Onboarding to Bitcoin Core

Table of Contents

Document originally written by Will Clark. Help maintain this document: https://github.com/chaincodelabs/onboarding-to-bitcoin-core

This section has been updated to Bitcoin Core @ v23.0

Contributor journeys

Some Contributors have documented their journeys into the space which lets us learn about approaches they found useful, and also any pitfalls and things they found difficult along the way.

- Amiti Uttarwar - Onboarding to Bitcoin Core
- Jon Atack - On Reviewing, and Helping Those Who Do It
- Jimmy Song - A Gentle Introduction to Bitcoin Core Development

Decentralized development

Olivia Lovenmark and Amiti Uttarwar describe in their blog post "Developing Bitcoin", how changes to bitcoin follow the pathway from proposal to being merged into the software, and finally into voluntary adoption by users choosing to use the software.

Developer guidelines

The Bitcoin Core project itself contains three documents of particular interest to Contributors:

1. CONTRIBUTING.md — How to get started contributing to the project. (Forking, creating branches, commit patches)
2. developer-notes.md — Development guidelines, coding style etc.
3. productivity.md — Many tips for improving developer productivity (ccache, reviewing code, refspecs, git diffs)
4. test/README.md — Guidance on running the test suite

Using `ccache` as described in *productivity.md* above will speed up builds of Bitcoin Core dramatically.

Setting up a ramdisk for the test suite as described in *test/README.md* will speed

> up running the test suite dramatically.

Development workflow

Bitcoin Core uses a GitHub-based workflow for development. The primary function of GitHub in the workflow is to discuss patches and connect them with review comments.

While some other prominent projects, e.g. the Linux kernel, use email to solicit feedback and review, Bitcoin Core has used GitHub for many years. Initially, Satoshi distributed the code through private emails and hosting source archives at bitcoin.org, and later by hosting on SourceForge (which used SVN but did not at that time have a pull request system like GitHub). The earliest reviewers submitted changes using patches either through email exchange with Satoshi, or by posting them on the bitcoin forum.

In August 2009, the source code was moved to GitHub by Sirius, and development has remained there and used the GitHub workflows ever since.

Use of GitHub

The GitHub side of the Bitcoin Core workflow for Contributors consists primarily of:

- Issues
- PRs
- Reviews
- Comments

Generally, issues are used for two purposes:

1. Posting known issues with the software, e.g., bug reports, crash logs
2. Soliciting feedback on potential changes without providing associated code, as would be required in a PR.

GitHub provides their own guide on mastering Issues which is worth reading to understand the feature-set available when working with an issue.

PRs are where Contributors can submit their code against the main codebase and solicit feedback on the concept, the approach taken for the implementation, and the actual implementation itself.

PRs and Issues are often linked to/from one another:

> One common workflow is when an Issue is opened to report a bug. After replicating the issue, a Contributor creates a patch and then opens a PR with their proposed changes.
>
> In this case, the Contributor should, in addition to comments about the patch, reference that the patch fixes the issue. For a patch which fixes issue 22889 this would be done by writing "fixes #22889" in the PR description or in a commit message. In this case, the syntax "fixes #issue-number" is caught by GitHub's pull request linker, which handles the cross-link

automatically.

Another use-case of Issues is soliciting feedback on ideas that might require *significant* changes. This helps free the project from having too many PRs open which aren't ready for review and might waste reviewers' time. In addition, this workflow can also save Contributors their *own* valuable time, as an idea might be identified as unlikely to be accepted *before* the contributor spends their time writing the code for it.

Most code changes to bitcoin are proposed directly as PRs — there's no need to open an Issue for every idea before implementing it unless it may require significant changes. Additionally, other Contributors (and would-be Reviewers) will often agree with the approach of a change, but want to "see the implementation" before they can really pass judgement on it.

GitHub is therefore used to help store and track reviews to PRs in a public way.

Comments (inside Issues, PRs, Projects etc.) are where all (GitHub) users can discuss relevant aspects of the item and have history of those discussions preserved for future reference. Often Contributors having "informal" discussions about changes on e.g. IRC will be advised that they should echo the gist of their conversation as a comment on GitHub, so that the rationale behind changes can be more easily determined in the future.

Reviewing code

Jon Atack provides a guide to reviewing a Bitcoin Core PR in his article How To Review Pull Requests in Bitcoin Core.

Gloria Zhao's review checklist details what a "good" review might look like, along with some examples of what she personally considers good reviews. In addition to this, it details how potential Reviewers can approach a new PR they have chosen to review, along with the sorts of questions they should be asking (and answering) in order to provide a meaningful review. Some examples of the subject areas Gloria covers include the PR's subject area, motivation, downsides, approach, security and privacy risks, implementation of the idea, performance impact, concurrency footguns, tests and documentation needed.

Contributing code

This section details some of the processes surrounding code contributions to the Bitcoin Core project along with some common pitfalls and tips to try and avoid them.

Branches

You should **not** use the built-in GitHub branch creation process, as this interferes with and confuses the Bitcoin Core git process.

Instead, you should use either the native `git` or the GitHub `gh` cli (requires `git`) tools to create your own branches locally, before pushing them to your fork of the repo, and opening a PR against the Bitcoin Core repo from there.

Creating a PR

Jon Atack's article How To Contribute Pull Requests To Bitcoin Core describes some less-obvious requirements that any PR you make might be subjected to during peer review, for example that it needs an accompanying test, or that an intermediate commit on the branch doesn't compile. It also describes the uncodified expectation that Contributors should not only be writing code, but perhaps more importantly be providing reviews on other Contributors' PRs. Most developers enjoy writing their own code more than reviewing code from others, but the decentralized review process is arguably the most critical defence Bitcoin development has against malicious actors and therefore important to try and uphold.

> Jon's estimates of "5-15 PR reviews|issues solved" per PR submitted is not a hard requirement, just what Jon himself feels would be best for the project. Don't be put off submitting a potentially valuable PR just because "you haven't done enough reviews"!

For some tips on how to maintain an open PR using git, such as how to redo commit history, as well as edit specific commits, check out this guide.

Commit messages

When writing commit messages be sure to have read Chris Beams' "How to Write a Git Commit Message" blog post. As described in CONTRIBUTING.md, PRs should be prefixed with the component or area the PR affects. Common areas are listed in CONTRIBUTING.md section: Creating the pull request. Individual commit messages are also often given similar prefixes in the commit title depending on which area of the codebase the changes primarily affect.

Continuous integration

When PRs are submitted against the primary Bitcoin Core repo a series of CI tests will automatically be run. These include a series of linters and formatters such as `clang-format`, `flake8` and `shellcheck`. It's possible (and advised) to run these checks locally against any changes you make before you push them.

In order to run the lints yourself you'll have to first make sure your python environment and system have the packages listed in the CI install script. You can then run a decent sub-set of the checks by running:

```
python test/lint/lint-circular-dependencies.py

# requires requires 'flake8', 'mypy', 'pyzmq', 'codespell', 'vulture'
python test/lint/lint-python.py

python test/lint/lint-whitespace.py
```

Or you can run all checks with:

```
python test/lint/all-lint.py
```

Previously these checks were shell scripts (`*.sh`), but they have now been migrated to python on master.

+ If you are following with tag v23.0 these may still exist as `*.sh`.

Linting your changes reduces the chances of pushing them as a PR and then having them quickly being marked as failing CI. The GitHub PR page auto-updates the CI status.

If you do fail a lint or any other CI check, force-pushing the fix to your branch will cancel the currently-running CI checks and restart them.

Build issues

Some compile-time issues can be caused by an unclean build directory. The comments in issue 19330 provide some clarifications and tips on how other Contributors clean their directories, as well as some ideas for shell aliases to boost productivity.

Debugging Bitcoin Core

Fabian Jahr has created a guide on "Debugging Bitcoin Core", aimed at detailing the ways in which various Bitcoin Core components can be debugged, including the Bitcoin Core binary itself, unit tests, functional tests along with an introduction to core dumps and the Valgrind memory leak detection suite.

Of particular note to Developers are the configure flags used to build Bitcoin Core without optimisations to permit more effective debugging of the various resulting binary files.

Fabian has also presented on this topic a number of times. A transcript of his edgedevplusplus talk is available.

Codebase archaeology

When considering changing code it can be helpful to try and first understand the rationale behind why it was implemented that way originally. One of the best ways to do this is by using a combination of git tools:

- `git blame`
- `git log -S`
- `git log -G`
- `git log -p`
- `git log -L`

As well as the discussions in various places on the GitHub repo.

git blame

The git `blame` command will show you when (and by who) a particular line of code was last *changed*.

For example, if we checkout Bitcoin Core at v22.0 and we are planning to make a change related to the `m_addr_send_times_mutex` found in *src/net_processing.cpp*, we might want to find out more about its history before touching it.

With `git 'blame` we can find out the last person who touched this code:

```
# Find the line number for blame
$ grep -n m_addr_send_times_mutex src/net_processing.cpp
233:    mutable Mutex m_addr_send_times_mutex;
235:    std::chrono::microseconds m_next_addr_send GUARDED_BY(
m_addr_send_times_mutex){0};
237:    std::chrono::microseconds m_next_local_addr_send GUARDED_BY
(m_addr_send_times_mutex){0};
4304:    LOCK(peer.m_addr_send_times_mutex);
```

```
$ git blame -L233,233 src/net_processing.cpp

76568a3351 (John Newbery 2020-07-10 16:29:57 +0100 233)     mutable Mutex
m_addr_send_times_mutex;
```

With this information we can easily look up that commit to gain some additional context:

```
$ git show 76568a3351

_____________________________________________________________________________

commit 76568a3351418c878d30ba0373cf76988f93f90e
Author: John Newbery <john@johnnewbery.com>
Date:   Fri Jul 10 16:29:57 2020 +0100

    [net processing] Move addr relay data and logic into net processing
```

So we've learned now that this mutex was moved here by John from *net.{cpp|h}* in it's most recent touch. Let's see what else we can find out about it.

git log -S

`git log -S` allows us to search for commits where this line was *modified* (not where it was only moved, for that use `git log -G`).

A 'modification' (vs. a 'move') in git parlance is the result of uneven instances of the search term in the commit diffs' add/remove sections.

This implies that this term has either been added or removed in the commit.

```
$ git log -S m_addr_send_times_mutex
```

```
commit 76568a3351418c878d30ba0373cf76988f93f90e
Author: John Newbery <john@johnnewbery.com>
Date:   Fri Jul 10 16:29:57 2020 +0100

    [net processing] Move addr relay data and logic into net processing

```

```
commit ad719297f2ecdd2394eff668b3be7070bc9cb3e2
Author: John Newbery <john@johnnewbery.com>
Date:   Thu Jul 9 10:51:20 2020 +0100

    [net processing] Extract `addr` send functionality into MaybeSendAddr()

    Reviewer hint: review with

     `git diff --color-moved=dimmed-zebra --ignore-all-space`

```

```
commit 4ad4abcf07efefafd439b28679dff8d6bbf62943
Author: John Newbery <john@johnnewbery.com>
Date:   Mon Mar 29 11:36:19 2021 +0100

    [net] Change addr send times fields to be guarded by new mutex
```

We learn now that John also originally added this to *net.{cpp|h}*, before later moving it into *net_processing.{cpp|h}* as part of a push to separate out `addr` relay data and logic from *net.cpp*.

git log -p

`git log -p` (usually also given with a file name argument) follows each commit message with a *patch* (diff) of the changes made by that commit to that file (or files). This is similar to `git blame` except that `git blame` shows the source of only lines *currently* in the file.

git log -L

The `-L` parameter provided to git log will allow you to trace certain lines of a file through a range given by `<start,<end>`.

However, newer versions of git will also allow you to provide `git log -L` with a function name and a file, using:

```
git log -L :<funcname>:<file>
```

This will then display commits which modified this function in your pager.

git log --follow *file...*

One of the most famous file renames was *src/main.{h,cpp}* to *src/validation.{h,cpp}* in 2016. If you simply run `git log src/validation.h`, the oldest displayed commit is one that implemented the rename. `git log --follow src/validation.h` will show the same recent commits followed by the older *src/main.h* commits.

To see the history of a file that's been removed, specify " — " before the file name, such as:

```
git log -- some_removed_file.cpp
```

PR discussion

To get even more context on the change we can leverage GitHub and take a look at the comments on the PR where this mutex was introduced (or at any subsequent commit where it was modified). To find the PR you can either paste the commit hash (`4ad4abcf07efefafd439b28679dff8d6bbf62943`) into GitHub, or list merge commits in reverse order, showing oldest merge with the commit at the top to show the specific PR number e.g.:

```
$ git log --merges --reverse --oneline --ancestry-path
4ad4abcf07efefafd439b28679dff8d6bbf62943..upstream | head -n 1

d3fa42c79 Merge bitcoin/bitcoin#21186: net/net processing: Move addr data into
net_processing
```

Reading up on PR#21186 will hopefully provide us with more context we can use.

> We can see from the linked issue 19398 what the motivation for this move was.

Building from source

When building Bitcoin Core from source, there are some platform-dependant instructions to follow.

To learn how to build for your platform, visit the Bitcoin Core bitcoin/doc directory, and read the file named "build-*.md", where "*" is the name of your platform. For windows this is "build-windows.md", for macOS this is "build-osx.md" and for most linux distributions this is "build-unix.md".

There is also a guide by Jon Atack on how to compile and test Bitcoin Core.

Finally, Blockchain Commons also offer a guide to building from source.

Cleaner builds

It can be helpful to use a separate build directory e.g. `build/` when compiling from source. This can help avoid spurious Linker errors without requiring you to run `make clean` often.

From within your Bitcoin Core source directory you can run:

```
# Clean current source dir in case it was already configured
make distclean

# Make new build dir
mkdir build && cd build

# Run normal build sequence with amended path
../autogen.sh
../configure --your-normal-options-here
make -j `nproc`
make check
```

To run individual functional tests using the bitcoind binary built in an out-of-source build change directory back to the root source and specify the *config.ini* file from within the build directory:

```
$ pwd
/path/to/source/build
$ cd ..
$ test/functional/p2p_ping.py --configfile build/test/config.ini
```

Codebase documentation

Bitcoin Core uses Doxygen to generate developer documentation automatically from its annotated C++ codebase. Developers can access documentation of the current release of Bitcoin Core online at doxygen.bitcoincore.org, or alternatively can generate documentation for their current git `HEAD` using `make docs` (see Generating Documentation for more info).

Testing

Three types of test network are available:

1. Testnet
2. Regtest
3. Signet

These three networks all use coins of zero value, so can be used experimentally.

They primary differences between the networks are as follows:

Table 1. Comparison of different test networks

Feature	Testnet	Regtest	Signet
Mining algorithm	Public hashing with difficulty	Local hashing, low difficulty	Signature from authorized signers
Block production schedule	Varies per hashrate	On-demand	Reliable intervals (default 2.5 mins)
P2P port	18333	18444	38333
RPC port	18332	18443	38332
Peers	Public	None	Public
Topology	Organic	Manual	Organic
Chain birthday	2011-02-02	At time of use	2020-09-01
Can initiate re-orgs	If Miner	Yes	No
Primary use	Networked testing	Automated integration tests	Networked testing

Signet

Signet is both a tool that allows Developers to create their own networks for testing interactions between different Bitcoin software, and the name of the most popular of these public testing networks. Signet was codified in BIP 325.

To connect to the "main" Signet network, simply start bitcoind with the signet option, e.g. `bitcoind -signet`. Don't forget to also pass the signet option to `bitcoin-cli` if using it to control bitcoind, e.g. `bitcoin-cli -signet your_command_here`. Instructions on how to setup your own Signet network can be found in the Bitcoin Core Signet README.md. The Bitcoin wiki Signet page provides additional background on Signet.

Regtest

Another test network named *regtest*, which stands for *regression test*, is also available. This network is enabled by starting bitcoind with the `-regtest` option. This is an entirely self-contained mode, giving you complete control of the state of the blockchain. Blocks can simply be mined on command by the network operator.

The functional tests use this mode, but you can also run it manually. It provides a good means to learn and experiment on your own terms. It's often run with a single node but may be run with multiple co-located (local) nodes (most of the functional tests do this). The blockchain initially contains only the genesis block, so you need to mine >100 blocks in order to have any spendable coins from a mature coinbase. Here's an example session (after you've built `bitcoind` and `bitcoin-cli`):

```
$ mkdir -p /tmp/regtest-datadir
$ src/bitcoind -regtest -datadir=/tmp/regtest-datadir
$ src/bitcoin-cli -regtest -datadir=/tmp/regtest-datadir getblockchaininfo
{
  "chain": "regtest",
```

```
  "blocks": 0,
  "headers": 0,
  "bestblockhash": "0f9188f13cb7b2c71f2a335e3a4fc328bf5beb436012afca590b1a11466e2206",
  _(...)_
}
$ src/bitcoin-cli -regtest -datadir=/tmp/regtest-datadir createwallet testwallet
$ src/bitcoin-cli -regtest -datadir=/tmp/regtest-datadir -generate 3
{
  "address": "bcrt1qpw3pjhtf9myl0qk9cxt54qt8qxu2mj955c7esx",
  "blocks": [
    "6b121b0c094b5e107509632e8acade3f6c8c2f837dc13c72153e7fa555a29984",
    "5da0c549c3fddf2959d38da20789f31fa7642c3959a559086436031ee7d7ba54",
    "3210f3a12c25ea3d8ab38c0c4c4e0d5664308b62af1a771dfe591324452c7aa9"
  ]
}
$ src/bitcoin-cli -regtest -datadir=/tmp/regtest-datadir getblockchaininfo
{
  "chain": "regtest",
  "blocks": 3,
  "headers": 3,
  "bestblockhash": "3210f3a12c25ea3d8ab38c0c4c4e0d5664308b62af1a771dfe591324452c7aa9",
  _(...)_
}
$ src/bitcoin-cli -regtest -datadir=/tmp/regtest-datadir getbalances
{
  "mine": {
    "trusted": 0.00000000,
    "untrusted_pending": 0.00000000,
    "immature": 150.00000000
  }
}
$ src/bitcoin-cli -regtest -datadir=/tmp/regtest-datadir stop
```

You may stop and restart the node and it will use the existing state. (Simply remove the data directory to start again from scratch.)

Blockchain Commons offer a guide to Using Bitcoin Regtest.

Testnet

Testnet is a public bitcoin network where mining is performed in the usual way (hashing) by decentralized miners.

However, due to much lower hashrate (than mainnet), testnet is susceptible extreme levels of inter-block volatility due to the way the difficulty adjustment (DA) works: if a mainnet-scale miner wants to "test" their mining setup on testnet then they may cause the difficulty to increase greatly. Once the miner has concluded their tests they may remove all hashpower from the network at once. This can leave the network with a high difficulty which the DA will take a long time to compensate for.

Therefore a "20 minute" rule was introduced such that the difficulty would reduce to the minimum

for one block before returning to its previous value. This ensures that there are no intra-block times greater than 20 minutes.

However there is a bug in the implementation which means that if this adjustment occurs on a difficulty adjustment block the difficulty is lowered to the minimum for one block but then not reset, making it permanent rather than a one-off adjustment. This will result in a large number of blocks being found until the DA catches up to the level of hashpower on the network.

It's usually preferable to test private changes on a local regtest, or public changes on a Signet for this reason.

Manual testing while running a functional test

Running regtest as described allows you to start from scratch with an empty chain, empty wallet, and no existing state.

An effective way to use regtest is to start a functional test and insert a python debug breakpoint. You can set a breakpoint in a test by adding `import pdb; pdb.set_trace()` at the desired stopping point; when the script reaches this point you'll see the debugger's `(Pdb)` prompt, at which you can type `help` and see and do all kinds of useful things.

While the (Python) test is paused, you can still control the node using `bitcoin-cli`. First you need to look up the data directory for the node(s), as below:

```
$ ps alx | grep bitcoind
0  1000   57478   57476  20   0 1031376 58604 pipe_r SLl+ pts/10    0:06
/g/bitcoin/src/bitcoind -datadir=/tmp/bitcoin_func_test_ovsi15f9/node0 -logtimemicros
-debug (...)
0  1000   57479   57476  20   0 965964 58448 pipe_r SLl+ pts/10     0:06
/g/bitcoin/src/bitcoind -datadir=/tmp/bitcoin_func_test_ovsi15f9/node1 -logtimemicros
-debug (...)
```

With the `-datadir` path you can look at the `bitcoin.conf` files within the data directories to see what config options are being specified for the test (there's always `regtest=1`) in addition to the runtime options, which is a good way to learn about some advanced uses of regtest.

In addition to this, we can use the `-datadir=` option with `bitcoin-cli` to control specific nodes, e.g.:

```
$ src/bitcoin-cli -datadir=/tmp/bitcoin_func_test_ovsi15f9/node0 getblockchaininfo
```

Getting started with development

One of the roles most in-demand from the project is that of code review, and in fact this is also one of the best ways of getting familiarized with the codebase too! Reviewing a few PRs and adding your review comments to the PR on GitHub can be really valuable for you, the PR author and the bitcoin community. This Google Code Health blog post gives some good advice on how to go about code review and getting past "feeling that you're not as smart as the programmer who wrote the change". If you're going to ask some questions as part of review, try and keep questions respectful.

There is also a Bitcoin Core PR Review Club held weekly on IRC which provides an ideal entry point into the Bitcoin Core codebase. A PR is selected, questions on the PR are provided beforehand to be discussed on irc.libera.chat #bitcoin-core-pr-reviews IRC room and a host will lead discussion around the changes.

Aside from review, there are 3 main avenues which might lead you to submitting your **own** PR to the repository:

1. Finding a `good first issue`, as tagged in the issue tracker
2. Fixing a bug
3. Adding a new feature (that you want for yourself?)

Choosing a "good first issue" from an area of the codebase that seems interesting to you is often a good approach. This is because these issues have been somewhat implicitly "concept ACKed" by other Contributors as "something that is likely worth someone working on". Don't confuse this for meaning that if you work on it that it is certain to be merged though.

If you don't have a bug fix or new feature in mind and you're struggling to find a good first issue which looks suitable for you, don't panic. Instead keep reviewing other Contributors' PRs to continue improving your understanding of the process (and the codebase) while you watch the Issue tracker for something which you like the look of.

When you've decided what to work on it's time to take a look at the current behaviour of that part of the code and perhaps more importantly, try to understand *why* this was originally implemented in this way. This process of codebase "archaeology" will prove invaluable in the future when you are trying to learn about other parts of the codebase on your own.

#bitcoin-core-dev IRC channel

The Bitcoin Core project has an IRC channel `#bitcoin-core-dev` available on the Libera.chat network. If you are unfamiliar with IRC there is a short guide on how to use it with a client called Matrix here. IRC clients for all platforms and many terminals are available.

"Lurking" (watching but not talking) in the IRC channel can both be a great way to learn about new developments as well as observe how new technical changes and issues are described and thought about from other developers with an adversarial mindset. Once you are comfortable with the rules of the room and have questions about development then you can join in too!

This channel is reserved for discussion about *development of the Bitcoin Core software only*, so please don't ask general Bitcoin questions or talk about the price or other things which would be off topic in there.

There are plenty of other channels on IRC where those topics can be discussed.

There are also regular meetings held on #bitcoin-core-dev which are free and open for anyone to attend. Details and timings of the various meetings are found here.

Communication

In reality there are no hard rules on choosing a discussion forum, but in practice there are some common conventions which are generally followed:

- If you want to discuss the codebase of the Bitcoin Core implementation, then discussion on either the GitHub repo or IRC channel is usually most-appropriate.
- If you want to discuss changes to the core bitcoin protocol, then discussion on the mailing list is usually warranted to solicit feedback from (all) bitcoin developers, including the many of them that do not work on Bitcoin Core directly.
    - If mailing list discussions seem to indicate interest for a proposal, then creation of a BIP usually follows.

If discussing something Bitcoin Core-related, there can also be a question of whether it's best to open an Issue, to first discuss the problem and brainstorm possible solution approaches, or whether you should implement the changes as you see best first, open a PR, and then discuss changes in the PR. Again, there are no hard rules here, but general advice would be that for potentially-controversial subjects, it might be worth opening an Issue first, before (potentially) wasting time implementing a PR fix which is unlikely to be accepted.

Regarding communication timelines it is important to remember that many contributors are unpaid volunteers, and even if they are sponsored or paid directly, nobody owes you their time. That being said, often during back-and-forth communication you might want to nudge somebody for a response and it's important that you do this in a courteous way. There are again no hard rules here, but it's often good practice to give somebody on the order of a few days to a week to respond. Remember that people have personal lives and often jobs outside of Bitcoin development.

ACK / NACK

If you are communicating on an Issue or PR, you might be met with "ACK"s and "NACK"s (or even "approach (N)ACK" or similar). ACK, or "acknowledge" generally means that the commenter approves with what is being discussed previously. NACK means they tend to not approve.

What should you do if your PR is met with NACKs or a mixture of ACKs and NACKs? Again there are no hard rules but generally you should try to consider all feedback as constructive criticism. This can feel hard when veteran contributors appear to drop by and with a single "NACK" shoot down your idea, but in reality it presents a good moment to pause and reflect on *why* someone is not agreeing with the path forward you have presented.

Although there are again no hard "rules" or "measurement" systems regarding (N)ACKs, maintainers — who's job it is to measure whether a change has consensus before merging — will often use their discretion to attribute more weight behind the (N)ACKs of contributors that they feel have a good understanding of the codebase in this area.

This makes sense for many reasons, but lets imagine the extreme scenario where members of a Reddit/Twitter thread (or other group) all "brigade" a certain pull request on GitHub, filling it with tens or even hundreds of NACKs… In this scenario it makes sense for a maintainer to somewhat reduce the weighting of these NACKs vs the (N)ACKs of regular contributors:

We are not sure which members of this brigade:

- Know how to code and with what competency
- Are familiar with the Bitcoin Core codebase
- Understand the impact and repercussions of the change

Whereas we can be more sure that regular contributors **and** those respondents who are providing additional rationale in addition to their (N)ACK, have some understanding of this nature. Therefore it makes sense that we should weight regular contributors' responses, and those with additional compelling rationale, more heavily than GitHub accounts created yesterday which reply with a single word (N)ACK.

From this extreme example we can then use a sliding scale to the other extreme where, if a proven expert in this area is providing a lone (N)ACK to a change, that we should perhaps step back and consider this more carefully.

Does this mean that your views as a new contributor are likely to be ignored? **No**!! However it might mean that you might like to include rationale in any ACK/NACK comments you leave on PRs, to give more credence to your views.

When others are (N)ACK-ing your work, you should not feel discouraged because they have been around longer than you. If they have not left rationale for the comment, then you should ask them for it. If they have left rationale but you disagree, then you can politely state your reasons for disagreement.

In terms of choosing a tone, the best thing to do it to participate in PR review for a while and observe the tone used in public when discussing changes.

Backports

Bitcoin Core often backports fixes for bugs and soft fork activations into previous software releases.

Generally maintainers will handle backporting for you, unless for some reason the process will be too difficult. If this point is reached a decision will be made on whether the backport is abandoned, or a specific (new) fix is created for the older software version.

Software Life-cycle

An overview of the software life-cycle for Bitcoin Core can be found at https://bitcoincore.org/en/lifecycle/

Reproducible Guix builds

Bitcoin Core uses the Guix package manager to achieve reproducible builds. Carl Dong gave an introduction to GUIX via a remote talk in 2019, and also discussed it further on a ChainCode podcast episode.

There are official instructions on how to run a Guix build in the Bitcoin Core repo which you should

certainly follow for your first build, but once you have managed to set up the Guix environment (along with e.g. MacOS SDK), hebasto provides a more concise workflow for subsequent or repeated builds in his gist.

Organisation & roles

The objective of the Bitcoin Core Organisation is to represent an entity that is decentralized as much as practically possible on a centralised platform. One where no single Contributor, Member, or Maintainer has unilateral control over what is/isn't merged into the project. Having multiple Maintainers, Members, Contributors, and Reviewers gives this objective the best chance of being realised.

Contributors

Anyone who contributes code to the codebase is labelled a Contributor by GitHub and also by the community. As of Version 23.0 of Bitcoin Core, there are > 850 individual Contributors credited with changes.

Members

Some Contributors are also labelled as Members of the Bitcoin organisation. There are no defined criteria for becoming a Member of the organisation; persons are usually nominated for addition or removal by current Maintainer/Member/Admin on an ad-hoc basis. Members are typically frequent Contributors/Reviewers and have good technical knowledge of the codebase.

Some members also have some additional permissions over Contributors, such as adding/removing tags on issues and Pull Requests (PRs); however, being a Member **does not** permit you to merge PRs into the project. Members can also be assigned sections of the codebase in which they have specific expertise to be more easily requested for review as Suggested Reviewers by PR authors.

Maintainers

Some organisation Members are also project Maintainers. The number of maintainers is arbitrary and is subject to change as people join and leave the project, but has historically been less than 10. PRs can only be merged into the main project by Maintainers. While this might give the illusion that Maintainers are in control of the project, the Maintainers' role dictates that they **should not** be unilaterally deciding which PRs get merged and which don't. Instead, they should be determining the mergability of changes based primarily on the reviews and discussions of other Contributors on the GitHub PR.

Working on that basis, the Maintainers' role becomes largely *janitorial*. They are simply executing the desires of the community review process, a community which is made up of a decentralized and diverse group of Contributors.

Organisation fail-safes

It is possible for a "rogue PR" to be submitted by a Contributor; we rely on systematic and thorough peer review to catch these. There has been discussion on the mailing list about purposefully

submitting malicious PRs to test the robustness of this review process.

In the event that a Maintainer goes rogue and starts merging controversial code, or conversely, *not* merging changes that are desired by the community at large, then there are two possible avenues of recourse:

1. Have the Lead Maintainer remove the malicious Maintainer
2. In the case that the Lead Maintainer themselves is considered to be the rogue agent: fork the project to a new location and continue development there.

In the case that GitHub itself becomes the rogue entity, there have been numerous discussions about how to move away from GitHub, which have been summarized on the devwiki here. This summary came in part from discussions on this GitHub issue.

BIPs

Bitcoin uses Bitcoin Improvement Proposals (BIPs) as a design document for introducing new features or behaviour into bitcoin. Bitcoin Magazine describes what a BIP is in their article What Is A Bitcoin Improvement Proposal (BIP), specifically highlighting how BIPs are not necessarily binding documents required to achieve consensus.

The BIPs are currently hosted on GitHub in the bitcoin/bips repo.

BIP process

The BIPs include BIP 2 which self-describes the BIP process in more detail. Of particular interest might be the sections BIP Types and BIP Workflow.

What does having a BIP number assigned to an idea mean

Bitcoin Core issue #22665 described how BIP125 was not being strictly adhered to by Bitcoin Core. This raised discussion amongst developers about whether the code (i.e. "the implementation") or the BIP itself should act as the specification, with most developers expressing that they felt that "the code was the spec" and any BIP generated was merely a design document to aid with re-implementation by others, and should be corrected if necessary.

This view was not completely unanimous in the community.

For consensus-critical code most Bitcoin Core Developers consider "the code is the spec" to be the ultimate source of truth, which is one of the reasons that recommending running other full node implementations can be so difficult. A knock-on effect of this was that there were calls for review on BIP2 itself, with respect to how BIPs should be updated/amended. Newly-appointed BIP maintainer Karl-Johan Alm (a.k.a. kallewoof) posted his thoughts on this to the bitcoin-dev mailing list.

In summary a BIP represents a design document which should assist others in implementing a specific feature in a compatible way. These features are optional to usage of Bitcoin, and therefore implementation of BIPs are not required to use Bitcoin, only to remain compatible. Simply being *assigned* a BIP does **not** mean that an idea is endorsed as being a "good" idea, only that it is fully-

specified in a way that others could use to re-implement. Many ideas are assigned a BIP and then never implemented or used on the wider network.

Project stats

Bitcoin Core @ v24.0.1

Language	Files	Lines	Code	Comments	Blanks
GNU Style Assembly	1	913	742	96	75
Autoconf	23	3530	1096	1727	707
Automake	5	1803	1505	85	213
BASH	10	1772	1100	438	234
Batch	1	1	1	0	0
C	22	37994	35681	1183	1130
C Header	481	72043	43968	17682	10393
CMake	3	901	706	86	109
C++	687	197249	153482	20132	23635
Dockerfile	2	43	32	5	6
HEX	29	576	576	0	0
Java	1	23	18	0	5
JSON	94	7968	7630	0	338
Makefile	51	2355	1823	198	334
MSBuild	2	88	87	0	1
Objective-C++	3	186	134	20	32
Prolog	2	22	16	0	6
Python	298	66473	48859	7598	10016
Scheme	1	638	577	29	32
Shell	50	2612	1745	535	332
SVG	20	720	697	15	8
Plain Text	6	1125	0	1113	12
TypeScript	98	228893	228831	0	62
Visual Studio Pro|	16	956	940	0	16
Visual Studio Sol|	1	162	162	0	0
XML	2	53	47	0	6
HTML	2	401	382	0	19
|- CSS	2	98	82	1	15
(Total)		499	464	1	34
Markdown	192	33460	0	26721	6739
|- BASH	16	206	173	12	21
|- C	2	53	47	3	3
|- C++	3	345	267	54	24
|- INI	1	7	6	0	1
|- Lisp	1	13	13	0	0
|- PowerShell	1	1	1	0	0
|- Python	2	346	280	61	5
|- Shell	3	21	17	1	3

```
 |- XML                    1           4           4           0           0
 (Total)                              34456         808       26852        6796
===============================================================================
 Total                 2103       662960       530837       77663       54460
===============================================================================
```

Source: tokei

Exercises

Subsequent sections will contain various exercises related to their subject areas which will require controlling Bitcoin Core nodes, compiling Bitcoin Core and making changes to the code.

To prepare for this we will begin with the following exercises which will ensure that our environment is ready:

1. Build Bitcoin Core from source
   - ☐ Clone Bitcoin Core repository from GitHub
   - ☐ Check out the latest release tag (e.g. `v24.0.1`)
   - ☐ Install any dependencies required for your system
   - ☐ Follow the build instructions to compile the programs
   - ☐ Run `make check` to run the unit tests
   - ☐ Follow the documentation to install dependencies required to run the functional tests
   - ☐ Run the functional tests
2. Run a `bitcoind` node in regtest mode and control it using the `cli` tool

`./src/bitcoind -regtest` will start bitcoind in regtest mode. You can then control it using `./src/bitcoin-cli -regtest -getinfo`

3. Run and control a Bitcoin Core node using the `TestShell` python class from the test framework in a Jupyter notebook
   - See Running nodes via Test Framework for more information on how to do this
4. Review a Pull Request from the repo
   - ☐ Find a PR (which can be open or closed) on GitHub which looks interesting and/or accessible
   - ☐ Checkout the PR locally
   - ☐ Review the changes
   - ☐ Record any questions that arise during code review
   - ☐ Build the PR
   - ☐ Test the PR
   - ☐ Break a test / add a new test
   - ☐ Leave review feedback on GitHub, possibly including:

ACK/NACK

Approach

How you reviewed it

Your system specifications if relevant

Any suggested nits

Running nodes via Test Framework

Why

Using Bitcoin Core's Test Framework means that nodes can be started, controlled and stopped using a python control class. Additionally, they are run in a temporary directory which is automatically removed by the operating system, if not done manually.

In addition to this, the `TestShell` class has an extremely similar interface to `bitcoin-cli`, where most `bitcoin-cli` commands have an equivalent `TestShell` method, and arguments can be supplied positionally or as named values. Specifically, all `bitcoind` RPCs are available to `TestShell`.

However, certain `bitcoin-cli` commands, for example `-getinfo` require `bitcoin-cli` to call multiple RPCs and combine the results into something more user-friendly. These commands are not natively available to `TestShell`, but you can re-create them yourself by running multiple `TestShell` RPCs and combining the outputs to mimic the `bitcoin-cli` commands!

When `TestShell` is combined with a jupyter notebook the result is easy-to-setup ephemeral nodes where iteration on complex commands is more pleasant than in the shell, and complex sequences of commands can be reproduced without having to write bash scripts or use shell history.

Once a complex command or sequence of commands is established, they can generally be translated to `bitcoin-cli` commands or a shell script without much difficulty.

How

You **MUST** have a compiled `bitcoind` binary in the Bitcoin Core source directory. You can use any recent supported version of Bitcoin Core.

In order to add startup (`bitcoind` program) options to our node(s) we need this commit. We can include this two ways:

1. Use the master branch of Bitcoin Core and running `git pull`, which will include the change.
2. Use any recent tag (e.g. v24.0.1) and running `git cherry-pick 989a52e0` to pull that change into the Test Framework code.

You **MUST** have a copy of the jupyter notebook, either manually downloaded from https://github.com/chaincodelabs/onboarding-to-bitcoin-core or by cloning the onboarding-to-bitcoin-core repo (recommended) with:

```
git clone https://github.com/chaincodelabs/onboarding-to-bitcoin-core.git
```

You **MAY** want to use a python virtual environment (recommended) which can be done as follows when in the onboarding to bitcoin core top level directory:

```
cd /path/to/source/onboarding-to-bitcoin-core
python3 -m venv "obc-venv"
source obc-venv/bin/activate
```

if using fish shell you can use: `source obc-venv/bin/activate.fish` instead

Once your venv is set up and activated you can install the requirements for jupyter notebook using:

```
pip install -r requirements.txt
```

Next start the notebook with:

```
jupyter notebook
```

This will open a list of all the files in this directory. Opening the file named `exercise_tutorial.ipynb` will start the notebook containing instructions on how to use `TestShell` from the test Framework.

When you are finished you can deactivate the venv using

```
deactivate
```

Don't forget to re-activate your venv each time you want to start the Jupyter notebook after deactivating the venv!

