Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für
Technische Informatik und
Kommunikationsnetze

Semester Thesis
at the Department of Information Technology
and Electrical Engineering

Making Bitcoin Exchanges Truly Transparent

AS 2014

James Guthrie

Advisors: Christian Decker (ETH Zürich)
Jochen Seidel (ETH Zürich)
Professor: Prof. Dr. Roger Wattenhofer (ETH Zürich)

Zurich
9th February 2015

Abstract

Bitcoin exchanges are an important component of the Bitcoin ecosystem, they open the Bitcoin market to a wider audience and publish open orders allowing for goods and services to be valued in Bitcoin. The trades and transactions between fiat currency and bitcoins are not performed on or recorded in the blockchain, and as such are not transparent to the customers of Bitcoin exchanges. Historically Bitcoin exchanges have been short-lived or suddenly closed with customers losing the bitcoins that the exchange held for them. The lack of transparency and recourse for victims of exchange closures has resulted in apprehension on behalf of the customers.

Some exchanges have taken to publishing anonymised customer account balances and Bitcoin addresses which they control as a rudimentary proof of solvency to customers. Others have gone as far as to implement a Merkle-tree based proof of liabilities which allows customer identities and account balances to remain anonymous, but the exchange must still publicly publish its Bitcoin assets. Although functional, this method necessarily makes more information available to the public than necessary, information which may be of strategic importance to the exchange or its customers.

This thesis proposes an automated, software-based audit of a Bitcoin exchange which does not expose any information which is private to the exchange or its customers to the public. The audit is executed on a trusted platform and methodologies are proposed with which customers can verify the entire audit, without becoming privy to more information than their own account balances. Essentially, the proposed audit delivers the result of *solvent* or *insolvent* and the necessary data structures to verify that the audit result must be correct, all of which are derived from public information.

Contents

1 Introduction

Since the conceptual introduction of Bitcoin in 2008 by Satoshi Nakamoto [1] and the appearance of the first Bitcoin client in 2009, Bitcoin has seen massive growth on a multitude of fronts. Bitcoin currently has a market capitalisation of 3 billion US dollars and an average daily transaction volume of approximately 50 million US dollars.

One factor which has driven widespread adoption of Bitcoin is the emergence of Bitcoin exchanges: companies which facilitate the trade between fiat currency (most commonly USD, EUR and CNY) and bitcoins.[1] Bitcoin exchanges have helped the adoption of the currency in two ways. Firstly, before the advent of Bitcoin exchanges, the only way to come by bitcoins was to *mine* them oneself or to informally trade bitcoins with other participants. Exchanges have opened the Bitcoin market to parties who might otherwise be averse to participating in a new or unproven market. Secondly, exchanges publish their trade books which establishes an accepted exchange rate between fiat currencies and bitcoins, allowing vendors to value their goods and services in bitcoins in accordance to the market rates in fiat currency.

Although Bitcoin exchanges have had a positive contribution to the Bitcoin economy, they are not without risks to the end users of the exchanges. In Moore and Christen's analysis of the risks involved with Bitcoin exchanges [2] they analyse 40 Bitcoin exchanges, at the time of publication 18 of the 40 exchanges had ceased operation. Of those 18, 5 exchanges did not reimburse customers on closure, 6 exchanges claim that they did and for the remaining 7 there is no data available.

Since the publication of that analysis the most high-profile exchange closure took place: the bankruptcy and closure of the Mt. Gox Bitcoin exchange, in which 650,000 bitcoins belonging to customers of the exchange were lost or stolen. At the time, Mt. Gox claimed that a flaw in the Bitcoin protocol was to blame for the loss of its client's bitcoins, a claim which has since been refuted by Decker and Wattenhofer [3]. At the time of the event, Mt. Gox was one of the longest-running exchanges in the Bitcoin market with its cumulative number of transactions accounting for approximately 70% of all Bitcoin transactions.

Unlike other financial services (for instance credit cards or bank transfers), Bitcoin transactions are irreversible by design. Once a user has transferred his bitcoins to another user there is no way that he

[1] In this document, *Bitcoin* refers to the Bitcoin application or network, whilst *bitcoins* refer to the currency

will get them back without the cooperation of the recipient. Compounding this is the fact that there is little recourse for the customer of an exchange: Bitcoin is new ground for insurers, regulators, and law enforcement who do not yet have any established methods for dealing with Bitcoin related issues.

In a system which was designed to be secure against malicious participants in its network of anonymous peers, the path of least resistance for attackers is not the system itself but the users of the system. Some efforts have been made by the Bitcoin developers to build more safety into the system, one form of this is *multi-signature transactions* which allows for escrow services to perform arbitration between the two parties of a transaction.

In an effort to calm customers fears, some exchanges have taken to periodically publishing data proving their solvency: an anonymised list of their customers account balances and a list of Bitcoin addresses which belong to the exchange. If the balance of the bitcoins available on the addresses is greater than the sum of the amounts owed by the exchange then it is solvent. Although customers may be appreciative of this type of transparency, it may put the exchange at a disadvantage as it relies on revealing information of strategic importance to the exchange, such as the number of customers, the amounts the exchange's customers keep on hand and the total balance of bitcoins held by the exchange.

In conventional financial markets trust is placed in the financial statements made by institutions such as exchanges or investment funds through the process of auditing. An independent third party, which is perceived to be trustworthy by the customers of the institution, inspects the financial records of the institution and publishes a statement that the financial records are believed to be accurate. From the financial statements, customers can determine the solvency of the institution and can decide whether they wish to use their services.

A full financial audit is an expensive and time-consuming process and is typically only performed in well-spaced intervals. These factors make such a service somewhat impractical for Bitcoin: customers aren't likely to trust a young exchange, which certainly will not possess the required finances to have an independent audit performed. Even established exchanges are not immune, most of the collapsed Bitcoin exchanges were not long-lived [2], with their closure either being immediate or over a relatively short period of time. On those time scales, an audit conducted every six months does not provide much assurance to customers of an exchange.

It is possible to imagine a scenario in which the financial auditor is not an auditor as we have come to know them in the traditional sense, but a piece of software which is executed in such a way that the correctness of the result which it delivers is trusted. Trusted Computing (TC) is an initiative supported by many leading companies in the technology sector which allows for one to place trust in the correctness of the result of a computation which is both computed and delivered by a third party. In such a scenario, the traditional limitations of financial auditing no longer apply, software executes orders of magnitudes faster than humans can process, and the execution of a piece of software is generally not costly at all. As such, it is feasible to imagine providing daily audits of a Bitcoin exchange through the use of Trusted Computing. This thesis will show that it is possible to perform a software-based audit of a Bitcoin exchange without revealing any information about the bitcoins that are possessed by either the exchange, or its customers to the public.

1.1 Related Work

Work has been done by Hal Finney to secure user's Bitcoin accounts using Flicker as a platform for Trusted Execution [4]. Finney's approach protects the user's private key in the trusted environment and lets the user define a daily maximum limit on transactions. The trusted environment signs new transactions until the daily limit is exhausted. This approach allows the user to create new transactions but limits the impact that malware could have if the user's platform were to be infected.

The Bitcoin developers introduced multi-signature transactions into the Bitcoin standards in 2011 [5] and are currently working on bringing the multi-signature system to modern Bitcoin accounts [6]. Multi-signature transactions and accounts open the Bitcoin ecosystem to services such as escrow, and provide additional security to users of Bitcoin.

Some investigative work has been done in the field of regulation, examining the impact of Bitcoin on current anti-money laundering (AML) policy [7]. The state of New York is working on a proposal dubbed BitLicense [8] which proposes regulations for providers of Bitcoin services in New York state. The proposed regulations are intended to protect consumers and enforce international AML policy.

