A Comparison of Bitcoin Covenant Proposals for Vaults

Lillian Wang

Introduction

Bitcoin covenants restrict how an output may be spent by placing conditions in that output's locking script. Proposed covenant mechanisms differ in which transaction fields can be constrained, what conditions they impose on those fields, and how those constraints are expressed in Script. This report compares applications of covenant mechanisms to vaults, which are designed to give an owner the ability to recover funds in response to an unauthorized withdrawal attempt.

We focus on covenant behavior emulated by presigned transactions and constructions enabled by proposed consensus changes such as OP_CHECKTEMPLATEVERIFY (CTV), the BIP 118 SIGHASH_ANYPREVOUT modes (APO/APOAS), OP_TXHASH, OP_CAT, and OP_CHECKCONTRACTVERIFY (CCV). We discuss tradeoffs for each vault construction in terms of withdrawal flexibility, fee management, operational complexity, and on-chain cost. Our comparison suggests that CTV is suited to simple vaults with precomputed outputs, CCV best supports vaults requiring partial withdrawals or trigger-time selection of a withdrawal address, and TXHASH enables greater commitment flexibility but places more responsibility on the vault designer. Finally, APOAS and OP_CAT may be more appealing if their broader non-vault applications are also valued. We hope this report can be a step towards more rigorous and comprehensive discussions on covenant proposals.

Background/Preliminaries

Covenants & Vaults

We define a covenant as a spending condition within the locking script of a UTXO that restricts the structure or contents of its spending transaction, rather than only requiring authorization data in the scriptSig or witness of the input spending that UTXO. These restrictions may apply to the spending transaction's output scripts and amounts, its set of inputs, and sequence values.

An application of covenants is vaults, which are designed to provide security against unauthorized spends. One early construction was proposed by Möser et al. in the 2016 paper "Bitcoin Covenants." In this report, we define a vault as a construction where spending from a vault UTXO to initiate a withdrawal creates an intermediate, timelocked UTXO. While in this intermediate stage, the vault owner can take an alternate spend path which redirects funds to

a safer location. These definitions and related terminology are elaborated on in the Basic Vault Template section.

The main benefit vaults provide is *reactive security*. For example, if an attacker gains control of the vault owner's private key and attempts to initiate a withdrawal, the delay gives the owner or a watchtower service time to detect the attempt and redirect the funds to a separately secured recovery address. A recent example of this threat is the July Coldcard hack which allowed attackers to reconstruct private keys and drain funds from over 5,200 addresses.

Vault behavior can be emulated on Bitcoin today using presigned transactions with ephemeral-key deletion. However, these constructions introduce setup, storage, and flexibility tradeoffs which we discuss further in the Presigned Transactions section. Other proof-of-concept constructions such as ColliderScript and PIPEs are compatible with Bitcoin's current consensus rules. However, these approaches remain research constructions with high computational costs or advanced cryptographic techniques, making their practicality and security difficult to evaluate. We therefore compare how proposed changes to the Bitcoin protocol affect vault constructions along several axes such as on-chain cost, complexity, and customizability. Where relevant, we note applications beyond vaults, but a systematic comparison of generalizability is outside the scope of this report.

Basic Vault Template

To compare covenant or covenant-like mechanisms consistently, we use a simplified vault template:

1. *Vaulting/Setup Stage:* Funds are placed in a vault UTXO that can only be spent to an unvault UTXO described in step 2.
2. *Unvaulting/Trigger Stage:* The vault owner can create a trigger or unvault transaction, spending the vault UTXO and creating an unvault UTXO. The transaction's confirmation begins a relative timelock. The unvault UTXO can only be spent through one of the following two spend paths.
3. *Hot Spend/Withdrawal:* After satisfying the relative timelock, a withdrawal transaction can spend the unvault UTXO to a destination permitted by the construction. The resulting UTXO is referred to as the hot-spend UTXO and the address which controls this UTXO is referred to as the withdrawal address.
4. *Cold Spend/Recovery:* The unvault UTXO can alternatively be spent without delay to a prespecified safe address, such as a cold-storage address. The resulting UTXO is called a recovery UTXO and is created by a recovery or cold spend transaction.

In covenant-based implementations, the vault UTXO's spending conditions include a covenant to ensure any spending transaction must create an unvault UTXO with the specified spending conditions. A second covenant in the unvault UTXO's script requires every transaction bypassing its relative timelock to redirect all funds to the specified recovery address.