Quick use

Once you have familiarized yourself with the `TestShell` method using `exercise_tutorial.ipynb`, you can instead start new notebooks for exercises based on the `exercise_base.ipynb` notebook, which has much of the instruction removed and will let you get started faster.

If you correct the import path for your system in this file and save it, you can then easily

make copies of it to use as start points for different exercises:

[jupyter duplicate] | *jupyter_duplicate.png*

Architecture

This section has been updated to Bitcoin Core @ v23.0

Bitcoin Core v0.1 contained 26 source code and header files (*.h/cpp) with *main.cpp* containing the majority of the business logic. As of v23.0 there are more than 800 source files (excluding *bench/*, *test/* and all subtrees), more than 200 benchmarking and cpp unit tests, and more than 200 python tests and lints.

General design principles

Over the last decade, as the scope, complexity and test coverage of the codebase has increased, there has been a general effort to not only break Bitcoin Core down from its monolithic structure but also to move towards it being a collection of self-contained subsystems. The rationale for such a goal is that this makes components easier to reason about, easier to test, and less-prone to layer violations, as subsystems can contain a full view of all the information they need to operate.

Subsystems can be notified of events relevant to them and take appropriate actions on their own. On the GUI/QT side this is handled with signals and slots, but in the core daemon this is largely still a producer/consumer pattern.

The various subsystems are often suffixed with `Manager` or `man`, e.g. `CConnman` or `ChainstateManager`.

The extra "C" in `CConnman` is a hangover from the Hungarian notation used originally by Satoshi. This is being phased out as-and-when affected code is touched during other changes.

You can see some (but not all) of these subsystems being initialized in *init.cpp#AppInitMain()*.

There is a recent preference to favour python over bash/sh for scripting, e.g. for linters, but many shell scripts remain in place for CI and contrib/ scripts.

Overview of bitcoind

The following diagram gives a brief overview of how some of the major components in bitcoind are related.

This diagram is **not** exhaustive and includes simplifications.

dashed lines indicate optional components

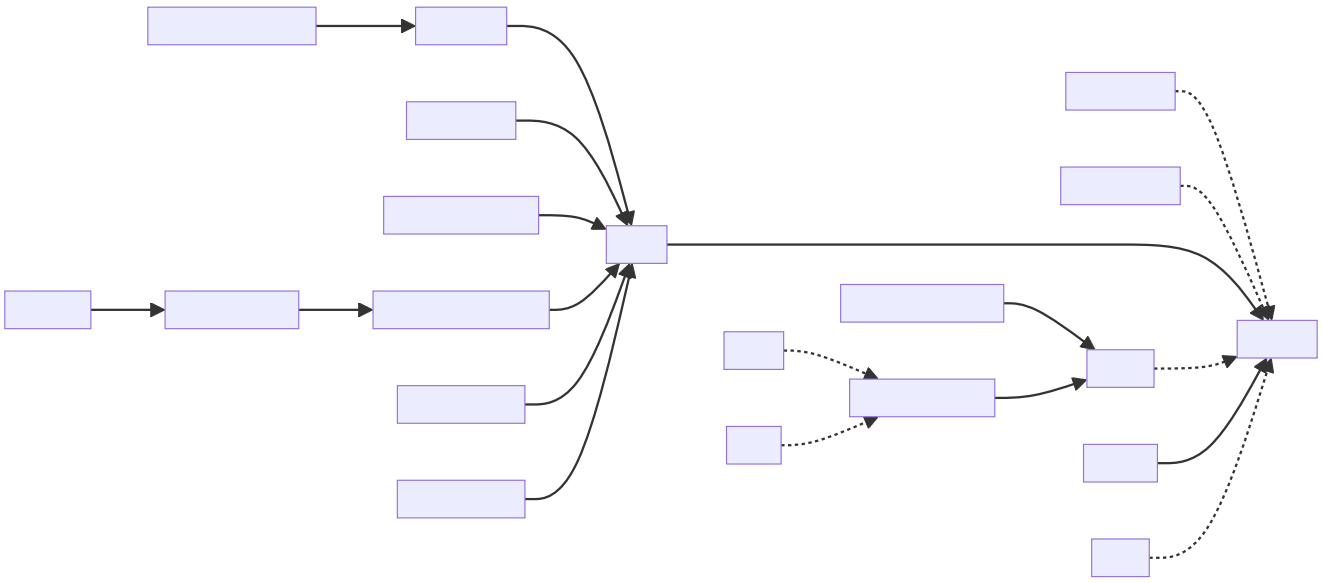

Figure 1. `bitcoind` overview

Table 2. Brief description of components in `bitcoind` overview

Component	Simplified description
`AddrMan`	Manage peers' network addresses
`CConnman`	Manage network connections to peers
`Interfaces::Chain`	Give clients access to chain state, fee rate estimates, notifications and allow tx submission
`ChainstateManager`	An interface for interacting with 1 or 2 chainstates (1. IBD-verified, 2. optional snapshot)
`NetGroupManager`	Manage net groups. Ensure we don't connect to multiple nodes in the same ASN bucket
`CTxMemPool`	Validate and store (valid) transactions which may be included in the next block
`PeerManager`	Manage peer state and interaction e.g. processing messages, fetching blocks & removing for misbehaviour
`BlockManager`	Maintains a tree of blocks on disk (via LevelDB) to determine most-work tip
`ScriptPubKeyMan`	Manages `scriptPubKeys` in a wallet. Can give out new `scriptPubKeys` as well as call into a `SigningProvider` to sign transactions

`bitcoin-cli` overview

The following diagram gives a brief overview of the major components in bitcoin-cli.

This diagram is **not** exhaustive and includes simplifications.

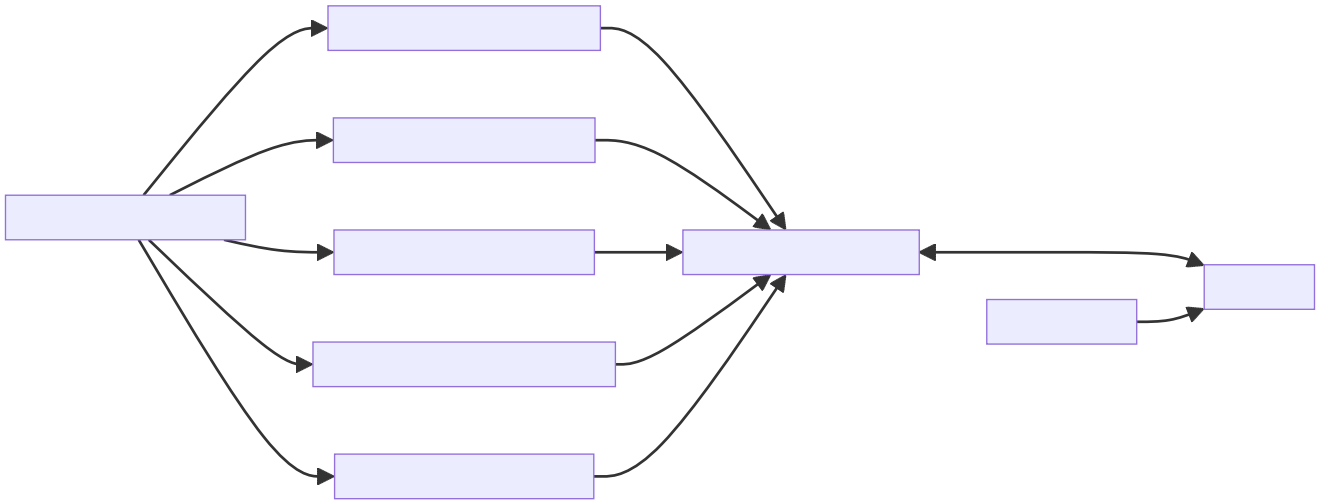

Figure 2. `bitcoin-cli` overview

Wallet structure

The following diagram gives a brief overview of how the wallet is structured.

This diagram is **not** exhaustive and includes simplifications.

dashed lines indicate optional components

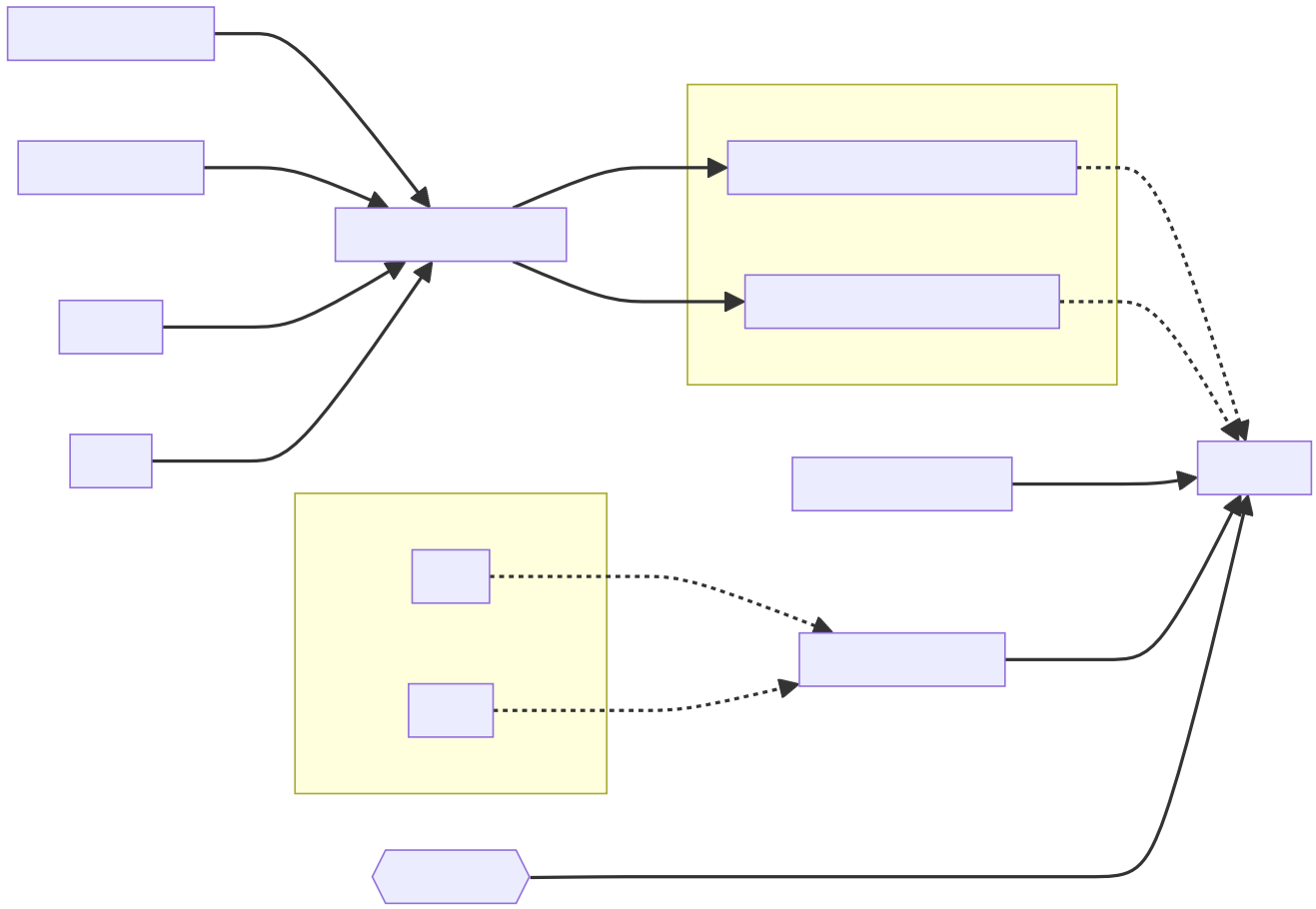

Figure 3. Wallet structure overview

Table 3. Brief description of components

Component	Simplified description
`WalletDatabase`	Represents a single wallet. Handles reads and writes to disk
`ScriptPubKeyMan`	Base class for the below SPKM classes to override before being used by `CWallet`
`DescriptorScriptPubKeyMan`	A SPKM for descriptor-based wallets
`LegacyScriptPubKeyMan`	A SPKM for legacy wallets
`SigningProvider`	An interface for a `KeyStore` to sign transactions from
`Interfaces::Chain`	Give clients access to chain state, fee rate estimates, notifications and allow tx submission
`cs_wallet`	The primary wallet lock, held for atomic wallet operations

Tests overview

Table 4. Tests overview

Tool	Usage
unit tests	`make check` or `./src/test_bitcoin`
functional tests	`test/functional/test_runner.py`
lint checks	`test/lint/all-lint.py`
fuzz	See the documentation
util tests	`test/util/test_runner.py`

Bitcoin Core is also introducing (functional) "stress tests" which challenge the program via interruptions and missing files to ensure that we fail gracefully, e.g. the tests introduced in PR#23289.

Test directory structure

The following diagram gives a brief overview of how the tests are structured within the source directory.

This diagram is **not** exhaustive and includes simplifications.

dashed lines indicate optional components

The `fuzz_targets` themselves are located in the `test` folder, however the fuzz tests are run via the `test_runner` in src/test so we point fuzz to there.

`qa_assets` are found in a separate repo altogether, as they are quite large (~3.5GB repo size and ~13.4GB on clone).

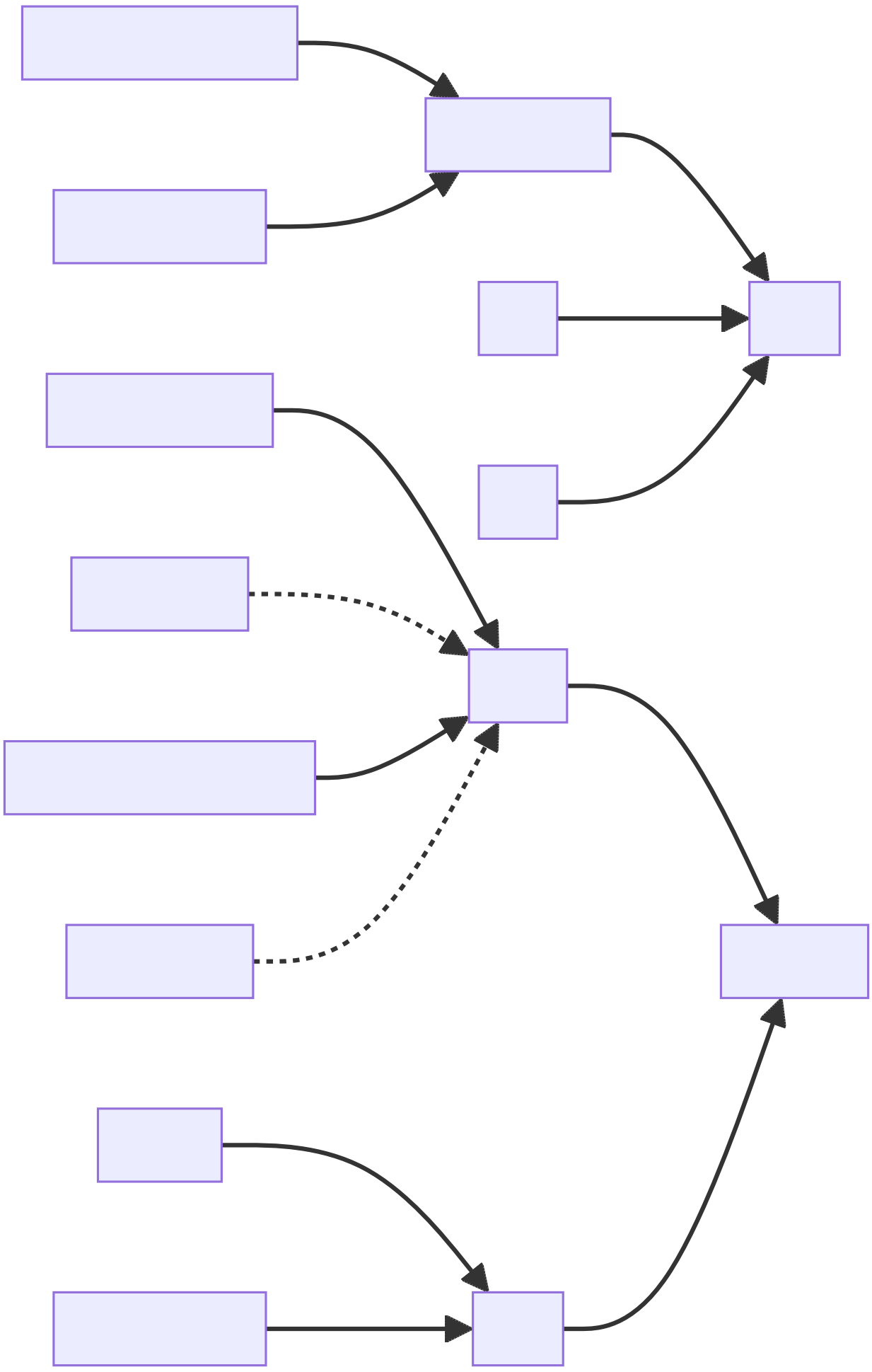

Figure 4. Test directory Structure

Test coverage

Bitcoin Core's test coverage reports can be found here.

Threads

The `main()` function starts the main bitcoind process thread, usefully named `bitcoind`. All subsequent threads are currently started as children of the `bitcoind` thread, although this is not an explicit design requirement.

The Bitcoin Core Developer docs contains a section on threads, which is summarised below in two tables, one for net threads, and one for other threads.

Table 5. Non-net threads

Name	Function	Description
`bitcoind`	`main()`	Responsible for starting up and shutting down the application, and spawning all sub-threads
`b-loadblk`	`ThreadImport`	Loads blocks from `blk*.dat` files or `-loadblock=<file>` on startup
`b-scriptch.x`	`ThreadScriptCheck`	Parallel script validation threads for transactions in blocks
`b-http`	`ThreadHTTP`	Libevent thread to listen for RPC and REST connections
`b-httpworker.x`	`StartHTTPServer`	HTTP worker threads. Threads to service RPC and REST requests
`b-txindex.x`	`ThreadSync`	Indexer threads. One thread per indexer
`b-scheduler`	`SchedulerThread`	Does asynchronous background tasks like dumping wallet contents, dumping `addrman` and running asynchronous `validationinterface` callbacks
`b-torcontrol`	`TorControlThread`	Libevent thread for tor connections

Net threads

Table 6. Net threads

Name	Function	Description
`b-msghand`	`ThreadMessageHandler`	Application level message handling (sending and receiving). Almost all `net_processing` and validation logic runs on this thread
`b-dnsseed`	`ThreadDNSAddressSeed`	Loads addresses of peers from the `ThreadDNS`
`b-upnp`	`ThreadMapPort`	Universal plug-and-play startup/shutdown
`b-net`	`ThreadSocketHandler`	Sends/Receives data from peers on port 8333
`b-addcon`	`ThreadOpenAddedConnections`	Opens network connections to added nodes

Name	Function	Description
b-opencon	ThreadOpenConnections	Initiates new connections to peers
b-i2pacceptt	ThreadI2PAcceptIncoming	Listens for and accepts incoming I2P connections through the I2P SAM proxy

Thread debugging

In order to debug a multi-threaded application like bitcoind using gdb you will need to enable following child processes. Below is shown the contents of a file `threads.brk` which can be sourced into gdb using `source threads.brk`, before you start debugging bitcoind. The file also loads break points where new threads are spawned.

threads.brk

```
set follow-fork-mode child
break node::ThreadImport
break StartScriptCheckWorkerThreads
break ThreadHTTP
break StartHTTPServer
break ThreadSync
break SingleThreadedSchedulerClient
break TorControlThread
break ThreadMessageHandler
break ThreadDNSAddressSeed
break ThreadMapPort
break ThreadSocketHandler
break ThreadOpenAddedConnections
break ThreadOpenConnections
break ThreadI2PAcceptIncoming
```

Library structure

Bitcoin Core compilation outputs a number of libraries, some which are designed to be used internally, and some which are designed to be re-used by external applications. The internally-used libraries generally have unstable APIs making them unsuitable for re-use, but `libbitcoin_consensus` and `libbitcoin_kernel` are designed to be re-used by external applications.

Bitcoin Core has a guide which describes the various libraries, their conventions, and their various dependencies. The dependency graph is shown below for convenience, but may not be up-to-date with the Bitcoin Core document.

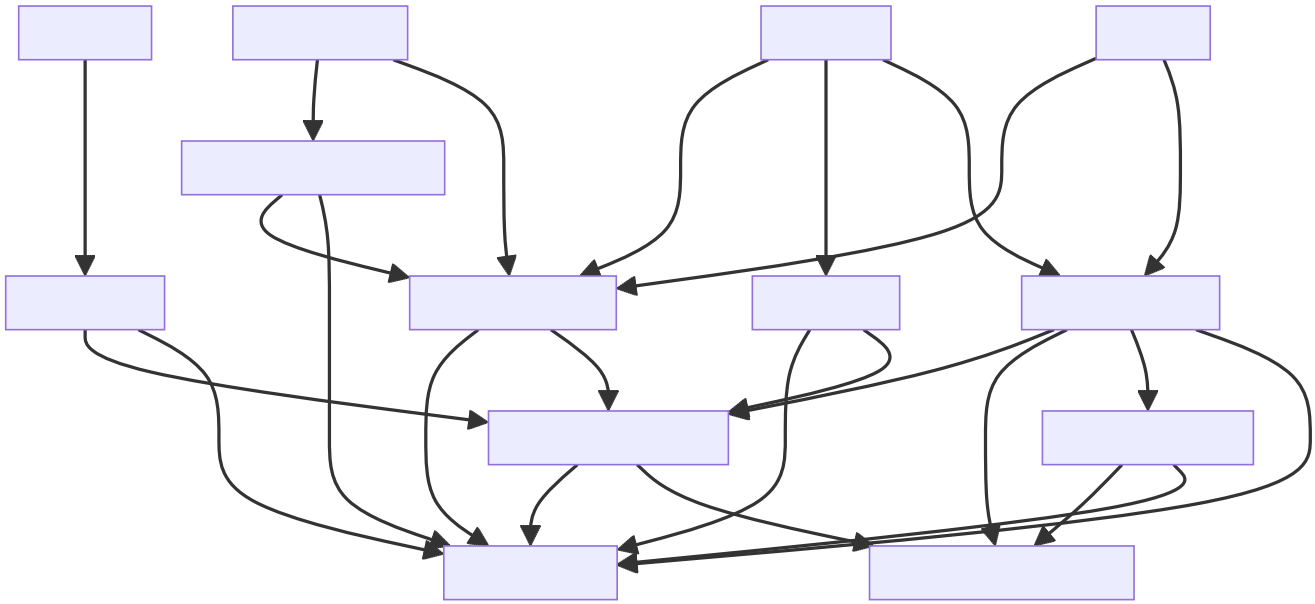

Figure 5. Bitcoin library dependency graph

It follows that API changes to the libraries which are internally-facing can be done slightly easier than for libraries with externally-facing APIs, for which more care for compatibility must be taken.

Source code organization

Issue #15732 describes how the Bitcoin Core project is striving to organize libraries and their associated source code, copied below for convenience:

> Here is how I am thinking about the organization:
>
> - `libbitcoin_server.a`, `libbitcoin_wallet.a`, and `libbitcoinqt.a` should all be terminal dependencies. They should be able to depend on other symbols in other libraries, but no other libraries should depend on symbols in them (and they shouldn't depend on each other).
> - `libbitcoin_consensus.a` should be a standalone library that doesn't depend on symbols in other libraries mentioned here
> - `libbitcoin_common.a` and `libbitcoin_util.a` seem very interchangeable right now and mutually depend on each other. I think we should either merge them into one library, or create a new top-level `src/common/` directory complementing `src/util/`, and start to distinguish general purpose utility code (like argument parsing) from bitcoin-specific utility code (like formatting bip32 paths and using ChainParams). Both these libraries can be depended on by `libbitcoin_server.a`, `libbitcoin_wallet.a`, and `libbitcoinqt.a`, and they can depend on `libbitcoin_consensus.a`. If we want to split util and common up, as opposed to merging them together, then util shouldn't depend on

> libconsensus, but common should.
>
> Over time, I think it'd be nice if source code organization reflected library organization . I think it'd be nice if all `libbitcoin_util` source files lived in `src/util`, all `libbitcoin_consensus.a` source files lived in `src/consensus`, and all `libbitcoin_server.a` code lived in `src/node` (and maybe the library was called `libbitcoin_node.a`).

You can track the progress of these changes by following links from the issue to associated PRs.

The `libbitcoin-kernel` project will provide further clean-ups and improvements in this area.

If you want to explore for yourself which sources certain libraries require on the current codebase, you can open the file `src/Makefile.am` and search for `_SOURCES`.

Userspace files

Bitcoin Core stores a number of files in its data directory ($DATADIR) at runtime.

Block and undo files

$DATADIR/blocks/blk*.dat

Stores raw network-format block data in order received.

$DATADIR/blocks/rev*.dat

Stores block "undo" data in order processed.

> You can see blocks as 'patches' to the chain state (they consume some unspent outputs, and produce new ones), and see the undo data as reverse patches. They are necessary for rolling back the chainstate, which is necessary in case of reorganisations.
>
> — Pieter Wuille, stackexchange

Indexes

With data from the raw block* and rev* files, various LevelDB indexes can be built. These indexes enable fast lookup of data without having to rescan the entire block chain on disk.

Some of these databases are mandatory and some of them are optional and can be enabled using run-time configuration flags.

Block Index

Filesystem location of blocks + some metadata

Chainstate Index

All current UTXOs + some metadata

Tx Index

Filesystem location of all transactions by txid

Block Filter Index

BIP157 filters, hashes and headers

Coinstats Index

UTXO set statistics

Name	Location	Optional	Class
Block Index	*$DATADIR/blocks/index*	No	`BlockIndex`
Chainstate Index	*$DATADIR/chainstate*	No	`Chainstate`
Tx Index	*$DATADIR/indexes/txindex*	Yes	`TxIndex`
Block Filter Index	*$DATADIR/indexes/blockfilter/<filter name>*	Yes	`BlockFilterIndex`
Coinstats Index	*$DATADIR/indexes/coinstats*	Yes	`CoinstatsIndex`

Deep technical dive

lsilva01 has written a deep technical dive into the architecture of Bitcoin Core as part of the Bitcoin Core Onboarding Documentation in Bitcoin Architecture.

Once you've gained some insight into the architecture of the program itself you can learn further details about which code files implement which functionality from the Bitcoin Core regions document.

James O'Beirne has recorded 3 videos which go into detail on how the codebase is laid out, how the build system works, what developer tools there are, as well as what the primary function of many of the files in the codebase are:

1. Architectural tour of Bitcoin Core (part 1 of 3)
2. Architectural tour of Bitcoin Core (part 2 of 3)
3. Architectural tour of Bitcoin Core (part 3 of 3)

ryanofsky has written a handy guide covering the different libraries contained within Bitcoin Core, along with some of their conventions and a dependency graph for them. Generally speaking, the desire is for the Bitcoin Core project to become more modular and less monolithic over time.

Subtrees

Several parts of the repository (LevelDB, crc32c, secp256k1 etc.) are subtrees of software maintained elsewhere.

Some of these are maintained by active developers of Bitcoin Core, in which case changes should go directly upstream without being PRed directly against the project. They will be merged back in the next subtree merge.

Others are external projects without a tight relationship with our project.

There is a tool in `test/lint/git-subtree-check.sh` to check a subtree directory for consistency with its upstream repository.

See the full subtrees documentation for more information.

Implementation separation

Many of the classes found throughout the codebase use the PIMPL technique to separate their implementation from the external representation. See PIMPL technique in the Appendix for more information.

Consensus and Validation

This section has been updated to Bitcoin Core @ v23.0

One of the fundamental concepts underlying bitcoin is that nodes on the network are able to maintain decentralized consensus on the ordering of transactions in the system.

The primary mechanism at work is that all nodes validate every block, and every transaction contained within that block, against their own copy of the consensus rules. The secondary mechanism is that in the event of a discrepancy between two competing chain tips nodes should follow the chain with the most cumulative proof-of-work. The result is that all honest nodes in the network will eventually converge onto a single, canonical, valid chain.

If all nodes do not compute consensus values identically (including edge cases) a chainsplit will result.

For more information on how the bitcoin networks' decentralized consensus mechanism works see the Mastering Bitcoin section on decentralized consensus.

In Bitcoin Core there are an extra level of validation checks applied to incoming transactions in addition to consensus checks called "policy" which have a slightly different purpose, see consensus vs policy for more information on the differences between the two.

Consensus

A collection of functions and variables which **must** be computed identically to all other nodes on the network in order to remain in consensus and therefore on the main chain.

Validation

Validation of blocks, transactions and scripts, with a view to permitting them to be added to

either the blockchain (must pass consensus checks) or our local mempool (must pass policy checks).

Consensus in Bitcoin Core

Naturally one might assume that all code related to consensus could be found in the *src/consensus/* directory, however this is not entirely the case. Components of consensus-related code can be found across the Bitcoin Core codebase in a number of files, including but not limited to:

```
 bitcoin
   src
     consensus
     script
       interpreter.cpp
     validation.h
     validation.cpp
```

Consensus-critical functions can also be found in proximity to code which could affect whether a node considers a transaction or block valid. This could extend to, for example, block storage database code.

An abbreviated list of some of the more notable consensus functions and variables is shown below.

Table 7. Some consensus functions and variables

File	Objects
src/consensus/amount.h	`COIN`, `MAX_MONEY`, `MoneyRange()`
src/consensus/consensus.h	`BLOCK{SIZE|WEIGHT|SIGOPS_COST}`, `COINBASE_MATURITY`, `WITNESS_SCALE_FACTOR`, `MIN_TX_WEIGHT`
src/consensus/merkle.{h|cpp}	`ComputeMerkleRoot()`, `BlockMerkleRoot()`, `BlockWitnessMerkleRoot()`
src/consensus/params.h	`BuriedDeployment`, `Params`(buried blocks which are valid but known to fail default script verify checks, BIP height activations, PoW params)
src/consensus/tx_check.{h|cpp}	`CheckTransaction()`
src/consensus/tx_verify.{h|cpp}	`CheckTxInputs()`, `Get{Legacy}SigOpCount()`, `IsFinalTx()`, `SequenceLock(s)()`
src/consensus/validation.h	`TxValidationResult` (validation result reason), `BlockValidationResult` (validation result reason), `ValidationState`, `Get{Transaction|Block|TransactionInput}Weight()`

Consensus model

The consensus model in the codebase can be thought of as a database of the current state of the blockchain. When a new block is learned about it is processed and the consensus code must determine which block is the current best. Consensus can be thought of as a function of available

information — it's output is simply a deterministic function of its input.

There are a simple set of rules for determining the best block:

1. Only consider valid blocks
2. Where multiple chains exist choose the one with the most cumulative Proof of Work (PoW)
3. If there is a tie-breaker (same height and work), then use first-seen

The result of these rules is a tree-like structure from genesis to the current day, building on only valid blocks.

Whilst this is easy-enough to reason about in theory, the implementation doesn't work exactly like that. It must consider state, do I go forward or backwards for example.

Validation in Bitcoin Core

Originally consensus and validation were much of the same thing, in the same source file. However splitting of the code into strongly delineated sections was never fully completed, so validation.* files still hold some consensus codepaths.

Consensus vs Policy

What is the difference between consensus and policy checks? Both seem to be related to validating transactions. We can learn a lot about the answer to this question from sdaftuar's StackExchange answer.

The answer teaches us that policy checks are a superset of validation checks — that is to say that a transaction that passes policy checks has implicitly passed consensus checks too. Nodes perform policy-level checks on all transactions they learn about before adding them to their local mempool. Many of the policy checks contained in `policy` are called from inside `validation`, in the context of adding a new transaction to the mempool.

Consensus and validation bugs

Consensus and validation bugs can arise both from inside the Bitcoin Core codebase itself, and from external dependencies. Bitcoin wiki lists some CVE and other Exposures.

OpenSSL consensus failure

Pieter Wuille disclosed the possibility of a consensus failure via usage of OpenSSL. The issue was that the OpenSSL signature verification was accepting **multiple** signature serialization formats (for the same signature) as valid. This effectively meant that a transactions' ID (txid) could be changed, because the signature contributes to the txid hash.

▼ *Click to show the code comments related to pubkey signature parsing from pubkey.cpp*

src/pubkey.cpp

```
/** This function is taken from the libsecp256k1 distribution and implements
 *  DER parsing for ECDSA signatures, while supporting an arbitrary subset of
 *  format violations.
 *
 *  Supported violations include negative integers, excessive padding, garbage
 *  at the end, and overly long length descriptors. This is safe to use in
 *  Bitcoin because since the activation of BIP66, signatures are verified to be
 *  strict DER before being passed to this module, and we know it supports all
 *  violations present in the blockchain before that point.
 */
int ecdsa_signature_parse_der_lax(const secp256k1_context* ctx,
secp256k1_ecdsa_signature* sig, const unsigned char *input, size_t inputlen) {
    // ...
}
```

There were a few cases to consider:

1. signature length descriptor malleation (extension to 5 bytes)
2. third party malleation: signature may be slightly "tweaked" or padded
3. third party malleation: negating the `S` value of the signature

In the length descriptor case there is a higher risk of causing a consensus-related chainsplit. The sender can create a normal-length valid signature, but which uses a 5 byte length descriptor meaning that it might not be accepted by OpenSSL on all platforms.

Note that the sender can also "malleate" the signature whenever they like, by simply creating a new one, but this will be handled differently than a length-descriptor-extended signature.

In the second case, signature tweaking or padding, there is a lesser risk of causing a consensus-related chainsplit. However the ability of third parties to tamper with valid transactions may open up off-chain attacks related to Bitcoin services or layers (e.g. Lightning) in the event that they are relying on txids to track transactions.

It is interesting to consider the order of the steps taken to fix this potential vulnerability:

1. First the default policy in Bitcoin Core was altered (via `isStandard()`) to prevent the software from relaying or accepting into the mempool transactions with non-DER signature encodings. This was carried out in PR#2520.
2. Following the policy change, the strict encoding rules were later enforced by consensus in PR#5713.

We can see the resulting flag in the script verification enum:

src/script/interpreter.h

```
// Passing a non-strict-DER signature or one with undefined hashtype to a checksig
```

```
operation causes script failure.
// Evaluating a pubkey that is not (0x04 + 64 bytes) or (0x02 or 0x03 + 32 bytes) by
checksig causes script failure.
// (not used or intended as a consensus rule).
SCRIPT_VERIFY_STRICTENC = (1U << 1),
```

▼ *Expand to see where this flag is checked in src/script/interpreter.cpp*

```
bool CheckSignatureEncoding(const std::vector<unsigned char> &vchSig, unsigned int
flags, ScriptError* serror) {
    // Empty signature. Not strictly DER encoded, but allowed to provide a
    // compact way to provide an invalid signature for use with CHECK(MULTI)SIG
    if (vchSig.size() == 0) {
        return true;
    }
    if ((flags & (SCRIPT_VERIFY_DERSIG | SCRIPT_VERIFY_LOW_S |
SCRIPT_VERIFY_STRICTENC)) != 0 && !IsValidSignatureEncoding(vchSig)) {
        return set_error(serror, SCRIPT_ERR_SIG_DER);
    } else if ((flags & SCRIPT_VERIFY_LOW_S) != 0 && !IsLowDERSignature(vchSig,
serror)) {
        // serror is set
        return false;
    } else if ((flags & SCRIPT_VERIFY_STRICTENC) != 0 && !
IsDefinedHashtypeSignature(vchSig)) {
        return set_error(serror, SCRIPT_ERR_SIG_HASHTYPE);
    }
    return true;
}

bool static CheckPubKeyEncoding(const valtype &vchPubKey, unsigned int flags, const
SigVersion &sigversion, ScriptError* serror) {
    if ((flags & SCRIPT_VERIFY_STRICTENC) != 0 && !IsCompressedOrUncompressedPubKey
(vchPubKey)) {
        return set_error(serror, SCRIPT_ERR_PUBKEYTYPE);
    }
    // Only compressed keys are accepted in segwit
    if ((flags & SCRIPT_VERIFY_WITNESS_PUBKEYTYPE) != 0 && sigversion ==
SigVersion::WITNESS_V0 && !IsCompressedPubKey(vchPubKey)) {
        return set_error(serror, SCRIPT_ERR_WITNESS_PUBKEYTYPE);
    }
    return true;
}
```

Do you think this approach — first altering policy, followed later by consensus — made sense for implementing the changes needed to fix this consensus vulnerability? Are there circumstances where it might not make sense?

Having OpenSSL as a consensus-critical dependency to the project was ultimately fixed in PR#6954 which switched to using the in-house libsecp256k1 library (as a subtree) for signature verification.

Database consensus

Historically Bitcoin Core used Berkeley DB (BDB) for transaction and block indices. In 2013 a migration to LevelDB for these indices was included with Bitcoin Core v0.8. What developers at the time could not foresee was that nodes that were still using BDB, all pre 0.8 nodes, were silently consensus-bound by a relatively obscure BDB-specific database lock counter.

BDB required a configuration setting for the total number of locks available to the database.

Bitcoin Core was interpreting a failure to grab the required number of locks as equivalent to block validation failing. This caused some BDB-using nodes to mark blocks created by LevelDB-using nodes as invalid and caused a consensus-level chain split. BIP 50 provides further explanation on this incident.

Although database code is not in close proximity to the `/src/consensus` region of the codebase it was still able to induce a consensus bug.

BDB has caused other potentially-dangerous behaviour in the past. Developer Greg Maxwell describes in a Q&A how even the same versions of BDB running on the same system exhibited non-deterministic behaviour which might have been able to initiate chain re-orgs.

An inflation bug

This Bitcoin Core disclosure details a potential inflation bug.

It originated from trying to speed up transaction validation in `main.cpp#CheckTransaction()` which is now `consensus/tx_check.cpp#CheckTransaction()`, something which would in theory help speed up IBD (and less noticeably singular/block transaction validation). The result in Bitcoin Core versions 0.15.x → 0.16.2 was that a coin that was created in a previous block, could be spent twice in the same block by a miner, without the block being rejected by other Bitcoin Core nodes (of the aforementioned versions).

