Aalto University
School of Science
Degree Programme in Security and Mobile Computing

Rakesh Gopinath Nirmala

Improving the Security and Efficiency of Blockchain-based Cryptocurrencies

Master's Thesis
Espoo, June 30, 2017

Supervisors: Professor N.Asokan, Aalto University
Professor Panos Papadimitratos, KTH Royal Institute of Technology
Advisor: Dr. Andrew Paverd, Aalto University

Aalto University
School of Science
Degree Programme in Security and Mobile Computing

ABSTRACT OF MASTER'S THESIS

Author: Rakesh Gopinath Nirmala

Title:
Improving the Security and Efficiency of Blockchain-based Cryptocurrencies

Date:	June 30, 2017	**Pages:**	77
Major:	Security and Mobile Computing	**Code:**	T3011

Supervisors: Professor N.Asokan, Aalto University
Professor Panos Papadimitratos, KTH Royal Institute of Technology

Advisor: Dr. Andrew Paverd, Aalto University

In recent years, the desire for financial privacy and anonymity spurred the growth of electronic cash and cryptocurrencies. The introduction of decentralized cryptocurrencies, such as Bitcoin, accelerated their adoption in society. Since digital information is easier to reproduce, digital currencies are vulnerable to be spent more than once – this is called a *double-spending attack*. In order to prevent double-spending, Bitcoin records transactions in a tamper-resilient shared ledger called the *blockchain*. However, the time required to generate new blocks in the blockchain causes a delay in the transaction confirmation. This delay, typically around one hour in Bitcoin, is impractical for real world trade and limits the wide-spread use of blockchain-based cryptocurrencies.

In this thesis, we propose a solution to prevent double-spending attacks and thus enable fast transaction confirmations using the security guarantees of Trusted Execution Environments (TEEs). We achieve this by enforcing *sign-once semantics* that prevent the payer from reusing designated signing keys to sign more than one transaction. We also provide a way for the payee to *verify* whether a specific signing key is subject to sign-once semantics. The payee, however still receives the funds later, once the transaction is verified similarly to existing credit card payments. In this way, our solution reduces transaction confirmation times of blockchain-based cryptocurrencies and is also compatible with existing deployments since it does not require any modifications to the base protocol, peers, or miners. We designed and implemented a proof-of-concept of our solution using Intel SGX technology and integrated it with Copay, a popular Bitcoin wallet from BitPay. This thesis also presents the security evaluation of our system along with other possible extensions and enhancements.

Keywords: Cryptocurrency, Bitcoin, Blockchain, Double-Spending, Trusted Hardware, SGX

Language: English

Acknowledgements

> **"When you practice gratefulness, there is a sense of respect towards others"**
>
> *The Dalai Lama*

The thesis was carried out in **Secure Systems Group** at Aalto University, Finland under the supervision of **Prof. N. Asokan** (Aalto), **Prof. Panos Papadimitratos** (KTH Royal Institute of Technology) and advised by **Dr. Andrew Paverd** (Aalto). It was part of my Erasmus Mundus Masters program (NordSecMob).

First and foremost, I express my sincere gratitude from the bottom of my heart to Prof. Asokan and Dr. Paverd for their invaluable guidance and encouragement throughout. Many thanks for patiently helping me choose the right direction and extending the moral support through my difficult times. I also thank Prof. Papadimitratos for his precious time and support as a co-supervisor.

I am greatly indebted to my family and my uncle for their unconditional love and firm belief in my decisions. This accomplishment would have been impossible without your constant support and encouragement. Also, I thank all my friends for bearing me and caring for my well being.

I am also very grateful to the European Union for funding my Masters studies and sowing the seed for a new beginning in my life. I deeply thank my friend Bhanu Teja Kotte for introducing me to this program. This has been a wonderful journey with exciting research and lots of learning. I convey my warmest regards to everyone who helped me in my way.

Espoo, June 30, 2017

Rakesh Gopinath Nirmala

Abbreviations and Acronyms

BTC	bitcoin
tps	Transactions Per Second
TEE	Trusted Execution Environment
SGX	Software Guard Extensions
ECDSA	Elliptic Curve Digital Signature Algorithm
UTXO	Unspent Transaction Output
P2PKH	Pay To Public Key Hash
P2PK	Pay To Public Key
P2SH	Pay To Script Hash
QR code	Quick Response code
PoW	Proof of Work
JBOK	Just a Bunch Of Keys
HD protocol	Hierarchical Deterministic protocol
BIP	Bitcoin Improvement Protocol
PBKDF	Password-Based Key Derivation Function
CKD	Child Key Derivation
PRM	Processor Reserved Memory
EPC	Enclave Page Cache
EPCM	Enclave Page Cache Metadata
TCB	Trusted Computing Base

EPID	Enhanced Privacy ID
IAS	Intel Attestation Service
API	Application Programming Interface
EDL	Enclave Definition Language
ECALL	Enclave Call
OCALL	Output Call
SLOC-L	Source Lines of Code - Logical
DMA	Direct Memory Access

Contents

Abbreviations and Acronyms **4**

1 Introduction **8**
- 1.1 Problem Statement . . . 9
- 1.2 Thesis Goals and Scope . . . 9
- 1.3 Solution Overview . . . 10
- 1.4 Structure of the Thesis . . . 10

2 Cryptocurrency and Trusted Hardware **12**
- 2.1 Cryptocurrency . . . 12
  - 2.1.1 Bitcoin Overview . . . 13
  - 2.1.2 Transaction . . . 13
  - 2.1.3 Bitcoin Addresses and Scripts . . . 16
  - 2.1.4 Spending and Key Reusage . . . 18
  - 2.1.5 Blockchain . . . 20
  - 2.1.6 Wallets . . . 23
  - 2.1.7 Hierarchical Deterministic Wallets . . . 26
  - 2.1.8 Double-Spending Attack . . . 31
- 2.2 Trusted Execution Environment - Intel Software Guard Extensions (SGX) . . . 32
  - 2.2.1 Enclave . . . 32
  - 2.2.2 Sealing and Replay Protection . . . 33
  - 2.2.3 Attestation . . . 34
- 2.3 Related Research . . . 35

3 Adversary Model and Requirements **37**
- 3.1 Adversary Model . . . 37
  - 3.1.1 Adversary: Goals and Motivation . . . 38
  - 3.1.2 Capabilities . . . 39
  - 3.1.3 Assumptions . . . 40
- 3.2 Security Requirements . . . 40

3.3 Usability Requirements 41
3.4 Deployability Requirements 41

4 Solution Design and Architecture 42
4.1 Design Overview 42
4.2 Functional Overview 42
4.2.1 Transaction Manager Component 43
4.2.2 Enclave Bridge Component 45
4.2.3 SGX Enclave Library 45

5 Implementation 51
5.1 Implementation Approach 51
5.2 Transaction Manager and Copay 52
5.3 SGX Enclave Library 53
5.3.1 Untrusted Module 53
5.3.2 Trusted Enclave Module 56
5.4 Integration Overview 60

6 Evaluation 61
6.1 Requirements Conformance 61
6.1.1 Security Conformance 61
6.1.2 Usability Conformance 62
6.1.3 Deployability Conformance 62
6.2 Attacks and Countermeasures 63
6.3 Side Channel Resistance 63
6.4 Minimizing Trusted Computing Base 65
6.5 Limitations 65
6.5.1 Resilience to Crashes 65
6.5.2 Backup and Recovery 66

7 Variations and Extensions 67
7.1 Multiple Accounts 67
7.2 Remote Attestation Verification 67
7.3 Bloom Filters for Storage 68
7.4 Backup and Recovery 68

8 Conclusions and Future Work 70
8.1 Future Work 71

Chapter 1

Introduction

In recent years, cryptocurrencies have gained huge popularity over the globe. They have been the subject of research since the proposal of blind signatures for untraceable payments [16] by David Chaum in 1983. The idea of anonymous payments became popular with the inception of DigiCash formulated by Chaum in 1990. Though there has been a steady rise of interest since then, the introduction of *Bitcoin* [37] in 2008 garnered significant attention and led to the growth of research and investments in this field. The major features that contributed to the success of Bitcoin are its decentralized peer-to-peer control and a secure and transparent public ledger, known as the *blockchain*.

The introduction of Bitcoin fueled the growth of cryptocurrencies and since then over 800 cryptocurrencies have been released into the market with the market capitalization of nearly 88 billion USD in total [17]. However, the market price of *bitcoin*[1] (BTC) has been highly volatile and fluctuating since its inception, partly because it is relatively new and is yet to be recognized widely unlike standard traditional currencies. By June 2017, one bitcoin was equivalent to 2417 USD [7]. As far as day-to-day transactions are concerned, bitcoins are being accepted as modes of payment at many places of purchase and the numbers are increasing.

Blockchain technology is perceived to be a major breakthrough innovation primarily due to its decentralized control and strong cryptographic guarantees. At present, several organizations are considering how to use this technology for their applications. For example, major financial institutions such as JPMorgan, Citigroup and Chase are investing in blockchain in order to widen their growth prospects [47]. In addition, many governmental institutions such as the European Central Bank, US Federal Reserve and

[1]Bitcoin (B capitalized) represents the digital currency protocol and bitcoin (b non-capitalized) represents the unit of the virtual currency.

UK Treasury have acknowledged and vowed to invest in this technology [52]. More recently, the Indian government also set up an interdisciplinary committee to study this virtual currency framework and its potential impact [28].

1.1 Problem Statement

Unlike physical currencies, digital currencies are vulnerable to be spent more than once by the same spender because it is easier to reproduce digital information, such as digital signatures. This attack is called a *double-spending* attack. In order to prevent such double-spending, a digital transaction has to be confirmed. The decentralized cryptocurrencies, such as Bitcoin, record transactions in an integrity-protected, shared public ledger called *blockchain*. These transactions are packed together in blocks. In blockchain, a new block is generated every 10 minutes [37] causing a delay in the confirmation of transactions. Currently, it is recommended that the blockchain-based cryptocurrency payments must be verified by waiting till the corresponding transaction is embedded in the blockchain. In case of Bitcoin, this waiting time is close to 60 minutes [43]. This long confirmation delay is undesirable for practical trading purposes and is limiting the wide-spread usage of cryptocurrencies. Given that a transaction cannot be modified once recorded in the blockchain, this mechanism prevents double-spending after confirmation. However, there is no defined solution to prevent a spender from spending the same bitcoins again without waiting for the confirmation.

For example, imagine a customer goes to a coffee shop, buys a coffee and pays the merchant using bitcoins. In this case, the merchant should either take the potential risk of double-spending by the customer, or should make him/her wait until that particular transaction is included in the blockchain. Existing solutions such as dynamic transaction fees depending on transaction value, and blockchain accepting the first-seen transaction do not completely prevent this problem. This limits the practical usage of cryptocurrencies for real trade.

1.2 Thesis Goals and Scope

The major *aim* of this thesis is to thoroughly study and analyze the current Bitcoin scheme of operations and build a system that prevents double-spending attacks. We intend to achieve the following:

1. Propose and design a technique for speeding up cryptocurrency pay-

ments by reducing the transaction time without compromising the security of the underlying ecosystem.

2. Propose and design a technique to provide a verifiable guarantee of payment to the payee.

3. Implement and evaluate the prototype of the proposed design for its functionality and security guarantees.

The *scope* of this thesis is limited to mitigating double-spending attacks. It does not deal with any other attacks possible in the Bitcoin ecosystem.

1.3 Solution Overview

Ideally, we need a cryptocurrency mode of payment that is resistant to double-spending attacks and also eliminates the long waiting time associated with the confirmation of transactions as explained in Section 1.1. In this thesis, we build a secure cryptocurrency wallet that prevents reusing the same key for signing more than one transaction (multiple signatures using the same key). This mechanism is referred to as *sign-once semantics* and ensures the payer cannot double-spend. The wallet also shares a guarantee that the payee can verify to check whether the specific key has been subjected to sign-once semantics. In order to achieve the desired secure functionality, we use a Trusted Execution Environment (TEE). Section 2.2 explains TEE in more detail. Unlike previous solutions, our approach does not require any changes to the Bitcoin protocol or Bitcoin nodes and miners.

Lets consider the coffee shop scenario where Alice uses a wallet integrated with our solution. First, she transacts a specific amount of bitcoins to the merchant. Second, she shares a verifiable guarantee via Quick Response (QR) code with the merchant. Finally, the merchant accepts or rejects the payment made by Alice after verification.