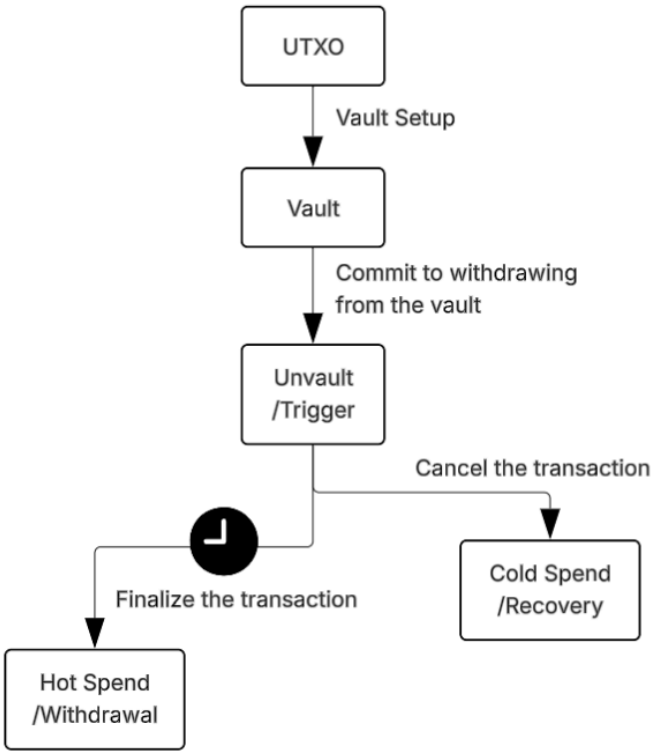

Fig. 1. A basic vault transaction graph. Boxes represent UTXOs with arrows depicting transactions that spend one UTXO and create the next. A clock represents the hot spend path's relative timelock.

Other extensions of the basic vault design include partial withdrawals which can unvault part of a vault UTXO's value and revault the remainder, batched withdrawals which allow withdrawals from multiple vault UTXOs in one transaction, and fee management techniques including the ability to add fee-paying inputs and create change outputs. We consider which extensions different covenant proposals can support beyond the basic vault template.

Comparison Factors

We compare each covenant and covenant-like mechanism's implementation of the basic vault and its ability to extend the vault to support additional features. A feature is supported if it can be expressed by modifying which fields to commit to or arguments to supply in each covenant mechanism. Some features may be supported by designing different constructions that are not explored in this work.

We consider three main features:

- *Partial Withdrawal:* This feature allows the vault owner to determine the value of the unvault UTXO and revault the remaining funds during the unvault stage rather than the setup stage.
- *Latest Withdrawal Address Commitment:* We compare the vault construction's ability to constrain the Hot Spend UTXO's address or spending conditions, which we also call the vault's withdrawal address. This can be constrained at vault setup, in the trigger transaction, or otherwise unconstrained. We determine the latest stage that can commit to the hot spend address (either setup or trigger).
- *In-Transaction Fee Management:* The ability to determine transaction fee source inputs and create change outputs within the unvault and recovery transactions. While fees may be sourced via anchor outputs with P2A, we consider the flexibility these vault constructions offer for calculating fees without additional transactions.

We consider three properties of each vault construction:

- *Conceptual Complexity:* This is a subjective measure of the difficulty ensuring security and correctness of the basic vault construction. In other words, how to ensure an adversary cannot introduce a new spend path outside of the predesigned paths.
- *Operational Complexity:* A subjective measure of the additional procedures required to operate the vault safely besides ensuring the correctness of the scripts and watchtower monitoring.
- *On-chain Cost:* We consider the on-chain cost in weight units of the full hotspend path and the coldspend path. This includes the setup transaction, unvault transaction, and hot/cold spend transaction. In order to compare the construction costs uniformly, we assume each transaction uses P2WSH or P2TR and includes the same number of inputs and outputs as in the OP_CAT Purrfect Vaults construction. Thus, the vault setup has 1 input and output, the trigger has 2 inputs and outputs, and the hot and cold spends each have 2 inputs and 1 output.
- *Taproot only:* This determines whether the proposed mechanism requires taproot to be enabled in order to use the mechanism.

Table 1 summarizes the comparison results for the different vault constructions. The following section introduces each proposal and analyzes the table entries and additional observations. General conclusions are made in the Comparisons section.

proposal	Partial Withdrawal	Latest Withdrawal Address Commitment	In-Transaction Fee management	Conceptual Complexity	Operational Complexity	On-chain Cost (Weight Units)	Taproot only
Presigned Txns	No	Setup	Sighash dependent	Low	High: presigning, key deletion, txn storage	2139 hot; 2138 cold	no
CTV	No	Setup	Fixed input slots, fixed output values	Low	Low: template transaction generation	2064 hot; 1990 cold	no
APOAS	No	Setup	Flexible inputs, sighash dependent outputs	Medium: understand sighash commitments	Low: output transaction generation	2139 hot; 2138 cold	yes
TXHASH	Partially	Setup	Flexible Inputs & Outputs	Medium: ensure selected fields maintain security	Medium: generate transactions conforming to selected fields	2074 hot; 2008 cold	yes
CCV	Yes	Trigger	Flexible Inputs, fixed total output value	Medium-High: key tweaking & output amount modes	Medium: satisfy output amount constraints.	2186 hot; 2177 cold	yes
CAT (Purrfect Vault)	No	Trigger	Fixed input slots, fixed output values	Very High: simple opcode but complex introspection construction	Medium-high: construction specific transaction & witness generation	3,011 hot; 2,888 cold	yes

Table 1: Comparison table for different proposed vault constructions.

Details on Vault Proposals

We now examine how each approach implements the basic vault template and supports the comparison factors described above.

Vaults with Presigned Transactions

Presigned transactions coupled with ephemeral keys can emulate covenant behavior and thus construct vaults within the existing Bitcoin protocol. A presigned transaction restricts the spending transaction's current input UTXO and all outputs, shown in figure 2.

In an ordinary spend, the spending transaction can be created and signed after the spent UTXO's transaction is broadcast. In a presigned transaction, the spending transaction is created and signed before the previous transaction is broadcast. This ensures the spend paths are protected before the first transaction is broadcast.

