Vaults and Covenants

James O'Beirne
vaults@au92.org

December 14, 2022

Abstract

Vaults are a technique for substantially reducing the risk of Bitcoin theft. In this paper, we examine their relationship to different covenant designs. We survey the vault implementations that are currently usable, as well as those that would be possible only with proposed consensus updates (e.g. `OP_CHECKTEMPLATEVERIFY`).

A new approach, `OP_VAULT`, is presented which avoids the pitfalls of a general covenant proposal while still allowing the general-covenant behavior necessary for a featureful vault implementation. The design assumes the deployment of package relay and ephemeral anchors for dynamic fee management, but allows for different fee management approaches should they come. It allows batching vault operations, partial unvaultings, dynamic withdrawal targets, and recursive deposits.

Introduction

Custodying bitcoin is notoriously difficult. It is roughly equivalent to another task famous for its trail of bodies: keeping sensitive data accessible but out of unauthorized reach. Luckily, due to the programmability of Bitcoin script, custodians do not necessarily have to rely solely on the failure-prone task of key management.

In Bitcoin, a *covenant* is a constraint put on how a coin can be spent on the basis of its spending transaction, above and beyond the traditional requirements of satisfying a one-time unlocking script. Covenants can be written in such a way that they express recursive constraints on outputs, so that the effective period of covenant enforcement can span an arbitrary number of transactions.

Currently there is no way to enforce covenants "on-chain," or endogenously within bitcoin's validation rules, but there are many pending proposals for how to modify Bitcoin's validation rules to allow for this functionality. Covenant proposals can be divided into two kinds:

- **precomputed**: where the state machine of possible transactions within the lifecycle of a covenant is predetermined and therefore bound (e.g. `OP_CTV` [Rub20b], `ANYPREVOUT` [Chr18], `OP_TX` [Rus22]), and
- **general**: where the state machine is expressed within Bitcoin script, and can reapply itself indefinitely, making it potentially unbounded (e.g. transaction introspection opcodes and `OP_CHECKSIGFROMSTACK` [Kan21]).

Vaults are an especially useful kind of covenant that give Bitcoin users operational simplicity during expected use, but heightened security in the event of unauthorized access to private keys. When they were originally presented in [MS16], a vault was defined as a simple covenant that ensured that the spend of a coin was only allowed after broadcasting an intent to unvault and waiting some period. During this delay, the funds could be "clawed back" into a prespecified recovery path in case the proposed spend was unexpected.

In order to make use of this covenant, a user would need to send their coins into a vault, then configure a watchtower process to monitor the chain. If the watchtower detected an unexpected unvaulting, it could automatically broadcast a transaction that sweeps the funds to safety.

Since then, more sophisticated vault designs have been proposed which allow more detailed constraints like the enforcement of multiple discrete spending policies. But in the opinion of the authors, these advanced features remain niche and only benefit sophisticated industrial users. The main benefit of vaults likely remains the enforcement of a "withdrawal confirmation period" that allows the owner of a vault to intervene in a predetermined way to avoid theft.

Precomputed vaults

Given Bitcoin's existing validation rules, the only current, viable means of implementing a vault that is enforced directly by the validation rules[1] is by presigning a limited graph of transactions, as proposed in [Bis19] and implemented in [Bis20]. A user must generate an ephemeral key, sign a transaction sending coins to an address controlled by that key, presign a tree of possible transactions using that key, and then delete the key.

This locks the coins into a predetermined flow, and technically satisfies the definition of a vault. A robust implementation of this approach is described in detail in [Swa+20], as well as many related considerations like running a watchtower.

Presigning transactions to construct vaults in this way does add the safety of a limited recovery window, but it has a number of drawbacks:

- Key deletion cannot be proved, so ensuring that the vault isn't backdoored is more difficult.
- The custodian must not lose the presigned transactions, since there is no other way of spending the bitcoin. The sensitive data that is necessary to store indefinitely grows linearly with the number of vaults created.
- The spend target for the vault is static, and presumably must correspond to some kind of hot (or "warm") wallet. Loss of control of that hot wallet necessitates sweeping all presigned transactions to the likely difficult-to-access recovery wallet.
- Arbitrary vault withdrawal amounts are not possible after the structure of the vault is locked in by presigning.
- Vault operations, namely recoveries and unvaultings, cannot be batched together. This is especially unfortunate because in the case of a key leak, it may be critical to sweep all vaults to the recovery path as soon as possible; otherwise the custodian may end up racing the attacker to spend out of the vault.

