Zcash Documentation

Release 2.1.1-1

Paige Peterson & Marshall Gaucher

Feb 07, 2020

Using Zcashd


Zcash

CHAPTER 1

What is Zcash?

Zcash is an implementation of the "Zerocash" protocol. Based on Bitcoin's code, it intends to offer a far higher standard of privacy through a sophisticated zero-knowledge proving scheme that preserves confidentiality of transaction metadata. For more technical details, please check out our Protocol Specification.

Before you get started with Zcash, please review the important items below:

Contributor Code of Conduct This project adheres to our Code of Conduct. By participating, you are expected to uphold this code.

Development Guidelines A set of guidelines and best practices to contribute to the development of Zcash.

Security Information Zcash is experimental and a work-in-progress. Use at your own risk.

Deprecation Policy A release is considered deprecated 16 weeks after the release day. There is an automatic deprecation shutdown feature which will halt the node sometime after this 16 week time period. The automatic feature is based on block height and can be explicitly disabled.

CHAPTER 2

Getting Started

For information on Zcash setup, upgrade, installation, build, configuration, and usage please see the *User Guide*.

CHAPTER 3

Need Help?

Answers to common questions from our users can be found in the FAQ.

Collaborate

Zcash development is an open collaborative process. If you'd like to contribute, join our chat system and check out some of these channels:

Chat Community Chat

Zcash General user chat

Zcash-Dev Software and Protocol Development

Community-Collaboration Other open source development related to Zcash

The-Zcash-Foundation A room to define and develop the Zcash Foundation An organization to steward the community, protocol, and science around Zcash.

Zcash-Apprentices A study and peer-education room

Zcash-Wizards Mad scientist brainstorming

CHAPTER 4

License

For license information please see *License*

4.1 User Guide

4.1.1 About

The Zcash repository is a fork of Bitcoin Core which contains protocol changes to support the Zerocash protocol. This implements the Zcash cryptocurrency, which maintains a separate ledger from the Bitcoin network, for several reasons, the most immediate of which is that the consensus protocol is different.

4.1.2 Getting Started

Welcome! This guide is intended to get you running on the official Zcash network. To ensure your Zcash client behaves gracefully throughout the setup process, please check your system meets the following requirements:

`64-bit` Linux OS
`64-bit` Processor
`5GB` of free RAM
`25GB` of free Disk (*the size of the block chain increases over time*)

Note: Currently we only officially support Linux (Debian), but we are actively investigating development for other operating systems and platforms(e.g. macOS, Ubuntu, Windows, Fedora).

Please let us know if you run into snags. We plan to make it less memory/CPU intensive and support more architectures and operating systems in the future.

If you are installing Zcash for the first time, please skip to the *Installation* section. Otherwise, the below upgrading section will provide information to update your current Zcash environment.

4.1.3 Upgrading?

If you're on a Debian-based distribution, you can follow the *Debian Binary Packages Setup* to install Zcash on your system. Otherwise, you can update your local snapshot of our code:

```
git fetch origin
```

Ensure you check the current release version from here .
If `v2.1.1-1` was current, issue the following commands:

```
git checkout v2.1.1-1
./zcutil/fetch-params.sh
./zcutil/build.sh -j$(nproc)
```

Note: If you don't have `nproc`, then substitute the number of cores on your system. If the build runs out of memory, try again without the `-j` argument, i.e. just `./zcutil/build.sh`. If you are upgrading from testnet, make sure that your `~/.zcash` directory contains only `zcash.conf` to start with, and that your `~/.zcash/zcash.conf` does not contain `testnet=1` or `addnode=testnet.z.cash`. If the build fails, move aside your `zcash` directory and try again by following the instructions in the *Installation* section below.

Important: Running `make clean` before building the update can eliminate random known link errors. If you ran into any other issues upgrading to Overwinter or Sapling, please see the *Network Upgrade Developer Guide*

4.1.4 Setup

There are a couple options to setup Zcash for the first time.

1. If you would like to install binary packages for Debian-based operating systems, see *Debian Binary Packages Setup*
2. If you would like to compile Zcash from source, please continue to the *Installation* section.
3. If you would like to install via a binary tarball download, see *Binary Tarball Download & Setup*.

4.1.5 Installation

Before we begin installing Zcash, we need to get some dependencies for your system.

UBUNTU/DEBIAN

```
sudo apt-get install \
build-essential pkg-config libc6-dev m4 g++-multilib \
autoconf libtool ncurses-dev unzip git python python-zmq \
zlib1g-dev curl bsdmainutils automake
```

Note: If you plan to cross-compile for Windows (that is, use your Linux system to build a Windows binary), there are a few additional setup steps. As of 2018-10-16 we have tested this using Ubuntu 18.04 ("Bionic Beaver").

Install the mingw-w64 package:

```
sudo apt-get install mingw-w64
```

The following two commands will display a current selection and prompt you for a new selection. Make sure the 'posix' compiler variants are selected for gcc and g++.

```
sudo update-alternatives --config x86_64-w64-mingw32-gcc
sudo update-alternatives --config x86_64-w64-mingw32-g++
```

Note: If you wish to run the test suite, you will need additional dependencies:

```
sudo apt-get install python-pip
sudo pip install pyblake2
```

FEDORA

```
sudo dnf install \
git pkgconfig automake autoconf ncurses-devel python \
python-zmq curl gtest-devel gcc gcc-c++ libtool \
patch glibc-static libstdc++-static
```

RHEL (including Scientific Linux)

- Install devtoolset-3 and autotools-latest (if not previously installed).
- Run `scl enable devtoolset-3 'scl enable autotools-latest bash'` and do the remainder of the build in the shell that this starts.

MACOS 10.12+ (Using the Terminal application)

1. **Install macOS command line tools:**

   ```
   xcode-select --install
   ```

2. **Install Homebrew:**

   ```
   /usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/
   ↪install/master/install)"
   ```

3. **Install packages:**

   ```
   brew install git pkgconfig automake autoconf libtool coreutils
   ```

4. **Install `pip` :**

   ```
   sudo easy_install pip
   ```

5. **Install python modules for rpc-tests**

   ```
   sudo pip install pyblake2 pyzmq
   ```

Note: There is an existing bug for macOS Mojave (10.14) that causes a failure in building Zcash. A work around for this includes one more step:

```
open /Library/Developer/CommandLineTools/Packages/macOS_SDK_headers_for_macOS_10.
↪14.pkg
```

CENTOS 7+

```
sudo yum install \
autoconf libtool unzip git python \
wget curl  automake gcc gcc-c++ patch \
glibc-static libstdc++-static
```

Please execute the below commands in order.

```
sudo yum install centos-release-scl-rh
sudo yum install devtoolset-3-gcc devtoolset-3-gcc-c++
sudo update-alternatives --install /usr/bin/gcc-4.9 gcc-4.9 /opt/rh/devtoolset-3/
↪root/usr/bin/gcc 10
sudo update-alternatives --install /usr/bin/g++-4.9 g++-4.9 /opt/rh/devtoolset-3/
↪root/usr/bin/g++ 10
scl enable devtoolset-3 bash
```

Note: Please see our *Supported Platform Policy* for additional details.

Dependency Version Check

Next, we need to ensure we have the correct version of `gcc` , `g++` , and `binutils`

1. **gcc/g++ 4.9 or later is required.**

   Zcash has been successfully built using gcc/g++ versions 4.9 to 7.x inclusive.

   Use `g++ --version` or `gcc --version` to check which version you have.

   On Ubuntu Trusty, if your version is too old then you can install gcc/g++ 4.9 as follows:

   ```
   $ sudo add-apt-repository ppa:ubuntu-toolchain-r/test
   $ sudo apt-get update
   $ sudo apt-get install g++-4.9
   ```

2. **binutils 2.22 or later is required.**

   Use `as --version` to check which version you have, and upgrade if necessary.

Downloading Zcash source

Now we need to get the Zcash software from the repository:

```
git clone https://github.com/zcash/zcash.git
cd zcash/
git checkout v2.1.1-1
./zcutil/fetch-params.sh
```

This will fetch the parameters generated in the Sapling MPC, and place them into `~/.zcash-params/` ``. These parameters are around 760 MB in size, so it may take some time to

download them. The message printed by ``git checkout about a "detached head" is normal and does not indicate a problem.

4.1.6 Build

Ensure you have successfully installed all system package dependencies as described above. Then run the build, e.g.:

```
./zcutil/build.sh -j$(nproc)
```

Note: To build a Windows binary on another platform (as described in the Ubuntu/Debian section above), add a HOST environment variable setting with value x86_64-w64-mingw32 to the build command, like this:

```
HOST=x86_64-w64-mingw32 ./zcutil/build.sh -j$(nproc)
```

Note: To build an ARMv8 binary (using *g++-aarch64-linux-gnu*) on another platform (as described in the Ubuntu/Debian section above), add a HOST environment variable setting with value aarch64-linux-gnu to the build command, like this:

```
HOST=aarch64-linux-gnu ./zcutil/build.sh -j$(nproc)
```

Note: If you wish to build zcashd with the Qpid Proton interface enabled, you will need an additional dependency:

```
sudo apt-get install cmake
```

Then, Proton must be enabled during the build as follows:

```
./zcutil/build.sh --enable-proton -j$(nproc)
```

Attention: If you recieved any errors, from the above command, please check out our *Troubleshooting Guide*

Note: This should compile our dependencies and build zcashd. (Note: if you don't have nproc, then substitute the number of cores on your system. If the build runs out of memory, try again without the -j argument, i.e. just ./zcutil/build.sh.

4.1.7 Configuration

Following the steps below will create your zcashd configuration file which can be edited to either connect to mainnet or testnet as well as applying settings to safely access the RPC interface.

Tip: For a complete list of parameters used in zcash.conf, please check out *Zcash.conf Guide*

Linux Create the data directory:

```
mkdir -p ~/.zcash
```

macOS Your data directory is already generated at `~/Library/Application Support/Zcash`.

Mainnet

Place a configuration file inside your data directory using the following commands:

> **Warning:** Note that this will overwrite any `zcash.conf` settings you may have added from testnet. (If you want to run on testnet, you can retain a *zcash.conf* from testnet.)

Linux

```
echo "addnode=mainnet.z.cash" >~/.zcash/zcash.conf
```

macOS

```
echo "addnode=mainnet.z.cash" >~/Library/Application Support/Zcash/zcash.conf
```

Example configured for `mainnet` :

`zcash.conf`

```
addnode=mainnet.z.cash
```

Testnet

After running the above commands to create the *zcash.conf* file, edit the following parameters in your *zcash.conf* file to indicate network and node discovery for *testnet*:

- add the line **testnet=1**
- **addnode=testnet.z.cash** instead of **addnode=mainnet.z.cash**

Example configured for `testnet`:

`zcash.conf`

```
testnet=1
addnode=testnet.z.cash
```

Enabling CPU Mining

If you want to enable CPU mining, run these commands:

Linux

```
echo 'gen=1' >> ~/.zcash/zcash.conf
echo "genproclimit=-1" >> ~/.zcash/zcash.conf
```

macOS

```
echo 'gen=1' >> ~/Library/Application Support/Zcash/zcash.conf
echo "genproclimit=-1" >> ~/Library/Application Support/Zcash/zcash.conf
```

Setting `genproclimit=-1` mines on the maximum number of threads possible on your CPU. If you want to mine with a lower number of threads, set `genproclimit` equal to the number of threads you would like to mine on.

The default miner is not efficient, but has been well reviewed. To use a much more efficient but unreviewed solver, you can run this command:

```
echo 'equihashsolver=tromp' >> ~/.zcash/zcash.conf
```

Note, you probably want to read the *Zcash Mining Guide* to learn more mining details.

4.1.8 Usage

Now, run zcashd!

```
./src/zcashd
```

To run it in the background (without the node metrics screen that is normally displayed) use `./src/zcashd --daemon`.

Important: If you are running Zcash for the first time you will need to allow your node to fully sync:

```
         :88SX@888@@X8:                    8;      %X         X%      ;8
      %%Xt%tt%SSSSS:XXXt@@               X              ::  ::              X
    @S;;tt%%%t     ;;::XXXXSX           %                SS                %
  .t::::;%8888     88888tXXXX8;        S                                    S
 .%....:::8             8::XXX%;       X                                    X
 8888...:t888888X      8t;;::XX8       8                                    8
%888888....:::;:8     :Xttt;;;;::X@
888888888...:St     8:%%tttt;;;:X       X                                   X
88888888888S8     :%;ttt%%tttt;;X        8                                 8
%888888888%t     8S:;;;tt%%%ttt;8          :                             :
 8t8888888      S8888888Stt%%%t@            ::                          ::
 .@tt888@               8;;ttt@;               t                    t
  .8ttt8@SSSSS     SXXXX%:;;;X;                  8                8
    X8ttt8888%     %88...::X8                      X.          .X
      %8@tt88;8888%8888%8X                           :;     ;:
         :@888@XXX@888:                                 tt

Thank you for running a Zcash node!
You're helping to strengthen the network and contributing to a social good :)

In order to ensure you are adequately protecting your privacy when using Zcash,
please see <https://z.cash/support/security/>.

        Block height | 319430
            Connections | 8
Network solution rate | 508319381 Sol/s

You are currently not mining.
To enable mining, add 'gen=1' to your zcash.conf and restart.

Since starting this node 9 minutes, 1 seconds ago:
- You have validated 7815 transactions!

[Press Ctrl+C to exit] [Set 'showmetrics=0' to hide]
```

Notice `319430`, in the above output, after the `Block height` field, this means your zcashd is fully synced. Alternatively, if you were *NOT* fully synced your output would look similar to below:

```
         :88SX@888@@X8:                    8;     %X        X%     ;8
      %%Xt%tt%SSSSS:XXXt@@               X            ::  ::            X
    @S;;tt%%%t    ;;::XXXXSX            %              SS              %
  .t::::;%8888    88888tXXXX8;         S                                S
 .%...::::8            8::XXX%;        X                                X
 8888...:t888888X     8t;;::XX8        8                                8
%888888...::::;:8    :Xttt;;;;::X@
888888888...:St    8:%%tttt;;;;:X      X                                X
88888888888S8    :%;ttt%%tttt;;X        8                              8
%8888888888%t    8S:;;;tt%%%ttt;8          :                          :
 8t8888888     S8888888Stt%%%t@             ::                      ::
 .@tt888@              8;;ttt@;                t                  t
  .8ttt8@SSSSS    SXXXX%:;;;X;                   8              8
    X8ttt8888%    %88...::X8                       X.         .X
      %8@tt88;8888%8888%8X                           :;    ;:
         :@888@XXX@888:                                 tt

Thank you for running a Zcash node!
You're helping to strengthen the network and contributing to a social good :)

In order to ensure you are adequately protecting your privacy when using Zcash,
please see <https://z.cash/support/security/>.

    Downloading blocks | 319610 / ~320290 (99%)
            Connections | 6
Network solution rate | 389211802 Sol/s

You are currently not mining.
To enable mining, add 'gen=1' to your zcash.conf and restart.

Since starting this node 59 seconds ago:
- You have validated 7144 transactions!

[Press Ctrl+C to exit] [Set 'showmetrics=0' to hide]
```

Notice now how the `Block height` field has changed to `Downloading blocks` with value `319610 / ~320290 (99%)`. This indicates that your node is attempting to sync with the current block height.

You should be able to use the RPC after it finishes syncing. If you are running `zcashd` in the background, issue the below command to test:

(If you did not run zcashd in the background, you will need to open a new terminal)

```
./src/zcash-cli getinfo
```

Note: If you are familiar with bitcoind's RPC interface, you can use many of those calls to send ZEC between *t-addr* addresses. We do not support the 'Accounts' feature (which has also been deprecated in `bitcoind`) — only the empty string `""` can be used as an account name. The main network node at mainnet.z.cash is also accessible via Tor hidden service at zcmaintvsivr7pcn.onion.

Using Zcash

First, you want to obtain Zcash. You can purchase them from an exchange, from other users, or sell goods and services for them! Exactly how to obtain Zcash (safely) is not in scope for this document, but you should be careful. Avoid scams!

Important: Terminology

Zcash supports two different kinds of addresses, a `z-addr` (which begins with a `z`) is an address that uses zero-knowledge proofs and other cryptography to protect user privacy. There are also `t-addrs` (which begin with a `t`) that are similar to Bitcoin's addresses.

The interfaces are a commandline client (*zcash-cli*) and a Remote Procedure Call (RPC) interface, which is documented here:

Zcash Payment API

Attention: Wallet Backup

To ensure you have properly backed up your wallet, we **strongly** encourage you to review the *Wallet Backup Instructions* .

Generating a t-addr

Let's generate a t-addr first. If you are running zcashd for the first time, you can issue `zcash-cli getaddressesbyaccount ""` to view existing addresses.

```
$ ./src/zcash-cli getnewaddress
t1example4vfmdgQ3v3SNuQga8JKHTNi2a1
```

Listing t-addr

```
$ ./src/zcash-cli getaddressesbyaccount ""
```

This should show the address that was just created.

Receiving Zcash with a z-addr

Now let's generate a z-addr.

```
$ ./src/zcash-cli z_getnewaddress
zs1examplea4lqxrtmlpkayj0hxpfdd3ve62xhd7jds8c2a8tqz5kekplt469eza5wu8djdvpauezv
```

This creates a private address and stores its key in your local wallet file. Give this address to the sender!

A z-addr is pretty large, so it's easy to make mistakes with them. Let's put it in an environment variable to avoid mistakes:

```
$ ZADDR=
↪'zs1examplea4lqxrtmlpkayj0hxpfdd3ve62xhd7jds8c2a8tqz5kekplt469eza5wu8djdvpauezv'
```

Listing z-addr

To get a list of all addresses in your wallet for which you have a spending key, run this command:

```
$ ./src/zcash-cli z_listaddresses
```

You should see something like:

```
[
   "zs1examplea4lqxrtmlpkayj0hxpfdd3ve62xhd7jds8c2a8tqz5kekplt469eza5wu8djdvpauezv"
]
```

Sending coins with your z-addr

If someone gives you their z-addr...

```
$ FRIEND=
↪'zs1exampleakux6zswvlvsrcuku6540kw318jcft8n8hwnq6ma57canydsn3r05nxylrmcew82ja59'
```

You can send 0.8 ZEC by doing...

```
$ ./src/zcash-cli z_sendmany "$ZADDR" "[{\"amount\": 0.8, \"address\": \"$FRIEND\"}]"
```

After waiting a few seconds, you can check to see if the operation has finished and produced a result:

```
$ ./src/zcash-cli z_getoperationresult
```

```
[
 {
     "id" : "opid-bc8f822c-68df-419e-ae8f-b14b7aca29fd",
     "status" : "success",
     "creation_time" : 1554693337,
     "result" : {
         "txid" : "2979318b051a63281caa23e181ac02d367f1611374981ccd812708d13c3ed550"
     },
     "execution_secs" : 2.25543096
 }
]
```

Additional operations for zcash-cli

As Zcash is an extension of bitcoin, zcash-cli supports all commands that are part of the Bitcoin Core API (as of version 0.11.2), https://en.bitcoin.it/wiki/Original_Bitcoin_client/API_calls_list

For a full list of new commands that are not part of bitcoin API (mostly addressing operations on z-addrs) see *Zcash Payment API*

To list all Zcash commands:

```
./src/zcash-cli help
```

To get help with a particular command:

```
./src/zcash-cli help <command>
```

Attention: Known Security Issues

Each release contains a *./doc/security-warnings.md* document describing security issues known to affect that release. You can find the most recent version of this document here:

Security Warnings

Please also see our security page for recent notifications and other resources:

https://z.cash/support/security.html

4.2 Debian Binary Packages Setup

The Electric Coin Company operates a package repository for 64-bit Debian-based distributions. If you'd like to try out the binary packages, you can set it up on your system and install Zcash from there.

First install the following dependency so you can talk to our repository using HTTPS:

```
sudo apt-get install apt-transport-https wget gnupg2
```

Next add the Zcash master signing key to apt's trusted keyring:

```
wget -qO - https://apt.z.cash/zcash.asc | sudo apt-key add -
```

```
Key fingerprint = 3FE6 3B67 F85E A808 DE9B 880E 6DEF 3BAF 2727 66C0
```

Add the repository to your sources:

```
echo "deb [arch=amd64] https://apt.z.cash/ jessie main" | sudo tee /etc/apt/sources.
↪list.d/zcash.list
```

Finally, update the cache of sources and install Zcash:

```
sudo apt-get update && sudo apt-get install zcash
```

Lastly you can run `zcash-fetch-params` to fetch the zero-knowledge parameters, and set up *~/.zcash/zcash.conf* before running Zcash as your local user, as documented in the *User Guide*.

Missing Public Key Error

If you see:

```
The following signatures couldn't be verified because the public key is not
available:  NO_PUBKEY C2A798EF998940FA
```

Get the new key either directly from the z.cash site:

```
wget -qO - https://apt.z.cash/zcash.asc | sudo apt-key add -
```

or download from a public keyserver:

```
sudo apt-key adv --keyserver keyserver.ubuntu.com --recv-keys 6DEF3BAF272766C0
```

to retrieve the new key and resolve this error.

For any other signing key issues see *Updating Signing Keys*

4.2.1 Troubleshooting

Note: Only x86-64 processors are supported.

If you're starting from a new virtual machine, sudo may not come installed. See this issue for instructions to get up and running: https://github.com/zcash/zcash/issues/1844

libgomp1 or libstdc++6 version problems

These libraries are provided with gcc/g++. If you see errors related to updating them, you may need to upgrade your gcc/g++ packages to version 4.9 or later. First check which version you have using `g++ --version`; if it is before 4.9 then you will need to upgrade.

On Ubuntu Trusty, you can install gcc/g++ 4.9 as follows:

```
sudo add-apt-repository ppa:ubuntu-toolchain-r/test
sudo apt-get update
sudo apt-get install g++-4.9
```

4.2.2 Tor

The repository is also accessible via Tor, after installing the *apt-transport-tor* package, at the address zcaptnv5ljsxpnjt.onion. Use the following pattern in your sources.list file: `deb [arch=amd64] tor+http://zcaptnv5ljsxpnjt.onion/ jessie main`

4.2.3 Updating Signing Keys

If your Debian binary package isn't updating due to an error with the public key, you can resolve the problem by updating to the new key.

Revoked Key error

If you see:

```
The following signatures were invalid:  REVKEYSIG AEFD26F966E279CD
```

Remove the key marked as revoked:

```
sudo apt-key del AEFD26F966E279CD
```

Then retrieve the updated key:

```
wget -qO - https://apt.z.cash/zcash.asc | sudo apt-key add -
```

Then update the list again:

```
sudo apt-get update
```

Expired Key error

If you see:

```
The following signatures were invalid:  KEYEXPIRED 1539886450
```

Remove the old signing key:

```
sudo apt-key del 63C4A2169C1B2FA2
```

Remove the list item from local apt:

```
sudo rm /etc/apt/sources.list.d/zcash.list
```

Update the repository list:

```
sudo apt-get update
```

Then retrieve new key:

```
wget -qO - https://apt.z.cash/zcash.asc | sudo apt-key add -
```

Re-get the apt info:

```
echo "deb [arch=amd64] https://apt.z.cash/ jessie main" | sudo tee /etc/apt/sources.
↪list.d/zcash.list
```

Then update the list again:

```
sudo apt-get update
```

At this point you should be able to upgrade with the new public key.

4.2.4 Upgrading Debian 8 Jessie to Debian 9 Stretch

Debian 8 will no longer be supported for zcashd v2.0.6 and above. The instructions here are provided as convenience for anyone still running Debian 8

Before you begin upgrading from Debian 8 to Debian 9, we **strongly encourage** you to follow *Wallet Backup Instructions* .