Whilst this bug originates from validation, it can certainly be described as a breach of consensus parameters. In addition, nodes of version 0.14.x ⇐ `node_version` >= 0.16.3 would reject inflation blocks, ultimately resulting in a chain split provided that miners existed using both inflation-resistant and inflation-permitting clients.

Hard & Soft Forks

Before continuing with this section, ensure that you have a good understanding of what soft and hard forks are, and how they differ. Some good resources to read up on this further are found in the table below.

Table 8. Hard and soft fork resources

Title	Resource	Link
What is a soft fork, what is a hard fork, what are their differences?	StackExchange	link

Title	Resource	Link
Soft forks	bitcoin.it/wiki	link
Hard forks	bitcoin.it/wiki	link
Soft fork activation	Bitcoin Optech	link
List of consensus forks	BitMex research	link
A taxonomy of forks (BIP99)	BIP	link
Modern Soft Fork Activation	bitcoin-dev mailing list	link
Chain splits and Resolutions	BitcoinMagazine guest	link

When making changes to Bitcoin Core its important to consider whether they could have any impact on the **consensus rules**, or the interpretation of those rules. If they do, then the changes will end up being either a soft or hard fork, depending on the nature of the rule change.

As described, certain Bitcoin Core components, such as the block database can also unwittingly introduce forking behaviour, even though they do not directly modify consensus rules.

Some of the components which are known to alter consensus behaviour, and should therefore be approached with caution, are listed in the section consensus components.

Changes are not made to consensus values or computations without extreme levels of review and necessity. In contrast, changes such as refactoring can be (and are) made to areas of consensus code, when we can be sure that they will not alter consensus validation.

Making forking changes

There is some debate around whether it's preferable to make changes via soft or hard fork. Each technique has advantages and disadvantages.

Table 9. Hard vs soft forks for changes

Type	Advantages	Disadvantages
Soft fork	• Backwards compatible • **Low risk of chain split** in worst case	• Cannot change all values (e.g. block size, money supply) • Might require clever programming tricks • Might introduce "technical debt" and associated comprehension burden on reviewers and future programmers

Type	Advantages	Disadvantages
Hard fork	• Can change any values you want (e.g. block size, money supply) • Might be cleaner (code-wise) and therefore easier to reason about	• Not backwards compatible ◦ Requires all nodes to upgrade in lock-step • **High risk of chainsplit** • We have no experience with them • Other changes often required • See bitcoincore.org for more information

Upgrading consensus rules with soft forks

When soft-forking in new bitcoin consensus rules it is important to consider how old nodes will interpret the new rules. For this reason the preferred method historically was to make something (e.g. an unused OPCODE which was to be repurposed) "non-standard" prior to the upgrade. Making the opcode non-standard has the effect that transaction scripts using it will not be relayed by nodes using this policy. Once the soft fork is activated policy is amended to make relaying transactions using this opcode standard policy again, so long as they comply with the ruleset of the soft fork.

Soft forking marble statues

An analogy might be to think of the current consensus ruleset like a big block of marble. The current rules have already been carved out of it and eventually it will form into a complex statue.

As we soft fork new rules into bitcoin we are taking an un-touched area of the marble and carving something new out of it. Importantly with soft forks we can only ever take parts of the marble *away*, so we must be considerate about what, where and how much we carve out for any upgrade.

There are parts of the statue currently untouched because they're reserved for future upgrades.

Using the analogy above, we could think of OP_NOP opcodes as unsculpted areas of marble.

Currently OP_NOP1 and OP_NOP4-NOP_NOP10 remain available for this.

Once the opcode has been made non-standard we can then sculpt the new rule from the marble and later re-standardize transactions using the opcode so long as they follow the new rule.

This makes sense from the perspective of an old, un-upgraded node who we are trying to remain in consensus with. From their perspective they see an OP_NOP performing (like the name implies) nothing, but not marking the transaction as invalid. After the soft fork they will *still* see the (repurposed) OP_NOP apparently doing nothing but also not failing the transaction.

However from the perspective of the upgraded node they now have two possible evaluation paths

for the OP_NOP: 1) Do nothing (for the success case) and 2) Fail evaluation (for the failure case). This is summarized in the table below.

Table 10. Soft forking changes using OP_NOP opcodes

	Before soft fork	**After soft fork**
Legacy node	1) Nothing	1) Nothing
Upgraded Node	1) Nothing	1) Nothing (soft forked rule evaluation success) 2) Mark transaction invalid (soft forked rule evaluation failure)

You may notice here that there is still room for discrepancy; a miner who is not upgraded could possibly include transactions in a block which were valid according to legacy nodes, but invalid according to upgraded nodes. If this miner had any significant hashpower this would be enough to initiate a chainsplit, as upgraded miners would not follow this tip.

Selecting upgrade activation times

Originally Satoshi used height-based upgrade points for activating soft forks. The bitcoin network was so small and concentrated, and Satoshi could dictate the height quite easily, that this worked OK in that era.

After Satoshi left attempts were made to make the activation point a more predictable moment in *time*; with the intent on assisting engineers and services who relied on knowing when the upgrade was likely to activate (as wall time). For this reason BIP16 and BIP30 were activated on a (block) timestamp, after miners had signalled readiness for the upgrade in their coinbase transactions.

The concept of miner activated soft forks (MASF) were invented with BIP34 which said that every coinbase transaction needed to include the (block) height as the first item in its scriptSig, along with an increased block version number. The block height requirement had the effect that no two coinbase transactions could have the same txid, which was previously possible (see 1 and 2 for example). The increased version number was accompanied by rules which stipulated a form of miner readiness signalling, which could avoid a diktat from any individual about what time a particular upgrade should be activated.

The UTXO in the second of those two blocks, along with a second block also containing a duplicate coinbase txid have a special carve-out in the code to enable them to pass validation.

Unfortunately though the second UTXO effectively overwrote the first in the UTXO set, so in both cases 50 BTC was lost from the spendable supply.

MASF was used for BIP65 and BIP66. A summary of the mechanism is:

- If 750/1000 blocks signal this new version number then the new rule is active.
- At 950/1000 you **must** signal.
  - Forcibly kick the last 5% stragglers out.

However, even using miner signalling for BIP16 had already caused drama, as the idea of activation based on miner signalling was interpreted as a vote (by only miners), rather than what it was, which was miners saying "yes, I am ready for the upgrade".

When upgrading via soft fork we want everyone to be on the same page to minimize the risk of a chainsplit and miner signalling was deemed the best method we had to achieve rough consensus on this.

> Whenever we want to change the consensus rules, this presents a serious problem because we don't really want to just force new rules on the network. There's no central authority that can do this really. We need to have a way for the network to adapt to the new rules, decide whether or not it wants to adjust to these rules, and to make sure that everyone still ends up agreeing in the end.
>
> — Eric Lombrozo, Bitcoin Magazine

In the end bitcoin developers concluded that MASF indeed had potential for centralization and so produced the BIP9 specification with which to use for future upgrades.

Repurposing OP_NOPs does have its limitations. First and foremost they cannot manipulate the stack, as this is something that un-upgraded nodes would not expect or validate identically. Getting rid of the OP_DROP requirement when using repurposed OP_NOPs would require a hard fork.

Examples of soft forks which re-purposed OP_NOPs include CLTV and CSV. Ideally these operations would remove the subsequent object from the stack when they had finished processing it, so you will often see them followed by OP_DROP which removes the object, for example in the script used for the `to_local` output in a lightning commitment transaction:

Lightning commitment transaction output

```
OP_IF
    # Penalty transaction
    <revocationpubkey>
OP_ELSE
    `to_self_delay`
    OP_CHECKSEQUENCEVERIFY
    OP_DROP
    <local_delayedpubkey>
OP_ENDIF
OP_CHECKSIG
```

There are other limitations associated with repurposing OP_NOPs, and ideally bitcoin needed a better upgrade system…

SegWit upgrade

SegWit was the first attempt to go beyond simply repurposing OP_NOPs for upgrades. The idea was that the `scriptPubKey`/`redeemScript` would consist of a 1 byte push opcode (0-16) followed by a data push between 2 and 40 bytes. The value of the first push would represent the version number, and the second push the witness program. If the conditions to interpret this as a SegWit script were matched, then this would be followed by a `witness`, whose data varied on whether this was a P2WPKH or P2WSH witness program.

Legacy nodes, who would not have the witness data, would interpret this output as `anyonecanspend` and so would be happy to validate it, whereas upgraded nodes could validate it using the additional `witness` against the new rules. To revert to the statue analogy this gave us the ability to work with a new area of the marble which was entirely untouched.

The addition of a versioning scheme to SegWit was a relatively late addition which stemmed from noticing that, due to the CLEANSTACK policy rule which required exactly 1 true element to remain on the stack after execution, SegWit outputs would be of the form `OP_N + DATA`. With SegWit we wanted a compact way of creating a new output which didn't have any consensus rules associated with it, yet had lots of freedom, was ideally already non-standard, and was permitted by CLEANSTACK.

The solution was to use two pushes: according to old nodes there are two elements, which was non-standard. The first push must be at least one byte, so we can use one of the `OP_N` opcodes, which we then interpret as the SegWit version. The second is the data we have to push.

Whilst this immediately gave us new upgrade paths via SegWit versions Taproot (SegWit version 1) went a step further and declared *new opcodes inside of SegWit*, also evaluated as `anyonecanspend` by nodes that don't support SegWit, giving us yet more soft fork upgradability. These opcodes could in theory be used for anything, for example if there was ever a need to have a new consensus rule on 64 bit numbers we could use one of these opcodes.

Fork wish lists

There are a number of items that developers have had on their wish lists to tidy up in future fork events.

An email from Matt Corallo with the subject "The Great Consensus Cleanup" described a "wish list" of items developers were keen to tidy up in a future soft fork.

The Hard Fork Wishlist is described on this en.bitcoin.it/wiki page. The rationale for collecting these changes together, is that if backwards-incompatible (hard forking) changes are being made, then we "might as well" try and get a few in at once, as these events are so rare.

Bitcoin core consensus specification

A common question is where the bitcoin protocol is documented, i.e. specified. However bitcoin does not have a formal specification, even though many ideas have some specification (in BIPS) to aid re-implementation.

The requirements to be compliant with "the bitcoin spec" are to be bug-for-bug compatible with the Bitcoin Core implementation.

The reasons for Bitcoin not having a codified specification are historical; Satoshi never released one. Instead, in true "Cypherpunks write code" style and after releasing a general whitepaper, they simply released the first client. This client existed on it's own for the best part of two years before others sought to re-implement the rule-set in other clients:

- libbitcoin
- BitcoinJ

A forum post from Satoshi in June 2010 had however previously discouraged alternative implementations with the rationale:

> ...
>
> I don't believe a second, compatible implementation of Bitcoin will ever be a good idea. So much of the design depends on all nodes getting exactly identical results in lockstep that a second implementation would be a menace to the network. The MIT license is compatible with all other licenses and commercial uses, so there is no need to rewrite it from a licensing standpoint.
>
> — Satoshi Nakamoto

It is still a point of contention amongst some developers in the community, however the fact remains that if you wish to remain in consensus with the majority of (Bitcoin Core) nodes on the network, you must be *exactly* bug-for-bug compatible with Bitcoin Core's consensus code.

If Satoshi *had* launched Bitcoin by providing a specification, could it have ever been specified well-enough to enable us to have multiple node implementations?

One mechanism often employed by those who want to run custom node software is to position an up-to-date Bitcoin Core node to act as a "gateway" to the network. Internally your own node can then make a single connection to this Bitcoin Core node. This means that your custom internal node will now only receive transactions and blocks which have passed Bitcoin Core's consensus (or policy) checks, allowing you to be sure that your custom node is not accepting objects which could cause you to split onto a different chain tip.

libbitcoinconsensus

The libbitcoinconsensus library is described in the 0.10.0 release notes:

> Consensus library

> Starting from 0.10.0, the Bitcoin Core distribution includes a consensus library.
>
> The purpose of this library is to make the verification functionality that is critical to Bitcoin's consensus available to other applications, e.g. to language bindings such as [python-bitcoinlib](https://pypi.python.org/pypi/python-bitcoinlib) or alternative node implementations.
>
> This library is called `libbitcoinconsensus.so` (or, `.dll` for Windows). Its interface is defined in the C header [bitcoinconsensus.h](https://github.com/bitcoin/bitcoin/blob/0.10/src/script/bitcoinconsensus.h).
>
> In its initial version the API includes two functions:
>
> - `bitcoinconsensus_verify_script` verifies a script. It returns whether the indicated input of the provided serialized transaction correctly spends the passed scriptPubKey under additional constraints indicated by flags
> - `bitcoinconsensus_version` returns the API version, currently at an experimental `0`
>
> The functionality is planned to be extended to e.g. UTXO management in upcoming releases, but the interface for existing methods should remain stable.

libbitcoinkernel

The libbitcoinkernel project seeks to modularise Bitcoin Cores' consensus engine and make it easier for developers to reason about when they are modifying code which could be consensus-critical.

This project differs from `libbitcoinconsensus` in that it is designed to be a stateful engine, with a view to eventually: being able to spawn its own threads, do caching (e.g. of script and signature verification), do its own I/O, and manage dynamic objects like a mempool. Another benefit of fully extracting the consensus engine in this way may be that it becomes easier to write and reason about consensus test cases.

In the future, if a full de-coupling is successfully completed, other Bitcoin applications might be able to use `libbitcoinkernel` as their own consensus engine permitting multiple full node implementations to operate on the network in a somewhat safer manner than many of them operate under today. The initial objective of this library however is to actually have it used by Bitcoin Core internally, something which is not possible with libbitcoinconsensus due to it's lack of caching and state (making it too slow to use).

Some examples have surfaced recently where script validation in the BTCD code (used internally by LND) has diverged from the results from Bitcoin Core:

1. Witness size check: issue and fix
2. Max witness items check: issue and fix.

The implementation approaches of libbitcoinconsensus and libbitcoinkernel also differ; with lb-consensus parts of consensus were moved into the library piece by piece, with the eventual goal that it would be encapsulated. lb-kernel takes a different approach — first cast a super wide net around everything needed to run a consensus engine, and then gradually strip pieces out where they can be. In theory this should get us something which Bitcoin Core can use much faster (in fact, you can build the optional `bitcoin-chainstate` binary which already has some functionality).

Part of libbitcoinkernel has been merged in via Carl Dong's `bitcoin-chainstate` PR. It also has its own project board to track progress.

Hardcoded consensus values

consensus/consensus.h contains a number of `static const` values relating to consensus rules. These are globally shared between files such as *validation.cpp*, *rpc_mining.cpp* and *rpc/mining.cpp*. These consensus-critical values are marked as `const` so that there is no possibility that they can be changed at any point during program execution.

One example of this would be the maximum block weight which should not ever be exceeded:

```
static const unsigned int MAX_BLOCK_WEIGHT = 4000000;
```

consensus/amount.h contains the conversion rate between satoshis and one "bitcoin", as well as a `MAX_MONEY` constant. These are marked as `constexpr` to indicate that they should be evaluated at compile time and then remain as `const` during execution.

```
/** The amount of satoshis in one BTC. */
static constexpr CAmount COIN = 100000000;

/** No amount larger than this (in satoshi) is valid.
 *
 * Note that this constant is *not* the total money supply, which in Bitcoin
 * currently happens to be less than 21,000,000 BTC for various reasons, but
 * rather a sanity check. As this sanity check is used by consensus-critical
 * validation code, the exact value of the MAX_MONEY constant is consensus
 * critical; in unusual circumstances like a(nother) overflow bug that allowed
 * for the creation of coins out of thin air modification could lead to a fork.
 * */
static constexpr CAmount MAX_MONEY = 21000000 * COIN;
```

Do you think that the `COIN` constant is necessary at a consensus level, or is it a Bitcoin Core-specific abstraction?

Transaction validation

Transactions can originate from the P2P network, the wallet, RPCs or from tests.

Transactions which originate from the wallet, RPCs or individually from the P2P network (from a `NetMsgType::TX` message) will follow a validation pathway which includes adding them to the mempool. This implies passing both consensus and policy checks. See the sections on single_transactions and Multiple transactions to learn more about transaction validation via the mempool.

Transactions which are learned about in a new block from the P2P network (from a `NetMsgType::BLOCK` or `NetMsgType::BLOCKTXN` message) do not have to be added to the mempool and so do not have to pass policy checks. See the section transactions from blocks to learn more about transaction validation bypassing the mempool.

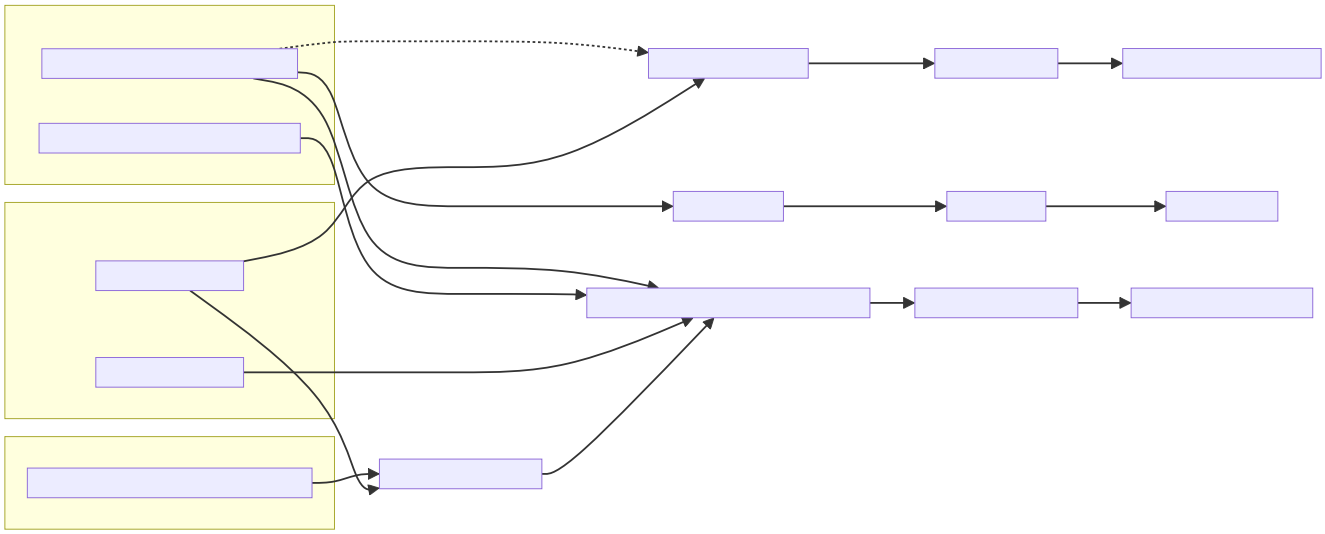

Figure 6. Transaction origination (excluding tests)

Dotted lines represent potential future upgrades

P2P network = Red
Wallet = Green
RPCs = Blue

For more information on `PeerManagerImpl` see PIMPL technique in the appendix.

Transactions are internally represented as either a `CTransaction`, a `CTransactionRef` (a shared pointer to a `CTransaction`) or in the case of packages a `Package` which is a `std::vector<CTransactionRef>`.

We can follow the journey of a transaction through the Bitcoin Core mempool by following glozow's notes on transaction "Validation and Submission to the Mempool". glozow details the different types of checks that are run on a new transaction before it's accepted into the mempool, as well as breaking down how these checks are different from each other: consensus vs policy, script vs non-script, contextual vs context-free.

The section on block validation describes the consensus checks performed on newly-learned blocks, specifically:

> Since v0.8, Bitcoin Core nodes have used a UTXO set rather than blockchain lookups to represent state and validate transactions. To fully validate new blocks nodes only need to consult their UTXO set and knowledge of the current consensus rules. Since consensus rules depend on block height and time (both of which can **decrease** during a reorg), they are recalculated for each block prior to validation.
>
> Regardless of whether or not transactions have already been previously validated and accepted to the mempool, nodes check block-wide consensus rules (e.g. total sigop cost, duplicate transactions, timestamps, witness commitments block subsidy amount) and transaction-wide consensus rules (e.g. availability of inputs, locktimes, and input scripts) for each block.
>
> Script checking is parallelized in block validation. Block transactions are checked in order (and coins set updated which allows for dependencies within the block), but input script checks are parallelizable. They are added to a work queue delegated to a set of threads while the main validation thread is working on other things. While failures should be rare - creating a valid proof of work for an invalid block is quite expensive - any consensus failure on a transaction invalidates the entire block, so no state changes are saved until these threads successfully complete.
>
> If the node already validated a transaction before it was included in a block, no consensus rules have changed, and the script cache has not evicted this transaction's entry, it doesn't need to run script checks again - it just uses the script cache!
>
> — glozow

The section from bitcoin-core-architecture on script verification also highlights how the script interpreter is called from at least 3 distinct sites within the codebase:

> - when the node receives a new transaction.
> - when the node wants to broadcast a new transaction.
> - when receiving a new block

Having considered both transactions that have entered into the mempool and transactions that were learned about in a new block we now understand both ways a transaction can be considered for validation.

As you read through the following sub-sections, consider whether making changes to them could affect policy or consensus.

Single transactions

`AcceptToMemoryPool()` (ATMP) is where the checks on single transactions occur before they enter the mempool.

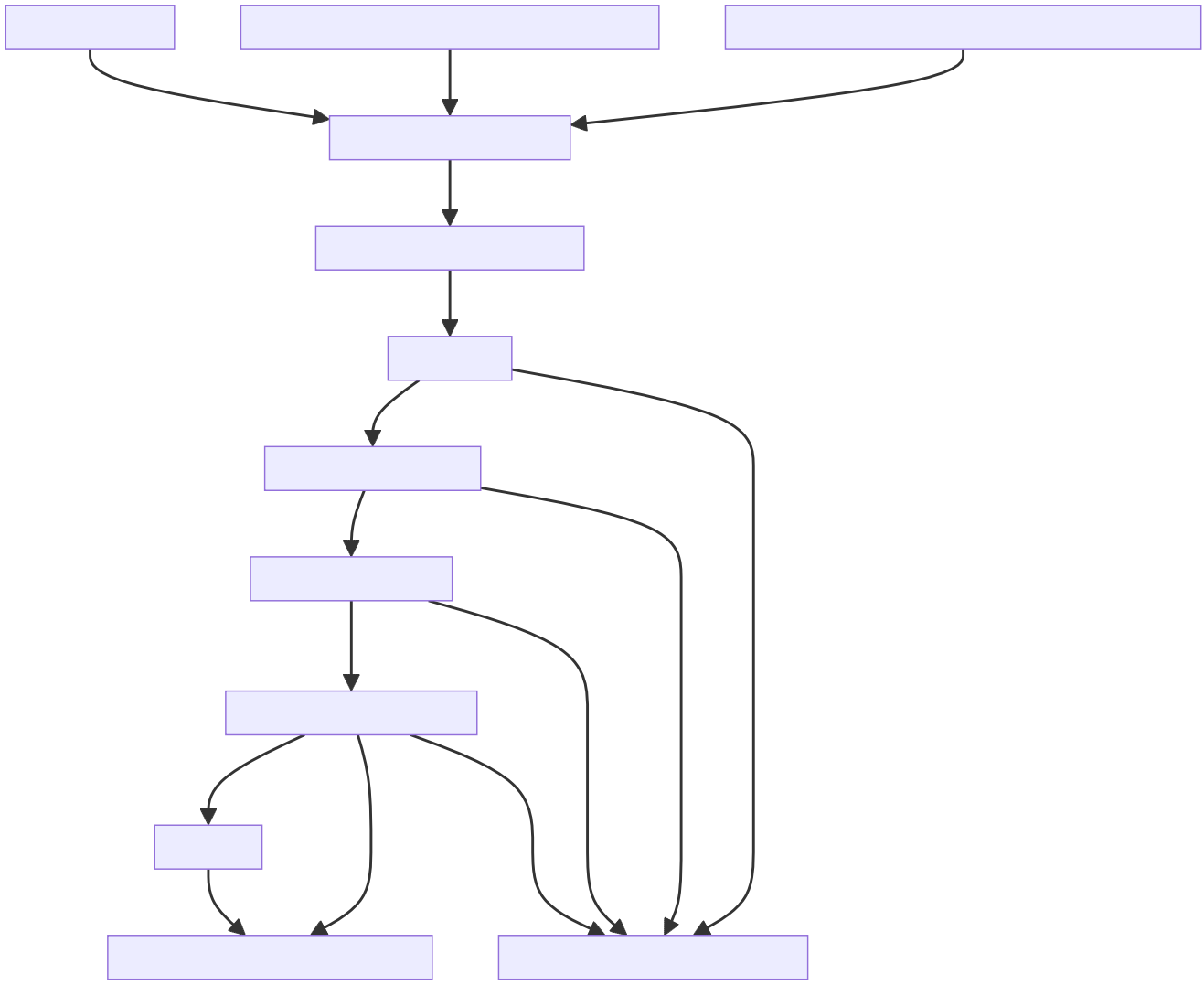

Figure 7. ATMP validation flow chart

You can see the calls to the various `*Checks()` functions in the call graph, and the order in which they are run.

Let's take a look inside `AcceptToMemoryPool()`'s inner function `AcceptSingleTransaction()` which handles running the checks:

src/validation.cpp

```
MempoolAcceptResult MemPoolAccept::AcceptSingleTransaction(const CTransactionRef& ptx,
ATMPArgs& args)
{
    AssertLockHeld(cs_main);
    LOCK(m_pool.cs); // mempool "read lock" (held through
```

```
GetMainSignals().TransactionAddedToMempool())

    Workspace ws(ptx);

    if (!PreChecks(args, ws)) return MempoolAcceptResult::Failure(ws.m_state);

    if (m_rbf && !ReplacementChecks(ws)) return MempoolAcceptResult::Failure(ws
.m_state);

    // Perform the inexpensive checks first and avoid hashing and signature
verification unless
    // those checks pass, to mitigate CPU exhaustion denial-of-service attacks.
    if (!PolicyScriptChecks(args, ws)) return MempoolAcceptResult::Failure(ws.
m_state);

    if (!ConsensusScriptChecks(args, ws)) return MempoolAcceptResult::Failure(ws
.m_state);

    // Tx was accepted, but not added
    if (args.m_test_accept) {
        return MempoolAcceptResult::Success(std::move(ws.m_replaced_transactions), ws
.m_vsize, ws.m_base_fees);
    }

    if (!Finalize(args, ws)) return MempoolAcceptResult::Failure(ws.m_state);

    GetMainSignals().TransactionAddedToMempool(ptx, m_pool.GetAndIncrementSequence());

    return MempoolAcceptResult::Success(std::move(ws.m_replaced_transactions), ws
.m_vsize, ws.m_base_fees);
}
```

We purposefully run checks in this order so that the least computationally-expensive checks are run first. This means that we can hopefully fail early and minimise CPU cycles used on invalid transactions.

If an attacker could force us to perform many expensive computations simply by sending us many invalid transactions then it would be inexpensive to bring our node to a halt.

Once `AcceptSingleTransaction` has acquired the `cs_main` and `m_pool.cs` locks it initializes a `Workspace` struct — a storage area for (validation status) state which can be shared by the different validation checks. Caching this state avoids performing the same computations multiple times and is important for performance. It will pass this workspace, along with the struct of `ATMPArgs` it received as argument, to the checks.

▼ *Click to see the code comments on why we hold two locks before performing consensus checks on transactions*

src/txmempool.h#CTxMemPool

```
/**
 * This mutex needs to be locked when accessing `mapTx` or other members
 * that are guarded by it.
 *
 * @par Consistency guarantees
 *
 * By design, it is guaranteed that:
 *
 * 1. Locking both `cs_main` and `mempool.cs` will give a view of mempool
 *    that is consistent with current chain tip (`::ChainActive()` and
 *    `CoinsTip()`) and is fully populated. Fully populated means that if the
 *    current active chain is missing transactions that were present in a
 *    previously active chain, all the missing transactions will have been
 *    re-added to the mempool and should be present if they meet size and
 *    consistency constraints.
 *
 * 2. Locking `mempool.cs` without `cs_main` will give a view of a mempool
 *    consistent with some chain that was active since `cs_main` was last
 *    locked, and that is fully populated as described above. It is ok for
 *    code that only needs to query or remove transactions from the mempool
 *    to lock just `mempool.cs` without `cs_main`.
 *
 * To provide these guarantees, it is necessary to lock both `cs_main` and
 * `mempool.cs` whenever adding transactions to the mempool and whenever
 * changing the chain tip. It's necessary to keep both mutexes locked until
 * the mempool is consistent with the new chain tip and fully populated.
 */
mutable RecursiveMutex cs;
```

The `Workspace` is initialized with a pointer to the transaction (as a `CTransactionRef`) and holds some additional information related to intermediate state.

We can look at the `ATMPArgs` struct to see what other information our mempool wants to know about in addition to transaction information.

ATMPArgs

`m_accept_time` is the local time when the transaction entered the mempool. It's used during the mempool transaction eviction selection process as part of `CTxMemPool::Expire()` where it is referenced by the name `entry_time`:

▼ *Click to see `entry_time` being used in `Expire()`*

src/txmempool.cpp#CTXMemPool::Expire()

```
int CTxMemPool::Expire(std::chrono::seconds time)
```

```
{
    AssertLockHeld(cs);
    indexed_transaction_set::index<entry_time>::type::iterator it = mapTx.get
<entry_time>().begin();
    setEntries toremove;
    while (it != mapTx.get<entry_time>().end() && it->GetTime() < time) {
        toremove.insert(mapTx.project<0>(it));
        it++;
    }
    setEntries stage;
    for (txiter removeit : toremove) {
        CalculateDescendants(removeit, stage);
    }
    RemoveStaged(stage, false, MemPoolRemovalReason::EXPIRY);
    return stage.size();
}
```

`m_bypass_limits` is used to determine whether we should enforce mempool fee limits for this transaction. If we are a miner we may want to ensure *our own* transactions would pass mempool checks, even if we don't attach a fee to them.

`m_test_accept` is used if we just want to run mempool checks to test validity, but not actually add the transaction into the mempool yet. This happens when we want to broadcast one of our own transactions, done by calling `BroadcastTransaction` from `node/transaction.cpp#BroadcastTransaction()` or from the `testmempoolaccept()` RPC.

If all the checks pass and this was not a `test_accept` submission then we will `MemPoolAccept::Finalize` the transaction, adding it to the mempool, before trimming the mempool size and updating any affected RBF transactions as required.

Multiple transactions (and packages)

TODO: This section should start from `AcceptPackage()` and flow through from there, including `AcceptMultipleTransactions()` as a sub-section.

It's possible to consider multiple transactions for validation together, via `AcceptMultipleTransactions()` found in *src/net_processing.cpp*. It's currently only available from tests (`test/tx_package_tests.cpp`) and the `testmempoolaccept` RPC (via `ProcessNewPackage()`), but the intention is for it to be available to packages received from the P2P network in the future.

This validation flow has been created for usage with Package Mempool Accept, which glozow has written up in a gist (archive).

The flow here is similar to `AcceptSingleTransaction()` in that we start by grabbing `cs_main` before initializing validation state and workspaces, however this time we use `PackageValidationState` and a vector of workspaces, `std::vector<Workspace>`. Each transaction therefore has it's own workspace but all transactions in the package share a single validation state. This aligns with the goal of either accepting or rejecting the entire package as a single entity.

Next come two `for` loops over the vector of workspaces (i.e. transactions). The first performs the `PreChecks()`, but this time also freeing up coins to be spent by other transactions in this package. This would not usually be possible (nor make sense) *within* an `AcceptTransaction()` flow, but within a package we want to be able to validate transactions who use as inputs, other transactions not yet added to our mempool:

```
    // Make the coins created by this transaction available for subsequent
transactions in the
    // package to spend. Since we already checked conflicts in the package and we
don't allow
    // replacements, we don't need to track the coins spent. Note that this logic will
need to be
    // updated if package replace-by-fee is allowed in the future.
    assert(!args.m_allow_bip125_replacement);
    m_viewmempool.PackageAddTransaction(ws.m_ptx);
```

If the `PreChecks` do not fail, we call `m_viewmempool.PackageAddTransaction()` passing in the workspace. This adds the transaction to a map in our Mempool called `std::unordered_map<COutPoint, Coin, SaltedOutpointHasher> m_temp_added;`, which is essentially a temporary cache somewhere in-between being validated and being fully added to the mempool.

TODO: Fix after adding section on `AcceptPackage`

After this first loop we perform `PackageMempoolChecks()` which first asserts that transactions are not already in the mempool, before checking the "PackageLimits".

PreChecks

The code comments for `PreChecks` give a clear description of what the PreChecks are for:

src/validation.cpp#MemPoolAccept::PreChecks()

```
// Run the policy checks on a given transaction, excluding any script checks.
// Looks up inputs, calculates feerate, considers replacement, evaluates
// package limits, etc. As this function can be invoked for "free" by a peer,
// only tests that are fast should be done here (to avoid CPU DoS).
```

The `PreChecks` function is very long but is worth examining to understand better which checks are undertaken as part of this first stage.

ReplacementChecks

During `PreChecks` the `m_rbf` bool will have been set to `true` if it is determined that this transaction would have to replace an existing transaction from our mempool. If this bool is set, then `ReplacementChecks` will be run. These checks are designed to check that BIP125 RBF rules are being adhered to.

PolicyScriptChecks

Following `ReplacementChecks` we initialise a `PrecomputedTransactionData` struct in the `Workspace` which will hold expensive-to-compute data that we might want to use again in subsequent validation steps.

▼ *Click to show the* `PrecomputedTransactionData` *struct*

script/interpreter.cpp

```
struct PrecomputedTransactionData
{
    // BIP341 precomputed data.
    // These are single-SHA256, see https://github.com/bitcoin/bips/blob/master/bip-
0341.mediawiki#cite_note-15.
    uint256 m_prevouts_single_hash;
    uint256 m_sequences_single_hash;
    uint256 m_outputs_single_hash;
    uint256 m_spent_amounts_single_hash;
    uint256 m_spent_scripts_single_hash;
    //! Whether the 5 fields above are initialized.
    bool m_bip341_taproot_ready = false;

    // BIP143 precomputed data (double-SHA256).
    uint256 hashPrevouts, hashSequence, hashOutputs;
    //! Whether the 3 fields above are initialized.
    bool m_bip143_segwit_ready = false;

    std::vector<CTxOut> m_spent_outputs;
    //! Whether m_spent_outputs is initialized.
    bool m_spent_outputs_ready = false;

    PrecomputedTransactionData() = default;

    template <class T>
    void Init(const T& tx, std::vector<CTxOut>&& spent_outputs);

    template <class T>
    explicit PrecomputedTransactionData(const T& tx);
};
```

Next we call `PolicyScriptChecks()` passing in the same `ATMPArgs` and `Workspace` that we used with PreChecks. This is going to check the transaction against our individual node's policies.

Note that local node policies are not necessarily consensus-binding, but are designed to help prevent resource exhaustion (e.g. DoS) on our node.

See the transaction validation and consensus in bitcoin core sections for more information on the differences between policy and consensus.

`PolicyScriptChecks()` starts with initialisation of the transaction into a `CTransaction`, before

beginning to check the input scripts against the script flags.

src/validation.cpp#PolicyScriptChecks

```
bool MemPoolAccept::PolicyScriptChecks(const ATMPArgs& args, Workspace& ws,
PrecomputedTransactionData& txdata)
{
    const CTransaction& tx = *ws.m_ptx;
    TxValidationState& state = ws.m_state;

    constexpr unsigned int scriptVerifyFlags = STANDARD_SCRIPT_VERIFY_FLAGS;

    // Check input scripts and signatures.
    // This is done last to help prevent CPU exhaustion denial-of-service attacks.
    if (!CheckInputScripts(tx, state, m_view, scriptVerifyFlags, true, false, txdata))
{ ①
        // SCRIPT_VERIFY_CLEANSTACK requires SCRIPT_VERIFY_WITNESS, so we
        // need to turn both off, and compare against just turning off CLEANSTACK
        // to see if the failure is specifically due to witness validation.
        TxValidationState state_dummy; // Want reported failures to be from first
CheckInputScripts
        if (!tx.HasWitness() && CheckInputScripts(tx, state_dummy, m_view,
scriptVerifyFlags & ~(SCRIPT_VERIFY_WITNESS | SCRIPT_VERIFY_CLEANSTACK), true, false,
txdata) &&
                !CheckInputScripts(tx, state_dummy, m_view, scriptVerifyFlags &
~SCRIPT_VERIFY_CLEANSTACK, true, false, txdata)) {
            // Only the witness is missing, so the transaction itself may be fine.
            state.Invalid(TxValidationResult::TX_WITNESS_STRIPPED,
                    state.GetRejectReason(), state.GetDebugMessage());
        }
        return false; // state filled in by CheckInputScripts
    }

    return true;
}
```

① Calling `CheckInputScripts()` involves ECDSA signature verification and is therefore computationally expensive.

If the script type is SegWit an additional round of checking is performed, this time including the `CLEANSTACK` rule. The call(s) flag `cacheSigStore` as `true`, and `cacheFullScriptStore` as `false`, which means that matched signatures will be persisted in the cache, but matched full scripts will be removed.

ConsensusScriptChecks

If the `PolicyScriptChecks` return `true` we will move on to consensus script checks, again passing in the same `ATMPArgs`, `Workspace` and now `PrecomputedTransactionData` that we used previously with `PolicyScriptChecks`.

The main check in here is `CheckInputsFromMempoolAndCache()` which is going to compare all the transaction inputs to our mempool, checking that they have not already been marked as spent. If the coin is not already spent, we reference it from either the UTXO set or our mempool, and finally submit it through `CheckInputScripts()` once more, this time caching both the signatures and the full scripts.

