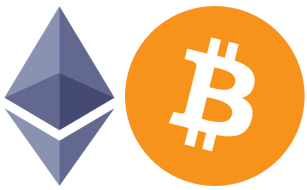
Fourmilab Blockchain Tools
User Guide

by John Walker

Version 1.0
October 2021



Build 806 — 2021-10-20 12:45 UTC

Contents

Chapter 1

User Guide

1.1 Overview

Fourmilab Blockchain Tools provide a variety of utilities for users, experimenters, and researchers working with blockchain-based cryptocurrencies such as Bitcoin and Ethereum. These are divided into two main categories.

1.1.1 Bitcoin and Ethereum Address Tools

These programs assist in generating, analysing, archiving, protecting, and monitoring addresses on the Bitcoin and Ethereum blockchains. They do not require you to run a local node or maintain a copy of the blockchain, and all security-related functions may be performed on an "air-gapped" machine with no connection to the Internet or any other computer.

- Blockchain Address Generator creates address and private key pairs for both the Bitcoin and Ethereum blockchains, supporting a variety of random generators, address types, and output formats.
- Multiple Key Manager allows you to split the secret keys associated with addresses into n multiple parts, from which any $k \leq n$ can be used to reconstruct the original key, allowing a variety of secure custodial strategies.
- Paper Wallet Utilities includes a Paper Wallet Generator which transforms a list of addresses and private keys generated by the Blockchain Address Generator or parts of keys produced by the Multiple Key Manager into a HTML file which may be printed for off-line "cold storage", and a Cold Storage Wallet Validator that provides independent verification of the correctness of off-line copies of addresses and keys.
- Cold Storage Monitor connects to free blockchain query services to allow periodic monitoring of a list of cold storage addresses to detect unauthorised transactions which may indicate that they have been compromised.

1.1.2 Bitcoin Blockchain Analysis Tools

This collection of tools allows various kinds of monitoring and analysis of the Bitcoin blockchain. It does not support Ethereum. These programs are intended for advanced, technically-oriented users who run their own full Bitcoin Core node on a local computer. Note that anybody can run a Bitcoin node as long as they have a computer with the modest CPU and memory capacity required, plus the very large (and inexorably growing) file storage capacity to archive the entire Bitcoin blockchain. You can run a Bitcoin node without being a "miner", or exposing your computer to external accesses from other nodes unless you so wish.

These tools are all read-only monitoring and analysis utilities. They do not generate transactions of any kind, nor do they require unlocked access to the node owner's wallet.

- Address Watch monitors the Bitcoin blockchain and reports any transactions which reference addresses on a "watch list", either deposits to the address or spending of funds from it. The program may also be used to watch activity on the blockchain, reporting statistics on blocks as they are mined and published.
- Confirmation Watch examines blocks as they are mined and reports confirmations for a transaction as they arrive.
- Transaction Fee Watch analyses the transaction fees paid to include transactions in blocks and the reward to miners, producing real-time statistics and log files which may be used to analyse transaction fees over time.

1.2 Blockchain Address Generator

The Blockchain Address Generator, with program name `blockchain_address`, is a stand-alone tool for generating addresses and private keys for both the Bitcoin and Ethereum blockchains. This program does not require access to a Bitcoin node and may be run on an "air gapped" machine without access to the Internet. This permits generating keys and addresses for offline cold storage of funds (for example, in paper wallets kept in secure locations) without the risk of having private keys compromised by spyware installed on the generating machine.

The Address Generator may be run from the command line (including being launched by another program) or interactively, where the user enters commands from the keyboard. The commands used in both modes of operation are identical.

1.2.1 Architecture

The address generator is not a single-purpose utility, but rather more of a toolkit which can be used in a variety of ways to meet your requirements. The program is implemented as a "stack machine", somewhat like the FORTH or PostScript languages. Its stack stores "seeds", which are 256-bit integers represented as 64 hexadecimal digits, "`0`" to "`F`" (when specifying seeds in hexadecimal, upper or lower case letters may be used interchangeably). Specifications on the command line are not options in the usual sense, but rather commands that perform operations on the stack. When in interactive mode, the same commands may be entered from the keyboard, without the leading "`-`", and perform identically.

Here are some sample commands which illustrate operations you can perform.

`blockchain_address -urandom -btc`
: Obtain a seed from the system's fast (non-blocking) entropy source and generate a Bitcoin key/address pair from it, printing the results on the console.

`blockchain_address -repeat 10 -pseudo -format CSV -eth`
: Generate 10 seeds using the program's built-in Mersenne Twister pseudorandom generator (seeded with entropy from the system's fast entropy source), then create Ethereum key/address pairs for each and write as a Comma-Separated Value (CSV) file intended, for example, as offline "paper wallet" cold storage.

`blockchain_address -repeat 16 -hbapik MyApiKey -hotbits -shuffle -repeat 1 -xor -test -btc`

: Request 16 seeds from Fourmilab's HotBits radioactive random number generator (requires Internet connection), shuffle the bytes among the 16 seeds, exclusive-or the two top seeds together, perform a randomness test on the result using Fourmilab's random sequence tester, then use the seed to generate a Bitcoin key/address pair.

1.2.2 Commands

`-aes`
: Encrypt the second item on the stack with the Advanced Encryption Standard, 256 bit key size version, with the key on the top of the stack. The stack data are encrypted in two 128 bit AES blocks in cipher-block chaining mode and the encrypted result is placed on the top of the stack.

`-bindump` ***filename***
: Write the entire stack in binary to the named `filename`. A dump to file may be reloaded onto the stack with the `-binfile` command.

`-binfile` ***filename***
: Read successive 64 byte blocks from the binary file *filename* and place them on the stack, pushing down the stack with each block.

`-btc`
: Use the seed on the top of the stack, which is removed after the command completes, to generate a Bitcoin private key and public address, which are displayed on the console in all of the various formats available. If the `-format` command has been to select CSV output, CSV records are generated using the specified format options. If a `-repeat` value has been set, that number of stack items will be used to generate multiple key/address pairs.