At the very minimum, it is best to copy these backup wallet files somewhere that you will be able to access independently, e.g. an external USB.

Once you have ensured all your keys, wallet, and conf files are backup properly it is worth reading through https://www.debian.org/releases/stable/amd64/release-notes/ch-upgrading.html

Specifically, you will want to pay attention to items described in https://www.debian.org/releases/stretch/amd64/release-notes/ch-upgrading.en.html#trouble

If you are comfortable with your Debian and Zcash backups, please follow the below instructions for a minimal system upgrade:

1. It is recommended to have your Debian 8 Jessie system up to date before beginning.

   ```
   sudo apt-get update
   sudo apt-get upgrade
   ```

2. Edit your */etc/apt/sources.list* file to replace all jessie fields with stretch.

   ```
   sed -i 's/jessie/stretch/g' /etc/apt/sources.list
   ```

   You should now notice the mirrors in */etc/apt/sources.list* contain stretch fields, not jessie

   Note: If you decide to use *stable* instead of *stretch*, you may run the risk of accidentally updating when Debian 10 becomes the stable version.

3. Update and upgrade the list of available packages for Debian 9 Stretch.

```
sudo apt-get update
sudo apt-get upgrade
```

4. Ensure you have proper disk space before completing system upgrade with latest available version

```
sudo apt-get dist-upgrade
```

5. Once upgrade is complete, remove the packages that are no longer needed

```
sudo apt-get autoremove
```

6. Reboot your system and sanity check the kernel version on boot

```
reboot
```

System restarts

```
uname -a
```

OR

```
cat /etc/debian_version
```

You should see a Debian 9 field and the upgrade is complete!

4.3 Binary Tarball Download & Setup

The Electric Coin Company provides a binary tarball for download.

Download Tarball for Debian Jessie v2.1.1-1

Download Tarball for Debian Stretch v2.1.1-1

After downloading but before extracting, verify that the checksum of the tarball matches the hash below for the version of the binary you downloaded:

Debian Jessie:

```
sha256sum zcash-2.1.1-1-linux64-debian-jessie.tar.gz
```

Result: `d2694b312521cb3c22d2ef46dcbf72fb6d2102e2b5609cf37d23543228d976d1`

Debian Stretch:

```
sha256sum zcash-2.1.1-1-linux64-debian-stretch.tar.gz
```

Result: `15780d5b34cc0f9536d85d7c424b9788327e0881d0c503ef9f8dc277b2e2a4ff`

This checksum was generated from our gitian deterministic build process. View all gitian signatures.

Once you've verified that it matches, extract the files and move the binaries into your executables $PATH:

```
tar -xvf zcash-2.1.1-1-linux64.tar.gz

mv -t /usr/local/bin/ zcash-2.1.1-1/bin/*
```

Now that Zcash is installed, run this command to download the parameters used to create and verify shielded transactions:

```
zcash-fetch-params
```

Finally, set up `~/.zcash/zcash.conf` before running Zcash as your local user, as documented in the *User Guide*.

4.4 Troubleshooting Guide

The General FAQ has been reorganized and relocated to https://z.cash/support/faq.html. Please check there for the latest updates to our frequently asked questions. The following is a list of questions for Troubleshooting zcashd, the core Zcash client software.

4.4.1 System Requirements

64-bit Linux (easiest with a Debian-based distribution)

A compiler for C++11 if building from source. Gcc 6.x and above has full C++11 support, and gcc 4.8 and above supports some but not all features. Zcash will not compile with versions of gcc lower than 4.8.

At least 4GB of RAM to generate shielded transactions.

At least 8GB of RAM to successfully run all of the tests.

Zcash runs on port numbers that are 100 less than the corresponding Bitcoin port number. They are:

`8232` for mainnet RPC

`8233` for mainnet peer-to-peer network

`18232` for testnet RPC

`18233` for testnet peer-to-peer network

4.4.2 Building from source

If you did not build by running *build.sh*, you will encounter errors. Be sure to build with:

```
$ ./zcutil/build.sh -j$(nproc)
```

Note: If you don't have nproc, then substitute the number of your processors.

```
Error message: g++: internal compiler error: Killed (program cc1plus)
```

This means your system does not have enough memory for the building process and has failed. Please allocate at least 4GB of computer memory for this process and try again.

```
Error message: 'runtime_error' (or other variable) is not a member of 'std'.␣
↪compilation terminated due to -Wfatal-errors. ``
```

Check your compiler version and ensure that it support C++11. If you're using a version of gcc below 4.8.x, you will need to upgrade.

```
Error message: gtest failing with undefined reference ``
```

If you are developing on different branches of Zcash, there may be an issue with different versions of linked libraries. Try `make clean` and build again.

4.4.3 Running zcashd

Trying to start zcashd for the first time, it fails with:

```
could not load param file at /home/rebroad/.zcash-params/sprout-verifying.key
```

You didn't fetch the parameters necessary for zk-SNARK proofs. If you installed the Debian package, run *zcash-fetch-params*. If you built from source, run *./zcutil/fetch-params.sh*.

zcashd crashes with the message:

```
``std::bad_alloc`` or ``St13runtime_exception``
```

These messages indicate that your computer has run out of memory for running zcashd. This will most likely happen with mining nodes which require more resources than a full node without running a miner. This can also happen while creating a transaction involving a z-address. You'll need to allocate at least 4GB memory for these transactions.

4.4.4 RPC Interface

To get help with the RPC interface from the command line, use *zcash-cli help* to list all commands.

To get help with a particular command, use `zcash-cli help $COMMAND`.

There is also additional documentation under *Zcash Payment API* .

`zcash-cli` stops responding after I use the command `z_importkey`

The command has added the key, but your node is currently scanning the blockchain for any transactions related to that key, causing there to be a delay before it returns. This immediate rescan is the default setting for *z_importkey*, which you can override by adding *false* to the command if you simply want to import the key, i.e. *zcash-cli z_importkey $KEY false*

If, when attempting to execute the `sendrawtransaction` RPC method, you receive the error:

```
AcceptToMemoryPool: absurdly high fees
```

This is most likely caused by not specifying an output address to receive the change when creating the transaction (`createrawtransaction`). This RPC call, unlike `sendmany` and `z_sendmany`, does not do this automatically.

With `createrawtransaction`, the fee is simply the sum of the inputs minus the sum of the outputs. If this difference is larger than 0.0021 ZEC (210000 zatoshis), the assumption is that this is unintentional, and the transaction is not sent. If you really do wish to send a transaction with a large fee, add `true` to the end of the `sendrawtransaction` command line. This will allow an arbitrarily high fee.

What if my question isn't answered here?

First search the issues section (https://github.com/zcash/zcash/issues) to see if someone else has posted a similar issue and if not, feel free to report your problem there. Please provide as much information about what you've tried and what failed so others can properly assess your situation to help.

Important: If you have ran into any issues upgrading to Overwinter, please see the *Network Upgrade Developer Guide*

4.5 Zcash Payment API

4.5.1 Overview

Zcash extends the Bitcoin Core API with new RPC calls to support private Zcash payments.

Important: Zcash payments make use of **two** address formats:

`taddr` - address for transparent funds (just like a Bitcoin address, value stored in UTXOs)
`zaddr` - address for private funds (value stored in objects called notes)

When transferring funds from one taddr to another taddr, you can use either the existing Bitcoin RPC calls or the new Zcash RPC calls.

When a transfer involves zaddrs, you must use the Zcash RPC calls.

4.5.2 Compatibility with Bitcoin Core

Zcash supports all commands in the Bitcoin Core API (as of version `0.11.2`). Where applicable, Zcash will extend commands in a backwards-compatible way to enable additional functionality.

We do not recommend use of accounts which are now deprecated in Bitcoin Core. Where the account parameter exists in the API, please use `""` as its value, otherwise an error will be returned.

To support multiple users in a single node's wallet, consider using `getnewaddress` or `z_getnewaddress` to obtain a new address for each user. Also consider mapping multiple addresses to each user.

4.5.3 List of Zcash API commands

RPC Calls by Category

Accounting

z_getbalance , *z_gettotalbalance*

Addresses

z_getnewaddress , *z_listaddresses* , *z_validateaddress* , *z_exportviewingkey* , *z_importviewingkey*

Key_Management

z_exportkey , *z_importkey* , *z_exportwallet* , *z_importwallet*

Operations

z_getoperationresult , *z_getoperationstatus* , *z_listoperationids*

Payment

z_listreceivedbyaddress , *z_listunspent* , *z_sendmany* , *z_shieldcoinbase* , *z_mergetoaddress*, *z_setmigration*, *z_getmigrationstatus*

RPC Parameter Conventions

Parameter	Definition
taddr	*Transparent address*
zaddr	*Private address*
address	*Accepts both private and transparent addresses.*
amount	*JSON format decimal number with at most 8 digits of precision, where 1 ZEC is expressed as 1.00000000.*
memo	*Metadata expressed in hexadecimal format. Limited to 512 bytes, the current size of the memo field of a private transaction. Zero padding is automatic.*

Accounting

Command: `z_getbalance`

Parameters

1. **`"address"`** ***(string)*** The selected address. It may be a transparent or private address.
2. **`minconf`** ***(numeric, optional, default=1)*** Only include transactions confirmed at least this many times.

Output

amount ***(numeric)*** The total amount in ZEC received for this address.

Description

Returns the balance of a taddr or zaddr belonging to the node's wallet. Optionally set the minimum number of confirmations a transaction must have in order to be included in the balance. Use 0 to unconfirmed transactions.

Examples

The total amount received by address "myaddress"

```
zcash-cli z_getbalance "myaddress"

0.00000000
```

Command: `z_gettotalbalance`

Parameters

1. **`minconf`** ***(numeric, optional, default=1)*** Only include transactions confirmed at least this many times.

Output

"transparent" ***(numeric)*** The total balance of transparent funds

"private" ***(numeric)*** The total balance of private funds

"total" ***(numeric)*** The total balance of both transparent and private funds

Description

Return the total value of funds stored in the node's wallet. Optionally set the minimum number of confirmations a private or transparent transaction must have in order to be included in the balance. Use 0 to count unconfirmed transactions.

Examples

The total amount in the wallet

```
zcash-cli z_gettotalbalance

{
    "transparent": "0.00",
    "private": "0.00",
    "total": "0.00"
}
```

Addresses

Command: `z_getnewaddress`

Parameters

1. **`type`** ***(string, optional, default="sprout")*** The type of address (e.g. "sprout", "sapling").

Output

"zcashaddress" ***(string)*** The new shielded address

Description

Return a new zaddr for sending and receiving payments. The spending key for this zaddr will be added to the node's wallet.

Examples

Create a new shielded address (as of v2.0.2 Sapling is default; v2.0.0 and v2.0.1 Sprout is default)

```
zcash-cli z_getnewaddress

zcU1Cd6zYyZCd2VJF8yKgmzjxdiiU1rgTTjEwoN1CGUWCziPkUTXUjXmX7TMqdMNsTfuiGN1jQoVN4kGxUR4sAPN4XZ7pxb
```

Create a new Sapling shielded address

```
zcash-cli z_getnewaddress sapling

zs1z7rejlpsa98s2rrrfkwmaxu53e4ue0ulcrw0h4x5g8jl04tak0d3mm47vdtahatqrlkngh9sly
```

Command: `z_listaddresses`

Parameters

1. **`includeWatchonly`** ***(bool, optional, default=false)*** Also include watchonly addresses (see 'z_importviewingkey')

Output

"zaddr" *(string)* A zaddr belonging to the wallet

Description

Returns a list of all the zaddrs in this node's wallet for which you have a spending key.

Examples

List all the zaddrs in this node's wallet

```
zcash-cli z_listaddresses

[
    "zcU1Cd6zYyZCd2VJ...",
    "zcddV3rosTRpWqNj..."
]
```

Command: `z_validateaddress`

Parameters

1. **`zaddr`** ***(string, required)*** The z address to validate

Output

"isvalid" [true|false, *(boolean)*] If the address is valid or not. If not, this is the only property returned.

"address" ["zaddr", *(string)*] The z address validated

"type" ["xxxx", *(string)*] "sprout" or "sapling"

"ismine" [true|false, *(boolean)*] If the address is yours or not

"payingkey" ["hex", *(string)*] [sprout] The hex value of the paying key, a_pk

"transmissionkey" ["hex", *(string)*] [sprout] The hex value of the transmission key, pk_enc

"diversifier" ["hex", *(string)*] [sapling] The hex value of the diversifier, d

"diversifiedtransmissionkey" ["hex", *(string)*] [sapling] The hex value of pk_d

Description

Return information about the given z address.

Examples

List all the information about a given zaddr.

```
zcash-cli z_validateaddress
↪"zcWsmqT4X2V4jgxbgiCzyrAfRT1vi1F4sn7M5Pkh66izzw8Uk7LBGAH3DtcSMJeUb2pi3W4SQF8LMKkU2cUuVP68yAGcomL
↪"

{
  "isvalid": true,
  "address": "zcbcb6XnP8hbV5y6ZwsY...",
  "payingkey": "b4ae837...",
  "ismine": true
}
```

Key Management

Command: `z_exportkey`

Parameters

1. **`zaddr`** ***(string, required)*** The zaddr for the private key

Output

"key" ***(string)*** The private key

Description

Requires an unlocked wallet or an unencrypted wallet. Return a zkey for a given zaddr belonging to the node's wallet. The key will be returned as a string formatted using Base58Check as described in the Zcash protocol spec.

Examples

Export a key for a given zaddr.

```
./zcash-cli z_exportkey
↪"zcWsmqT4X2V4jgxbgiCzyrAfRT1vi1F4sn7M5Pkh66izzw8Uk7LBGAH3DtcSMJeUb2pi3W4SQF8LMKkU2cUuVP68yAGcomL
↪"

AKWUAkypwQjhZ6LLNa
```

Command: `z_importkey`

Parameters

1. **"zkey"** ***(string, required)*** The zkey (see z_exportkey)
2. **rescan** ***(string, optional, default="whenkeyisnew")*** Rescan the wallet for transactions - can be "yes", "no" or "whenkeyisnew"
3. **startHeight** ***(numeric, optional, default=0)*** Block height to start rescan from

Output

NONE

Description

Wallet must be unlocked. Add a zkey as returned by z_exportkey to a node's wallet. The key should be formatted using Base58Check as described in the Zcash protocol spec. Rescan can be "yes", "no" or the default "whenkeyisnew" to rescan for transactions affecting any address or pubkey script in the wallet (including transactions affecting the newly-added address for this spending key). The startHeight parameter sets the block height to start the rescan from (default is 0).

Examples

Import the zkey with rescan

```
zcash-cli z_importkey "mykey"
```

Import the zkey with partial rescan

```
zcash-cli z_importkey "mykey" whenkeyisnew 30000
```

Re-import the zkey with longer partial rescan

```
zcash-cli z_importkey "mykey" whenkeyisnew 30000
```

Command: `z_exportwallet`

Parameters

1. **"filename"** ***(string, required)*** The filename, saved in folder set by zcashd -exportdir option

Output

"path" (string) The full path of the destination file

Description

Requires an unlocked wallet or an unencrypted wallet. Creates or overwrites a file with taddr private keys and zaddr private keys in a human-readable format. Filename is the file in which the wallet dump will be placed. May be prefaced by an absolute file path. An existing file with that name will be overwritten. No value is returned but a JSON-RPC error will be reported if a failure occurred.

As of Sapling activation, the shielded private keys in this file will be separated into legacy shielded private keys under the title Zkeys and Sapling shielded private keys. The export also includes (as of Sapling activation) a comment with an HD wallet seed and associated fingerprint, both as hex strings. This seed is only for the wallet's Sapling shielded keys and addresses.

Examples

Export a wallet

```
zcash-cli z_exportwallet "wallet_filename"

<No output will appear if successful>
```

Command: `z_importwallet`

Parameters

1. **"filename"** ***(string, required)*** The wallet file

Output

NONE

Description

Requires an unlocked wallet or an unencrypted wallet. Imports private keys from a file in wallet export file format (see z_exportwallet). These keys will be added to the keys currently in the wallet. This call may need to rescan all or parts of the block chain for transactions affecting the newly-added keys, which may take several minutes. Filename is the file to import. The path is relative to zcashd's working directory. No value is returned but a JSON-RPC error will be reported if a failure occurred. This command does not yet support importing HD seeds and will import Sapling addresses in a standard form (non-HD). To backup and restore the full wallet inclusive of the Sapling HD seed, use the backupwallet command.

Examples

Import a wallet

```
zcash-cli z_importwallet "path/to/exportdir/nameofbackup"

<No output will appear if successful>
```

Command: `z_exportviewingkey`

Parameters

1. **"zaddr"** ***(string, required)*** The zaddr for the viewing key

Output

"vkey" (string) The viewing key

Description

Reveals the viewing key corresponding to 'zaddr'. Then the z_importviewingkey can be used with this output.

Examples

Export a viewing key for a given address

```
zcash-cli z_exportviewingkey "myaddress"

ZiVtJjUXq5...
```

Command: `z_importviewingkey`

Parameters

1. **"vkey"** ***(string, required)*** The viewing key (see z_exportviewingkey)
2. **rescan** ***(string, optional, default="whenkeyisnew")*** Rescan the wallet for transactions - can be "yes", "no" or "whenkeyisnew"
3. **startHeight** ***(numeric, optional, default=0)*** Block height to start rescan from

Output

NONE

Description

Adds a viewing key (as returned by z_exportviewingkey) to your wallet.

Examples

Import a viewing key

```
zcash-cli z_importviewingkey "vkey"
```

Import the viewing key without rescan

```
zcash-cli z_importviewingkey "vkey", no
```

Import the viewing key with partial rescan

```
zcash-cli z_importviewingkey "vkey" whenkeyisnew 30000
```

Re-import the viewing key with longer partial rescan

```
zcash-cli z_importviewingkey "vkey" yes 20000
```

Payment

Command: `z_listreceivedbyaddress`

Parameters

1. **"address"** ***(string)*** The private address.
2. **minconf** ***(numeric, optional, default=1)*** Only include transactions confirmed at least this many times.

Output

"txid": xxxxx, ***(string)*** The transaction id

"amount": xxxxx, ***(numeric)*** The amount of value in the note

"memo": xxxxx, ***(string)*** Hexademical string representation of memo field

"change": true|false, ***(boolean)*** True if the address that received the note is also one of the sending addresses

Description

Return a list of amounts received by a zaddr belonging to the node's wallet. Optionally set the minimum number of confirmations which a received amount must have in order to be included in the result. Use 0 to count unconfirmed transactions.

Examples

Return a list of amounts recieved by a zaddr belonging to the node's wallet.

```
zcash-cli z_listreceivedbyaddress
↪"ztfaW34Gj9FrnGUEf833ywDVL62NWXBM81u6EQnM6VR45eYnXhwztecW1SjxA7JrmAXKJhxhj3vDNEpVCQoSvVoSpmbhtjf
↪"
```

Command: `z_listunspent`

Parameters

1. **minconf** ***(numeric, optional, default=1)*** The minimum confirmations to filter*
2. **maxconf** ***(numeric, optional, default=9999999)*** The maximum confirmations to filter
3. **"includeWatchonly"** ***(bool, optional, default=false)*** Also include watchonly addresses (see 'z_importviewingkey')
4. **"addresses"** ***(string)*** A json array of zaddrs to filter on. Duplicate addresses not allowed.

```
[
  "address"     (string) zaddr
  ,...
]
```

Output

"txid" ["txid", *(string)*] The transaction id

"jsindex" [n *(numeric)*] The joinsplit index

"jsoutindex" [n *(numeric)*] [sprout] The output index of the joinsplit

"outindex" [n *(numeric)*] [sapling] The output index

"confirmations" [n *(numeric)*] The number of confirmations

"spendable" [true|false *(boolean)*] True if note can be spent by wallet, false if note has zero confirmations, false if address is watchonly

"address" ["address", *(string)*] The shielded address

"amount": xxxxx, ***(numeric)*** The amount of value in the note

"memo": xxxxx, ***(string)*** Hexademical string representation of memo field

"change": true|false, ***(boolean)*** True if the address that received the note is also one of the sending addresses

Description

Returns array of unspent shielded notes with between minconf and maxconf (inclusive) confirmations. Optionally filter to only include notes sent to specified addresses. When minconf is 0, unspent notes with zero confirmations are returned even though they are not immediately spendable

Examples

Return an array of unspent shielded notes

```
zcash-cli z_listunspent
```

Returns array of unspent shielded notes with between minconf and maxconf (inclusive) confirmations. Optionally filter to only include notes sent to specified addresses.

```
zcash-cli z_listunspent 6 9999999 false "[\
↪"ztbx5DLDxa5ZLFTchHhoPNkKs57QzSyib6UqXpEdy76T1aUdFxJt1w9318Z8DJ73XzbnWHKEZP9Yjg712N5kMmP4QzS9iC9\
↪",\
↪"ztfaW34Gj9FrnGUEf833ywDVL62NWXBM81u6EQnM6VR45eYnXhwztecW1SjxA7JrmAXKJhxhj3vDNEpVCQoSvVoSpmbhtjf\
↪"]"
```

Command: z_sendmany

Parameters

1. **"fromaddress"** ***(string, required)*** The taddr or zaddr to send the funds from.
2. "amounts" (array, required) An array of json objects representing the amounts to send.

   "address":address (string, required) The address is a taddr or zaddr "amount":amount (numeric, required) The numeric amount in ZEC is the value "memo":memo (string, optional) If the address is a zaddr, raw data represented in hexadecimal string format
3. **minconf** ***(numeric, optional, default=1)*** Only use funds confirmed at least this many times.
4. **fee** ***(numeric, optional, default=0.0001)*** The fee amount to attach to this transaction.

Output

"operationid" ***(string)*** An operationid to pass to z_getoperationstatus to get the result of the operation.

Description

This is an Asynchronous RPC call. Send funds from an address to multiple outputs. The address can be a taddr or a zaddr. Amounts is a list containing key/value pairs corresponding to the addresses and amount to pay. Each output address can be in taddr or zaddr format. When sending to a zaddr, you also have the option of of attaching a memo in hexadecimal format.

When sending coinbase funds to a zaddr, the node's wallet does not allow any change. Put another way, spending a partial amount of a coinbase utxo is not allowed. This is not a consensus rule but a local wallet rule due to the current implementation of z_sendmany. In future, this may be removed.

Optionally set the minimum number of confirmations which a private or transparent transaction must have in order to be used as an input. When sending from a zaddr, minconf must must be greater than zero. Optionally set a transaction fee, which by default is 0.0001 ZEC. Any transparent change will be sent to a new transparent address. Any private change will be sent back to the zaddr being used as the source of funds Returns an operationid. You use the operationid value with z_getoperationstatus and z_getoperationresult to obtain the result of sending funds, which if successful, will be a txid.

Examples

Send funds from a t-address to z-address output

```
zcash-cli z_sendmany "t1M72Sfpbz1BPpXFHz9m3CdqATR44Jvaydd" '[{"address":
↪"ztfaW34Gj9FrnGUEf833ywDVL62NWXBM81u6EQnM6VR45eYnXhwztecW1SjxA7JrmAXKJhxhj3vDNEpVCQoSvVoSpmbht
↪" ,"amount": 5.0}]'
```

Command: `z_shieldcoinbase`

Parameters

1. **"fromaddress"** ***(string, required)*** The address is a taddr or "*" for all taddrs belonging to the wallet.
2. **"toaddress"** ***(string, required)*** The address is a zaddr.
3. **fee** ***(numeric, optional, default=0.0001)*** The fee amount to attach to this transaction.
4. **limit** ***(numeric, optional, default=50)*** Limit on the maximum number of utxos to shield. Set to 0 to use node option -mempooltxinputlimit (before Overwinter), or as many as will fit in the transaction (after Overwinter).