Precomputed vaults with covenants

Having some kind of general on-chain covenant mechanism improves the situation somewhat. Script functionality like `OP_CHECKTEMPLATEVERIFY` (CTV) [Rub20b] allows us to use a vault scheme very similar to presigned transaction vaults, but with the considerable benefit that we do not need to engineer and operate an ephemeral key signing and deletion ceremony, nor do we have to persist critical presigned transactions indefinitely.

Similar to the presigned transaction approach, the entire allowable transaction state machine needs to be generated ahead of time and committed to with an `OP_CHECKTEMPLATEVERIFY` hash. This is demonstrated in [OBe22].

[1] Note that [Rev] is a method of emulating vaults using large multi-sig quorums, but this comes with significant operational complexity.

This approach has the benefits of not requiring storage of anything aside from the vault parameters used to generate the CTV transaction graph, which is not particularly sensitive. It also doesn't require ephemeral keys due to the nature of CTV, which uses an on-chain commitment to enforce a precomputed covenant.

Unfortunately, this approach still has major limitations.

- By nature of being precomputed, the number of vault operations is limited and predefined, ruling out arbitrary partial unvaults and recursive re-vaults.
- The destinations are still fixed to a set of keys. Unvaulting must be done through a single predetermined path. Fee management keys must be precommitted to (more on fees later).
- Vault operations cannot be batched together; this turns out to be a big limitation when responding to attackers.

These caveats also apply to other, similar vault implementations using different precomputed-covenant approaches in which the allowed graph of transactions must be precomputed, e.g. `ANYPREVOUT` ([Poi22]) and the (proposed but not implemented) `OP_TX` ([Rus22]).

Recursive vaults with general covenants

If we introduce sufficiently powerful script functionality to allow arbitrary introspection of the spending transaction, we would have the ability to write any kind of covenant – including a fully-featured, recursive vault – without having to resort to presigned transactions, covenant emulation via multi-sig, or CTV-style precomputed transaction graphs.

Such a vault would be free from the limitations of the precomputed approaches described above, but at the cost of significant script complexity. Writing the necessary script for such a vault based on primitives described in e.g. [Kan21] would not only be very complex, but it would be very verbose on-chain. The `scriptPubKey` sizes would be quite large for what might be a very common operation.

There is also a significant chance that the Bitcoin community might not reach agreement on how to proceed towards a general covenant solution. Vaults would be considerably to useful all users of Bitcoin today, since all users are concerned with custody; gating this highly practical feature on the precondition of agreeing to and deploying a general covenant mechanism (which isn't even guaranteed to happen) seems suboptimal.

`OP_VAULT`

Instead of resigning ourselves to the limitations of precomputed vaults, or waiting for a general-covenant mechanism to be deployed to script (a daunting prospect), we propose to evaluate the the creation of opcodes `OP_VAULT` and `OP_UNVAULT`, which have covenant-like characteristics but do not attempt to address the general problem of covenants.

This approach has a complete set of desirable features for safer custodial operations, none of the limitations of precomputed vaults, and is more concise and usable than a vault implemented with more general covenant scripting primitives.

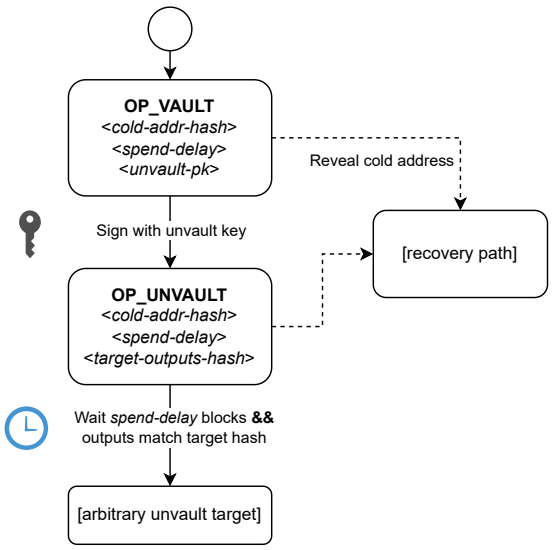

Figure 1: *A high-level description of the* `OP_VAULT` *state machine.*

Components of the vault

Each `OP_VAULT`-style vault makes use of a few pieces of essential data. These include

- **a recovery path**: the destination that vault funds can be swept to at any point prior to the finalization of withdrawal to the *unvault target*.

