Computing with Integrity

Ty Everett
ty@tyweb.us

Abstract: Computing has always been subjected to shortfalls in the areas of architectural cross-compatibility and information centralization. We define a new, blockchain-based computing platform that standardizes user authentication and file sharing while providing verifiable integrity for application behavior and network services. Filesystems can be created, updated and shared with specific people or the entire world on a per-folder or per-file basis. Application developers can use existing technologies to build never-before-possible solutions for problems that have long plagued the existing model. Open-ended and extensible protocols and APIs provide for future expansion.

Background

Over the past two centuries, computing has advanced at an astounding pace. It all started in 1812 with a singular belief—held by Charles Babbage—that addition and subtraction could be automated by mechanical devices[1]. In the modern era, the internet connects the globe while virtual reality and artificial intelligence are forging new frontiers. Mobile devices that can be bought for the price of a meal can perform billions of calculations per second.

However, users and businesses still struggle with the most basic of tasks—things that should have been made trivial ages ago. File transfers between computers running different operating systems are slow and cumbersome. The publication and sharing of videos and documents is a complicated process unless users rely on centralized services to hold their data. Often the only solutions involve uploading files, unencrypted, to a cloud storage provider or social network only to needlessly download them again on a new device.

The “big data”-based advertising model creates an incentive to build addictive and unhealthy experiences that sacrifice privacy in the name of convenience. Native applications and programs lack true cross-device compatibility while the only real standard for unified app development, the web, requires developers and users alike to entrust their data to cloud service providers, CDNs and SSL certificate authorities by binding them to the “client-server model.” Often, when disruptive startups enter the arena, they find themselves with a need to entrust their business to the very firms they aim to replace.

Systems that rely wholly on centralized service providers to operate are fundamentally susceptible to manipulation and unprecedented degrees of control. When they grow to encompass the entirety of the planet and to govern the affairs of every person on Earth, these networks become unacceptable and even dangerous to the human condition.

In 2008, a paper describing the first working version of a digital currency was published[2]. For the first time, computers connected to a fully decentralized network were able to come to agreement about the sequence of events and to keep a ledger that no one can tamper with. This phenomenon is known as “the blockchain” and is the basis for the “integrity” this paper describes.

Introduction

Removing the need to trust centralized service providers and allowing developers to build applications that run in a deterministic way on client hardware—independent of the architecture or underlying host system—will allow for standards to emerge that promote cross-device compatibility while protecting user privacy.

A blockchain can be leveraged to store user data and track the state of a computational system so that users no longer need to trust a central server with their data. Anyone can download their profile onto any device and see their files as they left them. Because the architecture for developing applications is independent of the host system, users can access and use the same operating system and set of apps across all of their devices.

Their wallpapers, contacts and all other aspects of the system will remain constantly in sync. Files and messages they share with friends remain completely secure and are accessible only to those with authorization. All configurations, customizations and settings are updated instantaneously across users’ physical devices.

A standard API for authentication and for requesting permission to access personal information removes the need for storing passwords and managing multiple user accounts. Certificate authorities can verify the identities of users while providing privacy and transparency as they interact with one another on a more decentralized web.

Bootstrapping and System Access

To access this system, a user will need to be able to find their data on the blockchain, broadcast new transactions to mutate the state of their system and securely access the cryptographic keys needed to view their files, sign messages and perform other tasks.

In order that we may later arrive at this higher-level functionality, we must first start by defining some lower-level terms and concepts. In this document, *Terms* meant to be interpreted as defined will be in capitalized italics and will use the Times New Roman font.

- **System:** The cryptosystem, virtual machine, APIs, filesystem and suite of applications that are built in the ways outlined in this paper.

- **User:** The user is the person who is using the *System*. They can do things like enter a password, scan a fingerprint or click a button. They are not to be confused with the *Client.*

- **Client:** The software that the *User* interacts with as they use the *System*. The client is responsible for the proper implementation of the *System* and should enable its use. Any compatible *Client* connecting to the *System* should attempt to deliver a similar (if not identical) experience for the *User* compared with other *Clients*.

- **Secret Server:** Responsible for safeguarding the *Presentation Key* given to it by a *Client*, establishing a mechanism for later verifying the identity of the *User* who provided the key and—upon successful verification—giving the *Presentation Key* back to the *Client* that originally provided it.