Output

"remainingUTXOs": xxx ***(numeric)*** Number of coinbase utxos still available for shielding.

"remainingValue": xxx ***(numeric)*** Value of coinbase utxos still available for shielding.

"shieldingUTXOs": xxx ***(numeric)*** Number of coinbase utxos being shielded.

"shieldingValue": xxx ***(numeric)*** Value of coinbase utxos being shielded.

"opid": xxx ***(string)*** An operationid to pass to z_getoperationstatus to get the result of the operation.

Description

This is an Asynchronous RPC call. Shield transparent coinbase funds by sending to a shielded z address. Utxos selected for shielding will be locked. If there is an error, they are unlocked. The RPC call listlockunspent can be used to return a list of locked utxos. The number of coinbase utxos selected for shielding can be set with the limit parameter, which has a default value of 50. If the parameter is set to 0, the number of utxos selected is limited by the -mempooltxinputlimit option. Any limit is constrained by a consensus rule defining a maximum transaction size of 10000 bytes. The from address is a taddr or "*" for all taddrs belonging to the wallet. The to address is a zaddr. The default fee is 0.0001. Returns an object containing an operationid which can be used with z_getoperationstatus and z_getoperationresult, along with key-value pairs regarding how many utxos are being shielded in this transaction and what remains to be shielded.

Examples

Shield transparent coinbase funds by sending to a shielded z-address.

```
zcash-cli z_shieldcoinbase "t1M72Sfpbz1BPpXFHz9m3CdqATR44Jvaydd"
↪"ztfaW34Gj9FrnGUEf833ywDVL62NWXBM81u6EQnM6VR45eYnXhwztecW1SjxA7JrmAXKJhxhj3vDNEpVCQoSvVoSpmbht
↪"
```

See also shield_coinbase

Command: `z_mergetoaddress`

Parameters

1. **fromaddresses** ***(array, required)***

   A JSON array with addresses.

   The following special strings are accepted inside the array:

   "ANY_TADDR": Merge UTXOs from any taddrs belonging to the wallet.

   "ANY_SPROUT": Merge notes from any Sprout zaddrs belonging to the wallet.

   "ANY_SAPLING": Merge notes from any Sapling zaddrs belonging to the wallet.

   ["address", ...]: A list of taddrs or a zaddrs

   If a special string is given, any given addresses of that type will be counted as duplicates and cause an error.

2. **"toaddress"** ***(string, required)*** The taddr or zaddr to send the funds to.
3. **fee** ***(numeric, optional, default=0.0001)*** The fee amount to attach to this transaction.
4. **transparent_limit** ***(numeric, optional, default=50)*** Limit on the maximum number of UTXOs to merge. Set to 0 to use node option -mempooltxinputlimit (before Overwinter), or as many as will fit in the transaction (after Overwinter).
5. **shielded_limit** ***(numeric, optional, default=20 Sprout or 200 Sapling Notes)*** Limit on the maximum number of notes to merge. Set to 0 to merge as many as will fit in the transaction.
6. **"memo"** ***(string, optional)*** Encoded as hex. When toaddress is a zaddr, this will be stored in the memo field of the new note.

Output

"remainingUTXOs": xxx ***(numeric)*** Number of UTXOs still available for merging.

"remainingTransparentValue": xxx ***(numeric)*** Value of UTXOs still available for merging.

"remainingNotes": xxx ***(numeric)*** Number of notes still available for merging.

"remainingShieldedValue": xxx ***(numeric)*** Value of notes still available for merging.

"mergingUTXOs": xxx ***(numeric)*** Number of UTXOs being merged.

"mergingTransparentValue": xxx ***(numeric)*** Value of UTXOs being merged.

"mergingNotes": xxx ***(numeric)*** Number of notes being merged.

"mergingShieldedValue": xxx ***(numeric)*** Value of notes being merged.

"opid": xxx ***(string)*** An operationid to pass to z_getoperationstatus to get the result of the operation.

Description

z_mergetoaddress is no longer an experimental feature as of zcash v2.1.1

Merge multiple UTXOs and notes into a single UTXO or note. Coinbase UTXOs are ignored; use *z_shieldcoinbase* to combine those into a single note.

This is an asynchronous operation, and UTXOs selected for merging will be locked. If there is an error, they are unlocked. The RPC call *listlockunspent* can be used to return a list of locked UTXOs.

The number of UTXOs and notes selected for merging can be limited by the caller. If the transparent limit parameter is set to zero, and Overwinter is not yet active, the -mempooltxinputlimit option will determine the number of UTXOs.

After Overwinter has activated -mempooltxinputlimit is ignored and having a transparent input limit of zero will mean limit the number of UTXOs based on the size of the transaction. Any limit is constrained by the consensus rule defining a maximum transaction size of 100000 bytes before Sapling, and 2000000 bytes once Sapling activates.

Examples

Send funds from one or more addresses to a single one.

```
zcash-cli z_mergetoaddress '["ANY_SAPLING",
↪"t1M72Sfpbz1BPpXFHz9m3CdqATR44Jvaydd"]'␣
↪ztestsapling19rnyu293v44f0kvtmszhx35lpdug574twc0lwyf4s7w0umtkrdq5nfcauxrxcyfmh3m7slemqsj
```

Command: `z_setmigration`

Parameters

1. enabled *(boolean, required)* 'true' or 'false' to enable or disable respectively.

Output

NONE

Description

When enabled the Sprout-to-Sapling migration will attempt to migrate all funds from this wallet's Sprout addresses to either the address for Sapling account 0 or the address specified by the parameter `-migrationdestaddress`.

This migration is designed to minimize information leakage. As a result for wallets with a significant Sprout balance, this process may take several weeks. The migration works by sending, up to 5, as many transactions as possible whenever the blockchain reaches a height equal to 499 modulo 500. The transaction amounts are picked according to the random distribution specified in ZIP 308. The migration will end once the wallet's Sprout balance is below .001 ZEC.

Examples

Enable migration.

```
zcash-cli z_setmigration true
```

Command: `z_getmigrationstatus`

Parameters

NONE

Output

"enabled": true|false ***(boolean)*** Whether or not migration is enabled.

"destination_address": "zaddr" ***(string)*** The Sapling address that will receive Sprout funds.

"unmigrated_amount": nnn.n ***(numeric)*** The total amount of unmigrated ZEC.

"unfinalized_migrated_amount": nnn.n ***(numeric)*** The total amount of unfinalized ZEC (less than 10 confirmations).

"finalized_migrated_amount": nnn.n ***(numeric)*** The total amount of finalized ZEC (10 or more confirmations)

"finalized_migration_transactions": nnn ***(numeric)*** The number of migration transactions involving this wallet.

"time_started": ttt *(numeric, optional)* The block time of the first migration transaction as a Unix timestamp.

"migration_txids": [txids] *(json array of strings)* An array of all migration txids involving this wallet.

Description

Returns information about the status of the Sprout to Sapling migration. In the result a transaction is defined as finalized if and only if it has at least ten confirmations (aka is finalized).

Note: It is possible that manually created transactions involving this wallet will be included in the result.

Examples

Check migration status.

```
zcash-cli z_getmigrationstatus
```

Operations

Asynchronous calls return an OperationStatus object which is a JSON object with the following defined key-value pairs:

Item operationid

Description Unique identifier for the async operation. Use this value with `z_getoperationstatus` or `z_getoperationresult` to poll and query the operation and obtain its result.

Item status

Description

Current status of operation:

queued : operation is pending execution **executing :** operation is currently being executed **cancelled:** operation is cancelled **failed :** operation has failed **success :** operation has succeeded

Item result

Description Result object if the status is *'success'*. The exact form of the result object is dependent on the call itself.

Item error

Description Error object if the status is *'failed'*. The error object has the following key-value pairs:

code : number **message:** error message

Important: Depending on the type of asynchronous call, there may be other key-value pairs. For example, a `z_sendmany` operation will also include the following in an OperationStatus object:

method : name of operation (e.g. `z_sendmany`)

params : an object containing the parameters to `z_sendmany`

Currently, as soon as you retrieve the operation status for an operation which has finished, that is it has either succeeded, failed, or been cancelled, the operation and any associated information is removed.

It is currently not possible to cancel operations.

Command `z_getoperationresult`

Parameters

1. **"operationid"** ***(array, optional)*** A list of operation ids we are interested in. If not provided, examine all operations known to the node.

Output

" [object, . . .]" (array) A list of JSON objects

Description

Return OperationStatus JSON objects for all completed operations the node is currently aware of, and then remove the operation from memory. Operationids is an optional array to filter which operations you want to receive status objects for. Output is a list of operation status objects, where the status is either "failed", "cancelled" or "success".

Example

Return OperationStatus JSON objects for all completed operations the node is currently aware of

```
zcash-cli z_getoperationresult '["operationid", ... ]'
```

Command: `z_getoperationstatus`

Parameters

1. **"operationid"** ***(array, optional)*** A list of operation ids we are interested in. If not provided, examine all operations known to the node.

Output

" [object, . . .]" ***(array)*** A list of JSON objects

Description

Return OperationStatus JSON objects for all operations the node is currently aware of. Operationids is an optional array to filter which operations you want to receive status objects for. Output is a list of operation status objects.

Example

Return OperationStatus JSON objects for all completed operations the node is currently aware of

```
zcash-cli z_getoperationstatus '["operationid", ... ]'
```

Command: `z_listoperationids`

Parameters

1. **"status"** ***(string, optional)*** Filter result by the operation's state e.g. "success".

Output

"operationid" ***(string)*** An operation id belonging to the wallet

Description

Return a list of operationids for all operations which the node is currently aware of. State is an optional string parameter to filter the operations you want listed by their state. Acceptable parameter values values are 'queued', 'executing', 'success', 'failed',

Examples

Return a list of operationids for all operations which the node is currently aware of

```
zcash-cli z_listoperationids
```

4.5.4 Asynchronous RPC Call Error Codes

Zcash error codes are defined in rpcprotocol.h

Table 1: z_sendmany

Value	Meaning
-8	*RPC_INVALID_PARAMETER*
-5	*RPC_INVALID_ADDRESS_OR_KEY*
-4	*RPC_WALLET_ERROR*
-6	*RPC_WALLET_INSUFFICIENT_FUNDS*
-16	*RPC_WALLET_ENCRYPTION_FAILED*
-12	*RPC_WALLET_KEYPOOL_RAN_OUT*

RPC_INVALID_PARAMETER

RPC_INVALID_PARAMETER	**Invalid, missing or duplicate parameter**
Minconf cannot be zero when sending from zaddr	*Cannot accept minimum confirmation value of zero when sending from zaddr*
Minconf cannot be negative	Cannot accept negative minimum confirmation number.
Minimum number of confirmations cannot be less than 0	Cannot accept negative minimum confirmation number.
From address parameter missing	Missing an address to send funds from.
No recipients	Missing recipient addresses.
Memo must be in hexadecimal format	Encrypted memo field data must be in hexadecimal format.
Memo size of __ is too big, maximum allowed is __	Encrypted memo field data exceeds maximum size of 512 bytes.
From address does not belong to this node, zaddr spending key not found.	Sender address spending key not found.
Invalid parameter, expected object	Expected object.
Invalid parameter, unknown key: __	Unknown key.
Invalid parameter, expected valid size	Invalid size.
Invalid parameter, expected hex txid	Invalid txid.
Invalid parameter, vout must be positive	Invalid vout.
Invalid parameter, duplicated address	Address is duplicated.
Invalid parameter, amounts array is empty	Amounts array is empty.
Invalid parameter, unknown key	Key not found.
Invalid parameter, unknown address format	Unknown address format.
Invalid parameter, size of memo	Invalid memo field size.
Invalid parameter, amount must be positive	Invalid or negative amount.
Invalid parameter, too many zaddr outputs	z_address outputs exceed maximum allowed.
Invalid parameter, expected memo data in hexadecimal format	Encrypted memo field is not in hexadecimal format.
Invalid parameter, size of memo is larger than maximum allowed __	Encrypted memo field data exceeds maximum size of 512 bytes.

RPC_INVALID_ADDRESS_OR_KEY

RPC_INVALID_ADDRESS_OR_KEY	**Invalid address or key**
Invalid from address, no spending key found for zaddr	z_address spending key not found.
Invalid output address, not a valid taddr.	Transparent output address is invalid.
Invalid from address, should be a taddr or zaddr.	Sender address is invalid.
From address does not belong to this node, zaddr spending key not found.	Sender address spending key not found.

RPC_WALLET_INSUFFICIENT_FUNDS

RPC_WALLET_INSUFFICIENT_FUNDS	**Not enough funds in wallet or account**
Insufficient funds, no UTXOs found for taddr from address.	Insufficient funds for sending address.
Could not find any non-coinbase UTXOs to spend. Coinbase UTXOs can only be sent to a single zaddr recipient.	Must send Coinbase UTXO to a single z_address.
Could not find any non-coinbase UTXOs to spend.	No available non-coinbase UTXOs.
Insufficient funds, no unspent notes found for zaddr from address.	Insufficient funds for sending address.
Insufficient transparent funds, have __, need __ plus fee __	Insufficient funds from transparent address.
Insufficient protected funds, have __, need __ plus fee __	Insufficient funds from shielded address.

RPC_WALLET_ERROR

RPC_WALLET_ERROR	**Unspecified problem with wallet**
Could not find previous JoinSplit anchor	Try restarting node with *-reindex*.
Error decrypting output note of previous JoinSplit: __	
Could not find witness for note commitment	Try restarting node with *-rescan*.
Witness for note commitment is null	Missing witness for note commitment.
Witness for spendable note does not have same anchor as change input	Invalid anchor for spendable note witness.
Not enough funds to pay miners fee	Retry with sufficient funds.
Missing hex data for raw transaction	Raw transaction data is null.
Missing hex data for signed transaction	Hex value for signed transaction is null.
Send raw transaction did not return an error or a txid.	

RPC_WALLET_ENCRYPTION_FAILED

RPC_WALLET_ENCRYPTION_FAILED	**Failed to encrypt the wallet**
Failed to sign transaction	Transaction was not signed, sign transaction and retry.

RPC_WALLET_KEYPOOL_RAN_OUT

RPC_WALLET_KEYPOOL_RAN_OUT	**Keypool ran out, call keypoolrefill first**
Could not generate a taddr to use as a change address	Call keypoolrefill and retry.

Important: To view a community maintained list of the API, please click here

4.6 Wallet Backup Instructions

4.6.1 Overview

Backing up your Zcash private keys is the best way to be proactive about preventing loss of access to your ZEC.

Problems resulting from bugs in the code, user error, device failure, etc. may lead to losing access to your wallet (and as a result, the private keys of addresses which are required to spend from them).

No matter what the cause of a corrupted or lost wallet could be, we highly recommend all users backup on a regular basis. Anytime a new transparent address or legacy shielded address (`zc...`) in the wallet is generated, we recommending making a new backup so all private keys for those types of addresses in your wallet are safe.

Sapling shielded addresses (`zs...`) are natively generated from an *HDseed*. Export and import of this seed is not fully supported in the RPC yet. Sapling keys can be exported and imported using the options below but will be treated as standard (non-HD) keys.

Note: A backup is a duplicate of data needed to spend ZEC so where you keep your backup(s) is another important consideration. You should not store backups where they would be equally or increasingly susceptible to loss or theft.

4.6.2 Backing up your wallet and/or private keys

These instructions are specific for the officially supported Zcash Linux client. For backing up with third-party wallets, please consult with user guides or support channels provided for those services.

There are multiple ways to make sure you have at least one other copy of the private keys needed to spend your ZEC and view your shielded ZEC.

For all methods, you will need to include an export directory setting in your config file (*zcash.conf* located in the data directory which is `~/.zcash/` unless it's been overridden with `datadir=` setting):

```
exportdir=path/to/chosen/export/directory
```

You may choose any directory within the home directory as the location for export & backup files. If the directory doesn't exist, it will be created.

Note: zcashd will need to be stopped and restarted for edits in the config file to take effect.

Using `backupwallet`

To create a backup of your wallet, use:

```
zcash-cli backupwallet <nameofbackup>
```

The backup will be an exact copy of the current state of your wallet.dat file stored in the export directory you specified in the config file. The file path will also be returned.

If your original *wallet.dat* file becomes inaccessible for whatever reason, you can use your backup by copying it into your data directory and renaming the copy to *wallet.dat*.

If you generate new addresses in your wallet after using `backupwallet`, they will not be reflected in the backup file. Due to the deterministic property of HD wallets, if you generate new Sapling shielded addresses in your wallet after using `backupwallet`, then restoring that wallet file and calling `z_getnewaddress sapling` will regenerate keys in the same order that they were created originally. After recreating keys with this method, it is recommended to

restart the client using `-rescan` to force the client to check for prior transactions and properly update the balances of those addresses.

Using `z_exportwallet` & `z_importwallet`

If you prefer to have an export of your private keys in human readable format, you can use:

```
zcash-cli z_exportwallet <nameofbackup>
```

This will generate a file in the export directory listing all transparent and shielded private keys with their associated public addresses. The file path will be returned in the command line. As of Sapling activation, the shielded private keys in this file will be separated into legacy shielded private keys under the title *Zkeys* and Sapling shielded private keys.

As of Sapling activation, the export also includes a comment with an HD wallet seed and associated fingerprint, both as hex strings. This seed is *only* for the wallet's Sapling shielded keys and addresses. For example:

```
HDSeed=5fa7753029b99c408e...
fingerprint=5fa7753029b99c408e...
```

To import keys into a wallet which were previously exported to a file, use:

```
zcash-cli z_importwallet <path/to/exportdir/nameofbackup>
```

Note: `z_importwallet` does not yet support importing HD seeds but will import Sapling addresses in a standard form (non-HD). To backup and restore the full wallet inclusive of the Sapling HD seed, use the instructions for `backupwallet` above..

Using `z_exportkey`, `z_importkey`, `dumpprivkey` & `importprivkey`

If you prefer to export a single private key for a shielded address, you can use:

```
zcash-cli z_exportkey <z-address>
```

This will return the private key and will not create a new file.

For exporting a single private key for a transparent address, you can use the command inherited from Bitcoin:

```
zcash-cli dumpprivkey <t-address>
```

This will return the private key and will not create a new file.

To import a private key for a shielded address, use:

```
zcash-cli z_importkey <z-priv-key>
```

This will add the key to your wallet and rescan the wallet for associated transactions if it is not already part of the wallet.

The rescanning process can take a few minutes for a new private key. To skip it, instead use:

```
zcash-cli z_importkey <z-private-key> no
```

For other instructions on fine-tuning the wallet rescan, see the command's help documentation:

```
zcash-cli help z_importkey
```

To import a private key for a transparent address, use:

```
zcash-cli importprivkey <t-priv-key>
```

This has the same functionality as `z_importkey` but works with transparent addresses.

See the command's help documentation for instructions on fine-tuning the wallet rescan:

```
zcash-cli help importprivkey
```

Using `dumpwallet`

This command inherited from Bitcoin is deprecated. It will export private keys in a similar fashion as `z_exportwallet` but only for transparent addresses. This file will also include a comment with an HD wallet seed and associated fingerprint as described above.

4.7 Addresses and Value Pools in Zcash

4.7.1 Addresses Overview

Zcash has two main types of addresses: shielded (z-addresses which start with "z") and transparent (t-addresses which start with "t").

The Sapling network upgrade added a new type of shielded address to support the usability and security improvements. The new Sapling shielded addresses start with "zs" whereas the legacy, Sprout shielded addresses start with "zc".

Example addresses:

Sapling:

```
zs1z7rejlpsa98s2rrrfkwmaxu53e4ue0ulcrw0h4x5g8jl04tak0d3mm47vdtahatqrlkngh9sly
```

Legacy:

```
zcU1Cd6zYyZCd2VJF8yKgmzjxdiiU1rgTTjEwoN1CGUWCziPkUTXUjXmX7TMqdMNsTfuiGN1jQoVN4kGxUR4sAPN4XZ7pxb
```

Transparent:

```
t14oHp2v54vfmdgQ3v3SNuQga8JKHTNi2a1
```

ZEC can be sent between transparent and shielded addresses. Therefore, there are four basic types of transactions:

4.7.2 Shielded Addresses

Shielded addresses are the address type that use zero-knowledge proofs to allow transaction data to be encrypted but remain verifiable by network nodes. Sapling shielded addresses are the primary shielded addresses as of the Sapling network upgrade activation. The legacy, Sprout addresses are still supported but will likely be deprecated in future. To learn about migrating from Sprout to Sapling addresses, see *Sprout-to-Sapling Migration* documentation.

A transaction between two shielded addresses (a shielded transaction) keeps the addresses, transaction amount and the memo field shielded from the public (with the exception of migrating funds between Sprout and Sapling shielded addresses).

Senders to a shielded address may or may not include an encrypted memo.

Recipients of a shielded or deshielding transaction do not learn about the senders address through the transaction recipt in their wallet. The receivers only learn the value sent to their address(es) and if receiving to shielded addresses, any encrypted memo that may have been included by the sender.

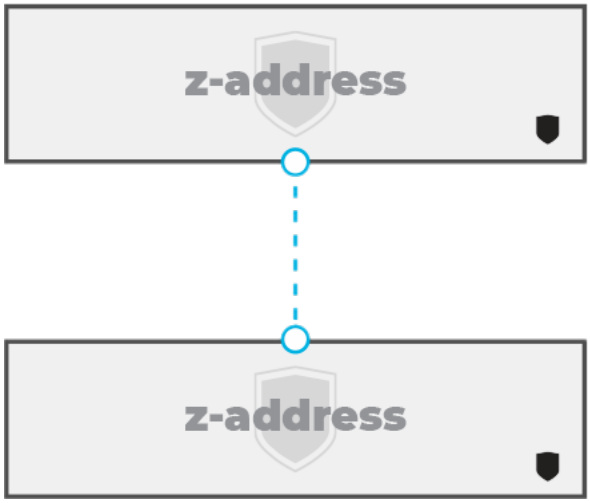


Fig. 1: **Shielded/private** (Value is not revealed on the blockchain)

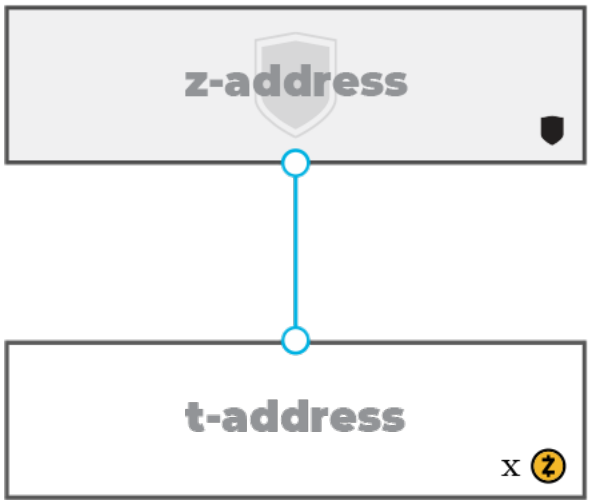


Fig. 2: **Deshielding** (Value is revealed on the receiver end)

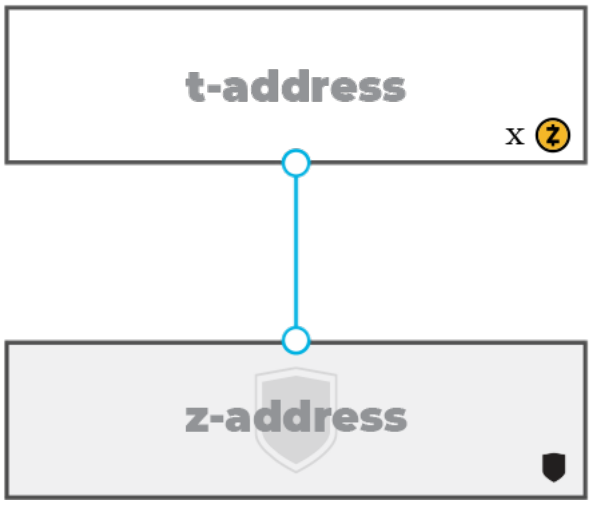


Fig. 3: **Shielding** (Value is revealed on the sender end)

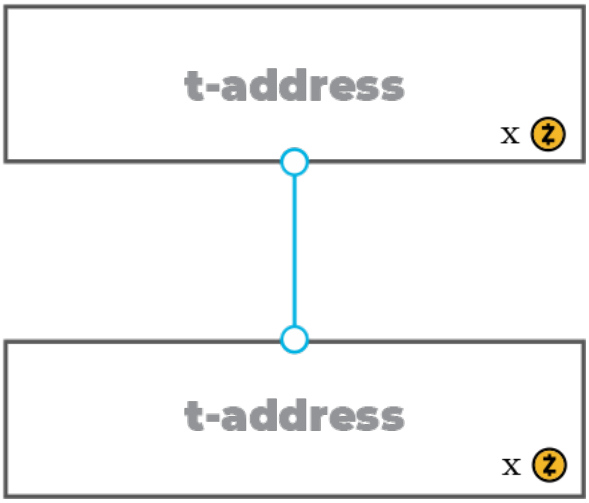


Fig. 4: **Transparent/public** (Value is revealed by both sender and receiver)

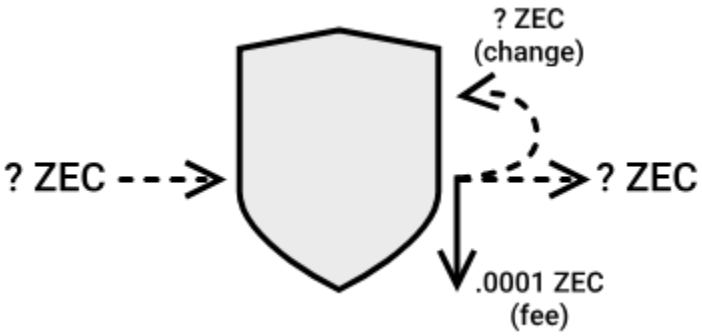


Fig. 5: Shielded addresses and the values sent to or from them are not publicly visible.