▼ *Click to show* `CheckInputsFromMempoolAndCache()`

src/validation.cpp#CheckInputsFromMempoolAndCache

```
/**
* Checks to avoid mempool polluting consensus critical paths since cached
* signature and script validity results will be reused if we validate this
* transaction again during block validation.
* */
static bool CheckInputsFromMempoolAndCache(const CTransaction& tx,
TxValidationState& state,
                const CCoinsViewCache& view, const CTxMemPool& pool,
                unsigned int flags, PrecomputedTransactionData& txdata,
CCoinsViewCache& coins_tip)
                EXCLUSIVE_LOCKS_REQUIRED(cs_main, pool.cs)
{
    AssertLockHeld(cs_main);
    AssertLockHeld(pool.cs);

    assert(!tx.IsCoinBase());
    for (const CTxIn& txin : tx.vin) {
        const Coin& coin = view.AccessCoin(txin.prevout);

        // This coin was checked in PreChecks and MemPoolAccept
        // has been holding cs_main since then.
        Assume(!coin.IsSpent());
        if (coin.IsSpent()) return false;

        // If the Coin is available, there are 2 possibilities:
        // it is available in our current ChainstateActive UTXO set,
        // or it's a UTXO provided by a transaction in our mempool.
        // Ensure the scriptPubKeys in Coins from CoinsView are correct.
        const CTransactionRef& txFrom = pool.get(txin.prevout.hash);
        if (txFrom) {
            assert(txFrom->GetHash() == txin.prevout.hash);
            assert(txFrom->vout.size() > txin.prevout.n);
            assert(txFrom->vout[txin.prevout.n] == coin.out);
        } else {
            assert(std::addressof(::ChainstateActive().CoinsTip()) == std::
addressof(coins_tip));
            const Coin& coinFromUTXOSet = coins_tip.AccessCoin(txin.prevout);
            assert(!coinFromUTXOSet.IsSpent());
            assert(coinFromUTXOSet.out == coin.out);
        }
    }
```

```
    // Call CheckInputScripts() to cache signature and script validity against
current tip consensus rules.
    return CheckInputScripts(tx, state, view, flags, /* cacheSigStore = */ true, /*
cacheFullSciptStore = */ true, txdata);
}
```

PackageMempoolChecks

`PackageMempoolChecks` are designed to "Enforce package mempool ancestor/descendant limits (distinct from individual ancestor/descendant limits done in PreChecks)". They take a vector of `CTransactionRef`s and a `PackageValidationState`.

Again we take two locks before checking that the transactions are not in the mempool. Any transactions which are part of the package and were in the mempool will have already been removed by `MemPoolAccept::AcceptPackage()`.

Finally we check the package limits, which consists of checking the {ancestor|descendant} {count|size}. This check is unique to packages because we can now add descendants whose ancestors would not otherwise qualify for entry into our mempool with their low effective fee rate.

Finalize

Provided that consensus script checks pass and this was not a test ATMP call, we will call `Finalize()` on the transaction. This will remove any conflicting (lower fee) transactions from the mempool before adding this one, finishing by trimming the mempool to the configured size (default: `static const unsigned int DEFAULT_MAX_MEMPOOL_SIZE = 300;` MB). In the event that **this** transaction got trimmed, we ensure that we return a `TxValidationResult::TX_MEMPOOL_POLICY, "mempool full"` result.

Transactions from blocks

Transactions learned about from blocks:

- Might not be present in our mempool
- Are not being considered for entry into our mempool and therefore do not have to pass policy checks
- Are only subject to consensus checks

This means that we can validate these transactions based only on our copy of the UTXO set and the data contained within the block itself. We call `ProcessBlock()` when processing new blocks received from the P2P network (in *net_processing.cpp*) from net message types: `NetMsgType::CMPCTBLOCK`, `NetMsgType::BLOCKTXN` and `NetMsgType::BLOCK`.

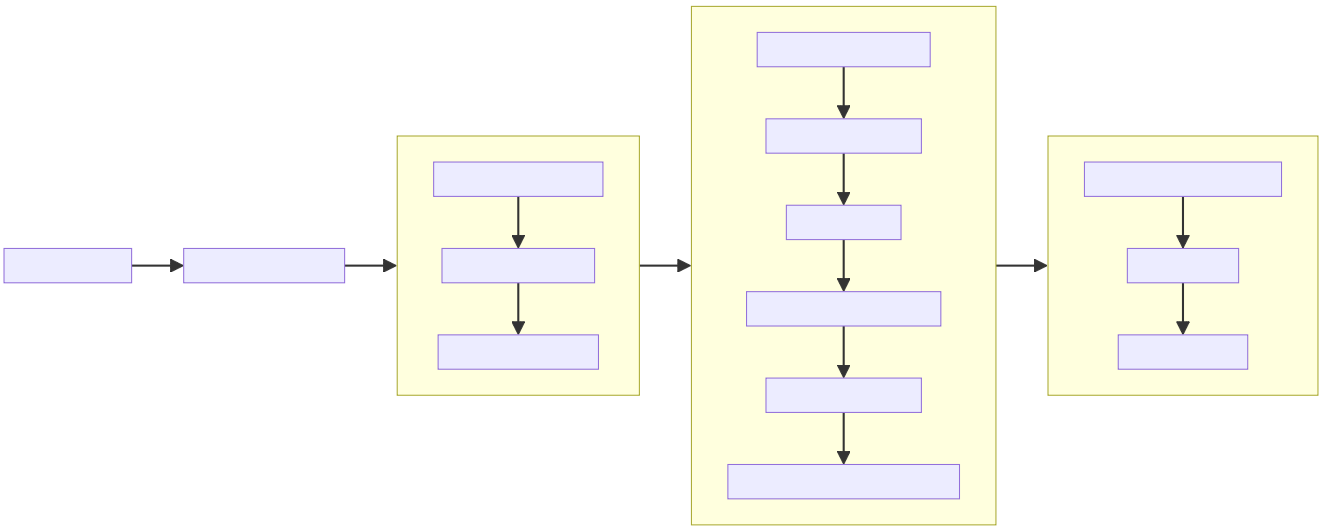

Figure 8. Abbreviated block transaction validation

The general flow of `ProcessBlock()` is that will call `CheckBlock()`, `AcceptBlock()` and then `ActivateBestChain()`. A block which has passed successfully through `CheckBlock()` and `AcceptBlock()` has **not** passed full consensus validation.

`CheckBlock()` does some cheap, context-independent structural validity checks, along with (re-)checking the proof of work in the header, however these checks just determine that the block is "valid-enough" to proceed to `AcceptBlock()`.

Once the checks have been completed, the `block.fChecked` value is set to `true`. This will enable any subsequent calls to this function *with this block* to be skipped.

`AcceptBlock()` is used to persist the block to disk so that we can (validate it and) add it to our chain immediately, use it later, or discard it later. `AcceptBlock()` makes a second call to `CheckBlock()` but because `block.fChecked` was set to `true` on the first pass this second check will be skipped.

`AcceptBlock()` contains an inner call to `CheckBlock()` because it can also be called directly by `CChainState::LoadExternalBlockFile()` where `CheckBlock()` will not have been previously called.

It also now runs some contextual checks such as checking the block time, transaction lock times (transaction are "finalized") and witness commitments are either non-existent or valid (link). After this the block will be serialized to disk.

At this stage we might still be writing blocks to disk that will fail full consensus checks. However, if they have reached here they have passed proof of work and structural checks, so consensus failures may be due to us missing intermediate blocks, or that there are competing chain tips. In these cases this block may still be useful to us in the future.

Once the block has been written to disk by `AcceptBlock()` full validation of the block and its transactions begins via `CChainState::ActivateBestChain()` and its inner call to `ActivateBestChainStep()`.

Multiple chains

TODO: Reorgs, undo data, `DisconnectBlock`

Bitcoin nodes should ultimately converge in consensus on the most-work chain. Being able to track and monitor multiple chain (tips) concurrently is a key requirement for this to take place. There are a number of different states which the client must be able to handle:

1. A single, most-work chain being followed
2. Stale blocks learned about but not used
3. Full reorganisation from one chain tip to another

`BlockManager` is tasked with maintaining a tree of all blocks learned about, along with their total work so that the most-work chain can be quickly determined.

`CChainstate` (renamed to `Chainstate` in v24.0) is responsible for updating our local view of the best tip, including reading and writing blocks to disk, and updating the UTXO set. A single `BlockManager` is shared between all instances of `CChainState`.

`ChainstateManager` is tasked with managing multiple `CChainState`s. Currently just a "regular" IBD chainstate and an optional snapshot chainstate, which might in the future be used as part of the assumeUTXO project.

When a new block is learned about (from `src/net_processing.cpp`) it will call into `ChainstateManager`s `ProcessNewBlockHeaders` method to validate it.

Responsible Disclosure

Bitcoin Core has a defined process for reporting security vulnerabilities via it's responsible disclosure process. This is detailed in SECURITY.md.

Bugs which would need to be disclosed by following this process are generally those which could result in a consensus-failure, theft of funds, or creation of additional supply tokens (new coin issuance). If bugs of this nature are posted publicly then inevitably one or more persons will try to enact them, possibly causing severe harm or loss to one or many people.

If you would like to learn more about the responsible disclosure process and why it's so important for Bitcoin Core, you can read the following:

1. Responsible disclosure in the era of cryptocurrencies
2. Responsible Vulnerability Disclosure in Cryptocurrencies

Exercises

1. *What is the difference between contextual and context-free validation checks?*

   ▼ *Click for answer*

   Contextual checks require some knowledge of the current "state", e.g. ChainState, chain tip or

UTXO set.

Context-free checks only require the information required in the transaction itself.

For more info, see glozow's notes on transaction "Validation and Submission to the Mempool".

2. *What are some examples of each?*
   ▼ *Click for answer*

   context-free:

   1. `tx.isCoinbase()`
   2. 0 ≤ tx_value ≤ MAX_MONEY
   3. tx not overweight

   contextual:

   1. check inputs are available

3. *In which function(s) do UTXO-related validity checks happen?*
   ▼ *Click for answer*
   `ConnectBlock()`

4. *What type of validation checks are `CheckBlockHeader()` and `CheckBlock()` performing?*
   ▼ *Click for answer*

   context-free

5. *Which class is in charge of managing the current blockchain?*
   ▼ *Click for answer*
   `ChainstateManager()`

6. *Which class is in charge of managing the UTXO set?*
   ▼ *Click for answer*
   `CCoinsViews()`

7. *Which functions are called when a longer chain is found that we need to re-org onto?*
   TODO

8. *Are there any areas of the codebase where the same consensus or validation checks are performed twice?*
   ▼ *Click for answer*

   Again see glozow's notes for examples

9. *Why does `CheckInputsFromMempoolAndCache` exist?*
   ▼ *Click for answer*

   To prevent us from re-checking the scripts of transactions already in our mempool during consensus validation on learning about a new block

10. *Which function(s) are in charge of validating the merkle root of a block?*

    ▼ *Click for answer*

    `BlockMerkleRoot()` and `BlockWitnessMerkleRoot()` construct a vector of merkle leaves, which is then passed to `ComputeMerkleRoot()` for calculation.

11. *Can you find any evidence (e.g. PRs) which have been made in an effort to modularize consensus code?*

    ▼ *Click for answer*

    A few examples: PR#10279, PR#20158

12. *What is the function of* `BlockManager()`*?*

    ▼ *Click for answer*

    It manages the current most-work chaintip and pruning of unneeded blocks (`*.blk`) and associated undo (`*.rev`) files

13. *What stops a malicious node from sending multiple invalid headers to try and use up a nodes' disk space? (hint: these might be stored in* `BlockManager.m_failed_blocks`*)*

    ▼ *Click for answer*

    Even invalid headers would need a valid proof of work which would be too costly to construct for a spammer

14. *Which functions are responsible for writing consensus-valid blocks to disk?*

    ▼ *Click for answer*

    `src/node/blockstorage.h#SaveBlockToDisk`

15. *Are there any other components to Bitcoin Core which, similarly to the block storage database, are not themselves performing validation but can still be consensus-critical?*

    Not sure myself, sounds like an interesting question though!

16. *In which module (and class) is signature verification handled?*

    ▼ *Click for answer*

    `src/script/interpreter.cpp#BaseSignatureChecker`

17. *Which function is used to calculate the Merkle root of a block, and from where is it called?*

    ▼ *Click for answer*

    `src/consensus/merkle.cpp#ComputeMerkleRoot` is used to compute the merkle root.

    It is called from `src/chainparams.cpp#CreateGenesisBlock`, `src/miner.cpp#IncrementExtraNonce` & `src/miner.cpp#RegenerateCommitments` and from `src/validation.cpp#CheckBlock` to validate incoming blocks.

18. *Practical question on Merkle root calculation*

    TODO, add more Exercises

    - Modify the code which is used to add new opcodes to a `CScript` without breaking consensus.

Wallet

This section has been updated to Bitcoin Core @ v23.0

Bitcoin Core includes an optional wallet component. The wallet allows users to make and receive transactions using their own node, so that they can validate incoming payment against their own node.

The wallet component has the following general aims:

1. Have best-in-class security
    - Be extremely well tested
    - Be reviewed by competent developers
2. Have good privacy by default
3. Be smart about coin selection with respect to:
    - Transaction fees
    - Privacy
4. Implement state-of-the-art features:
    - Taproot
    - Wallet descriptors
    - Miniscript
5. Be backwards compatible with old (Bitcoin Core) wallet files where possible

Wallets can be one of two types, "legacy" or "descriptor".

Bitcoin Core moved to descriptor wallets as they are unambiguous as to which public keys and scripts should be used.

They also simplify backups and make im/ex-porting wallet keys into other software less error-prone.

Wallet overview

Blockchain Commons provides some examples of Setting up a wallet using the `bitcoin-cli tool.

Wallet Database

Wallets are stored on disk as databases, either using Berkeley Database (BDB) or sqlite format.

The version of BDB we used for the wallet is unmaintained, so new wallets should

> prefer sqlite format

The wallet is stored on disk as a Key Value store.

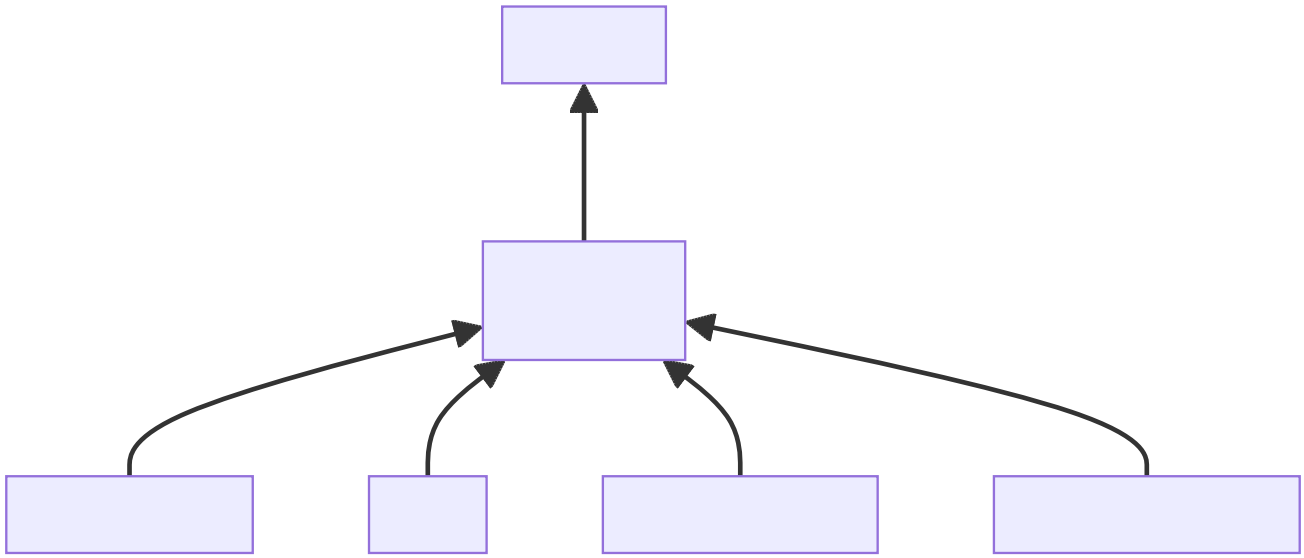

Figure 9. Wallet database

These are some of the records which help us regenerate a descriptor wallet (populating a `DescriptorScriptPubKeyMan` (DSPKM)) from the database:

```
// walletdb.cpp
const std::string WALLETDESCRIPTOR{"walletdescriptor"};
const std::string WALLETDESCRIPTORCACHE{"walletdescriptorcache"};
const std::string WALLETDESCRIPTORLHCACHE{"walletdescriptorlhcache"};
const std::string WALLETDESCRIPTORCKEY{"walletdescriptorckey"};
const std::string WALLETDESCRIPTORKEY{"walletdescriptorkey"};
```

For Legacy wallets (populating a `LegacyScriptPubKeyMan` (LSPKM)) we use the records with `*.KEY` & `SCRIPT`.

Wallet metadata may include a `tipLocator` — the most recent tip — and a wallet `version` which is used in database upgrades.

To load the wallet we read the database by iterating the records and loading them to `CWallet`, using `ReadKeyValue()` to deserialize.

Table 11. Loading wallet records from the database

Record	Load point
`DBKeys::TX`	(Bitcoin) transactions end up in `mapWallet` via the call to `pwallet->LoadToWallet(hash, fill_wtx)`
`DBKeys::KEY`	Keys for legacy wallets are loaded into `CKey` or `Key`, then read into the appropriate SPKM (or one is created and keys added to it) using `pwallet->GetOrCreateLegacyScriptPubKeyMan()`.

Record	Load point

`DBKeys::WALLETDESCRIPTOR`
`DBKeys::WALLETDESCRIPTORCACHE`
`DBKeys::WALLETDESCRIPTORLHCACHE`
`DBKeys::WALLETDESCRIPTORKEY`
`DBKeys::WALLETDESCRIPTORCKEY`
	Descriptor wallet information generally goes into `DescriptorScriptPubKeyMan`.

`DBKeys::NAME` `DBKeys::PURPOSE`	Addresses go into `m_address_book`

You can see where all the other DB records are deserialized to by examining the `ReadKeyValue()` function.

The various `*ScriptPubkeyMan` objects are all owned by the `CWallet` instance eventually, however `LegacyScriptPubKeyMan` is both created and owned by `CWallet`, whereas `DescriptorScriptPubKeyMan` is created externally to `CWallet` and only after loading exists in the `CWallet` context.

Note that `TxSpends` is **not** tracked in the wallet database (and loaded at startup), but instead is rebuilt from scratch because it's fast to do so and we must reload every transaction anyway, so it's not much more work to regenerate `TxSpends` at the same time.

Key-type classes in the wallet

There are a number of `Key` classes in the wallet code and keeping track of their functions can be confusing at times due to naming similarities. Below are listed some of these classes along with some primary functions.

`CKey`	An encapsulated private key. Used for signing and deriving child keys.
`CKeyID`	A *reference* to a `CKey` by the hash160 of its pubkey. Used for key lookups when fetching keys e.g. for signing.
`CPrivKey`	A serialized (OpenSSL format) private key with associated parameters. Used to read/write private keys to/from the database.
`CPubKey`	A public key. Used in many places.
`CExtKey`	An extended public key (includes private key and chaincode). Used for deriving BIP32 child keys.
`CMasterKey`	Contains an encryption salt `vchSalt` and a randomly generated encryption key `vchCryptedKey`. The `CMasterKey` object itself is what is encrypted by the user's passphrase and the inner `vchCryptedKey` is what is used to en/de-crypt the wallet keys.
`CKeyingMaterial`	Plain text which is to be encrypted or has been decrypted using the `CMasterKey`.

`CKeyPool`	A single key which has been taken from a `CWallet`'s keypool for use. `CKeyPool` keys are stored in the wallet database.
`CryptedKeyMap`	A map of `CKeyID` to a pair of (`CPubKey` + an encrypted private key). Used to lookup keys (by `CKeyID`) when the wallet is encrypted.

Encryption

There is encryption in the wallet code, but it is found within both `CWallet` and `*ScriptPubKeyMan` so is not yet well encapsulated.

When encryption is enabled secret data must only ever reside in memory and should **never** be written to disk.

When you unlock an encrypted wallet you can set a `timeout`. When the timeout expires secret data is deleted from memory, and the wallet "re-locked".

Decrypting the wallet

As detailed in Key types, the `CMasterKey.vchCryptedKey` is the actual secret key used to en/de-crypt the keys in the wallet.

`CWallet` stores a `CMasterKey`, which is **not** a master private key. The `CMasterKey` is encrypted by the user's passphrase.

When the user changes their passphrase, they are only changing the encryption applied to the `CMasterKey`, the inner `vchCryptedKey` is not changed. This means that we do not have to read all items in the wallet database, decrypt them with the old key, encrypt them with the new key, and then write them, back to the database again. Instead, we only have to change the encryption applied to the `CMasterKey`, which is much less error-prone, and more secure.

Each `CWallet` has a map of `CMasterKey`s and when unlock is called it will try each one to see if it can decrypt and then unlock the wallet.

Encrypting the wallet

Only private keys are encrypted. This allows us to watch for new transactions *without* having to decrypt the wallet as each new block|transaction arrives.

Decrypting the Bitcoin Core wallet requires the user to enter their passphrase, so is not convenient to do at every new block.

When encrypting a wallet, a `CMasterKey` encryption key is generated, which is then sent to the `ScriptPubKeyMan` to encrypt using its `.Encrypt()` method.

Once the wallet is encrypted for the first time, we re-generate all of our keys. This is to avoid the wallet using things which were not "born encrypted" in the future. For `LegacyScriptPubKeyMan` this means creating a new HD seed, and for `DescriptorScriptPubKeyMan` 8 new descriptors.

If the wallet has already been used before — while it existed in un-encrypted state — the old

`ScriptPubKeyMan`'s are retained and so remain usable, but are not marked as `active`. The wallet will switch to the new SPKM after encryption has completed by marking the new SPKM as `active`.

We take extra care during the encryption phase to either complete atomically or fail. This includes database writes where we don't want to write half and crash, for example. Therefore we will throw an assertion if the write fails.

When you instruct a BDB database to delete a record, they are actually kept but "marked as" deleted, and *might* be fully deleted some time in the future.

This is not appropriate for our use case, for example when asking the DB to delete private keys after the wallet is encrypted for the first time. Therefore we use some hacks so that when we request deletion of unencrypted private keys from the DB, they are properly deleted immediately and not "marked as" deleted.

When encryption is enabled secret data must only ever exist in decrypted form in memory.

See #27080 for details on how the master key was not always cleared fully from memory after the wallet was locked.

Transaction tracking

When we learn about a new block the `BlockConnected` signal is fired after successful validation. This prompts the wallet to iterate all inputs and outputs, calling `IsMine()` on all of them. As part of the check, we loop over the wallet's `scriptPubkeyMan`s to check if any of the scripts belong to us.

If a script does belong to us, it will be inserted into `mapWallet` along with some metadata related to the time. `mapWallet` contains all the transactions the wallet is interested in, including received and sent transactions.

When we load a wallet into memory, we iterate all `TxSpends`. `TxSpends` stores wallet transactions which were already spent and confirmed.

Therefore, when the wallet needs to select coins to spend, it can select from the coins:

`mapWallet - TxSpends - notMine`

Calculating a balance

For balance calculation we iterate `mapWallet` and add values to a `Balance` struct.

```
struct Balance {
    CAmount m_mine_trusted{0};           //!< Trusted, at depth=GetBalance.min_depth
or more
    CAmount m_mine_untrusted_pending{0}; //!< Untrusted, but in mempool (pending)
    CAmount m_mine_immature{0};          //!< Immature coinbases in the main chain
    CAmount m_watchonly_trusted{0};
    CAmount m_watchonly_untrusted_pending{0};
```

```
    CAmount m_watchonly_immature{0};
};
```

We do some caching during iteration so that we avoid re-calculating the same values for multiple transactions.

Wallet balance terminology

`debit`	amount out
`credit`	amount in
`availableCredit`	amount available to send out (not dirty or immature)

Calculating the above requires using `TxSpends` and `IsMine`.

When a new transaction involving the wallet takes place, really what happens is that it's marked as `DIRTY`, which deletes the cached entry for the parent transaction. This means that the next time `GetBalance()` is called, `debit` is recalculated correctly. This Bitcoin Core PR review club goes into more detail on coins being marked as `DIRTY` and `FRESH` in the cache.

`TxSpends` is calculated by looking at the outpoints in the transaction itself.

`COutput` vs `COutPoint`

`COutPoint`	a pair of `txid : index`, useful when you want to know which UTXO an input spends.
`COutput`	created for coin selection and contains the entire previous UTXO (script, amount), along with helpers for calculating fees and effective value.

`COutput`s are ephemeral — we create them, perform another operation with them and discard them. They are stored in `availableCoins` which is recreated when calling functions such as `GetAvailableBalance()`, `ListCoins()` and `CreateTransactionInternal()`.

In a spending transaction all inputs have their corresponding `OutPoints`, and we map these to spending transactions in `TxSpends`.

We assume anything (i.e. transactions) that reach the wallet have already been validated by the node and we therefore blindly assume that it is valid in wallet code.

If a transaction is our own we check for validity with `testMempoolAccept` before submitting to the P2P network.

IsMine

For DSPKM running `IsMine()` is really simple: descriptors generate a list of ScriptPubKeys, and, if the SPK we are interested in is in the list, then it's ours.

`IsMine` returns an enum. This is used as a return value, a filter and set of flags simultaneously. There is more background on the general `IsMine` semantics in the v0.21.0 release notes.

LSPKM can have watch-only and spendable flags set at the same time, but DSPKM is either or, because descriptor wallets do not allow mixtures of spendable and watch-only keys in the same SPKM. Because Legacy wallets are all key-based, we will need to see if a script *could have been generated by one of our keys*; what type of script it is; and if we have a (private) key for it.

For Legacy watch-only wallets we simply check "do we have this script stored as a script?" (where `CScripts` in the database are our watch-only scripts)". If we don't have a `CKey` for a script but it exists in `mapScripts` then it's implicitly watch-only.

A problem with this current method of `IsMine` for legacy wallets is that it's tough to figure out what your wallet considers "Mine" — it's probably a finite set, but maybe not…

Another consideration is that the LSPKM `IsMine` includes P2PK outputs — which don't have addresses! This un-enumerability can be an issue in migration of Legacy to Descriptor wallets.

There is also the possibility that someone can mutate address to different address type and you will still see it as `IsMine`. E.g. mutate P2PK into P2PKH address and wallet will still detect.

With descriptors we only look for scripts explicitly. With descriptor wallets `IsMine` might not recognise script hashes from scripts, because it was not told to watch for them and consider them as belonging to it.

We use the `IsMine` filters in many places, primarily to distinguish between spendable and watch-only:

`IsMine::All`
: spendable and watch-only (use for legacy wallet)

`IsMine::Used`
: not used by `IsMine`, but instead used as a filter for tracking when addresses have been reused.

> PR 19602 enables migration of legacy wallets → descriptor wallets from Bitcoin Core version 24.0. Although legacy wallets are now effectively end of life it's still relevant to have documentation for legacy wallets.

See the section on how wallets determine whether transactions belong to them using the enum for more in-depth information.

Conflict tracking

Conflict tracking is related to changing the state as the mempool tells us about conflicting

transactions.

`mapTxSpends` is a multimap which permits having the same `COutPoint` mapping to *two* transactions. (i.e. two transactions spending the same input) This is how we can tell if things are conflicted: look up an outpoint and check to see how many transactions are there, if > 1 then we know that there was a conflict.

If there is a conflict we can look up the wallet transaction and see what state it's in, and we can be sure about whether it is currently or previously conflicted.

Conflict tracking is particularly relevant for coin selection…

Coin selection

See Bitcoin Optech for more information on coin selection. There is a section digging deeper into the coin selection code found below. To select inputs to a transaction our primary considerations are privacy and fees.

The below sections form an overview of creating a transaction via `CreateTransactionInternal()`.

`AvailableCoins()`

The gist of how we generate a list of coins available to spend (via `AvailableCoins()`) is that we iterate `mapWallet` and check for coins that:

- Are not immature coinbase outputs
- Are not conflicted
- Must be at least in our mempool
- Not currently replacing or being replaced by another transaction
- Are not locked
- Are `IsMine`
- Are `spendable`

…and return them as a `std::vector<COutput>`.

`GroupOutputs()`

Once we have this vector of coins `GroupOutputs()` will turn them into `OutputGroup`s. An `OutputGroup` consists of outputs with the same script, i.e. "coins sent to the same address".

`selectCoins()`

If you manually choose inputs, it will add outputs to the transaction automatically. It tries first to make sure that all outputs selected have 6 confirmations, if unsuccessful it then tries again with 1 confirmation as the lower bound.

For change outputs it starts with 1 confirmation and then again with 0. If this is still unsuccessful it increases the number of ancestors and descendants that unconfirmed change can have.

`AttemptSelection()`

This function is orchestrating the Output group creation, and then the coin selection. Currently, this is always based on the waste metric.

It is using 3 algorithms and then selecting the "best" of the three (based on the waste metric):

1. Branch n bound (bnb)
2. Knapsack
3. Single Random Draw (SRD)

There is currently an idea that a limited SRD could replace Knapsack in the future. Due to this plan for removal, it would not make sense to focus development effort on improving the Knapsack algorithm at this time.

Transaction creation

Once the coins have been selected they are returned back to `CreateTransactionInternal()`, which will create the final transaction.

Right now when we determine the change output, we don't use what `selectionResult` says the change output should be. What we actually do is make the tx with in? outputs and set the change amount to be the sum inputs-outputs, so the change amount includes the transaction fee. To get the correct change amount we now calculate the size of this after signing, we use dummysigner to add a dummy signature (74 0's and the correct script), and now we can calculate the correct fee. We reduce that fee from the change output amount, and if this now goes below **some threshold?** (the "cost of change" thing from BnB) or if it is dust we drop the change output and add it's value to the fee.

So now we have an unsigned tx which we need to sign.

Signing

We pass the tx to `CWallet::SignTransaction()` which will call `IsMine()` on each input to figure out which ScriptPubKeyMan (spkman) owns that input, then ask the spkman to fetch its `SigningProviders` to provide the signer which can sign the transaction, and return that to us.

With PSBTs we have the `fillPSBT()` method in `CWallet` which calls `*ScriptPubKeyMan::fillPSBT()`. We do this because we can add previous UTXOs due to transaction tracking; the SPKM adds the scripts and key derivation paths and will then optionally sign.

Separation of wallet and node

Both the `bitcoind` and `bitcoin-qt` programs use the same source code for the wallet component. `bitcoin-qt` is not therefore a gui frontend for `bitcoind` but a stand-alone binary which happens to share much of the same code. There has been discussion since at least as early as 2014 about splitting wallet code out from the rest of the codebase, however this has not been completed yet.

The Process Separation project is tracking development working towards separating out node,

wallet and GUI code even further. In the mean time developers have preferred to focus on improving the organisation of the (wallet) source code within the project and to focus on making wallet code more asynchronous and independent of node code, to avoid locking the node while wallet code-paths are executing.

Wallet interfaces

In order to facilitate code separation, distinct interfaces between the node and the wallet have been created:

- The node holds a `WalletImpl` interface to call functions on the wallet.
- The wallet holds a `ChainImpl` interface to call functions on the node.
- The node notifies the wallet about new transactions and blocks through the `CValidationInterface`.

For more information on `*Impl` classes see PIMPL technique in the appendix.

Wallet component initialisation

The wallet component is initialized via the `WalletInitInterface` class as specified in *src/walletinitinterface.h*. The member functions are marked as virtual in the `WalletInitInterface` definition, indicating that they are going to be overridden later by a derived class.

Both *wallet/init.cpp* and *dummywallet.cpp* include derived classes which override the member functions of `WalletInitInterface`, depending on whether the wallet is being compiled in or not.

The primary *src/Makefile.am* describes which of these modules is chosen to override: if `./configure` has been run with the wallet feature enabled (default), then *wallet/init.cpp* is added to the sources, otherwise (`./configure --disable-wallet`) *dummywallet.cpp* is added:

src/Makefile.am

```
if ENABLE_WALLET
libbitcoin_server_a_SOURCES += wallet/init.cpp
endif
if !ENABLE_WALLET
libbitcoin_server_a_SOURCES += dummywallet.cpp
endif
```

src/walletinitinterface.h declares the global `g_wallet_init_interface` which will handle the configured `WalletInitInterface`.

The wallet interface is created when the `Construct()` method is called on the `g_wallet_init_interface` object by `AppInitMain()` in *init.cpp*. `Construct` takes a reference to a `NodeContext` as argument, and then checks that the wallet has not been disabled by a runtime argument before calling `interfaces::MakeWalletLoader()` on the node. This initialises a new `WalletLoader` object which is then added to the `node` object, both to the general list of `node.chain_clients` (wallet processes or other clients which want chain information from the node)

in addition to being assigned as the unique `node.wallet_client` role, which specifies the particular `node.chain_client` that should be used to load or create wallets.

The `NodeContext` struct is defined as the following:

src/node/context.h

> ...contains references to chain state and connection state.
>
> ...used by init, rpc, and test code to pass object references around without needing to declare the same variables and parameters repeatedly, or to use globals... The struct isn't intended to have any member functions. It should just be a collection of references that can be used without pulling in unwanted dependencies or functionality.

Wallets and program initialisation

Wallets can optionally be loaded as part of main program startup (i.e. from *src/init.cpp*). Any wallets loaded during the life cycle of the main program are also unloaded as part of program shutdown.

Specifying wallets loaded at startup

Wallet(s) to be loaded as part of program startup can be specified by passing `-wallet=` or `-walletdir=` arguments to `bitcoind`/`bitcoin-qt`. If the wallet has been compiled in but no `-wallet*=` arguments have been passed, then the default wallet directory (*$datadir/wallets*) will be checked as per `GetWalletDir()`:

Wallets can also be loaded after program startup via the `loadwallet` RPC.

VerifyWallets

Wallet verification refers to verification of the `-wallet` arguments as well as the underlying wallet database(s) on disk.

Wallets loaded via program arguments are first verified as part of `AppInitMain()` which first verifies wallet database integrity by calling `VerifyWallets()` via the `WalletClientImpl` override of `client→verify()`.

`VerifyWallets()` takes an `interfaces::Chain` object as argument, which is currently used to send init and error messages (about wallet verification) back to the GUI. `VerifyWallets()` starts by checking that the `walletdir` supplied by argument, or default of `""`, is valid. Next it loops through all wallets it finds in the `walletdir` and adds them to an `std::set` called `wallet_paths`, first de-duplicating them by tracking their absolute paths, and then checking that the `WalletDatabase` for each wallet exists (or is otherwise constructed successfully) and can be verified.

If this check passes for all wallets, then `VerifyWallets()` is complete and will return `true` to calling function `AppInitMain`, otherwise `false` will be returned. If `VerifyWallets()` fails and returns `false` (due to a corrupted wallet database, but notably not due to an incorrect wallet path), the main

program process `AppInit()` will be immediately interrupted and shutdown.

Program shutdown on a potentially-corrupt wallet database is a deliberate design decision. This is so that the wallet cannot display information to the user which is not guaranteed by the database.

LoadWallets

"Startup" wallet(s) are loaded when `client→load()` is called on each `node.chain_client` as part of init.cpp.

src/init.cpp#AppInitMain()

```
for (const auto& client : node.chain_clients) {
    if (!client->load()) {
        return false;
    }
}
```

The call to `load()` on the wallet `chain_clients` has again been overridden, this time by `WalletClientImpl`'s `LoadWallets()` method. This function works similarly to `VerifyWallets()`, first creating the `WalletDatabase` (memory) object for each wallet, although this time skipping the verify step, before creating a `CWallet` object from the database and adding it to the global list of wallets, the vector `vpwallets`, by calling `AddWallet()`.

There are a number of steps in `init.cpp` that happen before the wallet is loaded, notably the blockchain is synced first. This is a safeguard which means that wallet operations cannot be called on a wallet which has been loaded against stale blockchain data.

init.cpp is run on a single thread. This means that calls to wallet code block further initialisation of the node.

The `interfaces::Chain` object taken as argument by `LoadWallets()` is used to pass back any error messages, exactly as it was in `VerifyWallets()`. More information on `AddWallet()` can be found in *src/wallet.cpp*.

StartWallets

The wallet is finally ready when (all) `chain_clients` have been started in *init.cpp* which calls the overridden `client→start()` method from the `WalletClientImpl` class, resulting in src/wallet/load.cpp#StartWallets() being called.

This calls the `GetWallets()` function which returns a vector of pointers to the interfaces for all loaded `CWallet` objects, called `vpwallets`. As part of startup `PostInitProcess()` is called on each wallet which, after grabbing the main wallet lock `cs_wallet`, synchronises the wallet and mempool by adding wallet transactions not yet in a block to our mempool, and updating the wallet with any relevant transactions from the mempool.

Also, as part of `StartWallets`, `flushwallet` *might* be scheduled (if configured by argument) scheduling wallet transactions to be re-broadcast every second, although this interval is delayed upstream with a random timer.

FlushWallets

All wallets loaded into the program are "flushed" (to disk) before shutdown. As part of `init.cpp#Shutdown()` the `flush()` method is called on each member of `node.chain_clients` in sequence. `WalletClientImpl` again overrides this method to call `wallet/load.cpp#FlushWallets()` which makes sure all wallet changes have been successfully flushed to the wallet database.

Finally the `stop()` method is called on each member of `node.chain_clients` which is overridden by `StopWallets()`, flushing again and this time calling `close()` on the database file.