  Examples of verification methods that a *Secret Server* might employ include:

  - Verification via SMS text message
  - Verification by asking that a *User* take a photo of themself with special instructions
  - Verification by providing government-issued identification documents
  - Some combination of the above

  To provide incentive that *Users* might choose their servers, secret server operators may offer to pay them a small amount of cryptocurrency during the *Initial Bootstrap Process*—enough that the *User* is able to complete the *Initial Bootstrap Process* and perhaps have enough to perform some other tasks with their new *System*.

  To make money, secret server operators might also consider:

  - Providing *Client* software
  - Offering services so that the *User* can obtain more cryptocurrency and continue the use of their *System* (i.e. a cryptocurrency exchange)
  - Operating an identity verification certificate authority (since they may already have verified the identity of the *User)*
  - Operating a *Blockchain Service Provider*
  - Operating an *Image Server*

- **Blockchain:** The immutable ledger which the *System* uses to store and persist its state. The blockchain needs to provide reasonably consistent and liberal network rules as well as inexpensive access for use of the ledger to store large amounts of data.

- **Blockchain Service Providers:** Businesses that sell or otherwise make blockchain data available in a highly performant manner in such a way that *Clients* can efficiently read and write data without performance issues or limitations.

- **Image Servers:** Businesses that are paid to host *Mountable Images* and serve them to anyone who pays the fee to download them.

- **Mountable Images:** Collections of files, folders and other *Mountable Images* which are arranged into a directory structure. *Mountable Images* can be mounted at any point in either the filesystem or the domain system. They might contain:

    - Files and folders
    - References to other *Mountable Images* mounted at various points within their directory structure
    - Static websites written in HTML/CSS/JavaScript or other languages
    - Programs, apps or network services that leverage system APIs to define other dynamic protocols or methods for interacting with the blockchain
    - Definitions for consensus rules that define new decentralized protocols for payments, communication or governance
    - Databases and user data in use by members of the above programs, apps, network services and protocols
    - Programs that leverage system APIs to create other blockchains with their own sets of consensus rules (or that run a virtual copy of any pre-existing blockchain)
    - Operating systems intended for use with the *System*
    - Source code repositories

- **Password:** The *User* will have a password—one password for the entire *System*, including all applications that run on it (as long as those applications leverage the system APIs for authentication). This password should be secure enough to prevent brute-force attacks.

- **Password Key:** In order to avoid the direct use of the *Password* in cryptographic operations, we must run it through a key stretching function. A new, random password salt is generated whenever the *Password* changes. A few hundred thousand rounds of PBKDF2 should be sufficient to transform the *Password* and salt into the new symmetric password key.

- **Presentation Key:** A key that is generated by the *Client* during the *Initial Bootstrap Process* and subsequently shared with one or many *Secret Servers*. During the *Initial Bootstrap Process*, this key can act as a private asymmetric encryption key to receive cryptocurrency from secret servers. It is also used as a symmetric key like the *Password Key* and the *Backup Key*.

- **Backup Key:** A key that is generated by the *Client* during the *Initial Bootstrap Process* and shown to the *User* in a human-readable form. The *User* is asked to save this key to avoid losing access to their *System* in the future.

- **Primary Key:** A symmetric key that is used for cryptographic operations by various system components.

- **Privileged Access Key:** A symmetric key that is used by various system components for sensitive tasks.

- **Primary Key Pair:** An asymmetric keypair used by the *Client* when it interacts with other *Clients* to exchange encrypted data, and by various system components and third-party applications.

- **Privileged Access Key Pair:** An asymmetric keypair used by the *Client* when it interacts with other *Clients* to exchange sensitive or confidential data, and by various system components and third-party applications.

- **Filesystem Key Trees:** Two BIP32 root filesystem keypairs are generated during the *Initial Bootstrap Process*. They are used by the *System* for managing the filesystem, shared folders, public folders and network services. The BIP32 derivation paths of the two trees of keys mirror one another. The first set are the “read keys” and the second set are the “write keys.”

- **Filesystem Read Keys:** A tree of BIP32 HD keys where only the private keys are significant. The private keys are used as symmetric keys for encrypting data stored in the filesystem at the various points (nodes) in the HD tree structure.

  Knowledge of a given node constitutes the ability to decrypt the content of that node. When also combined with knowledge of the public component of the corresponding *Filesystem Write Key*, finding and decrypting the content of all child nodes also becomes possible.

- **Filesystem Write Keys:** A tree of BIP32 HD keys where the private key controls an address on the blockchain. Transactions broadcasted from the key’s address which contain data encrypted by the corresponding *Filesystem Read Key* constitute valid changes to the filesystem.