Many wallets do not include shielded address support yet but this is expected to change over time with the adoption of Sapling addresses.

Transaction fees are visible regardless of sending to and/or receiving from shielded addresses.

The transaction subsequent to a coinbase transaction (which is always to a transparent address) must be a shielding transaction.

HD Wallets

Sapling addresses support a hierarchical deterministic wallet structure. This allows a master wallet seed to be used as a backup method for all Sapling addresses in a wallet. See the blog post, Sapling in HD to understand more about how this feature is supported. Note that HD support is not enabled for Sprout or transparent addresses.

Viewing Keys

Viewing keys allow for the separation of spending and viewing permissions associated with shielded addresses. Users might want to give third-parties view access to their shielded addresses without also handing over spending capabilities or using a transparent address. For example, consider accounting or auditing use cases.

Currently, viewing keys are only partially supported in Sprout shielded addresses in the form of incoming viewing keys. This means, the viewing key will only be able to track incoming payments to a Sprout address. Sapling addresses do not have any viewing key support. This documentation will be updated when full Sapling address support is integrated.

4.7.3 Transparent Addresses

Transparent addresses work similarly to Bitcoin addresses and do not offer privacy for users.

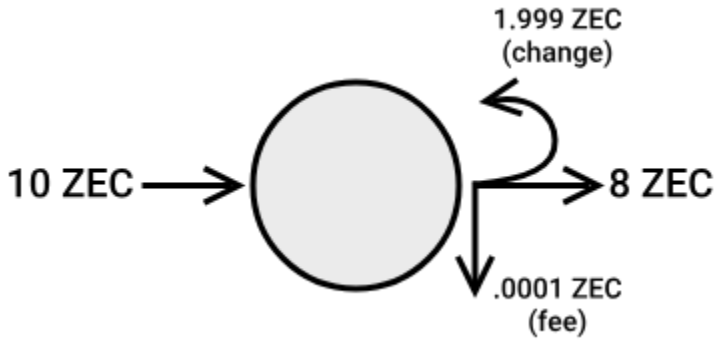


Fig. 6: Transparent addresses and the values sent to or from them are publicly visible

At this time, some advanced features such as multisignature and the use of bitcoin-style scripting with opcodes are only supported by transparent addresses. Multisignature addresses start with a "t3" as opposed to the single signature standard address which start with a "t1".

Many wallets only support transparent addresses.

Coinbase transactions (AKA block rewards and miner fee payouts) can only be sent to transparent addresses.

4.7.4 Value Pools

Since there are 3 distinct address types (transparent, Sapling and Sprout), this means there are 3 *value pools* in which ZEC can be held. All ZEC held in transparent addresses are part of the *transparent value pool*, all ZEC held in Sapling addresses are part of the *Sapling value pool* and all ZEC held in Sprout addresses are part of the *Sprout value pool*. The sum of the pools is equal to the total amount of ZEC in circulation.

Checking the Value Pool Totals

It's possible to use your own node to check the total value in each shielded value pool with a single RPC call to *getblockchaininfo*. One way to issue that is to call `zcash-cli getblockchaininfo` on a computer running a properly-functioning zcashd. The resulting JSON blob contains the perceived totals in the valuePool field. If the value corresponding with the "monitored" json key within the "Sprout" or "Sapling" entries are true, then your values for the pools are correct. If either of them are false, then your figures are wrong and you shouldn't rely on them, and you will need to reindex your node with `zcashd -reindex` to turn "monitored" to "true" at which point you can trust those figures.

The value pools are also monitored at the third-party website zcha.in.

4.7.5 Turnstiles

For each shielded value pool (see above), there exists a turnstile which can calculate the expected amount of ZEC held in it. Since ZEC must be mined to a transparent address before being sent to any shielded address, the value entering either the Sprout or Sapling value pools is visible. Similarly, because ZEC cannot be sent directly between shielded value pools without revealing the amount (see: *Sprout-to-Sapling Migration*), the value exiting a shielded value pool is also visible. This allows for publicly tracking the total value held by shielded pools without having the ability to know individual shielded address balances.

As A Defense Mechanism Against Balance Violations

While maintaining proper balances in Zcash transactions are primarily checked through other means (such as zero-knowledge proofs), the turnstiles are a way to publicly validate this property on a per-value pool basis. From there, defensive measures can be implemented to contain balance violations within an affected value pool.

A new consensus rule in Zcash is being implemented for this very purpose. As defined in ZIP 209: Prohibit Negative Shielded Value Pool, the rule states:

> If the "Sprout value pool balance" or "Sapling value pool balance" were to become negative as a result of accepting a block, then all nodes MUST reject the block as invalid.

4.7.6 Additional Reading

Privacy Considerations for Official Zcash Software & Third-Party Wallets

Anatomy of a Zcash Transaction

Transaction Linkability

Selective Disclosure & Shielded Viewing Keys

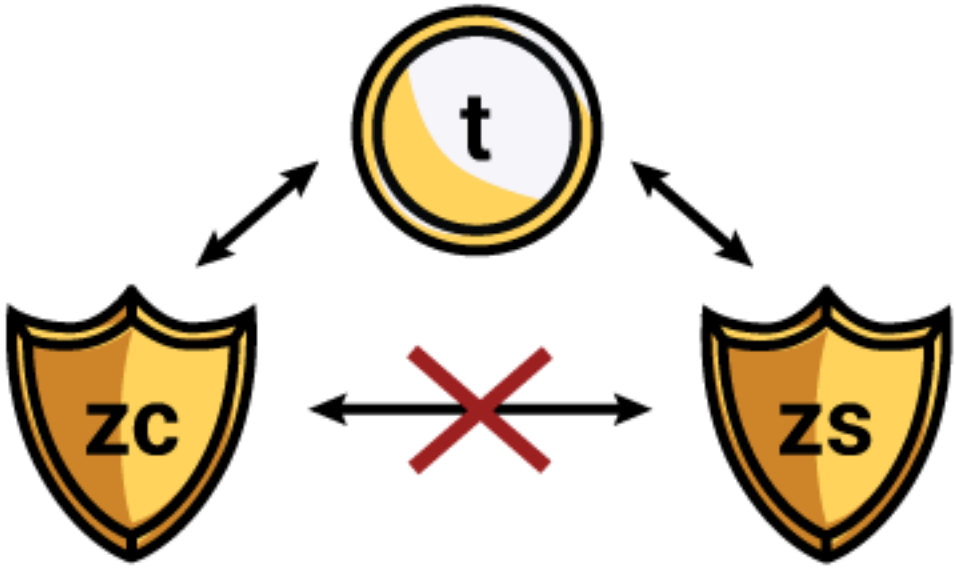
4.8 Sprout-to-Sapling Migration

4.8.1 Overview

The Sapling network upgrade requires a new type of shielded address to support the new usability and security improvements it brings to Zcash. Sapling shielded addresses start with "zs" whereas the legacy, Sprout shielded addresses start with "zc".

The Sprout-to-Sapling migration is an upgrade strategy for funds left in Sprout addresses.

Note: Due to the privacy properties of shielded addresses (both Sprout and Sapling), direct auditing of the total monetary supply is impossible, Turnstiles are a built in feature to monitor value entering and exiting their associated value pools. Users may wish to track the status of value pool totals.



There are two levels to the migration: user and consensus. The user level refers to the standard *zcashd* payment RPCs such as *z_sendmany* and *z_mergettoaddress* which prohibit any transaction from being sent directly from a Sprout address to a Sapling address (and vice versa). For sending between Sprout and Sapling addresses, users of these RPCs **must** use a transparent address as an intermediary which will obviously expose the balance being migrated. As described in *Addresses and Value Pools in Zcash*, all balance transfers from shielded addresses to transparent addresses reveal the value and become associated with those transparent addresses. Transfers from those transparent addresses back into shielded addresses reshield the value. This process is shown in the diagram below.

The consensus level mechanism allows a direct Sprout to Sapling transaction to take place but requires the balance be *passed through* the transparent value pool (see: *Value Pools*) before landing in a Sapling address, thus exposing the

value without a transparent address. Because this may not be obvious to users (and therefore a privacy risk), a UX decision was made to limit availability in the standard RPC. A migration tool is available to make use of this consensus level mechanism. See *Migration Tool* below.

4.8.2 Migration Tool

As of version 2.0.5-2 of zcashd, a Sprout-to-Sapling migration tool is available to help users who have funds stored in older Sprout addresses migrate them to a Sapling addresss. It is **highly recommended** that all users with funds in Sprout addresses make use of this tool instead of manual migration.

Since the exposure of the migrated amount potentially compromises the privacy of users, the tool works by hiding individual migration transactions among those of all users that are doing the migration at around the same time.

The tool will migrate funds from any Sprout addresses in the wallet into a single destination Sapling address. The migration works by creating up to 5 transactions whenever the blockchain reaches a 500 block height interval. The transaction amounts are picked according to a random distribution. The migration will end once the wallet's Sprout balance is below .01 ZEC.

The full design of the tool is specified in ZIP 308.

Using the Migration Tool

Use of the migration tool is implemented in two zcashd RPCs: `z_setmigration` and `z_getmigrationstatus`.

Note: Before running the tool, you may want to specify a specific destination address. The default is the Sapling account 0 address (aka the first address derived from the wallet's master seed). To change this, set another Sapling address by adding the `-migrationdestaddress=<SAPLINGADDR>` parameter in *zcash.conf*.

To activate the migration tool, simply run:

```
$ zcash-cli z_setmigration true
```

Or to deactivate:

```
$ zcash-cli z_setmigration false
```

Note: Nodes will need to stay running until all funds have been transferred. If the node is shut down before completion, you will need to reactivate the tool when the node has been restarted.

You can also enable migration in *zcash.conf* by adding the `-migration` parameter. This will start the migration automatically on restart. If the tool is deactivated with the RPC, a restart of zcashd will reenable it.

To check the status of migration, run:

```
$ zcash-cli z_getmigrationstatus
```

Which will output the following information:

```
{
  "enabled": true|false,                    (boolean) Whether or not migration is␣
↪enabled
```

(continues on next page)

(continued from previous page)

```
  "destination_address": "zaddr",           (string) The Sapling address that will␣
↪receive Sprout funds
  "unmigrated_amount": nnn.n,               (numeric) The total amount of unmigrated␣
↪ZEC
  "unfinalized_migrated_amount": nnn.n,     (numeric) The total amount of unfinalized␣
↪ZEC
  "finalized_migrated_amount": nnn.n,       (numeric) The total amount of finalized␣
↪ZEC
  "finalized_migration_transactions": nnn,  (numeric) The number of migration␣
↪transactions involving this wallet
  "time_started": ttt,                      (numeric, optional) The block time of the␣
↪first migration transaction as a Unix timestamp
  "migration_txids": [txids]                (json array of strings) An array of all␣
↪migration txids involving this wallet
}
```

Once the total held in Sprout address is less than 0.01 ZEC, the tool will disable itself automatically.

4.8.3 Additional Reading

Sapling Addresses & Turnstile Migration

Anatomy of a Zcash Transaction

Transaction Linkability

4.9 Zcash.conf Guide

Below contains information for additional configuration of the `zcash.conf` file.

4.9.1 Network-Related Settings

Parameter	Description & Example
testnet	Run on the test network instead of the real zcash network. `testnet=0`
regtest	Run a regression test network `regtest=0`
proxy	Connect via a SOCKS5 proxy `proxy=127.0.0.1:9050`
bind	Bind to given address and always listen on it. Use [host]:port notation for IPv6 `bind=<addr>`
whitebind	Bind to given address and whitelist peers connecting to it. Use [host]:port notation for IPv6 `whitebind=<addr>`

Quick Primer on addnode vs connect

Let's say for instance you use `addnode=4.2.2.4` addnode will connect you to and tell you about the nodes connected to 4.2.2.4. In addition it will tell the other nodes connected to it that you exist so they can connect to you. Connect will not do the above when you 'connect' to it. It will *only* connect you to 4.2.2.4 and no one else. So if you're behind a firewall, or have other problems finding nodes, add some using 'addnode'. If you want to stay private, use 'connect' to only connect to "trusted" nodes. If you run multiple nodes on a LAN, there's no need for all of them to open lots of connections. Instead 'connect' them all to one node that is port forwarded and has lots of connections.

Thanks goes to [Noodle] on Freenode.

Parameter	Description & Example

addnode	Use as many addnode= settings as you like to connect to specific peers
	`addnode=69.164.218.197`
	`addnode=10.0.0.2:8233`

connect	Alternatively use as many connect= settings as you like to connect
	ONLY to specific peers
	`connect=69.164.218.197`
	`connect=10.0.0.1:8233`

listen	Listening mode, enabled by default except when 'connect' is being used `listen=1`
maxconnections	Maximum number of inbound+outbound connections. `maxconnections=6`

4.9.2 JSON-RPC Options

Controlling a running Zcash/zcashd process

Parameter	Description & Example

addnode	Use as many addnode= settings as you like to connect to specific peers
	`addnode=69.164.218.197`
	`addnode=10.0.0.2:8233`

server	Tells zcashd to accept JSON-RPC commands (set as default if not specified) `server=1`
rpcbind	Bind to given address to listen for JSON-RPC connections. Use [host]:port notation for IPv6. This option can be specified multiple times (default: bind to all interfaces) `rpcbind=<addr>`
rpcuser	If you set an rpcpassword using that option, you must also set rpcuser. `rpcuser=<username>`
rpcpassword	If you specify this option, be sure it is sufficiently-secure, see the notes below. When no rpcpassword option is specified, the daemon now uses a special 'cookie' file for authentication. This file is generated with random content when the daemon starts, and deleted when it exits. Its contents are used as an authentication token. Read access to this file controls who can access through RPC. By default it is stored in the data directory but its location can be overridden with the option -rpccookiefile. `rpcpassword=<password>` **Warning:** You should still set a secure pass-

4.9.3 Transaction Fee

Parameter	Description & Example
sendfreetransactions	Send transactions as zero-fee transactions if possible (default: 0) `sendfreetransactions=0`
txconfirmtarget	Create transactions that have enough fees (or priority) so they are likely to # begin confirmation within n blocks (default: 1). This setting is overridden by the -paytxfee option. `txconfirmtarget=n`

4.9.4 Miscellaneous Options

Parameter	Description & Example
gen	Enable attempt to mine Zcash. `gen=1`
genproclimit	Set the number of threads to be used for mining Zcash (-1 = all cores). `genproclimit=1`
equihashsolver	Specify a different Equihash solver (e.g. “tromp”) to try to mine Zcash faster when gen=1. `equihashsolver=default`
keypool	Pre-generate this many public/private key pairs, so wallet backups will be valid for both prior transactions and several dozen future transactions. `keypool=100`
paytxfee	Pay an optional transaction fee every time you send Zcash. Transactions with fees are more likely than free transactions to be included in generated blocks, so may be validated sooner. This setting does not affect private transactions created with `z_sendmany` `paytxfee=0.00`

4.10 Zcash Mining Guide

Welcome! This guide is intended to get you mining Zcash, a.k.a. “ZEC”, on the Zcash mainnet. The unit for mining is Sol/s (Solutions per second).

If you run into snags, please let us know. There’s plenty of work needed to make this usable and your input will help us prioritize the worst sharpest edges earlier. For user help, we recommend using our forum:

https://forum.z.cash/

4.10.1 Setup

First, you need to set up your local Zcash node. Follow the *User Guide* up to the end of the section *Build* , then come back here.

4.10.2 Configuration

Configure your node as per *Configuration* , including the section *Enabling CPU Mining* .

4.10.3 Mining

Now, start Mining! `./src/zcashd`

To run it in the background (without the node metrics screen that is normally displayed):

`./src/zcashd -daemon`

You should see the following output in the debug log (*~/.zcash/debug.log*):

```
Zcash Miner started
```

Congratulations! You are now mining on the mainnet.

To stop the Zcash daemon, enter the command:

`./src/zcash-cli stop`

Spending Mining Rewards

Coins are mined into a t-addr (transparent address), but can only be spent to a z-addr (shielded address), and must be swept out of the t-addr in one transaction with no change. Refer to our *Zcash Payment API* for instructions on how to use the `z_sendmany` command to send coins from a **t-addr** to a **z-addr**. You will need at least 4GB of RAM for this operation.

Mining Pools

If you're mining by yourself or at home, you're most likely to succeed if you join an existing mining pool. See this community-maintained list of mining pools for further instructions.

P2PKH transactions

The internal *zcashd* miner inherited from Bitcoin used P2PK for coinbase transactions, but Zcash 1.0.6 and later use P2PKH by default , following the trend for Bitcoin.

4.10.4 Configuration Options

Mine to a single address

The internal *zcashd* miner uses a new transparent address for each mined block. If you want to instead use the same address for every mined block, use the `-mineraddress=` option available in Zcash 1.0.6 and later.

4.11 Security Warnings

4.11.1 Security Audit

Zcash has been subjected to a formal third-party security review. For security announcements, audit results and other general security information, see https://z.cash/support/security.html

4.11.2 x86-64 Linux Only

There are known bugs which make proving keys generated on 64-bit systems unusable on 32-bit and big-endian systems. It's unclear if a warning will be issued in this case, or if the proving system will be silently compromised.

4.11.3 Wallet Encryption

Wallet encryption is disabled, for several reasons:

- Encrypted wallets are unable to correctly detect shielded spends (due to the nature of unlinkability of JoinSplits) and can incorrectly show larger available shielded balances until the next time the wallet is unlocked. This problem was not limited to failing to recognize the spend; it was possible for the shown balance to increase by the amount of change from a spend, without deducting the spent amount.
- While encrypted wallets prevent spending of funds, they do not maintain the shielding properties of JoinSplits (due to the need to detect spends). That is, someone with access to an encrypted wallet.dat has full visibility of your entire transaction graph (other than newly-detected spends, which suffer from the earlier issue).
- We were concerned about the resistance of the algorithm used to derive wallet encryption keys (inherited from Bitcoin) to dictionary attacks by a powerful attacker. If and when we re-enable wallet encryption, it is likely to be with a modern passphrase-based key derivation algorithm designed for greater resistance to dictionary attack, such as Argon2i.

You should use full-disk encryption (or encryption of your home directory) to protect your wallet at rest, and should assume (even unprivileged) users who are running on your OS can read your wallet.dat file.

4.11.4 Side-Channel Attacks

This implementation of Zcash is not resistant to side-channel attacks. You should assume (even unprivileged) users who are running on the hardware, or who are physically near the hardware, that your `zcashd` process is running on will be able to:

- Determine the values of your secret spending keys, as well as which notes you are spending, by observing cache side-channels as you perform a JoinSplit operation. This is due to probable side-channel leakage in the libsnark proving machinery.
- Determine which notes you own by observing cache side-channel information leakage from the incremental witnesses as they are updated with new notes.
- Determine which notes you own by observing the trial decryption process of each note ciphertext on the blockchain.

You should ensure no other users have the ability to execute code (even unprivileged) on the hardware your `zcashd` process runs on until these vulnerabilities are fully analyzed and fixed.

4.11.5 REST Interface

The REST interface is a feature inherited from upstream Bitcoin. By default, it is disabled. We do not recommend you enable it until it has undergone a security review.

4.11.6 RPC Interface

Users should refrain from changing the default setting that only allows RPC connections from localhost. Allowing connections from remote hosts would enable a MITM to execute arbitrary RPC commands, which could lead to compromise of the account running zcashd and loss of funds. For multi-user services that use one or more zcashd instances on the backend, the parameters passed in by users should be controlled to prevent confused-deputy attacks which could spend from any keys held by that zcashd.

4.11.7 Block Chain Reorganization: Major Differences

Users should be aware of new behavior in Zcash that differs significantly from Bitcoin: in the case of a block chain reorganization, Bitcoin's coinbase maturity rule helps to ensure that any reorganization shorter than the maturity interval will not invalidate any of the rolled-back transactions. Zcash keeps Bitcoin's 100-block maturity interval for generation transactions, but because JoinSplits must be anchored within a block, this provides more limited protection against transactions becoming invalidated. In the case of a block chain reorganization for Zcash, all JoinSplits which were anchored within the reorganization interval and any transactions that depend on them will become invalid, rolling back transactions and reverting funds to the original owner. The transaction rebroadcast mechanism inherited from Bitcoin will not successfully rebroadcast transactions depending on invalidated JoinSplits if the anchor needs to change. The creator of an invalidated JoinSplit, as well as the creators of all transactions dependent on it, must rebroadcast the transactions themselves.

Receivers of funds from a JoinSplit can mitigate the risk of relying on funds received from transactions that may be rolled back by using a higher minconf (minimum number of confirmations).

4.11.8 Logging z_* RPC calls

The option `-debug=zrpc` covers logging of the `z_*` calls. This will reveal information about private notes which you might prefer not to disclose. For example, when calling *z_sendmany* to create a shielded transaction, input notes are consumed and new output notes are created.

The option `-debug=zrpcunsafe` covers logging of sensitive information in `z_*` calls which you would only need for debugging and audit purposes. For example, if you want to examine the memo field of a note being spent.

Private spending keys for z addresses are never logged.