1.4 Structure of the Thesis

The rest of the thesis is organized into the following chapters. Chapter 2 provides background on Bitcoin and the Bitcoin wallets in practice. It also introduces Intel SGX technology and explains its features. Chapter 3 defines the adversaries in our system, their capabilities, and our assumptions. Also, it lists the set of functional and security requirements that guide our design. Chapter 4 describes the proposed solution, components and their

functionalities, and depicts the sequence of operations and dataflow between them. Chapter 5 discusses the prototype implementation and integration strategies. It also describes the algorithmic details and the decisions taken. Chapter 6 explains the evaluation of the proposed design and implementation with respect to its functionality, security and performance. Chapter 7 discusses possible improvements and enhancements that can further increase the efficiency of the proposed design. Finally, chapter 8 lists future work and concludes the thesis.

Chapter 2

Background: Cryptocurrency and Trusted Hardware

In this chapter, we describe the topics that provide necessary background knowledge on the various concepts associated with cryptocurrency, Bitcoin wallets and trusted hardware. We also discuss the related research work.

2.1 Cryptocurrency

Cryptocurrency is the new age digital currency that serves as the medium of exchange between the transacting parties. It relies on the cryptographic primitives to secure, validate and regulate the creation and transaction of the currency. Unlike traditional currencies, cryptocurrency obviates the need for central trusted third parties such as banks, to track and record transactions.

The notion of decentralized cryptocurrency is believed to be formulated by Wei Dai in 1998 when he proposed his B-money [20] protocol based on asymmetric cryptographic primitives. Wei Dai described B-money as a distributed and anonymous currency exchange protocol between entities identified by untraceable digital pseudonyms. In 2005, Nick Szabo propounded a new version of cryptocurrency, Bit gold [46]. Bit gold is based on the idea of solving the cryptographic puzzle where clients would compute a bit string from a given string of challenge bits. The resultant bit string then serves as a challenge for the next round and the successful clients are rewarded. The clients are identified by their public keys, digital signatures and timestamps. Though Bit gold solved the problem of decentralization of payments and anonymity, it failed to garner much attention primarily due to the following reasons: There was no defined way to value different measures of work and also no solution proposed for its adoption in the society [40].

2.1.1 Bitcoin Overview

Bitcoin [37] is the first widely known decentralized cryptocurrency scheme [40] publicized under an unknown pseudonym, Satoshi Nakamoto, as a peer-to-peer electronic cash system in 2008. Bitcoin has its own metric of exchange called bitcoin abbreviated as BTC. Unlike traditional currency, the virtual currency has no intrinsic value and is not controlled by any central authority. It uses asymmetric cryptography, peer-to-peer networking and consensus to validate and execute payments and transfers.

The complex scheme of Bitcoin solely depends on the operation of the distributed network of peers competing to solve a resource-intensive cryptographic computation or *Proof-of-Work* (PoW). This process is called *mining* and is responsible for generating the blocks that pack together a set of incoming client transactions broadcast to the network. The client, here, refers to the various payers and payees using the bitcoins as their financial medium of exchange. The miners who mine a valid block are rewarded with a certain amount of bitcoins that can be spent in the future. In addition, they also get incentives for verifying and recording the transactions in the block. In this way, the security of the network depends on the collective control of computational power within the Bitcoin ecosystem and the system remains secure while the honest peers control more than half of the network [1]. The following segments explain several key structures involved in the overall operation of Bitcoin.

2.1.2 Transaction

Transaction is the heart of the Bitcoin scheme. All other things are the supporting structures built around it to ensure its smooth creation and operation. A Bitcoin transaction is a data structure that encodes the transfer of a defined sum of value between the exchanging parties. It is analogous to a traditional currency transaction used in real-world trading. This section describes the lifecycle of a transaction, its contents and the signing and verification operations associated with it.

Transaction Lifecycle - The lifecycle of a transaction begins with its creation. Every transaction contains one or more inputs and outputs. It is then signed with the required number of signatures and broadcast to the Bitcoin peer network. The peers then validate the transaction and propagate it to other peers. Finally, it is verified and recorded in the blockchain by the mining peer. Over time, the stored transaction is confirmed by the other peers in the network, thus, making it valid and permanent in the blockchain.

In order to create and record a valid transaction, each transacting party

owns a set of public-private key pairs. The private keys are used for digital signing of the transaction while the public keys are used for deriving the Bitcoin addresses to be shared with others. As described in [22], the key derivation is based on *Elliptic Curve Cryptography* [35] using the secp256k1 curve and the digital signature is generated using the Elliptic Curve Digital Signature Algorithm (ECDSA). A Bitcoin address is a base58-encoded string of the hashed public key, equivalent to an account number in the existing banking systems. By using the hashed public keys as addresses, user anonymity is partially protected and the digital signature verification authorizes the owner of the bitcoins. In addition, hashing also shortens and obfuscates the plain public key.

Transaction Structure - A typical transaction consists of a version number, locktime and at least one input and one output as illustrated in Figure 2.1. Each input and output further consists of different parameters.

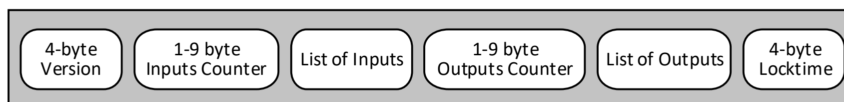

Figure 2.1: A simple Bitcoin Transaction

Version Number - Bitcoin peers and miners use the 4-byte version number to apply the specific set of rules to validate the transaction. This number is incremented every time a new set of rules are proposed.

Locktime - is a 4-byte unsigned integer that decides when a particular transaction is included in the blockchain. Locktime allows one to create time-locked transactions that would become valid only after the specified interval of time. In case of change-of-mind, a transacting party could create a new transaction with zero locktime that uses the same inputs of a time-locked transaction along with other inputs. The new transaction would now invalidate the previous time-locked transaction provided that it is included in the blockchain ahead of its locktime expiry.

Transaction Input - A transaction input is a reference to an output resulting from a previous valid transaction [22]. For a transaction to be valid, it must contain at least one input. However, multiple inputs are often included in the transaction and the sum of all the inputs decides the output value. Each transaction input in itself is a collection of parameters. Table 2.1 lists the input fields within a transaction.

The combination of transaction hash, `Previous tx_hash`, and the output index value, `Previous out_index`, uniquely identifies the previous transaction with an unspent output embedded in the blockchain. The input also

Fields	Size (bytes)
Previous tx_hash	32
Previous out_index	4
Txin-scriptLength	1-9
Txin-scriptSig	<Txin-scriptLength>
sequence_number	4

Table 2.1: Contents of a Transaction Input

contains a signature script that is responsible for storing the digital signature of the transacting owner. The digital signature serves as a proof of ownership over the bitcoins associated with that input. While the script length is defined by the `Txin-scriptLength` parameter, its contents are stored in the `Txin-scriptSig` field. The script and its contents are later discussed in detail in the scripts Section 2.1.3. The last field that is part of every transaction input is the `sequence_number`. The sequence number was intended to update the time-locked transactions before they are confirmed or before the time lock expires. It is inherited from the legacy Bitcoin and is generally not used in the current version. Presently, it is set to the four-byte unsigned max value (`0xFFFFFFFF`) that disables the locktime. To enable locktime at least one of the inputs must have its sequence number set to a value less than the maximum.

Transaction Output - The transaction outputs decides the owner of the resulting sum of bitcoins and contain the set of instructions to authorize such ownership. Every output is identified by an implied index value, starting with zero, based on its position within the list of transaction outputs. The non-negative `Tx_value` field specifies the value in satoshis (`1 BTC = 100,000,000 satoshis`) to be paid to the one who fulfills the conditions laid in the `Txout-scriptPubKey` script. The `Txout-scriptLength` field in the output of a transaction defines the script length as shown in Table 2.2.

Unspent Transaction Output (UTXO) - is the output generated in a valid transaction. In every Bitcoin transaction, the input of the transaction spends the previous unspent output and the conditions for spending are governed by the pubKey script, which is part of the transaction output [22]. In the live bitcoin network, the number of UTXOs is more than 54 million [53] as of June 2017. The nodes in the network track these UTXOs in an indexed set called the UTXO set and use them during the transaction verification.

Fields	Size (bytes)
`Tx_value`	`8`
`Txout-scriptLength`	`1-9`
`Txout-scriptPubKey`	`<Txout-scriptLength>`

Table 2.2: Contents of a Transaction Output

2.1.3 Bitcoin Addresses and Scripts

In Bitcoin, an **address** is the 20-byte string shared with others to fetch the bitcoins. It is computed from the public key of the key pair whose private counterpart is used to sign the corresponding spending transaction. It is derived using the base58 encoding of the double hashed Elliptic Curve (EC) secp256k1 public key [3] as represented by the following formula. The keys and its derivations are discussed in detail in Section 2.1.7.

```
Bitcoin Address = Base58Encode(RIPEMD160(SHA256(PUBLIC_KEY)))
```

To regulate and verify the authenticity of each transaction in a decentralized environment, Bitcoin uses **scripts** [44] based on public-key cryptography. These scripts are essentially a set of ordered instructions embedded as part of each transaction input and output. They are executed from left to right in a stack-based approach similar to Forth programming paradigm [14]. Bitcoin scripting in itself is a stateless, non Turing-complete language designed deliberately to avoid loops and complex programming structures.

Bitcoin transactions uses two scripts - *signature script* (`scriptSig`) included in every input and *public key script* (`scriptPubKey`) embedded in every output of the transaction. Bitcoin enforces future spending of the unspent outputs using these scripts. The signature script contains two elements as listed and explained below.

- Public key - on hashing must generate a result equivalent to the address specified in the output of the previous transaction corresponding to this input.

- Digital signature - generated using the private counterpart of the public key just provided. This affirms that the private key is in the possession of the transacting party. Since the data for signing essentially includes the entire transaction, any modifications after signing can be easily detected.

The public key script recorded in the output of a transaction contains a list of instructions that set the conditions on spending the output amount. Anyone who is holding the private key to the address (public key hash) encoded in this script can spend this output. These two scripts thus ensure that the person owning the private key is authorized to spend the unspent transaction amount. Additionally, they obviate the need for a trusted third party to mediate and regulate the transactions. The following paragraphs outline the standard script types [4], their instruction sets and the validation procedures currently used in the bitcoin network.

Pay To Public Key Hash (P2PKH) - is one of the most common scripting standards being used currently. Since Bitcoin address is nothing but the hash of the public key, P2PKH simply means pay to the address. The two P2PKH scripts used in the transactions are as follows.

```
scriptSig: <signature> <publicKey>
scriptPubKey: OP_DUP OP_HASH160 <pubkey_hash>
                               OP_EQUALVERIFY OP_CHECKSIG
```

While the `scriptSig` script is part of the input within the spending transaction, the `scriptPubKey` is recorded in the unspent output of a previous transaction. The `scriptSig` encapsulates the DER-encoded ECDSA SECP256K1 signature generated using the ECDSA private key and the corresponding raw ECDSA public key. The `scriptPubKey` carries the 20-byte destination address. It also contains the opcodes - `OP_DUP` (duplicate the stack item), `OP_HASH160` (hash input first with SHA-256 followed by RIPEMD-160), `OP_EQUALVERIFY` (checks inputs for equality) and `OP_CHECKSIG` (signature match). For transaction validation, the `scriptPubKey` of the previous transaction (UTXO) is appended to the `scriptSig` of the corresponding input and is evaluated using the opcodes in a stack-based approach.