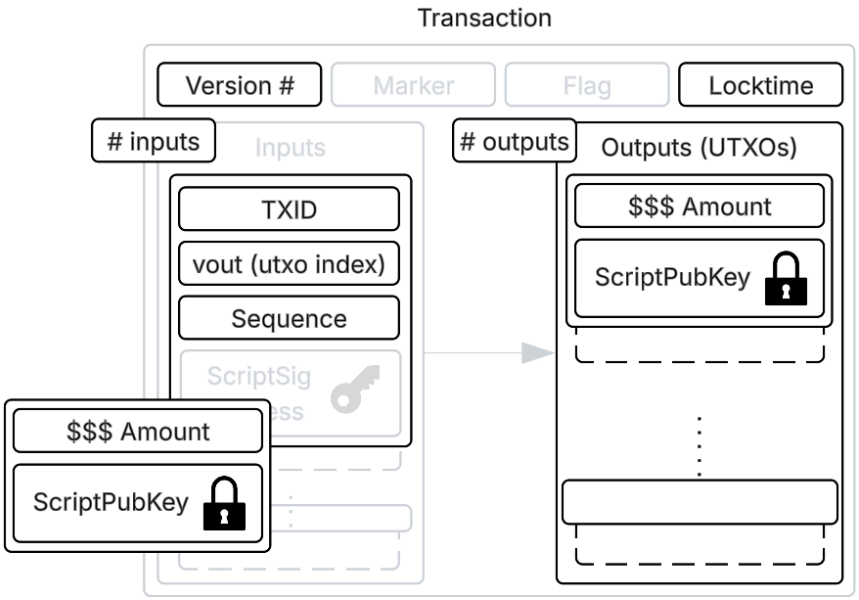

Fig 2. The transaction values a signature commits to with SIGHASH_ALL. The bottom left box represents the UTXO being spent.

Using ephemeral keys prevents new spend paths from being created. Ephemeral keys are signing keys we intend to delete immediately after use. By deleting the private key used for the presigned transaction's signature, we commit to the values signed since any alternate spending transactions would require new signatures from the deleted key.

Thus, a basic presigned-transaction vault follows the stages below:

- *Vaulting stage:* The owner creates the vault setup, unvault, and recovery transactions and signs the unvault and recovery transactions using ephemeral keys. Finally, they broadcast the vault transaction.
- *Unvaulting stage:* When the vault owner is ready to withdraw from the vault, they broadcast the unvault transaction.
- *Hot spend stage:* After the hot spend timelock is satisfied, the user can create a transaction selecting the IF branch by indicating a 1 in the witness. The vault owner can freely determine the rest of the transaction components.
- *Cold spend stage:* The user broadcasts the presigned recovery transaction. Its witness contains a 0 to direct code execution to the ELSE branch.