  Thus, only someone with knowledge of both the *Filesystem Read Key* and the private component of the *Filesystem Write Key* for a given node can make changes to the filesystem’s contents at that node and all its child nodes.

Write access to a given node and to all its child nodes in the filesystem can be granted to another *User* of the System by disclosing the read and write filesystem keys at this point in the BIP32 HD tree with such other *User*'s *Client*.

- **Exclusive Bitwise OR Notation:** To signify a representation of two cryptographic keys which have been XOR'd together, the notation *XOR(Key A, Key B)* will be used.

- **Symmetric Encryption and Decryption:** When data is symmetrically encrypted and decrypted, the AES-GCM operation is implied with a 256-bit key. Initialization vectors are prepended to the beginning of the ciphertext. When reference is made to "ciphertext" in the context of performing symmetric operations, inclusion of the prepended initialization vectors is implied. New, random initialization vectors should be generated for every operation.

- **Broadcasting Data to the Blockchain:** When data is broadcasted to the blockchain, it takes the form of an OP_RETURN payload. Multiple PUSHDATA operations are used. The first is a prefix for the *System,* the second is a message identifier for the type of data being broadcasted, and this is followed by zero or more further PUSHDATA payloads defined in accordance with the specification for the message type being sent.

- **Initial Bootstrap Process:** When the *User* registers for a new account on the *System*, the *Client* will do the following:

    1. Generate a *Password Salt* and *Password Key* based on the *Password* entered by the *User*
    2. Generate a *Presentation Key*
    3. Generate a *Backup Key* and make sure to encourage the *User* to save it
    4. Generate a *Primary Key, Primary Key Pair, Privileged Access Key* and *Privileged Access Key Pair*
    5. Send the *Presentation Key* to one or more[1] *Secret Servers* and complete any verification required by the *User*'s chosen *Secret Server(s)*
    6. Wait for the *Secret Server(s)* to send cryptocurrency to the public address that corresponds with the *Presentation Key*, or alternatively allow for the *User* to provide their own cryptocurrency to fund the establishment of their new *System*
    7. Broadcast the *Password Salt* to the *Blockchain* using the funds deposited into the address corresponding to the *Presentation Key* by either the *User* or the *Secret Server(s)*
    8. Encrypt the *Primary Key* with *XOR(Password Key, Presentation Key)*, broadcasting the ciphertext to the *Blockchain*

[1] It is possible for multiple secret servers to be used, each with its own presentation keyslice. The user goes to each server one by one and collects all their keyslices. Once they have all of the slices, the user may XOR all of them together to reconstruct their final presentation key. Shamir systems could also be used.

9. Encrypt the *Primary Key* with *XOR(Password Key, Backup Key)*, broadcasting the result to the *Blockchain*
10. Encrypt the *Primary Key* with *XOR(Presentation Key, Backup Key)*, broadcasting the result to the *Blockchain*
11. Encrypt the *Privileged Access Key* with *XOR(Password Key, Primary Key)*, broadcasting the result to the *Blockchain*
12. Encrypt the *Privileged Access Key* with *XOR(Presentation Key, Backup Key)*, broadcasting the result to the *Blockchain*
13. Broadcast the public keys for the *Primary Key Pair* and *Privileged Access Key Pair* to the *Blockchain*
14. Encrypt the private key for the *Primary Key Pair* with the *Primary Key* and broadcast the ciphertext to the *Blockchain*
15. Encrypt the private key for the *Privileged Access Key Pair* with the *Privileged Access Key* and broadcast the ciphertext to the *Blockchain*
16. Generate the two *Filesystem Key Tree* root nodes (one read tree and one write tree), encrypt them with the *Primary Key* and broadcast them to the *Blockchain*
17. Destroy the *Backup Key*, *Password Key* and *Presentation Key*

The *Client* may now move any remaining funds from the *Presentation Key* into a wallet application after initializing the filesystem and booting the *User*'s chosen OS, as described in a later section of this paper.

