadcast it. So while signers have a strong incentive to broadcast the transaction as early as possible, anyone may do so if the signers do not.

## Redemption Proof

A *redemption proof* is an SPV proof that a *redemption transaction* was confirmed by the Bitcoin blockchain. Once a request to redeem is confirmed, the deposit smart contract expects a *redemption proof* within 12 hours. To validate a *redemption proof*, the smart contract performs normal SPV proof verification, and additionally verifies that the recipient matches the requester's pulic key hash, and the value is greater than or equal `UTXO Size - highest allowed fee` (see Allowing for Bitcoin Fee Adjustment for more details).

## Validating a Signature

After the redemption request is sufficiently confirmed, the signers MUST produce a signature on the *redemption transaction* signature hash as requested. They have 3 hours in which to produce either a signature, or a *redemption proof* before being subject to penalties. Upon submission of a valid signature a *redemption proof* is still required, but the deadline is extended to 12 hours in total.

As discussed earlier, the host chain smart contract managing the deposit has all information necessary to calculate the *redemption transaction* signature hash. This includes the signers' threshold public key. Using the public key, the signature hash, and the redemption request the smart contract can know both the cryptographic validity of the signature and that a signature on that digest was requested as part of a redemption process.

## Allowing for Bitcoin Fee Adjustment

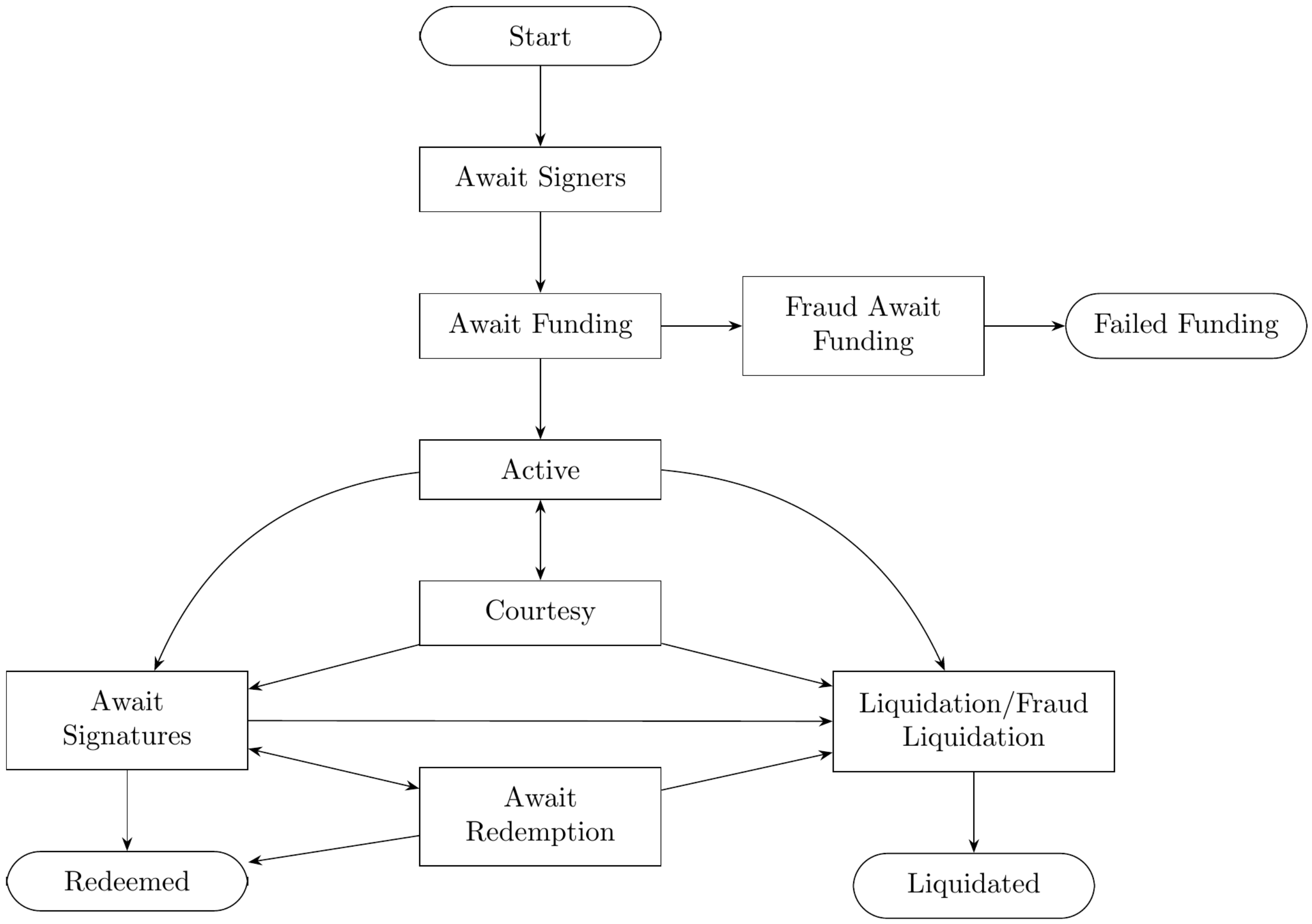
Because Bitcoin fees are determined by network congestion and other highly unpredictable factors, the requester may not select an appropriate fee. Signers are punished if no redemption proof is submitted **or** if they sign without explicit authorization. This could creates a no-win scenario for signers, in which they could not get the requester's transaction confirmed in the current fee climate and would eventually be punished despite honest behavior. Unfortunately, we cannot rely on the requester to stay online or update fee rates honestly. Ergo, the system requires some mechanism to fairly adjust fee rates without the requester's explicit consent.