4.11.9 Potentially-Missing Required Modifications

In addition to potential mistakes in code we added to Bitcoin Core, and potential mistakes in our modifications to Bitcoin Core, it is also possible that there were potential changes we were supposed to make to Bitcoin Core but didn't, either because we didn't even consider making those changes, or we ran out of time. We have brainstormed and documented a variety of such possibilities in issue #826 , and believe that we have changed or done everything that was necessary for the 1.0.0 launch. Users may want to review this list themselves.

4.12 Data Directory Files

Files within the zcashd data directory (`~/.zcash/`) on Linux unless otherwise specified) include:

File	Description
`zcash.conf`	contains configuration settings for zcashd
`zcashd.pid`	stores the process id of zcashd while running
`blocks/blk000*.dat`	block data (custom, 128 MiB per file)
`blocks/rev000*.dat`	block undo data (custom)
`blocks/index/*`	block index (LevelDB)
`chainstate/*`	block chain state database (LevelDB)
`database/*`	BDB database environment
`db.log`	wallet database log file
`debug.log`	contains debug information and general logging generated by zcashd
`fee_estimates.dat`	stores statistics used to estimate minimum transaction fees and priorities required for confirmation
`peers.dat`	peer IP address database (custom format)
`wallet.dat`	personal wallet (BDB) with keys and transactions (keep private, back this up!)
`.cookie`	session RPC authentication cookie (written at start when cookie authentication is used, deleted on shutdown)
`.lock`	data directory lock file (empty)
`testnet3/*`	contains testnet versions of these files, except `zcash.conf`, if running `-testnet`
`onion_private_key`	cached Tor hidden service private key for `-listenonion`

4.13 Tor Support in Zcash

Warning: Do not assume Tor support does the correct thing in Zcash; better Tor support is a future feature goal.

It is possible to run Zcash as a Tor hidden service, and connect to such services.

The following directions assume you have a Tor proxy running on port 9050. Many distributions default to having a SOCKS proxy listening on port 9050, but others may not. In particular, the Tor Browser Bundle defaults to listening on port 9150. See Tor Project FAQ:TBBSocksPort for how to properly configure Tor.

4.13.1 1. Run Zcash behind a Tor proxy

The first step is running Zcash behind a Tor proxy. This will already make all outgoing connections be anonymized, but more is possible.

-proxy=ip:port Set the proxy server. If SOCKS5 is selected (default), this proxy server will be used to try to reach .onion addresses as well.

-onion=ip:port Set the proxy server to use for Tor hidden services. You do not need to set this if it's the same as -proxy. You can use -noonion to explicitly disable access to hidden service.

-listen When using -proxy, listening is disabled by default. If you want to run a hidden service (see next section), you'll need to enable it explicitly.

-connect=X, -addnode=X, -seednode=X When behind a Tor proxy, you can specify .onion addresses instead of IP addresses or hostnames in these parameters. It requires SOCKS5. In Tor mode, such addresses can also be exchanged with other P2P nodes.

In a typical situation, this suffices to run behind a Tor proxy:

```
$ zcashd -proxy=127.0.0.1:9050
```

4.13.2 2. Run a Zcash hidden server

If you configure your Tor system accordingly, it is possible to make your node also reachable from the Tor network. Add these lines to your /etc/tor/torrc (or equivalent config file):

HiddenServiceDir /var/lib/tor/zcash-service/ HiddenServicePort 8233 127.0.0.1:8233 HiddenServicePort 18233 127.0.0.1:18233

The directory can be different of course, but (both) port numbers should be equal to your zcashd's P2P listen port (8233 by default).

-externalip=X You can tell Zcash about its publicly reachable address using this option, and this can be a .onion address. Given the above configuration, you can find your onion address in /var/lib/tor/zcash-service/hostname. Onion addresses are given preference for your node to advertize itself with, for connections coming from unroutable addresses (such as 127.0.0.1, where the Tor proxy typically runs).

-listen You'll need to enable listening for incoming connections, as this is off by default behind a proxy.

-discover When -externalip is specified, no attempt is made to discover local IPv4 or IPv6 addresses. If you want to run a dual stack, reachable from both Tor and IPv4 (or IPv6), you'll need to either pass your other addresses using -externalip, or explicitly enable -discover. Note that both addresses of a dual-stack system may be easily linkable using traffic analysis.

In a typical situation, where you're only reachable via Tor, this should suffice:

```
$ zcashd -proxy=127.0.0.1:9050 -externalip=zctestseie6wxgio.onion -listen
```

(obviously, replace the Onion address with your own). It should be noted that you still listen on all devices and another node could establish a clearnet connection, when knowing your address. To mitigate this, additionally bind the address of your Tor proxy:

```
$ zcashd ... -bind=127.0.0.1
```

If you don't care too much about hiding your node, and want to be reachable on IPv4 as well, use *discover* instead:

```
$ zcashd ... -discover
```

and open port 8233 on your firewall (or use -upnp).

If you only want to use Tor to reach onion addresses, but not use it as a proxy for normal IPv4/IPv6 communication, use:

```
$ zcashd -onion=127.0.0.1:9050 -externalip=zctestseie6wxgio.onion -discover
```

4.13.3 3. Automatically listen on Tor

Starting with Tor version 0.2.7.1 it is possible, through Tor's control socket API, to create and destroy 'ephemeral' hidden services programmatically. Zcash has been updated to make use of this.

This means that if Tor is running (and proper authentication has been configured), Zcash automatically creates a hidden service to listen on. Zcash will also use Tor automatically to connect to other .onion nodes if the control socket can be successfully opened. This will positively affect the number of available .onion nodes and their usage.

This new feature is enabled by default if Zcash is listening (`-listen`), and requires a Tor connection to work. It can be explicitly disabled with `-listenonion=0` and, if not disabled, configured using the `-torcontrol` and `-torpassword` settings. To show verbose debugging information, pass `-debug=tor`.

Connecting to Tor's control socket API requires one of two authentication methods to be configured. For cookie authentication the user running zcashd must have write access to the `CookieAuthFile` specified in Tor configuration. In some cases this is preconfigured and the creation of a hidden service is automatic. If permission problems are seen with `-debug=tor` they can be resolved by adding both the user running tor and the user running zcashd to the same group and setting permissions appropriately. On Debian-based systems the user running zcashd can be added to the debian-tor group, which has the appropriate permissions. An alternative authentication method is the use of the `-torpassword` flag and a `hash-password` which can be enabled and specified in Tor configuration.

4.13.4 4. Connect to a Zcash hidden server

To test your set-up, you might want to try connecting via Tor on a different computer to just a a single Zcash hidden server. Launch zcashd as follows:

```
$ zcashd -onion=127.0.0.1:9050 -connect=zctestseie6wxgio.onion
```

Now use zcash-cli to verify there is only a single peer connection.

```
$ zcash-cli getpeerinfo
```

```
[
    {
        "id" : 1,
        "addr" : "zctestseie6wxgio.onion:18233",
        ...
        "version" : 170002,
        "subver" : "/MagicBean:1.0.0/",
        ...
    }
]
```

To connect to multiple Tor nodes, use:

```
$ zcashd -onion=127.0.0.1:9050 -addnode=zctestseie6wxgio.onion -dnsseed=0 -
↪onlynet=onion
```

4.14 Glossary

Address A Zcash address is similar to a physical address or an email address. It is the only information you need to provide for someone to send you *ZEC*. There are two types of addresses in Zcash: a *shielded address* and a *transparent address*.

Block A block is a record in the Zcash blockchain that contains a set of transactions sent on the network. Pending inclusion in a block, a transaction is kept in the *mempool* in an *unconfirmed* state. Roughly every 2.5 minutes, on average, a new block is appended to the *blockchain* through *mining* and the transactions included receive their first *confirmation*.

Block reward A block reward is new *ZEC* released into the network after the successful *mining* of a block. For the first four years, the block reward in Zcash is split into a *miners' reward* and a *founders' reward*. During this time, miners receive 80% (or 10 *ZEC*) per block with the remaining 20% (or 2.5 ZEC) split between a range of beneficiaries including an Electric Coin Company strategic reserve, the *Zcash Foundation* and many stakeholders including Zcash founders, employees, investors and advisors. After 850,000 *blocks*, the block reward halves for the first time and miners start to receive 100% of the block reward (6.25 ZEC). Each subsequent 840,000 blocks triggers a new block reward halving.

Blockchain The blockchain is a public record of Zcash transactions in chronological order. The blockchain is shared between all Zcash users. It is used to verify the permanence of Zcash transactions and to prevent *double spending*.

Confirmation A transaction confirmation first occurs when that transaction has been included in a *block* and gains an additional confirmation for each subsequent block. The more confirmations a transaction has, the higher the security from a potential reversal (see: *rollback*). Some may consider a single confirmation to be secure for low value transactions, although it is generally recommended to wait for 10+ confirmations.

Cryptography Cryptography is the branch of mathematics that lets us create mathematical proofs that provide high levels of security and privacy. Services like online commerce and banking already use cryptography and in many countries, are required by law to protect customers and their data. In the case of Zcash, cryptography is used to:

1. protect user privacy (via *zk-SNARKs*)
2. make it impossible for anybody to spend funds from another user's wallet
3. prevent corruption of the blockchain database

Electric Coin Company This is the abbreviation for the Electric Coin Company, the team behind the *Zcash protocol*, previously known as the Zcash Company.

Encrypted memo The encrypted memo is an additional field for *transactions* sent to *shielded addresses* that is visible to the recipient of a payment. The encrypted memo is visible only the to the sender and recipient, unless the *viewing key* or *payment disclosure* gets shared with a third party.

Equihash Equihash is a proof-of-work *mining* algorithm that is memory-oriented with very efficient verification.

Experimental feature An experimental feature is one that is available to users on the main *Zcash network* but should undergo further testing by users and developers. Users must explicitly opt into enabling an experimental feature until they become fully supported.

Double spend A double spend happens when a user sends the same *ZEC* to two different recipients. Zcash *miners*, the Zcash *blockchain* and *zk-SNARKs* are integral for only allowing one transaction to *confirm* and be considered valid.

Memory pool The memory pool (or *mempool* for short) is a temporary staging location for *transactions* which have been verified by nodes in the *Zcash network* but have not yet been included in a *block*. Transactions in the memory pool are considered *unconfirmed*.

Mining Mining is the process where for each *block*, nodes in the Zcash network compete by doing complex mathematical calculations to find a solution based on a self-adjusting difficulty. Zcash miners are rewarded with both the *transaction fees* of the *transactions* they confirm and *block rewards*. Zcash uses a proof-of-work mining algorithm called *Equihash*.

Multi-signature A multi-signature address (also referred to as *multisig*) is a type of *address* which requires multiple *private key signatures* in order to spend funds. This is a security mechanism to protect against theft or loss of a private key. Currently, multisig functionality is only supported by *transparent addresses*.

Network upgrade A network upgrade is a *software-updates-required* release of the Zcash software. After *activation* of a network upgrade, network nodes running older versions that are not compatible with the upgrade will be

forked onto an outdated *blockchain* and will require a software upgrade to rejoin the main network. This is sometimes referred to as a *hard fork* upgrade.

Overwinter Overwinter is the first *network upgrade* for Zcash. Its purpose is strengthening the protocol for future network upgrades. It includes versioning, replay protection for network upgrades, performance improvements for transparent transactions and the *transaction expiry* feature. Overwinter *activated* at *block* height 347500.

Payment disclosure A payment disclosure is a method of proving that a payment was sent to a *shielded address* by revealing the value, receiving address and optional *encrypted memo*. The current implementation of this is as an *experimental feature*.

Private Key A private key is a secret string of data that gives access to spend the *ZEC* balance of an associated *address* through a cryptographic *signature*. Your private key(s) may be stored directly in your computer or smartphone, with a custodian such as an exchange or a combination of both using *multisig*. Private keys are important to keep safe as they are the only access to spending the funds you may own. For securing your private keys with the zcashd client, review the *Wallet Backup Instructions*.

Public parameters The Zcash public parameters are a set of global constraints required for constructing and verifying the *zk-SNARKs* used for *shielded addresses*.

Rollback A rollback is when a blockchain is rewound to a previous state and a set of the most recent *blocks* and the *transactions* they contain are discarded. Zcash has a rollback limit of 100 blocks.

Sapling Sapling is a *network upgrade* that introduces significant efficiency improvements for shielded transactions that will pave the way for broad mobile, exchange and vendor adoption of Zcash shielded addresses. Sapling is scheduled to *activate* at *block* height 419200.

Selective disclosure Selective disclosure refers to the features of *shielded addresses* where the owner may *selectively disclose* shielded transaction data. A user may share a *viewing key* or *payment disclosure* with any third party, allowing them to access shielded data while maintaining privacy from others.

Shielded address A shielded *address* (also referred to as a *zaddr*) sends or receives *transactions* such that the address, associated value and *encrypted memo* are not visible on the Zcash *blockchain*. These addresses start with the letter *z*. A shielded address uses *zk-SNARKs* to protect transaction data for value sent or received to it. A transaction consisting of only shielded addresses is called a *shielded transaction*. A transaction consisting of both shielded addresses and *transparent addresses* only protects the data associated with the shielded address. Each shielded address has a *spending key* and *viewing key*.

Shielded transaction A shielded transaction is a transaction exclusively between *shielded addresses*. The addresses, value and optional *encrypted memo* are shielded using *zk-SNARK cryptography* before the transaction is recorded in the *blockchain*.

Signature A cryptographic signature is a mathematical scheme that allows someone to authenticate digital information. When your Zcash *wallet* signs a transaction with the appropriate *private key*, the network can confirm that the signature matches the *ZEC* being spent. This signing is confirmed publicly for *transparent addresses* and through the use of *zk-SNARKs* for *shielded addresses*.

Sol/s Sol/s refers to solutions per second and measures the rate at which *Equihash* solutions are found. Each one of those solutions is tested against the current target (after adding to the block header and hashing), in the same way that in Bitcoin each nonce variation is tested against the target.

Spending key A spending key is a type of *private key* that allows any user in possession of it to spend the balance of the associated *address*. For *shielded addresses*, possessing the spending key also allows the user to view the address' balance and *transaction* data.

Sprout Sprout is the first version of Zcash, launched on October 28, 2016.

TAZ TAZ is the three letter code for the valueless Zcash *testnet* currency.

Testnet The Zcash testnet is an alternative *blockchain* that attempts to mimic the main *Zcash network* for testing purposes. Testnet coins (sometimes referred to as *TAZ*) are distinct from actual *ZEC* and do not have value.

Developers and users can experiment with the testnet without having to use valuable currency. The testnet is also used to test *network upgrades* and their *activation* before committing to the upgrade on the main *Zcash network*.

Transaction A transaction is a payment between users. They are locally created by the user or service then submitted to the *Zcash network* for verification by nodes and eventual *confirmation* into a *block*.

Transaction expiry A transaction expires after staying *unconfirmed* in the *mempool* for too long and is discarded. Once a transaction expires, it may be resubmitted to the network or a new transaction may be submitted in its place. The default expiry in Zcash is 20 *blocks*.

Transaction fee A transaction fee is an additional value added to a *transaction* used to incentivize *miners* to include the transaction into a *block*. Transactions with low or no fee may still be mined but transactions with the default fee or higher will be preferred. If a transaction has too low of a fee, it may stay in the *mempool* until the *transaction expires*. The fee value is not protected for transactions containing *shielded addresses* and therefore it is recommended to always use the default fee of *.0001 ZEC*. Unique fees may result in loss of privacy in some cases.

Transparent address A transparent *address* (also referred to as a *taddr*) sends or receives *transactions* such that the address and associated value are publicly recorded on the Zcash *blockchain*. These addresses start with the letter *t*. A transparent address does not use *zk-SNARKs* to protect transaction data for value sent or received to it. A transaction consisting of only transparent addresses reveals the entire transaction. A transaction consisting of both transparent addresses and *shielded addresses* only reveals the data associated with the transparent address.

Transparent transaction A transparent transaction is a transaction exclusively between *transparent addresses*. The addresses and value are recorded publicly on the *blockchain*.

Upgrade activation An upgrade activation is a specific *block* height that triggers a *network upgrade*.

Viewing key A viewing key is a type of *private key* that allows any user in possession of it to view the balance and transaction data of the associated *shielded address*.

Wallet A Zcash wallet contains *private key(s)* which allow the owner to spend the *ZEC* balance it contains. Each Zcash wallet can show you the total balance of all *ZEC* it controls and lets you pay a specific amount to a specific *address*, just like a real wallet you keep in your pocket or purse. This is different to credit cards where customers are charged by the merchant.

Zcash network The Zcash network is a *peer-to-peer* network of nodes where each node may interact directly with the others for broadcasting newly submitted *transactions*, *mined blocks* and various other messages that regulate behavior. This type of structure removes the need for a trusted regulating central party.

Zcash Zcash is an in-production cryptocurrency implementation of the Zerocash protocol, with security fixes and improvements to performance and functionality. It bridges the existing transparent payment scheme used by Bitcoin with a *shielded* payment scheme secured by *zk-SNARKs*. It implements the *Equihash* proof-of-work *mining* algorithm. Both the network and the associated currency are referred to as *Zcash* with *ZEC* referring specifically to the currency.

Zcash Foundation The Zcash Foundation is a 501(c)3 non-profit dedicated to building Internet payment and privacy infrastructure for the public good, primarily serving the users of the Zcash protocol and blockchain.

ZEC ZEC is the three letter currency code for the Zcash cryptocurrency. It is also used to help distinguish the *Zcash network* from the currency. Note that some exchanges use *XZC* as the Zcash currency code to conform with the ISO 4217 standard for currencies and similar assets not associated with a nation.

Zerocash Zerocash is a cryptographic protocol invented by Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza in 2014. It improves on the earlier *Zerocoin* protocol developed by some of the same authors both in functionality and efficiency.

Zerocoin Zerocoin is a cryptographic protocol invented by Ian Miers, Christina Garman, Matthew Green, and Aviel D. Rubin in 2013. It is a less efficient predecessor of *Zerocash*.

zk-SNARKs A zk-SNARK is a particular form of zero-knowledge proof used in the *Zcash protocol* which allows *shielded addresses* to prove the validity of associated *transactions* without revealing the *address* or value transacted. For Bitcoin and *transparent addresses*, *miners* can verify that a transaction has not been *double spent* because the addresses and their balances are publicly visible within transactions. zk-SNARKs allow this same double spend protection for shielded addresses. The term, which stands for *zero-knowledge Succinct Non-interactive ARguments of Knowledge*, was first used in the *Zerocash* whitepaper.

4.15 Zcash Integration Guide

Zcash is based on Bitcoin, and has a superset of functionality, both in the protocol and the RPC interface. This document describes Zcash integration into services and products. For help building and testing Zcash, see the *User Guide*.

4.15.1 Address Types

Zcash transparent addresses (aka t-addrs), begin with a "t" prefix and are very similar to Bitcoin addresses.

Zcash shielded addresses (z-addrs) which begin with a "z" prefix, are used for sending and receiving shielded funds, with transactions cryptographically protected with zero-knowledge proofs. The Sapling network upgrade added an improved type of shielded addresses which are much more efficient and user friendly. The new Sapling addresses begin with "zs" whereas the legacy, Sprout addresses begin with "zc".

For more information, see the guide to *Addresses and Value Pools in Zcash*.

4.15.2 Bitcoin API

The zcash daemon, *zcashd*, presents the same kind of RPC interface as Bitcoin Core, and this interface (see Bitcoin RPC reference) provides a very similar set of Bitcoin API calls, which we call the *Bitcoin API*. Transactions which only involve transparent addresses can be created with this API just as for Bitcoin.

This API can be used for advanced Bitcoin transactions, just as in Bitcoin Core, such as those involving multisig addresses. Multisig addresses begin with "t3" whereas standard transparent addresses begin with "t1".

4.15.3 Zcash Payment API

In addition, *zcashd* adds the *Payment API* (see *Zcash Payment API* reference). This is a high-level API that simplifies the common use cases of transfers. This API can send from or to both z-addrs and t-addrs through the `z_sendmany` call.

Example of using curl to make a `z_sendmany` call:

```
curl -user $USER -data-binary '{"jsonrpc": "1.0", "id":"curltest", "method": "z_
↪sendmany", "params": ["$FROM_ADDR", [{"address": "$TO_ADDR" ,"amount": $AMOUNT}]] }
↪' -H 'content-type: text/plain;' http://127.0.0.1:8232/
```

This API does not yet support advanced Bitcoin transaction types, such as those involving multisig addresses.

4.15.4 Integration Path

There are two approaches to integrating a service or product with Zcash: the *Bitcoin-compatible* approach, and the *Zcash API* approach. The Bitcoin-compatible approach is convenient for deployments that already use Bitcoin Core,

because the API is (almost) identical. Alternatively, if new integrations are being developed, using the Zcash API may be simpler for most use-cases.

Services that use the Zcash API can send to and receive from both z-addrs and t-addrs. One current drawback is that this API does not support multisig transactions. Services that use the Bitcoin-compatibility approach can only send or receive to/from t-addrs which do not provide the privacy features Zcash is known for.

A service that supports both z-addrs and multisig will use the Zcash API for all transactions except multisig, in which case it will use the Bitcoin API.

Designation		Features to Support			
Level	Description	Transparent Transactions	Transparent Multisig	Private Transactions	Encrypted Memo
1	Bitcoin-compatible	Bitcoin API	Bitcoin API		
2	Zcash API	Zcash Payment API	Bitcoin API	Zcash Payment API	Zcash Payment API

4.15.5 Bitcoin API (JSON-RPC)

- Backwards compatible with Bitcoin-Core 0.11.2 with minor modifications to JSON output.
- Recommended for: time to market for existing Bitcoin applications, familiarity with Bitcoin and multi-sig.

4.15.6 Zcash Payment API

- For sending both transparent and private payments. Extends the existing Bitcoin API with new commands.
- Recommended for: new applications looking to add private transactions and encrypted memo field support which do not need multisig.

4.15.7 Contact Us

For assistance with integrating Zcash into your product, send us a message at ecosystem@z.cash.

4.15.8 Resources

User Guide

Zcash Payment API

Bitcoin RPC reference and Bitcoin API calls

Zcash benchmarking site

4.16 Development Guidelines

We achieve our design goals primarily through this codebase as a reference implementation. This repository is a fork of Bitcoin Core as of upstream release 0.11.2 (many later Bitcoin PRs have also been ported to Zcash). It implements the Zcash protocol and a few other distinct features.

- Bitcoin Core: https://github.com/bitcoin/bitcoin

- Zcash Protocol: https://github.com/zcash/zips/blob/master/protocol/protocol.pdf

4.16.1 Zcash Github Workflow

This document describes the standard workflows and terminology for developers at Zcash. It is intended to provide procedures that will allow users to contribute to the open-source code base. Below are common workflows users will encounter:

1. *Fork Zcash Repository*
2. *Create Branch*
3. *Make & Commit Changes*
4. *Create Pull Request*
5. *Discuss / Review PR*
6. *Deploy / Merge PR*

Before continuing, please ensure you have an existing Github or Gitlab account. If not, visit Github or Gitlab to create an account.

Fork Zcash Repository

This step assumes you are starting with a new Github/Gitlab environment. If you have already forked the Zcash repository, please continue to *Create Branch* section. Otherwise, open up a terminal and issue the below commands:

Note: Please replace `your_username`, with your actual Github username

```
git clone git@github.com:your_username/zcash.git
cd zcash
git remote set-url origin git@github.com:your_username/zcash.git
git remote add upstream git@github.com:zcash/zcash.git
git remote set-url --push upstream DISABLED
git fetch upstream
git branch -u upstream/master master
```

After issuing the above commands, your `.git/config` file should look similar to the following:

```
[core]
    repositoryformatversion = 0
    filemode = true
    bare = false
    logallrefupdates = true
[remote "origin"]
    url = git@github.com:your_username/zcash.git
    fetch = +refs/heads/*:refs/remotes/origin/*
[branch "master"]
    remote = upstream
    merge = refs/heads/master
[remote "upstream"]
    url = git@github.com:zcash/zcash.git
    fetch = +refs/heads/*:refs/remotes/upstream/*
    pushurl = DISABLED
```

This setup provides a single cloned environment to develop for Zcash. There are alternative methods using multiple clones, but this document does not cover that process.