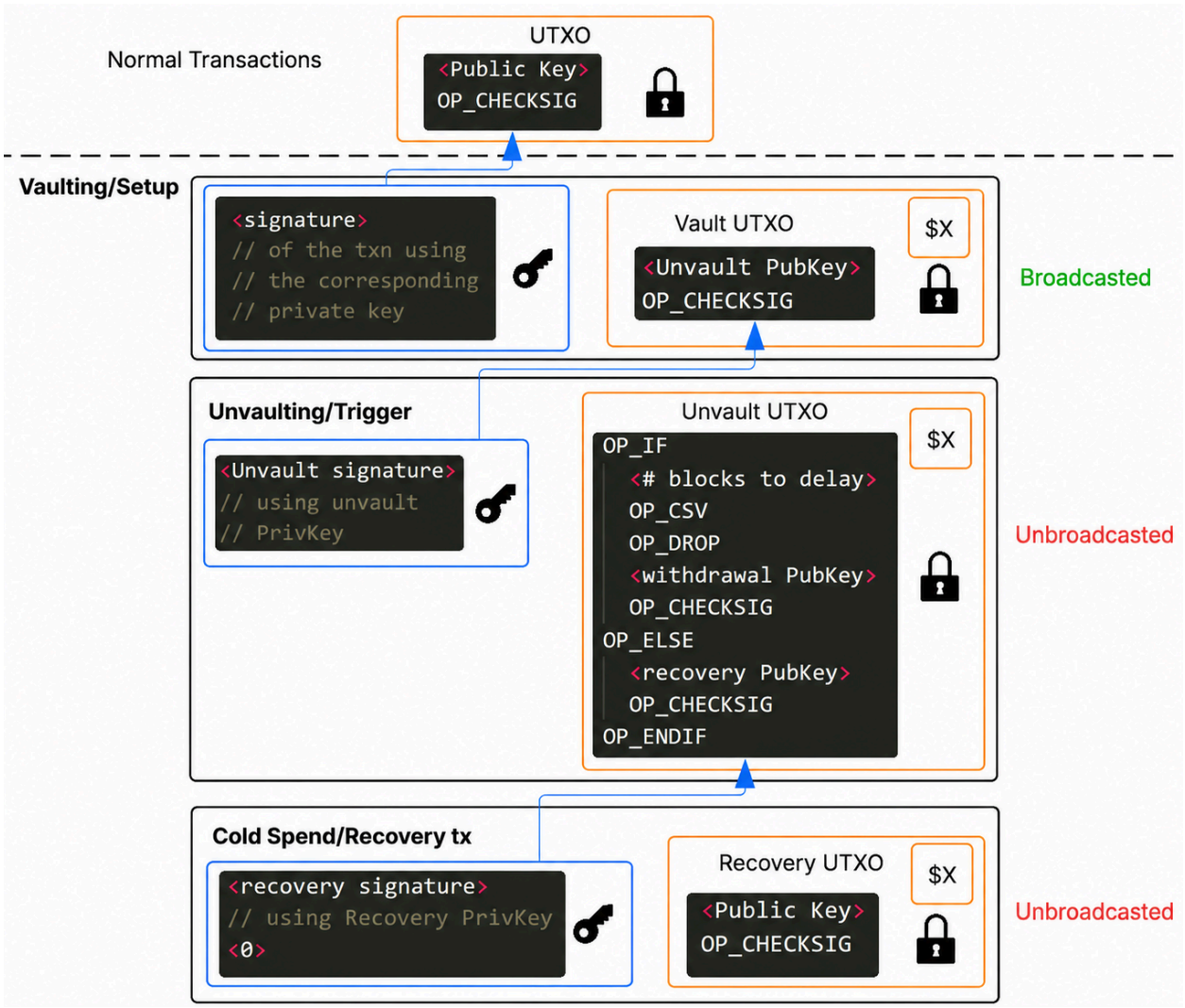

Fig 3. Vault Setup Stage using Presigned Transactions. Blue boxes represent witness arguments, orange boxes represent each UTXO's spending conditions, and blue arrows are drawn from inputs to the UTXO being spent. Using P2WSH, spending conditions are committed to in the UTXO and revealed as the spending input's witnessScript.

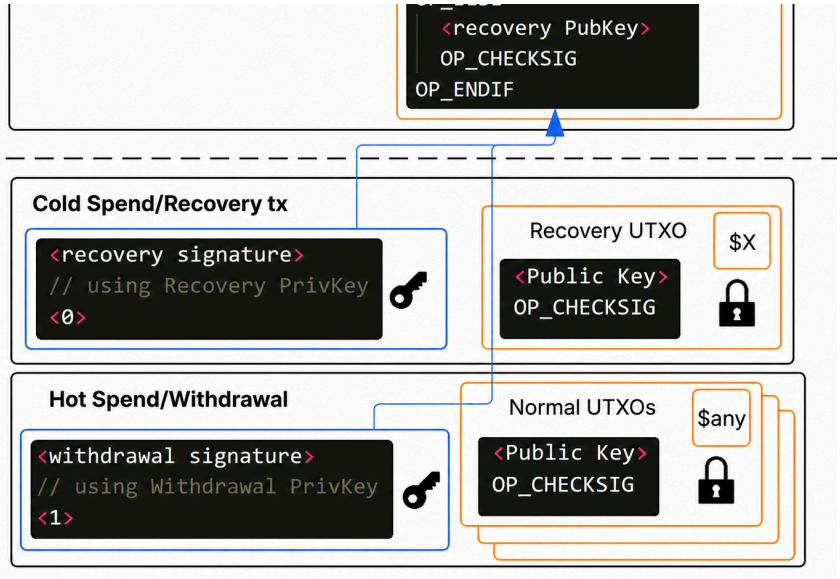

Fig 4. Possible hot and cold spends for the unvault UTXO.

While presigned transactions offer immediate mainnet implementability since they can be used under Bitcoin's current consensus rules, they introduce substantial utility and maintenance trade offs.

- *Transaction storage:* the user needs to store presigned transactions until they are used in the unvault stage or during a cold spend. If a user controls multiple vaults, the storage overhead becomes more pronounced. Additionally, if the user loses the presigned unvault transaction or loses access to the withdrawal address, the user loses the means to spend the vaulted UTXO, effectively burning it.
- *Key deletion:* Bitcoin consensus cannot enforce the signing key's deletion. Since vault security relies on the key deletion, security thus depends on additional trust assumptions, like secure hardware.
- *Fixed withdrawal address and amount*: Because the unvault and cold spend transactions must be constructed and signed beforehand, the entire transaction graph is precomputed. This means the hot spend amount and address is either unspecified as it is now or fixed at vault setup. Similarly, arbitrary partial withdrawals aren't supported since output amounts are prespecified. We discuss restrictions to the withdrawal address further in OP_CCV.
- *Fee Management:* Using SIGHASH_ALL, all outputs and inputs are fixed, meaning transaction fees within the recovery and unvault transactions must be predetermined at setup. With the SIGHASH_ANYONECANPAY flag, only the current input is included in the signature, so additional fee-paying inputs may be added. However, fee adjustments remain limited since outputs must still be predetermined.

The following vault proposals address these limitations and expand the enabled capabilities of vaults. We'll see that OP_CTV and APOAS vaults remain precomputed but do avoid key deletion and storage requirements. Meanwhile OP_TXHASH and OP_CCV support greater flexibility in fee management and partial withdrawals, and OP_CAT provides a lower-level opcode for more construction-dependent vault designs.

OP_CHECKTEMPLATEVERIFY Vaults (CTV)

OP_CHECKTEMPLATEVERIFY or BIP119 is an opcode which takes a 32-byte hash from the stack as input and asserts it matches the components of the spending transaction. The main components committed to are the number of inputs, the current input's index, and all outputs. This hash can be computed manually or using an additional opcode such as OP_TEMPLATEHASH documented in BIP 446.