The simplest scheme is to allow signers to increase the fee without requester consent after a timeout. As such, we allow signers to increase fees linearly every 4 hours. Which is to say, if the fee is `f`, after 4 hours the signers may notify the deposit contract of a fee increase to `2f` and if the transaction remains unconfirmed after , the signers may notify the contract of a fee increase to `3f`. This ensures that a redemption transaction will eventually be confirmed on the Bitcoin blockchain near the minimal fee rate given current network congestion. To prevent the signers from repeatedly requesting fee increases, they must actually provide a signature at each fee level. This ensures that each feerate is actually attempted before an increase is requested.

# Appendix

## Appendix

### Deposit and redemption state machine



We model each deposit as a simple state machine.

### Funding Flow

#### Overview

This is the process to set up a deposit, and fund it with BTC. Upon successful funding, the funder will own and new *Deposit* and will be able to create new TBTC. To start the funding process, a funder places a small bond, and requests creation of a new keep to custody BTC. If a new keep is successfully formed, the Keep contracts notify the *Deposit* of the signing group's public key. If keep setup fails, the funding process is aborted and the Keep system punishes the faulting parties.

Once a keep is formed, the funder transfers BTC to the keep's pay to witness public key hash (p2wpkh) address. This BTC becomes the underlying collateral for the new *Deposit*. The funder proves deposit via a stateless SPV proof of inclusion in the Bitcoin blockchain. If the funder fails to make this transfer in a timely manner, the funding process is aborted and the funder's keep bond is forfeit.

Once BTC collateralization has been proven, the *Deposit* becomes active. Then the funder may withdraw TBTC, up to the amount of BTC collateral (less the reserved TBTC). The funding process can only result in an active *Deposit* or an abort state.

#### States

`START`

- Deposit does not exist yet

`AWAITING_SIGNER_SETUP`

- The funder has placed a bond, and requested a signing group
- The Keep contracts must select a signing group or return failure

`AWAITING_BTC_FUNDING_PROOF`

- A signing group has been formed, and their public key hash returned
- The funder MUST return a SPV proof of funding before a timeout

`FRAUD_AWAITING_BTC_FUNDING_PROOF`

- Signing group fraud has been detected before the funding proof has been provided
- Signers bonds are seized when this state is entered.
- If the funder can provide a funding proof in a reasonable amount of time, then they will receive the singer bonds
- If the timeout elapses, signer bonds will be partially slashed and then returned.
- NOTE: the timeout on this state should be relatively short. We want to make it risky for a depositor who *has not already funded* when this state is entered to fund in order after this state is entered in order to try to receive the full signer bond amount

### Reachable exterior states

- `FAILED_SETUP`
    - via a timeout in `AWAITING_SIGNER_SETUP`
    - via a timeout in `AWAITING_BTC_FUNDING_PROOF`
    - via any state transitin from `FRAUD_AWAITING_BTC_FUNDING_PROOF`
- `ACTIVE`
    - via `provideBTCFundingProof`

#### Internal Transitions

`createNewDeposit`

- Anyone may put up a bond requesting a new signer group be formed
- **access control**
    - anyone
- **writes**

    - `mapping _depositBeneficiaries(address ⇒ address)`
        - on the TBTC system contract. for 721 compatibility, use uint256 when calling
- **from**
    - `START`
- **to**
    - `AWAITING_SIGNER_SETUP`

`notifySignerSetupFailure`

- Keep contract (or anyone else after a timer) notifies the deposit ath signer group setup has failed (or at least not proceeded in a timely manner)
- **access control**
    - Keep contracts
    - anyone (after a timeout)
- **from**
    - `AWAITING_SIGNER_SETUP`
- **to**
    - `FAILED_SETUP`

`notifySignerPubkey`

- Keep contract notifies the Deposit of its signing group's public key
- **access control**
    - Keep contracts
- **args**
    - `bytes _keepPubkey`
- **writes**
    - `bytes32 signingGroupPubkeyX;`
        - The X coordinate of the signing group's pubkey
    - `bytes32 signingGroupPubkeyY;`
        - The Y coordinate of the signing group's pubkey
    - `uint256 fundingProofTimerStart`
        - Start the funding proof timer
- **from**
    - `AWAITING_SIGNER_SETUP`
- **to**
    - `AWAITING_BTC_FUNDING_PROOF`

`notifyFundingTimeout`

- Anyone may notify a Deposit that its funder has failed to submit a funding proof. The funder's bond is forfeit due to non-completion at this point

- **access control**
    - anyone
- **reads**
    - `uint256 fundingProofTimerStart`
- **from**
    - `AWAITING_BTC_FUNDING_PROOF`
- **to**
    - `FAILED_SETUP`

`provideFundingECDSAFraudProof`

- Provide a fraud proof before a funding SPV proof has been verified
- The funder’s bond is returned here
- Signer bonds are seized here
- We consider this to be a different transition than `provideECDSAFraudProof` because it yields a different state. This also prevents edge cases with very short-lived deposits
- **access control**
    - anyone
- **args**
    - `bytes _signature`
        - The purportedly fraudulent signature
    - `bytes _digest`
        - The digest on which the signature was made
    - `bytes _preImage`
        - The sha256 preimage of that digest (on Bitcoin txns, this will always be the 32 byte intermediate sighash digest)
- **reads**
    - `bytes32 signingGroupPubkeyX;`
        - The X coordinate of the signing group’s pubkey
        - to check that the signature is valid
    - `bytes32 signingGroupPubkeyY;`
        - The Y coordinate of the signing group’s pubkey
        - to check that the signature is valid
    - `uint256 fundingProofTimerStart`
        - don’t allow this state transition if the funder has timed out
- **writes**
    - `uint256 fundingProofTimerStart`
        - update the funding proof timer for the new fraud time period