Create Branch

While working on the Zcash project, you are going to have bugs, features, and ideas to work on. Branching exists to aid these different tasks while you write code. Below are some conventions of branching at Zcash:

1. `master` branch is **ALWAYS** deployable
2. Avoid branching directly off `master`, instead use your local fork
3. Branch names **MUST** be descriptive (e.g. `issue#_short_description`)

To create a new branch (assuming you are in `zcash` directory):

```
git checkout -b [new_branch_name]
```

Note: Even though you have created a new branch, until you `git push` this local branch, it will not show up in your Zcash fork on Github (e.g. https://github.com/your_username/zcash)

To checkout an existing branch (assuming you are in `zcash` directory):

```
git checkout [existing_branch_name]
```

If you are fixing a bug or implementing a new feature, you likely will want to create a new branch. If you are reviewing code or working on existing branches, you likely will checkout an existing branch. To view the list of current Zcash Github issues, click here .

Make & Commit Changes

If you have created a new branch or checked out an existing one, it is time to make changes to your local source code. Below are some formalities for commits:

1. Commit messages **MUST** be clear
2. Commit messages **MUST** be descriptive
3. Commit messages **MUST** be clean (see *Squashing Commits* for details)

Commit messages should contain enough information in the first line to be able to scan a list of patches and identify which one is being searched for. Do not use "auto-close" keywords – tickets should be closed manually. The auto-close keywords are "close[ds]", "resolve[ds]", and "fix(e[ds])?"

While continuing to do development on a branch, keep in mind that other approved commits are getting merged into `master`. In order to ensure there are minimal to no merge conflicts, we need `rebase` with master.

If you are new to this process, please sanity check your remotes:

```
git remote -v
```

```
origin    git@github.com:your_username/zcash.git (fetch)
origin    git@github.com:your_username/zcash.git (push)
upstream    git@github.com:zcash/zcash.git (fetch)
upstream    DISABLED (push)
```

This output should be consistent with your `.git/config`:

```
[branch "master"]
    remote = upstream
    merge = refs/heads/master
[remote "origin"]
    url = git@github.com:your_username/zcash.git
    fetch = +refs/heads/*:refs/remotes/origin/*
[remote "upstream"]
    url = git@github.com:zcash/zcash.git
    fetch = +refs/heads/*:refs/remotes/upstream/*
    pushurl = DISABLED
```

Once you have confirmed your branch/remote is valid, issue the following commands (assumes you have **NO** existing uncommitted changes):

```
git fetch upstream
git rebase upstream/master
git push -f
```

If you have uncommitted changes, use `git stash` to preserve them:

```
git stash
git fetch upstream
git rebase upstream/master
git push -f
git stash pop
```

Using `git stash` allows you to temporarily store your changes while you rebase with `master`. Without this, you will rebase with master and lose your local changes.

Before committing changes, ensure your commit messages follow these guidelines:

1. Separate subject from body with a blank line
2. Limit the subject line to 50 characters
3. Capitalize the subject line
4. Do not end the subject line with a period
5. Wrap the body at 72 characters
6. Use the body to explain *what* and *why* vs. *how*

Once synced with `master`, let's commit our changes:

```
git add [files...] # default is all files, be careful not to add unintended files
git commit -m 'Message describing commit'
git push
```

Now that all the files changed have been committed, let's continue to Create Pull Request section.

Create Pull Request

On your Github page (e.g. https://github.com/your_username/zcash), you will notice a newly created banner containing your recent commit with a big green `Compare & pull request` button. Click on it.

First, write a brief summary comment for your PR – this first comment should be no more than a few lines because it ends up in the merge commit message. This comment should mention the issue number preceded by a hash symbol (e.g. #2984).

Add a second comment if more explanation is needed. It's important to explain why this pull request should be accepted. State whether the proposed change fixes part of the problem or all of it; if the change is temporary (a workaround) or permanent; if the problem also exists upstream (Bitcoin) and, if so, if and how it was fixed there.

If you click on *Commits*, you should see the diff of that commit; it's advisable to verify it's what you expect. You can also click on the small plus signs that appear when you hover over the lines on either the left or right side and add a comment specific to that part of the code. This is very helpful, as you don't have to tell the reviewers (in a general comment) that you're referring to a certain line in a certain file.

Add comments **before** adding reviewers, otherwise they will get a separate email for each comment you add. Once you're happy with the documentation you've added to your PR, select reviewers along the right side. For a trivial change (like the example here), one reviewer is enough, but generally you should have at least two reviewers, at least one of whom should be experienced. It may be good to add one less experienced engineer as a learning experience for that person.

Discuss / Review PR

In order to merge your PR with `master`, you will need to convince the reviewers of the intentions of your code.

Important: If your PR introduces code that does not have existing tests to ensure it operates gracefully, you **MUST** also create these tests to accompany your PR.

Reviewers will investigate your PR and provide feedback. Generally the comments are explicitly requesting code changes or clarifying implementations. Otherwise Reviewers will reply with PR terminology:

- **Concept ACK** - Agree with the idea and overall direction, but have neither reviewed nor tested the code changes.
- **utACK (untested ACK)** - Reviewed and agree with the code changes but haven't actually tested them.
- **Tested ACK** - Reviewed the code changes and have verified the functionality or bug fix.
- **ACK** - A loose ACK can be confusing. It's best to avoid them unless it's a documentation/comment only change in which case there is nothing to test/verify; therefore the tested/untested distinction is not there.
- **NACK** - Disagree with the code changes/concept. Should be accompanied by an explanation.

Squashing Commits

Before your PR is accepted, you might be requested to squash your commits to clean up the logs. This can be done using the following approach:

```
git checkout branch_name
git rebase -i HEAD~4
```

The integer value after ~ represents the number of commits you would like to interactively rebase. You can pick a value that makes sense for your situation. A template will pop-up in your terminal requesting you to specify what commands you would like to do with each prior commit:

```
Commands:
p, pick = use commit
r, reword = use commit, but edit the commit message
```

(continues on next page)

(continued from previous page)

```
e, edit = use commit, but stop for amending
s, squash = use commit, but meld into previous commit
f, fixup = like "squash", but discard this commit's log message
x, exec = run command (the rest of the line) using shell
```

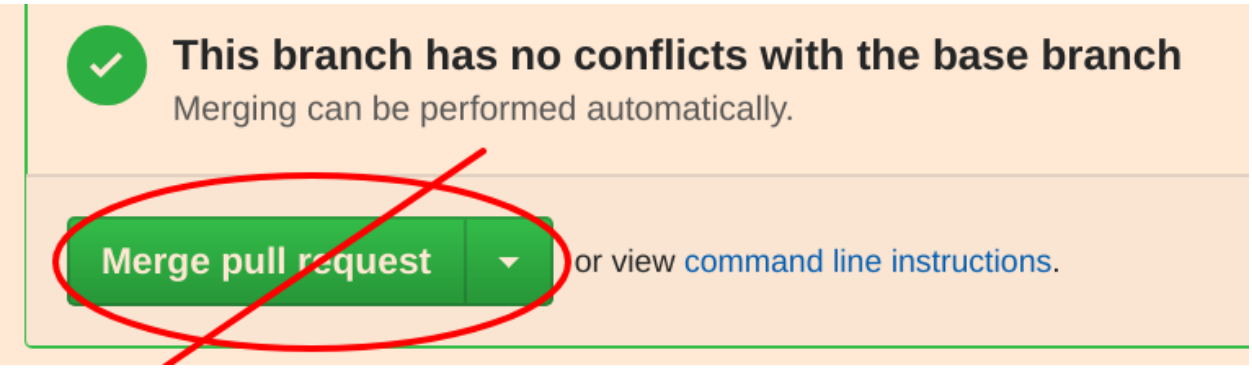
Modify each line with the according command, followed by the hash of the commit. For example, if I wanted to squash my last 4 commits into the most recent commit for this PR:

```
p 1fc6c95 Final commit message
s 6b2481b Third commit message
s dd1475d Second commit message
s c619268  First commit message
```

```
git push origin branch-name --force
```

Deploy / Merge PR

Important: **DO NOT** click on this button! We use a different process (`zkbot`, `Homu`) to merge code



---

zkbot

We use a homu instance called `zkbot` to merge *all* PRs. (Direct pushing to the `master` branch of the repo is not allowed.) Here's just a quick overview of how it works.

If you're on our team, you can do `@zkbot <command>` to tell zkbot to do things. Here are a few examples:

- `r+ [commithash]` this will test the merge and then actually commit the merge into the repo if the tests succeed.
- `try` this will test the merge and nothing else.
- `rollup` this is like `r+` but for insignificant changes. Use this when we want to test a bunch of merges at once to save Buildbot time.

More instructions are found here: http://ci.z.cash:12477/

Once you have addressed the comments in your PR, and it has received two *ACKs* from reviewers, you can attempt to test merge the PR:

```
@zkbot try
```

Note: `@zkbot` commands are entered into Github tickets as comments

This will instruct Buildbot(aka Homu) to test merging your PR with `master` and ensure it passes the full test suite. You may or may not have permissions to run this command, but Github will reply with output indicating if you can or not.

If the `@zkbot try` fails, you will need to go back and address the issues accordingly. Otherwise, you can now attempt to merge into `master`:

```
@zkbot r+
```

Note: `@zkbot` commands are entered into Github tickets as comments

There are very few people that have `@zkbot r+` privileges, so you can request one of these people to merge the PR, or leave it for the release process to pick it up. Finally, when the PR is merged into `master` successfully, your PR will close.

There will be times when your PR is waiting for some portion of the above process. If you are requested to rebase your PR, in order to gracefully merge into `master`, please do the following:

```
git checkout branch_name
git fetch upstream
git rebase upstream/master
git push -f
```

4.16.2 Zcash Developer Workflow

Tip: The flow below assumes you have already downloaded the parameters using `./zcutil/fetch-params.sh`

Below describes a standard workflow for developing code in the zcash repository:

1. **Clone your zcash fork**

   ```
   git clone git@github.com:your_username/zcash.git
   ```

2. **Create a branch for local changes**

   ```
   cd zcash
   git checkout -b [new_branch_name]
   ```

3. **Build zcash**

   ```
   /zcutil/build.sh -j$(nproc)
   ```

4. **Create & build changes to code**

```
make
```

This will allow you to create/edit existing Zcash code, and build it locally. If you want to submit a PR for this newly created code, please refer back to *Make & Commit Changes* section. After completing those steps, please ensure you have also followed *Create Pull Request* and *Deploy / Merge PR* sections.

Coding

See the Developer notes documentation which details coding style, thread handling and additional tips.

Testing

To ensure the existing Zcash code is tested, we use the following tools:

Gtest

Add unit tests for Zcash under `./src/gtest`.

To list all tests, run `./src/zcash-gtest --gtest_list_tests`.

To run a subset of tests, use a regular expression with the flag `--gtest_filter`. Example:

```
./src/zcash-gtest --gtest_filter=DeprecationTest.*
```

For debugging: `--gtest_break_on_failure`.

BOOST

To run a subset of BOOST tests:

```
src/test/test_bitcoin -t TESTGROUP/TESTNAME
```

RPC Tests

To run the main test suite:

```
qa/zcash/full_test_suite.py
```

To run the RPC tests:

```
qa/pull-tester/rpc-tests.sh
```

The main test suite uses two different testing frameworks. Tests using the Boost framework are under `src/test/`; tests using the Google Test/Google Mock framework are under `src/gtest/` and `src/wallet/gtest/`. The latter framework is preferred for new Zcash unit tests.

RPC tests are implemented in Python under the qa/rpc-tests/ directory.

4.16.3 Continuous Integration

Buildbot

Homu

4.16.4 Release Versioning

Starting from Zcash v1.0.0-beta1, Zcash version numbers and release tags take one of the following forms:

v<X>.<Y>.<Z>-beta<N>

v<X>.<Y>.<Z>-rc<N>

v<X>.<Y>.<Z>

v<X>.<Y>.<Z>-<N>

Alpha releases used a different convention: `v0.11.2.z<N>` (because Zcash was forked from Bitcoin v0.11.2).

4.16.5 Release Process

For details on zcashd release processes, see:

- Release Process
- Hotfix Release Process

4.17 Supported Platform Policy

Our supported platforms policy is as follows, largely inspired by the Tahoe-LAFS Buildbot policy

- A supported platform must have a Buildbot builder which builds and runs all tests.
- A Buildbot builder uses the 'default' or 'standard' configuration for its platform, and if we need something substantially different, we call that something different. Examples include default file systems, default compilers, default kernels, etc...
- If build/testing for a supported platform fails, this blocks progress on all platforms.
  - If the merge-acceptance test suite fails only on macOS, but not other platforms, a merge should fail.
  - If the pre-release test suite fails only on Debian, the release is blocked on all platforms.
- Furthermore, for pre-release testing, we should run tests of candidate packages on default systems which don't even have developer tools, unless those tools come by default on that platform.
- For platforms which have frequent updates, such as *Debian testing*, we should upgrade all installed packages on the builders during each *CI deployment*.

By contrast *unsupported platforms* do not block progress. These may still have Buildbot builders, partial test suites. Unsupported platforms should still use default configurations, or be appropriately named to distinguish their uniqueness.

4.18 Zcash Improvement Proposals (ZIPs)

4.18.1 Abstract

A Zcash Improvement Proposal (ZIP) is a design document providing information to the Zcash community, or describing a new feature for Zcash or its processes or environment. The ZIP should provide a concise technical specification of the feature and a rationale for the feature.

We intend ZIPs to be the primary mechanism for proposing new features, for collecting community input on an issue, and for documenting the design decisions that have gone into Zcash. The author(s) of the ZIP are responsible for building consensus within the community and documenting dissenting opinions.

Because the ZIPs are maintained as text files in a versioned repository, their revision history is the historical record of the feature proposal.

See ZIP 0 for complete details including workflow, formatting and management.

See the ZIP repository in GitHub for a full list of accepted proposals and the pull requests for that repository for drafts and suggested updates.

4.19 Network Upgrade Developer Guide

We recommend all wallets, exchanges, and clients that accept/support Zcash to follow these guidelines to prepare for the upcoming network upgrade. Network upgrades on a bi-annual basis to maintain the Zcash network.

Below is general advice that applies to all network upgrades:

Keep your zcashd node updated Check that you are running the latest stable version of zcashd

Version verifiability Clearly state the version of Zcash in a place users can find it. Somewhere inside the client's user interface, state the protocol name and version number (available from the getblockchaininfo method). This allows users to check what version of Zcash their client is running.

Pre-upgrade notification Inform users that a network upgrade is happening before it happens. 4000 blocks (approximately a week) in advance, tell users a network upgrade is happening soon, and that transactions will be unavailable for about an hour at the activation block height.

Defensive transition Disable the initiation of new transactions starting 24 blocks (approximately one hour) before the activation block-height. If a user sends a transaction right before the upgrade, it is likely to not make it onto the chain. This can cause user confusion and frustration.

Post-upgrade notification Tell users when the upgrade has finished and re-enable initiation of transactions. Notify users with a message or at their next login after the network transition.

4.19.1 Blossom

Blossom is the third network upgrade for Zcash.

Shorter Block Target Spacing

This feature increases the frequency of blocks, allowing transactions to resolve faster. This will improve Zcash's usability and increase how many transactions per hour the network can sustain while keeping transaction fees low. This feature has undergone a specification audit. It is currently being tested on testnet as part of the Blossom Network Upgrade Pipeline, before version 2.1.0 of zcashd supporting mainnet activation is released in September.

Consensus Branch ID change See ZIP 206

Amount of FR to be paid in coinbase transaction See ZIP 208 and Protocol Section 7.7

Function from block height to FR has changed See ZIP 208 and Protocol Section 7.8

4.19.2 Sapling

Sapling is a network upgrade that introduces significant efficiency improvements for shielded transactions that will pave the way for broad mobile, exchange and vendor adoption of Zcash shielded addresses.

Transaction formatting

> All transactions must use the new transaction format from Sapling onwards. Make sure that you can parse these *v4* transactions. Previous formats will not be valid after the Sapling upgrade, so if you create transactions, the *v4* format must be used after the upgrade has activated (but not until then). Hardware wallets and SPV clients are particularly affected here.

See ZIP 243. Test vectors for ZIP 243 have been pushed and are being reviewed.

Shielded HD Wallets All Sapling addresses will use hierarchical deterministic key generation according to ZIP 32 (`keypath m/32'/133'/k' on mainnet`). Transparent and Sprout addresses will still use traditional key generation.

See ZIP 32.

Also see Sapling Protocol Specification.

General Guidelines

Using zcashd unmodified

If you use the RPC as provided in the zcashd client, which is true for *most* exchanges and general users of Zcash, you must update your zcashd node to at least version 2.0.1.

For an updated list of specific parameter changes for Sapling in the zcashd wallet RPC, please see: `Sapling RPC Updates v2.0.1` (PDF).

Additionally, Sapling introduces new parameters which must be downloaded by running the `fetch-params.sh` script. These new parameters are placed in the same directory as the older Sprout parameters.

Using custom code to create/sign/send transactions

If you manually create transactions, the following changes are *critical*. Reference section 7.1 of the Sapling specification for complete details:

- The transactions version number **MUST** be 4.
- The version group ID **MUST** be 0x892F2085.
- At least one of tx_in_count, nShieldedSpend, and nJoinSplit **MUST** be nonzero.
- If version 4 and nShieldedSpend + nShieldedOutput > 0 then:
  - Let bvk and SigHash be as defined in §4.12 **'Balance and Binding Signature (Sapling)'**;
  - bindingSig **MUST** represent a valid signature under the *transaction binding verification key* bvk of SigHash - i.e. $\mathrm{BindingSig.Verify}_{\mathrm{bvk}}(\mathrm{SigHash}, \mathrm{bindingSig}) = 1$.
- If version 4 and nShieldedSpend + nShieldedOutput = 0, then valueBalance **MUST** be 0.
- A coinbase transaction **MUST NOT** have any *JoinSplit descriptions, Spend description, or Output descriptions*.

- valueBalance **MUST** be in the range {-MAX_MONEY .. MAX_MONEY}.

In addition, consensus rules associated with each JoinSplit description (§7.2 **'Encoding of JoinSplit Descriptions'**) each Spend description (§7.3 **'Encoding of Spend Descriptions'**) and each Output description (§7.4 **'Encoding of Output Descriptions'**) **MUST** be followed.

Mining Pools

Mining pools running the Stratum protocol will have to make some changes as well.

The `hashReserved` field in the Stratum Protocol will have to be replaced by the `hashFinalSaplingRoot` field from the block header (§7.5 **'Block Header'**).

Testing

Sapling is currently activated on testnet. To test transactions you'll want to follow the *Testnet Guide*. Alternatively, developers can use these features in regtest mode.

4.19.3 Overwinter

Overwinter is the first network upgrade for Zcash. Its purpose is strengthening the protocol for future network upgrades. It includes versioning, replay protection for network upgrades, performance improvements for transparent transactions, a new feature of transaction expiry, and more.

Overwinter activated successfully at block `347500`, mined at `June 25, 2018 20:42 UTC-04:00`

Transaction formatting All transactions must use the new transaction format from Overwinter and onwards. Make sure that you can parse these "v3" transactions (write a parser for them if you aren't using our code). Previous formats will not be valid after the Overwinter upgrade, so if you create transactions, the "v3" format must be used after the upgrade has activated (but not until then). Hardware wallets and SPV clients are particularly affected here. See ZIPs 202 and 203 .

Transaction version number The 4-byte transaction version will have its most significant bit set from Overwinter and onwards, for two-way replay protection of Overwinter and unambiguous transaction parsing of all current and future formats. For example, existing "v1" and "v2" transactions use version numbers "1" and "2", but "v3" Overwinter transactions will use the unsigned version number "(1 << 31) | 3" in the transaction serialization format. See ZIP 202 .

Version group IDs A transaction version will be uniquely paired with a version group ID to ensure unambiguous transaction parsing. For example, a "v3" transaction will always have the version group ID "0x03C48270" in its serialization format, even after future network upgrades. See ZIP 202 .

Branch IDs Each network upgrade has an associated branch ID that identifies its consensus rules. For two-way replay protection, creating transactions will require the branch ID of the current chain tip when signing a transaction (in the BLAKE2b personalization field.) You can obtain the branch ID of any block height from the getblock API. See ZIP 200 .

Signature hashing There are new SegWit-like features in this upgrade, such as transaction signatures committing to values of the inputs. We suggest reusing code from SegWit (e.g. for hashing transparent outputs) when implementing the new SignatureHash function. See ZIP 143 .

Transaction expiry We recommend that you do use the default expiry height (20 blocks/~1 hours) and follow these UX guidelines so that Zcash users can develop a consistent expectation of when Zcash transactions expire and what happens. Zee ZIP 203 .

This isn't an exhaustive list of the changes. Look at the Overwinter Zcash Improvement Proposals (ZIPs) below for complete details on the changes that will be made. The five ZIPs cover network handshaking, transaction format, transaction expiry, signature hashing, and network upgrade mechanisms.

- ZIP 143 Transaction Signature Verification for Overwinter
- ZIP 200 Network Upgrade Mechanism
- ZIP 201 Network Peer Management for Overwinter
- ZIP 202 Version 3 Transaction Format for Overwinter
- ZIP 203 Transaction Expiry

The network upgrade is coordinated via an on-chain activation mechanism.

Zcashd v1.1.0 (and future releases) running protocol version 170005 will activate Overwinter at block 347500 at which point only v3 transactions are processed. Older versions of Zcashd <= 1.0.14, running protocol versions <= 170004, will partition themselves away from the main network into a legacy chain.

Wipeout protection is provided by the new transaction format and signature hashing scheme. Blocks from the legacy chain will not be accepted by the upgraded network. That is, the upgraded network is permanent, and Zcashd v1.1.0 (and future releases) can not reorganize back to the older non-upgraded chain.

Common Issues

tx-overwinter-active This error is simply saying that Overwinter has been activated and your client must be upgraded to the latest version. Upgrade your client and try again. If the issue persists try restarting the client. If this error is appearing on a third party app like a mobile wallet, please file a support request with the developer of the product and let us know in the #user-support channel on the community chat - https://chat.zcashcommunity.com/

mandatory-script-verify-flag-failed (Script evaluated without error but finished with a false/empty top stack element)

This error has been most commonly seen when using *sendrawtransaction*. This can be caused by a few things.

1. When creating raw transactions, the *signrawtransaction* step must be completed correctly. There is a field in *signrawtransaction* called *prevtxs* which can be seen here (https://zcash-rpc.github.io/signrawtransaction.html). The *prevtxs* parameter is optional, but if it is specified, the *amount* parameter must also be specified. This amount is the total amount of the previous output. Prior to Overwinter the *amount* parameter was not required, this is a change between Overwinter and the previous version.

2. This issue can also arise in an edge case where a user is signing the transaction from an offline node. If this is the case the offline node must be synced to above the Overwinter activation height, block 347500.

Node sync is stuck before Overwinter activation height This bug occurs when you are starting a fresh node or restarting a node that is not synced to above the Overwinter activation height (block 347500) and causes the node to sync very slowly. The bug has to do with your node incorrectly banning peer nodes. The end result is your node will sync very slowly as it will not be able to maintain as many connections to other nodes as usual.