`-clear`
: Remove all items from the stack.

`-drop`
: Remove the top item from the stack.

`-dump`
: Dump the entire stack in hexadecimal to the console or to a file if `-outfile` has been set. A dump to file may be reloaded onto the stack with the `-hexfile` command.

`-dup`
: Duplicate the top item on the stack and push on the stack.

`-eth`
: Generate an Ethereum private key and public address from the seed at the top of the stack, which is removed. The key and address are displayed on the console in human-readable form. If the `-format` command has been to select CSV output, CSV records are generated using the specified format options. If a `-repeat` value has been set, that number of stack items will be used to generate multiple key/address pairs.

`-format` ***fmt***
: Set the format to be used for key/address pairs generated by the `-btc` and `-eth` commands. If the first three letters of *fmt* are "`CSV`" (case-sensitive), a Comma-Separated Value file is generated. Letters following "`CSV`" select options, which vary depending upon the type of address being generated. For Bitcoin addresses, the following options are available.

  - `q` Use uncompressed private key
  - `u` Use uncompressed public address
  - `l` Legacy ("`1`") public address
  - `c` Compatible ("`3`") public address
  - `s` Segwit "`bc1`" public address

  For Ethereum addresses, options are:

  - `n` No checksum on public address
  - `p` Include full public key

For either kind of address, the letter "`k`" indicates that a subsequent key generation command will not remove the keys it processes from the stack. This permits generating the same keys in different formats. The letter "`b`" on either address type causes the private key to be omitted from CSV format output, replaced by a null string. This allows generation of address lists containing only public addresses that may be used with utilities such as `cold_comfort` and `address_watch` without risking compromise of the private keys.

`-hbapik` ***APIkey***
When requesting true random data from Fourmilab's HotBits radioactive random number generator, use the *APIkey* to permit access to the generator. If you don't have an API key (they are free), you may request pseudorandom data based upon a radioactively-generated seed by specifying an API key of "`Pseudorandom`".

`-help`
Print a summary of these commands.

`-hexfile` ***filename***
Load one or more seeds from the named *filename*, which contains data in hexadecimal format. White space in the file (including line breaks) is ignored, and each successive sequence of 64 hexadecimal digits is pushed onto the stack as a 256 bit seed. The `-hexfile` command can load keys dumped to a file with the `-outfile` and `-dump` commands back onto the stack.

`-hotbits`
Retrieve one or more 256 bit seeds from Fourmilab's HotBits radioactive random number generator, using the API key specified by the `-hbapik` command. If the `-repeat` command has specified multiple keys, that number of keys will be retrieved from HotBits and pushed onto the stack.

`-inter`
Enter interactive mode. The user is prompted for commands, which are entered exactly as on the command line, except without the leading hyphen on the command name. To exit interactive mode and return to processing commands from the command line, enter "`end`", "`exit`", "`quit`", or the end of file character.

`-minigen`
Generate a Bitcoin mini private key, display the generated key, and push the full seed for the key onto the stack. Mini private keys were introduced to allow encoding a Bitcoin private key on physical coins, bills, or other objects which lack the space for a full private key, which can be up to 52 characters long. A mini key is just 30 characters, but can represent only a subset of possible Bitcoin addresses and is consequently less secure—they should be used only when absolutely necessary. Due to the nature of mini keys, the generation process differs from that used by the `-btc` command. The `-minigen` command internally generates the seed for the key by mixing the system's fast entropy generator and this program's internal pseudorandom generator seeded by the system fast entropy generator. After finding a suitable key, it pushes the seed onto the stack and displays the corresponding key. You may then use the `-btc` command to generate the corresponding public Bitcoin address in whichever format(s) you wish. If the `-format` is set to "`CSV`", an address file is generated which is compatible with the `btc` command, but with the addition of a fifth field in every record containing the mini key. You may use the `-repeat` command to generate multiple keys and the "`k`" option on the `-format` to keep the seeds on the stack.

`-minikey` *mini_private_key*
Validate and decode the specified mini private key (see above) and, if it is properly formatted, place the seed it encodes on the stack. You may then use the `-btc` command to generate other forms of private keys or public addresses from the seed. Both legacy 22 character and the present standard 30 character mini keys may be specified.

`-mnemonic`
Generate a Bitcoin Improvement Proposal 39 (BIP39) mnemonic phrase from the seed on the top of

the stack. The seed remains on the stack.

`-not`
: Invert the bits of the seed on the top of the stack.

`-outfile` ***filename***
: Output from subsequent `-btc`, `-eth`, and `dump` commands will be written to *filename* instead of standard output. Specifying a *filename* of "`-`" restores output to standard output. Each key generation command overwrites any previous output in *filename*; it is not concatenated. Note that a file written by `-dump` may be loaded back on the stack with the `-hexfile` command.

`-over`
: Duplicate the second item on the stack and push it onto the top of the stack.

`-p`
: Print the top item on the stack on the console.

`-phrase` ***words...***
: Push a key defined by a Bitcoin Improvement Proposal 39 (BIP39) mnemonic phrase on the stack. On the command line, the phrase should be enclosed in quotes.

`-pseudo`
: Push one or more seeds generated by the internal Mersenne Twister pseudorandom generator onto the stack. If the `-repeat` command has been set to greater than one, that number of seeds will be generated and pushed. The pseudorandom generator is itself seeded by entropy supplied by the system's fast entropy source (`/dev/urandom` on most Unix-like systems).

`-pseudoseed`
: Use the number of stack items set by `-repeat` to seed the pseudorandom generator. You may specify up to 78 stack items, representing 624 32-bit seed values. Any more than 78 are not used will be left on the stack. Any previous generator and seed are deleted. This is normally used only for regression testing where repeatable pseudorandom data are required.

`-random`
: Push one or more seeds read from the system's strong entropy source (`/dev/random` on most Unix-like systems) onto the stack. If the `-repeat` command has been set to greater than one, that number of seeds will be generated and pushed. Reading data from a strong source faster than the system can collect hardware entropy may result in delays: the program will wait as long as necessary to obtain the requested number of bytes.