Wallet Locks

Grepping the *src/wallet* directory for locks, conventionally of the form `cs_*`, yields ~500 matches. For comparison the entire remainder of the codebase excluding *src/wallet/** yields almost 1000 matches. Many of these matches are asserts and declarations, however this still illustrates that the wallet code is highly reliant on locks to perform atomic operations with respect to the current chain state.

The `cs_wallet` lock

In order to not block the rest of the program during wallet operations, each `CWallet` has its own recursive mutex `cs_wallet`:

There is currently an issue tracking replacement of Recursive Mutexes with Mutexes, to make locking logic easier to follow in the codebase.

src/wallet/wallet.h

```
/*
 * Main wallet lock.
 * This lock protects all the fields added by CWallet.
 */
mutable RecursiveMutex cs_wallet;
```

Most wallet operations whether reading or writing data require the use of the lock so that atomicity can be guaranteed. Some examples of wallet operations requiring the lock include:

1. Creating transactions
2. Signing transactions
3. Broadcasting/committing transactions
4. Abandoning transactions
5. Bumping transaction (fees)

6. Checking `IsMine`
7. Creating new addresses
8. Calculating balances
9. Creating new wallets
10. Importing new {priv|pub}keys/addresses
11. Importing/dumping wallets

In addition to these higher level functions, most of `CWallet`'s private member functions also require a hold on `cs_wallet`.

Other wallet locks

1. *src/wallet/bdb.cpp*, which is responsible for managing BDB wallet databases on disk, has its own mutex `cs_db`.
2. If external signers have been enabled (via `./configure --enable-external-signer`) then they too have their own mutex `cs_desc_man` which is acquired when descriptors are being setup.
3. `BlockUntilSyncedToCurrentChain()` has a unique lock exclude placed on it to prevent the caller from holding `cs_main` during its execution, and therefore prevent a possible deadlock:

   src/wallet/wallet.h

   ```
   /**
    * Blocks until the wallet state is up-to-date to /at least/ the current
    * chain at the time this function is entered
    * Obviously holding cs_main/cs_wallet when going into this call may cause
    * deadlock
    */
   void BlockUntilSyncedToCurrentChain() const LOCKS_EXCLUDED(::cs_main)
   EXCLUSIVE_LOCKS_REQUIRED(!cs_wallet);
   ```

Controlling the wallet

As we can see wallet component startup and shutdown is largely driven from outside the wallet codebase from *src/init.cpp*.

Once the wallet component is started and any wallets supplied via argument have been verified and loaded, wallet functionality ceases to be called from *init.cpp* and instead is controlled using external programs in a number of ways. The wallet can be controlled using `bitcoin-cli` or `bitcoin-qt` GUI, and wallet files can be interacted with using the stand-alone `bitcoin-wallet` tool.

Both `bitcoind` and `bitcoin-qt` run a (JSON) RPC server which is ready to service, amongst other things, commands to interact with wallets. The command line tool `bitcoin-cli` will allow interaction of any RPC server started by either `bitcoind` or `bitcoin-qt`.

If using `bitcoin-qt` there is also an RPC console built into the GUI or you can run

> with `-server=1` to allow access via `bitcoin-cli`.

If using the `bitcoin-qt` GUI itself then communication with the wallet is done directly via qt's `WalletModel` interface.

Commands which can be used to control the wallet via RPC are listed in *rpcwallet.cpp*.

Wallet via RPC

If we take a look at the `loadwallet` RPC we can see similarities to `WalletClientImpl`'s `LoadWallets()` function.

However this time the function will check the `WalletContext` to check that we have a wallet context (in this case a reference to a chain interface) loaded. Next it will call `wallet.cpp#LoadWallet` which starts by grabbing `g_loading_wallet_mutex` and adding the wallet to `g_loading_wallet_set`, before calling `LoadWalletInternal` which adds the wallet to `vpwallets` and sets up various event notifications.

Further operation of the wallet RPCs are detailed in their man pages, but one thing to take note of is that whilst `loadwallet()` (and `unloadwallet()`) both take a `wallet_name` argument, the other wallet RPCs do not. Therefore in order to control a specific wallet from an instance of `bitcoin{d|-qt}` that has multiple wallets loaded, bitcoin-cli must be called with the `-rpcwallet` argument, to specify the wallet which the action should be performed against, e.g. `bitcoin-cli --rpcwallet=your_wallet_name getbalance`

Via `bitcoin-cli` tool

Blockchain Commons contains numerous guides and examples of controlling the wallet using `bitcoin-cli`, including:

- Sending Bitcoin Transactions including using raw transactions
- Controlling Bitcoin Transactions using RBF and CPFP
- Using multisig to send and receive
- Creating and using PSBTs and integrating them with hardware wallets
- Adding locktimes and OP_RETURN data

CWallet

The `CWallet` object is the fundamental wallet representation inside Bitcoin Core. `CWallet` stores transactions and balances and has the ability to create new transactions. `CWallet` also contains references to the chain interface for the wallet along with storing wallet metadata such as `nWalletVersion`, wallet flags, wallet name and address book.

CWallet creation

The `CWallet` constructor takes a pointer to the chain interface for the wallet, a wallet name and a

pointer to the underlying `WalletDatabase`:

The constructor is not called directly, but instead from the public function `CWallet::Create()`, which is itself called from `CreateWallet()`, `LoadWallets()` (or `TestLoadWallet()`). In addition to the arguments required by the constructor, `CWallet::Create()` also has a `wallet_flags` argument. Wallet flags are represented as a single `unit64_t` bit field which encode certain wallet properties:

src/wallet/walletutil.h

```
enum WalletFlags : uint64_t {
    WALLET_FLAG_AVOID_REUSE = (1ULL << 0),
    WALLET_FLAG_KEY_ORIGIN_METADATA = (1ULL << 1),
    WALLET_FLAG_DISABLE_PRIVATE_KEYS = (1ULL << 32),
    WALLET_FLAG_BLANK_WALLET = (1ULL << 33),
    WALLET_FLAG_DESCRIPTORS = (1ULL << 34),
    WALLET_FLAG_EXTERNAL_SIGNER = (1ULL << 35),
};
```

See *src/wallet/walletutil.h* for additional information on the meanings of the wallet flags.

`CWallet::Create()` will first attempt to create the `CWallet` object and load it, returning if any errors are encountered.

If `CWallet::Create` is creating a new wallet — on its 'first run' — the wallet version and wallet flags will be set, before either `LegacyScriptPubKeyMan` or `DescriptorScriptPubKeyMan`'s are setup, depending on whether the `WALLET_FLAG_DESCRIPTORS` flag was set on the wallet.

Following successful creation, various program arguments are checked and applied to the wallet. These include options such as `-addresstype`, `-changetype`, `-mintxfee` and `-maxtxfee` amongst others. It is at this stage that warnings for unusual or unsafe values of these arguments are generated to be returned to the user.

After the wallet is fully initialized and setup, its keypool will be topped up before the wallet is locked and registered with the Validation interface, which will handle callback notifications generated during the (optional) upcoming chain rescan. The rescan is smart in detecting the wallet "birthday" using metadata stored in the SPKM and won't scan blocks produced before this date.

Finally, the `walletinterface` is setup for the wallet before the `WalletInstance` is returned to the caller.

ScriptPubKeyManagers (SPKM)

Each wallet contains one or more `ScriptPubKeyManager`s which are derived from the base SPKM class and are in control of storing the `scriptPubkey`s managed by that wallet.

> "A wallet" in the general sense therefore becomes "a collection of `ScriptPubKeyManager`s", which are each managing an address type.

In the current implementation, this means that a default (descriptor) wallet consists of 8 `ScriptPubKeyManager`s, one SPKM for each combination shown in the table below.

Table 12. Descriptor wallet SPKMans

	LEGACY	P2SH-SEGWIT	BECH32	BECH32M
Receive	□	□	□	□
Change	□	□	□	□

Here is the *descriptor* wallet code fragment which sets up an SPKM for each `OUTPUT_TYPE`:

src/wallet/wallet.cpp#SetupDescriptorScriptPubKeyMans()

```
// ...

for (bool internal : {false, true}) {
    for (OutputType t : OUTPUT_TYPES) {
        auto spk_manager = std::unique_ptr<DescriptorScriptPubKeyMan>(new
DescriptorScriptPubKeyMan(*this));
        if (IsCrypted()) {
            if (IsLocked()) {
                throw std::runtime_error(std::string(__func__) + ": Wallet is locked,
cannot setup new descriptors");
            }
            if (!spk_manager->CheckDecryptionKey(vMasterKey) && !spk_manager->Encrypt
(vMasterKey, nullptr)) {
                throw std::runtime_error(std::string(__func__) + ": Could not encrypt
new descriptors");
            }
        }
        spk_manager->SetupDescriptorGeneration(master_key, t, internal);
        uint256 id = spk_manager->GetID();
        m_spk_managers[id] = std::move(spk_manager);
        AddActiveScriptPubKeyMan(id, t, internal);
    }
}

// ...
```

By contrast a Legacy wallet will set up a **single** SPKM which will then be *aliased* to a SPKM for each of the 6 `LEGACY_OUTPUT_TYPES`: `LEGACY`, `P2SH-SEGWIT` and `BECH32`. This gives it the external appearance of 6 distinct SPKMans, when really it only has 1:

src/wallet/wallet.cpp#SetupLegacyScriptPubKeyMan()

```
// ...

auto spk_manager = std::unique_ptr<ScriptPubKeyMan>(new LegacyScriptPubKeyMan(*this));
for (const auto& type : LEGACY_OUTPUT_TYPES) {
    m_internal_spk_managers[type] = spk_manager.get();
```

```
    m_external_spk_managers[type] = spk_manager.get();
}
m_spk_managers[spk_manager->GetID()] = std::move(spk_manager);

// ...
```

SPKMans are stored in maps inside a `CWallet` according to output type. "External" and "Internal" (SPKMans) refer to whether the addresses generated are designated for giving out "externally", i.e. for receiving new payments to, or for "internal", i.e. change addresses.

Prior to c729afd0 the equivalent SPKM functionality (fetching new addresses and signing transactions) was contained within `CWallet` itself, now however is split out for better maintainability and upgradability properties as brought about by the wallet box class structure changes. Therefore `CWallet` objects no longer handle keys and addresses.

The change to a `CWallet` made up of (multiple) `{Descriptor|Legacy}ScriptPubKeyMan`'s is also sometimes referred to as the "Wallet Box model", where each SPKM is thought of as a distinct "box" within the wallet, which can be called upon to perform new address generation and signing functions.

Keys in the wallet

Legacy wallet keys

Legacy wallets used the "keypool" model which stored a bunch of keys. See *src/wallet/scriptbpubkeyman.h*#L52-L100 for historical context on the "keypool" model.

The wallet would then simply iterate over each public key and generate a scriptPubKey (a.k.a. PubKey script) and address for each type of script the wallet supported. However this approach has a number of shortcomings (from least to most important):

1. One key could have multiple addresses
2. It was difficult to sign for multisig
3. Adding new script functionality required adding new hardcoded script types into the wallet code *for each new type of script*.

Such an approach was not scalable in the long term and so a new format of wallet needed to be introduced.

Descriptor wallet keys

Descriptor wallets instead store output script "descriptors". These descriptors can be of **any** valid script type, including arbitrary scripts which might be "unknown" to the wallet software, and this means that wallets can deterministically generate addresses for any type of valid descriptor provided by the user.

Descriptors not only contain what is needed to generate an address, they also include all the script template data needed to "solve" (i.e. spend) outputs received at them. In other words they permit a valid `scriptSig` (`redeemScript` or `witnessScript`) to be generated. The document Support for Output

Descriptors in Bitcoin Core provides more details and examples of these output descriptors.

How wallets identify relevant transactions

1. Receiving notifications about new transactions or new blocks

When a Bitcoin Core node learns about a new transaction, the wallet component needs to determine whether it's related to one of it's loaded `CWallet`s. The first thing to notice is that `CWallet` implements the `interfaces::Chain::Notifications`.

```
class CWallet final : public WalletStorage, public interfaces::Chain::Notifications
```

This interface gives the wallet the ability to receive notifications such as `transactionAddedToMempool`, `transactionRemovedFromMempool`, `blockConnected` and so on. The names of these methods are self-explanatory.

To register itself as notification client, the wallet has the `std::unique_ptr<interfaces::Handler> m_chain_notifications_handler` attribute and it is initialized in `CWallet::AttachChain(⋯)` method.

This method updates the wallet according to the current chain, scanning new blocks, updating the best block locator, and registering for notifications about new blocks and transactions. This is called when the wallet is created or loaded (`CWallet::Create(⋯)`).

```
bool CWallet::AttachChain(const std::shared_ptr<CWallet>& walletInstance, interfaces
::Chain& chain, const bool rescan_required, bilingual_str& error, std::vector
<bilingual_str>& warnings)
{
    LOCK(walletInstance->cs_wallet);
    // allow setting the chain if it hasn't been set already but prevent changing it
    assert(!walletInstance->m_chain || walletInstance->m_chain == &chain);
    walletInstance->m_chain = &chain;

    walletInstance->m_chain_notifications_handler = walletInstance->chain
().handleNotifications(walletInstance);
    // ...
}
```

This briefly explains how the wallet is able to listen to new transactions or blocks. More information about the notification mechanism can be seen in the Notifications Mechanism (ValidationInterface) section of Bitcoin Architecture article.

2. Notification Handlers

The next step is to filter which transactions interest the wallet.

Four of these notification handlers are the ones that are relevant to filter transactions. All of them call `CWallet::SyncTransaction(⋯)`.

```
// src/wallet/wallet.h
void SyncTransaction(const CTransactionRef& tx, const SyncTxState& state, bool
update_tx = true, bool rescanning_old_block = false) EXCLUSIVE_LOCKS_REQUIRED
(cs_wallet);

// src/wallet/wallet.cpp
void CWallet::SyncTransaction(const CTransactionRef& ptx, const SyncTxState& state,
bool update_tx, bool rescanning_old_block)
{
    if (!AddToWalletIfInvolvingMe(ptx, state, update_tx, rescanning_old_block))
        return; // Not one of ours

    // If a transaction changes 'conflicted' state, that changes the balance
    // available of the outputs it spends. So force those to be
    // recomputed, also:
    MarkInputsDirty(ptx);
}

void CWallet::transactionAddedToMempool(const CTransactionRef& tx, uint64_t
mempool_sequence) {
    LOCK(cs_wallet);
    SyncTransaction(tx, TxStateInMempool{});
    // ...
}

void CWallet::transactionRemovedFromMempool(const CTransactionRef& tx,
MemPoolRemovalReason reason, uint64_t mempool_sequence) {
    // ...
    if (reason == MemPoolRemovalReason::CONFLICT) {
        // ...
        SyncTransaction(tx, TxStateInactive{});
    }
}

void CWallet::blockConnected(const CBlock& block, int height)
{
    // ...
    for (size_t index = 0; index < block.vtx.size(); index++) {
        SyncTransaction(block.vtx[index], TxStateConfirmed{block_hash, height,
static_cast<int>(index)});
        transactionRemovedFromMempool(block.vtx[index], MemPoolRemovalReason::BLOCK, 0
/* mempool_sequence */);
    }
}

void CWallet::blockDisconnected(const CBlock& block, int height)
{
    // ...
    for (const CTransactionRef& ptx : block.vtx) {
        SyncTransaction(ptx, TxStateInactive{});
```

```
    }
}
```

Note that `CWallet::SyncTransaction(⋯)` adds the transaction(s) to wallet if it is relevant and then marks each input of the transaction (`const std::vector<CTxIn> CTransaction::vin`) as dirty so the balance can be recalculated correctly.

3. Scanning the block chain

Another method that calls `CWallet::SyncTransaction(⋯)` is the `CWallet::ScanForWalletTransactions(⋯)`, which scans the block chain (starting in `start_block` parameter) for transactions relevant to the wallet.

This method is called when manually requesting a rescan (`rescanblockchain` RPC), when adding a new descriptor or when a new key is added to the wallet.

```
CWallet::ScanResult CWallet::ScanForWalletTransactions(const uint256& start_block, int
start_height, std::optional<int> max_height, const WalletRescanReserver& reserver,
bool fUpdate)
{
    // ...
    for (size_t posInBlock = 0; posInBlock < block.vtx.size(); ++posInBlock) {
        SyncTransaction(block.vtx[posInBlock], TxStateConfirmed{block_hash,
block_height, static_cast<int>(posInBlock)}, fUpdate, /*rescanning_old_block=*/true);
    }
    // ...
}
```

4. `AddToWalletIfInvolvingMe(⋯)`

`CWallet::AddToWalletIfInvolvingMe` performs the following steps:

1. If the transaction is confirmed, it checks if it conflicts with another. If so, marks the transaction (and its in-wallet descendants) as conflicting with a particular block (`if (auto* conf = std::get_if<TxStateConfirmed>(&state))`).
2. It checks if the wallet already contains the transaction. If so, updates if requested in the `fUpdate` parameter or finishes the execution (`if (fExisted && !fUpdate) return false;`).
3. It checks if the transaction interests the wallet (`if (fExisted || IsMine(tx) || IsFromMe(tx))`)
    - If so, it checks if any keys in the wallet keypool that were supposed to be unused have appeared in a new transaction.
        - If so, removes those keys from the keypool (`for (auto &dest : spk_man->MarkUnusedAddresses(txout.scriptPubKey))`).
4. Finally, it adds the transaction to the wallet (`AddToWallet(⋯)`). This function inserts the new transaction in `CWallet::mapWallet`, updates it with relevant information such as `CWalletTx::nTimeReceived` (time it was received by the node), `CWalletTx::nOrderPos` (position in ordered transaction list) and so on.

This function also writes the transaction to database (`batch.WriteTx(wtx)`) and mark the transaction as dirty to recalculate balance.

src/wallet/wallet.cpp

```
bool CWallet::AddToWalletIfInvolvingMe(const CTransactionRef& ptx, const SyncTxState&
state, bool fUpdate, bool rescanning_old_block)
{
    const CTransaction& tx = *ptx;
    {
        AssertLockHeld(cs_wallet);

        if (auto* conf = std::get_if<TxStateConfirmed>(&state)) {
            // ...
        }

        bool fExisted = mapWallet.count(tx.GetHash()) != 0;
        if (fExisted && !fUpdate) return false;
        if (fExisted || IsMine(tx) || IsFromMe(tx))
        {
            for (const CTxOut& txout: tx.vout) {
                for (const auto& spk_man : GetScriptPubKeyMans(txout.scriptPubKey)) {
                    for (auto &dest : spk_man->MarkUnusedAddresses(txout.
scriptPubKey)) {
                        // ...
                    }
                }
            }

            TxState tx_state = std::visit([](auto&& s) -> TxState { return s; },
state);
            return AddToWallet(MakeTransactionRef(tx), tx_state,
/*update_wtx=*/nullptr, /*fFlushOnClose=*/false, rescanning_old_block);
        }
    }
    return false;
}

CWalletTx* CWallet::AddToWallet(CTransactionRef tx, const TxState& state, const
UpdateWalletTxFn& update_wtx, bool fFlushOnClose, bool rescanning_old_block)
{
    LOCK(cs_wallet);

    WalletBatch batch(GetDatabase(), fFlushOnClose);

    uint256 hash = tx->GetHash();

    // ...

    auto ret = mapWallet.emplace(std::piecewise_construct, std::forward_as_tuple(
hash), std::forward_as_tuple(tx, state));
```

```
    CWalletTx& wtx = (*ret.first).second;
    // ...
    if (fInsertedNew) {
        wtx.nTimeReceived = GetTime();
        wtx.nOrderPos = IncOrderPosNext(&batch);
        // ...
    }

    // ...

    // Write to disk
    if (fInsertedNew || fUpdated)
        if (!batch.WriteTx(wtx))
            return nullptr;

    // Break debit/credit balance caches:
    wtx.MarkDirty();

    // ...

    return &wtx;
}
```

5. `CWallet::IsMine(…)`

As the name implies, the method that actually identifies which transactions belong to the wallet is `IsMine()`.

```
isminetype CWallet::IsMine(const CScript& script) const
{
    AssertLockHeld(cs_wallet);
    isminetype result = ISMINE_NO;
    for (const auto& spk_man_pair : m_spk_managers) {
        result = std::max(result, spk_man_pair.second->IsMine(script));
    }
    return result;
}
```

Note the `CWallet::IsMine(const CScript& script)` is just a proxy to the `ScriptPubKeyMan::IsMine(const CScript &script)`. This is an important distinction, because in Bitcoin Core the class `CWallet` does not manage the keys. This work is done by `ScriptPubKeyMan` subclasses: `DescriptorScriptPubKeyMan` and `LegacyScriptPubKeyMan`. All `ScriptPubKeyMan` instances belonging to the wallet are stored in `CWallet::m_spk_managers`.

Another important aspect of that method is the return type, the `enum isminetype`. This type is defined in `src/wallet/ismine.h`.

```
enum isminetype : unsigned int {
```

```
    ISMINE_NO         = 0,
    ISMINE_WATCH_ONLY = 1 << 0,
    ISMINE_SPENDABLE  = 1 << 1,
    ISMINE_USED       = 1 << 2,
    ISMINE_ALL        = ISMINE_WATCH_ONLY | ISMINE_SPENDABLE,
    ISMINE_ALL_USED   = ISMINE_ALL | ISMINE_USED,
    ISMINE_ENUM_ELEMENTS,
};
```

For `LegacyScriptPubKeyMan`: * `ISMINE_NO`: the scriptPubKey is not in the wallet; * `ISMINE_WATCH_ONLY`: the scriptPubKey has been imported into the wallet; * `ISMINE_SPENDABLE`: the scriptPubKey corresponds to an address owned by the wallet user (who can spend with the private key); * `ISMINE_USED`: the scriptPubKey corresponds to a used address owned by the wallet user; * `ISMINE_ALL`: all ISMINE flags except for USED; * `ISMINE_ALL_USED`: all ISMINE flags including USED; * `ISMINE_ENUM_ELEMENTS`: the number of isminetype enum elements.

For `DescriptorScriptPubKeyMan` and future `ScriptPubKeyMan`: * `ISMINE_NO`: the scriptPubKey is not in the wallet; * `ISMINE_SPENDABLE`: the scriptPubKey matches a scriptPubKey in the wallet. * `ISMINE_USED`: the scriptPubKey corresponds to a used address owned by the wallet user.

`IsMine` historically was located outside of the wallet code, but now takes a more logical position as a member function of `CWallet` which returns an `isminetype` value from an enum.

More information on the `IsMine` semantics can be found in release-notes-0.21.0.md#ismine-semantics.

6. `DescriptorScriptPubKeyMan::IsMine(…)`

`DescriptorScriptPubKeyMan::IsMine(…)` basically checks if `DescriptorScriptPubKeyMan::m_map_script_pub_keys` contains the `CScript scriptPubKey` passed in parameter.

```
isminetype DescriptorScriptPubKeyMan::IsMine(const CScript& script) const
{
    LOCK(cs_desc_man);
    if (m_map_script_pub_keys.count(script) > 0) {
        return ISMINE_SPENDABLE;
    }
    return ISMINE_NO;
}
```

`DescriptorScriptPubKeyMan::m_map_script_pub_keys` is a `std::map<CScript, int32_t>` type (a map of scripts to the descriptor range index).

7. `LegacyScriptPubKeyMan::IsMine(…)`

`LegacyScriptPubKeyMan::IsMine(…)` is only a proxy for `IsMineResult IsMineInner(…)`.

```
isminetype LegacyScriptPubKeyMan::IsMine(const CScript& script) const
{
    switch (IsMineInner(*this, script, IsMineSigVersion::TOP)) {
    case IsMineResult::INVALID:
    case IsMineResult::NO:
        return ISMINE_NO;
    case IsMineResult::WATCH_ONLY:
        return ISMINE_WATCH_ONLY;
    case IsMineResult::SPENDABLE:
        return ISMINE_SPENDABLE;
    }
    assert(false);
}
```

`IsMineResult IsMineInner(…)` is only used by `LegacyScriptPubKeyMan` (which should be deprecated at some point) and is considerably more complex than its equivalent in the more modern `DescriptorScriptPubKeyMan`.

The first step is to call `Solver(scriptPubKey, vSolutions)` method, which parses a scriptPubKey and identifies the script type for standard scripts. If successful, returns the script type and parsed pubkeys or hashes, depending on the type. For example, for a P2SH script, `vSolutionsRet` will contain the script hash, for P2PKH it will contain the key hash, an so on.

```
IsMineResult IsMineInner(const LegacyScriptPubKeyMan& keystore, const CScript&
scriptPubKey, IsMineSigVersion sigversion, bool recurse_scripthash=true)
{
    IsMineResult ret = IsMineResult::NO;

    std::vector<valtype> vSolutions;
    TxoutType whichType = Solver(scriptPubKey, vSolutions);
    // ...
}
```

The next step is to handle each script type separately. Note that if it is a Taproot transaction, it will not be considered spendable by legacy wallets. They purposely do not support Taproot as they are marked for deprecation.

```
IsMineResult IsMineInner(...)
{
    // ...
    TxoutType whichType = Solver(scriptPubKey, vSolutions);

    CKeyID keyID;
    switch (whichType) {
    case TxoutType::NONSTANDARD:
    case TxoutType::NULL_DATA:
    case TxoutType::WITNESS_UNKNOWN:
```

```
    case TxoutType::WITNESS_V1_TAPROOT:
        break;
    case TxoutType::PUBKEY:
        // ...
    case TxoutType::WITNESS_V0_KEYHASH:
        // ...
    case TxoutType::PUBKEYHASH:
        // ...
    case TxoutType::SCRIPTHASH:
        // ...
    case TxoutType::WITNESS_V0_SCRIPTHASH:
        // ...
    case TxoutType::MULTISIG:
        // ...
    }
    } // no default case, so the compiler can warn about missing cases

    if (ret == IsMineResult::NO && keystore.HaveWatchOnly(scriptPubKey)) {
        ret = std::max(ret, IsMineResult::WATCH_ONLY);
    }
    return ret;
}
```

If no script type conditions are met for a `scriptPubKey`, the function checks at the end if it is a watch-only script in the wallet.

```
IsMineResult IsMineInner(...)
{
    // ...
    switch (whichType) {
        // ...
        case TxoutType::PUBKEY:
        keyID = CPubKey(vSolutions[0]).GetID();
        if (!PermitsUncompressed(sigversion) && vSolutions[0].size() != 33) {
            return IsMineResult::INVALID;
        }
        if (keystore.HaveKey(keyID)) {
            ret = std::max(ret, IsMineResult::SPENDABLE);
        }
        break;
        // ...
    }
    // ...
}
```

When the script type is a public key, the function first checks if it is a `P2PK` (uncompressed public key), otherwise it must be 33 bytes (compressed format).

It then checks if the wallet keystore has the key. In this case, it means the script can be spent by the

wallet.

In the early days of Bitcoin, the transactions were of type P2PK, which were specified in uncompressed format. However using this format turned out to be both wasteful for storing unspent transaction outputs (UTXOs) and a compressed format was adopted for P2PKH and P2WPKH.

Uncompressed format has:

- 04 - Marker
- x coordinate - 32 bytes, big endian
- y coordinate - 32 bytes, big endian

And the compressed has:

- 02 if y is even, 03 if odd - Marker
- x coordinate - 32 bytes, big endian

Note that the compressed format has a total of 33 bytes (x coordinate + marker).

More recently, taproot address P2TR was introduced and it uses a format called x-only, with only x coordinate - 32 bytes, big endian.

The next step is the SegWit format (P2WPKH). First the function invalidates the script if this has a P2WPKH nested inside P2WSH. It then checks that the script is in the expected format with the OP_0 before the witness output.

If these two validations pass, the script will be recreated as Public Key Hash and the function will be called recursively. Note that in this second call, the script will be handled as TxoutType::PUBKEYHASH.

```
IsMineResult IsMineInner(...)
{
    // ...
    case TxoutType::WITNESS_V0_KEYHASH:
    {
        if (sigversion == IsMineSigVersion::WITNESS_V0) {
            // P2WPKH inside P2WSH is invalid.
            return IsMineResult::INVALID;
        }
        if (sigversion == IsMineSigVersion::TOP && !keystore.HaveCScript(CScriptID
(CScript() << OP_0 << vSolutions[0]))) {
            // We do not support bare witness outputs unless the P2SH version of it
would be
            // acceptable as well. This protects against matching before segwit
activates.
            // This also applies to the P2WSH case.
            break;
        }
```

```
        ret = std::max(ret, IsMineInner(keystore, GetScriptForDestination(PKHash
(uint160(vSolutions[0]))), IsMineSigVersion::WITNESS_V0));
        break;
    }
    // ...
}
```

The `TxoutType::PUBKEYHASH` logic is very similar to the `TxoutType::PUBKEY`: it checks if the wallet keystore has the key, which means the script can be spent by the wallet.

Before that, however, the function validates whether the key must be compressed.

```
IsMineResult IsMineInner(...)
{
    // ...
    case TxoutType::PUBKEYHASH:
        keyID = CKeyID(uint160(vSolutions[0]));
        if (!PermitsUncompressed(sigversion)) {
            CPubKey pubkey;
            if (keystore.GetPubKey(keyID, pubkey) && !pubkey.IsCompressed()) {
                return IsMineResult::INVALID;
            }
        }
        if (keystore.HaveKey(keyID)) {
            ret = std::max(ret, IsMineResult::SPENDABLE);
        }
        break;
    // ...
}
```

The next item to be dealt with is `TxoutType::SCRIPTHASH`. The logic is very similar to the one seen before. First the script is validated (`P2SH` inside `P2WSH` or `P2SH` is invalid) and the function checks if the script exists in THE wallet keystore. As with `TxoutType::WITNESS_V0_KEYHASH`, the function will recurse into nested p2sh and p2wsh scripts or will simply treat any script that has been stored in the keystore as spendable.

```
IsMineResult IsMineInner(...)
{
    // ...
    case TxoutType::SCRIPTHASH:
    {
        if (sigversion != IsMineSigVersion::TOP) {
            // P2SH inside P2WSH or P2SH is invalid.
            return IsMineResult::INVALID;
        }
        CScriptID scriptID = CScriptID(uint160(vSolutions[0]));
        CScript subscript;
        if (keystore.GetCScript(scriptID, subscript)) {
```

```
            ret = std::max(ret, recurse_scripthash ? IsMineInner(keystore, subscript,
IsMineSigVersion::P2SH) : IsMineResult::SPENDABLE);
        }
        break;
    }
    // ...
}
```

`TxoutType::WITNESS_V0_SCRIPTHASH` has the same logic seen in the previous item. The only difference is that the has `Hash160` is recreated with the solved script hash, since `P2SH-P2WSH` is allowed.

```
IsMineResult IsMineInner(...)
{
    // ...
    case TxoutType::WITNESS_V0_SCRIPTHASH:
    {
        if (sigversion == IsMineSigVersion::WITNESS_V0) {
            // P2WSH inside P2WSH is invalid.
            return IsMineResult::INVALID;
        }
        if (sigversion == IsMineSigVersion::TOP && !keystore.HaveCScript(CScriptID
(CScript() << OP_0 << vSolutions[0]))) {
            break;
        }
        uint160 hash;
        CRIPEMD160().Write(vSolutions[0].data(), vSolutions[0].size()).Finalize(hash
.begin());
        CScriptID scriptID = CScriptID(hash);
        CScript subscript;
        if (keystore.GetCScript(scriptID, subscript)) {
            ret = std::max(ret, recurse_scripthash ? IsMineInner(keystore, subscript,
IsMineSigVersion::WITNESS_V0) : IsMineResult::SPENDABLE);
        }
        break;
    }
    // ...
}
```

The last type of script is `TxoutType ::MULTISIG`, whose logic is straightforward. `Solver (···)` returns all the keys of the script and then they are validated in the same way as the previous scripts. Transactions are only considered `ISMINE_SPENDABLE` if the node has all keys.

```
IsMineResult IsMineInner(...)
{
    // ...
    case TxoutType::MULTISIG:
    {
        if (sigversion == IsMineSigVersion::TOP) {
```

```
            break;
        }

        std::vector<valtype> keys(vSolutions.begin()+1, vSolutions.begin()+vSolutions
.size()-1);
        if (!PermitsUncompressed(sigversion)) {
            for (size_t i = 0; i < keys.size(); i++) {
                if (keys[i].size() != 33) {
                    return IsMineResult::INVALID;
                }
            }
        }
        if (HaveKeys(keys, keystore)) {
            ret = std::max(ret, IsMineResult::SPENDABLE);
        }
        break;
    }
    // ...
}
```

Thus, we cover most of the code responsible for identifying which transactions belong to the wallet. The code related to `IsMine(…)` or `IsMineInner(…)` is used either when the transactions arrive through the mempool or by blocks.

Constructing transactions

In order to construct a transaction the wallet will validate the outputs, before selecting some coins to use in the transaction. This involves multiple steps and we can follow an outline of the process by walking through the `sendtoaddress` RPC command, which returns by calling `SendMoney()`.

After initialisation `SendMoney()` will call `wallet.CreateTransaction()` (`CWallet::CreateTransaction()`) followed by `wallet.CommitTransaction()` if successful. If we follow `wallet.CreateTransaction()` we see that it is a wrapper function which calls private member function `CWallet::CreateTransactionInternal()`.

CreateTransactionInternal

We fetch change addresses of an "appropriate type" here, where "appropriate" means that it should try to minimise revealing that it is a change address, for example by being a different `OUTPUT_TYPE` to the other outputs. Once a suitable change address is selected A new `ReserveDestination` object is created which keeps track of reserved addresses to prevent address re-use.

The address is not "fully" reserved until `GetReservedDestination()` is called later.

Next some basic checks on the requested transaction parameters are carried out (e.g. sanity checking of amounts and recipients) by looping through each pair of (recipient, amount). After initializing a new transaction (`txNew`), a fee calculation (`feeCalc`) and variables for the transaction size, we enter into a new code block where the `cs_wallet` lock is acquired and the `nLockTime` for the

transaction is set:

src/wallet/wallet.cpp#CWallet::CreateTransactionInternal()

```
// ...

CMutableTransaction txNew;
FeeCalculation feeCalc;
CAmount nFeeNeeded;
std::pair<int64_t, int64_t> tx_sizes;
int nBytes;
{
    std::set<CInputCoin> setCoins;
    LOCK(cs_wallet);
    txNew.nLockTime = GetLocktimeForNewTransaction(chain(), GetLastBlockHash(),
GetLastBlockHeight());
        {
            std::vector<COutput> vAvailableCoins;
            AvailableCoins(vAvailableCoins, true, &coin_control, 1, MAX_MONEY,
MAX_MONEY, 0);

    // ...
```

Bitcoin Core chooses to set `nLockTime` to the current block to discourage fee sniping.

We must acquire the lock here because we are about to attempt to select coins for spending, and optionally reserve change addresses.

If we did not have the lock it might be possible for the wallet to construct two transactions which attempted to spend the same coins, or which used the same change address.

AvailableCoins

After this, a *second* new code block is entered where "available coins" are inserted into a vector of `COutput`s named `vAvailableCoins`. The concept of an "available coin" is somewhat complex, but roughly it excludes:

1. "used" coins
2. coins which do not have enough confirmations (N.B. confirmations required differs for own change)
3. coins which are part of an immature coinbase (< 100 confirmations)
4. coins which have not entered into our mempool
5. coins which are already being used to (attempt) replacement of other coins

This call to `AvailableCoins()` is our first reference back to the underlying `ScriptPubKeyMan`s controlled by the wallet. The function iterates over all coins belonging to us — found in the `CWallet.mapWallet` mapping — checking coin availability before querying for a `SolvingProvider`

(ultimately calling `GetSigningProvider()`): essentially querying whether the active `CWallet` has a `ScriptPubKeyMan` which can sign for the given output.

src/wallet/wallet.cpp#CWallet::GetSolvingProvider()

```
std::unique_ptr<SigningProvider> CWallet::GetSolvingProvider(const CScript& script,
SignatureData& sigdata) const
{
    for (const auto& spk_man_pair : m_spk_managers) {
        if (spk_man_pair.second->CanProvide(script, sigdata)) {
            return spk_man_pair.second->GetSolvingProvider(script);
        }
    }
    return nullptr;
}
```

Below is a section of the `AvailableCoins()` function which illustrates available coins being added to the `vAvailableCoins` vector, with the call to `GetSolvingProvider()` visible.

If a `SigningProvider` is found a second check is performed: to see if the coin is "solvable" by calling `IsSolvable()`.

Whilst `getSolvingProvider()` might return a `SigningProvider` (read: SPKM), not all SPKMs will be able to provide **private** key data needed for signing transactions, e.g. in the case of a watch-only wallet.

After we have determined solvablility, "spendability" is calculated for each potential output along with any coin control limitations:

src/wallet/wallet.cpp#AvailableCoins()

```
    // ...

    for (unsigned int i = 0; i < wtx.tx->vout.size(); i++) {

        // ...

        std::unique_ptr<SigningProvider> provider = GetSolvingProvider(wtx.tx->vout[
i].scriptPubKey);

        bool solvable = provider ? IsSolvable(*provider, wtx.tx->vout[i].scriptPubKey)
: false;
        bool spendable = ((mine & ISMINE_SPENDABLE) != ISMINE_NO) || (((mine &
ISMINE_WATCH_ONLY) != ISMINE_NO) && (coinControl && coinControl->fAllowWatchOnly &&
solvable));

        vCoins.push_back(COutput(&wtx, i, nDepth, spendable, solvable, safeTx,
(coinControl && coinControl->fAllowWatchOnly)));

        // Checks the sum amount of all UTXO's.
```

```
        if (nMinimumSumAmount != MAX_MONEY) {
            nTotal += wtx.tx->vout[i].nValue;

            if (nTotal >= nMinimumSumAmount) {
                return;
            }
        }

        // Checks the maximum number of UTXO's.
        if (nMaximumCount > 0 && vCoins.size() >= nMaximumCount) {
            return;
        }

        // ...
```

See the full `CWallet::AvailableCoins()` implementation for additional details and caveats.