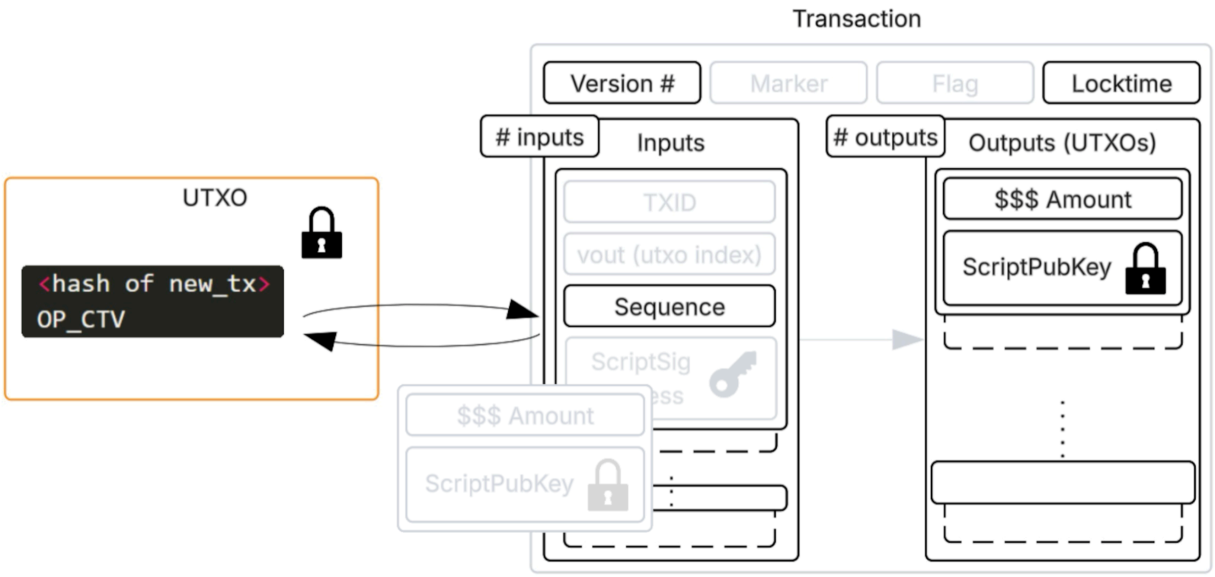

Fig 5. Left is an example of an OP_CTV call. Right are components of a CTV hash of a SegWit input, shown in black. Hashed input data includes the total number of inputs and each input's sequence number and ScriptSig if nonempty. Witnesses are not included.

Since TXIDs and the ScriptPubKey of the UTXO being spent are omitted, the spending transaction's hash can be included in the previous UTXO's script without creating a circular hash dependency. Thus, presigning is not necessary. Therefore, CTV enables a fully on-chain vault construction. An example is shown in figure 6.

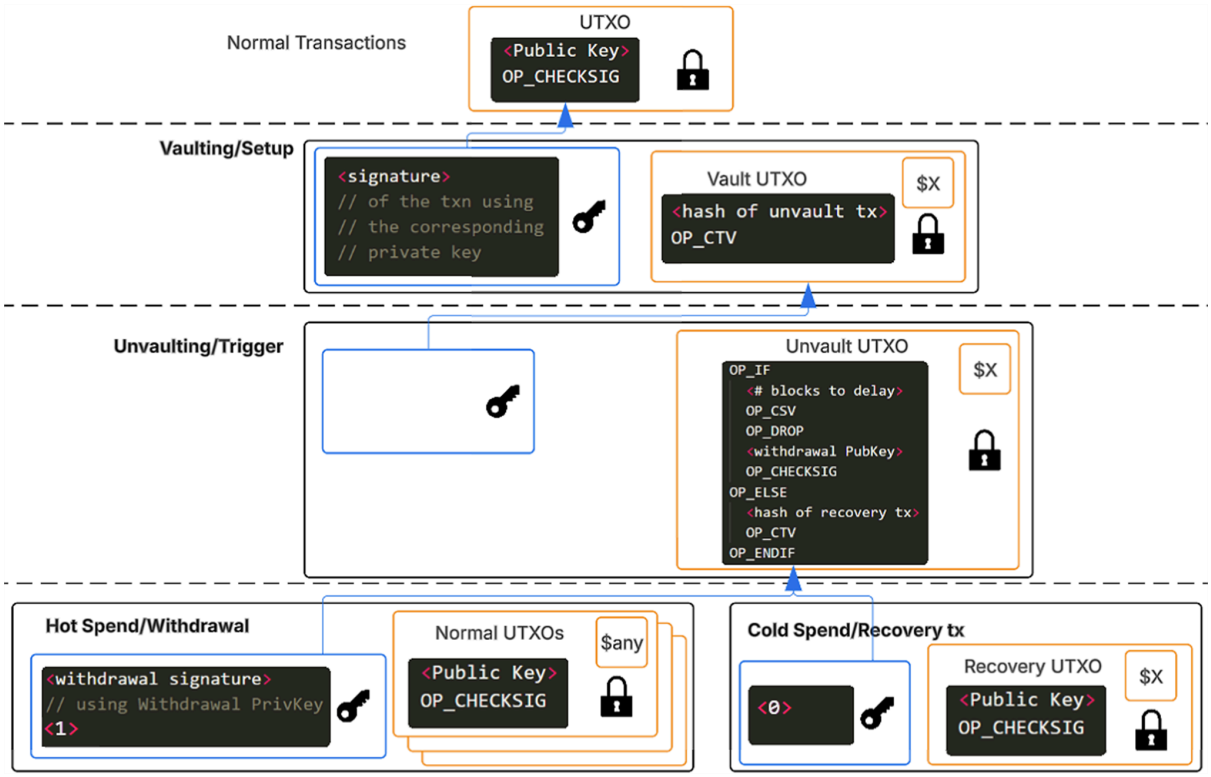

Fig 6. A basic vault created using OP_CTV and P2WSH scripts.