  This recovery address would usually correspond to a spending script that is inconvenient to exercise but is highly secure, e.g. a key generated offline or a geographically distributed multisig. It could be a P2TR address that incorporates a number of different spending conditions.

  Vaults which share the same recovery path can be swept in batch operations, which is an important practical aspect of managing large numbers of vaults.

- **an unvault key**: used to authorize beginning an unvault process, i.e. the spending of an `OP_VAULT` into a suitable `OP_UNVAULT`, which "announces" the intent to unvault and begins the withdrawal lock-in period. If an attacker obtains access to this key, the outcome is not catastrophic because any unvault can be interrupted and swept to the recovery address.

  Vaults which have this in common can unvaulted in batch.

- **an unvault target**: an arbitrary target or destination that is specified as a parameter to `OP_UNVAULT`, and dictates where unvaulted funds go. This target consists of a list of destination outputs (including amounts) which the vault will be spent into after the delay period. These destinations can of course include recursive `OP_VAULT` outputs to facilitate partial-amount unvaults.

Initial vaulting

To create a vault, a user would spend to an output with the following `scriptPubKey` (or equivalent taproot structure):

```
OP_VAULT <recovery-hash> <delay-period> <unvault-pk-hash>
```

where

- `<recovery-hash>` is the SHA256 hash of the recovery `scriptPubKey`. This could e.g. correspond to a P2TR address with many spending conditions.
- `<delay-period>` is the number of blocks between when the `OP_UNVAULT` confirms and when the funds can be spent to the unvault target outputs.
- `<unvault-pk-hash>` is the SHA256 hash of the public key whose signature is required to start the unvault process.

This output can only be spent one of two ways:

1. it can be swept to the recovery address at any point with no witness data, or
2. it can be sent to an `OP_UNVAULT` output matching the specification described below.

Triggering an unvault

To withdraw bitcoin from an `OP_VAULT`-encumbered output to some arbitrary destination, it must first be spent into an `OP_UNVAULT` output.

Witness requirements

A witness must be provided for the `OP_VAULT` input being spent which contains

1. a witness program (`scriptPubKey`) that, when hashed with `sha256d`, yields the `<unvault-pk-hash>`, and
2. a witness stack satisfying that witness program.

This witness is crucial for preventing unauthorized unvault attempts.

Spending output requirements (covenant)

The `OP_UNVAULT` `scriptPubKey` must have the form

```
OP_UNVAULT <recovery-hash> <delay-period> <unvault-target-hash>
```

where the first two parameters are carried over from the input `OP_VAULT` (described above), and *unvault-target-hash* is defined as

```
sha256d(sha256d(d.scriptPubKey) || sha256d(d.nValue)
        for d in target_outputs)
```

In other words, the target hash commits to the set of outputs that are proposing to be withdrawn to.

Withdrawal to the unvault target

The unvaulting party must now wait a total of *delay-period* blocks before being able to broadcast a transaction spending the `OP_UNVAULT` output into a set of outputs satisfying *unvault-target-hash*. No witness is needed for this finalizing "withdrawal" transaction, which represents the completion of the vault lifecycle for all amounts which are not revaulted.

If the owner of the vault does not recognize this proposed withdrawal, he can sweep the vault into the recovery path before the delay period ends.

Sweeping to recovery

In order to sweep an `OP_VAULT` or `OP_UNVAULT` output to the recovery address, a transaction is broadcast spending that output into a new output corresponding to the recovery address, consuming the full amount of the output being swept. This can happen at any point before withdrawal to the unvault target is finalized.

The witness for the swept output is empty, since the only authorization necessary is simply that the `scriptPubKey` being spent to matches the `<recovery-hash>` committed to in the parent `OP_VAULT` output.