- **System Sign-on Process:** After the *Initial Bootstrap Process*, when the *User* wishes to access the *System* (on a new device or in a new context), the *Client* performs the following steps to bootstrap access to the *System*:

1. Contact the *Secret Server(s)* the *User* uses for authentication and allow the *User* to verify their identity with the *Secret Server(s)*, thereby obtaining their *Presentation Key*
2. Scan the blockchain for the transactions broadcasted by the *Presentation Key* that contain the requisite data (*Password Salt*, encrypted copies of the *Primary* and *Privileged Access Keys*, etc).
3. Use the *Password* and *Password Salt* to obtain the *Password Key*
4. Use *XOR(Password Key, Presentation Key)* to decrypt *Primary Key*
5. Use *XOR(Primary Key, Password Key)* to decrypt *Privileged Access Key*
6. Use *Primary Key* and *Privileged Access Key* to decrypt and obtain all other data
7. Use the *Filesystem Read Tree* and *Filesystem Write Tree* to pull down and sync the filesystem from the *Image Servers*
8. Initialize the virtual machine and boot up the operating system

- **System Recovery Process:** In case the *User* loses access to any one of their keys, they may rely on the other versions of the encrypted *Primary* and *Privileged Access* keys which were broadcasted to the blockchain during the *Initial Bootstrap Process*. New versions of the affected keys can be re-broadcasted to the blockchain once the lost credential has been reset. However, since the XORed combinations only act as wrappers, the underlying keys themselves never need to change.

- **Proper Use of Privileged Access Key:** The *Client* may store and persist the *Primary Key*. However, the *Privileged Access Key* should not be kept by the *Client*. Since the *Primary Key* is known, and since *XOR(Primary Key, Password Key)* is used to wrap the *Privileged Access Key*, we can simply ask the user for their password in order to decrypt the *Privileged Access Key* when it is needed. *Clients* should prompt for the *Password* or for biometric authentication often—generally every 5 minutes—when the *Privileged Access Key* is continuously needed.

  If the particular physical device on which the *User* is accessing the *System* supports it, the *Client* may use hardware-backed biometric authentication to protect a copy of the *Privileged Access Key* with a device-specific key wrapper. Copies of the *Privileged Access Key* encrypted with this device-specific wrapper should never be broadcasted to the *Blockchain* and should only be stored on that specific device. When needed by the *System*, the *User* will use the biometric authentication method provided by the device to obtain access.

Filesystems, Operating Systems and Image Servers

Each *User* in the *System* will want a place to store their files, documents, applications, configurations, preferences and other information. A filesystem should allow for permissioning, cross-device access, live data streaming, indexability and highly-performant operation. Cryptography should also ensure that data can only be accessed with authorization. For data stored in a blockchain, the filesystem should also be conscious about not replicating the same data more than necessary.

To achieve this, we denote each node (file, folder or mountable image) in the filesystem as its own distinct blockchain address. Since BIP32 already provides a tree-like structure of addresses and private keys, we will leverage this system to create our filesystem.

We start with a single BIP32 root key, known as the *Filesystem Write Tree* root key. The blockchain address corresponding to this key is conventionally referred to as "/". To write to the filesystem, a transaction should be broadcasted from this address that contains the data to be written. In this way, a history of the "commits" to a given filesystem node (file, folder or mountable image) can be obtained simply by asking a *Blockchain Service Provider* for all the transactions coming from this address.

All filesystem data is encrypted before it is broadcasted by the write key (except public *Mountable Images* and public directories). A second, mirrored tree of keys is used for this encryption. These keys are known as the *Filesystem Read Tree*.

Knowledge of a given node in the *Filesystem Read Tree* constitutes the ability to read the data at that location. Knowledge of the private component of a given node in the *Filesystem Write Tree* constitutes the ability to broadcast transactions which mutate the state for that node (by appending “commits” to the corresponding blockchain address).

Generally, three categories of filesystem operations are defined:

- **Folder Operations:** Operations that denote a given node as a folder, name/rename a given folder node, change permissions or mark the folder as deleted.

- **File Operations:** Operations that denote a given node as a file, name/rename the file, write data to the file, patch the file with a diff, change permissions or mark it as deleted. File nodes are the only nodes which may not have any children.

- **Mountable Image Operations:** Operations that denote a given node as a mountable image, provide the hash of the mountable image to mount, provide contact information for *Image Servers* to download the image, provide an updated image hash or mark the image as unmounted.

During the *Initial Bootstrap Process*, the *Client* will normally mount a base operating system image of the *User*’s choice at the root “/” filesystem directory. This constitutes installation of that OS onto the *User*’s *System*.

The *Client* must implement a virtual machine that runs the WebAssembly architecture. All operating systems and applications must compile to this architecture to avoid architectural incompatibilities.

Clients should also broadcast an entry point that the *System* (the virtual machine) can use to boot the installed OS. This startup program (bootloader) is a WebAssembly binary serving as the initial entry point when executing the virtual machine. It is to be broadcasted as a special type of transaction to the “/” filesystem node address.