`-repeat` ***n***
: Commands which generate and consume seeds will create and use n seeds instead of the default of 1. To restore the default, specify `repeat 1`.

`-roll` ***n***
: Rotate the top n stack items, moving item n to the top of the stack and pushing other items down.

`-rot`
: Rotate the top three stack items. Item three becomes the top of the stack and the other items are pushed down.

`-rrot`
: Reverse rotate the top three stack items. The seed on the top of the stack becomes the third item and the two items below it move up, with the second becoming the top.

`-seed` ***hex_data***
: The 256 bit seed, specified as 64 hexadecimal digits, is pushed onto the stack. The seed may be preceded by "`0x`", but this is not required.

`-sha2`
: The seed on the top of the stack is replaced by the hash (digest) generated by the Secure Hash Algorithm 2 (SHA-2), 256 bit version (SHA2-256). If `-repeat` has been set greater than one, the specified number of seeds will be removed from the stack and concatenated, top down, the digest computed, and placed back on the stack.

`-sha3`
: The seed on the top of the stack is replaced by the hash (digest) generated by the Secure Hash Algorithm 3 (SHA-3), 256 bit version (SHA3-256). If `-repeat` has been set greater than one, the specified number of seeds will be removed from the stack and concatenated, top down, the digest computed, and placed back on the stack.

`-shuffle`
: Shuffle bytes of items on the stack using pseudorandom values generated as for the `-pseudo` command. Shuffling bytes can mitigate the risk of interception of seeds generated remotely and transmitted across the Internet. (Secure `https:` connections are used for all such requests, but you never know….) The number of items shuffled is set by `-repeat`.

`-swap`
: Exchange the top two items on the stack.

`-test`
: Use the Fourmilab `ent` random sequence tester to evaluate the apparent randomness of the top items on the stack. The number of items tested may be set with `-repeat`. You must have `ent` installed on your system to use this command. Randomness is evaluated at the bit stream level.

`-testall`
: Use the Fourmilab `ent` random sequence tester to evaluate the apparent randomness of the entire contents of the stack. You must have `ent` installed on your system to use this command. Randomness is evaluated at the bit stream level.

`-testmode` ***n***
: Set developer test modes to the bit-coded value *n*, which is the sum of the mode bits to enable. These are intended for development and regression testing and should not be enabled for production use, leaving the setting at the default of 0. The 1 bit makes the `-minigen` produce deterministic output from a fixed `-pseudoseed`. The 2 bit causes `blockchain_address` to list all of the Perl library modules it has used during its execution.

`-type` ***Any text***
: Display the text on the console. This is often used in command files to inform the user of what's going on.

`-urandom`
: Push one or more seeds read from the system's fast entropy source (`/dev/urandom` on most Unix-like systems) onto the stack. If the `-repeat` command has been set to greater than one, that number of seeds will be generated and pushed. The fast generator has no limitation on generation rate, so you may request any amount of data without possibility of delay.

`-wif` ***private_key***
: Push the seed represented by the Bitcoin Wallet Import Format (WIF) key onto the stack.

`-xor`
: Perform a bitwise exclusive or of the top two items on the stack and push the result on the stack.

`-zero`
: Push an all zero seed on the stack.

1.3 Multiple Key Manager

The Multiple Key Manager (`multi_key`) splits the private keys used to access funds stored in Bitcoin or Ethereum addresses into multiple independent parts, allowing them to be distributed among a number of custodians or storage locations. The original keys may subsequently be reconstructed from a minimum specified number of parts. Each secret key is split into n parts ($n \geq 2$), of which any $k, 2 \leq k \leq n$ are sufficient to reconstruct the entire original key, but from which the key cannot be computed from fewer than k parts. In the discussion below, we refer to n as the number of `parts` and k as the number `needed`. The splitting and reconstruction of keys is performed using the Shamir Secret Sharing technique.

The ability to split secret keys into parts allows implementing a wide variety of custodial arrangements. For example, a company treasury's cold storage vault might have secret keys split five ways, with copies entrusted to the chief executive officer, chief financial officer, an inside director, an outside director, and one kept in a safe at the office of the company's legal firm. If the parts were generated so that any three would re-generate the secret keys, then at least three people would have to approve access to the funds stored in the vault, which reduces the likelihood of their misappropriation. The existence of more parts than required guards against loss or theft of one of the parts: should that happen, three of the remaining copies can be used to withdraw the funds and transfer them to new accounts protected by new multi-part keys.

To create multiple keys, start with a comma-separated value (CSV) file in the format created by `blockchain_address` with "`format CSV`" selected. Let's call this file `keyfile.csv`. Now, to split the keys in this file into five parts, any three of which are sufficient to reconstruct the original keys, use the command:

```
multi_key -parts 5 -needed 3 keyfile.csv
```

This will generate five split key files named `keyfile-1.csv`, `keyfile-2.csv`, … `keyfile-5.csv`. These are the files which are distributed to the five custodians. After verifying independently that the parts can be successfully reconstructed (you can't be too careful!), the original `keyfile.csv` is destroyed, leaving no copy of the complete keys. (All of this should, of course, be done on an "air gapped" machine not connected to any network or external device which might compromise the complete keys while they exist.)

When access to the keys is required, any three of the five parts should be provided by their holders and combined with a command like:

```
multi_key -join keyfile-4.csv keyfile-1.csv keyfile-2.csv
```

Again, you can use any three parts and specify them in any order. This will create a file named `keyfile-merged.csv` containing the original keys in the same format as was created by `blockchain_address`. You can then use this file with any of the other utilities in this collection or use one or more of the secret keys to "sweep" the funds into a new address. To maximise security, once a set of keys has been recombined, funds should be removed from all and those not used transferred to new cold storage addresses, broken into parts as you wish. In many cases, it makes sense to split individual keys rather than a collection of many so you need only join the ones you immediately intend to use.

Once the parts have been generated on the air-gapped machine, they are usually written to offline paper storage (using the `paper_wallet` program, for example, which works with split key files as well as complete key files) or archival media such as write-once optical discs, perhaps with several identical redundant copies per part. Their custodians should store the copies of their parts in multiple secure, private locations to protect against mishaps that might destroy all copies of their part.