Denial-of-service protection

During the creation of `OP_VAULT` and `OP_UNVAULT` outputs, the recovery address remains hidden behind a hash. This avoids denial-of-service (DoS) attacks that involve a third party attempting to blindly sweep vault transactions to their recovery path, which could result in a temporary halt in liquidity while the vault owner goes through the potentially lengthy process of activating the recovery keys for funds retrieval.

If it becomes necessary to make use of the recovery path, the recovery `scriptPubKey` will be revealed, which means that any other vaults with that recovery path may be swept there by an unauthenticated party.

This seems like a reasonable trade-off, since presumably if the vault's owner had to make use of the recovery path for one vault, any other vault sharing the recovery path may be at risk due to a compromise in the owner's infrastructure.

Batching

If multiple vaults share a common *recovery-hash*, they can be swept in batch. This is an especially important feature for users that maintain many vaults, since a key compromise might necessitate rapidly sweeping of many vaults at once before a delay period has elapsed. Batching the sweep reduces the chainspace required significantly, because we're able to sweep to a total of two outputs instead of creating $2v$ [2] outputs for v vaults.

Note that this batching option is not possible in vault schemes that rely on precomputed covenant techniques.

[2] An anchor output for fee control is needed for each sweep transaction

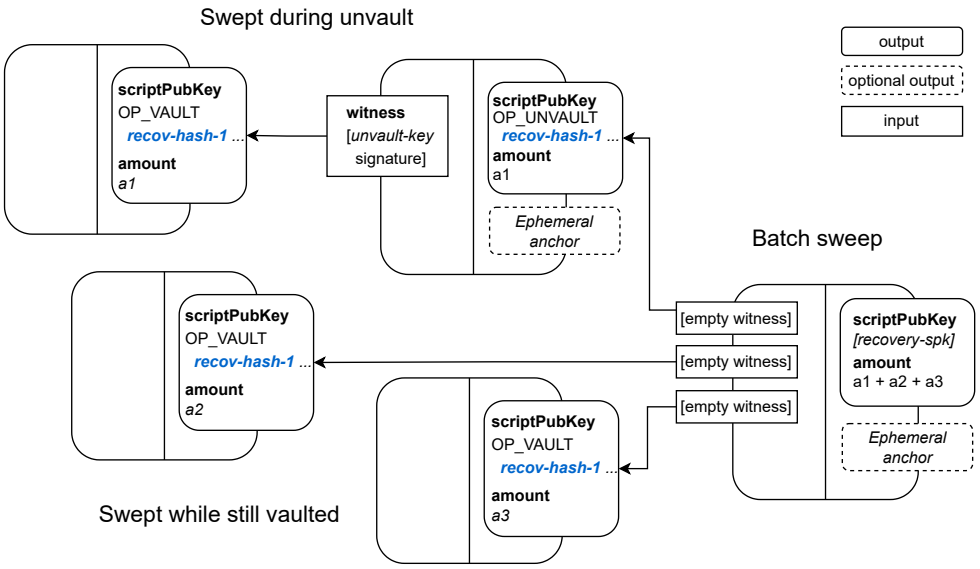

Figure 2: *Demonstration of multiple vaults being batch-swept simultaneously. Two of the vaults (bottom) are still vaulted while one (top) is interrupting an attempted unvault.*

Managing fees safely

One difficulty when designing this scheme was determining how to adjust the fee rate of transactions that happen sometime after vault creation. The fee environment may differ considerably from when the vault was initially created to when it will be accessed; fees may be much higher.

The most common method of increasing fee rate is to lower output amounts, which leaves more bitcoin available to miners and thus adds to the incentive to include a given transaction. During an `OP_VAULT` lifecycle, allowing variable amounts would unfortunately present some problems.

If an attacker discovers the recovery preimage, they may trigger undesired recovery sweeps. If the `OP_VAULT` validation rules allowed output amounts to vary for the recovery transaction, an attacker could simply set that value to 0 and burn the vaulted coins. A similar consideration must be applied to `OP_UNVAULT` transactions.

In order to avoid such a failure, there are two options:

1. script validation rules could require some allowable "range" of amount discounts during unvault/recovery to facilitate fee payment, or
2. script vaildation rules could require that the unvault/recovery outputs preserve the full value of any vaults being acted on.