OP_CTV and Presigned transactions differ primarily in how they constrain inputs. Presigned transactions with SIGHASH_ALL commits to each input while OP_CTV only commits to the total number of inputs and each input's sequence numbers. Thus, which additional UTXOs are included in each of the spending transaction's input slots remain free to choose. Therefore, transactions have greater flexibility to choose fee-paying inputs. However, the outputs still remain fixed, again limiting the amount of fee adjustments that can be made after vault setup. This resembles SIGHASH_ANYONECANPAY signatures, except OP_CTV additionally constrains the number of inputs and which input index that UTXO occupies. Thus, for one transaction, OP_CTV offers greater fee management compared to SIGHASH_ALL presigned transactions and less compared to SIGHASH_ANYONECANPAY signatures.

In total, OP_CTV maintains a precomputed transaction graph meaning arbitrary partial withdrawal amounts are still unsupported, and withdrawal address restraints remain the same as in presigned transactions. We also note that while users don't need to store the exact spending transactions, they must still retain or be able to reconstruct the information committed in the CTV hash in order to reproduce the template spending transaction.

SIGHASH_ANYPREVOUTANYSCRIPT (APOAS)

BIP 118 introduces SIGHASH_ANYPREVOUT and SIGHASH_ANYPREVOUTANYSCRIPT, two sighash modes for BIP 118 public keys for tapscript. We'll refer to the former signature type as APO and the latter signature type as APOAS. In the SIGHASH_ALL construction here, APOAS commits to the current input's sequence number and all output UTXOs in the spending transaction. APO additionally commits to the amount and scriptPubKey of the spent UTXO.

Its original motivating use case was to increase efficiency for layer-two networks, although it may also be used as an alternative to OP_CTV. We see that APOAS commits to the same data as CTV does with the exception of the number of inputs, other inputs' sequence numbers, and the current input's index. Notably, the absence of the input TXIDs and UTXO outpoints allows APOAS signatures to reside in UTXO locking scripts rather than the corresponding witnesses.

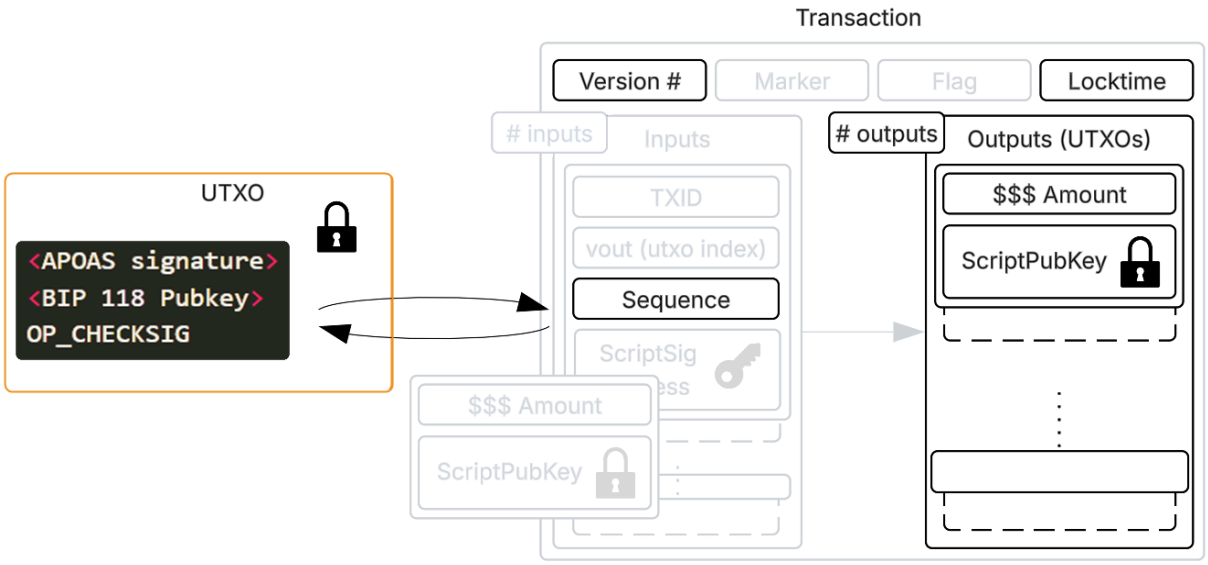

Fig 7. The values committed to in a SIGHASH_ANYPREVOUTANYSCRIPT signature.

Thus, APOAS signatures can be used similarly to OP_CTV by modifying the ScriptPubKey as in figure 8 and leaving the corresponding witness stack empty. Note that the two spend paths in the unvault UTXO are split into two tapleaves, with a NUMS Taproot internal key.