The ability to create multiple parts allows flexibility in their distribution. You might, for example, entrust two parts to the company CEO, who would only need one part from another officer or director to access the vault, while requiring three people other than the CEO to access it.

Although primarily intended to split blockchain secret keys into parts, `multi_key` may be used to protect and control access to any kind of secret which can be expressed as 1024 or fewer text characters: for example, passwords on root signing certificates, decryption keys for private client information, or the formula for fizzy soft drinks.

1.3.1 Command line options

`-help`
: Print how to call information.

`-join`
: Reconstruct the original private keys from the parts included in the files specified on the command line. You must supply at least the `-needed` number of parts when they were created (if you specify more, the extras are ignored). The output is written to a file with the specified `-name` or, if none is given, that of the first part with its number replaced with "`-merged`". The file will be in the comma-separated value (CSV) format in which `blockchain_address` writes addresses and keys it generates and is used by other programs in this collection.

`-name` ***name***
: When splitting keys, the individual part files will be named "*name*-*n*`.csv`", where *n* is the part number. If no `-name` is specified, the name of the first key file supplied will be used.

`-needed` ***k***
: When reconstructing the original keys, at least *k* parts (default 3) must be specified. This option is ignored when joining the parts.

`-parts` ***n***
: Keys will be split into *n* parts (default 3). This option is ignored when joining parts.

`-prime` ***p***
: Use the prime number *p* when splitting parts. This should only be specified if you're a super expert who has read the code, understands the algorithm, and knows what you're doing, otherwise you're likely to mess things up. The default is 257.

1.4 Paper Wallet Utilities

The safest way to store cryptocurrency assets not needed for transactions in the near term is in "cold storage": kept offline either on a secure (and redundant) digital medium or, safest of all, paper (again, replicated and stored in multiple secure locations). A cold storage wallet consists simply of a list of one or more pairs of blockchain public addresses and private keys. Funds are sent to the public address and the corresponding private key is never used until the funds are needed and they are "swept" into an online wallet by entering the public key.

The `blockchain_address` program makes it easy to generate address and key pairs for offline cold storage, encoding them as comma-separated value (CSV) files which can easily be read by programs. For storage on paper, a more legible human-oriented format is preferable, which the utilities in this chapter aid in creating and verifying.

1.4.1 Paper Wallet Generator

The `paper_wallet` program reads a list of Bitcoin or Ethereum public address and private key pairs, generated by the `blockchain_address` program in comma-separated value (CSV) format, and creates an HTML file which can be loaded into a browser and then printed locally to create paper cold storage wallets. In the interest of security, this process, as with generation of the CSV file, should be done on a machine with no connection to the Internet ("air gapped"), and copies of the files deleted from its storage before the machine is connected to a public network.

1.4.1.1 Creating a paper wallet

Assume you've created a cold storage wallet with twenty Ethereum addresses using the `blockchain_address` program, for example with the command:

```
blockchain_address -repeat 20 -urandom -outfile coldstore.csv -format CSV -eth
```

This should be done on the same air-gapped machine on which you'll now create the paper wallet. Be careful to generate the `coldstore.csv` file in a location you'll erase before connecting the machine to a public network. If you wish to keep a machine-readable cold storage wallet, copy the `coldstore.csv` file to multiple removable media (for example, flash storage devices [perhaps encrypted], writeable compact discs, etc.) Be aware that no digital storage medium has unlimited data retention life, and even if the data are physically present, it may be difficult to near-impossible to find a drive which can read it in the not-so-distant future. By contrast, we have millennia of experience with ink on paper, and if protected from physical damage, a printed cold storage wallet will remain legible for centuries.

Now let's create a paper wallet. Using the `coldstore.csv` file we've just generated and the default parameters, this can be done with:

```
paper_wallet coldstore.csv >coldstore.html
```

You can now load the `coldstore.html` file into a Web browser with a `file:coldstore.html` URL, use print preview to verify it is properly formatted, then print as many copies as you require for safe storage to a local printer. Even though you're using a Web browser to load and print the file, security is not compromised as long as the computer running it is not connected to the Internet. After printing the paper wallet, be sure to clear the browser's cache, deleting any copy it may have made of the file.

1.4.1.2 Command line options

`-date` ***text***
: The specified *text* will be used as the date in the printed wallet. Any text may be used: enclose it in quotes if it contains spaces or special characters interpreted by the shell. If no `-date` is specified, the current date is used, in ISO-8601 `YYYY-MM-DD` format.

`-font` ***fname***
: Use HTML/CSS font name *fname* to display addresses and keys. The default is `monospace`.

`-help`
: Print a summary of the command line options.

`-offset` ***n***
: The integer *n* will be added to the address numbers (first CSV field) in the input file. If you've generated a number of cold storage wallets with the same numbers and wish to distinguish them in the printed versions, this allows doing so.

`-perpage` ***n***
: Addresses will be printed *n* per page. The default is 10 addresses per page. The number which will fit on a page depends upon your paper size, font selection, and margins used when printing—experiment with print preview to choose suitable settings.

`-prefix` ***text***
: Use *text* as a prefix for address numbers from the CSV file (optionally adjusted by the `-offset` option). This allows further distinguishing addresses in the printed document.

`-separator` ***text***
: Display addresses and private keys as groups of four letters and numbers separated by the sequence *text*, which may be an HTML text entity such as "`&ndash;`".

`-size` ***sspec***
: Use HTML/CSS font size *sspec* to display addresses and keys. The default is `medium`.

`-title` ***text***
: Use the specified `text` as the title for the cold storage wallet. If no title is specified, "Bitcoin Wallet" or "Ethereum Wallet" will be used, depending upon the type of address in the CSV file.

`-weight` ***wgt***
Use HTML/CSS font weight *wgt* to display addresses and keys. The default is `normal`.