This issue has been fixed in 2.0.0. Please update your client to 2.0.0 or above.

4.20 Testnet Guide

The Zcash testnet is an alternative blockchain that attempts to mimic the main Zcash network for testing purposes. Testnet coins are distinct from actual ZEC and do not have value. Developers and users can experiment with the testnet

without having to use valuable currency. The testnet is also used to test network upgrades and their activation before committing to the upgrade on the main Zcash network.

4.20.1 Joining

In order to use the testnet, you must:

Follow the *User Guide* to install a Zcash node

Set the *Configuration* for your node to sync with testnet (and optionally mine testnet coins)

4.20.2 Using

Obtain testnet coins (TAZ) You can use the testnet faucet or enable mining in your configuration to obtain TAZ

Creating transactions Use the *Zcash Payment API* to create transactions on testnet

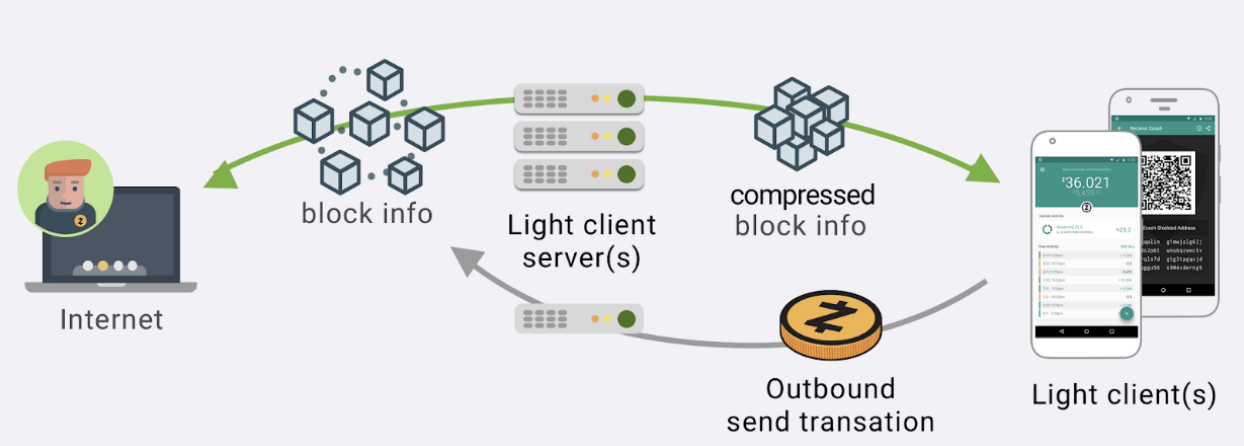
Testnet explorers You can use the Zcash testnet explorer or other, third-party testnet explorers including testnet.zcha.in.

Note: See the *Network Upgrade Developer Guide* for details on testing network upgrades.

4.21 Shielded Support Resources

Everything you need to integrate and support zcash shielded addresses into your light wallet.

Resources are at a proof-of-concept stage and is subject to breaking changes at any time.



4.21.1 Librustzcash

Latest working branch: https://github.com/str4d/librustzcash/tree/note-spending-v5

A set of APIs that collectively implement an SQLite-based light client for the Zcash network. Compiles for Android, iOS, and JS web apps.

Functions

get_address	Returns the address for the account.
get_balance	Returns the balance for the account, including all mined unspent notes that we know about.
get_received_memo_as_utf8	Returns the memo for a received note, if it is known and a valid UTF-8 string.
get_sent_memo_as_utf8	Returns the memo for a sent note, if it is known and a valid UTF-8 string.
get_verified_balance	Returns the verified balance for the account, which ignores notes that have been received too recently and are not yet deemed spendable.
init_accounts_table	Initialises the data database with the given [`ExtendedFullViewingKey`]s.
init_blocks_table	Initialises the data database with the given block.
init_cache_database	Sets up the internal structure of the cache database.
init_data_database	Sets up the internal structure of the data database.
scan_cached_blocks	Scans new blocks added to the cache for any transactions received by the tracked accounts.
send_to_address	Creates a transaction paying the specified address from the given account.

Objectives

1. separate out zcash-specific on-wallet cryptographic functionalities into a module and expose it via an API
2. maintain for third parties, handing updates from releases to zcashd, network upgrades for the Zcash

Notes

- Communicates with a light wallet via shared SDK.
- Works on testnet and mainnet.
- Only handles Sapling shielded addresses and transactions, not compatible with Sprout shielded addresses or transactions. This is on purpose, as Spout addresses are retired.

It has documentation that you can build and view with the following commands:

```
git clone https://github.com/str4d/librustzcash.git
cd librustzcash
git checkout note-spending-v5
cd zcash_client_sqlite
cargo +nightly doc --all --open
```

You may need to run this command first to get the doc to open:

```
rustup toolchain install nightly
```

Or alternatively, you can open docs the latest stable doc (not nightly build):

```
cargo doc --all --open
```

4.21.2 Lightwalletd

https://github.com/zcash-hackworks/lightwalletd/

A stateless server that light clients pull relevant data from. It fetches blockchain data from zcashd, processes them to reduce data, and stores it in a database.

Objectives

- Cache Zcash compact blocks to serve to multiple light clients on demand
- Pre-processes Zcash blocks to save bandwidth for light clients
- An adaptive layer that will allows light clients with different requirements to get relevant data without interacting with zcashd directly

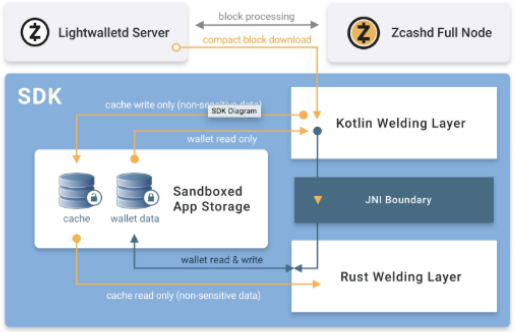
How blocks are pre-processed:

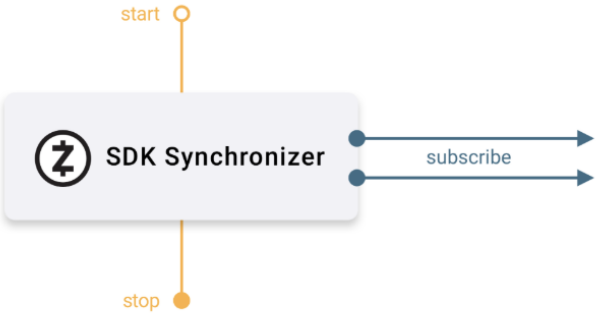
- Drop extra information from blocks without sapling inputs or outputs (drops ~95% of blocks)
- Drop any header information that is not relevant for a light client functionalities (5% of a block); i.e. a light wallet does not need to validate the blockchain.
- Drop the memo field (70% of the block), which is currently rarely used. This renders validation functions invalid, but this is not relevant to light clients.

4.21.3 Android SDK

https://github.com/zcash/zcash-android-wallet-sdk/tree/preview

A lightweight SDK that consumes data from librustzcash and exercises librustzcash to send and receive shielded transactions.





For an overview of the Android SDK, see the android sdk readme. The synchronizer is the primary interface for interacting with the SDK. It exposes an API and the set of functions which can be found on this github synchronizer readme.

Objectives

- Demonstrate how to interact with lightwalletd and librustzcash
- Showcase design best practices from ECC around Zcash specific features
- Optimize for efficiency leveraging Android-specific best practices

Known issues

- The latest developer preview code lives in the preview branch. Always use the preview branch even for documentation, master is behind. (This is due to a lack of engineers that can code review in house, we're hiring to fix this.)
- This has not yet been published to a package manager (bintray).

4.21.4 iOS SDK

This does not exist yet, but is planned work. We hired an iOS developer on August 26th, 2019!

Note: librustzcash can be compiled on iOS, so a motivated development team can currently make an iOS shielded app happen! What we're working on are supportive resources and reference code.

4.21.5 WASM framework

https://github.com/str4d/zcon1-demo-wasm

A minimal functioning demo web wallet. This is not officially maintained.

Objectives * Target desktop OSes and allow them to use shielded addresses * Separate out web-specific wallet functionalities * Allow web browsers to talk to gRPC servers, via a proxy (cannot talk directly)

4.21.6 FAQ

Librustzcash

Do I need to import the entire Rust standard library? There is a base overhead when including Rust code in non-Rust binaries, but what gets compiled in is (roughly) only what the Zcash Rust code actually uses.

What is the smallest size that this can be? We have not tested this yet.

Has this code been audited? Yes, internally and externally. For more information, read the blog post: https://electriccoin.co/blog/blossom-network-upgrade-and-wallet-security-audits/. We have not addressed all the issues from the audits, but none of the outstanding issues were marked as high priority.

How is this related to/communicating with zcashd? zcashd calls into the librustzcash library for some functions, which depends on the same core Zcash Rust libraries as zcash_client_sqlite. librustzcash.h is the API used by zcashd.

How is this related to the Android SDK? The Android SDK uses zcash_client_sqlite's API to get balances, update balances, scan for blocks, and send memos (see full list of functions above). There is a thin "welding layer" between zcash_client_sqlite and the Android SDK that allows this communication.

How is this related to lightwalletd? Lightwalletd is abstracted away, and they commuicate only via two SQLite databases: 1) A cache database, used to inform the light client about new CompactBlocks. It is read-only within all light client APIs except for [init_cache_database] which can be used to initialize the database. 2) A data database, where the light client's state is stored. It is read-write within the light client APIs, and assumed to be read-only outside these APIs. Callers MUST NOT write to the database without using these APIs. Callers MAY read the database directly in order to extract information for display to users.

Lightwalletd

How is this related to the Android SDK? It pre-processes blocks so that information not necessary for light clients is dropped, but burdens the Android SDK to process the information further for its specific implementation.

How is this related to librustzcash? It does not need to know about librustzcash.

What is the threat model? We assume that the full-sized zcash blocks given to lightwalletd are correct, that light clients ask for things that they need and honestly (no DDoS or malformed request protections), and that t the Zcash network is more or less stable (reorgs, forks, etc. cause some disruption and in severe cases malfunction).

Can I run this on an EC2 instance? Yes.

Are you using containers? Yes.

Are you using an orchestrator/scheduler? Not yet.

Are you using load balancers? Not yet.

Can we run many instances and use a load balancer? Yes! It's designed to be stateless and work with a load balancer

What are the expected costs? We have not estimated the costs yet.

What database are you using? SQLite, but it's not required. Using another database will require modifications to the lightwalletd codebase, though.

How do you handle testing and lifecycle? Do you have a CI/CD pipeline? We are actively updating the test stack and lifecycles. We do not have a CI/CD pipeline yet, but MRs are in place.

How do you handle logs? We look at them on an as-needed basis.

What metrics stack do you use, if any? We don't have one.

Android SDK

What resources are available? We have two sample apps: addressAndKeys and memo. The former shows what's necessary for receiving funds. The latter shows the bare minimum for sending. We also have two full demo apps: the reference wallet, Zcon1 swag app.

What does SDK not do? We purposefully leave key management, import, export, seed generation to the specific third party wallets because our partners have told us these things are what their wallet apps do best. We have yet to implement: visibility into incoming memos, background operations that extend beyond the life of the app (not using workmanager yet), modularization of dependencies (make it easier to plug and play different tech stacks for networking, persistence, etc.).

What is the threat model? See the Android SDK Threat Model.

How is this related to librustzcash? It shields librustzcash from needing to know about lightwalletd, and gives it the data it needs for the computation it needs.

How is this related to lightwalletd? It is the only thing that needs to know how about lightwalletd's existence and interacts with it directly. It synchronizes compact blocks and persists them on the phone for librustzcash to use.

What are your minSdk and targetSdk versions? We support API 16 and above for certain chip sets but we optimize for API versions 21 and above. We currently target API 28 and will update to 29 (Android Q) once it is out of beta.

What architectures are you targeting? Currently we're targeting ARM64, ARMv7 and x86.

Which steps do you require for keeping your play store app size small? Do you use APK splitting, ABI filtering, proguard, etc.? We do not release an app on the play store, our reference is app is for learning purposes and we do not use APK splitting, ABI filtering, or proguard.

What Android networking libraries/utilities do you use? Grpc and Protobufs.

Are you doing cross-platform development i.e. sharing code with iOS/Web via things like C, kotlin, rust, or react? No.

Do you use Kotlin coroutines, channels, RxJava, etc? Yes, we make heavy use of coroutines which can adapt to support RxJava.

4.22 Zcash Rust Architecture

4.22.1 Current Design

zkcrypto/pairing

- Implements BLS12-381

zkcrypto/bellman

- Implements Groth16, Circuit API

zcash-hackworks/sapling-crypto

- Implements Sapling/Sprout circuits on top of bellman
- Implements Jubjub
- Implements some Sapling primitives necessary for testing

zcash/librustzcash

- "Thin" FFI surrounding our crypto, for zcashd

4.22.2 Current Issues

- We're doing lots of refactorings and improvements to the code, but these will span many different crates until we get to a stable point. Hard to review and coordinate.
  - Example: bellman is going to be a "circuit-only" thing, agnostic to the proving system. groth16 crate will handle groth16.
  - Example: hardware wallets only want/need jubjub and sapling primitives, so we need to pull out zk-SNARK stuff (which requires an allocator, standard library, etc.)
- Code is inconsistent (with naming, as far as we know) with specification
- Nothing is labeled as constant/variable time

4.22.3 New Design

zkcrypto/jubjub

- Implements Fp, Jubjub
- No standard library requirement

zkcrypto/bls12-381 (depends on jubjub)

- Implements BLS12-381, serial FFT

zkcrypto/bellman

- Implements common circuit synthesis API, gadgets

zkcrypto/groth16

- Implements groth16

4.22.4 Strategy

- librustzcash repository is a Rust workspace containing all of our dependencies, for the time being, via git subtrees
- We refactor code and integrate test vectors closely, following stringent code review processes and quality policies
- Later, we break the subtrees out into crates with stable APIs

4.22.5 End Goal

- Complete cleanup of code (match spec, best practices)
- More members of the team learn how all this stuff works, good documentation
- Refactor of code into modular pieces that all relate to each other nicely
- no_std support for hardware wallets and other projects
- In the meantime, everything is CI'd and developed together
- The coolest, most awesome crypto codebase written in Rust anywhere in the world

4.23 Wallet Developer UX Checklist

We have compiled a checklist of good practices that can be applied to any cryptocurrency wallet with a graphic user interface. `Download the UX Checklist PDF.`

4.23.1 Zcash Features

If you're building a wallet that supports Zcash, we encourage that you follow these guidelines.

Addresses

Use an address persistently for each use Use an address like how you would use different bank accounts (spending, saving, business spending). Transparent addresses aren't private, so we urge users to keep this in mind. We discourage using a new transparent address for each transaction; this only provides a false sense of privacy . Shielded addresses maintain the privacy of transactions, there is no added benefit of using a new shielded address per transaction.

Indicate that transparent addresses are not encrypted! A transaction involving a transparent address (either as sender or recipient) posts the details of the transparent address and amount publicly on the blockchain.

Indicate that shielded addresses are encrypted! A shielded transaction, where funds are sent from one shielded address to another shielded address, only reveals a transaction legitimately and safely happened. The sender, receiver, and amount are not revealed on the blockchain.

If there is no zaddr support, state so clearly Most wallets throw an error stating, "invalid address" or "invalid input." We suggest instead saying, "shielded address are not supported." to indicate that shielded addresses are still valid addresses.

Provide taddr and zaddr support if possible Shielded addresses provide privacy via encryption and is Zcash's main feature. However, most mobile and desktop wallets don't support sending to shielded addresses, so some users will likely be unable to send ZEC to a shielded address.

Warn users when sending from zaddrs to taddrs (deshielding transactions) Explicitly tell users that they are about to reveal transaction information. We don't think warnings other types of transactions are necessary.

Show an available balance vs owned balance Show two balances, one which includes unconfirmed finds, and another not including unconfirmed funds, i.e. "Balance: 621.14321 ZEC (605.35620 ZEC spendable)."

Transactions

Clearly state the fee structure The default transaction fee is set at 0.0001 ZEC. We encourage that you use the default fee, so that transparent and shielded transactions have the same fee–that way, privacy doesn't cost more for users!

Disable users from setting their own transaction fees Do not allow users to customize fees. Our network is fast enough that mining incentivization is not an issue. Unique transaction fees can cause linkability within transactions, especially for zaddrs.

Do not differentiate between types of transactions We do not currently distinguish between different types of Zcash transactions: transparent (transparent to transparent), shielding (transparent to shielded), deshielding (shielded to transparent), and shielded (shielded to shielded) transactions. This is complicated by the ability to send to/from a combination of shielded and transparent addresses.

Use the default transaction expiry time Transaction expiry (see ZIP here) is set to 20 blocks by default, which is ~1 hour. Use this default global runtime option so Zcash users can develop a consistent expectation of when Zcash transactions expire. We don't support expiry time as a per-transaction runtime option.

Visibly mark newly sent transactions in a "pending" state We suggest having a "pending transactions" or "unconfirmed transactions" section, but you can also distinguish it in the list of chronological transactions by using a color or an icon.

Tell the user the expected remaining time to expiry Users should be able to see how much time/blocks are remaining until their transaction expires. Once confirmed (10+ confirmations), unmark the sent transaction visibly in a "complete" state.

If expired, visibly mark the transaction expired and notify the user Rather than deleting the attempted transaction, keep the expired transaction in the log, but distinguished as such. We also encourage giving users suggestions on troubleshooting their transaction .

Viewing Keys

Use viewing keys for watch only wallets Share a viewing key with yourself to create a wallet that tracks your funds while keeping your main funds offline. Watch-only wallets are the first application of viewing keys; we exploring additional use cases as well.

Secure communication channel Encourage secure communication channels by supporting one; viewing keys should not be copy and pasted into a text or email.

Indicate that viewing keys are for all incoming transactions: At version 1.0.14, a viewing key allows the holder of the key to see all incoming transactions since the zaddr was created, but not outgoing transactions.

Memo Fields

Show the memo field in the UI Even if the memo field is empty, show that the field is empty rather than removing it from the UI. This is a good nudge to remind users that a memo field exists.

Liberal Use The memo field is always present and is always exactly 512 bytes long; this is necessary for privacy so that an observer can't detect the different usage patterns and memos. This means that the cost is baked in so that you don't pay a higher fee for including a memo, so encourage users to use the memo.

4.23.2 General

We encourage that you use this checklist to catch common usability problems before launch or user testing.

User Interaction

Progressive Disclosure Things should progress from simple to complex; only the necessary or requested information is displayed at a given time. This helps manage complexity without becoming confused, frustrated, or disoriented.

Feedback Every action should produce a visible, understandable, and immediate reaction. Failing to acknowledge an interaction can lead to unnecessary repetition of actions or errors (i.e. clicking "send" multiple times).

Priming Tell people what they can expect and what they should do. For example, explaining that a camera is needed to scan QR codes before you ask for camera permissions is likely to have users who want that feature to accept it.

Communication Be context-aware of what the user is doing and the nature of the message. For instance, notify of events like transaction confirmations with push notifications, since they're probably not waiting on the app for the confirmation.

Error Handling The best way to handle errors is to prevent them. But if one occurs, put next to the relevant input field (not just at the top or bottom of the screen) to show users what they need to fix without searching for it. It should describe what happened, why it happened, suggest a fix, and not blame the user.

User interface

Hierarchy Information is presented in order of importance and the visual hierarchy of actions on a screen matches what the user expects to do first, second, third, etc.

Simplification Limit the choices that a user is presented with per screen. Provide appropriate filters if there is a large data set.

Consistency Components with a similar behavior should have a similar appearance. For example, all buttons that send a transaction should be blue, square, and labeled 'send.'

Predictability Set good expectations. From looking at your interface, with no previous use, users should be able to answer things such as "where am I?," "what can I do here?," "where can I go from here?," and "what does this button do?."

Visibility Discoverability shouldn't involve luck or chance. If a page requires scrolling, indicate that more content is below the screen by showing half of an image. If there are some screens you want users to find, the menu that links to those pages persists everywhere.

Content

Market Information Provide an up-to-date crypto to FIAT currency conversion, along with current exchange rates between cryptocurrencies.

Network Information Tell users if their transaction is likely to be processed quickly or not, based on mempool congestion.

Account Information Show the balance, minimum spendable, maximum spendable, and other account-specific information.

Fee Information Show how much the fee is, what % of the transaction it is, and if it's added on top or included.

Simplify Jargon Translate what a concept or event affects the user, rather than exposing or explaining what it is technically. For instance, say if the transaction has been confirmed or not, instead showing the number of confirmations or how many confirmations is considered safe.

Navigation

Persistence The navigation bar should always be visible on every screen. If it isn't, users don't know what to do next or don't know how to do the next thing.

Uniformity Similarly styled navigational elements should behave similarly. Additionally, elements of navigation should never appear and disappear, rearrange in order, or move to a different location.

Method Choose the method that most easily lets the users find what they want. This is specific to the use case. Method include browsing via a navigation system, searching with keywords, or filtering to narrow large lists.

Sorting Alphabetical sorting is avoided unless necessitated by many navigational choices (7+). Sort by relevance, related groups, or anything else instead.

Labeling Use meaningful labels and icons for navigation menu items, links, and buttons. Don't force people to chase information they need.

Visual Design

Alignment Every element in the UI should be aligned with one or more other elements. Alignment provides cognitive stability and creates visual relationships. In this same vein, left-align large blocks of text as users need to expend more energy to track the lines. Eyes fatigue faster, comprehension slows, but the users may not be aware why.

Proximity Group certain elements (navigation, header, articles, etc.) contextually to form a perceived whole. For the same reason, visually separate unrelated items.

Repetition Use repetition to create a hierarchy of visual styles . This principle applies to fonts but also colors, textures, and graphical elements. (For instance, all titles should be of one size, all buttons are square, all colors are in a color palette, etc.) Reusing elements of visual styles in visual elements creates cohesiveness.

Contrast Text is easily readable when stark, complementary colors are used A lack of contrast between text and background strains the eyes because they don't know which color to focus on.

4.24 Contributor Code of Conduct

As contributors and maintainers of this project, and in the interest of fostering an open and welcoming community, we pledge to respect all people who contribute through reporting issues, posting feature requests, updating documentation, submitting pull requests or patches, and other activities.

We are committed to making participation in this project a harassment-free experience for everyone, regardless of level of experience, gender, gender identity and expression, sexual orientation, disability, personal appearance, body size, race, ethnicity, age, religion, or nationality.

Examples of unacceptable behavior by participants include:

- The use of sexualized language or imagery
- Personal attacks
- Trolling or insulting/derogatory comments
- Public or private harassment
- Publishing other's private information, such as physical or electronic addresses, without explicit permission
- Other unethical or unprofessional conduct

Project maintainers have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, or to ban temporarily or permanently any contributor for other behaviors that they deem inappropriate, threatening, offensive, or harmful.

By adopting this Code of Conduct, project maintainers commit themselves to fairly and consistently applying these principles to every aspect of managing this project. Project maintainers who do not follow or enforce the Code of Conduct may be permanently removed from the project team.

This Code of Conduct applies both within project spaces and in public spaces when an individual is representing the project or its community.

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by contacting a project maintainer (see below). All complaints will be reviewed and investigated and will result in a response that is deemed necessary and appropriate to the circumstances. Maintainers are obligated to maintain confidentiality with regard to the reporter of an incident.

You may send reports to our Conduct email.