Intel has worked to improve on the functionality provided by its Trusted Execution Technology and has released the Intel Software Guard Extensions specification [9] which implements the functionality provided by Flicker directly into the CPU. In the SGX model, the CPU has provides a secure *enclave* which contains a protected heap, stack, and instructions which are to be executed in a trusted execution environment.

2 Preliminaries

The software-based audit of a Bitcoin exchange relies on an understanding of both the Bitcoin project as well as Trusted Computing. This chapter introduces the fundamentals of Bitcoin and Trusted Computing.

2.1 Bitcoin

Bitcoin is a decentralised digital currency built on cryptographic protocols and a system of *proof of work*. Instead of relying on traditional, centralised financial institutions to administer transactions as well as other aspects concerning the economic valuation of the currency, peers within the Bitcoin network process transactions and control the creation of bitcoins. The major problems to be solved by a distributed currency are related to how consensus can be reached in a group of anonymous participants, some of whom may be behaving with malicious intent.

The core design of Bitcoin focuses on achieving a stable consensus amongst peers within its network of users. Transactions within the Bitcoin network are based on public key cryptography, users of Bitcoin generate an *address* which is used to receive funds. The Bitcoin address is derived, through cryptographic hash functions, from the public component of an ECDSA private/public key pair. A Bitcoin transaction records the transfer of bitcoins from some input address to output addresses. A transaction consists of one or more inputs and one or more outputs, each input to a transaction is the output of a previous transaction. The output of a transaction may only be used as the input to a single transaction.

When the sender composes a transaction, the output component consists of the Bitcoin address of the receiver and a value of bitcoins to be transferred. Similarly, the sender must prove that they are the intended recipient of the inputs i.e. that they are in possession of the private key belonging to the public address. The sender does so by providing the public key from which the address is derived and a signature which is signed by the corresponding private key. Although there are other transaction types, this is the most common type of transaction in the Bitcoin network and as such, the most relevant.

Transactions are generated by the sender and distributed amongst the peers in the Bitcoin network.

Transactions are only valid once they have been accepted into the public history of transactions, the *blockchain*. Transactions are grouped into units called blocks, each of which contains a reference to a previous block, building a chain of blocks which can be followed from the newest to the oldest block. As the blockchain contains the entire transaction history and is publicly distributed, a user can determine their balance of bitcoins by examining the unspent transaction outputs (UTXOs) for which they possess the private key.

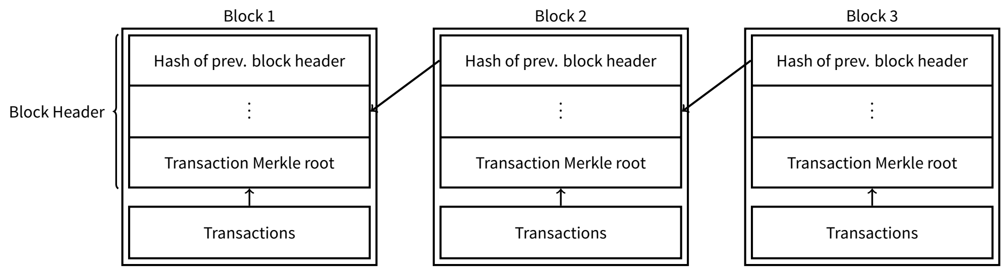

Figure 2.1: A simplified representation of the blockchain.

Blocks are only accepted into the blockchain if they fulfil a certain criteria or target: the hash of the block header must contain a specific number of zeros as a prefix. The block header consists of, among other things, a 4-byte timestamp and a 4-byte nonce. Permuting the nonce results in a different hash digest of the block header. Nodes wishing to add a block to the blockchain permute the nonce and recompute the hash until a hash is calculated which contains the desired number of zeros as a prefix. This calculation is not trivial, and composes the *work* that nodes must perform in order to generate a new block. The hash of the block header is therefore a proof that the node had to do a certain amount of work to generate the new block. Nodes in the network are incentivised to perform this work, called *mining*, through a reward[1] which they receive for each block added to the blockchain. The target is scaled with the computational power of the Bitcoin network so as to produce bitcoins at a constant rate.

2.1.1 Bitcoin Exchanges

Bitcoin exchanges facilitate trade between fiat currency and bitcoins. In order to trade on the exchange, users create an account with the exchange and transfer either fiat currency or bitcoins to the exchange. Should the customer wish to retrieve their bitcoins, they must make a request that the exchange transfers the bitcoins to an address which they control. The exchange manages a balance of the bitcoins that the customer has deposited with the exchange or traded for against fiat currency.

The customer may place buy and sell orders for bitcoins or fiat currency which are executed for the customer by the exchange, adjusting the balances of the customer's Bitcoin or fiat currency accounts. The orders are executed internally within the exchange, that is they are not recorded in the blockchain. Given this model of operation, a Bitcoin exchange is not merely a marketplace but also acts as a fiduciary, administrating both fiat currency and bitcoin accounts for its clients.

The audit proposed in this thesis does not require performing any transactions on the Bitcoin blockchain, it merely treats the blockchain as a ledger of the exchange's Bitcoin accounts.

[1]Presently 25 bitcoins per block and is halved every four years

2.2 Trusted Computing

In traditional personal computing paradigms, consisting of a multitasking Operating System (OS) and any number of client applications interacting with the OS, there is no method for the verification of the integrity of a computational result. That is, there is no way to ensure that the result of a computation has not been tampered with, for instance by some malicious software executing on the same platform. This problem can be extended to interactions with third parties. If a third party is tasked with performing a computation, there is no method for the verification of the integrity of the result, short of performing the computation locally, which in some circumstances may not be feasible.

Trusted Computing (TC) provides methodologies which allow for such a computing paradigm to exist. The Trusted Computing Group (TCG)[2] has the following definition of trust:

> "Trust is the expectation that a device will behave in a particular manner for a specific purpose." [10]

TC allows for the definition of *device* to extend beyond physical hardware, allowing for software which is executing on a remote device to also be trusted. The tools which provide a third party with the ability to prove trust are defined by the TCG Architecture [10].

The TCG defines a device which can provide trust as a *trusted platform* and states that a trusted platform "should provide at least three basic features: protected capabilities, integrity measurement and integrity reporting" [10].

Protected Capabilities are commands which may access *shielded locations*, areas in memory or registers which are only accessible to the trusted platform. These memory areas may contain sensitive data such as private keys or a digest of some aspect of the current system state. An important component in proving trust are the Platform Configuration Registers (PCRs), 20-byte registers which are only accessible through the *extend* operation. The extend operation can be used to modify the state of a PCR with a hash digest h according to the following formula: $\text{PCR}_{k+1}[i] = \text{SHA-1}(\text{PCR}_k[i] \parallel h)$. The properties of a crypographic hash ensure that the value held in a PCR cannot be deliberately set or chosen.

Integrity Measurement is the process of *measuring* the software which is executing on the current platform. A measurement is the cryptographic hash of the software which is executing throughout each stage of execution, the digest of the cryptographic hash is stored in a PCR through the *extend* function.

Integrity Reporting is the process of delivering a platform measurement to a third party such that it can be verified to have originated from a trusted platform.

These features of the trusted platform are deployed on consumer hardware in a unit called the Trusted Platform Module (TPM), a secure cryptographic co-processor, which is usually incorporated on the mainboard of the hardware. The TPM offers additional functionality, some important aspects of which are:

Cryptographic Keys

The TPM is equipped with two private keys which are stored in non-volatile memory and may not leave the TPM. The first is installed into the device by the manufacturer and is called the Endorsement Key

[2] `http://www.trustedcomputinggroup.org`

(EK). The EK may only be used to decrypt messages, never for encryption or signing, this can be used to authenticate that a device is a real TPM. The second key is called the Storage Rook Key (SRK) which is used to protect or *wrap* further keys which may be generated by or loaded into the TPM, but which are not stored on the TPM.