CreateTransactionInternal continued

After available coins have been determined, we check to see if the user has provided a custom change address (used coin control), or whether the earlier not-fully-reserved change address should finally be reserved and selected by calling `GetReservedDestination()`. The change outputs' `size`, `discard_free_rate` and `effective_fee_rate` are then calculated. The `discard_fee_rate` refers to any change output which would be dust at the `discard_rate`, and that you would be willing to discard completely and add to fee (as well as continuing to pay the fee that would have been needed for creating the change).

Coin selection

Now that we have a vector of available coins and our fee rate settings estimated, we are ready to start coin selection itself. This is still an active area of research, with two possible coin selection solving algorithms currently implemented:

1. Branch and bound ("bnb")
2. Knapsack

The branch and bound algorithm is well-documented in the codebase itself:

src/wallet/coinselection.cpp

```
/*
This is the Branch and Bound Coin Selection algorithm designed by Murch. It searches
for an input
set that can pay for the spending target and does not exceed the spending target by
more than the
cost of creating and spending a change output. The algorithm uses a depth-first search
on a binary
tree. In the binary tree, each node corresponds to the inclusion or the omission of a
UTXO. UTXOs
```

```
are sorted by their effective values and the trees is explored deterministically per
the inclusion
branch first. At each node, the algorithm checks whether the selection is within the
target range.
While the selection has not reached the target range, more UTXOs are included. When a
selection's
value exceeds the target range, the complete subtree deriving from this selection can
be omitted.
At that point, the last included UTXO is deselected and the corresponding omission
branch explored
instead. The search ends after the complete tree has been searched or after a limited
number of tries.

The search continues to search for better solutions after one solution has been found.
The best
solution is chosen by minimizing the waste metric. The waste metric is defined as the
cost to
spend the current inputs at the given fee rate minus the long term expected cost to
spend the
inputs, plus the amount the selection exceeds the spending target:

waste = selectionTotal - target + inputs x (currentFeeRate - longTermFeeRate)

The algorithm uses two additional optimizations. A lookahead keeps track of the total
value of
the unexplored UTXOs. A subtree is not explored if the lookahead indicates that the
target range
cannot be reached. Further, it is unnecessary to test equivalent combinations. This
allows us
to skip testing the inclusion of UTXOs that match the effective value and waste of an
omitted
predecessor.

The Branch and Bound algorithm is described in detail in Murch's Master Thesis:
https://murch.one/wp-content/uploads/2016/11/erhardt2016coinselection.pdf

@param const std::vector<CInputCoin>& utxo_pool The set of UTXOs that we are choosing
from.
       These UTXOs will be sorted in descending order by effective value and the
CInputCoins'
       values are their effective values.
@param const CAmount& target_value This is the value that we want to select. It is the
lower
       bound of the range.
@param const CAmount& cost_of_change This is the cost of creating and spending a
change output.
       This plus target_value is the upper bound of the range.
@param std::set<CInputCoin>& out_set -> This is an output parameter for the set of
CInputCoins
       that have been selected.
@param CAmount& value_ret -> This is an output parameter for the total value of the
```

```
CInputCoins
       that were selected.
@param CAmount not_input_fees -> The fees that need to be paid for the outputs and
fixed size
       overhead (version, locktime, marker and flag)
*/
```

You can read a little more about the differences between these two coin selection algorithms in this StackExchange answer.

You can read more about `waste` and the waste metric in this StackExchange answer.

Coin selection is performed as a loop, as it may take multiple iterations to select the optimal coins for a given transaction.

Multiwallet

Work on the multiwallet project means that Bitcoin Core can now handle dynamic loading and unloading of multiple wallets while running.

Exercises

> Using either `bitcoin-cli` in your terminal, or a Jupyter notebook in conjunction with the `TestShell` class from the Bitcoin Core Test Framework, try to complete the following exercises.
>
> Changes to the codebase will require you to re-compile afterwards.
>
> Don't forget to use the compiled binaries found in your source directory, for example `/home/user/bitcoin/src/bitcoind`, otherwise your system might select a previously-installed (non-modified) version of bitcoind.

1. Modify a wallet RPC
   - ☐ Create a descriptor wallet
   - ☐ Generate coins to yourself
   - ☐ Remove the `"dummy"` parameter from the `getbalance` Wallet RPC
   - ☐ Ensure that the `rpc_help.py` functional test passes (but ignore other test failures), fixing any errors

     

     run `test/functional/rpc_help.py` to just run a single test
   - ☐ Check that the rpc call `getbalance 3 true true` passes with the `dummy` parameter removed
1. `IsMine`
   - ☐ Create a descriptor wallet
   - ☐ Generate coins to yourself

- ▢ Send coins to yourself in a transaction and generate a block to confirm
- ▢ Modify the wallet's `IsMine()` logic to always return `false`
- ▢ Generate a new block and try to send coins to yourself in a transaction again

  Observe the changes

2. Coin Selection
   - ▢ Create a descriptor wallet
   - ▢ Generate 200 blocks to yourself
   - ▢ Call `listunspent` and then `send` a large amount (e.g. 600 BTC) to yourself and observe how many inputs were used
   - ▢ Add a new preferred coin selection algorithm to the wallet that uses **all** UTXOs in the wallet and optionally remove the other algorithms.
   - ▢ Redo the send and confirm that this time it will select all inputs in the wallet for the transaction
3. Adding a new RPC
   - ▢ Add a new RPC which when called will simply return to the user a random UTXO from the wallet in the form

```
{
  "txid": <txid>,
  "vout": <vout>
}
```

GUI

This section has been updated to Bitcoin Core @ v23.0

The GUI has its own separate repo at bitcoin-core/gui. PRs which primarily target the GUI should be made here, and then they will get merged into the primary repo. Developer Marco Falke created an issue in his fork which detailed some of the rationale for the split, but essentially it came down to:

1. Separate issue and patch management
2. More focused review and interests
3. Maintain high quality assurance

He also stated that:

> Splitting up the GUI (and splitting out modules in general) has been brought up often in recent years. Now that the GUI is primarily connected through interfaces with a bitcoin node, it seems an appropriate time to revive this discussion.

> — Marco Falke

PR#19071 contained the documentation change now contained in the Bitcoin Core primary repository, along with details of the monotree approach that was ultimately taken. The documentation change provides guidance on what a "GUI change" is:

> As a rule of thumb, everything that only modifies `src/qt` is a GUI-only pull request. However:
>
> - For global refactoring or other transversal changes the node repository should be used.
> - For GUI-related build system changes, the node repository should be used because the change needs review by the build systems reviewers.
> - Changes in `src/interfaces` need to go to the node repository because they might affect other components like the wallet.
>
> For large GUI changes that include build system and interface changes, it is recommended to first open a PR against the GUI repository. When there is agreement to proceed with the changes, a PR with the build system and interfaces changes can be submitted to the node repository.
>
> — src/CONTRIBUTING.md

On a related note, another issue was recently opened by Falke, to discuss the possibility of instituting the same monotree changes for wallet code.

Motivation for a GUI

Bitcoin Core has shipped with a GUI since the first version. Originally this was a wxWidgets GUI, but in 2011 a move to QT was completed. Satoshi originally had plans to have a decentralized market place and even poker game inside Bitcoin, so including a GUI, which also had wallet and address book functionality, made sense from the get-go.

The motivation to *continue* to include a GUI with Bitcoin Core today is for accessibility. New users can access a best-in-class Bitcoin experience via a single software package. It's not safe or realistic to expect users to download multiple programs and connect them securely into a software suite, just to use bitcoin.

It does not have to be the prettiest UI, but needs to provide the functionality to use bitcoin. It is possible to connect other frontends to Bitcoin Core, but they are connected via RPCs, and do not have the first-class interface (to the node component) that the bundled GUI has.

Building the GUI

`bitcoin-qt`, which includes the QT GUI with the node, is built automatically when the build

dependencies are met. Required packages to meet dependencies can be found in the build instructions in *src/doc/build-*.md* as appropriate for your platform. If you have the required packages installed but do not wish to build the `bitcoin-qt` then you must run `./configure` with the option `--with-gui=no`.

If the build is configured with `--enable-multiprocess` then additional binaries will be built:

1. `bitcoin-node`
2. `bitcoin-wallet`
3. `bitcoin-gui`

Qt

QT is currently very intertwined with the rest of the codebase. See the library depencency graph for more context.

Developers would ideally like to reduce these dependencies in the future.

Qt documentation

There is useful documentation for developers looking to contribute to the Qt side of the codebase found at Developer Notes for Qt Code.

Main GUI program

The loading point for the GUI is *src/qt/main.cpp*. `main()` calls `GuiMain()` from *src/qt/bitcoin.cpp*, passing along any program arguments with it. `GuiMain` starts by calling `SetupEnvironment()` which amongst other things, configures the runtime locale and charset.

Next an empty `NodeContext` is set up, which is then populated into a fully-fledged node interface via being passed to `interfaces::MakeNode()`, which returns an `interfaces::Node`. Recall that in wallet component initialization we also saw the wallet utilizing a `NodeContext` as part of its `WalletInitInterface`. In both cases the `NodeContext` is being used to pass chain and network references around without needing to create globals.

After some QT setup, command-line and application arguments are parsed. What follows can be outlined from the code comments:

3. Application identification
4. Initialization of translations, so that intro dialogue is in user's language
5. Now that settings and translations are available, ask user for data directory
6. Determine availability of data directory and parse bitcoin.conf
7. Determine network (and switch to network specific options)
8. URI IPC sending

9. Main GUI initialization

GUI initialisation

After configuration the GUI is initialized. Here the `Node` object created earlier is passed to `app.SetNode()` before a window is created and the application executed.

The bulk of the Qt GUI classes are defined in *src/qt/bitcoingui.{h|cpp}*.

QML GUI

Since writing this documentation focus has been directed towards re-writing the Qt code leveraging the Qt QML framework. This will allow developers to create visually-superior, and easier to write and reason-about GUI code, whilst also lowering the barriers to entry for potential new developers who want to be able to focus on GUI code.

The recommendation therefore is to familiarise yourself with Qt QML and review the current codebase for the latest developments. You can follow along with the latest QML work in the specific bitcoin-core/qml-gui repo.

Bitcoin design

The Bitcoin design guide provides some guidance on common pitfalls that Bitcoin GUI designers should look out for when designing apps (like `bitcoin-qt`).

Testing QT

Currently, although several QT tests exist in *src/qt/test*, there is no good way to test QT changes except by hand. A good way to try and have QT code included in the test framework is to target having the RPC layer be a thin as possible, so more code can be re-used between RPC and GUI.

P2P

This section has been updated to Bitcoin Core @ v23.0

With bitcoin we are seeking to create a permissionless network in which anyone can make a bitcoin transaction. Anybody should be free and able to run a node and join the network.

The Bitcoin P2P network serves 3 purposes:

- Gossiping addresses of known reachable nodes on the network
- Relaying unconfirmed transactions
- Propagating blocks

Although these three purposes share the same network, they have different design goals and properties. Transaction relay is optimized for a combination of redundancy/robustness to peer

misbehaviour as well as bandwidth minimization, while block relay is optimized to minimize delay.

Design philosophy

The P2P design philosophy is outlined in the bitcoin devwiki article P2P Design Philosophy. A synopsis of the ideas can be found in the first few paragraphs:

> For the Bitcoin network to remain in consensus, the network of nodes must not be partitioned. So for an individual node to remain in consensus with the network, it must have at least one connection to that network of peers that share its consensus rules.
>
> …
>
> We can't rely on inbound peers to be honest, because they are initiated by others. It's impossible for us to know, for example, whether all our inbound peers are controlled by the same adversary.
>
> Therefore, in order to try to be connected to the honest network, we focus on having good outbound peers, as we get to choose who those are.

The document, which is worth reading in its entirely, continues by assuming the case that we don't have any inbound peers but also considering that any inbound peers we *do* have shouldn't be able to interfere with the P2P logic proposed.

Design goals

Amiti Uttarwar created a framework of 5 goals she sees for the P2P network.

TLDR; We want valid messages to make it out to the network (**reliable**) in a reasonable amount of time (**timely**) and for nodes to be able to get onto the network and stay on the network of their own accord (**accesible**). These three values seem quite important for any peer-to-peer network to be successful but in Bitcoin we have two additional. **Privacy** because it is money and **upgradeability** because of the ethos of Bitcoin.

1. **Reliable**; if a node submits a valid message to the network it will eventually be delivered to all other nodes on the network.
2. **Timely**; each of the messages have to make it out in a reasonable amount of time.
    - Reasonable amount of time for a transaction is different than for a block and reasonable amount of time for a block to be propagated for a normal user versus a miner is very different as well.
3. **Accessible**; the requirement to be able to participate must be low. Also an adversary shouldn't be able to keep a node off the network.
    - Currently it is still possible to run a full Bitcoin Core node on a Raspberry Pi which is a low

barrier-to-entry.

4. **Private**; because it is money and fundamentally it comes down to the idea of not wanting to connect your real world identity with your onchain interactions.
5. **Upgradeable**; stems from the ethos that if a user decides to buy into the rule set at a specific point in time they should always be able to transact with the rule set they initially bought into.

Reliability vs **Privacy** can seem at odds with one another as is really hard to design and achieve both of them at the same time. For example, **value long-lasting connections**, can help for reliable delivery but **comes against privacy**. **Dynamic connections** help maintain transaction privacy, but **comes against reliability**. Reliability is you want to tell everyone your message, but privacy is you don't want them to know that it is your message.

See the transcript for more detail on each of those points.

P2P attacks

In a permissionless system two types of users are both *equally* free to access and attempt to use the network:

1. Honest users
2. Attackers/spammers

Types of activities an attacker might attempt to perform on a target node which involve the P2P layer include:

- Exhaust CPU/memory
  - Create infinite loops
  - Cause OOM (exhaust memory)
  - Clog up network traffic
  - Fill mempool with garbage
  - Temporarily stall the network
- Eclipse/sybil attacks
  - Reduce privacy
  - Cause network splits
  - Eclipse attack
  - Sybil attack

The Bitcoin protocol does not have a concept of node identifiers or other reputation system through which we can permanently block a node we identify as malicious from future communications. If a node reconnects to us using a different IP address we will not be able to tell it was the same node we had seen before. Make no mistake that this is a large win for the censorship-resistance of the network, but it makes P2P implementation more precarious.

Our program must contain logic to protect against the above attacks in a scenario where they may

happen often and freely. Bitcoin Core employs a number of techniques in the P2P domain to try and protect against these types of attacks including:

Table 13. Protective counter-measures

Technique	Protection
Proof of Work*	Exhaust CPU/memory
Mempool policy for transactions	Exhaust CPU/memory
Peer address bucketing	Eclipse/Sybil attacks
`block-relay-only` connections	Eclipse attacks
Ephemeral block-relay-only connections for headers	Eclipse attacks
Disconnecting "misbehaving" peers	Exhaust CPU/memory
Peer rotation/eviction	Eclipse/sybil attacks
Protected peers (from eviction)	Eclipse attacks
Anchor peers	Eclipse attacks

* If an "attacker" has sufficient hash power, then from a PoW perspective they are not really an attacker.

Eclipse attacks

Eclipse attacks occur when an adversary is able to isolate a victim's node from the rest of the network.

A *restart-based eclipse attack* occurs when the adversary is able to add its own addresses to the victim's address manager and then force the victim to restart. If the attack succeeds, the victim will make all of its connections to the adversary's addresses when it restarts.

Issue 17326 proposed persisting the node's outbound connection list to disk, and on restart reconnecting to the same peers. It's worth reading the full discussion in that issue, since there are a lot of subtle points around which peers should be persisted.

Addrman and eclipse attacks(bitcoin-devwiki) attempts to describe the mechanisms implemented in Bitcoin Core to mitigate eclipse attacks followed by open questions and areas of further research.

Identification of the network topology

If a malicious entity was able to identify the topography of the network then they could see that by taking specific nodes down, maybe via a DOS service or any attack that they can use, they can cause a partition in the entire network.

There are three main messages that are gossiped around the network and **each message offers a unique set of information that allows an adversary to identify who your neighbors are**.

Block relay leaks the least information and we can leverage that for a feature called `block-relay-only` connections, a type of connection where nodes do not participate in transaction or address

relay and only relay blocks. An effective way for a spy node to infer the network topology is to observe the timing and details of transaction and address relay, so **these block-relay-only connections obfuscate network topology and help to mitigate eclipse attacks**.

PR#15759 introduced `block-relay-only` connections. After these changes, nodes by default open two outbound block-relay-only connections on startup.

PR#17428 introduced the idea of anchors, persist peers to reconnect after restart. If you persist the connection to some peers is great for reliability but it would not be very good for privacy if we were to reconnect to the full relay connections. So instead, we use the `block-relay-only` connections and reconnect to those.

PR#19858 proposes a more advanced use of block-relay-only connections to further mitigate eclipse attacks. The node will periodically initiate an *additional* block-relay-only connection which it uses only to sync headers in order to try and learn about new blocks. If this reveals new blocks, the eviction logic will rotate out an existing block-relay-only connection. If no new blocks are discovered, the connection is closed.

Node P2P components

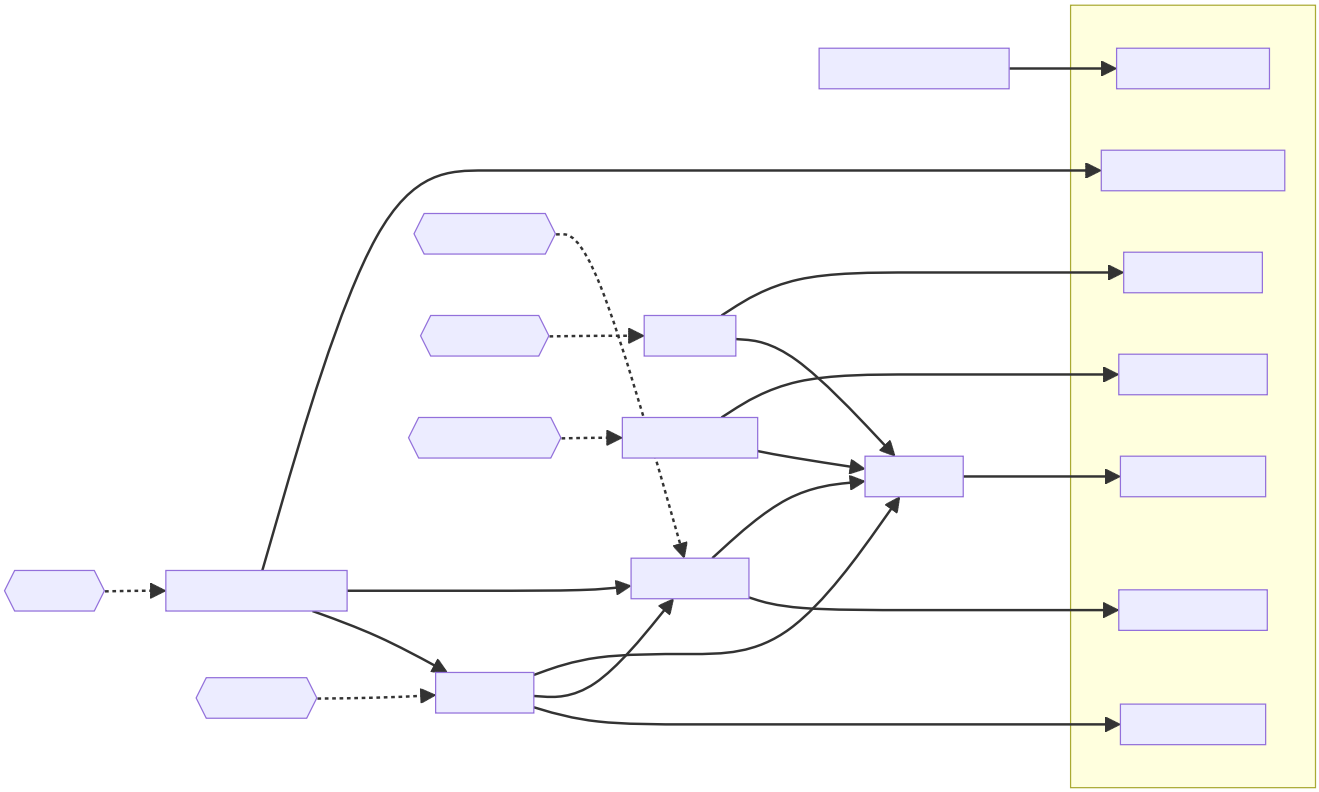

Figure 10. Node P2P components

NetGroupManager

NetGroupManager is used to encapsulate all asmap data and logic. It is setup by loading any provided asmap file passed during startup.

History

- PR#16702 introduced asmap as part of Addrman.
- PR#22910 introduced NetGroupManager as a better way to access asmap data by both Addrman and CConnman.

Addrman

Addrman is the in-memory database of peers and consists of the new and tried tables. These tables are stored in `peers.dat` and serve as cache for network information that the node gathered from previous connections, so that if it is rebooted it can quickly re-establish connections with its former peer network and avoid performing bootstrapping again.

Addrman is setup using `LoadAddrman` from *src/addrdb.cpp*, passing in the `NetGroupManager`, our global program `args` and a pointer to the (to be initialized) `Addrman`. `args` are used to determine whether consistency checks should run and to pass on the `datadir` value in order to attempt deserialization on any addrman database (`peers.dat`) that is found.

Addresses are serialized back to disk both after the call to `CConnman::StopNodes()`, but also periodically (by default every 15 minutes) as scheduled by `CConnman::Start()`:

```
    // Dump network addresses
    scheduler.scheduleEvery([this] { DumpAddresses(); }, DUMP_PEERS_INTERVAL);
```

Adding addresses to addrman

Addresses learned about over the wire will be deserialized into a vector of `CAddress`-es. After determining whether we should expend resources on processing these addresses — check that address relay with this peer is permitted *and* that peer is not marked as misbehaving — we shuffle the addresses and begin testing them as candidates for addition to our addrman.

Address candidate testing consists of checking:

- we are not rate-limiting the peer who sent us the address
- it is a full node (via service flag bits)
- if we already know of the address
- if they're automatically discouraged or manually banned
- `IsReachable()` and `IsRoutable()`

Once these checks have finished we will add all the addresses we were happy with by calling `AddrMan::Add()` and passing the vector of good addresses in along with metadata on who sent us this address in the form of a `CNetAddr` (the source address). The source address is notably used later in `Add()` (by `AddrmanImpl::AddSingle()`) to determine which new bucket this new address should be placed into as an anti-sybil measure.

Addresses are added into the appropriate bucket and position in `vvNew`. If there **is not** an address in the corresponding bucket/position then the new address will be added there immediately. If there

is currently an address in the corresponding bucket/position then `IsTerrible()` is called to determine whether the old address should be evicted to make room for the new one or not, in which case the new address is simply dropped.

This eviction behaviour is distinct from test-before-evict described below in Good.

Good

New connections are initiated by Connman, in `CConnman::ThreadOpenConnections()`. Addresses are considered "good" and will begin being processed by `Addrman::Good()` if:

1. we have received a version message from them
2. it was an outbound connection

Next we use the following process to determine whether the address should be added to one of the buckets in the `vvTried` set:

1. we will first check that the address i) does not already exist in `vvTried`, and that ii) it *does* exist in `vvNew`.
2. if the address is not yet in `vvTried` we will determine its bucket and position and then check if there is already an address at that position.
3. if there is an address there, we will initiate a `FEELER` connection to the existing node.
4. if the feeler is successful then we drop the new address and keep what we have.
5. if the feeler is unsuccessful then we drop the old address and insert the new address at this location using `MakeTried()`.

This process is called test-before-evict.

Select

`CConnman::ThreadOpenConnections()` also handles selection of new peers to connect to, via `Addrman::Select()`.

This first occurs when we want to try a new feeler, but we will use the same approach for non-feeler connections too.

The `Select()` function contains a lot of interesting logic, specifically related to injecting randomness into the process of drawing a new address to connect to from our untried buckets.

It starts by using a 50% chance between selecting an address from our `tried` and `new` buckets, before using additional (non-cryptographic) randomness to select a bucket and position, before iterating over the bucket until it finds an address. Once it has selected an address, it uses additional randomness via `GetChance()`, to determine whether it will actually *use* this address to connect to.

The purpose of the additional `chance` in address selection is that it helps to deprioritize recently-tried and failed addresses.

The use of randomness like this in addrman is to combat types of attack where our addrman might

become "poisoned" with a large number of sybil or otherwise-bad addresses. The use of bucketing and randomness means that these types of attacks are much harder to pull off by an attacker, requiring for example a large number of nodes on different Autonomous Systems.

Banman

Banman is generally used as a filter to determine whether we should accept a new incoming connection from a certain IP address, or less-frequently to check whether we should make an outbound connection to a certain IP address:

- We do not accept connections from banned peers
- We only accept connections from discouraged peers if our inbound slots aren't (almost) full
- We do not process (check `IsReachable()` and `IsRoutable()` and `RelayAddress()`) addresses received in an `ADDR` / `ADDRV2` which are banned, but do remember that we have received them

Banman is setup with a simple call to its constructor, passing in a `banlist` and `bantime` argument. `banlist` will store previously-banned peers from last shutdown, while `bantime` determines how long the node discourages "misbehaving" peers.

Banman operates primarily with bare IP addresses (`CNetAddr`) but can also, when initiated by the user, ban an entire subnet (as a `CSubNet`).

Note that banman handles both manual bans initiated by the user (with `setban`) and also automatic discouragement of peers based on P2P behaviour.

The banman header file contains some good background on what banning can and can't protect against, as well as why we do not automatically ban peers in Bitcoin Core.

Connman

Connman is used to manage connections and maintain statistics on each node connected, as well as network totals. There are many connection-related program options for it such as number of connections and whitebound ports/interfaces. It takes an Addrman and a NetGroupManager to its constructor, along with two random seeds used to seed the SipHash randomizer.

The nonces generated by the randomizer are used to detect us making new connections to ourself, as the incoming nonce in the version message would match our `nLocalHostNonce`

Connman is started via `node.connman->Start()` in *init.cpp*. This begins by calling `init()` which binds to any ports selected, before starting up an I2P session if the I2P proxy is found. Next it schedules sending `GETADDR` to any seednodes provided (via `-seednodes`) using the `ThreadOpenConnections()` loop, and then continues by loading anchor connections from *anchors.dat*. Following this the various net threads are started up.

As connman has a pointer to the node's addrman it can directly fetch new addresses to serve via `CConnman:GetAddresses()`. If new addresses are requested from a remote P2P node (via `GETADDR`), then it will use a cached addr response to respond with. This helps to defeat surveillance which is

seeking to determine which other peers your node is connected to.

Within `CConnman` we maintain `m_nodes`, a vector of connections to other nodes. That vector is updated and accessed by various threads, including:

1. The socket handler thread, which is responsible for reading data from the sockets into receive buffers, and also for accepting new incoming connections.
2. The open connections thread, which is responsible for opening new connections to peers on the network.
3. The message handler thread, which is responsible for reading messages from the receive buffer and passing them up to net_processing.

Since the vector can be updated by multiple threads, it is guarded by a mutex called `m_nodes_mutex`.

CConnman::ThreadOpenConnections()

This thread begins by making any manually-specified connections before entering a double-nested `while` loop. The outer loop handles making a connection on each loop according certain priorities and the number of connections we currently have:

net.cpp#L2028

```
// Determine what type of connection to open. Opening
// BLOCK_RELAY connections to addresses from anchors.dat gets the highest
// priority. Then we open OUTBOUND_FULL_RELAY priority until we
// meet our full-relay capacity. Then we open BLOCK_RELAY connection
// until we hit our block-relay-only peer limit.
// GetTryNewOutboundPeer() gets set when a stale tip is detected, so we
// try opening an additional OUTBOUND_FULL_RELAY connection. If none of
// these conditions are met, check to see if it's time to try an extra
// block-relay-only peer (to confirm our tip is current, see below) or the next_feeler
// timer to decide if we should open a FEELER.
```

In addition to filling out connections up to full-relay and block-relay-only capacity it also periodically makes a feeler connection to a random node from addrman to sync headers and test that we haven't been eclipsed.

After selecting which type of connection we are going to attempt on this iteration we enter the inner loop which attempts to make the connection itself. We select the connection by assigning it to `addrconnect`.

1. If it is trying to make an anchor connection then simply set `addrconnect` to the selected `addr` and break from the loop early
2. If it is trying to make a feeler connection then we request a collision address or if one is not available then select another `vvTried` table address using `addrman.Select()`.
3. If it is neither an anchor or a feeler just call `addrman.Select()`.

A "collision address" means that another address had tried to evict this address

from `vvTried` table, these addresses are marked in `Addrman.m_tried_collisions`.

If the various checks pass, then finish by calling `OpenNetworkConnection()`. `OpenNetworkConnection()` makes the connection by calling `ConnectNode()`, which if successful creates a new `CNode` object for the connected node and returns it. Next we initialize the `CNode` with cconnman's pointer to `peerman`, via `m_msgproc->InitializeNode(pnode)`. Finally we add the connected and initialized node to `CConnman.m_nodes`.

Bootstrapping

Bootstrapping is probably the most dangerous moment in a node's life. If the new node cannot make at least one connection to an honest node, from whom it can eventually learn more honest addresses, then it may not ever be able to join the most-work bitcoin chain without manual user intervention.

Manual intervention here would require the user to find the IP address of a known-honest node and connect to it either using `addnode` or `connect`.

When the node first starts up, and if no node addresses are manually specified, we have no choice but to fetch addresses from one (or more) hardcoded DNS seed(s) the list of which can be found in *src/chainparams.cpp*.

If the node is fed only attacker-controlled addresses by one or more dishonest DNS seed(s) then it has little opportunity to join the rest of the honest network. However, if one or more of the addresses returned by the DNS query are honest then we want the node to be able to (eventually) find and connect to the honest network.

Note that if the DNS seed queries are unsuccessful, or the node is being run in a Tor-only mode (and currently the DNS seeds cannot support long Tor V3 addresses) then bitcoind will fall back to connecting to a hard-coded list of seed nodes. This fall back functionality could help to protect against e.g. an attack on the DNS seed infrastructure.

Service flags

Nodes can advertise service flags (a.k.a. "service bits") indicating which services that node supports.

Managing connections

An enumeration of the different types of connections, along with detailed descriptions on their functions, can be found in src/*net.h*.

Message relay

Table 14. Relay policy of different messages

Message type	Function	Who
Addresses	`PeerManagerImpl::RelayAddress()`	Outbound peers & inbound peers who send an addr-related message but not block-relay-only peers Reachable addresses to 2 peers. Unreachable addresses randomly to 1 or 2 peers.
Transactions	`PeerManagerImpl::RelayTransaction()`	All connected peers

Blocks	`PeerManagerImpl::UpdatedBlockTip()`
	
	`PeerManagerImpl::MaybeSendAddr()`
		All connected peers

Address relay

The Bitcoin network uses `addr` messages to communicate (node) network addresses. See the Bitcoin wiki p2p documentation for more details. Good address propagation improves network connectivity and increases the difficulty of executing an eclipse attack.

Bitcoin Core nodes will periodically self-announce (also known as self-advertise) their own network address to peers. When a Bitcoin Core node receives an `addr` message that contains 10 addresses or fewer, it forwards those addresses with a timestamp within 10 minutes of the current time to 1 or 2 peers, selected at random. If we assume all nodes do this, then self-announcements should reach a large portion of the nodes on the network. The timestamp condition is there to ensure that the relay of a given address stops after some time.

Since PR#22387, there is a rate limit for address relay processing, so that addresses from peers that send too many of them are ignored which can help to prevent CPU/memory exhaustion attacks.

`Addr` privacy

For some time, it was possible for a spy node to easily scrape the full contents of any reachable node's `AddrMan`. The spy just had to connect to a victim node multiple times and execute `GETADDR`. This scraped data could then be used to infer private information about the victim.

For example, a spy could monitor the victim's `AddrMan` content in real time and figure out which peers a node is connected to. A spy could also compare the `AddrMan` content from two different connections (e.g. one identified by Tor address and one identified by IPv4) and figure out that it's actually the same physical node (`fingerprinting`).

PR#18991 was a first step towards fixing these privacy issues. By limiting (caching) the leaked portion of `AddrMan`, these inference activities became much harder. Caching in this context means that the `ADDR` response (which is only a small subset of a node's `AddrMan` content) remains the same for every `GETADDR` call during (roughly) a day.

Addr black holes

We know that some nodes on the network do *not* relay `addr` messages that they receive. Two known

cases are block-relay-only connections from Bitcoin Core nodes, and connections from certain light clients. We refer to these connections as `addr` black holes. `addr` messages go in, but they never escape!

If a large portion of the connections on the network are `addr` black holes, then `addr` propagation may be negatively impacted: self-announcements might not reach a majority of nodes on the network in a timely fashion. It'd be better if we could somehow avoid picking black holes as the 1 or 2 peers that we select for relaying `addr` messages to.

PR#21528 defers initialization of `m_addr_known` of inbound peers until the peer sends an address related message (`addr`, `addrv2`, `getaddr` or `sendaddrv2`). The node uses the presence of `m_addr_known` to decide whether the peer is a candidate for relaying `addr` messages received from the network.

addrv2

PR#19031 is a proposed implementation of the BIP155 `addrv2` message, a new P2P message format proposed in early 2019 by Wladimir J. van der Laan to gossip longer node addresses.

The `addrv2` message is required to support next-generation Tor v3 Onion addresses, the Invisible Internet Project (I2P), and potentially other networks that have longer endpoint addresses than fit in the 128 bits/16 bytes of the current `addr` message.

Transaction relay

Relaying transactions is a core tenet of a Bitcoin node, along with address relay and block relay. However, we don't necessarily want to immediately relay transactions we accept into our mempool immediately for the following reasons:

1. Privacy: Adding a small delay in transaction relay helps obscure the route transactions take, making it harder to use transaction timing to infer the structure of the network or the original source of the transaction.
2. Load balancing: Having a small delay in transaction relay helps avoid the possibility that all transactions will be requested from the peer with the lowest network latency simply because they announce the transaction first.
3. Saving bandwidth: Having a longer delay in transaction relay may allow some transactions to not be relayed at all, eg in the case where a low fee rate transaction is accepted into the mempool and then evicted due to being at the bottom of the mempool, or RBFed prior to being relayed.

Rejecting incoming transactions

In addition to being careful about transaction relay, we must also reject (some) incoming transactions before they enter our mempool, which acts as a DoS prevention measure for our node. If we were to accept and blindly relay all transactions INVed to us by our peers, then an attacker could cheaply use (waste) a node's system resources and bandwidth, and have their attack amplified by the transaction flooding mechanism.

How do we currently limit incoming transactions?

1. We reject transactions which don't pass policy checks e.g.:
    a. We reject transactions that don't pay the mempool min fee (set based on maximum mempool size)
    b. We reject RBF transactions that don't increase the fee rate by more than `-incrementalrelayfee`
2. We reject transactions which don't pass replacement/package checks.
3. We reject transactions which don't pass consensus checks.

What other mechanisms *could* we consider using before the ATMP checks are performed?

1. We *could* reject transactions from individual peers that send transactions at too high a rate, however this would just encourage attackers to make multiple connections, using up additional inbound slots
2. We *could* ignore transactions from any peer once some rate limit is hit, however this would drop high feerate transactions from innocent peers which would be doubly undesirable
3. We *could* artificially increase our mempool min fee when a rate limit is exceeded, even if the mempool is not full?

Initial broadcast

If a spy is able to identify which node initially broadcast a transaction, there's a high probability that that node is the source wallet for the transaction. To avoid that privacy leak, we try to be intentional about how we relay and request transactions. We don't want to reveal the exact contents of our mempool or the precise timing when we received a transaction.

PR#18861 improved transaction-origin privacy. The idea is that if we haven't yet announced a transaction to a peer, we shouldn't fulfil any `GETDATA` requests for that transaction from that peer. The implementation for that PR checks the list of transactions we are about to announce to the peer (`setInventoryTxToSend`), and if it finds the transaction that the peer has requested, then responds with a `NOTFOUND` instead of with the transaction.

While this helps in many cases, why is it still an imperfect heuristic?

PR#19109 further reduces the possible attack surface. It introduces a per-peer rolling bloom filter (`m_recently_announced_invs`) to track which transactions were recently announced to the peer. When the peer requests a transaction, we check the filter before fulfilling the request and relaying the transaction.

Rebroadcasting transactions

Hiding links between wallet addresses and IP addresses is a key part of Bitcoin privacy. Many techniques exist to help users obfuscate their IP address when submitting their own transactions, and various P2P changes have been proposed with the goal of hiding transaction origins.

Beyond initial broadcast, *rebroadcast* behaviour can also leak information. If a node rebroadcasts its own wallet transactions differently from transactions received from its peers, for example more frequently, then adversaries could use this information to infer transaction origins even if the

initial broadcast revealed nothing.

The goal is to improve privacy by making node rebroadcast behaviour for wallet transactions indistinguishable from that of other peers' transactions.

PR#21061 adds a `TxRebroadcast` module responsible for selecting transactions to be rebroadcast and keeping track of how many times each transaction has been rebroadcast. After each block, the module uses the miner and other heuristics to select transactions from the mempool that it believes "should" have been included in the block and re-announces them (disabled by default for now).

Rebroadcasts happen once per new block. The set of transactions to be rebroadcast is calculated as follows:

- The node regularly estimates the minimum feerate for transactions to be included in the next block, `m_cached_fee_rate`.
- When a new block arrives, the transactions included in the block are removed from the mempool. The node then uses `BlockAssembler` to calculate which transactions (with a total weight up to 3/4 of the block maximum) from the mempool are more than 30 minutes old and have a minimum feerate of `m_cached_fee_rate`. This results in a set of transactions that our node would have included in the last block.
- The rebroadcast attempt tracker, `m_attempt_tracker`, tracks how many times and how recently we've attempted to rebroadcast a transaction so that we don't spam the network with re-announcements.