1.4.2 Cold Storage Wallet Validator

When placing funds in offline cold storage wallets, an abundance of caution is the prudent approach. By their very nature, once funds are sent to the public address of a cold storage wallet, that address is never used again, nor is its private key ever used at all until the time comes, perhaps years or decades later, to "sweep" the funds from cold storage back into an online wallet. Consequently, if, for whatever reason, there should be an error in which the private key in the offline wallet does not correspond to the public address to which the funds were sent, those funds will be irretrievably lost, with no hope whatsoever of recovery. Entering the private key into a machine connected to the Internet in order to verify it would defeat the entire purpose of a cold storage wallet: that its private keys, once generated on an air-gapped machine, are never used prior to returning the funds from cold storage.

While the circumstances in which a bad address/key pair might be generated and stored may seem remote, the consequences of this happening, whether due to software or hardware errors, incorrect operation of the utilities used to generate them, or malice, are so dire that a completely independent way to verify their correctness is valuable.

The `validate_wallet` program performs this validation on cold storage wallets, either in the CSV format generated by `blockchain_address` or the printable HTML produced by `paper_wallet`. Further verification that the printed output from the HTML corresponds to the file which was printed will require manual inspection or scanning and subsequent verification. The `validate_wallet` program is a "clean room" re-implementation of the blockchain address generation process used by `blockchain_address` to create cold storage wallets. It is written in a completely different programming language (Python version 3 as opposed to Perl), and uses the Python cryptographic libraries instead of Perl's. While it is possible that errors in lower-level system libraries shared by both programming languages might corrupt the results, this is much less likely than an error in the primary code or the language-specific libraries they use.

1.5 Cold Storage Monitor

For safety, cryptocurrency balances which are not needed for active transactions are often kept in "cold storage", either off-line in redundant digital media not accessible over a network or printed on paper (for example, produced with the `paper_wallet` program) kept in multiple separate locations. Once sent to these cold storage addresses, there should be no further transactions whatsoever referencing them until they are "swept" back into an active account for use.

But under the principle of *doveryay, no proveryay* (trust, but verify), a prudent custodian should monitor cold storage addresses to confirm they remain intact and have not been plundered by any means. (It's usually an inside job, but you never know.) One option is to run a "hot monitor" that constantly watches transactions on the blockchain such as the `address_watch` utility included here, but that requires you to operate a full Bitcoin node and does not, at present, support monitoring of Ethereum addresses.

The `cold_comfort` utility provides a less intensive form of monitoring which works for both Bitcoin and Ethereum cold storage addresses, does not require access to a local node, but instead uses free query services that return the current balance for addresses. You can run this job periodically (once a week is generally sufficient) with a list of your cold storage addresses, producing a report of any discrepancies between their expected balances and those returned by the query.

Multiple query servers are supported for both Bitcoin and Ethereum addresses, which may be selected by command line options, and automatic recovery from transient errors while querying servers is provided.

1.5.1 Watching cold storage addresses

The list of cold storage addresses to be watched is specified in a CSV file in the same format produced by `blockchain_address` and read by `paper_wallet`, plus an extra field giving the expected balance in the cold storage address. For example, an Ethereum address in which a balance of 10.25 Ether has been deposited might be specified as:

```
1,"0x1F77Ea4C2d49fB89a72A5F690fc80deFbb712021","",10.25
```

The private key field is not used by the `cold_comfort` program and should, in the interest of security, be replaced by a blank field as has been done here. There is no reason to expose the private keys of cold storage addresses on a machine intended only to monitor them. You can use the "`b`" and "`k`" options on a `-format CSV` command to generate a copy of the addresses without the private keys. To query all addresses specified in a file named `coldstore.csv` and report the current and expected balances, noting any discrepancies, use:

```
cold_comfort -verbose coldstore.csv
```

If you don't specify `-verbose`, only addresses whose balance differs from that specified in the CSV file will be reported.

1.5.2 Command line options

The `cold_comfort` program is configured by the following command line options.

`-btcsource` ***sitename***
: Specify the site queried to obtain the balance of Bitcoin addresses. The sites supported are:

  - `blockchain.info`
  - `blockcypher.com`
  - `btc.com`

  You must specify the site name exactly as given above.

`-dust` ***n***
: Some miscreants use the blockchain as a means of "spamming" users, generally to promote some shady, scammy scheme. They do this by sending tiny amounts of currency to a large number of accounts, whose holders they hope will be curious and investigate the transaction that sent them, in which the spam message is embedded, usually as bogus addresses. You might think getting paid to receive spam is kind of cool, but the amounts sent are smaller than the transaction cost it would take to spend or aggregate them with other balances. This is an irritation to cold storage managers, who may find their inactive accounts occasionally receiving these tiny payments, which in blockchain argot are called "dust". This option sets the threshold n (default 0.001) below which reported balances in excess of that expected will be ignored and not considered discrepancies. If `-verbose` is specified, they will be flagged in the report as "Dust".

`-ethsource` ***sitename***
: Specify the site queried to obtain the balance of Ethereum addresses. The sites supported are:

  - `blockchain.com`
  - `etherscan.io`
  - `ethplorer.io`

  You must specify the site name exactly as given above.

`-help`
: Print a summary of the command line options.

`-loop`
: Loop forever querying addresses. After each pass through all the addresses, a pause of `-waitloop` seconds will occur.

`-retry` ***n***
: If a query fails, retry it *n* times before abandoning the request and reporting the failure (default 3).

`-shuffle`
: Shuffle the order in which addresses are queried before each pass checking them. This may (or may not) make it less obvious they represent a single cold storage vault.

`-sort`
: When `-shuffle` is specified, sort the results from queries back into the order the addresses were specified in the files on the command line.

`-verbose`
: Report all addresses, even if an address's current balance is the same as expected. Transient query failures and retries are also reported.

`-waitconst` ***n***
: Wait *n* seconds (default 17) between queries for address balances. This avoids overloading the sites providing this free service and getting banned for abusing them.

`-waitloop` ***n***
: When using the `-loop` option, pause for *n* seconds (default 3600) after completing queries for all the addresses in the list before commencing the next pass.