Mountable Images may be mounted at any node in the filesystem (including the root node), just like files and folders. *Mountable Images* are not stored on the *Blockchain*. Instead, when a *Client* creates a *Mountable Image*, it uploads the image to an *Image Server* and pays a fee for the server to store the image for some length of time.

The *Client* will then broadcast the hash of the image and contact information for any *Image Server(s)* that host the image. *Image Servers* may charge a fee for downloading the image, payable transparently by the *Client* in cryptocurrency.

Some *Mountable Images* (like operating systems, public network services and downloadable application packages) are left unencrypted. Anyone may download, mount and freely use these *Mountable Images*:

- As part of their own *Systems*
- As dependencies in their source code projects
- As part of their websites and apps
- As components for art/graphics or design projects
- As components for new songs or movies (which are also public *Mountable Images*)[2]

Image Server operators may set up arrangements to charge *Clients* on a per-download basis and compensate the *Mountable Image* publisher or author with a percentage of the revenue from their image downloads. Image authors can also operate their own *Image Servers* to avoid paying for hosting. For public images, the hash of the unencrypted *Mountable Image* is stored on the *Blockchain*.

Other *Mountable Images* (such as personal files, photographs and documents) can be encrypted with the *Filesystem Read Key* associated with the *Mountable Image's* filesystem node before being sent to the *Image Server*. In this case, the hash of the encrypted image data is stored on the *Blockchain*.

When the *Client* needs to change data that is stored in a file contained within a *Mountable Image* (to make customizations), the diff between the image's version and the modified version is broadcasted to the *Blockchain* at the filesystem node (a child of the *Mountable Image*'s mount point) where the file would have existed if the image's files were to be incorporated into the filesystem tree.

In addition to files and folders, *Mountable Images* can also incorporate other *Mountable Images* into their directory structure, and these child images can be mounted at any location within the parent image (including the image root). Multiple *Mountable Images* may not be mounted in the same directory at the same time. However, a second *Mountable Image* can be mounted in a directory after the first one is unmounted.

As an example, If the *User* wishes to back up photographs into a folder called "/backups/photos", the *Client* will do the following:

[2] This allows for artists to use each other's work and make derivative works. For example, imagine a movie which is defined as a mountable image. If the movie uses a song and if the movie's mountable image incorporates the song's mountable image into its own, anyone who downloads the movie will also need to download the song. The publisher of the song can dictate which image servers their work is hosted on and can control the terms under which it may be distributed (for example, charging on a per-download basis). Unauthorized hosting of copyrighted material in mountable images can later be proven by looking at the hash of the data on the blockchain.

1. **Create the /backups folder:** If the backups folder does not yet exist, the *Client* will:

    a. Look at the root-level branches of the *Filesystem Write Tree*
    b. Find the first unused branch (in this example branch #35)
    c. Send the address at m/34 of the *Filesystem Write Tree* enough cryptocurrency to broadcast a few transactions
    d. The address at m/34 of the *Filesystem Write Tree* will send transactions that denote the filesystem node as a folder and name the folder "backups"
    e. The content of the OP_RETURN payloads for these broadcasts are to be encrypted by the corresponding symmetric encryption key on the *Filesystem Read Tree* at read key tree branch m/34

2. **Create the /backups/photos folder:** If the photos folder does not yet exist in the backups folder, the *Client* will:

    a. Start with the root *Filesystem Write Tree* key
    b. Find the tree branch it had previously allocated for the /backups folder (the 35th branch, or m/34)
    c. From within the m/34 tree branch, find the next available child (in this case the first child, or m/34/0)
    d. Send enough cryptocurrency to the address at m/34/0 so that it can broadcast a few transactions
    e. The m/34/0 address will broadcast transactions denoting this filesystem node as a folder and naming the folder "photos"
    f. The OP_RETURN payloads of these transactions will be encrypted by the corresponding symmetric encryption key at m/34/0 in the *Filesystem Read Tree*