The TCG specifies a number of different types of cryptographic keys which may be present on the TPM, for the purpose of this document there is only one further type of key which is of interest, the Attestation Identity Key (AIK). AIKs are non-migratable signing keys which are used to sign data which originates from the TPM, in order to attest to the values originating from the TPM. Parties which need to verify a TPM attestation require the signed attestation, the public component of the AIK and a proof that the AIK belongs to a valid TPM. To this end, the TCG specifies a protocol for Direct Anonymous Attestation (DAA) [11].

Sealing Data

The TPM can be used to *seal* data which encrypts the data with a key which is loaded in the TPM and binds the data to the state of some of the PCRs. The encrypted data may only be decrypted or *unsealed* if PCRs are in the same state as when the data was sealed.

2.2.1 Root of Trust

The TPM has traditionally been used to provide a trusted platform based on a static root of trust for measurement (SRTM). In this paradigm, the system begins booting in a piece of firmware which is always trusted, the static root, and each component of the boot process is measured and verified against a known-good configuration before it is executed in order to assert that no component has been tampered with.

Although SRTM technology can be used to establish a trusted platform, it suffers from a number of deficits. The first is that software execution scenarios are by no means simple: a piece of software will interact with a number of other pieces of software in the same execution environment as well as with hardware devices and with any number of entities through networking. Although the establishment of a trusted platform at boot time is helpful to ensure that the platform is untainted on initialisation, it is no guarantee that the platform will remain untainted throughout its execution. Once a platform has become tainted, SRTM is of little use as it can merely determine that the software is no longer trusted and the boot procedure should be stopped. Secondly, as the SRTM builds its chain of trust, the software components become increasingly complex, with increasingly more opportunities to suffer from bugs or other logic errors. A boot loader may be a few hundred lines of code, but an operating system kernel may be in the millions of lines of code. The execution of a few routines which are security-crucial does not need to rely on a trusted computing base (TCB) consisting of all of the code from bootloader to kernel.

The solution to this problem is provided through technology which facilitates a so-called Dynamic Root of Trust for Measurement (DRTM). DRTM allows for a trusted platform to be established dynamically without requiring a system reboot. It even allows for a trusted platform to be established within a platform which is known to be compromised with malicious software.

DRTM is implemented in consumer general purpose processors from Intel and AMD under the names Intel Trusted eXecution Technology (TXT) and AMD Secure Virtual Machine (SVM). In the case of Intel TXT the DRTM is established using an Authenticated Code Module (ACM) as the root of dynamic trust. The ACM is digitally signed by the chipset vendor, the public key required for validation is present in the processor; only if the signature is successfully validated can the launch procedure continue.

The ACM functions as a secure bootloader for a lightweight piece of code which is to be executed on the processor in complete isolation from any other software or hardware access. Intel TXT and AMD SVM provide additional security features when executing in the secure mode, such as turning off system interrupts to prevent other execution paths, as well as memory protection for specific memory ranges which also prevents DMA access [12].

2.2.2 Flicker Platform

Flicker [13] is a software platform for the Windows and Linux operating systems which leverages DRTM to allow security sensitive components of software applications to execute in a secure, isolated environment. The developers of Flicker call such a component a Piece of Application Logic (PAL). In order to reduce the TCB to the bare minimum, the PAL is envisioned as only the routines required to perform some security critical computation component of the application, leveraging both protection provided by the processor for DRTM as well as features provided by the TPM.

Flicker consists of two components, the kernel module which prepares and launches the DRTM process, and the Secure Loader Block (SLB) core which performs bootstrapping of the secure execution environment for the PAL. The PAL code is linked against the SLB core to form the SLB, a term which originates from the AMD SVM documentation the Intel equivalent for which is the Measured Launch Environment (MLE). In this document, SLB, MLE and PAL binary are used interchangeably.

PAL Execution

The execution scenario in which the PAL runs is made up of four distinct components: the user application, the Flicker kernel module, the ACM, and one or more PAL binaries, consisting of the SLB core and PAL.

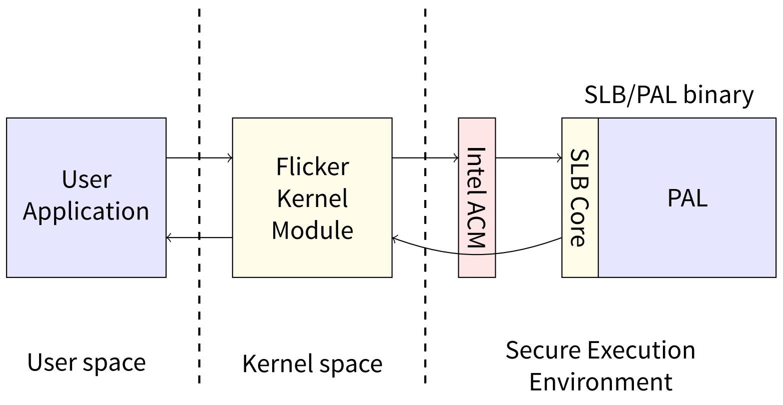

Figure 2.2: Flicker PAL execution scenario.

The user application is a conventional application executing in userspace. The Flicker kernel module provides `data` and `control` file system entries with which the user application may interact in order to provide the Flicker kernel module with the SLB, PAL, and the inputs, as well as to read the outputs when execution of the PAL terminates.

Figure 2.2 illustrates the control flow when the user application needs to perform a security-critical task. First the application passes the PAL binary and inputs to the flicker kernel module and instructs the kernel module to execute the PAL. The flicker kernel module prepares the necessary data structures and memory protection to launch the DRTM and start the PAL, it then invokes the GETSEC[SENTER] CPU instruction which disables interrupts and triggers the start of the DRTM. These data structures are measured by the Intel ACM, which forms the root of the DRTM. The ACM hands control over to the flicker SLB core which invokes the PAL and contains the necessary data structures to return the control flow directly to the Flicker kernel module when the PAL has finished executing.

During the execution of the SENTER operation, the dynamic TPM PCRs (17-23) are initialised to zero. PCR 17 is then extended with the hashes of the SINIT ACM and a number of configuration parameters. During the execution of the SINIT, PCR 18 is extended with the hash of the MLE. These PCR values are provided in the TPM's attestation, which can be used to prove to a third party that the PAL binary was executed and calculated the output values.

3 Auditing

A naïve approach to auditing may be for the exchange to publicly post the anonymised balances that customers hold with the exchange, and to show that the exchange is in possession of private keys which have large enough balances to cover the customers' accounts. There are however some deficiencies to this approach: there is a risk that customers privacy may be violated despite anonymisation of the data, and the exchange must disclose much more information than necessary in order to prove its solvency.

Leveraging Trusted Computing it is possible to convince customers of the exchange that it is solvent without revealing any of the above information to the broader public. The automated software-based audit has similar semantics to that of a traditional audit in which the auditor has access to all of the exchange's records, some of which are private. Based on the contents of these records, the auditor can deliver a verdict as the state of the exchanges finances. The auditor's verdict is accepted by customers of the exchange because of trust which is placed in the auditor, trust that the auditor is operating transparently and without error.

Trusted computing allows for trust to be placed in the platform which is executing a software-based audit, giving it similar, if not more, legitimacy to that of a traditional auditor. The software executing the audit must be open-sourced so that customers can inspect and verify that the auditing software does as advertised. Through the open sourced software, customers can make the connection between the source code of the audit and the binary which is executed and attested to in the trusted environment.

3.1 Overview

The audit should determine the solvency of the exchange, in principle this is a binary result, either *solvent* in the case that the exchanges assets in bitcoins are greater than its liabilities in bitcoins, or *insolvent* otherwise. It is plausible that there are situations in which this binary result does not suffice, for instance an exchange which wishes to prove fractional reserves. In these cases a multiplicative factor can be multiplied with the liabilities of the exchange to show that the exchange can cover some percentage of its liabilities with its assets.