- **from**
  - `AWAITING_BTC_FUNDING_PROOF`
- **to**
  - `FRAUD_AWAITING_BTC_FUNDING_PROOF`

`notifyFraudFundingTimeout`

- Anyone may notify a Deposit that its funder has failed to submit a funding proof during the fraud period. The funder is not penalized for this
- When this occurs, signer bonds are partially slashed and then returned
- The partial slash is distributed to the current beneficiary
- We consider this to be a different transition than `notifyFundingTimeout` because it yields a different state and has different behavior
- **access control**
  - anyone
- **reads**
  - `uint256 fundingProofTimerStart`
    - for determining timeout of proof period
- **from**
  - `FRAUD_AWAITING_BTC_FUNDING_PROOF`
- **to**
  - `FAILED_SETUP`

`provideFraudBTCFundingProof`

- Anyone may notify a Deposit that its funder has sent funds to the signers' Bitcoin public key hash
- If this occurs, signer bonds are distributed to the funder
- We consider this to be a different transition than `provideBTCFundingProof` because it yields a different state and has different behavior
- **access control**
  - anyone
- **from**
  - `FRAUD_AWAITING_BTC_FUNDING_PROOF`
- **to**
  - `FAILED_SETUP`

#### External Transitions

`provideBTCFundingProof`

- Funder (or anyone else) provides a proof of BTC funding for the Deposit The funder’s bond is returned once this proof is successfully verified

- **access control**
  - Anyone
  - expected: funder
- **args**
  - `bytes _tx`
  - `bytes _proof`
  - `uint _index`
  - `bytes _headers`
- **writes**
  - `bytes8 depositSizeBytes`
    - size of UTXO in satoshis
  - `bytes utxoOutpoint`
    - unique identifier for the UTXO
- **from**
  - `AWAITING_BTC_FUNDING_PROOF`
- **to**
  - `ACTIVE`

### Redemption Flow

#### Overview

This is the process to redeem a deposit. Once started, redemption cannot be cancelled, except by proving signer fraud. Cancellation is impossible because as soon as redemption is requested the signers are permitted to sign, and a signature (even one neither chain knows about) can’t be revoked.

Ergo, cancellation of this process could result in BTC moved from the signers' address, and an Active Deposit with TBTC outstanding. This would result in a broken supply peg.

The requester notifies the `Deposit` of the bitcoin tx information (fee and recipient pubkeyhash) they are requesting, along with enough TBTC to cover the outstanding TBTC from the `Deposit`, plus enough to cover signer fees and the funder bond payment.

#### States

`AWAITING_WITHDRAWAL_SIGNATURE`

- A redemption has been initiated
- The signers MUST sign a digest
- The signers may return the signature for verification
- **NOTE**: there is a disincentive to return a signature, as the caller must pay for ecrecover gas and storage slot updates (to transition states).

`AWAITING_WITHDRAWAL_PROOF`

- The signers has returned a valid signature on the message
- The signers MUST provide a settlement proof
- In happy cases, we may skip the this state entirely.

**Flow reachable from**

- `ACTIVE`
    - via `requestWithdrawal`

**Reachable exterior states**

- `LIQUIDATION_IN_PROGRESS`
    - via an ECDSA or BTC fraud proof
    - via a state timeout
- `REDEEMED`
    - By providing a valid proof showing payment to the requester

**Internal Transitions**

`provideWithdrawalSignature`

- signers provide a valid ECDSA signature under their pubkey
- **access control**
    - Anyone
    - expected: 1 or more signers
- **args**
    - `uint8 _v`
    - `bytes32 _r`
    - `bytes32 _s`
        - The redemption signature
- **reads**
    - `bytes32 signingGroupPubkeyX;`
        - The X coordinate of the signing group's pubkey
    - `bytes32 signingGroupPubkeyY;`
        - The Y coordinate of the signing group's pubkey
    - `uint256 withdrawalRequestTime`
    - `bytes32 lastRequestedDigest`
        - Only accept signatures on the *most recent* requested digest
- **from**
    - `AWAITING_WITHDRAWAL_SIGNATURE`

- **to**
    - `AWAITING_WITHDRAWAL_PROOF`

`increaseWithdrawalFee`

- Explicitly allow a new signature with an increased fee. The fee may increased in linear steps over time. The new fee must be explicitly authorized by the contract, and the authorizing tx confirmed, before a new signature is created. To prevent bad behavior, signers must provide a signature at each fee level well before the next increase is available.
- **access control**
    - Anyone
        - after a timer
- **args**
    - `bytes8 _previousOutputValue`
        - the previous output value
    - `bytes8 _newFee`
- **reads**
    - `uint256 initialWithdrawalFee`
    - `bytes requesterPKH`
    - `uint256 block.timestamp`
- **writes**
    - `uint256 withdrawalRequestTime`
        - rewrite this time to give signers a time extension
    - `bytes32 lastRequestedDigest`
        - update the most recently requested signature
- **from**
    - `AWAITING_WITHDRAWAL_PROOF`
- **to**
    - `AWAITING_WITHDRAWAL_SIGNATURE`

`provideWithdrawalProof`

- signers provides a valid Bitcoin SPV Proof of payment to the requester
- **access control**
    - Anyone
    - expected: 1 or more signers
- **args**
    - `bytes _bitcoinTx`
    - `bytes _merkleProof`
    - `bytes _bitcoinHeaders`
- **reads**

    - `bytes requesterPKH`
    - `uint256 oracleDifficultyReq`
        - from oracle contract
    - `uint256 depositSize`
    - `uint256 initialWithdrawalFee`
- **writes**
    - `mapping(address ⇒ uint256) balances`
        - on TBTC ERC20 Contract
        - 1 time for each signer
        - 1 time for the deposit contract
- **from**
    - `AWAITING_WITHDRAWAL_PROOF`
    - `AWAITING_WITHDRAWAL_SIGNATURE`
- **to**
    - `REDEEMED`