`-waitrand` ***n***
: Add a random number between 0 and *n* seconds (default 20) to the constant set by `waitconst` between individual queries. This further reduces the load on the query sites and makes it less obvious they're coming from an automated process.

1.6 Address Watch

The `address_watch` program monitors the Bitcoin blockchain, watching for transactions which involve one or more watched Bitcoin addresses, specified on the command line, in a file listing addresses to watch, or from the addresses in a Bitcoin Core wallet. Address Watch can be used by those who keep Bitcoin reserves in "cold storage", on paper or offline devices for security, alerting them if one of these addresses is used in a transaction, indicating its security has been compromised. The program can also display statistics of blocks added to the blockchain and write a log that can be used for analysis of the blockchain's behaviour. This program requires access to a Bitcoin node with a full copy of the blockchain, configured with transaction indexing ("`txindex=1`").

1.6.1 Command line options

Address Watch is configured by the following command line options. In addition to the options listed here, an additional set of options, common to other programs in the collection, specifies how the program communicates with the Bitcoin Core Application Programming Interface (API): see "RPC API configuration" for details.

`-bfile` ***filename***
: Specifies a file used to save the most recent block examined by the program. When the program starts, it begins scanning at the next block. As each block is processed, the block file is updated so a subsequent run of the program will start at the next block.

`-end` ***n***
: Stop scanning and exit after processing block *n*. If no `-end` is specified, `address_watch` will continue

scanning for newly-published blocks at the specified `-poll` interval.

`-help`
: Print a summary of the command line options.

`-lfile` ***filename***
: For each transaction involving a watched address, append an entry to a log file containing fields in Comma Separated Value (CSV) format as described in "Watched address log file" below.

`-poll` ***time***
: After reaching the current end of the blockchain, check for newly-published blocks after the specified *time* in seconds. If *time* is set to zero, `address_watch` will exit after scanning the last block.

`-sfile` ***filename***
: As each block is processed, append an entry describing it to the statistics file *filename*. Records are written in Comma Separated Value (CSV) format as described in "Block statistics log file" below.

`-start` ***n***
: Start scanning the blockchain at block *n*. If no `-start` is specified, scanning will begin with the next block after that specified in the `-bfile` file or with the next block published.

`-stats`
: For each block processed, print statistics about its content on the console. The statistics are the same as written to a file by the `-sfile` option, but formatted in a primate-readable format.

`-type` ***Any text***
: Print the text on the console.

`-verbose`
: Print detailed information about the contents of blocks. The more times you specify `-verbose`, the more output you'll get.

`-wallet`
: Include addresses in the Bitcoin Core wallet with unspent balances in those watched for transactions. Since every spend transaction in Bitcoin Core completely spends the source address and places unspent funds in a new change address, the option will automatically track these newly-generated addresses as they appear and are used. The list of wallet addresses is updated before scanning each new block that arrives.

`-watch` [***label,***] ***address***
: Add the specified Bitcoin *address* to the watch list. You can specify a label before the address, separated by a comma, for example: `"Money Bin,1ScroogeYebEqDTbdjk36WzLxjCZTkNe3w"`.

`-wfile` ***filename***
: Add addresses read from the specified *filename* in Comma Separated Value (CSV) format to the watch list. Each line in the file specifies an address as: *Label*, *Bitcoin address*, *Private key*, *Balance*. The *Label* is an optional human-readable name for the address, and the *Private key* and *Balance* fields are not used by this program.

1.6.2 Log file formats

The `address_watch` program can write two log files, both in Comma Separated Value format, with fields as follows. New items are appended to an existing log file.

1.6.3 Watched address log file

The `-lfile` option enables logging of transactions involving watched addresses. Each log item is as follows.

1. Address label from wallet
2. Bitcoin address
3. Value (negative if spent, positive if received)
4. Date and time (ISO 8601 format)
5. Block number
6. Transaction ID
7. Block hash

1.6.4 Block statistics log file

The `-sfile` option logs statistics for blocks as they are added to the blockchain, with records containing the following fields.

1. Block number
2. Date and time (Unix `time()` format)
3. Number of transactions in block
4. Smallest transaction (bytes)
5. Largest transaction (bytes)
6. Mean transaction size (bytes)
7. Transaction size standard deviation
8. Total size of transactions (bytes)
9. Smallest transaction value (BTC)
10. Largest transaction value (BTC)
11. Mean transaction value (BTC)
12. Transaction value standard deviation
13. Total transaction value (BTC)
14. Total miner reward for block (including transaction fees)
15. Base miner reward for block (less transaction fees)

1.7 Confirmation Watch

When a Bitcoin transaction is posted to the network, it first is placed in the "mempool" by nodes which receive it. Miner nodes choose transactions from the mempool, usually based upon the transaction fee per byte they offer, validate them against their local copy of the entire Bitcoin blockchain and, if and when they find a hash for a candidate block that meets the present difficulty requirement, publish the block to the blockchain and notify other nodes of its publication. Other nodes independently validate the transactions it contains and add their confirmations to the transaction, which are recorded on the blockchain. By convention, a transaction is deemed fully confirmed once six or more independent confirmations for it are recorded on the blockchain. Most Bitcoin wallet programs will not spend funds received (even "change" from funds in your own wallet which have been partially spent) until at least six confirmations are received for its transfer to your wallet.

The `confirmation_watch` utility monitors a transaction on the blockchain and reports confirmations as they arrive. It can be used to monitor pending transactions and report when a specified number of confirmations are received. Depending upon the configuration, you can run `confirmation_watch` with the following command lines.

`confirmation_watch` ***transaction_id block_hash***
: This form of command may always be used, regardless of configuration. It specifies the hexadecimal transaction ID and hash of the block which contains it. Both of these can be found in the console output and log file generated by `address_watch`.

`confirmation_watch` ***transaction_id***
: If your Bitcoin Core node has been configured with "`txindex=1`", which maintains an index of transactions, you can specify just the *transaction_id*, with the block hash found from the transaction index.