Pay To Public Key (P2PK) - is one of the simplest scripting standard wherein the ECDSA public key itself is stored in the `scriptPubKey` rather than the hash of it. Also, the `scriptSig` contains only the signature as shown below. This technique is rarely used by the older versions and is mostly obsolete.

```
scriptSig: <signature>
scriptPubKey: <public_key> OP_CHECKSIG
```

Multisig - is a standard technique to enforce multiple signature requirement to spend a UTXO. It is also called as m-of-n standard where m represents the minimum number of signatures required with each of them verifiable

by at least one of the n public keys supplied. Currently, a maximum of 15 such signatures and public keys can be incorporated in the `scriptSig` and `scriptPubKey` respectively. The defined format for the multi-sig scripts with three signatures are as mentioned below.

```
scriptSig: OP_0 <A sig> [B sig] [C sig]
scriptPubKey: <m> <A pubkey> [B pubkey] [C pubkey]
                                <n> OP_CHECKMULTISIG
```

Pay To Script Hash (P2SH) - is the recent addition intended to replace the multisig standard. The downside of m-n multisig approach is that even for a simple transaction, the `scriptPubKey` is required to store all the n public keys, thereby, increasing the script size. Given that the storage space on the blockchain is costly, P2SH introduced the notion of `redeem scripts` to use lesser storage. With P2SH, the `scriptPubKey` contents is replaced with its own digital fingerprint (hash). The redeem script can now hold any number of public keys and is only used when the corresponding transaction output needs to be spent. The P2SH script formats are as illustrated below.

```
redeemScript: <m> <pubkey> [pubkey...] <n> OP_CHECKMULTISIG
scriptSig: <sig> [sig...] <redeemScript>
scriptPubKey: OP_HASH160 <Hash160(redeemScript)> OP_EQUAL
```

Besides its simplicity P2SH, still, suffers from a major risk of accepting the receiving transaction even when the redeem script is invalid since the script content is opaque. However, the spending transaction fails in the event of providing a wrong redeem script. Hence, enough care should be exercised to prevent locking the bitcoins in a transaction.

Signature Hash Types - is a feature that permit users to selectively choose the information within the transaction to be signed [45]. At present, there are three different options provided to decide the information to be signed. The default option is `SIGHASH_ALL` that signs all the inputs and outputs except the signature scripts within the inputs. `SIGHASH_NONE` is used to sign only the inputs, thus, permitting anyone to meddle with the outputs. `SIGHASH_SINGLE` is used to sign only the output corresponding to the signing input identified by the index number.

2.1.4 Spending and Key Reusage

In this subsection, we describe a simple two-party P2PKH transaction and also explain the problem associated with key reuse.

Transaction Spending - In a Bitcoin transaction, all the referenced inputs need to be completely spent. Consider, for example, Alice has 25

BTC as a single UTXO and wants to transfer 20 BTC to Bob. In this case, Alice should create a transaction with two outputs - one encoding a value of 20 BTC to the address provided by Bob and the other encoding the remaining value after subtracting the mining fee (explained in Section 2.1.5) to one of her own addresses. The second output is called the change output and the corresponding address used is called the change address that is different from the one shared with Bob. The change address and its derivation is described in Section 2.1.7.

Consider a use case where Alice wants to send some bitcoins to Bob and Bob later spends them in a different transaction. Assume that both uses the standard P2PKH transaction type.

1. Bob generates a key-pair using ECDSA with secp256k1 curve and computes the 20-byte P2PKH bitcoin address by hashing the public key while retaining the private key. He then shares the address with Alice via *Quick Response* (QR) code or some one-way medium.

2. Alice decodes the address to get the pub-key hash, creates a transaction with needed inputs and an output with `scriptPubKey` containing the pub-key hash besides the change output.

3. Alice now broadcasts the created transaction to the bitcoin network where it is validated and included in the blockchain as a UTXO. This UTXO is then added to Bob's spendable balance.

4. After some time if Bob decides to spend the bitcoins received from Alice, he should create a new transaction with an input referencing the transaction previously created by Alice. He does so by using the previous transaction hash (txid) and the output index corresponding to the spending UTXO. He must also include the full public key used to generate the address in the previous transaction and a signature obtained with its private couterpart in the `scriptSig`. The `scriptSig` authorizes his control over the private key, and in turn, the UTXO associated with it.

5. Finally, Bob encodes all the inputs and outputs in the new transaction and broadcasts it to the network to be recorded in the blockchain. The entire scenario is depicted in the Figure 2.2.

Key Reusage and Consequences - A typical transaction spending exposes the public keys or addresses of the communicating parties to each other. In the event of using the same address (or public key) often, one

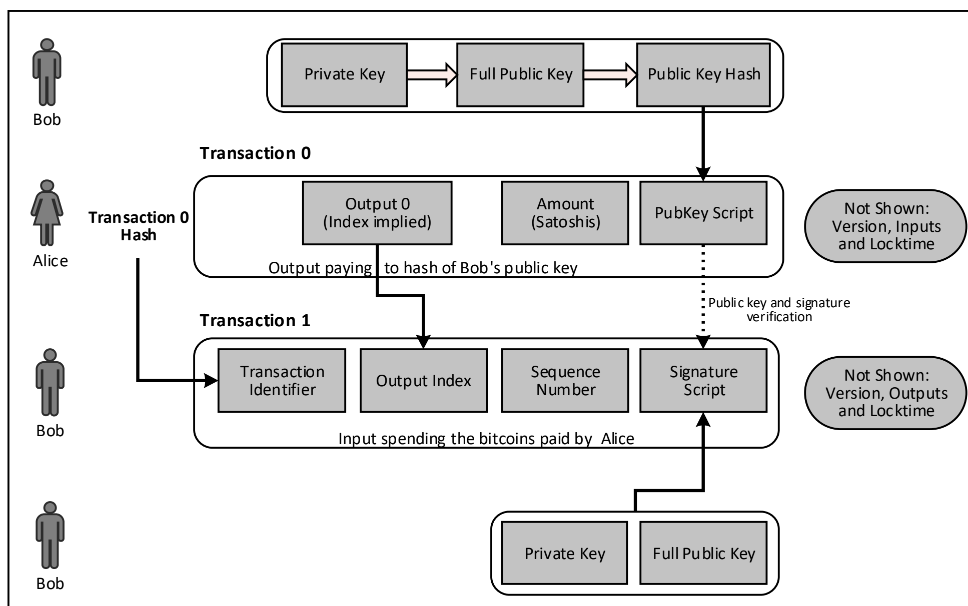

Figure 2.2: Alice-Bob P2PKH Transaction Spending

can monitor and track the transaction history and the amount each address holds, thus revealing the spending/receiving patterns of that person. Since the Blockchain is a public ledger, such tracking is possible, thereby leading to the loss of financial privacy. In addition, attackers could launch comparison-based attacks on signatures [33]. Using new and unique key-pairs for every transaction avoids such privacy breaches. Also, the attackers' efforts to perform comparison-based attacks become less relevant as the keys are no longer reused.

2.1.5 Blockchain

One of the fundamental blocks of Bitcoin is its underlying consensus technology, *blockchain*. Blockchain is a shared public ledger that tracks and records all the transactions broadcast to the network in chronological order [21]. It is a distributed data structure maintained by a set of peer nodes in the Bitcoin network. Each node stores its own blockchain consisting of the blocks validated by itself. They are said to be in consensus when several such nodes contain the same blocks. With a defined set of consensus rules in place, each transaction is cryptographically verified before it is permanently included in the blockchain. Such consensus-based decentralized approach has gained significant attention and is being studied for its applications in various fields.

Blockchain Architecture - As the name suggests, Blockchain is a chain of blocks with each block containing a 80-byte header and the body [9]. The header consists of a version number indicating the software version of the block, a nonce, target (explained in the following paragraph) and the current timestamp of size 4-bytes each and the 32-byte hash of the parent block. In addition, a merkle tree [36] is constructed with the hash of each transaction at the bottom of the tree resulting in a single 32-byte hash at its root. This hash, termed as merkle root, is also stored in the header. On the other hand, the body of the block stores the list of transactions resulting in the merkle root. Figure 2.3 and Figure 2.4 illustrates a simple blockchain and the merkle tree with four transactions respectively.

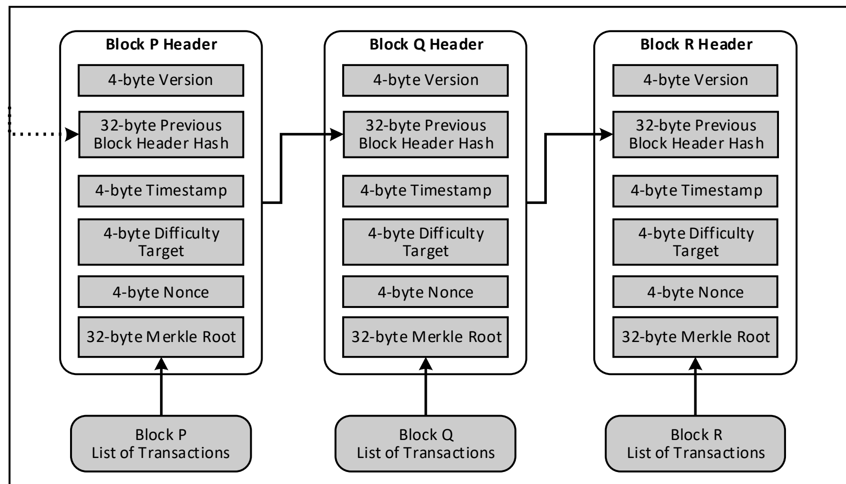

Figure 2.3: A snapshot of a Blockchain

Proof-of-Work and Mining - For each block to become a part of the shared ledger, it has to be approved by the majority of peers (miners) in the network. The approval is only obtained by establishing a proof that a considerable amount of work has been performed by the respective miner. This process of computation that requires a substantial amount of CPU resources is defined as *mining*, while the work is termed *Proof-of-Work (PoW)*. As explained in [37], this PoW is calculated, using hashcash [26] PoW concept, by incrementing a 32-bit nonce value within the block so that when the block is hashed, the resultant hash consists of a defined number of leading zeros. This number depends upon the difficulty set by the consensus protocol. Once the nodes are in agreement, the hash is stored along with the current timestamp within the block that becomes a part of the ledger.

In Bitcoin, *difficulty* means the amount of work required to produce a

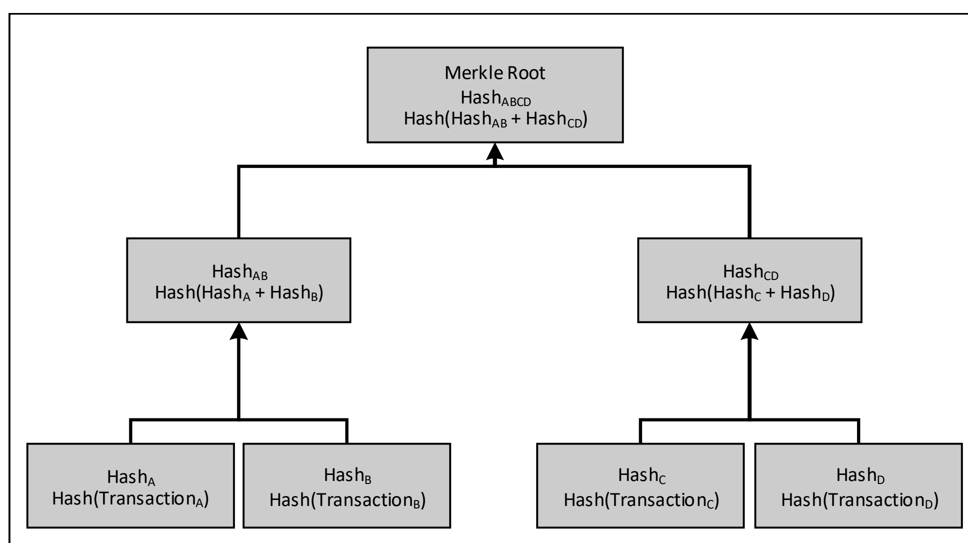

Figure 2.4: Merkle tree with four transactions A,B,C and D

hash value below a certain target threshold, 256-bit number. Such a target is revised every 2016 blocks based on the rate of creation of blocks given the fact that the ideal mining time per block is 10 minutes. If the time taken to generate 2016 blocks is less than two weeks, *difficulty* is increased by almost 300%, otherwise it is decreased proportionally by 75% [23]. In order to calculate the rate, the differences in timestamps stored within the block headers are used. In recognition of such effort and also to encourage them further, a reward of 12.5 BTC [19] is credited to the miner who mines a valid block. The miner also gets incentives in the form of fees for recording the user transactions within the block after confirmation. The total amount of reward and the incentives are collected in a special transaction within the block called coinbase or generation transaction. It is the first transaction in the block whose UTXO points to one of the miners addresses and can be spent by the miner in the future.

One of the key security aspects to be noted is that the data for hash calculation also includes the hash of the previous valid block in addition to the current block's transactions, nonce and timestamp values. Such an inclusion protects the integrity of the ledger and makes it difficult for an attacker to include a fake transaction in the block since he must recompute the PoW for all the succeeding blocks (generated by the honest nodes) starting from that block. This requires a lot of effort and resources and is extremely difficult unless the attacker controls the majority of the Bitcoin network.