#### External Transitions

##### `requestWithdrawal` (inbound)

- Anyone requests a withdrawal
- **access control**
    - Anyone
- **args**
    - `bytes8 _outputValueBytes`
    - `bytes _requesterPKH`
- **reads**
    - `mapping(address ⇒ address) depositBeneficiaries`
        - for auth
    - `bytes utxoOutpoint`
        - For calculating the sighash
    - `bytes20 signerPKH`
        - For calculating the sighash
    - `bytes8 depositSizeBytes`
        - For calculating the sighash
- **writes**
    - `mapping(bytes32 ⇒ uint256) wasRequested`
        - record that the digest was requested
    - `uint256 initialWithdrawalFee`

        - the requested withdrawal fee
    - `bytes20 requesterPKH`
        - the bitcoin hash160 pubkeyhash to which to deliver BTC
    - `uint256 outstandingTBTC`
        - check that the `Deposit’s TBTC has been returned
        - this is a derived attribute from UTXO size, the signer fee, and the funder bond value
    - `uint256 withdrawalRequestTime`
        - start timeouts for signers wrt signing and withdrawal
    - `mapping(address => uint256) balances`
        - change requester balance on TBTC ERC20 Contract
    - `uint256 totalSupply`
        - change total supply (burn) on TBTC ERC20 Contract
    - `bytes32 lastRequestedDigest`
        - record the digest as the newest
- **from**
    - `ACTIVE`
- **to**
    - `AWAITING_WITHDRAWAL_SIGNATURE`

#### `provideECDSAFraudProof` (outbound)

- **access control**
    - anyone
- **from**
    - `AWAITING_WITHDRAWAL_PROOF`
    - `AWAITING_WITHDRAWAL_SIGNATURE`
- **to**
    - `LIQUIDATION_IN_PROGRESS`

#### `provideSPVFraudProof` (outbound)

- **access control**
    - anyone
- **from**
    - `AWAITING_WITHDRAWAL_PROOF`
    - `AWAITING_WITHDRAWAL_SIGNATURE`
- **to**
    - `LIQUIDATION_IN_PROGRESS`

`notifyRedemptionProofTimeout` **(outbound)**

- **access control**
  - anyone
- **from**
  - `AWAITING_WITHDRAWAL_PROOF`
- **to**
  - `LIQUIDATION_IN_PROGRESS`

`notifySignatureTimeout` **(outbound)**

- **access control**
  - anyone
- **from**
  - `AWAITING_WITHDRAWAL_SIGNATURE`
- **to**
  - `LIQUIDATION_IN_PROGRESS`

## Frauds & Aborts

### Overview

Fraud and abort processes handle custody failures. This includes punishing signers and starting the bond liquidation process. These transitions can be invoked from almost any *Deposit* state, as faults may occur during any other flow. Once fault has been proven, the bonds are put up for auction to the public via the Liquidation flow.

While there is no fraud or abort state per se, it seems helpful to put the fraud-related state transitions in a single document.

### States

`COURTESY_CALL`

- The signers have been courtesy called
- They SHOULD request redemption of the deposit
- Anyone may request redemption

`LIQUIDATION_IN_PROGRESS`

- Liquidation due to undercollateralization or an abort has started
- Automatic (on-chain) liquidation was unsuccessful

`FRAUD_LIQUIDATION_IN_PROGRESS`

- Liquidation due to fraud has started
- Automatic (on-chain) liquidation was unsuccessful

`LIQUIDATED`

- End state
- The bonds have been liquidated and the position has been closed out

### Flow reachable from

- `ACTIVE`
- `AWAITING_WITHDRAWAL_SIGNATURE`
- `AWAITING_WITHDRAWAL_PROOF`
- `SIGNER_MARGIN_CALLED`

#### Internal Transitions

`purchaseSignerBondsAtAuction`

- anyone may purchase the seized signer bonds at auction
- **access control**
    - anyone
- **reads**
    - `uint256 liquidationInitiated`
        - for calculating auction value
    - `mapping (address => uint256) balances`
        - on the TBTC token contract
- **writes**
    - `mapping (address => uint256) balances`
        - on the TBTC token contract, to burn tokens
- **from**
    - `FRAUD_LIQUIDATION_IN_PROGRESS`
    - `LIQUIDATION_IN_PROGRESS`
- **to**
    - `LIQUIDATED`

`notifyCourtesyTimeout`

- Anyone may poke the contract to show that the courtesy period has elapsed
- Starts signer liquidation for abort
- **access control**
    - anyone
- **reads**
    - `uint256 courtesyCallInitiated`
- **writes**
    - `uint256 liquidationInitiated`

        - the timestamp when liquidation was started
- **from**
    - `COURTESY_CALL`
- **to**
    - `LIQUIDATION_IN_PROGRESS`

`notifyUndercollateralizedLiquidation`

- Anyone may notify the contract that it is severely undercollateralized
- Undercollateralization does not halt the redemption process. Only fraud does.
- **access controls**
    - anyone
- **reads**
    - ORACLE
- **writes**
- **from**
    - `ACTIVE`
    - `COURTESY_CALL`
- **to**
    - `LIQUIDATION_IN_PROGRESS`

#### External Transitions

`provideECDSAFraudProof`

- Anyone provides a valid signature under the signers' group key. Proof is fraud if the signature is valid and was not explicitly requested.
- **access control**
    - anyone
- **args**
    - `bytes _signature`
        - The purportedly fraudulent signature
    - `bytes _publicKey`
        - The public key to verify the signature under (must match signer account)
    - `bytes _digest`
        - The digest on which the signature was made
    - `bytes _preImage`
        - The sha256 preimage of that digest (on Bitcoin txns, this will always be the 32 byte intermediate sighash digest)
- **reads**

```
bytes32 signingGroupPubkeyX;
```