The auditing process can be broken into three individual steps: proof of liabilities, proof of reserves (assets), and proof that the reserves are greater than the liabilities (proof of solvency). Figure 3.1 illustrates an overview of the components of the audit, the inputs to each of the components of the calculation and the outputs of the audit.

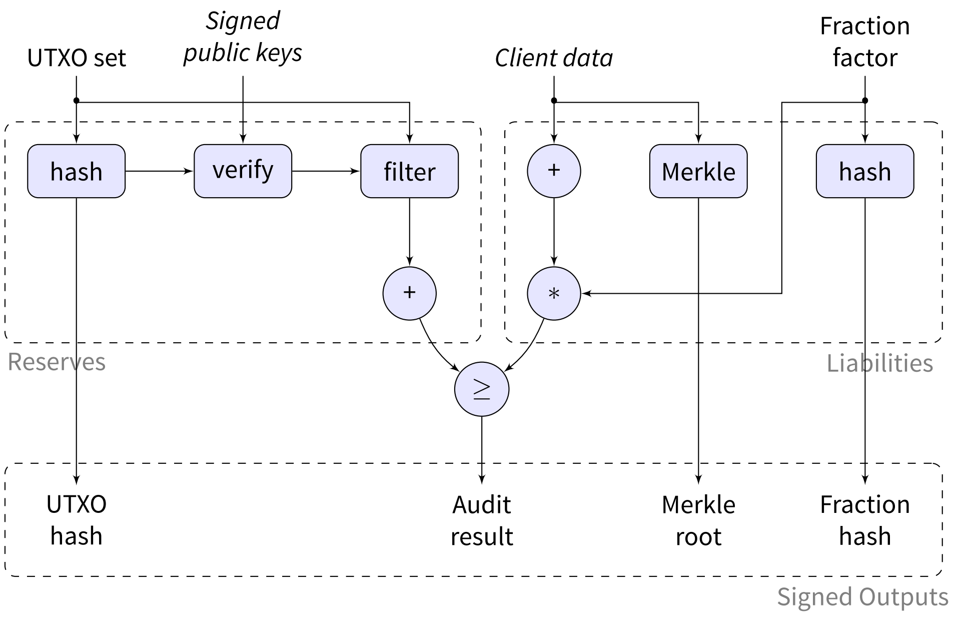

Figure 3.1: An overview of the audit process. Italicised values are not published publicly.

The four inputs to the audit are:

UTXO set is a list of all unspent transaction outputs in the Bitcoin blockchain. Although the blockchain is publicly distributed, the method with which the UTXO set is determined should be made public as customers need to determine the same UTXO values in order to verify the audit.

Signed public keys are used to determine which addresses in the UTXO set contain outputs which belong to the exchange. The signature should be of both the public key which the exchange controls, as well as the hash of the UTXO set and serves as proof of possession of the private key belonging to a public Bitcoin address. The exchange's public keys are not publicly declared (indicated in Figure 3.1 by italicised text).

Client data is a listing of the customer identifiers and account balances of the customers of the exchange. This information is not publicly declared, each customer has knowledge of their customer identifier and balance, but not that of other customers.

Fraction factor is a fraction between 0 and 1 indicating to which percentage the liabilities of the exchange are covered through its assets.

Additionally, the audit has four output values:

UTXO hash is a cryptographic hash of the UTXO set which was input to the audit.

Audit result is a boolean result, either `true` if the exchanges assets are greater than the fraction of the liabilities or `false` otherwise.

Merkle root is the root of the merkle tree over the client data.

Fraction hash is the cryptographic hash of the fractional value which is input to the calculation.

The output values are signed with the Attestation Identity Key (AIK) by the TPM, which also signs a hash digest of the binary which was executing at the time. The UTXO hash, Merkle root, Fraction hash, binary hash, and signature are all required to prove to customers of the exchange which values were processed in the audit. The signature of these data by the TPM combined with knowledge of how the audit software functions provide proof that the audit produced the output value based on specific input values.

3.2 Proof of Reserves

The reserves (assets) that the exchange possesses are in the form of bitcoins in the blockchain. The sum of assets is therefore calculated by determining which balances in the blockchain the exchange has access to and calculating the sum of those balances. In order for the exchange to access the bitcoins which are in the blockchain it needs to be in possession of the private keys belonging to the public addresses in the blockchain.

To simplify the calculation, the audit program does not need to parse the entire blockchain to determine which balances should be summed. Instead, a preprocessor can be used to extract all of unspent transaction outputs (UTXOs) from the blockchain. The UTXO set is a few hundres megabytes in size, as opposed to the approximately 25GB of the full blockchain. A UTXO entry is a tuple consisting of a public Bitcoin address and a value of bitcoins which can be collected from that address:

$$\langle \text{Address}, \text{Value} \rangle$$

The exchange can prove control of a Bitcoin address by providing the public key belonging to that Bitcoin address and signing it with the private key belonging to that public key.

For additional safety, the exchange should also sign a value which can be used to prove the freshness of the signature, a nonce. The Bitcoin blockchain is an ideal candidate for this, instead of using the hash of a block header we propose using a hash of the UTXO set that is provided as a timestamp, as it is guaranteed to change with every block added to the blockchain, it is not predictable, and it is not possible to manipulate the blockchain to create a desired UTXO set. Thus, the *Public keys + Signatures* input consists of a list of tuples which are made up of a public key, and a signature of the public key and a nonce:

$$\langle \text{PubKey}, \{\text{PubKey}, \text{Nonce}\}_\sigma \rangle$$

where $\{\text{data}\}_\sigma$ indicates that *data* is signed with the corresponding private key.

The overview of the steps of the calculation of reserves is shown in Figure 3.1, internally it consists of four different stages. The first component, computes the hash of the UTXO set, which is required in the *verify* stage. The verify stage asserts that the signatures for the public keys are valid and that the provided nonce matches the hash of the UTXO set provided. It then passes the public keys to the *filter* stage, determines the Bitcoin address and filters for entries in the UTXO set which match the exchange's Bitcoin addresses. Finally, these balances of these entries are summed. The sum, as well as the hash of the UTXO set are produced as outputs of the proof of reserves.

3.3 Proof of Liabilities

The liabilities of the exchange are the balances in bitcoins owed to its customers. The liabilities could be bitcoins which a customer deposits with the exchange before receiving their fiat currency, or the bitcoins which the exchange must pay out to the customer in exchange for fiat currency. The liabilities can be represented as list of a tuples consisting of a customer identifier and a positive balance owed to the customer:

$$\langle \text{CustID}, \text{Balance} \rangle$$

In Figure 3.1, these are labelled *Client data*. An additional input to the proof of liabilities is the *fraction factor*, which is multiplied with the sum of client account balances to prove fractional reserves. The client data is private data and is not publicly published, the fraction factor is publicly published.

Using the above definition of liabilities, the total liabilities of the exchange are calculated as the sum of all customer account balances. The calculated sum is later compared against the sum of reserves to determine solvency. Additionally to the sum, the proof of liabilities component calculates the root of a Merkle tree [14], as well as a hash of the fraction factor.

The method for proof of liabilities pursued in this thesis was initially proposed by Greg Maxwell and published by Zak Wilcox [15]. It suggests using a modified Merkle tree as a method to prove to customers that their account balances have been taken into consideration in a calculation without having to reveal any more information to the customer than that which they already possess.

The basic schema is to construct a Merkle tree with the customer input. That is, in order to compute a leaf in the tree one would take the cryptographic hash of the customer identifier and the balance owed to the customer. The leaves are then combined in a pairwise fashion and hashed, forming the nodes in the next layer of the tree. Nodes are combined and hashed until the root of the tree is constructed.