Block relay

After a block is mined it is broadcast to the P2P network where it will eventually be relayed to all nodes on the network. There are two methods available for relaying blocks:

1. **Legacy Relay**
    - A node participating in legacy relaying will always send or request entire blocks.
    - For nodes that maintain a mempool this is quite bandwidth inefficient, since they probably already have most of the transactions from a new block in their mempool.
2. **Compact Block Relay**
    - Specified in BIP 152.
    - The goal is to address the bandwidth inefficiencies of legacy relaying by only relaying the transactions of a new block that the requesting peer has not yet seen.
    - Check out this Compact Blocks FAQ for benchmarks and more info.

blocksonly versus block-relay-only

Bitcoin Core 0.12 introduced a `-blocksonly` setting that can reduce a node's bandwidth usage by 88%. The reduction is achieved by not participating in transaction relay. For more info check out this post on blocksonly mode by Gregory Maxwell.

Blocksonly nodes currently use compact block relaying to download blocks even though they don't

maintain a full mempool. PR#22340 makes blocksonly nodes use legacy relaying to download new blocks. Because `-blocksonly` is a global startup option, it therefore applies to all connections

block-relay-only connections are a specific type of connection which is used by Bitcoin Core full nodes to only participate in block relay.

As currently implemented block-relay-only connections (introduced in PR#15759), disables both transaction and address relay. Bitcoin Core nodes per default settings make two **outbound** block-relay-only connections in addition to 8 regular outbound connections (also see eclipse attacks for more use cases of these connections).

Table 15. blocksonly mode vs block-relay-only connections

	`-blocksonly`	block-relay-only
Applies to	All node connections (global)	Two randomly-chosen connections
Does `Addr` relay	☐	☐
Sends transactions	May do in special cases (e.g. submitted via RPC)	☐
Receives transactions	Signals not to with `fRelay`, will disconnect if breached	?
Other connections	still makes two block-relay-only connections (for which block-relay-only rules apply)	N/A

Bloom filters and SPV

A bloom filter is a probabilistic data structure. It supports two operations:

1. *adding* an element to the filter
2. *querying* an element from the filter

If an element has been previously added, then querying for the element will return *true*. If an element has not been added, then querying for the element may return *true* or *false*. In other words, querying may return a *false positive*, but will never return a *false negative*.

See the wikipedia page for how a bloom filter is implemented with hash functions onto a bitfield. Note that the false positive rate depends on the size of the filter and the number of hash functions.

BIP 37 introduced a new method for Simple Payment Verification (SPV) clients to use bloom filters to track transactions that affect their addresses. BIP 37 was implemented in Bitcoin Core in PR#1795.

Using the P2P messages defined in BIP 37, an SPV client can request that a full node send it transactions which match a bloom filter. The full node will then relay unconfirmed transactions that match the filter, and the client can request merkle blocks, which only contain the transactions that match the filter.

The SPV client chooses the bloom filter parameters (filter size, number of hashes and a 'tweak' for

the hashes) and sends them to the node in a `filterload` message.

The original implementation contained a logic bug. If the client sent a `filterload` message with a zero-sized filter, then the serving node could later attempt a divide-by-zero and crash when querying an element from the filter. See CVE-2013-5700 for further details.

This bug was quietly fixed in PR#2914 without advertising the reason. That fix added the `isFull` and `isEmpty` booleans, which have proven to be confusing for developers.

PR#18806 removed those `isFull` and `isEmpty` booleans and adds a more straightforward fix for the issue.

Compact Block Filters for Light Clients

Compact Block Filters were introduced with BIP 157/158 as an improvement upon Bloom filters, as used in BIP 37. **Instead of the client sending a filter** to a full node peer, **full nodes generate deterministic filters on block data that are served to the client**. The light client gets these filters from the server and checks for itself if any of its objects match what is seen in the filter. If it does match, then the light client asks for the full block.

BIP 158 describes a structure for compact filters on block data. It specifies one filter type called **Basic block filters**, which encodes the scriptPubKeys of all the UTXOs spent in the block, and the scriptPubKeys of all the new UTXOs created in the block. **This is the only block filter currently supported**. PR#12254 implemented compact block filters in Bitcoin Core, and PR#14121 added a new index (`-blockfilterindex=1`), which stores the compact block filters for blocks that have been validated.

BIP 157 is the proposed specification for requesting and sending compact filters between nodes on the p2p network. It was implemented with a series of PRs, demonstrated in PR#18876.

Benefits:

- Less asymmetry in the client. If light clients request a filter for a block, the server wont have to do any more work than the client had to do when making the request.
- More privacy and less trust. The light client no longer sends a fingerprint of the data it is interested in to the server, and so it becomes way more difficult to analyse the light client's activity.
- Conceptually, BIP158's Golomb-Coded Set (GCS) filter is similar to a Bloom filter (no false negatives, a controllable rate of false positives), but more compact.

Downsides:

- They require more disk space because of the overhead that comes with the new index.
- GCS filters are write-once (you can't update them once created), and querying is much slower.
    - Bloom filters are effectively `O(n)` for finding `n` elements in them. GCS are `O(m+n)` for finding `n` elements in a filter of size `m`. So, Bloom filters are way faster if you're only going to do one or a few queries. But as you're querying for larger and larger number of elements, the relative downside of a GCS's performance goes down.

glimpse of the future; PR#25957 uses BIP 157 block filters for faster wallet rescans.

Notifying peers of relay preferences

Currently, block-relay-only connections are established indirectly:

- When making an outbound block-relay-only connection, a node sets the boolean flag `fRelay` in the version message to `false`.
- `fRelay` (introduced in the context of BIP 37) does not imply that transactions cannot be sent for the entire duration of the connection - in its original use case with BIP37, relay of transactions can be activated later on.
- `fRelay=false` is also used in **-blocksonly** mode, a low-bandwidth option in which a node does not want to receive transactions from **any peer**, but does participate in address relay.

Therefore, nodes currently don't have a notion which of their incoming peers see the connection as block-relay-only and don't have any logic attached to it.

PR#20726, accompanied by the new BIP proposal BIP 338, introduces the new p2p message `disabletx` for block-relay-only connections, which makes it explicit that no messages related to transaction relay should ever be exchanged over the duration of the connection.

P2P message encryption

P2P messages are currently all unencrypted which can potentially open up vulnerabilities like:

- Associated metadata in P2P messages may reveal private information.
- Possibilities for attackers who control the routing infrastructure of the P2P network to censor P2P messages since P2P messages can be detected trivially - they always start with a fixed sequence of magic bytes.

BIP 324 proposes a new Bitcoin P2P protocol which features transport encryption and slightly lower bandwidth usage.

bip324.com contains a list of all the open PRs and great resources to understand the proposal. A visual explanation of how BIP 324 works can be found in this blog - How to encrypt the P2P protocol?

Networking contribution to node RNG entropy

Entropy for the RNG is often harvested from network connections:

src/net.cpp

```
net.cpp
488-
489:    // We're making a new connection, harvest entropy from the time (and our peer
count)
```

```
490-    RandAddEvent((uint32_t)id);
--
743-
744:    // We just received a message off the wire, harvest entropy from the time (and
the message checksum)
745-    RandAddEvent(ReadLE32(hash.begin()));
--
1160-
1161:    // We received a new connection, harvest entropy from the time (and our peer
count)
1162-    RandAddEvent((uint32_t)id);
```

Peer state

Peer state is divided into two types:

- **Network/Connection state**; any low level stuff, sending/receiving bytes, keeping statistics, eviction logic, etc.
- **Application state**; any data that is transmitted within P2P message payloads, and the processing of that data. Examples are tx inventory, addr gossiping, ping/pong processing.

There are three main data structures that handle peer state:

- `CNode` (defined in `net.h`, used by `m_nodes`(`CConnman`) and covered by `m_nodes_mutex`) is concerned with the **connection state** of the peer.
- `CNodeState` (defined in `netprocessing.cpp`, used by `m_node_states`(`PeerManager`) and covered by `cs_main`) is concerned with the **application state** of the peer.
  - It maintains validation-specific state about nodes, therefore guarded by `cs_main`.
- `Peer` (defined in `netprocessing.cpp`, used by `m_peer_map`(`PeerManager`) and covered by `m_peer_mutex`) is concerned with the **application state** of the peer.
  - It doesn't contain validation-critical data, therefore it is not guarded by `cs_main`

However, there is still some *application state* contained in `CNode` for historic reasons. Issue 19398 outlines the process to eventually move this out of `CNode` as well as the reasoning behind the introduction of the `Peer` struct.

P2P violations

Bitcoin Core has several options for how to treat peers that violate the rules of the P2P protocol:

1. Ignore the individual message, but continue processing other messages from that peer
2. Increment the peer's "misbehaviour" score, and punish the peer once its score goes above a certain amount
3. Disconnect from the peer
4. Disconnect from the peer and prevent any later connections from that peer's address

(discouragement)

Since PR#20079 we now treat handshake misbehaviour like an unknown message

Testing P2P changes

It can be challenging to test P2P changes as tooling and functional tests are lacking. Often devs simply setup a new node with the patch and leave it for some time!?

Is there fuzzing for P2P messages yet?

Testing transaction and block relay under SegWit

SegWit was a softfork defined in BIP 141, with P2P changes defined in BIP 144.

SegWit was activated at block 481,824 in August 2017. Prior to activation, some very careful testing was carried out to verify different scenarios, for example:

1. How are transactions and blocks relayed between un-upgraded and upgraded nodes?
2. How do upgraded nodes find other upgraded nodes to connect to?
3. If a node is un-upgraded at activation time and subsequently upgrades, how does it ensure that the blocks that it previously validated (without segwit rules) are valid according to segwit rules?

To enable this kind of testing, PR#8418 made it possible to configure the segwit activation parameters using a `-bip9params` configuration option. That configuration option was later renamed to `-vbparams` in PR#10463, and replaced with `-segwitheight` in PR#16060.

Those options allowed starting a node which would never activate segwit by passing `-vbparams=segwit:0:0` (or later, `-segwitheight=-1`). This was used in the functional tests to test the node's behaviour across activation.

The segwit mainnet activation was a one-time event. Now that segwit has been activated, those tests are no longer required.

PR#21090 removed the final tests that made use of `-segwitheight=0`. With those tests removed, the special casing for `-segwitheight=-1` behaviour can also be removed. That special casing impacted logic in *net_processing*, *validation* and *mining*.

include:links-onepage.adoc == Exercises

Using either `bitcoin-cli` in your terminal, or a Jupyter notebook in conjunction with the `TestShell` class from the Bitcoin Core Test Framework, try to complete the following exercises.

Changes to the codebase will require you to re-compile afterwards.

Don't forget to use the compiled binaries found in your source directory, for example `/home/user/bitcoin/src/bitcoind`, otherwise your system might select a previously-installed

(non-modified) version of bitcoind.

1. Make manual connections

- ▢ Add the following configuration options to a new Bitcoin Core node running on signet to have it start it with no connections:

```
signet=1
dnsseed=0
fixedseeds=0
debug=addrman
```

- ▢ Find the (only!) Signet DNS seeder node (in the `SigNetParams` class starting with "seed") and using a terminal poll this seed node for an address to connect to.

You can use `dig` or `nslookup` to retrieve seeds from the DNS seeder from the DNS seeders.

If you try this with the mainnet seeds you will need to consider which service flags the seeder advertises support for. For example, if a seed node advertises `x1` support this means they return IP addresses of nodes advertizing the `NODE_NETWORK` service flag.

You could query these from sipa's mainnet seeder by prepending `x1` to the subdomain e.g. `nslookup x1.seeder.bitcoin.sipa.be`

- ▢ Check how many addresses are known to your node: `bitcoin-cli -signet getnodeaddresses 0 | jq length`
- ▢ Using one of the addresses returned from the previous exercise, connect to this node using the `addnode` RPC.
- ▢ Observe new addresses being received and connected to in the bitcoind terminal or *$DATADIR/debug.log* file.
- ▢ What dangers can there be in retrieving node addresses in this way?
- ▢ Is this more of less safe than using the hardcoded seeds? Can you think of a better way to distribute seeds to new users?

Mempool

Mempool terminology

Ancestor(s)

One or more "parent" transactions which must be confirmed **before** the current transaction. The ancestor transaction(s) *create* outputs which are depended on by the current transaction.

Descendant(s)

One or more "child" transactions which must be confirmed **after** the current transaction.
The descendant transaction(s) *depend* on outputs from the current transaction.

Orphan

A transaction with missing ancestors.

When *ancestor* and *descendant* are encountered in the codebase, they refer specifically to other **in-mempool** transactions.

Ancestors and descendants can be confirmed in the same block but they must be in the correct order within the list of `transactions` for the block to be valid.

Mempool purpose

1. The mempool is designed to hold a list of unconfirmed-but-valid transactions that the node has learned about.
2. Miners will select transactions from the mempool for assembly into a block using the `getblocktemplate` RPC.
3. Transactions have to pass all policy and validation checks before being allowed to enter the mempool.
   The mempool therefore also acts as DoS protection for the node.
4. Transactions will not be added to the mempool if they do not meet fee requirements, are non-standard, or double-spend an input of a transaction already in the mempool (excluding BIP 125 RBF transactions).

There is a bitcoin-devwiki page Mempool and mining which includes some additional mempool philosophy.

James O'Beirne has written a comprehensive overview of the current challenges and work in mempool design. It "documents the existing design, failures, and vulnerabilities of the mempool as well as some proposals that exist to remedy the shortcomings."

Mempool policy goals

The documentation subfolder doc/policy contains up-to-date information on **some**, but not all, of the current mempool policy rules.

Mempool life cycle

Initialisation

The primary mempool object itself is initialized onto the `node` in *init.cpp* as part of `AppInitMain()` which takes `NodeContext& node` as an argument.

init.cpp#AppInitMain()

```
assert(!node.mempool);
int check_ratio = std::min<int>(std::max<int>(args.GetIntArg("-checkmempool",
chainparams.DefaultConsistencyChecks() ? 1 : 0), 0), 1000000);
node.mempool = std::make_unique<CTxMemPool>(node.fee_estimator.get(), check_ratio);
```

The `check_ratio`, used to determine sanity checks, defaults to `0` for all networks except regtest, unless the `checkmempool` program option has been specified.

Sanity checking here refers to checking the consistency of the entire mempool every 1 in `n` times a new transaction is added, so is potentially computationally expensive to have enabled.
See `CTxMemPool::Check()` for more information on what the check does.

Loading a previous mempool

If the node has been run before then it might have some blocks and a mempool to load. "Step 11: import blocks" of `AppInitMain()` in *init.cpp* calls `ThreadImport()` to load the mempool from disk where it is saved to file `mempool.dat`:

init.cpp#AppInitMain()

```
    chainman.m_load_block = std::thread(&TraceThread<std::function<void()>>,
"loadblk", [=, &chainman, &args] {
        ThreadImport(chainman, vImportFiles, args);
    });
```

This is run in its own thread so that (potentially) slow disk I/O has a minimal impact on startup times, and the remainder of startup execution can be continued.

`ThreadImport` runs a few jobs sequentially:

1. Optionally perform a reindex
2. Load the block files from disk
3. Check that we are still on the best chain according to the blocks loaded from disk
4. Load the mempool via `chainman.ActiveChainstate().LoadMempool(args);`

`validation.cpp#LoadMempool()` is an almost mirror of `DumpMempool()` described in more detail below in Mempool shutdown:

1. Read the version and count of serialized transactions to follow
2. Test each tx for expiry before submitting it to MemPoolAccept
3. Read any remaining `mapDeltas` and `unbroadcast_txids` from the file and apply them

We test for expiry because it is current default policy not to keep transactions in

> the mempool longer than 336 hours, i.e. two weeks.
> The default value comes from the constant `DEFAULT_MEMPOOL_EXPIRE` which can be overridden by the user with the `-mempoolexpiry` option.
> Loading (and validating) a mempool of transactions this old is likely a waste of time and resources.

Runtime execution

While the node is running the mempool is persisted in memory. By default the mempool is limited to 300MB as specified by `DEFAULT_MAX_MEMPOOL_SIZE`. This can be overridden by the program option `maxmempoolsize`.

See mempool tx format for more information on what data counts towards this limit, or review the `CTxMemPool` data members which store current usage metrics e.g. `CTxMemPool::cachedInnerUsage` and the implementation of e.g. `CTxMemPool::DynamicMemoryUsage()`.

Mempool shutdown

When the node is shut down its mempool is (by default) persisted to disk, called from `init.cpp#Shutdown()`:

init.cpp#Shutdown()

```
    if (node.mempool && node.mempool->IsLoaded() && node.args->GetArg("-
persistmempool", DEFAULT_PERSIST_MEMPOOL)) {
        DumpMempool(*node.mempool);
    }
```

A pointer to the mempool object is passed to `DumpMempool()`, which begins by locking the mempool mutex, `pool.cs`, before a snapshot of the mempool is created using local variables `mapDeltas`, `vinfo` and `unbroadcast_txids`.

`mapDeltas` is used by miners to apply (fee) prioritisation to certain transactions when creating new block templates.

`vinfo` stores information on each transaction as a vector of `CTxMemPoolInfo` objects.

validation.cpp#DumpMempool()

```
bool DumpMempool(const CTxMemPool& pool, FopenFn mockable_fopen_function, bool
skip_file_commit)
{
    int64_t start = GetTimeMicros();

    std::map<uint256, CAmount> mapDeltas;
    std::vector<TxMempoolInfo> vinfo;
    std::set<uint256> unbroadcast_txids;
```

```
    static Mutex dump_mutex;
    LOCK(dump_mutex);

    {
        LOCK(pool.cs);
        for (const auto &i : pool.mapDeltas) {
            mapDeltas[i.first] = i.second;
        }
        vinfo = pool.infoAll();
        unbroadcast_txids = pool.GetUnbroadcastTxs();
    }
```

Next a new (temporary) file is opened and some metadata related to mempool version and size is written to the front. Afterwards we loop through `vinfo` writing the transaction, the time it entered the mempool and the fee delta (prioritisation) to the file, before deleting its entry from our `mapDeltas` mirror.

Finally, any remaining info in `mapDeltas` is appended to the file. This might include prioritisation information on transactions not in our mempool.

validation.cpp#DumpMempool()

```
    // ...
    try {
        FILE* filestr{mockable_fopen_function(GetDataDir() / "mempool.dat.new",
"wb")};
        if (!filestr) {
            return false;
        }

        CAutoFile file(filestr, SER_DISK, CLIENT_VERSION);

        uint64_t version = MEMPOOL_DUMP_VERSION;
        file << version;

        file << (uint64_t)vinfo.size();
        for (const auto& i : vinfo) {
            file << *(i.tx);
            file << int64_t{count_seconds(i.m_time)};
            file << int64_t{i.nFeeDelta};
            mapDeltas.erase(i.tx->GetHash());
        }

        file << mapDeltas;

        LogPrintf("Writing %d unbroadcast transactions to disk.\n", unbroadcast_txids
.size());
        file << unbroadcast_txids;
    // ...
}
```

We are able to write (and later read) `mapDeltas` and `unbroadcast_txids` to the file only using the `<<` operator. This is due to the operator overload on the `CAutoFile` class found in *streams.h*:

streams.h

```
/**
 * map
 */
template<typename Stream, typename K, typename T, typename Pred, typename A>
void Serialize(Stream& os, const std::map<K, T, Pred, A>& m)
{
    WriteCompactSize(os, m.size());
    for (const auto& entry : m)
        Serialize(os, entry);
}

class: CAutoFile
{
public:
    // ...
    template<typename T>
    CAutoFile& operator<<(const T& obj)
    {
        // Serialize to this stream
        if (!file)
            throw std::ios_base::failure("CAutoFile::operator<<: file handle is
nullptr");
        ::Serialize(*this, obj);
        return (*this);
    }
    // ...
};
```

Finally, if writing the elements to the temporary file was successful, we close the file and rename it to `mempool.dat`.

Addition to the mempool

Transactions are added to the mempool via `addUnchecked()` as part of the `AcceptToMemoryPool()` flow. See Transaction validation for more information on how this flow is entered.

The function name `addUnchecked` specifically refers to the fact that no checks are being performed, so this must not be called until policy checks have passed.

This function is called from within *validation.cpp* (`MemPoolAccept::Finalize()`) where the appropriate consensus and policy checks *have* already been performed on the transaction. The transaction is added to the primary index `mapTx` before any fee prioritisation ("delta") is applied to it.

Next any links to parent transactions are generated by looping through the inputs and mapping the `COutPoint` of the input to this transaction `CTransaction` in the `mapNextTx` map. Additionally the tx input is added to a set which is used to update parent transactions if they are still in the mempool.

After all inputs have been considered, `UpdateAncestorsOf()` is called which will add this transaction as a descendant to any ancestors in the mempool. This is followed by `UpdateEntryForAncestors()` which will re-calculate and apply descendant `count`, `size`, `fee` and `sigOpCost` of the ancestors with the new descendant being accounted for.

Finally update `totalTxSize` and `totalFee` (both sum totals of the mempool) to account for this new transaction.

Removal from the mempool

Transactions are removed from the mempool for a number of reasons:

1. A new block has been connected `removeForBlock()`
2. A re-org is taking place `removeForReorg()`
3. The transaction has expired `Expire()`
4. The transaction is being replaced by a higher-fee version `MemPoolAccept::Finalize()`
5. The mempool must be trimmed back down below its maximum size `TrimToSize()`

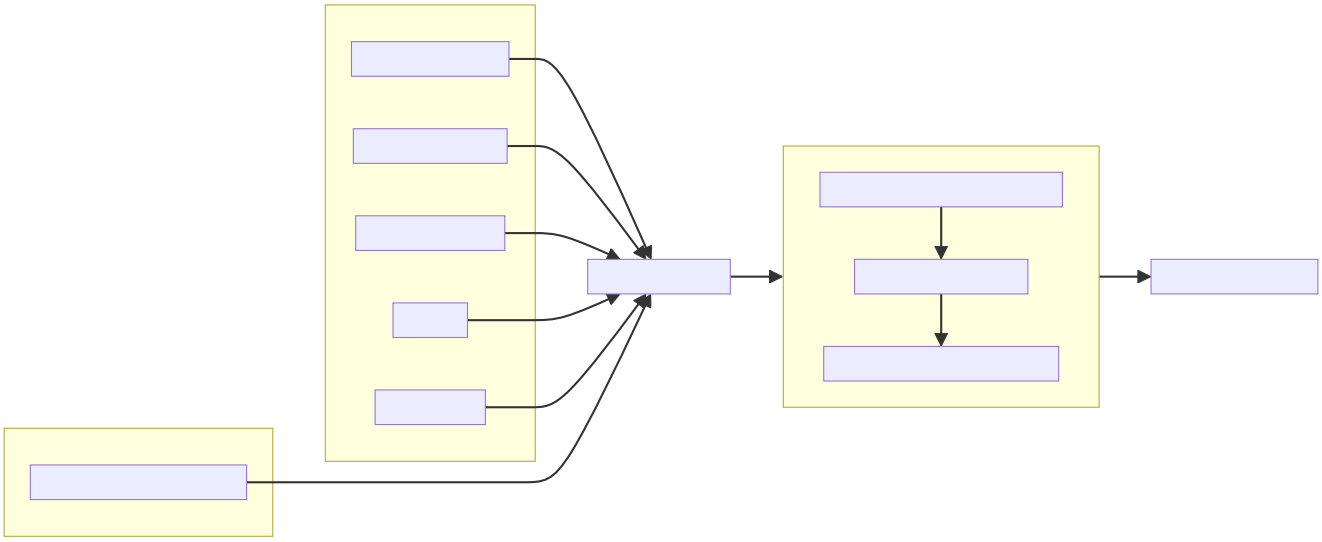

Figure 11. Removal from the mempool

`RemoveStaged()` takes a set of transactions referenced by their txid along with their removal reason, and removes them sequentially. It does this by first updating the ancestors of the transaction, followed by the descendants. After calculating and updating related transaction information it calls `removeUnchecked()` which actions the removal from the mempool.

`removeUnchecked()` starts by notifying the validation interface that a transaction has been removed from the mempool for all reasons other than a new block arriving, as there is a different `BlockConnected` signal which can be used for that.

Next it loops over the `txin`s of the transaction, and removes each `prevout` of each `txin` from the

`mapNextTx` map.

`mapNextTx` is used to map a `COutPoint` to the unconfirmed transaction spending it. This way there is a quick lookup available to check that a new transaction being added to the mempool is not trying to double spend a UTXO.

You can see the map being created as new transactions are learned about in `addUnchecked()`.

If the node has upgraded to SegWit the `vTxHashes` vector, which stores `wtxid`s is then updated. As `vTxHashes` stores the `wtxid`s in random order, first we move the transaction's entry to the back, and then pop it off, resizing the vector if needed.

Finally, as with `addUnchecked()` we update the mempool sum totals for `txSize` and `fee` and erase the transaction from the primary mempool index `mapTx`.

Both adding and removing transactions increment the `mempool_seqence` counter. This is used by the `getrawmempool` RPC (via `MempoolToJSON`) in tracking the number of mempool database transaction operations.

Mempool unbroadcast set

The mempool contains an "unbroadcast" set called `m_unbroadcast_txids`. As the name implies this stores the txids of transactions which are in our mempool but have not been verified as broadcast to the wider P2P network. This might occur for example if a transaction is submitted locally (e.g. from the wallet or RPC), but we are not yet connected to any peers.

1. When a transaction is submitted to the network via `BroadcastTransaction()` it is added to the unbroadcast set in the mempool, before `PeerManager` calls `RelayTransaction()` to attempt initial broadcast.
2. When a transaction is heard about from the P2P network (via `getdata` in response to an `INV`), the transaction will be removed from the unbroadcast set.

Transactions are also removed from the set on reorgs, new blocks arriving or if they've "expired" via `RemoveStaged()`

`PeerManager` schedules `ReattemptInitialBroadcast()` to be run every 10 minutes. This function loops over the unbroadcast set and either attempts rebroadcast or removes the transaction from the unbroadcast set if it is no longer in our mempool.

amiti wrote a gist on her proposal to improve rebroadcast logic in Bitcoin Core.

Transaction format in the mempool

A `CTXMemPoolEntry` describes a mempool entry (i.e. transaction) in the mempool. It stores not only transaction information, but also pre-computed information about ancestors.

txmempool.h

```
class CTxMemPoolEntry
{
public:
    typedef std::reference_wrapper<const CTxMemPoolEntry> CTxMemPoolEntryRef;
    // two aliases, should the types ever diverge
    typedef std::set<CTxMemPoolEntryRef, CompareIteratorByHash> Parents;
    typedef std::set<CTxMemPoolEntryRef, CompareIteratorByHash> Children;

private:
    const CTransactionRef tx;
    mutable Parents m_parents;
    mutable Children m_children;
    const CAmount nFee;              //!< Cached to avoid expensive parent-transaction
lookups
    const size_t nTxWeight;          //!< ... and avoid recomputing tx weight (also
used for GetTxSize())
    const size_t nUsageSize;         //!< ... and total memory usage
    const int64_t nTime;             //!< Local time when entering the mempool
    const unsigned int entryHeight; //!< Chain height when entering the mempool
    const bool spendsCoinbase;       //!< keep track of transactions that spend a
coinbase
    const int64_t sigOpCost;         //!< Total sigop cost
    int64_t feeDelta;          //!< Used for determining the priority of the
transaction for mining in a block
    LockPoints lockPoints;     //!< Track the height and time at which tx was final

    // Information about descendants of this transaction that are in the
    // mempool; if we remove this transaction we must remove all of these
    // descendants as well.
    uint64_t nCountWithDescendants;  //!< number of descendant transactions
    uint64_t nSizeWithDescendants;   //!< ... and size
    CAmount nModFeesWithDescendants; //!< ... and total fees (all including us)

    // Analogous statistics for ancestor transactions
    uint64_t nCountWithAncestors;
    uint64_t nSizeWithAncestors;
    CAmount nModFeesWithAncestors;
    int64_t nSigOpCostWithAncestors;

    // ...
```

The advantage to having pre-computed data on descendants and ancestors stored with each transaction in the mempool is that operations involving adding and removing transactions can be performed faster. When a transaction is added to the mempool we must update the descendant data for all ancestor `CTxMemPoolEntry`'s. Conversely if a transaction is removed from the mempool, we must also remove all of its descendants. A particular area where speed can be critical is in block template assembly.

Some of this extra transaction metadata counts towards the mempool's maximum size, therefore a default mempool of 300MB will contain less than 300MB of serialized transactions.

Mapping transactions in the mempool

A lot of how fee-maximizing block templates can be swiftly generated from chains of potentially-complex interlinked and dependant transactions comes down to `CTxMemPool`'s `boost::multi_index` `mapTx`, which is able to natively store transactions in an index against multiple criteria as described in the documentation and code comments:

txmempool.h#CTxMemPool

```
/*
 * mapTx is a boost::multi_index that sorts the mempool on 5 criteria:
 * - transaction hash (txid)
 * - witness-transaction hash (wtxid)
 * - descendant feerate [we use max(feerate of tx, feerate of tx with all
descendants)]
 * - time in mempool
 * - ancestor feerate [we use min(feerate of tx, feerate of tx with all unconfirmed
ancestors)]
 */
```

The index has 5 sort fields: the default, and tagged fields `index_by_wtxid`, `descendant_score`, `entry_time` and `ancestor_score`:

1. The default, and untagged, sort field of the index, which is using the hashed_unique sort; hashing the `txid` using Bitcoin Core's implementation of the SipHash hasher for txids.
   This is used when adding and removing transactions from the mempool, requesting and looking up mempool transactions (by txid) and checking whether RBF is enabled.
2. `index_by_wtxid` is used when checking whether transactions received over the P2P network already exist in the mempool (via the `exists()` function).
3. `descendant_score` is used when trying to trim the mempool to size (via `TrimToSize()`).
   In this case we want to keep parent (ancestor) transactions in the mempool who have high fee-paying children (descendants).
4. `entry_time` is used to calculate when transactions in the mempool should expire.
5. `ancestor_score` is used to create new block templates by selecting the most valuable effective-feerate transaction chains.

Package relay

Package Relay is a long-discussed concept and, at the time of writing, is a work in progress in Bitcoin Core. A significant portion of the project involves changes to mempool validation, which glozow describes in her gist Package mempool accept.

PR#20833 added the ability for mempool validation to assess a set of dependent transactions and enabled the `testmempoolaccept` RPC to support multiple transactions.

PR#21800 added the ability to analyse and limit the ancestor and descendant sets of packages in relation to the mempool.

PR#22674 defined child-with-unconfirmed-parents packages and enabled submission of such packages to the mempool.

These PRs were also accompanied by several refactoring efforts: PR#21062, PR#22796, PR#22675, PR#22855, PR#23381.

The document doc/policy/packages.md contains current information on the stated package acceptance rules.

Pinning attacks

glozow describes pinning attacks in her document "Pinning zoo".

Script

This section has been updated to Bitcoin Core @ v23.0

Script origins

1. New scripts are created when creating a new address.
2. Scripts can be learned about when we receive a new transaction from the P2P network or from a newly-connected block.
3. With Taproot there may be scripts in alternative Tapscript execution paths which nobody on the network will ever learn about.

Scripts in Bitcoin Core

The primary script objects are found in *script.h*. An enum over all the permitted OPCODES, `enum opcodetype`. The `CScriptNum` class which handles arithmetic operations on integer `CScriptNum`s, whether from a loose `int_64t` or from a second `CScriptNum` object. The `CScript` class which supports serializing data into scripts, along with many helper functions such as returning the script type.

Validating scripts

For some additional context on how scripts are validated in Bitcoin see executing scripts in the Appendix.

Transactions contain a vector of inputs (`CTxIn`) and vector of outputs (`CTxOut`), along with other required data.

Each `CTxIn` contains:

- `COutPoint prevout;`
- `CScript scriptSig;`
- `uint32_t nSequence;`
- `CScriptWitness scriptWitness;`

Each `CTxOut` contains:

- `CAmount nValue;`
- `CScript scriptPubKey;`

When a new transaction is learned about from the wallet or P2P network (as a TX INV) it is passed to `AcceptToMemoryPool()` which will run the various script checks.

Transactions learned about directly in blocks have their scripts validated via `ActivateBestChainStep()` -→ `ConnectBlock()` -→ `ConnectTip()` -→ `CChainState::ConnectBlock()` (link), which will end up calling `CheckTxInputs()` and `CheckInputScripts()`, as described in the subsequent section on PolicyScriptChecks.

PreCheck script checks

`PreChecks()` performs some structural checks inside of `CheckTransaction()` before passing the transaction to `IsStandard()`. In here the transaction weight is checked, along with the `scriptSig` size of every input and the type of every output. Any failures are written back into the `reason` string which will be propagated up in the case the function returns `false`.

The next script checks come after the mempool is consulted to test for conflicts, and inputs are checked against our `CoinsCache` (UTXO set). `AreInputsStandard()` will take the transaction and access each `vin` from a copy of our UTXO set `CCoinsViewCache`.

We use a cached version of `CCoinsView` here because although we want to introspect the transaction by doing a mock evaluation of the script, we do not want to modify the UTXO set yet, nor mark any coins as `DIRTY`.

The type of script can be evaluated using the *script/standard.cpp#Solver()* function, which will return the script type as a member of the `TxOutType` enum.

`Solver()` takes a scriptPubkey as a `CScript` and a vector of `unsigned_char` vectors called `vSolutionsRet`. It will attempt to evaluate and return the script type, along with any parsed pubKeys or pubKeyHashes in the `vSolutionsRet` vector.

For example, if the script type is P2SH it will execute:

```
    // Shortcut for pay-to-script-hash, which are more constrained than the other
types:
    // it is always OP_HASH160 20 [20 byte hash] OP_EQUAL
```

```
    if (scriptPubKey.IsPayToScriptHash())
    {
        std::vector<unsigned char> hashBytes(scriptPubKey.begin()+2, scriptPubKey
.begin()+22);
        vSolutionsRet.push_back(hashBytes);
        return TxoutType::SCRIPTHASH;
    }
```

In this case, simply reading the scriptHash into the `vSolutionsRet` vector before returning with the type.

For SegWit inputs the witness program is returned, for PayToPubKey (which although basically unused now is still supported) the pubKey is returned, and for P2PKH the pubKeyHash is returned. The MultiSig case returns the number of required signatures, all the pubKeys and, the total number of keys.

If the input is `NONSTANDARD` or `WITNESS_UNKNOWN` then we can return early with `false`. If the transaction is of type `SCRIPTHASH` (P2SH) then we want to check that the `scriptSig` does not have extra data included which is not relevant to the `scriptPubKey`, and that the `SigOpCount` for the input obeys the specific P2SH limits. To do this we perform a mini evaluation of the script by passing in the `SCRIPT_VERIFY_NONE` flag, which instructs the interpreter not to verify operations guarded by flags.

Looking into `EvalScript()` itself we can see which verification operations are going to be skipped by using this flag; in the positions we see the flag being tested e.g.:

```
case OP_CHECKLOCKTIMEVERIFY:
{
    if (!(flags & SCRIPT_VERIFY_CHECKLOCKTIMEVERIFY)) {
        // not enabled; treat as a NOP2
        break;
    }
```

With `SCRIPT_VERIFY_NONE` set this will skip `fRequireMinimal`, `OP_CHECKLOCKTIMEVERIFY`, `OP_CHECKSEQUENCEVERIFY`, discouragement of the upgradable `NOP`s 1; 4; 5; 6; 7; 8; 9; 10; `OP_CHECKSIG` and `OP_CHECKSIGVERIFY`. This makes the evaluation much cheaper by avoiding expensive signature verification, whilst still allowing quick testing that stack will not be empty (if signature verification succeeded), and that `MAX_P2SH_SIGOPS` count is not exceeded.

Avoiding expensive operations, e.g. full script evaluation, for as long as possible, whilst also avoiding repeating work, is a key anti-DoS consideration of transaction and script validation.

After `AreInputsStandard()` has returned, if the transaction is SegWit the witnesses are checked by `IsWitnessStandard()`. This functions similarly to `AreInputsStandard()` is that it will loop over every `vin` to the transaction and access the coin using the same `CCoinsViewCache` as used previously.

The input's script `prevScript` is initialised to the input's scriptPubKey, but then a check is done to

see if the input is of P2SH type (corresponding to a P2SH-wrapped address), again performing the mock script validation with the `SCRIPT_VERIFY_NONE` flag applied. If it is found to be P2SH-wrapped then the input's script is set to the `scriptSig` as converted into a stack.

With the input script set witness evaluation can begin. First the script is checked to be a valid witness program, i.e. a single byte `PUSH` opcode, followed by a sized data push. This is using `CScript::IsWitnessProgram()`.

Segwit V0 or V1 script size limits (as appropriate) are checked before returning `true`. The final script checks inside of `PreChecks()` are to get the full transaction sigOp cost, which is a total of the legacy, P2SH and Witness sigOps.

PolicyScriptChecks script checks

This block is going to re-use the same `Workspace` as `PreChecks`, but at this stage doesn't re-use any cached `PreComputedTransactionData`.