- - The X coordinate of the signing group’s pubkey
    - to check that the signature is valid
  - `bytes32 signingGroupPubkeyY;`
    - The Y coordinate of the signing group’s pubkey
    - to check that the signature is valid
  - `mapping(bytes32 => uint256) wasRequested`
    - check whether the signature was requested
- **from**
  - `AWAITING_SIGNER_SETUP`
  - `AWAITING_BTC_FUNDING_PROOF`
  - `ACTIVE`
  - `AWAITING_WITHDRAWAL_SIGNATURE`
  - `AWAITING_WITHDRAWAL_PROOF`
  - `SIGNER_MARGIN_CALLED`
- **to**
  - `FRAUD_LIQUIDATION_IN_PROGRESS`

`provideSPVFraudProof`

- Anyone provides a SPV proof that the Deposit UTXO has been consumed. If the proof is valid at recent difficulty, it is proof of signer fraud.
- **access control**
  - anyone
- **args**
  - `bytes _tx`
    - the bitcoin tx
  - `bytes _proof`
    - the bitcoin merkle inclusion proof
  - `uint _index`
    - the index of the leaf in the merkle tree (1-indexed, sorry)
  - `bytes _headers`
    - the header chain, earliest first, no padding
- **reads**
  - `bytes utxoOutpoint`
    - check if the tx spends the deposit outpoint
  - `uint256 currentDifficulty` — from light relay
    - check if the proof difficulty matches bitcoin main chain
- **from**
  - `AWAITING_SIGNER_SETUP`

    - `AWAITING_BTC_FUNDING_PROOF`
    - `ACTIVE`
    - `AWAITING_WITHDRAWAL_SIGNATURE`
    - `AWAITING_WITHDRAWAL_PROOF`
    - `SIGNER_MARGIN_CALLED`
- **to**
    - `FRAUD_LIQUIDATION_IN_PROGRESS`

`notifyRedemptionProofTimeout`

- Anyone may poke the contract to show that a redemption proof was not provided within the permissible time frame. Treated as Abort
- **access control**
    - anyone
- **reads**
    - `uint256 withdrawalRequestTime`
        - for checking if the timer has elapsed
- **writes**
    - `uint256 liquidationInitiated`
        - the timestamp when liquidation was started
- **from**
    - `AWAITING_WITHDRAWAL_PROOF`
- **to**
    - `LIQUIDATION_IN_PROGRESS`

`notifySignatureTimeout`

- Anyone may poke the contract to show that a redemption signature was not provided within the permissible time frame. Treated as Abort
- **access control**
    - anyone
- **reads**
    - `uint256 withdrawalRequestTime`
        - for checking if the timer has elapsed
- **writes**
    - `uint256 liquidationInitiated`
        - the timestamp when liquidation was started
- **from**
    - `AWAITING_WITHDRAWAL_SIGNATURE`
- **to**

    - `LIQUIDATION_IN_PROGRESS`

`notifyCourtesyCall`

- Anyone may notify the contract that it is undercollateralized and should be closed
- **access controls**
    - anyone
- **reads**
    - ORACLE
- **writes**
    - `uint256 courtesyCallInitiated`
        - timestamp when the call was initiated
- **from**
    - `ACTIVE`
- **to**
    - `COURTESY_CALL`

`notifyDepositExpiryCourtesyCall`

- Anyone may notify the contract that it has reached its end-of-term
- This triggers the courtesy call phase
- **access controls**
    - anyone
- **reads**
    - `block.timestamp`
    - `uint256 DEPOSIT_TERM_LENGTH`
        - tbtc constants
- **writes**
    - `uint256 courtesyCallInitiated`
        - timestamp when the call was initiated
- **from**
    - `ACTIVE`
- **to**
    - `COURTESY_CALL`

**exitCourtesyCall**

- During a courtesy call period, if the deposit is not expired
- Anyone may notify the contract that it is no longer undercollateralized
- This returns the contract to `ACTIVE` state
- **access controls**

    - anyone
- **reads**
    - `block.timestamp`
    - `uint256 fundedAt`
        - to check if the deposit is expiring
    - `bool getCollateralizationPercentage() < TBTCConstants.getUndercollateralizedPercent()`
        - Check the oracle to see if collateral is sufficient
- **from**
    - `COURTESY_CALL`
- **to**
    - `ACTIVE`

## Simplified Payment Verification (SPV): A Primer

#### Overview

While a full discussion of SPV proofs is outside the scope of this document, it is important to develop a working understanding of their properties, as many system-critical processes rely on the SPV security assumptions. SPV proofs are used during the funding, redemption, and fraud processes to provide the host chain with information about the state of the remote chain. Practically speaking, there is no other way that the host chain can learn about the state or history of the remote chain.

### Objectivity in Proof of Work

The SPV proofs used in this system rely on a property of Proof of Work (PoW) called "objectivity." Simply put, proof of work cannot be forged and no outside information is needed to check its validity. Without knowing the history of the chain, we can examine a Bitcoin block header and determine (probabilistically) how many hashes were performed to generate it. The number of hashes used to generate a header represents an unforgable cost inherent to that header, independent of its context or history.

Contrast this with Proof of Stake, in which the cost of generating a header is dependent on the entire history to date. We cannot know whether staker signatures represent the current validator set without complete history. In other words, Proof of Work in isolation still carries meaning, while Proof of Stake in isolation does not. While SPV inspection of Proof of Stake systems is possible, the security model is completely different. In addition, implementation approaches are much more costly than SPV inspection of objective systems. As such, this section concerns itself only with verification of Proof of Work, and future versions of the system utilizing SPV inspection of Proof of Stake systems are left for another day.