As the root of the Merkle tree is dependent on all of the individual values within the tree, it can serve as public record of the account balances which were counted in the summation of all account balances, without revealing individual customers account balances. The verification of the Merkle tree is performed individually by customers of the exchange. The customer identifies itself with the exchange, and requests the nodes in the Merkle tree which it would require to reconstruct the root hash which was published with the proof of solvency.

The modification proposed by Maxwell is to track the sum of the account balances as the tree is constructed, including the sum calculated so far as an input the the hashes of the subtrees. This was proposed as a safety mechanism against a malicious exchange inserting accounts with negative balances into the tree in order to reduce the overall balance of liabilities to be proved. The theory is that if negative account balances were to be inserted into the tree in such a way as to cancel out the balance of another customer (or group of customers) then at some point in the tree a negative balance ought to be revealed to a customer who would be alerted to the irregularity.

This modified Merkle tree also allows a customer to determine the total balance of liabilities in the tree, which we would prefer to avoid as it gives out more information than strictly necessary. This problem can however be avoided entirely by disallowing or ignoring customers with negative account balances during the construction of the merkle tree in the auditing program which is executed in the trusted environment.

3.4 Proof of Solvency and Verification

The proof of the solvency of a bitcoin exchange consists of two components, one is the outputs of the audit, the other is an attestation which can be used to verify that the auditing software was executed in the trusted environment, and that it computed the outputs which are attested.

All but one of the outputs of the audit have previously been described. The final output is the *Audit result*, which is a binary value, *true* if the reserves are greater than or equal to the liabilities, and *false* otherwise. The attestation of the output values is performed by the TPM whilst the platform is in the trusted execution mode. The attestation includes the PCR values 17 and 18, which are initialised during the trusted platform launch and contain measurements of the binary which was executed. Given the source code of the auditing program, a customer can compile the PAL binary, determine the hash and then determine that the output values were calculated and signed by the auditing program. With the public component of the AIK and proof that the AIK belongs to the TPM, the client can verify that the signature is valid and must have been produced by a TPM whilst the audit was executing in a trusted execution environment.

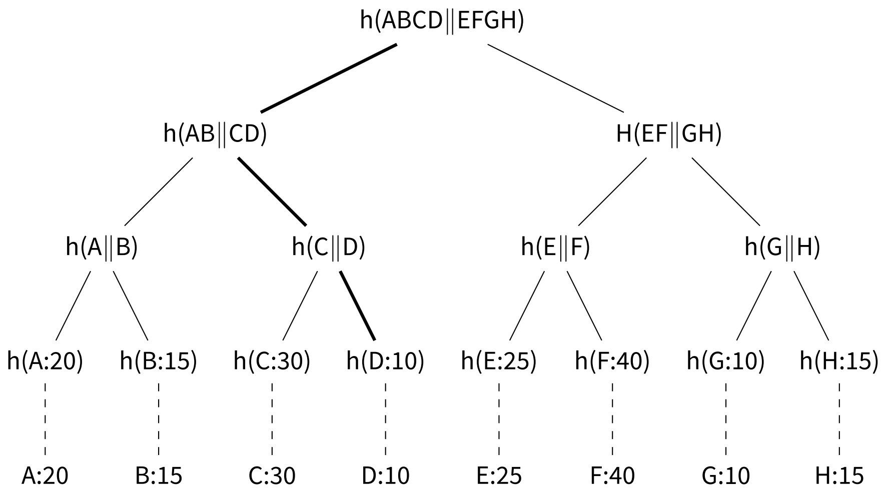

Figure 3.2: An example Merkle tree with the path from h(D:10) to the root highlighted

The customer can use the root of the Merkle tree to verify that its account balance was included in the calculation. The Merkle tree in Figure 3.2 shows a potential scenario in which customer D wishes to determine whether it was accounted for in the hash h(ABCD||EFGH). The nodes which D requires are the children of the nodes along the path from D's leaf node to the root excluding the nodes along that path. These are the nodes h(C:30), h(A||B), h(EF||GH). With these node values, D can reconstruct the path from its leaf node to the root, calculating the same value of h(ABCD||EFGH) that was provided by the exchange.

In order for such a scheme to function reliably, customers of the exchange must actively verify the Merkle root provided by the exchange in when it publishes the results of an audit. If this were not to be the case, a malicious exchange could very well zero some account balances, reducing the total sum of liabilities and potentially allowing the exchange to get away with holding less assets than it would need in order to assure solvency. Should a single customer of the exchange discover that their balance was not accounted for in an audit, it immediately brings into question the legitimacy of the audit.

4
Implementation

The Flicker platform was designed with lightweight, short-lived computations in mind, as such it imposes a number of restrictions which make a direct implementation of the audit as outlined in Chapter 3 unfeasible. The major restriction which poses problems for the automated software audit is memory. The Flicker environment has a stack size of 4KB, a heap size of 128KB, and a maximum input size of approximately 116KB. An additional restriction is the time overhead of invoking a Flicker session. Each Flicker session has a significant overhead, between 0.2 and 1 second, depending on which TPM functionality is used during the invocation [13].

4.1 Memory Requirements

Of the four inputs to the audit process illustrated in Figure 3.1, three may be of considerable size: the UTXO set, the public keys and signatures, and the client data. The blockchain currently contains approximately 17 million UTXO entries, each of which consists of a 25-byte address and an 8-byte value for an overall size of 33 bytes per entry. The memory requirement to store 17 million UTXO entries at 33 bytes per entry is 561MB, significantly larger than the input size of 116KB.

The size of the set of public keys and signatures is impossible to estimate, as it depends on the number of addresses on which the exchange distributes their bitcoins. Each entry of the public keys and signatures input consists of a secp256k1 ECDSA public key and signature which are 65 bytes and 72 bytes long respectively, for a total of 137 bytes per public key and signature. Given that both UTXO entries and addresses and their signatures need to be processed simultaneously, we propose an upper limit of 100 addresses and signatures, although this is not a fixed constraint.

The total size of the set of client data is also somewhat difficult to predict as it depends on the number of customer accounts which are held with the exchange. Unfortunately there are no current statistics available directly from exchanges as to the size of their customer base. Coinbase, the largest bitcoin wallet and exchange service announced in February 2014 that they reached the milestone of 1 million consumer wallets [16]. Blockchain.info, the provider of a web-based blockchain explorer interface also provides a wallet service which has up-to-date statistics as to the number of users. Blockchain.info had a similar number of users to Coinbase in February 2014 and currently boasts 2.9 million users of its wallet service [17]. From this data it is plausible that the Coinbase wallet service has a customer base of between 1 and 3 million wallets. Each entry in the customer set consists of a customer identifier and a value. We propose that the customer identifier be in the form of a hash digest, which has the advantage that it prevents any customer-identifying information from being gleaned from the customer. The choice of hashing algorithm is not of consequence, for arguments sake we propose the SHA-256 algorithm, which produces 32-byte digests. The value field is also chosen as an 8-byte unsigned integer. Assuming approximately 3 million users, the total memory requirement for the customer data is approximately 120MB, also significantly larger than the 116KB of input space.

4.2 Architecture

It is clear from the memory requirements posed by the input data set sizes and the available input sizes of the Flicker platform that the monolithic architecture of the audit as proposed in Figure 3.1 must be broken into smaller components in order for the input data to be split into input-sized chunks and processed in an incremental fashion. This architectureal change does not change the result of the audit, however the calculation of the outputs which are required to verify the input data must change as a result of the components only having a view of a small subset of the input data in each iteration.

We propose breaking the audit process into three separate components which are each invoked multiple times in their own trusted execution environment, each invocation with some subset of the input data. The components are the same three as in Chapter 3: Proof of Reserves, Proof of Liabilities and Proof of Solvency. The fact that each component is invoked individually from the user application and that intermediate data is passed between iterations and components introduces new potential points of attack to the application.