3. **Encrypt, Pay and Upload:** The *Client* will now:

    a. Allocate child nodes of the image mount point node (m/34/0/0) in a deterministic way for all files, subfolders and *Mountable Images* that are to constitute the new *Mountable Image*.
    b. Use the newly-allocated keys (m/34/0/0/{0...n}) of the *Filesystem Read Tree* to encrypt each subfolder and file of the new image
    c. Concatenate the encrypted data into a *Mountable Image* blob
    d. Find one or many *Image Servers* that accept and host encrypted data
    e. Pay the *Image Server(s)* to host the data
    f. Optionally, sign a message of the *Image Server*'s choosing with the *Filesystem Write Tree* key at m/34/0/0 (so that the *Image Server* may later update the image if requested by the *User*)
    g. Upload the *Mountable Image* blob to at least one of the *Image Servers*

h. (Optional) If more than one *Image Server* is used, the *Client* could select one of them and pay it to send the *Mountable Image* blob over to the other servers on its behalf. This way, the *Client* can avoid uploading the blob more than once.
i. Send the address at m/34/0/0 of the *Filesystem Write Tree* enough cryptocurrency to broadcast a few transactions
j. Broadcast a transaction from the address at m/34/0/0 of the *Filesystem Write Tree* denoting this filesystem node as a *Mountable Image* mount point within its parent directory (/backups/photos). The parent directory becomes the root of the *Mountable Image*.
k. Broadcast the hash of the encrypted *Mountable Image* blob to the *Blockchain* from the address at m/34/0/0 of the *Filesystem Write Tree*
l. Broadcast contact information for the *Image Servers* used by the *Client*
m. The above three transactions should have their OP_RETURN payload encrypted by the key at m/34/0/0 of the *Filesystem Read Tree*.

Suppose the *User* later wishes to delete the 23rd photo from this backup. The *Client* will:

1. Find the location of the *Mountable Image* mount point (m/34/0/0)
2. Find the address that corresponds to the file in the *Filesystem Write Tree* (address at m/34/0/0/22), and fund it with enough cryptocurrency to broadcast a transaction
3. Broadcast a transaction denoting this file as having been deleted
4. Optionally, the *Client* may also wish to re-image the /backups/photos directory if it finds the network, storage and other economic conditions to be favorable. In this case, the *Client* would:

    a. Contact the *Image Server(s)*, authenticating its identity by signing a message of the *Image Servers'* choosing with the *Filesystem Write Tree* key at m/34/0/0
    b. Send a signed diff to the *Image Server(s)* that will patch the image with the update
    c. Receive back and verify the *Image Servers'* hash of the updated image
    d. Broadcast the updated image hash to the *Blockchain* from the address at m/34/0/0
    e. Give the *Image Servers* a signed message from *Filesystem Write Tree* key m/34/0/0 telling them they are free to stop hosting the old version of the image (perhaps the *Client* or *Image Server* might owe each other some cryptocurrency after the image has been modified due to changed cost of hosting).

For public images, the same process will occur except that the image data is not encrypted by *Filesystem Read Tree* keys. The publisher of an image can still update their image by providing the server with a signed message from a key given to it at creation[3].

[3] It is cryptographically possible to split a single key so that, for example, three out of five signers are required to produce a valid signature to update a mountable image.

Image Servers may set policies for hosting their images, which might entail:

- Offering to host public mountable images for free but requiring payment in order to download them
- Requiring payment in order to host mountable images and offering free downloads
- Allowing the public to crowdfund the hosting of popular mountable images
- Not hosting images they find offensive or controversial
- Not hosting copyrighted material without permission

Applications and System APIs

Applications and programs that run on the *System* (which is a virtual machine running the WebAssembly architecture) must compile down to WebAssembly. This is to avoid architectural cross-compatibility issues.

These applications and programs can leverage all of the APIs made available by the OS which the user has installed on the *System*. Additionally, a few APIs are exposed at the virtual machine's "hardware" level.

Developers access these APIs the same way they would access any hardware device—by writing drivers so that the running operating system and user-space programs can make use of them.

- **Blockchain Data Interface:** This provides a standard way to query data from the blockchain and use it in the virtual environment. Operations might include:

    - Get transactions for an address
    - Get balance for an address
    - Get addresses involved with a transaction
    - Broadcast a transaction

  Since most *Blockchain Service Providers* will likely charge a fee for access to blockchain data, the *Client* should transparently manage its relationships with *Blockchain Service Providers* and provide a standard interface for the running operating system and user space programs to access blockchain data.