Block Forking and Detection - In the peer-to-peer Bitcoin model, it is possible that more than one miner mines a valid block at the same exact time. In other cases, the first seen valid block is accepted by the nodes. However in this scenario, both the blocks are added thus creating a fork in the blockchain. When the next valid block arrives, it is added to only one of the branches. Then the longest chain with more work done (PoW) becomes the valid chain while the block on the other branch, or sidechain, is orphaned [15]. The non-recorded valid transactions in such orphaned blocks are added back to the *mempool* to be included in the next valid block. *mempool* is a storage area for the transactions in each Bitcoin node. Sometimes, very rarely, the low-value transactions with very low fees are dropped by the miners to reduce their mempool occupancy. Using such forks to their advantage, the attackers attempt to strengthen the invalid sidechains by including fake transactions. However, such an attack would only be successful when more than 50% of the nodes in the network are compromised.

2.1.6 Wallets

Unlike traditional banking systems, bitcoin users do not own explicit accounts to manage their transactions, but rather use wallets. In the Bitcoin context, a wallet refers to two different yet related entities - one is the wallet program, also known as a Bitcoin client, and the other is a wallet file. A wallet file is a storage medium that stores the set of private keys, public keys, balances and other wallet information. The wallet client and its types are explained below.

Wallet Clients - A wallet client is a piece of software that generates keys, creates addresses from public keys by hashing, distributes the addresses, signs transactions using private keys, broadcasts the signed proposals to the Bitcoin P2P network and optionally provides other ease-of-use facilities. These functionalities can be broadly grouped into three independent segments - a public key distribution component, a signing component, and a networking component. The public keys are generally distributed as P2PKH or P2SH addresses except when spending an UTXO which requires a raw public key to be included in the scriptSig. The networking component, apart from broadcasting, also fetches the previous transaction information from the blockchain for signing. Depending on the functionalities offered, these clients can be classified into the following categories:

1. *Full-Service Wallet* includes all of the above three components. Figure 2.5 depicts the workflow of a typical full-service wallet. They do not require users to switch between multiple applications, making it more

convenient and easy-to-use. However, the downside of such a wallet is that the private keys are stored in the wallet files on devices always connected to the Internet. Encryption is one way to prevent thefts but offers no protection against memory-access attacks or stolen decryption keys.

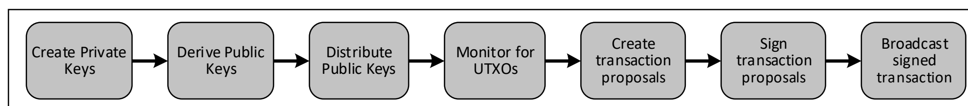

Figure 2.5: Operation of Full-Service Wallet

2. *Signing-only Wallets* are those that generate and store the private keys in a more protected environment. To create keys, these wallets use Deterministic key derivation explained in Section 2.1.7. They, however, only provide a safe and secure storage for the keys. A networking wallet is required to spend and receive bitcoins. While the signing wallet generates and stores the master private and public keys, the networking wallet gets the master public key, generates child public keys from master public key (explained in Section 2.1.7), computes addresses and distributes them. In addition, it also keeps track of outputs, creates transaction proposals and broadcasts the signed transactions to the network. For signing a transaction, the signing-wallet also derives the corresponding child private key from the master private key. Figure 2.6 illustrates the signing-wallet operations.

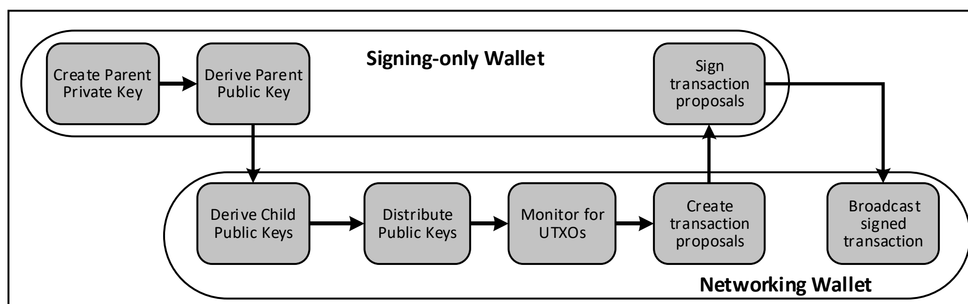

Figure 2.6: Operation of Signing-only Wallet

Signing-only wallets can be further classified into two types: *Offline wallets* and *Hardware wallets*. In offline wallets, the master keys are generated and stored in a completely offline (disconnected from Internet) system and the networking wallet is installed on a different online

system. When retrieving the master public key or signing the transaction, the information is copied to and fro using a removable device, such as USB. This greatly reduces the attack surface and provides improved security assuming that the offline wallet is not compromised. However, it is inconvenient to copy information between the systems and is neither practical nor easy-to-use. Hardware wallets overcome this manual transfer by using a dedicated device to generate and store the private keys that can be safely connected to an online system. These small and portable devices contain and run only the minimal code required to do these computations and are void of a standard system infrastructure. This eliminates the vulnerabilities that arises from the untrusted operating system and other processes. Though, hardware wallets seem more promising, users still need to carry an extra device to make a transaction. Also, purchasing a separate device increases the cost factor and the risk of losing it.

3. *Distributing-only Wallets* are used to only compute the public keys and addresses, and distribute them to the payers. They are more useful in a commercial setting where a public server is responsible for accepting the user payments. Such a server can generate addresses in two ways. One is to use the public keys from the pre-pooled database and the other way is to simply generate the child public keys from the master public key. In either way, the server has to keep track of the used addresses to avoid key reusage which is not desirable. As shown in Figure 2.7, these wallets depend on other wallet types to create keys, sign and broadcast transactions. To sign transactions, child private keys are derived from the parent private key using ECDSA as explained in the next section.

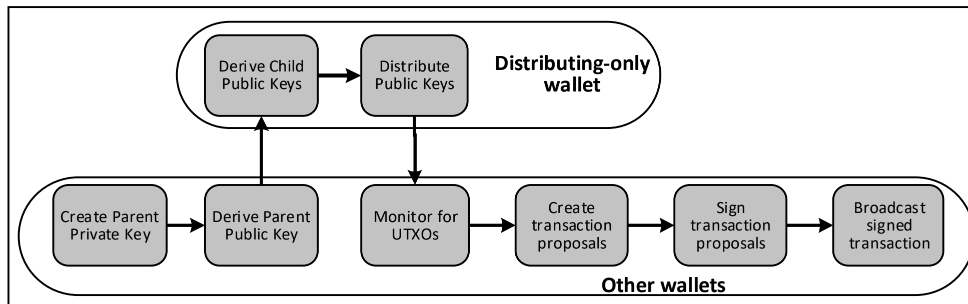

Figure 2.7: Operation of Distribution-only Wallet

2.1.7 Hierarchical Deterministic Wallets

Having said that key reusage risks privacy and security, the first Bitcoin wallets (or clients) used keys from the set of pre-generated key-pairs stored in the wallet file and would generate new pairs as needed. Such wallets are called Type-0 nondeterministic wallets. Due to their nature of using pre-generated keys, they are also nicknamed as "Just a Bunch Of Keys" (JBOK) [5]. However, Bitcoin developers soon realized the cumbersomeness involved in using such random key pairs. The primary drawback of this approach is that each wallet has to store and backup all the keys frequently to avoid losing bitcoins in case of wallets being lost or corrupted. This also increases wallet file size and causes issues during wallet import.

To solve this issue, Hierarchical Deterministic (HD) key derivation and transfer protocol is proposed in the Bitcoin Improvement Protocol (BIP) standard, bip0032 [55]. HD key creation resembles a tree structure with the parent key generating a sequence of child keys, each of which, in turn, produces a sequence of grand-child keys. Currently, most of the standard bitcoin wallets use deterministic derivation. In this subsection, we look at the deterministic key creation and its applicability in the bitcoin world.

Properties of ECDSA: HD protocol uses ECDSA [29] with secp256k1 curve for deterministic key generation. In ECDSA, a private key is any 256-bit number in the range 1 to $2^{256} - 1$, while the public key is a point on that particular secp256k1 curve derived using the ECDSA point function and the private key [27] as represented below.

```
public_key = point(private_key)
```

An interesting property of the ECDSA point function is that a child public key can be derived given an integer value, i, and the parent public key as shown in the following expression

```
child public_key = parent_publicKey + point(i)
                 = point((parent_privateKey + i) % G)
```

where `G` is the generator point specified in the secp curve standard. This property is used in creating the child public keys without using the parent private keys for sharing the addresses with other bitcoin users.

Root seed and Mnemonics: The source for deriving the master parent key-pair is any 128, 256 or 512 bit random value, usually known as the root seed. All the keys are further derived from this seed. Hence, it is sufficient to backup just this seed to recover the entire wallet. However, the randomly generated root seed is not human-readable and is difficult to transcribe. This

makes the wallet imports and manual backups vulnerable to errors where the root seed has to be entered manually by the user, thus necessitating digital back-ups. To make back-ups and imports easier and more convenient, mnemonic codes are proposed in the bip0039 standard [39]. Mnemonic codes are a sequence of words in a natural spoken language used to generate the 512-bit root seed. This seed is then used for deriving the deterministic keys. The following are the series of steps involved in the seed creation.

1. Generate a 128-256 bit random sequence (`R`).
2. Hash (`R`) using SHA256, extract the first few bits of the hash to produce the entropy checksum and then append it to (`R`).
3. Now, divide the concatenated sequence into 11-bit segments. Each segment is then used to index a pre-defined dictionary of 2048 words.
4. Depending on the number of bits of entropy in R, the above step results in a sequence of 12-24 words called mnemonic codes. Table 2.3 represents the correlation between the number of bits of source entropy and the resulting number of mnemonic codes.

Entropy (bits)	Checksum (bits)	Total bits	Number of Words
128	4	132	12
160	5	165	15
192	6	198	18
224	7	231	21
256	8	264	24

Table 2.3: Mnemonic word length as a function of source entropy bits

5. The mnemonic can be protected by an optional password of variable length appended at the end. The Password-Based Key Derivation Function (PBKDF2) [31] with HMAC-SHA512 pseudo-random function and iteration count of 2048 is then applied to the concatenated string to obtain a 512-bit binary root seed.

Using this approach, it is only required to save the set of mnemonic words which are more human-readable. In the context of Bitcoin wallets, these set of words are collectively termed as the *Recovery Phrase*.

Master Key and Extended Keys: HD wallets uses mnemonic words to generate the root seed. The 512-bit seed then derives the *master parent key-pair* using a HMAC-SHA512 one-way hash function. The ECDSA point function derives the raw 256-bit master public key (denoted as 'M') from the leftmost 256-bits of hash, or raw master private key (denoted as 'm'). The rightmost 256 hash bits, or *master chain code*, is the deterministic seed generated to introduce sufficient entropy in the child key derivations [27]. Figure 2.8 illustrates the generation of master keys. Though the generated public key is 256-bits in size, an extra byte is appended in front in Bitcoin to differentiate compressed (0x02 or 0x03) from uncompressed (0x04) keys.

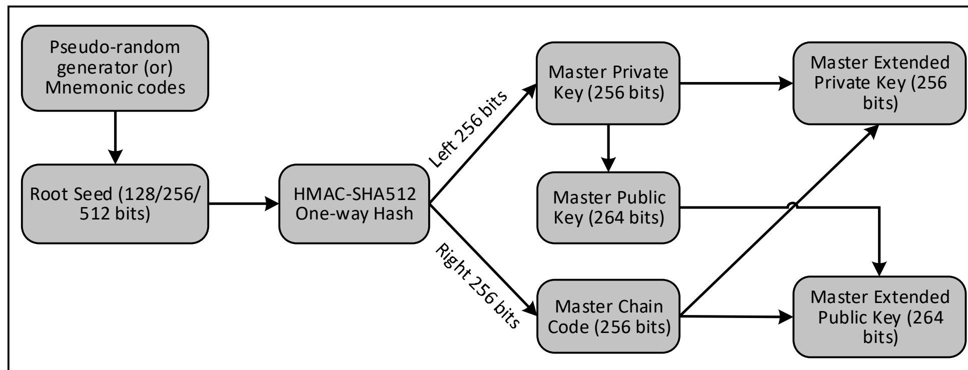

Figure 2.8: Master and Extended Master Keys

As shown in Figure 2.8, an *extended key* is a combination of public or private key and the chain code. While an extended public key is a combination of public key and the chain code, an extended private key is the concatenation of private key and the chain code.