1. seems like a bad design. *2.* seems like the preferable approach, but it introduces a complication: how are we to pay fees for sweep and unvault transactions if the full value of the vaulted coins must be preserved?

Saved by package relay and ephemeral anchors

Fee management is a thorny part of designing any contracting protocol built atop Bitcoin script. Transactions which are presigned (or covenant structures that fix amounts) cannot vary their amounts to adjust fees dynamically. A common workaround is to attach an "anchor output" [Vara], which is an output that is intended to be spent by some child transaction. Spending the child can raise the effective feerate of its parent, the precommitted contract transaction. This is referred to as child-pays-for-parent (CPFP).

This technique alone is insufficient for reliable fee control. If fees have risen considerably since the initial contract negotiation and the feerate of a precommitted transaction is beneath the minimum mempool feerate, it may be unable to broadcast in the first place and CPFP cannot take place.

Luckily, it is looking increasingly likely that package relay policies ([Varb]) that remedy this will become part of bitcoin. The `OP_VAULT` proposal detailed here not only assumes the presence of package relay, but also makes use of *ephemeral anchors* ([San22]), zero-value outputs which must be spent within the package they're relayed with, in order to avoid unnecessarily wasting vaulted value for fee control.

Our rationale for the permissibility of this hard dependency is that many – perhaps all – contracting schemes built on Bitcoin become unworkable without a robust solution for dynamic fee management, which itself appears to depend on package relay. We fully expect that if `OP_VAULT` were found to be a desirable soft-fork, its deployment would naturally come after the deployment of package relay and some kind of fee control mechanism like ephemeral anchors.

Given the deploy of these modern facilities, the `OP_VAULT` strategy is able to both provide efficient fee management and rule out coin-burning denial-of-service attacks.

Future-proofing fee control

To avoid wasting value and to allow dynamic fee control, each `OP_UNVAULT` transaction, recovery sweep transaction, and unvault target transaction *may* have an ephemeral anchor output that facilitates robust fee management with CPFP and package relay.

But there are other possible designs for dynamic fee management. One is transaction sponsors [Rub20a]. Sponsors allow fee rate improvement of any unconfirmed transaction by any party, and they do not require any structural "awareness" from the transaction whose fee is being bumped. In the proposed design, sponsors would be required by consensus to be mined alongside other transactions which they are sponsoring. This allows CPFP-like adjustment of fee rate, but without a wasted output or specific planning.

In order to allow for the possible adoption of such a proposal, any ephemeral anchor outputs in the `OP_VAULT` scheme are strictly optional. To avoid pinning by an adversary who discovers the recovery path, any transaction for which the ephemeral anchor is optional must be marked for replacement by fee (RBF) (by policy, as a part of this proposal). This ensures that the vault owner can override spurious sweep transactions that exclude the anchor with higher-fee packages making use of the anchor.

The importance of dynamic unvault targets

In vault schemes which are usable today, the unvault target is always fixed at vault creation time. Even in schemes that assume the use of a precomputed covenant mechanism like `CHECKTEMPLATEVERIFY` or `ANYPREVOUT`, the set of withdrawal targets is similarly fixed.

The design presented here allows for the withdrawal target to be decided at unvault time, before the delay period, rather than needing to be specified during the creation of the vault. This presents a significant benefit, because if the withdrawal target is fixed, an intermediary hot or warm wallet must be used to unvault and then send the coins to their ultimate destination.

If a withdrawal target can be set at unvault time, no such intermediary wallet is needed. This saves on operational complexity, avoids a potential avenue of compromise, and also removes the need for an additional transaction – which may be no small consideration when considering possible future demand for chainspace.

Batching

Dynamic unvault targets also allow for output amount adjustment that enables multiple vaults to be unvaulted (or recovered) to a single set of output targets. This isn't only important as blockspace becomes scarcer. After ruminating for some time on the failure modes associated with vaults, the authors have become convinced that the ability to recover or unvault as efficiently as possible is crucial to thwarting an attacker.

Recursive deposits (revaults)

Another benefit of supporting dynamic unvault targets is that an arbitrary number of partial unvaults can happen by withdrawing to some output outside the vault, but then redepositing the remaining balance back into the same `OP_VAULT [...]` construction.