`confirmation_watch` ***address/label***
: If you have specified a log file maintained by `address_watch` on the command line with the `-lfile` option, you may specify just the Bitcoin public address to which the transaction pertains or the label you have assigned to it in the Bitcoin Core wallet. The most recent transaction involving that address will be retrieved from the log file and monitored for confirmations.

1.7.1 Command line options

Confirmation Watch is configured by the following command line options. In addition to the options listed here, an additional set of options, common to other programs in the collection, specify how the program communicates with the Bitcoin Core Application Programming Interface (API): see "RPC API configuration" for details.

`-confirmed` ***n***
: Specifies the number of confirmations which must be received before a transaction is deemed confirmed. If a transaction is being monitored by the `-watch` option, `confirmation_watch` will exit after this number of confirmations have arrived.

`-help`
: Print a summary of how to call and command line options.

`-lfile` ***filename***
: Use the log file written by the `address_watch` program to locate transactions for a Bitcoin address specified either by its public address or a label given to it in the Bitcoin Core wallet. If this option is not specified, transactions must be identified by their transaction ID.

`-testmode`
: Instead of taking the transaction to be watched from the command line or indirectly from the `address_watch` log file, choose a transaction from the most recently mined block and watch its confirmations. This allows developers to test the program on a representative transaction without the need to to submit one or manually find one in a block dump.

`-type` ***Any text***
: Print the text on the console.

`-verbose` ***n***
: Print detailed information about transactions and confirmations. The more times you specify `-verbose`, the more information you'll see.

`-watch`
: Poll for new confirmations every `-poll` seconds until the `-confirmed` number have arrived.

1.8 Transaction Fee Watch

Bitcoin transactions submitted for inclusion in the blockchain are accompanied by a transaction fee paid to the miner who includes the transaction in a block published to the blockchain. Transactions can be selected by miners at their discretion, but in most cases will be chosen to maximise the reward for including them in a block, which usually means those which offer the highest transaction fee per byte (or, more precisely, "virtual byte") of the transaction. Whenever a block is added to the blockchain, Bitcoin Core computes statistics of the fees for transactions within it. In addition, Bitcoin Core computes an "estimated smart fee" as a suggestion to those submitting transactions at the current time.

The `fee_watch` program monitors the blockchain and reports the fee statistics for each block published and fee recommendations from Bitcoin Core, optionally writing both of these to a log file for analysis by other programs. The program is configured by the following command line options.

1.8.1 Command line options

Fee Watch is configured by the following command line options. In addition to the options listed here, an additional set of options, common to other programs in the collection, specify how the program communicates with the Bitcoin Core Application Programming Interface (API): see "RPC API configuration" for details.

`-confirmed` ***n***
: Specifies the number of confirmations which must be received before a transaction is deemed confirmed. This is used when requesting an estimate of the current transaction fee with the Bitcoin Core API call `estimatesmartfee` to indicate the priority of the transaction. The default, 6, corresponds to standard priority for this call.

`-ffile` ***filename***
: Write a log file of fee information collected by `fee_watch`. The log is written in Comma Separated Value (CSV) format, and contains two kinds of records, distinguished by a digit in the first field. See "Log file format" below for details.

`-help`
: Print a summary of how to call and command line options.

`-poll` ***time***
: Query and report transaction fee estimates and statistics every *time* seconds, by default 300 seconds (five minutes).

`-quiet`
: Suppress console output for periodic transaction fee polls. Use this option when writing a log file with the `-ffile` option if you don't want to also see information as it is collected.

`-type` ***Any text***
: Print the text on the console.

`-verbose` ***n***
: Print detailed information about operations. The more times you specify `-verbose`, the more information you'll see.

1.8.2 Log file format

When the `-ffile` option is specified, `fee_watch` writes a log file recording the transaction fee information it collects. This file is written in Comma Separated Value (CSV) format, and consists of two types of records, as follows.

1.8.2.1 Estimated fee record

These records report the estimated fee, according to the Bitcoin Core `estimatesmartfee` API call, at the indicated time. The estimated transaction fee in the record is expressed in BTC per virtual kilobyte of transaction size, where virtual transaction size is as defined in Bitcoin Improvement Proposal 141 section "Transaction size calculations". One record of this type is generated for every `-poll` interval.

1. Record type, `1`
2. Date and time (Unix `time()` format)
3. Date and time (ISO 8601 format)
4. Estimated transaction fee, BTC per virtual kilobyte

1.8.2.2 Block fee statistics record

If any blocks have been added to the blockchain since the last `-poll` interval, a record will be written, reporting fee statistics for transactions in the block. Note that the time in these records is the time the block was added to the blockchain, not the time of the `fee_watch` poll. The values reported in these records are those returned by the `getblockstats` API call for the block, with fees reported in units of satoshis (BTC 0.00000001) per virtual byte of transaction, where virtual bytes are as defined for the Estimated fee record above.

1. Record type, `2`
2. Block date and time (Unix `time()` format)
3. Block date and time (ISO 8601 format)
4. Block number
5. Minimum fee rate
6. Mean (average) fee rate
7. Maximum fee rate
8. 10th percentile fee rate
9. 25th percentile fee rate
10. 50th percentile fee rate
11. 75th percentile fee rate
12. 90th percentile fee rate

1.9 RPC API configuration

The `address_watch`, `confirmation_watch`, and `fee_watch` programs all require access to the Application Programming Interface (API) provided by a Bitcoin Core node. Access to this interface can be via three mechanisms:

`local` Access to a Bitcoin Core node running on the same machine via the `bitcoin-cli` command line program.

`rpc` Access to a Bitcoin Core node via its Remote Procedure Call (RPC) interface. The node may either be on the same machine or on a different machine configured to accept requests from the host submitting them.

`ssh` Access a remote Bitcoin Core node by submitting commands to its `bitcoin-cli` utility via the Secure Shell (SSH) facility. The client and node machines must be configured to permit password-less access via public key authentication.

The following options, common to all of these programs, allow you to configure access to the API. These options may be set on the command line or via a configuration file common to all of the programs.