Child Key Derivation: In order to derive the child keys from the parent keys, a Child Key Derivation (CKD) function is used as proposed in [55]. The CKD function uses a one-way hash function, HMAC-SHA512, with the extended master public key and a 32-bit index number as inputs. In the HD protocol, this index number is part of a deterministic sequence of integers starting from zero. The index zero uniquely identifies the first child key-pair generated. The master chain code in the extended key prevents CKD from depending only on the deterministic integer and the parent key. CKD function can be further classified into two categories as explained below:

1. *Private Key Derivation* is the process of deriving the child private key from the parent private key. With the 32-bit index number, each parent key can generate more than 2 billion child keys. The derivation is illustrated in Figure 2.9.

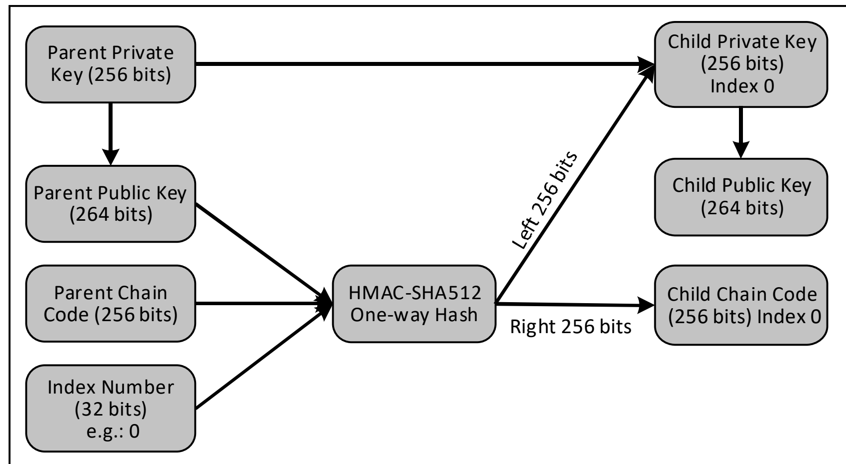

Figure 2.9: Child Private Key Derivation

2. *Public Key Derivation* is the process of deriving a child public key from the parent. There are two ways to generate a child public key. One way is to generate it from the already generated child private key using the ECDSA point function. The other way is to completely generate by using the extended parent public key and the index number as depicted in Figure 2.10.

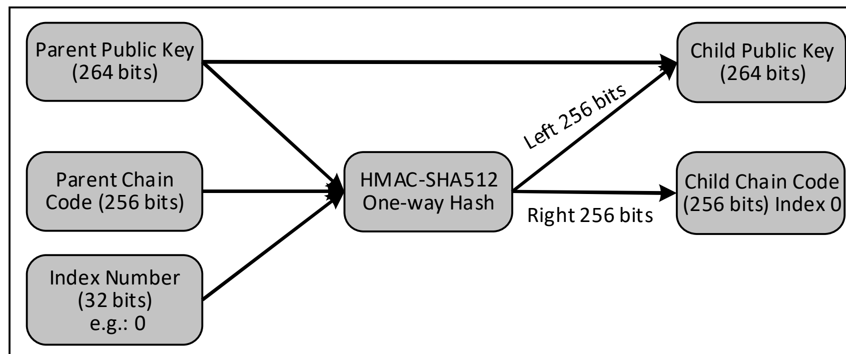

Figure 2.10: Child Public Key Derivation

Since public key derivation is independent of the parent private key, it is used to generate the addresses (hash of the child public key) using the parent public key in case of wallet clients that depend on offline or hardware wallets for signing the transactions.

Hardened Key Derivation: The public key derivation exposes the

parent chain code via the extended public key. In an untoward event of the corresponding child private key compromise, the leaked private key combined with the parent chain code can be used to derive the private keys of all the subsequent children. Even worse, the parent private key can also be deduced by reversing the normal child private key derivation [25]. In order to mitigate such key compromise, *hardened key derivation* was introduced. It generates the child chain code by using the parent private key instead of the parent public key as shown in Figure 2.11.

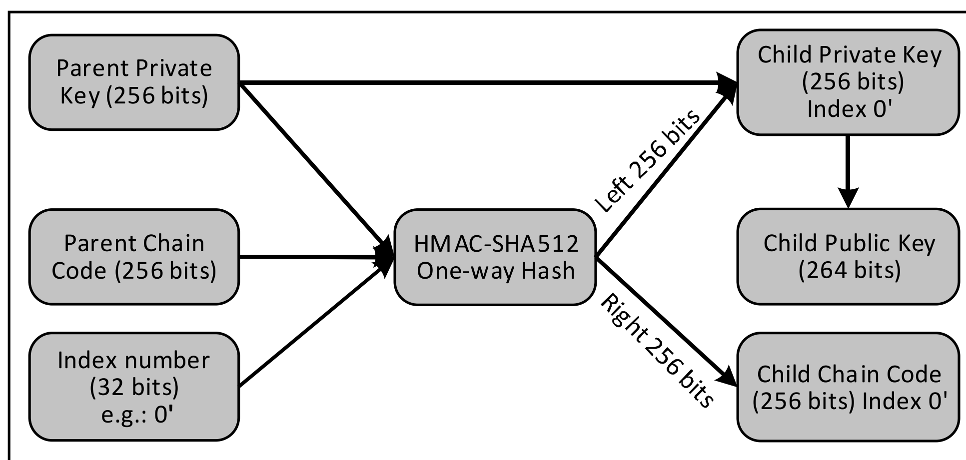

Figure 2.11: Hardened Child Key Derivation

The concept of hardened derivation is used in HD wallets to generate the bitcoin addresses from the hardened parent. The type of key derivation is determined by using the 32-bit index number. Normal derivation uses the indexes in the range 0 to $2^{31} - 1$, while the hardened derivation uses indexes in the range 2^{31} to $2^{32} - 1$. In order to make the hardened indexes easy to read and transcribe, they are represented starting from zero followed by a prime symbol, i.e., $2^{31} = 0'$ [25].

HD Wallet Path: A HD wallet is a collection of many accounts derived from a common root seed with each account uniquely identified by a number starting from zero. Every account consists of two different chains: external and internal chain, that identify different Bitcoin addresses. The external chain is used to derive the public addresses shared with bitcoin senders while the internal chain derives the change addresses. A change address holds the surplus bitcoins from the spending transaction directed towards the sender. Figure 2.12 represents a bip0032 HD wallet.

The HD wallet path defines the account number, bitcoin scheme (purpose), coin type, chain type and the corresponding address index as proposed

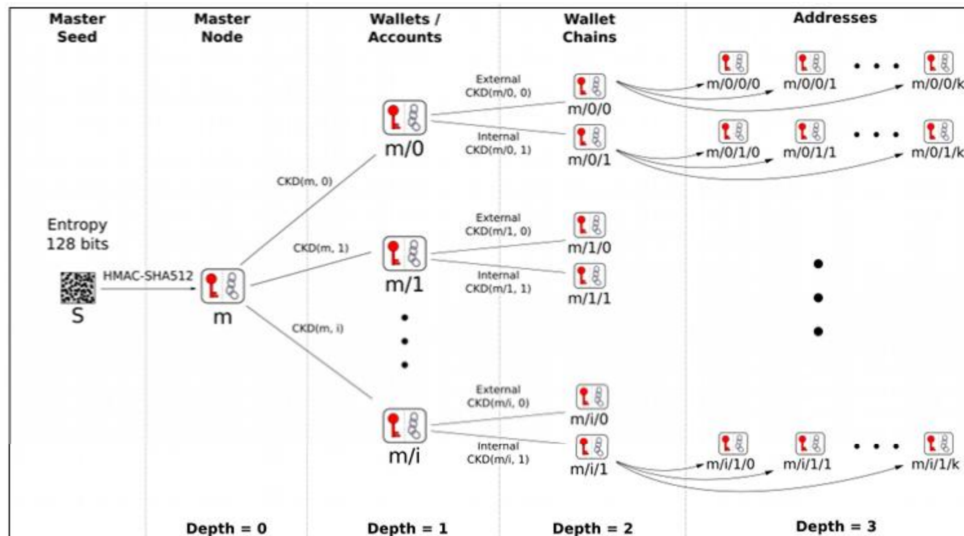

Figure 2.12: Hierarchical Deterministic (bip0032) Wallet [55]

in bip0044 [38]. The following is the general representation of a complete HD wallet path.

`m/purpose'/coin_type'/account_number'/chain_type/address_index`

For example, `m/44'/0'/2'/1/5` represents sixth change (internal) address public key of the third bitcoin account using the Bitcoin scheme `44`. As discussed earlier, hardened key derivation is used for purpose, coin type and account number child keys as indicated by the prime (′) symbol.

2.1.8 Double-Spending Attack

One of the potential challenges inhibiting the wide-spread use of digital currencies for real world trade is double-spending [37]. Double-spending happens when two or more transactions spend the same UTXO. When a transaction is broadcast, it has to be mined into a block and should be approved by other peers in the network. Such an approval is called a *confirmation*. After broadcasting the signed transaction that pays a certain amount to a particular receiver, the malicious spender could sign and broadcast a second transaction spending the same UTXO before the previous transaction is accepted.

Once a transaction is confirmed and permanently included in a particular block, it is impossible to double-spend the respective UTXO, because the

malicious spender needs to modify the blockchain starting from that block and must create a valid sidechain as explained in Section 2.1.5. Thus, it is safe to claim that a particular transaction cannot be double-spent after receiving a certain number of confirmations. This number represents the number of succeeding blocks added to the blockchain after that particular block with each block reinforcing security. The Bitcoin protocol recommends to wait for at least six confirmations to ensure no double-spending. Since it takes nearly 10 minutes for a single block to be added to the blockchain, on an average six confirmations requires upto an hour.

Though Bitcoin is inherently designed to deal with double-spending, the confirmation-based mechanism involves more wait time. Even a single confirmation requires at least 10 minutes and is highly undesirable for practical purposes. This solution won't serve its purpose if the bitcoin receiver accepts unconfirmed payments and is unable to wait for the confirmations to arrive. This involves a potential risk for the receiver, especially in case of high-valued transactions. Recent research towards mitigating double-spending attacks is presented in Section 2.3.

2.2 Trusted Execution Environment - Intel Software Guard Extensions (SGX)

A Trusted Execution Environment (TEE) is a secure and isolated processing environment. It can be realized in several ways including special processor extensions that guarantee isolation. The major features of TEEs are isolated execution, secure provisioning, remote attestation, secure storage (e.g. using secure hardware element) and trusted execution path [54]. Applications running in the TEEs are confidentiality and integrity-protected and are inaccessible to external parties. GlobalPlatform is an international industry body standardizing TEE specifications[1].

Intel Software Guard Extensions (SGX) is a set of architectural extensions introduced to solve secure remote computation using trusted hardware [18]. It formulates a new set of instructions that provides trusted execution environment to the user-level code.

2.2.1 Enclave

Any user-space application running on SGX-enabled processors can designate their code as an *enclave*. Enclaves are protected regions of memory

[1] More details of TEE at https://www.globalplatform.org/mediaguidetee.asp

that securely load and execute the user code and contain the related data used for the computation. These regions are isolated from the external environment by the SGX-enabled Intel processors and are not accessible even to the operating system, BIOS, hypervisors and device drivers. These regions of memory are collectively called the Processor Reserved Memory (PRM) and are protected by the CPU from all non-enclave accesses. However, the memory and the other computer resources are still managed by the OS kernel.

The PRM consists of Enclave Page Cache (EPC) and Enclave Page Cache Metadata (EPCM). EPC constitutes a set of 4 KB pages that are used by the enclaves to load and store user code and data. The associated page state information is tracked in the EPCM by the CPU. When an enclave is being created, the untrusted system software requests the CPU to assign the required number of EPC pages to the enclave. It also instructs the CPU to copy the enclave code and data from outside the PRM into the allocated EPC pages and vice-versa. Therefore, the system software is aware of the initial state of the enclave and the resultant output returned by the enclave [18].