### Security Model

In Nakamoto Consensus, each node follows the heaviest valid chain. "Heaviest" refers to the objective proof of work metric. The chain with the most accumulated work is deemed the heaviest chain. Validity within the consensus is a bit more involved. Conceptually, nodes agree to evaluate new information according to a set of rules, and to reject anything that does not meet those rules. In practice, these rules define blocks consisting of headers and transactions, describe the format of transactions, and provide some user-programmable rules like Script and the EVM. Protocol-following nodes will always make the same validity decisions and will always choose the heaviest header chain containing only valid transactions and blocks. Therefore honest nodes will always reach the same state, which is to say, will always reach consensus.

The SPV security model is strictly weaker than the Nakamoto Consensus model, but still sufficient for our purposes. The SPV model checks work on headers, but enforces only a small subset of the validity rules. In essence, SPV verifiers assume that miners will not spend resources producing proofs of work on top of invalid blocks or transactions. They check validity of some set of headers, including verifying the work included in those headers, but do not verify each transaction. Instead, SPV verifiers check only transactions in which they have some interest. In the context of tBTC, we are interested only in specific UTXOs on the Bitcoin blockchain, so we validate only the transactions and headers related to those UTXOs, rather than all transactions.

When the assumption fails, and significant work is put on top of invalid transactions, the security model may also fail. We call these "fake" proofs and "fake" headers, because they are not semantically valid Bitcoin transactions or headers. We argue that fake proofs will be extremely rare. Our argument against them is rooted in the objective economics of Proof of Work. If a miner chooses to devote resources to producing work on top of an invalid transaction she must give up mining rewards while still bearing the electricity and hardware costs of mining. She gives up mining rewards because the invalid transaction may never be included in the main Bitcoin chain. It will be rejected by all fully validating nodes. Therefore producing a fake proof has a large inherent cost. We argue that the system is economically secure so long as the cost of producing a fake proof is high and the value that can be gained by producing a fake proof is orders of magnitude less than that cost.

The security of SPV systems also benefits from a built-in assumption of the Nakamoto Consensus model: that no attacker has greater than 50% of the hashrate. Assuming that is true, no attacker can generate Bitcoin proofs of work faster than the main Bitcoin blockchain. This implies that honest headers are generated (within the tolerance of the Poisson distribution) before any dishonest header. Extending the model, if no attacker has greater than an `n`-fraction of the current Bitcoin hashrate (where `n >= 2`) then honest headers may be generated $n^{-1} - 1$ times faster. For example, an attacker controlling 25% (1/4) of the Bitcoin hashrate could generate a header on average every 40 minutes. The main chain, slowed by the loss of that 25%, would generate a header every 13 1/3 minutes — three times faster. To take advantage of this, the proof must commit to some recent information that was previously unknown to the attacker, e.g. a past block header, or a new public key hash. This provides a lower bound on the time at which the attacker begins to generate a false proof.

### Relays

The most conceptually straightforward SPV system is a relay. In a relay system each Proof of Work header is submitted to and verified by the host chain. The host chain smart contracts keep track of the best known header, and all past headers seen. An SPV proof in a relay system demonstrates that a transaction is confirmed by the best-seen header and is deep enough that its disconfirmation is unlikely. Each additional header in a relay, as in the consensus it tracks, secures all previous headers. So we grow more certain of older chain events over time.

### Stateless SPV

Where relays decline to check validity, stateless SPV systems both decline to check validity and fail to follow the heaviest chain. In fact, a stateless SPV system does not track anything at all. Instead stateless SPV proof relies entirely on the objective work present in a discrete slice of headers. A stateless SPV proof consists of one or more transactions, merkle proofs of inclusion for those transactions, and a set of consecutive headers on top of those transactions. A verifier can then inspect the headers, and give the proof an objective quality score based on the amount of work in those headers. Anyone interested in using the state and history information in the stateless SPV proof's information can determine whether to accept or reject it based on the proof's quality.

Stateless SPVs are relatively recent work. Their compelling advantage is size and cost-efficieny. A stateless SPV proof is less than 1KB, all of which can be discarded after validation. A relay, on the other hand, stores each header on-chain. This means a relay will consume linearly increasing state space over time. Maintenance costs are already unsustainably high, as evidenced by the failure of BTCRelay in December 2017. Given the already high cost of on-chain storage and the likely introduction of state rent in major host chain candidates, relying on a stateful relay seems short-sighted. A high-state system that is viable today may not be viable in the future.

We argue that for recent transactions stateless SPV's security is equivalent to a relay's. An attacker would have to spend the same number of hashes to provide the relay with fake headers as it would to provide the stateless SPV verifier with a stateless proof with sufficient work. However, compared to relays, stateless SPV proofs do not gain security over time without extending each proof to include new headers. It is important to this argument that the recency of the transaction is known, without this, an attacker could begin to generate a proof well in advance of proving time, essentially getting a head start on the main chain. Relays get recency assurances at each block, as each new header must reference the header immediately preceding it, but a stateless SPV proof must get its recency from some outside source.

### Standardized Sighash Construction

#### Overview

For signing, Bitcoin transforms transactions using a process known as the SignatureHash (sighash) algorithm. The original sighash algorithm had many drawbacks and sharp edges. In SegWit scripts, the algorithm was changed to follow BIP143 (legacy addresses still use the original algorithm).

The goal of the sighash algorithm is to commit to selected aspects of the transaction in the signed digest. This prevents malleation, and indicates the signer's intent with respect to them. Specifically, BIP143 sighash commits to the following (not in this order):

1. One or All inputs
2. None, One, or All outputs
3. The specific prevout this signature witnesses
4. The pubkey script or redeem script code locking that prevout
5. The value of the prevout this signature witnesses
6. The sequence of the input spending that prevout

7. The transaction version
8. The transaction locktime

These are committed to via the double-sha256 of an ordered bytestring. This digest is signed, and can be reproduced by anyone inspecting the transaction (provided they have access to historical chain data to validate the prevout value). Sighash calculation is thus a crucial part of the Bitcoin consensus process.