- **Filesystem Interface:** This allows the virtual environment to perform tasks that relate to the filesystem. Operations might include:

    - Read a file from a given path
    - Write a file to a given path
    - Create a directory
    - Mount or unmount a *Mountable Image* at a given mountpoint

    - Mark a file as deleted
    - List all files in a given directory
    - Mark a given directory as publicly readable or writable (causing the *Client* to broadcast the *Filesystem Read Tree* key and/or the *Filesystem Write Tree* key at the given filesystem node to the blockchain)
    - Grant read-write access to a given directory for a given *User* (causing the *Client* to discover that *User*'s public key, get a shared secret and send the forign *User* an encrypted copy of the filesystem read and write tree keys at that specific point in the filesystem trees)
    - Explicitly request that a new *Mountable Image* (either encrypted or public) be created for a given directory. Normally, *Mountable Images* are managed transparently to the running operating system by the *Client* and are encrypted by default, but this gives the *User* the ability to create and circulate specific *Mountable Images* as desired.

  The key material associated with the read and write filesystem trees should never be disclosed to the running operating system or to userspace programs by the *Client*. All file sharing and publication operations are to be implemented by the *Client* via APIs, not by the running operating system. This reduces the attack surface and minimizes implementational complexity for operating system and driver vendors.

- **Identity Interface:** Provides a way for the virtual environment to make use of the *Primary Key, Privileged Access Key, Primary Key Pair* and *Privileged Access Key Pair*. Operations might include:

    - Encrypt the given data with the *Primary Key*
    - Obtain an ECDH shared secret between the *Privileged Access Key Pair* and a given public key
    - Sign a given message with the *Primary Key Pair*
    - Get the public component of the *Privileged Access Key Pair*

  The underlying cryptographic key material associated with the keys should never be disclosed to the running system or be accessible to any program running on the *System*. This allows the keys to be used for authentication and reduces the risk of theft or unintended disclosure. All key material should be held and safeguarded by the *Client*.

- **Networking Interface:** An interface, implemented as the virtual hardware NIC device in the VM, responsible for carrying out all the functions of any other NIC. Additionally, this NIC should implement some extended functionality accessible to the *System*. Responsibilities of this interface include:

    - Sending and receiving ethernet frames (and all other functions of a traditional NIC)
    - A mechanism for the running virtual operating system and user space programs to resolve domain names (first by attempting a traditional DNS lookup and then through the *Domain System* root trust anchor configured by the *Client*.
    - Establishing connectivity with the internet (either by conventional means or by utilizing other *Clients* and *Network Bridges* and a purely peer-to-peer mesh).
    - Transparently to the running system, the *Client* should scan for and connect with other nearby *Clients* and establish a purely peer-to-peer mesh. If one or more of the *Clients* in the mesh has internet connectivity, they can offer to sell their bandwidth to the other *Clients*. When internet connectivity is achieved in this way, it should be transparent to the running operating system and user-space programs.

This interface should be implemented as the "network card" or "NIC" virtual hardware device for the VM, but would have some added features and functionality. Legacy drivers and software would still be able to use the NIC to access the traditional internet.

When, for example, the client-to-client network obtains internet connectivity, it is to be managed transparently for the *System* by the *Client*. The *Client* is responsible for implementing peer-to-peer protocols that facilitate decentralized connectivity and may use Wi-Fi, Bluetooth, cellular systems or any other mechanism available to the *User*'s physical hardware.

Shared Folders

Users have long desired a standard, seamless and interoperable way to easily and privately share files and collaborate on projects across the web. Building this functionality into machines at the (virtual) hardware level will allow for operating systems to make use of a standard API for the sharing of files and folders.

Folders in the filesystem (which may contain other folders, files and *Mountable Images*) can be shared with other *Users* or published for the world. This allows for private collaboration without the use of a central server.

Sharing with other *Users* is accomplished by the following simple process:

1. Obtain an ECDH shared secret between the private component of our *Primary Key Pair* and the public component of the other *User*'s *Primary Key Pair*.
2. Encrypt the key(s) corresponding with the filesystem node to be shared with the shared secret:

    a. To grant read-only access, share only the corresponding *Filesystem Read Tree* key

    b. To grant read-write access, share both the *Filesystem Read Tree* key and the *Filesystem Write Tree* key.
3. Contact the other *User*'s *Client* and provide the keys

To make the folder completely public, simply broadcast the key(s), unencrypted, to the *Blockchain* at the address for the filesystem node that is being published. Anyone may then use the keys to read or write to that part of the filesystem.

This functionality should be implemented as part of the *Filesystem Interface*, not by the operating system that runs in the virtual environment. This reduces complexity and attack surfaces for operating system vendors. High-level filesystem and sharing functionality should be exposed by the *Filesystem Interface*.