Every enclave records the values of two measurement registers, MRENCLAVE and MRSIGNER, at the end of its creation [2]. MRENCLAVE is a cryptographic measurement of the enclave contents whose value is the *Enclave Identity*. It is a SHA-256 hash of the contents of the enclave pages (code, data, stack and heap), order and relative positions of the pages within the enclave, and the security properties associated with each page. Every enclave is signed by a sealing (or signing) authority before it is released. MRSIGNER stores the hash of the public key of this sealing authority. This value, called *Sealing Identity*, remains same for multiple enclaves, when they are signed by the same authority. Some of the major security properties of the enclaves are explained in the following paragraphs.

2.2.2 Sealing and Replay Protection

When the enclave is running, its contents are protected by the underlying hardware to provide confidentiality and integrity. However, its contents are erased when the respective process exits and the enclave is uninitialized. To persist the enclave data for future use, the enclave contents can be encrypted with the enclave-specific keys called sealing keys. This process is called sealing and the sealing keys can be used and are accessible only within the running enclave. Intel SGX supports two sealing policies [2]. One is sealing to the *Enclave Identity* where the sealing key is based on the value of MRENCLAVE and differs every time the enclave content changes. The other policy is sealing to the *Sealing Identity* where the sealing key is based on the value of MRSIGNER that depends on the identity of the sealing authority.

The data sealed within the enclave can be stored in the untrusted file system and passed to the enclave when necessary. Any attempts by the adversary to replay the sealed data can be thwarted by including and checking a version number using the monotonic counters accessible only within the enclave. A monotonic counter is the only value that is persisted even when the enclave is destroyed.

2.2.3 Attestation

Attestation is the process of proving the authenticity of specific software running on a particular platform under certain prescribed settings. It provides the necessary proof to an external party using the software for some sensitive purposes. Intel SGX provisions this feature for its enclaves and provides a signature proof by signing with an *attestation key* unique to the Trusted Computing Base (TCB) of a platform. This proof asserts the identity of the enclave environment, its mode of operation, enclave contents, specific user data and a cryptographic binding to the TCB of the platform. This proof is generated by a special enclave running on the platform called *Quoting enclave* using the Intel Enhanced Privacy ID (EPID) protocol [13] and such a proof is called a *Quote*. Figure 2.13 depicts the creation of an attestation proof.

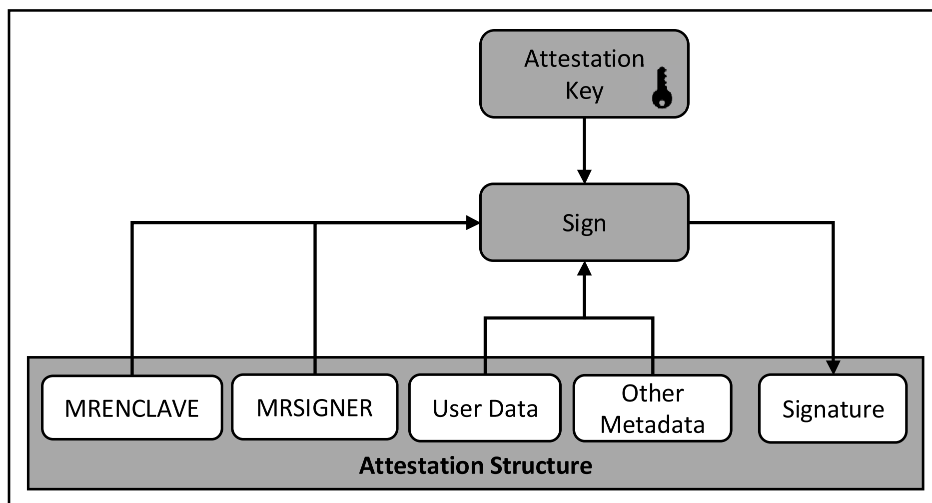

Figure 2.13: SGX Attestation Assertion Creation [2]

The SGX architecture provides two attestation mechanisms. **Local Attestation** proves the authenticity of one enclave to another running on the

same platform (or machine) and is also known as intra-platform attestation of enclaves. **Remote Attestation** is the process of proving the authenticity of an enclave running on a particular machine to an external party. The external party need not necessarily have an SGX-enabled machine, but, could verify the authenticity and integrity of the proof using a challenge-response mechanism and the Intel Attestation Service (IAS) as described in [2].

2.3 Related Research

The growing popularity of Bitcoin and the difficulties in its widespread adoption induced significant growth of research interest. In 2016, Tschorsch et al. [51] published a detailed technical study of digital currencies including Bitcoin. Similarly, Bonneau et al. [10] explained the various challenges faced by Bitcoin and outlined the possible research prospects. Double-spending, in particular, still remains a cause of concern preventing the greater acceptance of bitcoins for the real trade. While there have been papers analyzing double-spending such as hashrate-based analysis by Meni [43], some solutions are proposed recently to deal with it as explained in the following paragraphs.

Dmitrienko et al. [24] proposed to revoke double-spending transactions in Bitcoin by employing an *offline wallet* on the payer side assisted by either secure hardware or bitcoin deposits. The proposed system operates in three phases. In the first phase, the payer preloads a specific amount into the offline wallet to be later transferred to the payee. The offline wallet then transfers this amount to the payee and executes an offline transaction. In the process, the offline wallet provides a signed proof that uniquely identifies the transaction and the payer. The payee then redeems the bitcoins received by broadcasting the offline transaction. In the event of double-spending rejection, the payee broadcasts the signed proof by the payer to the Bitcoin network for revocation. The significant costs associated with the attacks against secure hardware and the risk of losing the bitcoins loaded to a Bitcoin deposit (in case of not using secure hardware) causes financial losses to the malicious payer. Even then, such a malicious payer is detected soon using the revocation mechanism. In this way it proposes to penalize double-spenders, but it does not necessarily prevent double-spending.

Podolanko et al. [41] proposed to detect double-spending in less than 28 seconds. They study the effectiveness of various countermeasures for double-spending. Based on their analysis, they propose to introduce new special type of nodes into the Bitcoin network called *Enhanced Observers* (ENHOBS) that detect double-spending transactions and alert the vendors within 28 seconds.

These ENHOBS carry out deeper inspection of the transactions broadcast to the network. On detection of double-spending, these ENHOBS send the messages to the peers in the network who are tasked with the responsibility of removing the double-spent transactions from the blockchain. This work aggregates and enhances the different observer-based solutions advocated by Karame et al. [32] and Bamert et al. [6]. However, it involves high overhead costs of transaction inspection, maintaining the observer nodes, balancing the number of observers and miners in the system and other potential issues critical for its adoption.

Teechan by Lind et al. [34] builds a full duplex fast bitcoin payment channel leveraging the SGX security guarantees. It defines a high-performance repeated payment protocol (Teechan) for executing low-latency, high throughput, secure bitcoin transactions offline. In Teechan, the payer and payee exchange mutual secrets and the remote attestation quotes to establish a secure payment channel between them. Next, they construct a setup transaction where both the payer and the payee deposit funds to a 2-of-2 multisig Bitcoin address. Then, the refund transaction is constructed that spends the deposited funds and returns the amounts to the payee and payer accordingly. This solution does not, however, define any mechanisms that directly avert or deal with double-spending, but rather proposes direct payments using multisignature, time-locked transactions.

Recently, Bitpay also announced a collaboration with Intel [48] to use SGX to carry out sensitive wallet operations in the protected enclaves thus defending against any malware or adversary on the system. However, they do not provide any indication on how to prevent double-spending.

Chapter 3

Adversary Model and Requirements

One of the key aspects of building a secure system is to identify the various actors involved with the system. The goals and capabilities of these actors influence and dictate the requirements of a such a system. It is, therefore, essential to understand the system from the point of view of these entities before designing a solution. In this chapter, we describe the relevant actors, their abilities and the set of requirements to be satisfied for the system to be secure.

3.1 Adversary Model

An *adversary* in a security system can be defined as any individual, group or organization with malicious intent to manifest an attack [42]. The success of such an attack depends on two important factors. One external factor is the ability of an adversary to defeat the security structures of the system. The other internal factor is the vulnerabilities within the system that can be avoided with a better and more secure design. A model that describes the powers of such adversaries that helps to detect, analyze and avoid the plausible attacks is called an *adversary model*.

A typical Bitcoin environment consists of several ***actors*** - bitcoin sender or payer, receiver or payee, bitcoin nodes or peers, and block miners whose functionalities are described below.

- A sender or payer is the one who transacts and signs a specific amount of bitcoins to a particular address. He/she creates the transaction proposal, signs it and broadcasts it to the Bitcoin network.

- A receiver or payee is the one who receives the bitcoins transacted to the address provided to the sender.
- A node or peer is the one who relays the transactions and the blocks broadcast to the Bitcoin network. They also verify and validate the transactions and vouch for them to be included in the blockchain.
- A block miner is any single individual or a group who works on generating the PoW to mine new blocks. A miner can also, optionally, be a node or peer in the Bitcoin network.

The major ***assets*** in a Bitcoin environment are the bitcoins involved in the transaction, network speed, node/peer memory pool, computational capacity of the miner and other network resources.

3.1.1 Adversary: Goals and Motivation

In our system, an adversary is *any transacting party* (or sender) with an intent to double-spend the bitcoins that are already used earlier for making a purchase. Our main asset in focus is the bitcoins involved in such a transaction.

The major goal of an adversary is to spend the same bitcoins more than once for different purchases. In order to do so, he/she makes use of the transaction confirmation window to broadcast a new transaction that double/reverse spends previously spent UTXOs as explained in Section 2.1.8. The major motivating factor behind such an attack is the monetary gain resulting from a successful attack. Moreover, in the event of rejection of a double-spending transaction, currently the adversary is neither restricted from using the services nor incurs any financial losses or penalties.

Consider a typical cafeteria scenario with two transacting parties: Alice, the buyer and Bob, the seller. Let us assume that Alice spends a particular amount of bitcoins from the previous transaction T1 (spendable by Alice) in a transaction T2 to buy a coffee from Bob. As soon as Bob delivers the coffee, Alice refers to the same UTXO from T1 and reverse spends it to herself in another transaction T3. Due to the peer-to-peer nature of the Bitcoin network and the delay involved in the propagation and the transaction confirmation, it is possible for T3 to get embedded in the blockchain before T2. Alice can use various techniques to ensure that T3 is confirmed in the blockchain [32]. In this case, T2 is later rejected as the particular UTXO is already spent as depicted in the Figure 3.1.

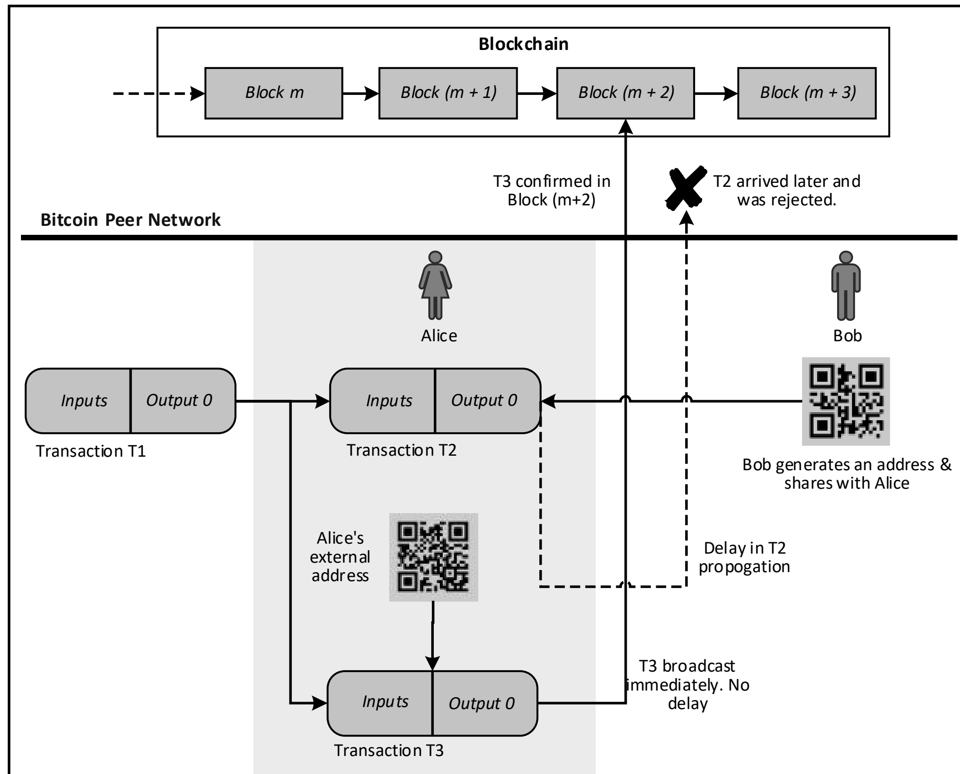

Figure 3.1: Double-spending Attack Scenario

3.1.2 Capabilities

As explained in the Section 2.2.1, the SGX enclave is the only trusted and protected part of our system which cannot be accessed by the attacker. Therefore, the ***attack surface*** consists of the entire context except the enclave. Also, the adversary can be anywhere outside the trusted enclave with the capabilities listed below.