The individual invocations of a component of the audit require a secure method of storing intermediate values, for instance a sum which is calculated over multiple iterations. The TPM *seal* functionality provides a perfect mechanism for doing so. The PAL can use the TPM to seal intermediate values to the current PCR state, encrypting them such that they can only be decrypted by the TPM when it is executing the same PAL. The encrypted blob is passed back to the user application which should provide it as an input to the next iteration of the component.

As the encrypted blobs are passed back to the user application, which is executing in an untrusted and potentially malicious environment, there is the potential for a replay attack to be performed on the encrypted blob. Fortunately the process of hashing the input ensures that replay attacks can be detected by the client when verifying the result of the audit.

4.2.1 Proof of Reserves

In the monolithic audit presented in Chapter 3, the entire UTXO set is input to the Proof of Reserves (POR) component, the hash of which is required to verify the signatures of the public Bitcoin addresses. Although it would be possible to split the POR into two stages: one which determines the UTXO hash, and one which verifies the signatures and sums the UTXO balances, this would require iterating over the entire UTXO input twice: once to determine the hash, and once to determine the sum. Due to the overhead involved in invoking a Flicker session it is sensible for the user application to calculate the hash of the UTXO set and provide it to the POR. As the POR component sees all UTXO inputs throughout the lifetime of its invocations, it can calculate the hash of the UTXO values that it sees, which can be compared against the hash that was provided by the user application to ensure that they are identical.

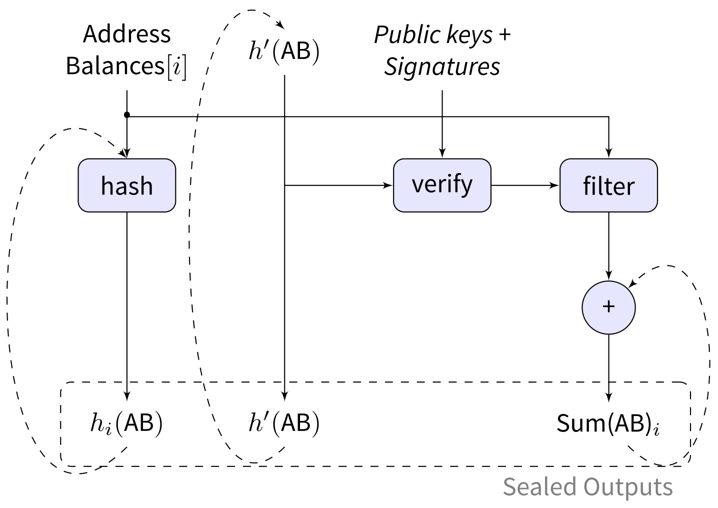

Figure 4.1: An overview of the proof of reserves component.
Italicised values are not published publicly.
Dashed arrows indicate values which are passed from invocation to invocation

In order for the above scheme to work, the POR component must be able to calculate the hash over all UTXO elements despite the fact that it only has access to a subset of UTXO elements which are input to it in that iteration. For this purpose we redefine the UTXO hash as a chain of hashes of the UTXO elements, similar to the approach used to extend the TPM's PCRs. The hash chain begins with the hash of the first element of the UTXO set:

$$H_0 = h(\text{UTXO}[0])$$

Where UTXO$[0]$ is the first UTXO entry and h is a suitable cryptographic hash function. The hash chain up to the ith element is calculated as:

$$H_i = h(H_{i-1} \mid\mid \text{UTXO}[i])$$

The overhead in the invocation of a Flicker session motivates the reduction of the input size wherever possible, in order to reduce the number of Flicker sessions that must be invoked. The UTXO set is a somewhat inefficient representation of the data of interest: it is possible that there are multiple UTXOs which refer to the same public Bitcoin address. These can be aggregated to form a set of *address balances* (AB), a list in which each Bitcoin address appears once with the number of bitcoins which can be retrieved from that address. The 17 million UTXO entries can be reduced to approximately 3.5 million address balances, close to a 5x reduction in entries to be evaluated.

Figure 4.1 shows an overview of the new POR component. For the initial invocation of the POR the user application passes the first block of address balances, the address balances hash (h'(AB)), and the public keys and signatures to the POR. The POR calculates its own hash chain of the address balances it received in this invocation (h_i(AB)), it uses the hash chain of all address balances provided by the user application to verify the public key signatures. With the public key signatures it calculates the sum of the address balances provided in this invocation (Sum(AB)$_i$). These two intermediate output values are sealed together with the hash h'(AB), to be fed into the next invocation of the POR. On the next invocation of the POR, the next block of address balances is passed to the POR with the public keys and signatures, together with the sealed blob which contains the outputs of the previous invocation, the values which are fed back into the calculation are indicated in Figure 4.1 by dashed arrows.

4.2.2 Proof of Liabilities

The Proof of Liabilities (POL) is invoked iteratively, similarly to the Proof of Reserves. Figure 4.2 shows an overview of the POL component, notably missing from this diagram when compared to Figure 3.1 is the *Fraction factor* which has been moved into the Proof of Solvency component. Additionally the path for the calculation of the Merkle tree root has been modified with the addition of the *fold* unit. The reason for this modification is that an invocation of the POL does not have a view of the whole customer data, so the calculation of the merkle root must be performed iteratively. The fold operator is an important component in the ability to iteratively compute the Merkle tree.

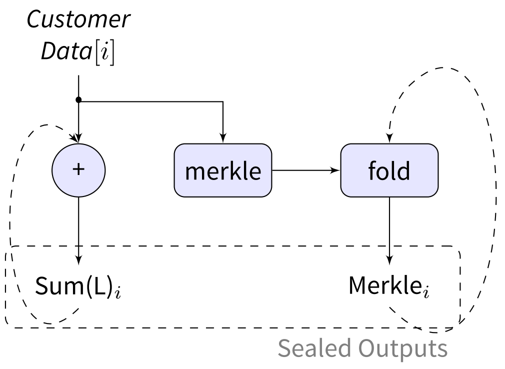

Figure 4.2: An overview of the proof of liabilities component.
Italicised values are not published publicly.
Dashed arrows indicate values which are passed from invocation to invocation

The output Sum(L)$_i$ contains the sum of the customer balances up to and including block i. Merkle$_i$ contains a stack of tuples which represent the current progress in the calculation of the root of the Merkle tree.

Iterative Merkle

Figure 4.3 shows a compacted binary Merkle tree, all of the child nodes of C, C′ and B′ are hidden. The tree is constructed such that there are 2^n leaf nodes below C and C′, and somewhere between 1 and 2^{n+1} leaf nodes below B′.

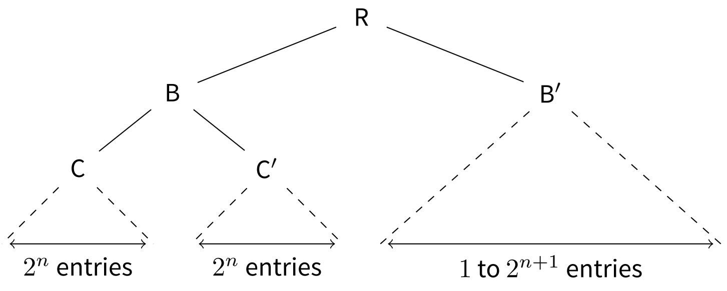

Figure 4.3: Merkle tree

One property of the Merkle tree root calculation is that once the values in a subtree have been evaluated, they are no longer needed. In this concrete case, once the value of C is calculated, its child nodes are no longer needed for any further calculation. Also of note is that the root node of a subtree of R is the root of the Merkle tree of its 2^n leaf nodes. The implication of this is that the root of the merkle tree of a 2^n sized and aligned subset of the input data is a node in the merkle tree of the full set of input data.