If you wish to contact specific maintainers directly, the following have made themselves available for conduct issues:

- Daira Hopwood (daira at z.cash)
- Sean Bowe (sean at z.cash)

This Code of Conduct is adapted from the Contributor Covenant, version 1.3.0, available at https://www.contributor-covenant.org/version/1/3/0/

4.25 Expectations for DNS Seed operators

Zcash attempts to minimize the level of trust in DNS seeds, but DNS seeds still pose a small amount of risk for the network. As such, DNS seeds must be run by entities which have some minimum level of trust within the Zcash community.

Other implementations of Zcash software may also use the same seeds and may be more exposed. In light of this exposure, this document establishes some basic expectations for operating DNS seeds.

0. A DNS seed operating organization or person is expected to follow good host security practices, maintain control of applicable infrastructure, and not sell or transfer control of the DNS seed. Any hosting services contracted by the operator are equally expected to uphold these expectations.
1. The DNS seed results must consist exclusively of fairly selected and functioning Zcash nodes from the public network to the best of the operator's understanding and capability.
2. For the avoidance of doubt, the results may be randomized but must not single out any group of hosts to receive different results unless due to an urgent technical necessity and disclosed.
3. The results may not be served with a DNS TTL of less than one minute.
4. Any logging of DNS queries should be only that which is necessary for the operation of the service or urgent health of the Zcash network and must not be retained longer than necessary nor disclosed to any third party.
5. Information gathered as a result of the operators node-spidering (not from DNS queries) may be freely published or retained, but only if this data was not made more complete by biasing node connectivity (a violation of expectation (1)).
6. Operators are encouraged, but not required, to publicly document the details of their operating practices.
7. A reachable email contact address must be published for inquiries related to the DNS seed operation.

If these expectations cannot be satisfied the operator should discontinue providing services and contact the active Zcash development team as well as creating an issue in the Zcash repository.

Behavior outside of these expectations may be reasonable in some situations but should be discussed in public in advance.

4.25.1 See also

- zcash-seeder is a reference implementation of a DNS seed.

4.26 Insight Block Explorer Guide

You can run the `zcashd` client with additional features enabled that allow it to support the Insight block explorer. These features are not enabled by default because they increase the disk space used by the data directory `~/.zcash` and will also reduce performance. A block-explorer-enabled `zcashd` can perform all the functions of a normal `zcashd` but typically will not be used as a wallet (will contain no private keys).

There are three areas of change, which are described below:

- Configuration (additional `~/.zcash/zcash.conf` lines)
- New fields added to the existing `getrawtransaction` RPC
- A set of new RPC (cli) methods

The examples show actual (though in some cases, truncated for simplicity) testnet results, and can be reproduced on testnet by copying and pasting the command line (after starting an explorer-enabled `zcashd`). Since the blockchain state will have changed since this writing, the results may no longer be the same as shown here. Also, for brevity, this document doesn't show some of the example command results, when the purpose is to illustrate how parameters can be specified.

You can search for the transactions and addresses used in these examples using an existing testnet block explorer such as https://explorer.testnet.z.cash so you can see the correspondence between the RPC results and the block explorer results.

4.26.1 Configuration changes

The `zcashd` process must be started with the following options, either on the command line or in the `zcash.conf` file (see *Zcash.conf Guide*):

```
txindex=1
experimentalfeatures=1
insightexplorer=1
```

After enabling these configuration options, it will be necessary to reindex the database by starting `zcashd` one time with the `--reindex` command-line option (this will take some time) and waiting for it to fully sync with the blockchain. The `txindex=1` option causes `zcashd` to store all blockchain transactions locally, which requires more storage.

4.26.2 Additional `getrawtransaction` fields

If the `insightexplorer` configuration option is enabled, the `getrawtransaction` RPC verbose result will include new fields whose values are derived (of course, transactions themselves include no additional information). None of the existing fields is modified, so these additions should be backward compatible; existing consumers of the result will continue to work unchanged, and they will be unaware of the additional fields.

Here is an example result, edited for clarity here by showing **only** the added fields (this is a real testnet transaction, so you may try this yourself, or search a block explorer for this transaction and see the correspondence):

```
$ zcash-cli getrawtransaction␣
↪91cbdf382fc7244fab745a2322e336444cb23ba4aec149574dc9b5c77dea7005 1
{
  [...]
  "vin": [
    {
      [...]
      "value": 654.99971929,
      "valueSat": 65499971929,
      "address": "tmV5PUniaX9NMUfe76kTRWd7B8pAXhkq3Sf"
    }
  ],
  [...]
  "vout": [
    {
      [...]
      "spentTxId": "ed202d138c7103fbd2271d577644f4500c68daaccfb7b008995a06e501256cf7",
      "spentIndex": 0,
      "spentHeight": 458561
    },
    {
      [...]
      "spentTxId": "2c6896af644c29481b1682f3da00d2f218f340293c6c7f009d6f21a19ea7310f",
      "spentIndex": 3,
      "spentHeight": 493577
    }
  ],
  [...]
  "height": 458218
  [...]
}
```

(Note that there may be multiple entries in the `vin` array; this transaction happens to have only one.) The `spent` fields in a particular `vout` array element will only be populated if a later transaction that (whose ID is shown) has spent that output.

4.26.3 Additional API commands (RPCs)

Command: `getaddressbalance`

Parameters

1. a JSON object *(required)*

   "addresses" ***(required)*** A JSON list of zero or more taddrs

or

1. A single taddr *(quoted string, required)*

Output

A JSON object

"balance" ***(numeric)*** The total balance in zatoshis of transparent funds held by the given addresses

"received" ***(numeric)*** The total amount in zatoshis of transparent funds received by the given addresses

Description

This RPC returns the total balance and total amount ever received (even if already spent), by the given transparent addresses.

Examples

```
$ zcash-cli getaddressbalance '{"addresses": ["tmYXBYJj1K7vhejSec5osXK2QsGa5MTisUQ",
↪"tmTzyyT7PDiAfhx7V53kvtbnU1SKCv3niiz"]}'
{
  "balance": 58000347656,
  "received": 77000361504
}
$ zcash-cli getaddressbalance '{"addresses": ["tmYXBYJj1K7vhejSec5osXK2QsGa5MTisUQ"]}'
$ zcash-cli getaddressbalance '"tmYXBYJj1K7vhejSec5osXK2QsGa5MTisUQ"'
```

Command: `getaddressdeltas`

Parameters

1. a JSON object *(required)*

   "addresses" ***(required)*** A JSON list of taddrs

   "chainInfo" ***(boolean, optional, default=false)*** Include additional information about the current chain in the results

   "start" ***(numeric, optional, default=0)*** Restrict results to blocks starting at this height

   "end" ***(numeric, optional, default=9999999)*** Restrict results to blocks less than or equal to this height

or

1. A single taddr *(quoted string, required)*

Output

If *chainInfo* is *false*, or *start* or *end* are zero:

A JSON object with the following key/value pairs:

"address" ***(string)*** The taddr that information is being requested for

"blockindex" ***(numeric)*** The zero-based index of the transaction within the block

"height" ***(numeric)*** The height of the block in which the transaction was mined

"index" ***(numeric)*** The offset within the transaction's `vin` or `vout` array

"satoshis" ***(numeric)*** The value (zatoshis) transferred to (if positive) or from (if negative) the taddr

"txid" ***(string)*** The transaction ID

or (if "chainInfo" is true, and "start" and "end" are nonzero):

A JSON object with the following key/value pairs:

"deltas" ***(JSON object)*** **(same as above)**

"address" ***(string)*** The taddr that information is being requested for

"blockindex" ***(numeric)*** The zero-based index of the transaction within the block

"height" ***(numeric)*** The height of the block in which the transaction was mined

"index" *(numeric)* The offset within the transaction's `vin` or `vout` array

"satoshis" *(numeric)* The value (zatoshis) transferred to (if positive) or from (if negative) the taddr

"txid" *(string)* The transaction ID

"start" *(JSON object)*

"hash" *(hex string)* The block hash of the first block in the range

"height" *(numeric)* The height of the first block in the range

"end" *(JSON object)*

"hash" *(hex string)* The block hash of the last block in the range

"height" *(numeric)* The height of the last block in the range

Description

This RPC returns a JSON list in which each entry contains information about a transaction that includes the given transparent address or addresses as either an input or an output. You may restrict the output to transactions contained within a specified range of blocks. Duplicate addresses are ignored.

Examples

```
$ zcash-cli getaddressdeltas '"tmEGycwsvcEhEfR43Dj7w6jYGY6JfqqYsAR"'
[
  {
    "satoshis": 66399972661,
    "txid": "71a7ea74f071f0cad221b79c17e1455f19e0c4cb292385d564a232b75b2f634b",
    "index": 1,
    "blockindex": 1,
    "height": 457166,
    "address": "tmEGycwsvcEhEfR43Dj7w6jYGY6JfqqYsAR"
  },
  {
    "satoshis": -66399972661,
    "txid": "4b7de59de8f9e2e976f9b11e30382bec417be3dd0a66b3b173dfd07c44efde83",
    "index": 0,
    "blockindex": 1,
    "height": 457703,
    "address": "tmEGycwsvcEhEfR43Dj7w6jYGY6JfqqYsAR"
  }
]
$ zcash-cli getaddressdeltas '{"addresses": ["tmTzyyT7PDiAfhx7V53kvtbnU1SKCv3niiz"],
↪"start":492191, "end":492191, "chainInfo":true}'
{
  "deltas": [
    {
      "satoshis": -1000000000,
      "txid": "77e5f1f6326a5e11d80b87e2b29ab61df5d8c2722cbc1113c0d1710d05538c05",
      "index": 1,
      "blockindex": 1,
      "height": 492191,
      "address": "tmTzyyT7PDiAfhx7V53kvtbnU1SKCv3niiz"
    },
    {
      "satoshis": -1000000000,
      "txid": "77e5f1f6326a5e11d80b87e2b29ab61df5d8c2722cbc1113c0d1710d05538c05",
      "index": 2,
      "blockindex": 1,
```

(continues on next page)

(continued from previous page)

```
      "height": 492191,
      "address": "tmTzyyT7PDiAfhx7V53kvtbnU1SKCv3niiz"
    },
  ],
  "start": {
    "hash": "001d6eff5fa2c9f0e1024f185ce9f2787143bfe993ba9e14144d8875cbfe4295",
    "height": 492191
  },
  "end": {
    "hash": "001d6eff5fa2c9f0e1024f185ce9f2787143bfe993ba9e14144d8875cbfe4295",
    "height": 492191
  }
}
```

Command: `getaddresstxids`

Parameters

1. a JSON object *(required)*

   "addresses" ***(required)*** A JSON list of taddr strings

   "start" ***(numeric, optional, default=0)*** Restrict results to blocks starting at this height

   "end" ***(numeric, optional, default=9999999)*** Restrict results to blocks less than or equal to this height

or

1. A single taddr *(quoted string, required)*

Output

A JSON array containing transaction ID strings

Description

This RPC returns a list of transactions IDs associated with a given list of transparent addresses (transactions for which the addresses are either an input or an output). You may restrict the output to transactions contained within a specified range of blocks. Duplicate addresses are ignored. See *getaddressdeltas* for an extended version of this RPC.

Examples

```
$ zcash-cli getaddresstxids '"tmEGycwsvcEhEfR43Dj7w6jYGY6JfqqYsAR"'
[
  "71a7ea74f071f0cad221b79c17e1455f19e0c4cb292385d564a232b75b2f634b",
  "4b7de59de8f9e2e976f9b11e30382bec417be3dd0a66b3b173dfd07c44efde83"
]
$ zcash-cli getaddresstxids '{"addresses": ["tmYXBYJj1K7vhejSec5osXK2QsGa5MTisUQ",
↪"tmTzyyT7PDiAfhx7V53kvtbnU1SKCv3niiz"]}'
$ zcash-cli getaddresstxids '{"addresses": ["tmYXBYJj1K7vhejSec5osXK2QsGa5MTisUQ"],
↪"start": 400000, "end": 500000}'
```

Command: `getaddressutxos`

Parameters

1. a JSON object *(required)*

"addresses" ***(required)*** A JSON list of taddrs

"chainInfo" ***(boolean, optional, default=false)*** Include additional information about the current chain in the results

Output

If `chainInfo` is false:

An array of JSON objects with the following key/value pairs:

"address" ***(string)*** The taddr that information is being requested for

"txid" ***(hex string)*** The transaction ID (hash) of the utxo

"outputIndex" ***(numeric)*** the index into the transaction's `vout` array

"script" ***(hex string)*** The utxo's locking script

"height" ***(numeric)*** The height of the block in which the transaction was mined

"satoshis" ***(numeric)*** The value (zatoshis) that is available to transfer

If `chainInfo` is true:

A JSON object with the following key/value pairs:

"utxos" ***(array of JSON objects)*** As described above (same as result when `chainInfo` is false)

"hash" ***(hex string)*** Current block hash

"height" ***(numeric)*** Current block height

Description

This RPC returns a list of per-transaction JSON objects for transactions that include the given addresses an input or an output. If `chainInfo` is given as true, the current, the hash and height of the best blockchain tip is included in the output.

Examples

```
$ zcash-cli getaddressutxos '"tmYXBYJj1K7vhejSec5osXK2QsGa5MTisUQ"'
[
  {
    "address": "tmYXBYJj1K7vhejSec5osXK2QsGa5MTisUQ",
    "txid": "5d9341bd629fb8decb58e562bc07730c5640a7339f2b0b962e820817201c6df3",
    "outputIndex": 0,
    "script": "76a914fa2e24ff03abfa96945275307d7c8cb3bbbf927588ac",
    "satoshis": 1000010000,
    "height": 481688
  },
  {
    "address": "tmYXBYJj1K7vhejSec5osXK2QsGa5MTisUQ",
    "txid": "15e12955413c777f8f75f27b57ce594e6d3558afa3c7b360d233812b366eff8b",
    "outputIndex": 0,
    "script": "76a914fa2e24ff03abfa96945275307d7c8cb3bbbf927588ac",
    "satoshis": 1000000000,
    "height": 481698
  },
]
$ zcash-cli getaddressutxos '{"addresses": ["tmYXBYJj1K7vhejSec5osXK2QsGa5MTisUQ"],
↪"chainInfo": true}'
{
  "utxos": [
```

(continues on next page)

(continued from previous page)


```
    {
      "address": "tmYXBYJj1K7vhejSec5osXK2QsGa5MTisUQ",
      "txid": "5d9341bd629fb8decb58e562bc07730c5640a7339f2b0b962e820817201c6df3",
      "outputIndex": 0,
      "script": "76a914fa2e24ff03abfa96945275307d7c8cb3bbbf927588ac",
      "satoshis": 1000010000,
      "height": 481688
    },
    {
      "address": "tmYXBYJj1K7vhejSec5osXK2QsGa5MTisUQ",
      "txid": "15e12955413c777f8f75f27b57ce594e6d3558afa3c7b360d233812b366eff8b",
      "outputIndex": 0,
      "script": "76a914fa2e24ff03abfa96945275307d7c8cb3bbbf927588ac",
      "satoshis": 1000000000,
      "height": 481698
    },
  ],
  "hash": "00073cb34978e068e742e27c175688d02da23f341e89373ca68b5e6b5744a847",
  "height": 498989
}
```

Command: `getaddressmempool`

Parameters

1. a JSON object *(required)*

   "addresses" ***(required)*** A JSON list of taddrs

or

1. A single taddr *(quoted string, required)*

Output

A JSON object with the following key/value pairs:

"address" ***(string)*** The taddr that information is being requested for

"index" ***(numeric)*** The offset within the transaction's `vin` or `vout` array

"prevout" ***(numeric)*** Index in the `vout` array of the previous (source) transaction

"prevtxid" ***(string)*** The ID of the previous (source) transaction

"satoshis" ***(numeric)*** The value (zatoshis) transferred to (if positive) or from (if negative) the taddr

"timestamp" ***(string)*** The transaction ID

"txid" ***(string)*** The transaction ID

Note that the two keys beginning with `prev` are only present if this entry describes the use of the address as an output (satoshis is less than zero).

Description

This RPC returns a JSON list in which each entry contains information about an unconfirmed (in the mempool) transaction that includes the given transparent address or addresses as either an input or an output. Duplicate addresses are ignored.

The `getaddressdeltas` RPC returns similar information for confirmed transactions.

Examples

(Note the examples shown here are not reproducible since these transactions have been mined and are no longer in the mempool.)

```
$ zcash-cli getaddressmempool '"tmE4gTU1Qf2ViAKPTBNRJt8BTRdwnRtguLo"'
[
  {
    "address": "tmE4gTU1Qf2ViAKPTBNRJt8BTRdwnRtguLo",
    "txid": "4a27679d4ca1e1f68009a28ae2589c3dac7d3b721533d1199360bfd133102526",
    "index": 0,
    "satoshis": -1000000000,
    "timestamp": 1557867167,
    "prevtxid": "d6267ab2fe5e9a76d252eea8942af70a31df8a3b287001fd6e098dbb3fa8d62f",
    "prevout": 0
  }
]
$ zcash-cli getaddressmempool '{"addresses": ["tmE4gTU1Qf2ViAKPTBNRJt8BTRdwnRtguLo",
↪"tmTzyyT7PDiAfhx7V53kvtbnU1SKCv3niiz"]}'
```

Command: `getspentinfo`

Parameters

1. a JSON object with the following keys:

   "txid" ***(string, required)*** A transaction ID (hash)

   "index" ***(numeric, required)*** The index in the transaction's `vout` array

Output

A JSON object with the following key/value pairs:

"txid" ***(string)*** A transaction ID (hash)

"index" ***(numeric)*** The index in the transaction's `vin` array

"height" ***(numeric)*** The height of the block containing the spending transaction

Description

Given a transaction output specification (transaction ID and `vout` index), this RPC returns a JSON object specifying the later input (transaction ID and `vin` index) that consumed (spent) this output. This RPC will fail if the output hasn't yet been spent.

Examples

```
$ zcash-cli getspentinfo '{"txid":
↪"33990288fb116981260be1de10b8c764f997674545ab14f9240f00346333b780", "index": 4}'
{
  "txid": "b42738de87a3191544fcfca4eed3b326f51b5c1edd2e29920e9846fbeb30ceb9",
  "index": 1,
  "height": 493746
}
```

Command: `getblockhashes`

Parameters

1. high *(timestamp, required)*

Return results from blocks mined before this time

2. low *(timestamp, required)*

   Return results from blocks mined at or after this time

3. A JSON object with the following key/value pairs: *(string, optional)*

   "noOrphans" ***(boolean, optional, default=false)*** Only include blocks on the main chain

   "logicalTimes" ***(boolean, optional, default=false)*** Include the timestamp of each returned block hash

Output

A JSON array containing block hashes *(strings)*

or, if `logicalTimes` is passed as `true`:

A JSON array containing objects with the following keys:

"blockhash" ***(string)*** The hash of a block

"logicalts" ***(numeric)*** The time the block was mined

Description

This method identifies the blocks that were mined between the given timestamps (greater or equal to `low` and less than `high`). The returned entries are sorted by time. If `noOrphans` is set to true, only blocks on the main chain are returned. If `logicalTimes` is set to true, the results include the timestamp of each block.

Examples

(Note: The block beginning with `00127...` is orphaned on testnet, so it's omitted from the result of the second example.)

```
$ zcash-cli getblockhashes 1558141697 1558141576
[
  "000710e39d80861642829a7f46db665398e4e774f62d0086eeb7e47e22dbb27f",
  "0012749b890523094737cf9b08685e80db0b3f56f4188812264cfaa20317f9e4",
  "000e0a2ca48716b77f00890aeda67f5fd6847ea7e7d8cef5e5c1635d6e0bf778"
]
$ zcash-cli getblockhashes 1558141697 1558141576 '{"noOrphans": true}'
[
  "000710e39d80861642829a7f46db665398e4e774f62d0086eeb7e47e22dbb27f",
  "000e0a2ca48716b77f00890aeda67f5fd6847ea7e7d8cef5e5c1635d6e0bf778"
]
$ zcash-cli getblockhashes 1558141697 1558141576 '{"noOrphans": false}'
[
  "000710e39d80861642829a7f46db665398e4e774f62d0086eeb7e47e22dbb27f",
  "0012749b890523094737cf9b08685e80db0b3f56f4188812264cfaa20317f9e4",
  "000e0a2ca48716b77f00890aeda67f5fd6847ea7e7d8cef5e5c1635d6e0bf778"
]
$ zcash-cli getblockhashes 1558141697 1558141576 '{"noOrphans": true, "logicalTimes":␣
↪true}'
[
  {
    "blockhash": "000710e39d80861642829a7f46db665398e4e774f62d0086eeb7e47e22dbb27f",
    "logicalts": 1558141585
  },
  {
    "blockhash": "000e0a2ca48716b77f00890aeda67f5fd6847ea7e7d8cef5e5c1635d6e0bf778",
    "logicalts": 1558141600
```

(continues on next page)

(continued from previous page)

```
  }
]
```

Command: `getblockdeltas`

Parameters

1. A block hash *(string, required)*

Output

A JSON object containing information about the block

Description

Given a block hash, return information about all inputs and outputs of all the transactions within the block, and about the block itself (similar to `getblock`).

Examples

```
$ zcash-cli getblockdeltas '
↪"00227e566682aebd6a7a5b772c96d7a999cadaebeaf1ce96f4191a3aad58b00b"'
{
  "hash": "00227e566682aebd6a7a5b772c96d7a999cadaebeaf1ce96f4191a3aad58b00b",
  "confirmations": 8,
  "size": 1875,
  "height": 497812,
  "version": 4,
  "merkleroot": "3e82b975b005281f6a80c83b0b6378227e25cf5358d4dceddcd9b4616f9c51ba",
  "deltas": [
    {
      "txid": "9fe77b726474cd9b46ec1d0cb23ff11b4a50a97a3af89b8b0fa845d611a9077d",
      "index": 0,
      "inputs": [
      ],
      "outputs": [
        {
          "address": "tmA4rvdJU3HZ4ZUzZSjEUg7wbf1unbDBvGb",
          "satoshis": 1000003848,
          "index": 0
        },
        {
          "address": "t2RpffkzyLRevGM3w9aWdqMX6bd8uuAK3vn",
          "satoshis": 250000000,
          "index": 1
        }
      ]
    },
    {
      "txid": "9dce375054465d5770bbcdbaf1fa6516f23fabfb85f86be460ade0ad177c1731",
      "index": 1,
      "inputs": [
        {
          "address": "tmGygFvgg1B35XeX3oC4e78VSiAyRGcCgME",
          "satoshis": -15083800,
          "index": 0,
          "prevtxid":
↪"2da7f76a2e0a6a6efc22e58d17e58a8d2cef29d119a12d8cb6432f1e4013621a",
```

(continues on next page)

(continued from previous page)


```
          "prevout": 1
        }
      ],
      "outputs": [
        {
          "address": "tmDWk6PT1pE9JbDHqR6an8xJjVj8y8E9xDz",
          "satoshis": 310000,
          "index": 0
        },
        {
          "address": "tmGygFvgg1B35XeX3oC4e78VSiAyRGcCgME",
          "satoshis": 14769952,
          "index": 1
        }
      ]
    }
  ],
  "time": 1558474927,
  "mediantime": 1558474449,
  "nonce": "0000da3417e30c61fa2ce17a1e20e309d7a871e35c4cac1a947f294a91850030",
  "bits": "1f26e516",
  "difficulty": 52.65466254794991,
  "chainwork": "000000000000000000000000000000000000000000000000000192051a8eb",
  "previousblockhash":
↪"0005e0cb410c2a25dd8de2d3e0b92e2ce485bc24bd05147986007514ada8f9a6",
  "nextblockhash": "001f49bd431d5facbfbce72565d151d74aae7859bab88a8fec966017ab7f30e4"
}
```