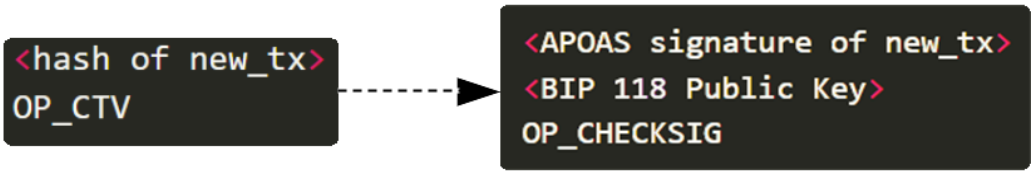

Fig 8. The substitution scripts for an OP_CTV call.

Compared to the signatures used in presigned transactions, APOAS allows vault constructions that avoid presigning and ephemeral keys. Meanwhile, since APO signatures include the previous UTXO's ScriptPubKey, these signatures cannot be moved into the locking script and thus also require presigning. We further note concerns about replaying BIP 118 signatures. In APOAS vaults, APOAS signatures reside in the spent UTXO locking script instead of the spending input witness, so it can only be evaluated by spending a UTXO committing to that script. Thus, replay should only be limited to other UTXOs sent to the same vault address. However, we still recommend using unique keys for each vault to avoid this risk.

Because APOAS signatures commit to less input information compared to CTV, APOAS has more freedom in managing fees or enabling fee bumping since additional fee paying inputs can be freely added. However, since the outputs remain fully prespecified, the APOAS construction faces the same limitations in adjusting change outputs, supporting partial withdrawals, and committing to withdrawal addresses and amounts.

OP_TXHASH (TXHASH)

The draft BIP 346 introduces OP_TXHASH which takes as input a variable length TxFieldSelector (TXFS) that indicates the transaction components to include, and outputs the 32-byte hash of the serialization of those components in the spending transaction. We can compare this produced hash to an existing hash using OP_EQUAL.

Note that when the TXFS is empty, TXHASH defaults to committing to the same components as the ones in CTV, although the resulting hash may not be entirely interchangeable with CTV hashes. With the addition of the opcode OP_CHECKSIGFROMSTACK which allows signature verification for arbitrary messages rather than only the current transaction hash, OP_TXHASH can also emulate APO or APOAS signature hashes. Thus, the basic CTV vault can be implemented by replacing each OP_CTV call with the top right script in figure 9.

Because the vault designer can choose which transaction fields to commit, TXHASH allows more flexible vault constructions compared to CTV or APOAS. For instance, additional inputs can be added if the hash only commits to the current input rather than all inputs. TXHASH can also partially support partial withdrawal by committing only to the ScriptPubKeys for two UTXOs: the unvault UTXO and another vault UTXO. This allows the vault owner to set the withdrawal amount at the unvault time, though TXHASH lacks the ability to require all remaining vaulted funds be revaulted. Additionally, the user can aggregate multiple vaulted UTXOs or withdraw from multiple vaults at once if the outputs are properly specified in each

vault. Additionally, by using multiple instances of OP_TXHASH, the user can enforce different restrictions on each output in a transaction.

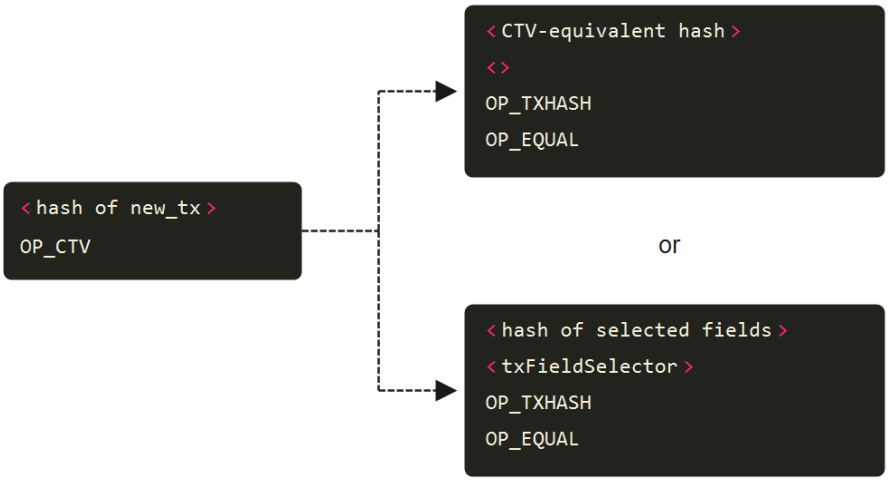

Fig 9. The substitution scripts for an OP_CTV call. Top right commits to the same components as in OP_CTV while bottom right verifies the components in a nonempty TXFS.

However, allowing more freedom to choose the commitment components also adds greater complexity and places more responsibility on the vault designer to maintain security. For example, if in the partial withdrawal construction, the unvault UTXO can contain a 3rd change output with unspecified value, then an adversary could drain all vaulted funds to the change UTXO, bypassing the unvault and revault UTXO requirements.

Other Proposals

OP_CHECKCONTRACTVERIFY or OP_CCV is an opcode in BIP 443 that commits to a specific input or output's P2TR contract in the spending transaction. This differs from previous proposals since each OP_CCV call only checks one input or output's script rather than the selected fields of the entire transaction. Additionally, the opcode can check that its input UTXO amount is distributed a specified way among a set of output UTXOs. Therefore vaults with partial withdrawals can be created by enforcing that all funds in a vault will be split among an unvault and a revault UTXO.