The above properties of the Merkle tree allow for a simple, low memory-overhead, stack-based iterative implementation of the calculation of the root of a Merkle tree. Referring again to Figure 4.3, we consider the nodes C and C′ to be at height 0, nodes B and B′ to be at height 1, and node R to be at height 2. The POL receives the first block of 2^n entries and calculates the merkle root of this data (node C in Figure 3.2), passing the tuple

$$\langle 1, \text{merkleroot}(\text{block}[0]) \rangle$$

to the fold stage, which places the value on the stack and attempts to reduce the values on the stack. As there is only one value on the stack, no reduction is possible. The stack is then sealed and passed to the user application. Next, the POL receives the next block of 2^n entries, and the sealed stack from the previous iteration. It calculates the merkle root of the data (node C′ in Figure 3.2 and passes the tuple

$$\langle 1, \text{merkleroot}(\text{block}[1]) \rangle$$

to the fold stage, which places the tuple on the stack and attempts to reduce the values on the stack. The fold pops both tuples off the stack, concatenates their hashes and calculates the hash of the concatenation (node B in Figure 3.2), placing the tuple

$$\langle 2, h(\text{merkleroot}(\text{block}[0]) \parallel \text{merkleroot}(\text{block}[1])) \rangle$$

onto the stack. The value of the tuple on the stack evaluates to the value of node B in Figure 3.2, the index of the tuple is $2^{\text{height(B)}}$. The fold operation is named as such because it folds two child nodes of the same height into the parent node.

The example above shows that the values in the stack which are passed from one iteration of the POL to another contain the progress made in the calculation of the overall Merkle root. Each invocation of the POL need not know which block of data it is operating on, as long as the block size is fixed to a power of two, and the blocks are fed sequentially to iterations of the POL. The stack holds all the information required to continue calculating the merkle root. If the customer data does not consist of 2^n entries the last invocation of the POL will produce an output with a stack with multiple values on it which still need to be processed. This processing is done in the Proof of Solvency component.

As the sealed blob may not be arbitrarily large, it is of interest to know what the maximum number of values that need to be stored on the stack is. The stack is at its largest when the last block of data is being processed, in which there is one entry for the left half of the merkle tree, one entry for the left half of the right subtree and so on. This is logarithmic in the number of blocks of data that the input is split into. The input size of the POL is approximately 116KB, each customer account entry requires 40 bytes and the block input to the POL should contain 2^n entries. An input block size of 2048 entries uses approximately 70% of the input to the POL. Assuming 4 million (2^{22}) customer accounts, and that a block of customer accounts contains 2048 (2^{11}) entries there will 2^{11} blocks of customer data. For an input size of four million customer accounts the maximum stack size required is 11 entries, which is minimal overhead given that the unused input space to the POL could accomodate a stack of a few hundred entries..

4.2.3 Proof of Solvency

The Proof of Solvency (POS) component takes as inputs the sealed blobs from the Proof of Reserves (POR) and Proof of Liabilities (POS) components as well as the Fraction factor. It verifies that the hash of the address balances provided by the user application is correct, determines the solvency of the exchange and processes the values remaining on the Merkle stack to determine the Merkle root. The outputs are similar to those presented in Chapter 3, instead of producing a hash of the entire UTXO set, the chained hash of the address balances is ouput.

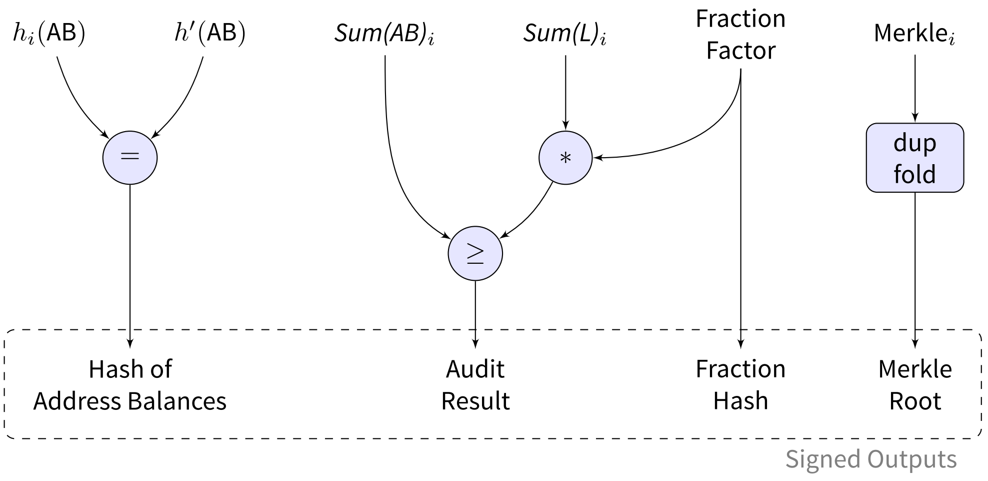

Figure 4.4: An overview of the proof of solvency component.

The only stage of this component which has not yet been previously introduced is the *dup fold* operator. The dup fold operator is used to process the remaining stack entries which were passed from the last invocation of the POL component. If the stack has more than one element, the dup fold pops the top tuple off of the stack, duplicates it and folds these two tuples into one, which it places on the stack. It continues this until the stack size is one, at which point the top tuple contains the root of the Merkle tree, which can be output.

The final component of the POS component is the attestation of the PAL binary. The audit no longer consists of an individual invocation of a PAL, instead the POR and POL components are invoked hundreds or thousands of times each of these invocations requires attestation. The solution to this problem is to put the separate logic for the POR, POL and POS components into a single binary. The initial invocations of both the POR and POL logic of the PAL produces a sealed blob which is tied to that PAL. The sealed blob is then unsealed by the next invocation, the intermediate values are modified and then resealed. When the POS is invoked, it unseals the blobs produced by the POR and POL.

The fact that the sealed blobs are unsealed and modified in each invocation of the POR and POL and that they can only be unsealed by the same PAL that initially created them means that the values in the sealed blob form a chain of trust from their respective first invocations until the invocation of the POS. An attesation of the POS is transitive to all previous PAL executions which were able to unseal the blobs that the POS unseals.

4.3 Execution time

As previously mentioned, the Flicker invocation and some TPM operations pose an overhead of up to 1s which is somewhat significant. During this execution time, the operating system on which the Flicker session is invoked does not process any interrupts. When the Flicker session ends, the operating system requires a small amount of time to process any interrupts and respond to system events. Tests showed that the operating system needs pauses of 500ms to 1s in order to continue processing without locking up or crashing. As the processing time for such a small number of inputs is quite low in comparison to the TPM overhead, we can safely assume that each Flicker invocation costs approximately two seconds.

For input sizes in the range previously discussed, 4 million address balances and 4 million customer accounts, the POR must be invoked approximately 1400 times (each invocation of the POR can process approximately 3000 address balances), the POL must be invoked approximately 2000 times. This comes to a total of 3400 invocations, each of which requires 2s to execute and wait for the operating system to recover. The overall execution time for an audit with inputs of this size is approximately two hours and scales linearly in the number of address balances and customer accounts. As the TPM must be made available to respond to requests as part of the Direct Anonymous Attestation (DAA) scheme, we propose that the audit could be conducted once daily, after which the platform with the TPM responds to DAA requests until it begins calculating the next audit, during which time DAA requests cannot be processed.

4.4 Additional Interfaces

Although the audit is the core component of proving solvency of a Bitcoin exchange, the signed audit output is not all that a customer requires in order to verify the audit. Customers must retrieve additional values from the exchange and perform some local computations in order to be able to verify the audit, and to have some form of recourse should the verification fail. The implementation of these interfaces is not in the scope of this thesis, what follows is an outline of the requirements of the peripheral software and interfaces.