Because a signing group may withhold signatures, the redemption flow forces them to provide a valid signature within a certain timeout. This implies that the redemption flow must be able to evaluate "validity" means in this context. Because the goal is redemption, a "valid" signature is one that witnesses a transaction that sends funds to the public key hash requested at the beginning of the redemption flow. In order to check that a given signature witnesses such a transaction, we need to enable our contracts to validate the sighash digest signed.

By far the easiest way to do this is to create a canonical transaction. We can then implement a greatly-reduced set of BIP143's functionality while still being able to assess signature validity during redemption. This allows us to, instead of calculating the sighash of an input transaction, specify a sighash, and force construction of a transaction that matches it. This way the contract can request extremely precise redemption transactions with minimal overhead.

#### Canonical Redemption Sighash

BIP143 follows this general format:

```
Double SHA256 of the serialization of:
   1. nVersion of the transaction (4-byte little endian)
   2. hashPrevouts (32-byte hash)
   3. hashSequence (32-byte hash)
   4. outpoint (32-byte hash + 4-byte little endian)
   5. scriptCode of the input (serialized as scripts inside CTxOuts)
   6. value of the output spent by this input (8-byte little endian)
   7. nSequence of the input (4-byte little endian)
   8. hashOutputs (32-byte hash)
   9. nLocktime of the transaction (4-byte little endian)
  10. sighash type of the signature (4-byte little endian)
```

Because we don't need to use timelocks in our redemption transaction, we forbid their usage, allowing us to immediately standardize many elements. We also forbid use of any sighash flag, other than `SIGHASH_ALL`, so we can standardize that as well. Here we replace those elements with the standardized hex strings:

```
Double SHA256 of the serialization of:
   1. 01000000
   2. hashPrevouts (32-byte hash)
   3. hashSequence (32-byte hash)
   4. outpoint (32-byte hash + 4-byte little endian)
   5. scriptCode of the input (serialized as scripts inside CTxOuts)
   6. value of the output spent by this input (8-byte little endian)
   7. 00000000
   8. hashOutputs (32-byte hash)
   9. 00000000
  10. 01000000
```

Forbidding the transaction to have more than 1 input or output gives us one additional victory. Point 3, `hashSequence` is defined as "the double SHA256 of the serialization of nSequence of all inputs." By having 1 input and disabling its timelock feature we can standardize this as well:

```
Double SHA256 of the serialization of:
   1. 01000000
   2. hashPrevouts (32-byte hash)
   3. 8cb9012517c817fead650287d61bdd9c68803b6bf9c64133dcab3e65b5a50cb9
   4. outpoint (32-byte hash + 4-byte little endian)
   5. scriptCode of the input (serialized as scripts inside CTxOuts)
   6. value of the output spent by this input (8-byte little endian)
   7. 00000000
   8. hashOutputs (32-byte hash)
   9. 00000000
  10. 01000000
```

Next, we fill in information that the contract has access to, starting with the details of its custodied UTXO. The Deposit contract has validated the SPV funding proof, and stored its value as well as its outpoint. BIP143 specifies `hashPrevouts` as "the double SHA256 of the serialization of all input outpoints" So we can populate steps 2, 4, and 6 using known information:

```
bytes8 depositSizeBytes
bytes utxoOutpoint

Double SHA256 of the serialization of:
   1. 01000000
   2. {hash256(utxoOutpoint)}
   3. 8cb9012517c817fead650287d61bdd9c68803b6bf9c64133dcab3e65b5a50cb9
   4. {utxoOutpoint}
   5. scriptCode of the input (serialized as scripts inside CTxOuts)
   6. {depositSizeBytes}
   7. 00000000
   8. hashOutputs (32-byte hash)
   9. 00000000
  10. 01000000
```

The scriptCode is also available to the contract, as it is derived from the signers' threshold public key hash. According to BIP143, "For P2WPKH witness program, the `scriptCode` is `0x1976a914{20-byte-pubkey-hash}88ac`."

```
bytes8 depositSizeBytes
bytes utxoOutpoint
bytes20 signerPKH

Double SHA256 of the serialization of:
   1. 01000000
   2. {hash256(utxoOutpoint)}
   3. 8cb9012517c817fead650287d61bdd9c68803b6bf9c64133dcab3e65b5a50cb9
   4. {utxoOutpoint}
   5.
       1. 1976a914
       2. {signerPKH}
       3. 88ac
   6. {depositSizeBytes}
   7. 00000000
   8. hashOutputs (32-byte hash)
   9. 00000000
  10. 01000000
```

This leaves us with only `hashOutputs` unknown to the contract at redemption time. Intuitively, this makes sense, as the contract knows where the money is, but not where it should be sent on redemption. As always, we reference BIP143 which says "hashOutputs is the double SHA256 of the serialization of all output amount [sic] (8-byte little endian) with scriptPubKey." This can get quite long with multiple outputs, but as mentioned earlier, we can standardize on single-output transactions. This means that it’s the double-sha256 of the 8-byte LE value being redeemed (less a mining fee), and the pubkey script containing the redeemer’s script hash. In our redemption flow, both of these things are set by the user at request time. This means the contract has access to them as function arguments when it requests that the signer group produces a signature. Therefore the contract can specify a precise digest for that signature:

```
bytes8 depositSizeBytes
bytes utxoOutpoint
bytes20 signerPKH

Double SHA256 of the serialization of:
   1. 01000000
   2. {hash256(utxoOutpoint)}
   3. 8cb9012517c817fead650287d61bdd9c68803b6bf9c64133dcab3e65b5a50cb9
   4. {utxoOutpoint}
   5.
       1. 1976a914
       2. {signerPKH}
       3. 88ac
   6. {depositSizeBytes}
   7. 00000000
   8.
       1. hash256(
       2. {_outputValueBytes}
       3. {_requesterPKH}
       4. )
   9. 00000000
  10. 01000000
```