This could enable, for example, an exchange to offer a vault address to a customer, allow them to make multiple deposits, and support partial withdrawals without having to redo a vault initialization.

Downsides relative to other designs

We have already written about the important benefits that this scheme provides relative to alternatives. While we think the set of trade-offs for `OP_VAULT` are appealing, it is important to acknowledge downsides.

A scheme that does not codify vault semantics at the script interpreter layer, e.g. one that uses only a general covenant primitive like transaction introspection within script, allows for more flexibility in the particular control flow of the vault, since the rules of the vault are specified by the end-user instead of the consensus engine. It more easily facilitates user experimentation.

For example, perhaps a user prefers to safeguard their recovery path with a signature rather than revealing the preimage to a witness (program) hash, so that discovery of the recovery path doesn't allow an unauthenticated user to sweep open vaults to recovery.

The downside, of course, is that this results in very large on-chain script sizes. In order to make vault constructions concise, they ultimately must be codified in validation rules, barring some breakthrough in a zero-knowledge proof system that could be supported on-chain.

Conclusion

We have presented a new set of opcodes, `OP_VAULT` and `OP_UNVAULT`, which enable featureful vaults in Bitcoin. These opcodes allow encumbering a set of coins in such a way that their spending requires passing through a delay period, during which the coins can be recovered to a set address. Enabling on-chain enforcement of such a control flow presents significant benefits to custodians of bitcoin, whether large or small, because in essence this scheme offers multisig-like security for an expected operational burden roughly on par with single sig use.

The `OP_VAULT` proposal is unique in enabling robust dynamic fee management (thanks to package relay and ephmeral anchors). It also enables batch management, a critical feature when considering the need to respond to attackers efficiently and make judicious use of chain space.

References

[Bis19] Bryan Bishop. *Bitcoin vaults with anti-theft recovery/clawback mechanisms*. 2019. URL: https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2019-August/017229.html.

[Bis20] Bryan Bishop. *python-vaults*. 2020. URL: https://github.com/kanzure/python-vaults.

[Chr18] Anthony Towns Christian Decker. *BIP-118: SIGHASHANYPREVOUT for Taproot Scripts*. 2018. URL: https://github.com/bitcoin/bips/blob/master/bip-0118.mediawiki.

[Kan21] Sanket Kanjalkar. *Elements Taproot Introspection Opcodes*. 2021. URL: https://github.com/ElementsProject/elements/blob/master/doc/tapscript_opcodes.md.

[MS16] Ittay Eyal Malte Möser and Emin Gün Sirer. *Bitcoin Covenants*. 2016.

[OBe22] James O'Beirne. *simple-ctv-vault*. 2022. URL: https://github.com/jamesob/simple-ctv-vault.

[Poi22] Antoine Poinsot. *simple-anyprevout-vault*. 2022. URL: https://github.com/darosior/simple-anyprevout-vault.

[Rev] Revault. *Revault*. URL: https://revault.dev.

[Rub20a] Jeremy Rubin. *A Replacement for RBF and CPFP: Non-Destructive TXID Dependencies for Fee Sponsoring*. 2020. URL: https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2020-September/018168.html.

[Rub20b] Jeremy Rubin. *BIP 119: CHECKTEMPLATEVERIFY*. 2020. URL: https://github.com/bitcoin/bips/blob/master/bip-0119.mediawiki.

[Rus22] Rusty Russell. *OP_TX: generalized covenants reduced to OPCHECKTEMPLATEVERIFY*. 2022. URL: https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2022-May/020450.html.

[San22] Gregory Sanders. *Ephemeral Anchors: Fixing V3 Package RBF against package limit pinning*. 2022. URL: https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2022-October/021036.html.

[Swa+20] Jacob Swambo et al. *Custody Protocols Using Bitcoin Vaults*. 2020. URL: https://arxiv.org/pdf/2005.11776.

[Vara] Various. *Bitcoin Optech: Anchor Outputs*. URL: https://bitcoinops.org/en/topics/anchor-outputs/.

[Varb] Various. *Bitcoin Optech: Package Relay*. URL: https://bitcoinops.org/en/topics/package-relay/.