4.4.1 Audit Verification

Most important for customers is the ability to verify the audit's result. This consists of the verification that the customer's balance was included in the calculation, verification of the address balances, and verification of the attestation. The customer of an exchange must be able to retrieve the nodes in the Merkle tree which can be used to calculate the path from the customer's leaf node to the root of the Merkle tree. If the customer is able to reproduce the Merkle root using the nodes provided by the exchange and their own customer identifier and account balance at the time of the audit, then they can be assured that they were accounted for in the calculation. The interface for this purpose must take the hash of the tuple

$$\langle \text{Customer Identifier}, \text{Balance} \rangle$$

as an argument and deliver the set of nodes required to calculate the path from the customer's leaf node to the Merkle root. Each node would consist of a tuple

$$\langle \text{Height}, \text{Hash} \rangle$$

Where *Height* is the height of the node in the Merkle tree and *Hash* is the hash digest stored at that node in the tree. The customer must also be able to verify that the hash of the address balances provided by the exchange represents the true account balances for a set blockchain height. For this purpose, the customer must be able to determine the blockchain height that was used to determine the address balances. The customer would require a software client which can determine the address balances for the blockchain at a given height. This consists of: extraction and aggregation of UTXOs, and sorting of the address balances. With the address balances calculated, the customer can calculate the hash and compare it with the hash provided by the audit. Finally, the customer must be able to verify the attestation. This consists of two components: verification that the attestation originates from a TPM, and verification of the binary which was executed in the trusted platform.

Attestation Verification
The customer needs to be able to verify that the attestation was indeed issued by a TPM, in other words, what the customer needs to know is that the Attestation Identity Key (AIK) used to sign the attestion was provided by a TPM. The method proposed by the TCG is Direct Anonymous Attestation (DAA) which allows for a customer to verify directly that an AIK belongs to a TPM. For this the exchange must provide an interface which performs DAA and the customer requires client software which can verify the DAA provided by the exchange.

Binary Verification
In order for the customer to verify that the PAL executed in the trusted environment actually calculates the audit, as opposed to always returning true, the customer must have access to the source code of the PAL and be able to reproduce the value of PCR 17 which is signed in the TPM's attestation. The exchange needs to provide a platform from which the PAL source code can be retrieved, as well as a method for compiling a reproducible binary, and instructions on how to transform the hash of the binary to the value of PCR 17 in the attestation.

4.4.2 Signed Account Balance

If a customer should determine that their account balance was not included in an audit, they require some form of proof that their account balances ought to have been taken into account in the audit. For this purpose the exchange should provide an interface which allows a customer to retrieve a signature of the hash of their ⟨CustomrID, balance⟩ tuple. With this signature, other customers or the community at large could verify that the exchange signed a value which is not included in the latest audit.

5 Conclusion

A string of Bitcoin exchange closures as well as various thefts from Bitcoin exchanges have left customers of exchange services somewhat hesitant as there has often been little transparency when such events took place. Exchanges have publicly published customer account balances as well as proof of ownership of Bitcoin addressed which allow for customers and the public to determine the Bitcoin assets of the exchange.

This thesis proposed using an automated software-based audit to determine the solvency of Bitcoin exchanges without revealing any private data. Methods are proposed, based on the Flicker Trusted Computing platform, with which the audit result can be verified and trusted to be correct. An architecture is proposed which allows for the computation to be split into individual pieces which iteratively compute a subset of the complete input to overcome the memory limitations posed by the Flicker platform. The verification methodology is expanded to cover the iterative execution scenario, allowing for customers of an exchange to verify the inputs to the audit. An analysis of the execution time showed that it is entirely feasible to conduct audits on a daily basis at the current estimate size of the Bitcoin ecosystem.

A
List of Acronyms

ACM	Authenticated Code Module
AML	Anti-Money Laundering
AIK	Attestation Identity Key
CNY	Chinese Yuan
CPU	Central Processing Unit
DAA	Direct Anonymous Attestation
DRTM	Dynamic Root of Trust for Measurement
ECDSA	Elliptic Curve Digital Signature Algorithm
EK	Endorsement Key
EUR	European Union Euro
OS	Operating System
PAL	Piece of Application Logic
POL	Proof of Liabilities
POR	Proof of Reserves
POS	Proof of Solvency
PCR	Platform Configuration Register
SHA	Secure Hash Algorithm
SRK	Storage Root Key
SRTM	Statis Root of Trust for Measurement
SVM	Secure Virtual Machine
TC	Trusted Computing
TCB	Trusted Computing Base
TCG	Trusted Computing Group
TPM	Trusted Platform Module
TXT	Trusted eXecution Technology
USD	United States Dollar
UTXO	Unspent Transaction Output

Bibliography

[1] S. Nakamoto, “Bitcoin: A peer-to-peer electronic cash system,” https://bitcoin.org/bitcoin.pdf, 2008 (Online; accessed February 3rd, 2015).

[2] T. Moore and N. Christin, “Beware the middleman: Empirical analysis of bitcoin-exchange risk,” in *Financial Cryptography and Data Security*. Springer, 2013, pp. 25–33.

[3] C. Decker and R. Wattenhofer, “Bitcoin transaction malleability and MtGox,” in *19th European Symposium on Research in Computer Security (ESORICS), Wroclaw, Poland*, September 2014.

[4] H. Finney, “bcflick - using TPM’s and trusted computing to strengthen bitcoin wallets,” https://bitcointalk.org/index.php?topic=154290.msg1635481#msg1635481, 2013 (Online; accessed February 9th, 2015).

[5] G. Andresen, “Bitcoin improvement proposal 11: M-of-N standard transactions,” https://github.com/bitcoin/bips/blob/master/bip-0011.mediawiki, 2011 (Online; accessed February 9th, 2015).

[6] M. Araoz, R. X. Charles, and M. A. Garcia, “Bitcoin improvement proposal 45: Structure for deterministic P2SH multisignature wallets,” https://github.com/bitcoin/bips/blob/master/bip-0045.mediawiki, 2014 (Online; accessed February 9th, 2015).

[7] D. Bryans, “Bitcoin and money laundering: mining for an effective solution,” *Ind. LJ*, vol. 89, p. 441, 2014.

[8] N. Y. S. D. of Financial Services, “Regulations of the superintendent of financial services part 200. virtual currencies,” http://www.dfs.ny.gov/legal/regulations/revised_vc_regulation.pdf, 2015 (Online; accessed February 9th, 2015).

[9] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, “Innovative instructions and software model for isolated execution,” in *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*. ACM, 2013, pp. 1–1.

[10] *TCG Specification Architecture Overview*, Trusted Computing Group, August 2007, rev. 1.4.

[11] E. Brickell, J. Camenisch, and L. Chen, “Direct anonymous attestation,” in *Proceedings of the 11th ACM conference on Computer and communications security*. ACM, 2004, pp. 132–145.

[12] *Intel Trusted Execution Technology Software Developers Guide*, Intel Corporation, May 2014.

[13] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki, “Flicker: An execution infrastructure for TCB minimization,” in *ACM SIGOPS Operating Systems Review*, vol. 42, no. 4. ACM, 2008, pp. 315–328.

[14] R. C. Merkle, "A digital signature based on a conventional encryption function," in *Advances in Cryptology-CRYPTO'87*. Springer, 1988, pp. 369–378.

[15] G. Maxwell and Z. Wilcox, "Proving your bitcoin reserves," https://iwilcox.me.uk/2014/proving-bitcoin-reserves, February 2014 (Online; accessed January 5th, 2015).

[16] Coinbase, "A major coinbase milestone: 1 million consumer wallets," http://blog.coinbase.com/post/78016535692/a-major-coinbase-milestone-1-million-consumer, 2014 (Online; accessed February 7th, 2015).

[17] Blockchain.info, "My wallet number of users," https://blockchain.info/charts/my-wallet-n-users, 2014 (Online; accessed February 7th, 2015).