`-clipath` ***path***
Specify the *path* used to invoke the `bitcoin-cli` program on the node machine. This option is used for the `local` and `ssh` access methods. Note that on an SSH login, the user's terminal login scripts are not executed, so you may have to specify an explicit path even if `bitcoin-cli` is in a directory included in the `PATH` declared by those scripts.

`-host` ***hostname***
Specifies the host (machine network name) on which Bitcoin Core is running. If this is the same computer, use `localhost`, otherwise specify the local machine name, fully qualified domain name, or IP address of the machine.

`-method` ***which***
Sets the method used to access the API. Use `local` if accessing a Bitcoin Core node on the same

machine, or `ssh` to access a Bitcoin Core node on another machine. The `rpc` option selects direct access via the RPC interface on the same or a different host. RPC access is the most efficient and should be used if available.

`-rpccpass` ***password***
: Set the password for access via the `rpc` method. This password is configured in the `bitcoin.conf` file via the `rpcpassword` statement. If the *password* specified is the null string (`""`), the user will be prompted to enter the password from the console, which is far more secure than specifying it on the command line.

`-port` ***number***
: Sets the port used to communicate with the Bitcoin Core node when the `rpc` method is selected. The default is 8332.

`-user` ***userid***
: Sets the User ID (login name) for access to a Bitcoin Core node on another machine via the `ssh` method.

1.10 Installation

Fourmilab Blockchain Tools are written in the Perl and Python programming languages, which are pre-installed on most modern versions of Unix-like operating systems such as Linux, FreeBSD, and Macintosh OS X, and available for many other systems. Consequently, you can run any of the pre-built versions of the tools, all of which have file types of "`.pl`" or "`.py` by simply invoking them with the `perl` or `python3` commands. The programs use a number of modules, some of which are "core" or "standard" (included as part of current language distributions), and others which may have to be installed either from the operating system's software library or the Comprehensive Perl Archive Network and its search engine, MetaCPAN or with the `pip3` utility for Python. If a module is available from your operating system's distribution library, that's generally the best way to install it, since it will be automatically updated by the system's software update mechanism.

1.10.1 Required Perl modules

Here is a list of all Perl modules used by the programs. Not all programs use all modules: if you're only interested in some of the programs, you need only install those they require. Modules marked as "*core*" will be pre-installed on most modern versions of Perl.

- `Bitcoin::BIP39`
- `Bitcoin::Crypto::Key::Private`
- `Bitcoin::Crypto::Key::Public`
- `Crypt::CBC`
- `Crypt::Digest::Keccak256`
- `Crypt::OpenSSL::AES`
- `Crypt::Random::Seed`
- `Crypt::SSSS`
- `Data::Dumper` *core*
- `Digest::SHA` *core*
- `Digest::SHA3`
- `Getopt::Long` *core*
- `JSON` *core*
- `List::Util` *core*
- `LWP::Protocol::https`

- `LWP::Simple`
- `LWP`
- `MIME::Base64` *core*
- `Math::Random::MT`
- `POSIX` *core*
- `Statistics::Descriptive`
- `Term::ReadKey`
- `Text::CSV`

1.10.2 Required Python modules

To avoid commonality in language and libraries in the interest of avoiding single points of failure when validating the correctness of generated wallets, the `validate_wallet` program is written in the Python language (version 3 or greater), and requires the following modules be installed on systems that run it. Modules marked "*standard*" are part of Python's standard libraries and should be installed on most systems that support the language. If you don't run `validate_wallet`, you needn't bother installing these modules.

- `base58`
- `binascii` *standard*
- `coincurve`
- `cryptos`
- `fileinput` *standard*
- `re` *standard*
- `sha3`
- `sys` *standard*

1.10.3 Building from original source code

This software, including all programs, support files, utilities, and documentation was developed using the Literate Programming methodology, where the goal is that programs should be as readable to humans as they are by computers. The package is written using the **nuweb** literate programming system, which is language-agnostic: it can be used to develop software in any programming language, including multiple languages in a single project, as is the case for this one. The **nuweb** tools are free software written in portable C, with source code downloadable from the link above.

Programs in **nuweb** are called "Web files", which have nothing whatsoever to do with the World-Wide Web (which it predates), having a file type of "`.w`". All of the other files in the distribution are generated automatically from the master Web. If you wish to modify one or more of the programs, it's best to modify the master code in the Web file and re-generate the programs from it. All of the building and maintenance operations are performed by a `Makefile` which is, itself, generated from the Web. If you edit any of the files associated with this program, be sure to use a text editor which supports the Unicode-compatible UTF-8 character set: otherwise some special characters may be turned into gibberish.

Documentation is generated automatically in the LaTeX document preparation language, with the final PDF documents produced with XeTeX, a version of TeX extended to support the full Unicode character set. These utilities can be installed from the distribution archives of most Unix-like systems.

1.10.4 Configuration parameters

When you build from source code, a number of build-time configuration parameters are incorporated from the Web file `configuration.w`. Please see the documentation for that file in the source code listing (in the Introduction chapter, section "Configuration"). Most of the configuration parameters set defaults which can

be overridden by command-line options, so setting them is normally a convenience to avoid having to specify the options you prefer, not a necessity.

1.10.5 Build procedure

Once you have installed all of the required utilities (**nuweb**, XeTeX, Perl, Python, and the modules required), you can build the programs by entering the top level directory of the distribution (the one which contains the `blockchain_tools.w` file) and entering the following commands. (I've added comments to the commands to explain what they do—you need not enter them.)

```
make dist       # Build all programs and documents
make regress    # Run regression test
```

It is not unusual to see a few differences in the balances reported for some of the addresses in the regression test output: the blockchains never sleep and balances sometimes change. If that's the only discrepancy reported in the regression test, you can run "`make regress_update`" to incorporate the changes in the expected output of the regression test.

After the build process, the ready-to-run Perl and Python programs will be in the `bin` subdirectory while User Guide and program listing PDF files will be in the `doc` subdirectory. You can, if you wish, re-generate the distribution archive with "`make release`".

1.11 License and Disclaimer of Warranty and Liability