The Pymatt CCV-only minivault demonstrates how the withdrawal address can be selected in the unvault transaction rather than in the vault transaction or remaining unspecified as in the other proposed constructions. Committing to the withdrawal address during setup means the vault can essentially only be used once. To change the withdrawal address or amount, the user

would need to withdraw or recover all funds from the vault and create a new vault. Meanwhile, not constraining the hotspend address means unauthorized withdrawals look identical to authorized withdrawals up to the hot-spend. In comparison, committing to the withdrawal address in the trigger allows watchtower services to detect bad spends by determining if the set address is outside of some prespecified whitelist. As a tradeoff, withdrawals are more rigid: to change the withdrawal address after unvaulting, the vault owner would need to recover and revault the funds. The ability to create more dynamic vaults resulted in BIP 345 or OP_VAULT's withdrawal in favor of BIP 443.

OP_CAT concatenates two elements from the stack. This allows the user to construct messages using values from both the ScriptPubKey and ScriptSig or witness. One use-case is reconstructing a transaction's signature message from witness arguments and checking it against a precomputed signature. Thus, OP_CAT can support flexible covenant constructions, potentially enabling vault designs that impose different constraints on different inputs and outputs.

Compared to TXHASH, CAT-based covenants could commit to more custom messages. For instance, a CAT vault can require the withdrawal address to be specified in the unvault transaction rather than committed to at setup. However, enabling these constraints require much larger and more complex scripts. We also note that OP_CAT was not designed specifically for vaults or even covenants. Because the specifications for OP_CAT are very simple and ubiquitous, it enables many more functionalities outside of vaults such as Bitcoin virtual machines. For further details for OP_CAT-only vaults implementation, see the taproot wizards Purrfect Vault implementation.

Comparisons

Presigned transaction vaults are still inconvenient to use in practice compared to what is possible with new opcodes because they have precomputed transaction graphs and require secure key deletion and storage for presigned transactions. Presigned transactions remain the most practical way to implement vaults under Bitcoin's current consensus rules, although proof of concept constructions like ColliderScript and PIPEs can produce covenant-like behavior under impractical computational costs. Proposed covenant opcodes remove the need for secure key deletion and presigned-transaction storage but introduce different tradeoffs.

In the basic vault constructions we considered, CTV and APOAS have fully precomputed outputs and thus, like presigned transactions using SIGHASH_ALL, have vault behavior that is comparatively straightforward to analyze but do not support arbitrary partial withdrawals.

OP_CTV constrains the number of inputs, sequence values, and the current input index but not the input outpoints, allowing some flexibility in choosing fee-paying inputs. Meanwhile, APOAS omits more input information, meaning inputs can be added more freely. Additionally, since the committed values in APOAS and presigned transactions are determined by the SIGHASH flags, if flags other than SIGHASH_ALL are considered, there is more flexibility to adjust output values and add outputs to handle change.

TXHASH and CCV enable vaults with greater functionality, though the type of functionality enabled is different. TXHASH allows vault designers to select which transaction field and input or output indices to commit to. This versatility provides more support for fee management and partial withdrawals at the cost of greater burden on the implementer to ensure the correctness and security of each vault. CCV can enforce how the current input's amount is distributed across different outputs, supporting partial withdrawals that require remaining values to be revaulted. Additionally, CCV vaults can commit to the withdrawal address in the unvault transaction rather than only at the vault setup. However, unlike the other proposals, because CCV preserves input value across the outputs, transaction fees for unvault transactions cannot be sourced from the vault UTXO and fees for recovery transactions cannot come from the unvaulted UTXO. These complementary capabilities make combining CCV and TXHASH a promising direction for greater vault customizability, though evaluating such a construction remains outside the scope of this report.

OP_CAT can enable more introspection-like covenant constructions and greater vault functionality, depending on the construction. However, the basic Purrfect Vault construction examined here requires complex scripts and witnesses. This complexity makes its behavior harder to analyze and more susceptible to errors. Thus, without any other added opcodes, the construction discussed here remains a proof of concept.

Future Work

Through this project, we surveyed various popular vaults proposals and analyzed them according to a similar standard. In the future, we hope to be able to expand this project to provide a more thorough analysis of vaults. This includes a more thorough breakdown of CCV-based vaults and less well-known opcodes such as Tapleaf Update Verify (TLUV). We also currently only consider variations of basic vault implementations rather than full functionality constructions.

Moreover, this project only considers each proposed opcode individually whereas most debate in forums, the bitcoin mailing list, and other avenues of discussion needs to compare different

combinations of opcodes. For instance, OP_CTV is often paired with OP_CHECKSIGFROMSTACK (CSFS) and various proposals include OP_CAT as part of their proposals. Combining different opcodes could enable support for more versatile vaults which should be analyzed in more detail.

Finally, there are also many more axes of comparison to standardize such as the features discussed in the BIP 345 (OP_VAULT). These include batched withdrawals which is the ability to unvault from multiple vaults at once and vault consolidation, or the ability to combine multiple vault UTXOs into a single vault UTXO. While we discussed security where necessary, we did not thoroughly analyze avenues of attacks such as Denial of Service (DoS) or manipulating transaction fee values. We point to a comparison of attacks on CTV, CCV, and OP_VAULT constructions, found after this project was conducted, for work in this direction. We also leave how much privacy can be supported, such as whether or not vault-type UTXOs versus normal UTXOs are distinguishable, to future work.

Acknowledgements

I thank Michael Maurer and Neha Narula for their mentorship and feedback throughout this project. I also appreciate the feedback I received from members of the MIT Digital Currency Initiative.