C1. The adversary has full control and access to all the transaction information including the signature and hash, and is the one who decides when to create, sign and broadcast the transaction.

C2. The adversary has complete authority to invoke all the externally exposed functions of the trusted enclave to execute signing and other operations any number of times and is in full control of the data being passed in to the enclave.

C3. The adversary is also capable of carrying out side-channel attacks in an attempt to extract sensitive information from the TEE.

3.1.3 Assumptions

The proposed solution and system design relies on the following premises and assumptions that all hold true in a typical spending environment. The assumptions A1-A4 are based on the security guarantees of Intel SGX published in [18].

A1. With SGX enclave memory isolation and encryption in place, the adversary can neither read/write to the enclave memory directly nor attempt to access it indirectly via the operating system, BIOS, firmware, drivers and other malicious components that are not part of the TCB.

A2. Any modifications by the adversary to sealed (or encrypted) information can be easily detected within the enclave. Also, we assume that the adversary has no access to the sealing keys outside the enclave and the keys used for sealing cannot be retrieved or derived in any form.

A3. The information sealed by an enclave is specific to it on that particular hardware machine and cannot be unsealed or accessed by any other enclave either on the same machine or a different machine.

A4. The remote attestation quote is valid and verifiable and it does only endorse the actual piece of code running in the specific enclave on a particular machine. It neither misrepresents insecure portions nor can it be tampered to produce same quote for different software.

A5. We also assume that the security and cryptographic guarantees ensured by the current deployed Bitcoin mechanism are intact.

3.2 Security Requirements

In this subsection, we describe the security requirements derived based on the security perimeter (enclave), adversaries and their abilities:

S1. *Sign-once semantics:* The proposed system must assure that the transacting party cannot use a signing key more than once to sign multiple transactions i.e., only one signature per signing key. In the Bitcoin context, every UTXO has a specific key-pair and one needs its private

key to prove their ownership and spend the associated bitcoins. Therefore, the sign-once primitive can be used to prevent double-spending of bitcoins.

S2. *Signing keys protection:* The system must also restrict all forms of access to the private keys used for signing the transactions. The adversary must not be able to read or modify the signing key information i.e., both confidentiality and integrity must be provided.

S3. *Sign-once Guarantee:* The transacting party must provide a trustworthy guarantee verifiable by a remote party involved in the transaction. Such a guarantee shall prove to the remote party the authenticity of the hardware and trusted code running on the sender's machine.

3.3 Usability Requirements

The ease-of-use of an application determines its rate of adoption in the user community. The following usability features are desirable:

U1. *No Additional Devices:* Unlike some of the existing wallets, the system must not require the users to carry any external devices for signing the transactions. However, the payer still needs to have the wallet on a device containing a TEE to use our solution.

3.4 Deployability Requirements

An important factor that decides the application success is its ease of deployment. The following deployability requirements are applicable:

D1. *No Protocol Modifications:* The solution must operate in the existing Bitcoin ecosystem without requiring any modifications to the miners, nodes or the networking protocol.

D2. *Off-the-shelf Hardware:* The proposed solution must be realizable using off-the-shelf hardware in the host devices.

D3. *Incremental Deployment:* The solution when integrated with the existing deployments should not render the existing mechanisms inoperable and must still be usable in the absence of off-the-shelf hardware.

Chapter 4

Solution Design and Architecture

In this chapter, we discuss the solution and system design that incorporates the various requirements defined in the previous chapter. It introduces the various components and describes the functionality and constraints of operation of each component.

4.1 Design Overview

Our solution proposes to prevent a payer from double-spending the bitcoins by enforcing the sign-once semantics and also provide a mechanism for the payee to verify the same. To achieve it, we modify the bitcoin wallet used by the payer to prevent reusing the keys for signing multiple transactions and to transmit the verifiable guarantee (quote) to the payee. We also modify the wallet used by the payee to incorporate a procedure to verify the quote.

Our wallet is a HD wallet consisting of three major components - transaction manager, enclave bridge and SGX enclave library. They communicate and exchange data between each other to execute the wallet functions and are transparent to the user. Each component encompasses a number of sub-modules responsible for executing various tasks. Figure 4.1 illustrates a component overview of the system.

4.2 Functional Overview

Our wallet application encompassing all of its components is a full-service wallet as described in Section 2.1.6. However, the enclave component of the wallet itself could be a stand-alone, signing-only wallet. The various functions of the wallet can be broadly grouped into three major tasks. They are

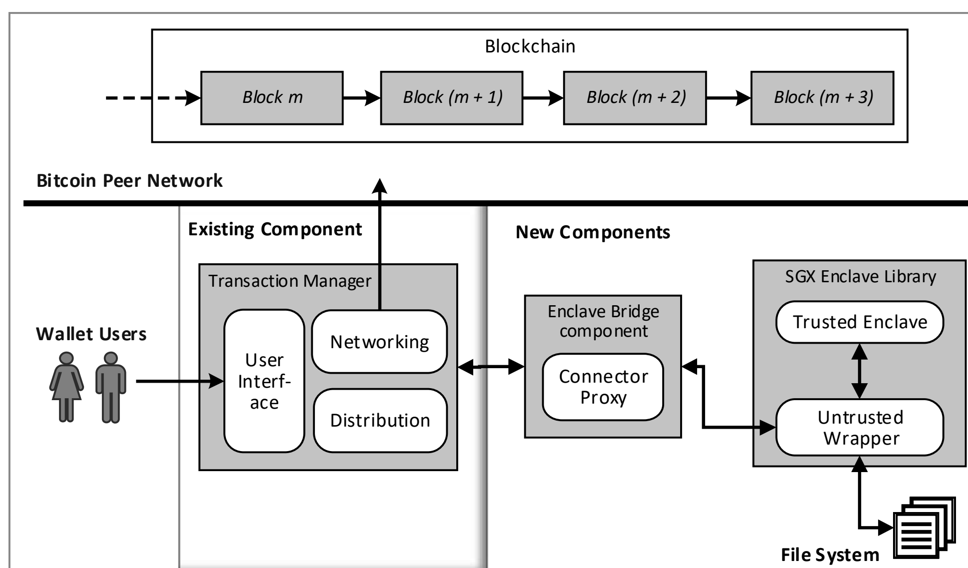

Figure 4.1: Component overview of SGX-based wallet

wallet creation, sending bitcoins and receiving bitcoins. Each task requires all three components for its operation. The following subsections describe each component and its respective functionality in detail.

4.2.1 Transaction Manager Component (Existing)

Most of the functionality of this component is already implemented in existing bitcoin wallets. We still describe it here to give a detailed picture of our intended system. We would, however, use one of the existing wallets in our prototype implementation.

The transaction manager component is the combination of three major modules - user interface, distribution and networking. This component is equivalent in functionality to the combination of a distribution-only wallet with the networking wallet. The user interface module is the front-end that interacts with the end-user and accepts the inputs. The networking module is responsible for managing the communications with the external components such as the Bitcoin peer network and other wallet users. The distribution module takes care of address generation and distribution to receive bitcoins from others. However, it does not control or possess any private key information and only manages the less-sensitive public data. To a normal wallet user, this is the starting point for all the wallet operations. The following list outlines the various responsibilities of this component:

1. *User input validation* - The user interface module verifies and validates the inputs entered by the user at different instances and does the required input sanitization. It checks if the user has submitted all the required inputs such as coin type (bitcoin, testnet), receiver address, amount, etc. It also checks for sufficient balance when the user enters a specific amount of bitcoins to be transferred.

2. *Fetch wallet public key* - When the user creates a new wallet, it is created with a default account identified by the unique index (usually zero) to be used for sending and receiving the bitcoins. Each account in the HD wallet is identified by the unique and hardened public-private keypair also known as *parent keys*. These hardened keys are further used to derive the normal child keys. Such a hardened derivation requiring private key information is derived within the trusted enclave. Hence, the parent public key has to be fetched by the transaction manager from the enclave.

3. *Address generation and distribution* - A wallet is identified and associated with the parent public key fetched from the enclave. The distribution module uses this parent public key of the wallet to derive the child public keys which are further hashed to get the bitcoin addresses as described in Section 2.1.7. The 20-byte address is then shared with the payers using *QR codes* to receive the bitcoins.

4. *Balance tracking* - The transaction manager also monitors incoming UTXOs that are spendable by the wallet user and displays their aggregation as the balance in his/her account. However, it does not validate the UTXOs.

5. *Transaction proposal creation* - When the user wants to spend bitcoins, he/she enters the valid receiver address and the amount to transact. Then the transaction manager creates a transaction proposal that consists of a set of inputs referring to previous UTXOs (Table 2.1) and outputs (Table 2.2) including the change output. It also retrieves the previous transaction data corresponding to the UTXOs included in the inputs and packages it into the proposal.

6. *Fetch signatures* - Given that the signing keys are controlled by the trusted enclave, the transaction manager needs to send in the list of inputs, outputs and previous transaction data to the enclave for transaction signing and getting the signatures. The existing transaction manager component is modified to provide this new functionality specific to our proposed system.

7. *Broadcast transaction* - Once the transaction is signed, the networking module propagates the signed proposal to the Bitcoin peer network where it is validated again before it is permanently recorded in the Blockchain.

8. *Display attestation quote* - Finally, the transaction manager displays the attestation quote obtained from the enclave (Section 4.2.3) proving the sign-once semantics to the merchant or seller. This is a new functionality introduced as part of our solution proposal.

4.2.2 Enclave Bridge Component (New)

The enclave bridge is a connecting proxy middleware that sits between the transaction manager and the enclave. It is the interface through which the transaction manager accesses and invokes enclave functions and exchange data. As the name suggests, the major functionality of this component consists of:

1. Accepting inputs from the transaction manager, validating and converting them to appropriate data structures used by the enclave Application Programming Interfaces (APIs). In case of fetching the parent public key of the wallet, the HD wallet path needs to be passed in as an input parameter to the enclave. In case of signing the transactions, the array of inputs, outputs and the previous transaction data are passed to the enclave.

2. Invoking the enclave APIs with inputs and getting the results back from the enclave synchronously. Also, based on the status of the operation, to return either the appropriate error message or the relevant output structures back to the transaction manager.

4.2.3 SGX Enclave Library (New)

This is the most important component of our system that extends the trusted services to the wallet. It contains two major modules - ***untrusted module*** and the ***trusted enclave module***. In the entire system, the only trusted part inaccessible to the adversary is the trusted enclave.

In our system, the only assets that must be protected from the adversary to prevent double-spending are the private signing keys. For address generation, the child public keys are derived from the master public key outside the enclave. In HD wallets, all these keys are derived from the single seed. Therefore, the ***trusted enclave*** executes only those operations that access

the seed and the private keys. On the other hand, the ***untrusted module*** is responsible for validating the inputs, invoking the enclave functions and retrieving the results. It is the entry point to the library through which the trusted functions are accessed. In the event of any failure, it must also return an appropriate error message to the caller. The following paragraphs describe the major functions of this component and also discuss the roles of the trusted and untrusted modules in each of them:

Wallet Initialization is responsible for creating and initializing a new wallet. The transaction manager component invokes the enclave library function and passes the HD wallet path as input via enclave bridge to fetch the parent public key associated with the new wallet account. The untrusted part verifies the input, checks if the enclave is initialized and then invokes the corresponding enclave function. Next, the trusted part performs multiple functions to initialize the wallet. Figure 4.2 illustrates the sequence of steps involved in the wallet initialization.