Domain Names

Without human-readable names, the ability to find information and access network services is severely restricted. A two-pronged approach can be used to achieve the best results:

- **Traditional domain system extensions:** DNS TXT records can point at blockchain addresses which act as mount points for *Mountable Images*. This allows the owners of internet domains to support fully-decentralized applications.

- **Domain System:** Anyone may start a new domain system by broadcasting a transaction from a blockchain address. The address becomes the root trust anchor for that particular domain system. It will then have the right to broadcast other transactions that will delegate other addresses as being responsible for top-level names. Those delegates may in turn delegate child addresses to be responsible for specific second-level domain names. Parents (including the root trust anchor) may broadcast a transaction revoking a child's right to use a subname at any time.

- **Addresses as Name Custodians:** Once a name has been delegated to an address by an authorized parent trust anchor or by the root trust anchor, that address has the right to:

    - Host records by broadcasting transactions that create such records, and
    - Delegate other addresses to be responsible for its subname children

- **New Capabilities:** In addition to the creation of records which map to IP addresses and other types of records from the traditional DNS system, addresses which are delegated as the custodian of a name may create records that:

    - Point to a filesystem directory or a *Mountable Image,* allowing anyone to easily access and run the program or website contained in that *Mountable Image*

    - Point to a “beacon” address watched by the *Client* that created the record. Other *Clients* can broadcast transactions that contain a small dust output as well as an OP_RETURN output allowing the creator of the beacon to find and connect with them.

- **Backwards Compatibility:** All name queries are first run through the traditional DNS system. They only run through the *Domain System* if a traditional DNS lookup fails. This allows support for the use of traditional internet names.

- **Convergence:** Because anyone may run a root trust anchor for a *Domain System*, it is up to the *Client* which *Domain System* trust anchor is used. As time goes on, *Client* software vendors may wish to decide which root trust anchor is to be authoritative.[4]

Network Services

Because *Mountable Images* may contain executable programs, and because executable programs can rely on *System* APIs, it is possible for *Mountable Images* to define applications, services and decentralized network protocols for communication between users. This can include social networks, micropayment incentive schemes and other decentralized services. Because *Mountable Images* can also be attached to the *Domain System*, accessing these decentralized services is as easy as accessing any traditional internet website.

Just like with the world-wide web, standards for *Mountable Image* formats and *Mountable Image* “browsers” could emerge to facilitate easier access to decentralized apps. Because a *Mountable Image* is just a directory, a good starting point could be to allow for HTML, CSS and JavaScript to represent a static website within a *Mountable Image*.

Authentication

Because the *Primary Key/Key Pair* and the *Privileged Access Key/Key Pair* are not disclosed by the *Client* to the operating system or running programs (applications can only encrypt/decrypt and sign/verify messages with these keys through the *Identity Interface*), these keys can be relied upon for authentication.

A simple request for the *Client* to sign an authentication request with the *Privileged Access Key Pair* is sufficient for all forms of user authentication. Any system API call that relies on the *Privileged Access Key* or *Privileged Access Key Pair* should cause the *Client* to prompt the *User* to either enter their *Password* or to perform biometric authentication.

[4] In production environments, the root trust anchor will likely be controlled by multiple trustworthy parties. Some m-of-n threshold signature scheme should probably be used.

Identity Verification and Management

At the *User*'s request, the *Client* can submit CSRs to identity verification certificate authorities. The verification service can verify the user's real-world identity and issue a certificate attesting that a given *Privileged Access Key Pair* belongs to a given real-world identity. A more privacy-centric approach is to have the CA simply attest to the validity of the person's identity without disclosing their name/the identifying information itself.

In the first case, financial applications can be certain of who they are dealing with, as long as they trust the CA. In the second case, social networks can know that someone has verified their identity without needing to know the person's real name. This reduces the likelihood for spam and eliminates the need to trust centralized platforms.

Client to Client Connectivity

Since the *Client* transparently operates a cryptocurrency wallet that it uses to keep the *System* running, it can use various networking and peer-to-peer technologies to connect with other *Clients* and to the internet.

For example, if Client A is in reach of Client B and if Client B is connected to the internet, Client A can contact Client B and offer to pay for Client B to relay internet traffic on its behalf. Client B makes money while Client A becomes able to connect to the internet, or to any other clients in range of Client B. Network bridges will emerge that attempt to earn a profit by making network access constantly available to all clients. Onion routing and ring signatures will protect privacy.

References

[1]: B. V. Bowden, Faster than Thought, Pitman (1953), p. 8.

[2]: https://bitcoin.org/bitcoin.pdf