The main check block is shown below:

validation.cpp:982

```
    // Check input scripts and signatures.
    // This is done last to help prevent CPU exhaustion denial-of-service attacks.
    if (!CheckInputScripts(tx, state, m_view, scriptVerifyFlags, true, false, ws
.m_precomputed_txdata)) {
        // SCRIPT_VERIFY_CLEANSTACK requires SCRIPT_VERIFY_WITNESS, so we
        // need to turn both off, and compare against just turning off CLEANSTACK
        // to see if the failure is specifically due to witness validation.
        TxValidationState state_dummy; // Want reported failures to be from first
CheckInputScripts
        if (!tx.HasWitness() && CheckInputScripts(tx, state_dummy, m_view,
scriptVerifyFlags & ~(SCRIPT_VERIFY_WITNESS | SCRIPT_VERIFY_CLEANSTACK), true, false,
ws.m_precomputed_txdata) &&
                !CheckInputScripts(tx, state_dummy, m_view, scriptVerifyFlags &
~SCRIPT_VERIFY_CLEANSTACK, true, false, ws.m_precomputed_txdata)) {
            // Only the witness is missing, so the transaction itself may be fine.
            state.Invalid(TxValidationResult::TX_WITNESS_STRIPPED,
                    state.GetRejectReason(), state.GetDebugMessage());
        }
        return false; // state filled in by CheckInputScripts
    }
```

This performs validation of the input scripts using our "policy flags", where policy flags refer to a list of script verification flags that form "standard transactions", i.e. those transactions that will be relayed around the network by other nodes running the same policies.

> Notice that `CheckInputScripts()` is run up to 3 times. The first run will check all the inputs using the whole `STANDARD_SCRIPT_VERIFY_FLAGS` and `cacheSigStore` set to `true`, so that we cache expensive signature verification results. If this returns `true` then `PolicyScriptChecks()` is

complete and will also return `true` to the caller.

If this first check fails we then check to see if it is specifically a missing witness which is causing the failure. In order to do this we will execute two more runs, one with `SCRIPT_VERIFY_WITNESS` and `SCRIPT_VERIFY_CLEANSTACK` disabled which should pass, and a second in series with only `SCRIPT_VERIFY_CLEANSTACK` disabled which should fail.

From this call-site inside MempoolAccept `CheckInputScripts()` is called with `cacheSigStore` set to `true`, and `cacheFullScriptStore` set to `false`. This means that we will keep signature verifications in the `CSignatureCache` (named `signatureCache`). Full scripts will not be cached. The two caches are setup as part of `AppInitMain()`.

`CheckInputScripts()` begins by checking that we have not already executed this input script and stored it in the global Cuckoo Cache `g_scriptExecutionCacheHasher`, if we have, then this means the previous execution already succeeded so we can return `true` early. Next check that we have all our input coins loaded from the cached copy of the UTXO set `CCoinsViewCache`.

Now script execution begins by looping over each input and storing the input and transaction in a `CScriptCheck` closure (`check`) for later evaluation. Calling the `()` operator on the closure will initialize a new `CScript` and `CScriptWitness` for the evaluation, and execute `VerifyScript()`.

You can see the `cacheSigStore` boolean being propagated to the `CachingSignatureTransactionChecker` signalling that we should cache these signature evalations.

Execution of `VerifyScript` is described below.

VerifyScript

`Verifyscript()`s function is to very a single scriptSig (SS) against a scriptPubKey (SPK) and return a boolean `true` or `false`, returning a more specific error description via the passed in `ScriptError`. Historically (in Bitcoin versions < 0.3.5) this was done by concatenating the SS and the SPK and evaluating as one, however this meant that malicious actors could leave arbitrary extra objects on the stack, ultimately resulting in being able to spend coins using any scripts with what should have been an invalid SS. Therefore now evaluation takes place in two stages, first the SS, who's pre-populated `stack` is then passed in as an argument to SPK evaluation.

The mechanics of `EvalScript()` are shown in the section EvalScript.

If both calls to `EvalScript` succeed, then any witness program is verified, followed by P2SH scripts. Notice here how in each of these cases the stack is trimmed to size `1` at the end of evaluation, because in both cases extra elements would ordinarily remain on the stack (P2SH and witness inputs). If the evaluation succeeds then the `CLEANSTACK` rule is enforced afterwards.

EvalScript

`EvalScript()` handles the Forth-like script interpretation itself. It takes in a stack, script, interpretation flags, a signature checker, a signature version and a `ScriptExecutionData` struct.

After checking that it's not about to evaluate a Taproot key-path spend (`SIGVERSION::TAPROOT`), which has no script to execute, we initialize some iterators on the script, along with variables to represent the current opcode, the push value, the condition stack and the altstack. The condition stack is used to help evaluation of IF/ELSE clauses and the altstack is used to push and pop items from the main stack during execution (using `OP_TOALTSTACK` and `OP_FROMALTSTACK`).

Next we check script size is less that `MAX_SCRIPT_SIZE` (10KB). Although total serialized transaction size, and SigOpCount has been checked previously, this is the first time the size of the scripts themselves are checked.

Then comes the main evaluation for loop. Whilst many conditions are checked, and specific invalidity errors returned, there is also the possibility of other un-tested errors occurring during evaluation, and so the loop is enclosed by a try-except block which will catch these errors, instead of causing a program crash.

Script execution is effectively executing uncontrolled, 3rd party data. If a malicious actor found a way to purposefully provoke an unhandled error during evaluation, without the try-catch block, they would be able to effectively crash any node on the network of their choosing by sending it the malicious script.

The main loop is simple conceptually:

1. Read an instruction using the `CScript::GetOp()` method. This will read an `opcodetype` into the `opcode` variable, and the raw instruction into the `vchPushValue` variable.
2. Test for the script element size, number of script ops, and whether this is a disabled opcode.
3. Enter a switch on `opcode` to perform specific evaluation according to the operation specified.

Signing a transaction

`script/sign.cpp#SignTransaction()` will sign a transaction one input at a time, by looping through the `vin`s of the `CMutableTransaction` it has been passed.

The critical section of the `SignTransaction()` loop is shown below:

src/script/sign.cpp#SignTransaction()

```
    for (unsigned int i = 0; i < mtx.vin.size(); i++) {
        CTxIn& txin = mtx.vin[i];
        auto coin = coins.find(txin.prevout);
        if (coin == coins.end() || coin->second.IsSpent()) {
            input_errors[i] = "Input not found or already spent";
            continue;
        }
        const CScript& prevPubKey = coin->second.out.scriptPubKey;
        const CAmount& amount = coin->second.out.nValue;

        SignatureData sigdata = DataFromTransaction(mtx, i, coin->second.out);
        // Only sign SIGHASH_SINGLE if there's a corresponding output:
        if (!fHashSingle || (i < mtx.vout.size())) {
```

```
            ProduceSignature(*keystore, MutableTransactionSignatureCreator(&mtx, i,
amount, nHashType), prevPubKey, sigdata);
        }

        UpdateInput(txin, sigdata);
```

The Pubkey and amount for each coin are retrieved, along with signature data for the coin. `DataFromTransaction()` returns all the information needed to produce a signature for that coin as a `SignatureData` struct:

src/script/sign.h#SignatureData

```
// This struct contains information from a transaction input and also contains
signatures for that input.
// The information contained here can be used to create a signature and is also filled
by ProduceSignature
// in order to construct final scriptSigs and scriptWitnesses.
struct SignatureData {
    bool complete = false; ///< Stores whether the scriptSig and scriptWitness are
complete
    bool witness = false; ///< Stores whether the input this SigData corresponds to is
a witness input
    CScript scriptSig; ///< The scriptSig of an input. Contains complete signatures or
the traditional partial signatures format
    CScript redeem_script; ///< The redeemScript (if any) for the input
    CScript witness_script; ///< The witnessScript (if any) for the input.
witnessScripts are used in P2WSH outputs.
    CScriptWitness scriptWitness; ///< The scriptWitness of an input. Contains
complete signatures or the traditional partial signatures format. scriptWitness is
part of a transaction input per BIP 144.
    std::map<CKeyID, SigPair> signatures; ///< BIP 174 style partial signatures for
the input. May contain all signatures necessary for producing a final scriptSig or
scriptWitness.
    std::map<CKeyID, std::pair<CPubKey, KeyOriginInfo>> misc_pubkeys;
    std::vector<CKeyID> missing_pubkeys; ///< KeyIDs of pubkeys which could not be
found
    std::vector<CKeyID> missing_sigs; ///< KeyIDs of pubkeys for signatures which
could not be found
    uint160 missing_redeem_script; ///< ScriptID of the missing redeemScript (if any)
    uint256 missing_witness_script; ///< SHA256 of the missing witnessScript (if any)

    SignatureData() {}
    explicit SignatureData(const CScript& script) : scriptSig(script) {}
    void MergeSignatureData(SignatureData sigdata);
};
```

With the signing `SigningProvider`, `scriptPubKey` and `sigdata` we are able to call `script/sign.cpp#ProduceSignature()` for signing on each individual input. Inputs by default will signed with a sighash of `SIGHASH_ALL`, but this can be re-configured as appropriate.

Producing a signature

Taking a look inside `ProduceSignature()` we can see how this works.

src/script/sign.cpp

```
bool ProduceSignature(const SigningProvider& provider, const BaseSignatureCreator&
creator, const CScript& fromPubKey, SignatureData& sigdata)
{
    if (sigdata.complete) return true;

    std::vector<valtype> result;
    TxoutType whichType;
    bool solved = SignStep(provider, creator, fromPubKey, result, whichType,
SigVersion::BASE, sigdata);
    bool P2SH = false;
    CScript subscript;
    sigdata.scriptWitness.stack.clear();

    // ...
}
```

The function performs some initialisations before calling `script/sign.cpp#SignStep()` for the first time, with the `SigVersion SIGVERSION::BASE`. `SignStep()` in turn calls `Solver()`, which is a function designed to detect the script type encoding of the `scriptPubKey`, and then return the detected type along with the parsed `scriptPubKey`s/hashes.

If it is successful, `SignStep` continues by switching over the script type and, depending on the script type, calling the required signing operation and pushing the required elements onto the `sigdata` variable.

script/sign.cpp

```
static bool SignStep(const SigningProvider& provider, const BaseSignatureCreator&
creator, const CScript& scriptPubKey,
                     std::vector<valtype>& ret, TxoutType& whichTypeRet, SigVersion
sigversion, SignatureData& sigdata)
{
    // ...
    whichTypeRet = Solver(scriptPubKey, vSolutions);

    switch (whichTypeRet) {
    case TxoutType::NONSTANDARD:
    case TxoutType::NULL_DATA:
    case TxoutType::WITNESS_UNKNOWN:
    case TxoutType::WITNESS_V1_TAPROOT:
        // ...
    case TxoutType::PUBKEY:
        // ...
    case TxoutType::PUBKEYHASH:
```

```
        // ...
    case TxoutType::SCRIPTHASH:
        // ...
    case TxoutType::MULTISIG:
        // ...
    case TxoutType::WITNESS_V0_KEYHASH:
        // ...
    case TxoutType::WITNESS_V0_SCRIPTHASH:
        // ...
    }
    // ...
}
```

Once `SignStep()` returns to `ProduceSignature()`, a second switch takes place. If we are trying to produce a signature for P2SH, P2WPKH or P2WSH then the first pass from `SignStep()` will have been enough to detect the `TxOutType` and assemble the (redeem/witness) scripts, but not yet generate the entire signature in required format. In order to get this signature, `SignStep()` is called again, this time with the assembled redeem/witness script and the appropriate `TxOutType`.

This recursion makes sense if you consider that, in order to sign for these script-encumbered inputs, we don't want to sign for the `scriptPubKey` that we are starting with but for the {redeem|witness} script instead.

We can see this switch in `ProduceSignature()`:

src/script/sign.cpp#ProduceSignature()

```
    if (solved && whichType == TxoutType::SCRIPTHASH)
    {
        // Solver returns the subscript that needs to be evaluated;
        // the final scriptSig is the signatures from that
        // and then the serialized subscript:
        subscript = CScript(result[0].begin(), result[0].end());
        sigdata.redeem_script = subscript;
        solved = solved && SignStep(provider, creator, subscript, result, whichType,
SigVersion::BASE, sigdata) && whichType != TxoutType::SCRIPTHASH;
        P2SH = true;
    }

    if (solved && whichType == TxoutType::WITNESS_V0_KEYHASH)
    {
        CScript witnessscript;
        // This puts the parsed pubkeys from the first pass into the witness script
        witnessscript << OP_DUP << OP_HASH160 << ToByteVector(result[0]) <<
OP_EQUALVERIFY << OP_CHECKSIG;
        TxoutType subType;
        solved = solved && SignStep(provider, creator, witnessscript, result, subType,
SigVersion::WITNESS_V0, sigdata);
        sigdata.scriptWitness.stack = result;
        sigdata.witness = true;
```

```
        result.clear();
    }
    else if (solved && whichType == TxoutType::WITNESS_V0_SCRIPTHASH)
    {
        CScript witnessscript(result[0].begin(), result[0].end());
        sigdata.witness_script = witnessscript;
        TxoutType subType;
        solved = solved && SignStep(provider, creator, witnessscript, result, subType,
SigVersion::WITNESS_V0, sigdata) && subType != TxoutType::SCRIPTHASH && subType !=
TxoutType::WITNESS_V0_SCRIPTHASH && subType != TxoutType::WITNESS_V0_KEYHASH;
        result.push_back(std::vector<unsigned char>(witnessscript.begin(),
witnessscript.end()));
        sigdata.scriptWitness.stack = result;
        sigdata.witness = true;
        result.clear();
    } else if (solved && whichType == TxoutType::WITNESS_UNKNOWN) {
        sigdata.witness = true;
    }
```

Finally, if all went well the signature is checked with `VerifyScript()`.

Creating a signature

TODO: dig into `CreateSig()`

Working with bitcoin script from the command line

Blockchain Commons contains guides related to Bitcoin Script including:

- Accessing scripts from transactions with `bitcoin-cli`
- How scripts are evaluated
- Testing bitcoin scripts using `btcdeb`

Appendix

Executing scripts

Bitcoin differs from most other cryptocurrencies by not including the script with the unspent transaction output on the blockchain, only the scriptPubKey is publicly viewable until spending time. The practical effects of this are:

- Users wishing to sign transactions which are locked using locking scripts require **two** pieces of information:
  a. The relevant private key(s)
  b. The `redeemScript`, i.e. the contract of the script itself.

Scripts are executed by first evaluating the unlocking script, then evaluating the locking script on the same stack. If both of these steps result in a `1` (or any other non-zero value) being the only item on the stack, the script is verified as `true`.

TODO: Not true exactly: https://bitcoin.stackexchange.com/questions/112439/how-can-the-genesis-block-contain-arbitrary-data-on-it-if-the-script-is-invalid

If any of the following are true, the script will evaluate to `false`:

- The final stack is empty
- The top element on the stack is `0`
- There is more than one element remaining on the stack
- The script returns prematurely

There are a number of other ways which scripts can fail TODO

Script inside of addresses

Bitcoin addresses can be of a "script hash" type (P2SH, and now P2WSH). As the name implies a valid script is created before being hashed. This hash is then used to generate an address which coins can be sent to. Once coins have been received to this address a (redeem / witness) script which hashes to the same hash must be provided (`scriptPubKey`), along with a satisfactory `scriptSig` in order to authorize a new spend.

The origins of this revolutionary (at the time) style of address are touched upon in this email from ZmnSCPxj. The general context of the email is recursive covenants. A portion of the email is quoted below for convenience:

> Covenants were first expressed as a possibility, I believe, during discussions around P2SH. Basically, at the time, the problem was this:
>
> - Some receivers wanted to use k-of-n multisignature for improved security.
> - The only way to implement this, pre-P2SH, was by putting in the `scriptPubKey` all the public keys.
> - The sender is the one paying for the size of the `scriptPubKey`.
> - It was considered unfair that the sender is paying for the security of the receiver.
>
> Thus, `OP_EVAL` and the P2SH concept was conceived. Instead of the `scriptPubKey` containing the k-of-n multisignature, you create a separate script containing the public keys, then hash it, and the `scriptPubKey` would contain the hash of the script. By symmetry with the P2PKH template:

```
OP_DUP OP_HASH160 <hash160(pubkey)> OP_EQUALVERIFY OP_CHECKSIG
```

The P2SH template would be:

```
OP_DUP OP_HASH160 <hash160(redeemScript)> OP_EQUALVERIFY OP_EVAL
```

`OP_EVAL` would take the stack top vector and treat it as a Bitcoin SCRIPT.

It was then pointed out that `OP_EVAL` could be used to create recursive SCRIPTs by quining using `OP_CAT`. `OP_CAT` was already disabled by then, but people were talking about re-enabling it somehow by restricting the output size of `OP_CAT` to limit the O(2^N) behavior.

Thus, since then, `OP_CAT` has been associated with **recursive** covenants (and people are now reluctant to re-enable it even with a limit on its output size, because recursive covenants). In particular, `OP_CAT` in combination with `OP_CHECKSIGFROMSTACK` and `OP_CHECKSIG`, you could get a deferred `OP_EVAL` and then use `OP_CAT` too to quine.

Because of those concerns, the modern P2SH is now "just a template" with an implicit `OP_EVAL` of the `redeemScript`, but without any `OP_EVAL` being actually enabled.

— ZmnSCPxj

For more details refer to BIP16 which introduced P2SH addresses.

Build system

```
return Error("Build system section not implemented yet!");
```

RPC / REST / ZMQ

This section has been updated to Bitcoin Core @ v24.0.1

Adding new RPCs

When trying to expose new information to users there are generally two possible approaches for developers to consider:

1. Add a new server-side RPC which directly delivers the new data
2. Create a new function on the client-side (e.g. the cli tool `bitcoin-cli`) which calls one or more existing RPCs and manipulates the results into the required format.

If the data is not available from existing RPCs then option 1) must be taken. However if option 2) is available then this is the preferred first choice. This is for the following reasons:

- Minimalistic approach: put client-side functionality into the client, not the server
  - Adding server-side increases maintenance burden
- Client-side functionality does not have to worry about API stability (as the RPCs do)
- Functions can more easily start client side and migrate to server-side if heavily used, than visa-versa

There may be other considerations though too:

- If this functionality might be wanted in multiple clients, e.g. `bitcoin-cli` and the GUI `bitcoin-qt`, then rather than making two implementations in two clients it may make sense to add a single server-side function
- Doing expensive computations on the client side when an inexpensive pathway is available server-side

Ultimately there is no "correct" answer for all cases, but considering some of the above before implementing one way or another

HTTP Server

Bitcoin Core's HTTP server is responsible for handling both RPC and REST requests, but not ZMQ. Since PR#5677 the server is based on libevent2. Libevent is a general purpose event notification library, but is used in Bitcoin Core specifically for HTTP requests (which it supports natively).

Much (not all) of the libevent interface is hidden behind wrappers. For example, `HTTPRequest` wraps `evhttp_request` and `HTTPEvent` wraps `event_base`.

The relevant workflow for how (for example) an RPC request is handled is roughly as follows:

1. The HTTP server receives an RPC command from a caller, creates an `evhttp_request` object and passes its pointer to `http_request_cb()` (this step is completely handled by libevent).
2. An `HTTPWorkItem` is created, containing the `evhttp_request` (wrapped in `HTTPRequest hreq`) as well as the path and reference to the handler function (which contains the business logic to be executed to deal with the request).
   - There are 2 handlers for RPCs.
   - There are 12 handlers for REST.
3. The `HTTPWorkItem` is put on the global `WorkQueue g_work_queue`, which is processed by multiple worker threads asynchronously.
4. When the handler function of a `HTTPWorkItem` completes successfully, it calls

`HTTPRequest::WriteReply()`, which triggers the libevent function `evhttp_send_reply()`, which in turn returns a response to the caller and destroys the `evhttp_request` object.

Endpoints are registered to the HTTP server by calling `RegisterHTTPHandler()`, such as e.g. in `StartHTTPRPC()`.

The HTTP server is initiated and started from `AppInitServers()`, and stopped from `Shutdown()`.

`StartHTTPServer()` adds a thread for each worker to `g_thread_http_workers`. These threads will keep running until `WorkQueue::Interrupt()` sets `running` to `false` and the queue is empty.

Appendix

PIMPL technique

The Bitcoin Core codebase contains many classes of the form `class *Impl`. These classes are taking advantage of the Pointer to Implementation technique which helps to both provide more stable ABIs and also to reduce compile-time dependencies.

Some of the current Bitcoin Core PIMPL classes

```
AddrManImpl
ChainImpl
NodeImpl
PeerManagerImpl
WalletImpl

FieldImpl
DBImpl
ExternalSignerImpl
NotificationsHandlerImpl
RPCHandlerImpl
IpcImpl
ProcessImpl
RPCMethodImpl
SketchImpl
DescriptorImpl
```

Amiti Uttarwar hosted a PR review club "Pimpl AddrMan to abstract implementation details" which contains information on the design aims, advantages and disadvantages. Below are copies of the annotated pictures she created and included to assist learning.

[pimpl peerman amiti] | *pimpl_peerman_amiti.png*

Figure 12. PIMPL peerman

[pimpl txrequest amiti] | *pimpl_txrequest_amiti.png*

Figure 13. PIMPL txrequest

[compilation firewall amiti] | *compilation_firewall_amiti.jpg*

Figure 14. Compilation firewall

Glossary

A

Address

A string consisting of alphanumerics from the encoding scheme used, e.g. `bc1qcwfw5vekqeyx3j8negc4lltdafg3dpqs6cw24n`. The exact format specifications of the string vary by address type. Just as you ask others to send an email to your email address, you would ask others to send you bitcoin to one of your Bitcoin addresses.

B

BIP

Bitcoin Improvement Proposals. A set of proposals that members of the bitcoin community have submitted to improve bitcoin. For example, BIP-21 is a proposal to improve the bitcoin uniform resource identifier (URI) scheme.

Bitcoin

The name of the currency unit (the coin), the network, and the software.

Block

An ordered grouping of valid transactions, marked with a timestamp and a hash of the previous block. The block header is hashed to produce a proof of work, thereby validating the transactions. Valid blocks are added to the most-work chain by network consensus.

Blockchain

A chain of validated blocks, each linking to its predecessor all the way to the genesis block.

Block Fees

The difference between the total input and output amounts for all transactions included in the block are able to be claimed by the miner in the Coinbase Transaction.

Block Reward

The total of a Block Subsidy + Block fees.

Block Subsidy

An amount included in each new block as a reward by the network to the miner who found the Proof-of-Work solution. Approximately every four years, or more accurately every 210,000 blocks, the block reward is halved. It is currently 6.25 BTC per block.

Byzantine Generals Problem

A reliable computer system must be able to cope with the failure of one or more of its components. A failed component may exhibit a type of behavior that is often overlooked—

namely, sending conflicting information to different parts of the system. The problem of coping with this type of failure is expressed abstractly as the Byzantine Generals Problem.

C

Candidate Block

A block that a miner is still trying to mine. It is not yet a valid block, because it does not contain a valid Proof-of-Work.

Child-Pays-For-Parent (CPFP)

A Child Pays For Parent (CPFP) transaction is one where you pay a high fee to incentivize miners to also confirm the unconfirmed transaction from which you are drawing the inputs i.e. the parent transaction.

CKD

Child key derivation (CKD) functions. Given a parent extended key and an index i, it is possible to compute the corresponding child extended key. The algorithm to do so depends on whether the child is a hardened key or not (or, equivalently, whether $i \geq 231$), and whether we're talking about private or public keys. Read More

Coinbase (aka coinbase data)

A special field used as the sole input for coinbase transactions. The coinbase data field allows claiming the block reward and provides up to 100 bytes for arbitrary data. Not to be confused with coinbase transaction or coinbase reward.

Coinbase Transaction

The first transaction in a block. Always created by a miner, it includes a single coinbase. Not to be confused with coinbase (coinbase data) or coinbase reward

Cold Storage

When bitcoin private keys are created and stored in a secure offline environment. Cold storage is important for anyone with bitcoin holdings. Online computers are vulnerable to hackers and should not be used to store a significant amount of bitcoin.

Confirmation

Once a transaction is included in a block, it has one confirmation. As soon as *another* block is mined on the same chain tip, the transaction has two confirmations, and so on. Six or more confirmations is considered sufficient proof that a transaction cannot be reversed.

Consensus

When several nodes, usually most nodes on the network, all have the same blocks in their locally-validated best block chain. Not to be confused with consensus rules.

Consensus Rules

The block validation rules that full nodes follow to stay in consensus with other nodes. Not to be confused with consensus.

CSV

`CHECKSEQUENCEVERIFY` or `CSV` is an opcode for the Bitcoin scripting system that in combination with BIP 68 allows execution pathways of a script to be restricted based on the age of the UTXO being spent. BIP 0112

CLTV

`CHECKLOCKTIMEVERIFY` or `CTLV` is an opcode for the Bitcoin scripting system that allows a transaction output to be made unspendable until some point in the future. i.e. a coin cannot be spent until a certain time or blockchain height has been past. BIP 65

D

Difficulty

A network-wide consensus parameter that controls how much computation is required to produce a proof of work.

Difficulty Re-targeting

A network-wide recalculation of the difficulty that occurs once every 2,016 blocks and considers the hashing power of the previous 2,015 blocks (due to an off-by-one error).

Difficulty Target

A difficulty at which all the computation in the network will find blocks approximately every 10 minutes.

Double-Spending

Double spending is the result of successfully spending the same coin more than once. Bitcoin protects against double-spending by verifying each transaction added to the block chain to ensure that the inputs for the transaction had not previously already been spent.

E

ECDSA

Elliptic Curve Digital Signature Algorithm or ECDSA is a cryptographic algorithm used by bitcoin to ensure that funds can only be spent by the owner of the associated private key.

Extra Nonce

As difficulty increased, miners often cycled through all 4 billion values of the nonce without finding a block. Because the coinbase script can store between 2 and 100 bytes of data, miners started using that space as extra nonce space, allowing them to explore a much larger range of block header values to find valid blocks.

F

Fees

The sender of a transaction often includes a fee to the network for processing the requested transaction. Most transactions require a minimum fee of 0.5 mBTC.

Fork

Fork, also known as accidental fork, occurs when two or more blocks have the same block height, forking the block chain. Typically occurs when two or more miners find blocks at nearly the same time. Can also happen as part of an attack.

G

Genesis Block

The first block in the blockchain, used as the root for all future blocks. The bitcoin genesis block has an unspendable Coinbase Output.

H

Halving

A halving event occurs when the block reward is cut in half, which happens approximately every four years (or precisely every 210,000 blocks). Bitcoin already had three halving events: in 2012 (from 50 to 25 BTC), in 2016 (from 25 to 12.5 BTC), and in 2020 (from 12.5 to 6.25 BTC).

Hard Fork

A loosening of consensus rules, such that transactions obeying the new ruleset *may* appear invalid to old, un-upgraded nodes. Not to be confused with fork, soft fork, software fork or Git fork.

Hardware Wallet

A hardware wallet is a special type of bitcoin wallet which stores the user's private keys in a secure hardware device.

Hash

A digital fingerprint of some binary input.

Hashlocks

A hashlock is a type of encumbrance that restricts the spending of an output until a specified piece of data is publicly revealed. Hashlocks have the useful property that once any hashlock is opened publicly, any other hashlock secured using the same key can also be opened. This makes it possible to create multiple outputs that are all encumbered by the same hashlock and which all become spendable at the same time.

HD Protocol

The Hierarchical Deterministic (HD) key creation and transfer protocol (BIP-32), which allows creating child keys from parent keys in a hierarchy.

HD Wallet

Wallets using the Hierarchical Deterministic (HD Protocol) key creation and transfer protocol (BIP-32).

HD Wallet Seed

HD wallet seed or root seed is a potentially-short value used as a seed to generate the master

private key and master chain code for an HD wallet.

HTLC

A Hashed Time Lock Contract or HTLC is a class of payments that use hashlocks and timelocks to require that the receiver of a payment either acknowledge receiving the payment prior to a deadline by generating cryptographic proof of payment or forfeit the ability to claim the payment, allowing it to be claimed back by the sender.

K

KYC

Know your customer (KYC) is the process of a business, identifying and verifying the identity of its clients. The term is also used to refer to the bank regulation which governs these activities.

L

LevelDB

LevelDB is an open source on-disk key-value store. LevelDB is a light-weight, single-purpose library for persistence with bindings to many platforms.

Lightning Network

Lightning Network is an implementation of Hashed Timelock Contracts (HTLCs) with bi-directional payment channels which allows payments to be securely routed across multiple peer-to-peer payment channels. This allows the formation of a network where any peer on the network can pay any other peer even if they don't directly have a channel open between each other.

Locktime

Locktime, or more technically `nLockTime`, is the part of a transaction which indicates the earliest time or earliest block when that transaction may be added to the block chain.

M

Mempool

The mempool (memory pool) is a collection of all the valid transactions which have been learned about from the P2P network, but have not yet been confirmed in a block. Whilst nodes must stay in consensus about which transactions are in each block, they may have (slightly) different mempools to each other due to transaction propagation delays, amongst other things.

Merkle Root

The root node of a merkle tree, a descendant of all the hashed pairs in the tree. Block headers must include a valid merkle root descended from all transactions in that block.

Merkle Tree

A tree constructed by hashing paired data (the leaves), then pairing and hashing the results until a single hash remains, the merkle root. In bitcoin, the leaves are almost always transactions

from a single block.

Miner

A network node that finds valid proof of work for new blocks, by repeated hashing of the Block Header until they find a Hash which is lower than the current Difficulty.

Mining Reward

Also known as Block Reward. The reward miners receive in return for the security provided by mining. Includes the new coins created with each new block, also known as a block reward or coinbase reward, and the transaction fees from all the transactions included in the block.

Multisignature

Multisignature (multisig) transactions require signatures from multiple keys to authorize a transaction using an `m-of-m` scheme. Also see Threshold Multisignature.

N

Network

A peer-to-peer network that propagates transactions and blocks to every Bitcoin node on the network.

Nonce

The "nonce" in a bitcoin block is a 32-bit (4-byte) field whose value is permuted by miners until the hash of the block will contain a run of leading zeros.

O

Off-chain Transactions

An off-chain transaction is a movement of on-chain coins which is not immediately reflected on the main block chain, e.g. a payment through a Lightning Channel. While an on-chain transaction — usually referred to as simply *a transaction* — modifies the blockchain and depends on the blockchain to determine its validity an off-chain transaction relies on other methods to record and validate the transaction, and may require "settlement" on-chain again at some point in the future.

Opcode

Operation codes from the Bitcoin Scripting language which push data or perform functions within a pubkey script or signature script.

OP_RETURN

An opcode used in one of the outputs in an `OP_RETURN` Transaction. Not to be confused with `OP_RETURN` transaction.

OP_RETURN Transaction

A transaction type that adds arbitrary data to a provably unspendable pubkey script that full nodes don’t have to store in their UTXO database. Not to be confused with OP_RETURN opcode.

Orphan Block

Blocks whose parent block has not been processed by the local node, so they can't be fully validated yet. Orphan blocks are usually cached rather than discarded, in case they make up the most-work chain in the future. Relatively rare as of 2022. Not to be confused with Stale Block.

Orphan Transactions

Transactions that can't go into the Mempool due to one or more missing inputs.

Output

Output, transaction output, or `TxOut` is an output of a transaction which contains two fields: a value field for transferring zero or more satoshis and a pubkey script for indicating what conditions must be fulfilled for those satoshis to be spent when this Output is used as an input to a future transaction.

P

P2PKH

P2PKH (Pay-To-PubKey-Hash) is script pattern formed from hashing the pubkey being used to encumber the output. An output locked by a P2PKH script can be unlocked (spent) by presenting a public key (which hashes to the same value) and a digital signature created by the corresponding private key.

P2SH

P2SH or (Pay-to-Script-Hash) is script pattern that greatly simplifies the use of complex transaction scripts, as well as reduces transaction fees for the sender. The script that encumbers the output (redeem script) is not presented in the locking script. Instead, only a hash of it is in the locking script requiring the recipient to provide the script in their redeem script on spending it in the future.

P2SH Address

P2SH addresses are Base58Check encodings of the 20-byte hash of a script. They use the version prefix "5", which results in Base58Check-encoded addresses that start with a "3". P2SH addresses hide all of the complexity, so that the person making a payment does not see the script.

P2WPKH

The signature of a P2WPKH (Pay-to-Witness-Public-Key-Hash) contains the same information as P2PKH, but is located in the witness field instead of the scriptSig field. The `scriptPubKey` is also modified.

P2WSH

The difference between P2SH and P2WSH (Pay-to-Witness-Script-Hash) is about the cryptographic proof location change from the scriptSig field to the witness field and the scriptPubKey that is also modified.

Paper Wallet

In the most specific sense, a paper wallet is a document containing one or more Private Keys. However, people often use the term to mean any way of storing bitcoin offline as a physical

document. This second definition also includes paper keys and redeemable codes.

Passphrase

A passphrase is an optional string created by the user that serves as an additional security factor protecting a wallet seed. It can also be used as a form of plausible deniability, where a chosen passphrase leads to a wallet with a small amount of funds used to distract an attacker from the “real” wallet that contains the majority of funds, when two different passphrases are used on the same Seed.

Payment Channel

A micropayment channel or payment channel is a class of techniques designed to allow users to make multiple bitcoin transactions without committing all of the transactions to the Bitcoin blockchain. In a typical payment channel, only two transactions are added to the block chain but an unlimited or nearly unlimited number of payments can be made between the participants.

Pooled Mining

Pooled mining is a mining approach where multiple generating clients contribute to the generation of a block, and then split the block reward according the contributed processing power.

Proof-of-Work

A hash adhering to a pattern that requires significant computation to find, therefore "proving" work was done to find it (on average). Miners must construct a block template which, when hashed using SHA256 (the work), will have a value at or below a network-wide Difficulty Target.

Partially Spent Bitcoin Transaction (PSBT)

The Partially Signed Bitcoin Transaction (PSBT) format consists of key-value maps. Each map consists of a sequence of key-value records, terminated by a 0x00 byte. BIP 174 and V2 BIP 370

R

RBF

The concept of replace-by-fee or RBF was developed by requiring replacements to pay for not only its own cost, but also the fee of the transactions being replaced, the DoS risk was strictly less than the risk of flooding with separate transactions. Read More

RIPEMD-160

A 160-bit cryptographic hash function. A strengthened version of RIPEMD with a 160-bit hash result, expected to be secure for the next ten years or more. Used in bitcoin as a second hash, resulting in shorter outputs, when hashing a Public Key to an Address.

S

Satoshi

A Satoshi is the base denomination of coins on the Bitcoin network used in all transactions and validation. "1 Bitcoin" is just an abstraction representing 1*10^8 satoshis which presented to users as a convenience to avoid them interacting with large number powers during network

bootstrapping. Displaying bitcoin payment values relative to "1 Bitcoin", e.g. "Send 0.0015 bitcoin to bc1qfw..." is merely continuation of this abstraction. Named after Satoshi Nakamoto.

Satoshi Nakamoto

Satoshi Nakamoto is the name or pseudonym used by the person or group who designed bitcoin and created its original reference implementation. As a part of the implementation, they also devised the first blockchain database. In the process they were the first to solve the double-spending problem for digital currency. Their real identity remains unknown.

Script

Bitcoin uses a scripting system for transactions. Forth-like, Script is simple, stack-based, and processed from left to right. It is purposefully not Turing-complete, with no loops.

ScriptPubKey (aka pubkey script)

ScriptPubKey or pubkey script, is a script included in outputs which sets the conditions that must be fulfilled for those satoshis to be spent. Data for fulfilling the conditions can be provided in a signature script.

ScriptSig (aka signature script)

ScriptSig or signature script, is the data generated by a spender which is almost always used as variables to satisfy a pubkey script.

Secret Key (aka private key)

A point on the secp256k1 curve which can be used as a private key in an ECDSA signature operation to authorize spending of Bitcoins. A secret key might take the form:

```
5J76sF8L5jTtzE96r66Sf8cka9y44wdpJjMwCxR3tzLh3ibVPxh
```

Segregated Witness

An upgrade to the Bitcoin protocol in which signature ("witness") data is separated from sender/receiver data to further optimize the structure of transactions. It was implemented as a Soft Fork.

SHA

The Secure Hash Algorithm or SHA is a family of cryptographic hash functions published by the National Institute of Standards and Technology (NIST).

Simplified Payment Verification (SPV)

SPV or simplified payment verification is a method for verifying that particular transactions were included in a block, without downloading the entire block using Merkle Proofs. This method of verification can be used by lightweight Bitcoin clients.

Soft Fork

A tightening of consensus rules, such that transactions obeying the new ruleset must appear valid to old, un-upgraded nodes. Not to be confused with fork, hard fork, software fork or Git fork.

Stale Block

A valid block that was successfully mined but that isn't included on the current most-work chain tip, because some other valid block that was mined at the same height extended the old tip first. The miner of a stale block doesn't get the block reward or the transactions fees of this block. Not to be confused with Orphan Block or Candidate Block.

Stratum (STM)

Stratum or STM is used by Pooled Miners to request new work from a centralized server.

T

Threshold Multisignature

Threshold Multisignature transactions require signatures from `n-of-m` keys to authorize a transaction. Also see Multisignature.

Timelocks

A timelock is a type of encumbrance that restricts the spending of some bitcoin until a specified future time or block height. Timelocks feature prominently in many bitcoin contracts, including payment channels and hashed timelock contracts.

Transaction

A signed data structure expressing a transfer of value from one or more UTXOs to one or more recipients. Transactions are transmitted over the Bitcoin network, collected by miners, and included into blocks, being made permanent on the blockchain.

Turing Completeness

A programming language is "Turing complete" if it can run any program that a Turing machine can run, given enough time and memory.

U

Unspent Transaction Output (UTXO)

An unspent transaction output that can be spent as an input in a new transaction with a valid `ScriptSig`.

W

Wallet

Software used to send and receive bitcoin. May store private keys, public keys, addresses or descriptors depending on wallet type and security setup and may be able to generate:

1. Addresses (derived from Descriptor or Public Keys)
2. PSBTs
3. Fully signed Transactions

Wallet Import Format (WIF)

WIF or Wallet Import Format is a data interchange format designed to allow exporting and importing a single private key with a flag indicating whether or not it uses a compressed public key.

Some contributed definitions have been sourced under a CC-BY license from the bitcoin Wiki or from other open source documentation sources.