It is easy to implement this as a pure function in Solidity:

```
/// @notice                 calculates the sighash of a redemption tx
/// @dev                    documented in bip143. many values are hardcoded
/// @param _outpoint        the bitcoin output script
/// @param _inputPKH        the input pubkeyhash (hash160(sender_pubkey))
/// @param _inputValue      the value of the input in satoshi
/// @param _outputValue     the value of the output in satoshi
/// @param _outputPKH       the output pubkeyhash (hash160(recipient_pubkey))
/// @return                 the double-sha256 (hash256) signature hash
function oneInputOneOutputSighash(
    bytes _outpoint,  // 36 byte UTXO id
    bytes20 _inputPKH,  // 20 byte hash160
    bytes8 _inputValue,  // 8-byte LE
    bytes8 _outputValue,  // 8-byte LE
    bytes20 _outputPKH  // 20 byte hash160
) public pure returns (bytes32) {
    // Fixes elements to easily make a 1-in 1-out sighash digest
    // Does not support timelocks
    bytes memory _scriptCode = abi.encodePacked(
        hex"1976a914",  // length, dup, hash160, pkh_length
        _inputPKH,
        hex"88ac");  // equal, checksig
    bytes32 _hashOutputs = abi.encodePacked(
        _outputValue,  // 8-byte LE
        hex"160014",  // this assumes p2wpkh
        _outputPKH).hash256();
    bytes memory _sighashPreimage = abi.encodePacked(
        hex"01000000",  // version
        _outpoint.hash256(),  // hashPrevouts
        // hashSequence(hash256(00000000))
        hex"8cb9012517c817fead650287d61bdd9c68803b6bf9c64133dcab3e65b5a50cb9",
        _outpoint,  // outpoint
        _scriptCode,  // p2wpkh script code
        _inputValue,  // value of the input in 8-byte LE
        hex"00000000",  // input nSequence
        _hashOutputs,  // hash of the single output
        hex"00000000",  // nLockTime
        hex"01000000"  // SIGHASH_ALL
    );
    return _sighashPreimage.hash256();
}
```

## Glossary

**Host chain**

The chain on which TBTC is minted

**Keep**

Secure multiparty computation setups powering tBTC signing

**PKH**

Public key hash

**Random beacon**

A secure, verifiable source of randomness accessible on the host chain.

## tBTC

**Deposit**

A deposit is the core component in the system architecture. Each deposit represents a set of bonded signers that generate a Bitcoin public key and accept a single Bitcoin UTXO, from which TBTC can be drawn.

**Deposit beneficiary**

A deposit has a single beneficiary Ethereum account — originally set to the depositor. The beneficiary owns the right to some fees on deposit redemption.

**Deposit request**

A request for signers to be selected and generate a new Bitcoin ECDSA keypair. A successful request yields a new Bitcoin address ready to accept funds as well as a set of bonded signers.

**Lot size**

The ideal size of a funded deposit's BTC UTXO. Standardizing lot sizes across deposits simplifies the BTC redemption process and pricing of deposits by the market.

**Signing bond**

The bond signers put up before a deposit is funded. This bond ensures signers will be punished for fraud or poor uptime.

**Reserved TBTC**

The amount of TBTC that can't be drawn from a new deposit. Reserving TBTC on deposit funding sets aside funds to pay signers' custodial fees through the deposit term.

## Cross-chain communication

**Consensus relay**

Chain-tracking SPV on some other chain, e.g. BTCRelay. Consensus relays are long-running cross-chain mechanisms that track the consensus state of another chain. They're distinguished from other uses of the term "relay", eg the "threshold relay" random beacon mechanism.

**SPV**

Simplified payment verification

**Stateless SPV**

Non-chain-tracking SPV

**Bitcoin & friends**

**P2PKH / P2WPKH**
: Pay to (witness) public key hash

**P2SH / P2WSH**
: Pay to (witness) script hash

**Sighash**
: Bitcoin signature hash algorithm

**BIP 143**
: Adopted SegWit sighash proposal

**ALL**
: Sighash mode committing to **all** outputs and **all** inputs

**SINGLE**
: Sighash mode committing to **one** output and **all** inputs

**ANYONECANPAY**
: Sighash modifier. Changes **all** or **single** to commit to only `ONE` input

**SACP, singleACP**
: `SINGLEANYONECANPAY`

**Hash160**
: Bitcoin hash function `rmd160(sha256(message))`. Used for PKH commitments in outputs. Used for SH commitments before segwit

**Hash256**
: Bitcoin hash function `sha256(sha256(message))`. Used for txids, sighash, merkle trees, and PoW

**UTXO**
: (unspent) transaction output

**Weight unit**
: a measure of bitcoin transaction size. Main transaction info is 4 weight units per byte. Witness info is 1 weight unit per byte

**Vbyte**
: 4 weight units

**Outpoint**
: A 36-byte string that uniquely identifies Bitcoin UTXOs. It consists of the creating transaction's `tx_id` as a 32-byte LE `uint256` (because Satoshi was bad), and a 4-byte LE `uint32` denoting the UTXO's position in the creating transaction's output vector

[1] The tBTC system participates in fairly limited fashion here, mostly coordinating work done in a secondary system responsible

for managing the secure random number generation, private data storage, and multiparty computation needed to provide the system’s relevant security properties. In this diagram, that role is fulfilled by the Keep network, described in its whitepaper. The Keep Random Beacon is described in more detail in the Keep Random Beacon yellowpaper.

[2] A system is only as decentralized as its most centralized component, so the beacon must be decentralized to achieve proper decentralization of the tBTC system as a whole.