1. *Mnemonic and root seed*: Generates a mnemonic phrase comprising of 12 randomly chosen words. From the mnemonic phrase, it derives a 512-bit root seed as specified by the bip0039 standard [39].

2. *Master and parent node*: Next, it executes the HD protocol [55] and derives the master public and private key pair. Using the account index in the HD wallet path input, the parent key-pair associated with the account is derived from the master key-pair.

3. *Key usage tracking*: To enforce sign-once semantics, it is required to track the keys that have already been used for signing. In HD protocol, a key-pair is identified by an unique index that is part of the wallet path. It is, therefore, sufficient to track just these indices and protect them from being tampered by the user. To do this, the enclave maintains the ranges of unused address indices which are usually the non-hardened keys of range 0 to 2^{31}. The last bit of 32-bit is used to indicate the hardened key index. The non-hardened key range and the root seed are then sealed to prevent disclosure and tampering. Since the enclave does not have any permanent storage, the sealed contents must be safely stored by the caller. Also every time a transaction has to be signed, this sealed data has to be passed to the enclave to check key-reusage. However in this approach, the caller has complete control over the sealed contents and is the one who sends the sealed data to sign a transaction. He/she can attempt to send in a previous version of the sealed data and trick the enclave to reuse the key, leaving the enclave no way to determine if the sealed contents are being replayed.

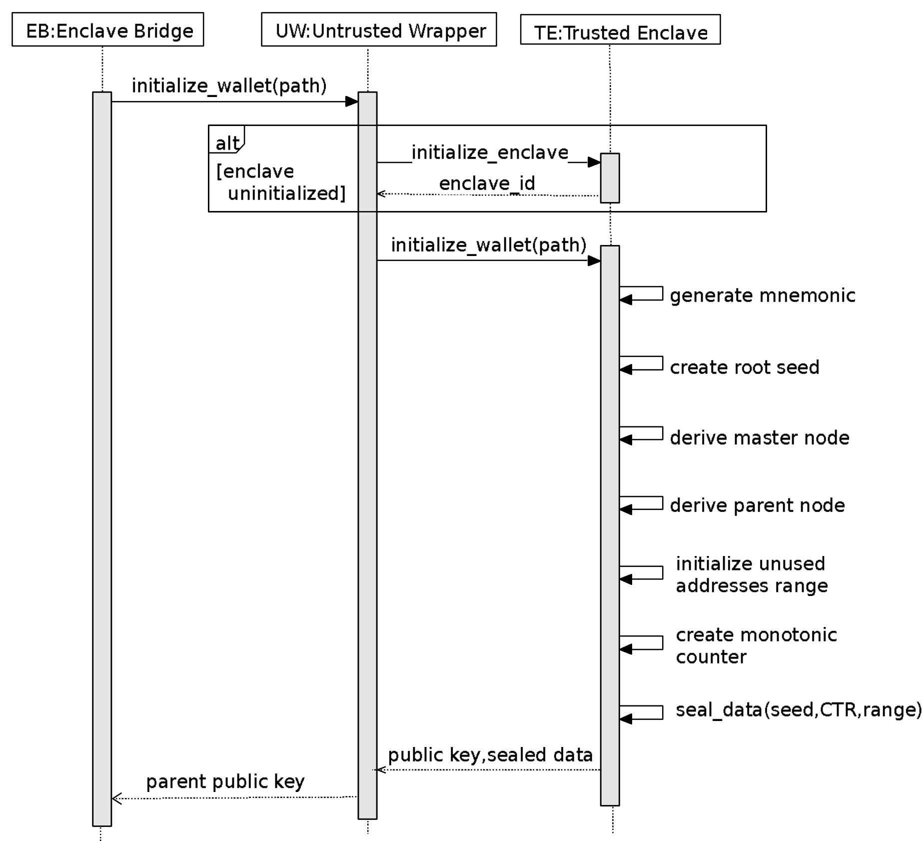

Figure 4.2: Wallet Initialization - Sequence of Operations

4. *Replay prevention*: To prevent adversary from replaying the sealed contents, the changes to the unused address range are tracked via a version parameter that is incremented each time the range changes. It is then sealed along with the seed and the unused address range and is also tracked inside the enclave using enclave-specific monotonic counter (Section 2.2.2). These two values (one in the sealed buffer passed from untrusted part and the other is the monotonic counter) are always compared in all the sign transaction calls to prevent replay attacks.

5. *Sealing*: The seed, monotonic counter and the unused address range elements are then sealed with the enclave-specific key. The seed is not sealed separately (different buffer) to prevent the caller/adversary from using the unused address range of a different wallet. Finally, the sealed buffer and the unsealed parent public key are returned to the caller

which, in turn, returns to the transaction manager.

Signing Transaction functionality gets the transactions signed by deriving the corresponding private keys. The transaction manager component invokes the library by passing in a list of inputs, outputs and previous transactions. The library needs to return a list of signatures and the remote attestation quote as outputs with the number of signatures returned equal to the number of UTXOs (or inputs) being spent by the proposed transaction. To do this, both the untrusted and the trusted parts perform many crucial operations critical to the safety and security of the system as described below. Figure 4.3 depicts the complete sequence of operations.

1. The ***untrusted part*** validates the inputs and outputs of a transaction by verifying if each input included in the transaction is referring to a valid UTXO. If the inputs are invalid, an appropriate error message is returned to the caller and the function exits.

2. For each input, the untrusted part invokes the trusted enclave function to get the signature. Each time the untrusted part needs to retrieve the previously saved sealed contents and pass it as input to the enclave.

3. The ***trusted part*** unseals the sealed buffer passed in as input and executes the next set of steps. All the following steps are executed in a single enclave invocation and the output is returned only at the end of the invocation. This ensures the atomicity of the sequence of operations i.e., if any of the operations fail, no output is returned, instead the enclave will return only an error.

4. *Rollback and Replay prevention:* It retrieves the version counter from the unsealed data, reads the monotonic version counter value from the enclave and compares it with the one obtained from the unsealed contents. If the versions do not match, an error message is returned to the caller. This checking prevents the adversary from rolling back to the previous versions and replaying the old sealed data.

5. *Sign-once enforcement:* It retrieves the unused address range from the unsealed data. It then obtains the address index (index representing the child key-pair) from HD wallet path and checks whether the index is in the unused address range. If it is in the range, the index can be used for signing the current transaction and thus it updates the unused address range by removing the particular index and proceeds. If not, the transacting party is trying to reuse the key in which case the function returns an error message to the caller and signing terminates.

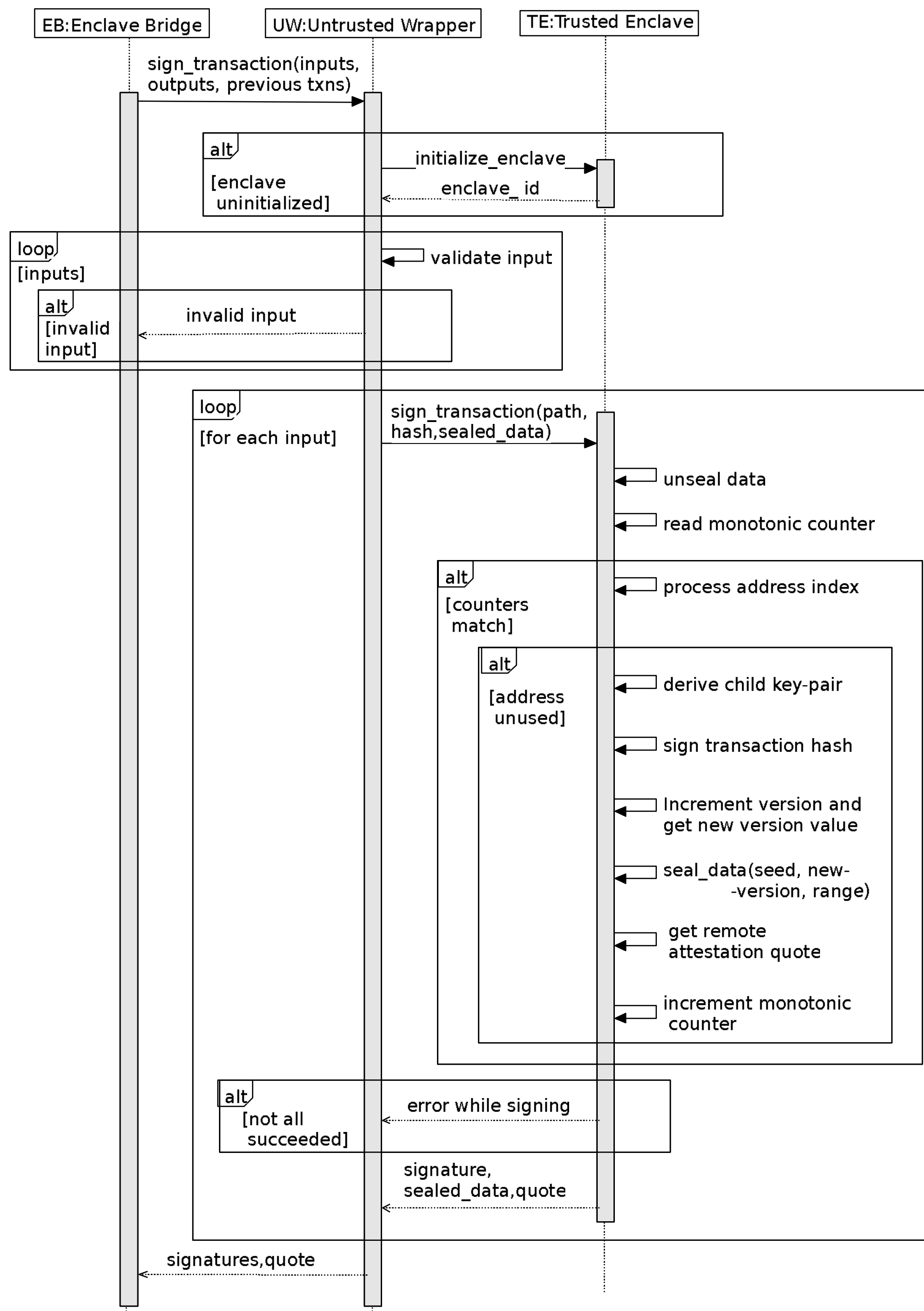

Figure 4.3: Sign Transaction - Sequence of Operations

6. *Sign transaction:* It reads the seed from the unsealed buffer. Using the seed and the address index from the HD wallet path, it derives the child key-pair. Then, it signs the transaction hash with the child private key to obtain the signature.

7. *Track changes:* After signing, the previous version counter (retrieved from the unsealed data) is incremented to track the changes to the unused address range (step 5) and the new version value is read. Note that at this point the monotonic counter within the enclave is not yet incremented.

8. *Sealing:* It then seals all the data - seed, new counter value and the updated unused addresses range.

9. *Remote attestation quote:* Next, it obtains the remote attestation quote from the quoting enclave with the quote containing the hash of the transaction (Trasaction ID) in place of user data as illustrated in Figure 2.13. This quote proves two things to the payee. First, it provides a measurement of the trusted piece of code running within the enclave that manages the wallet initialization and signing. When an adversary tries to use modified code in a different enclave, this measurement differs and can be easily detected. Second, the hash within the quote confirms that the particular transaction has been signed inside the enclave and is subjected to the sign-once rule. This also prevents the adversary from replaying the same quote for different transactions and deceive the payee.

10. *Increment enclave counter:* The monotonic counter of the enclave is then incremented and the version is persisted in the enclave after obtaining the signature and the quote. This reduces the risk of losing the signatures permanently and eventually locking the bitcoins in case of crashes. Now if the system crashes before incrementing the counter, the payer can re-invoke the function to get the signatures for the same transaction again. This is discussed further in Section 6.5.1.

11. Finally, it returns the sealed contents, attestation quote and the signature to the untrusted part as outputs.

Chapter 5

Implementation

In this chapter, we explain the implementation of our design, outline the various decisions taken during the process and also specify the pseudo code and the technical details of the algorithms implemented in our system.

5.1 Implementation Approach

In our design, the SGX enclave (Section 4.2.3) is the important component that realizes our goal of preventing double-spending and provides the necessary trust guarantees in our system. With this in mind, we made the following implementation decisions.

1. Choose an existing wallet that provides the same or equivalent functionality of the transaction manager component as discussed in Section 4.2.1.

2. Implement the enclave functionality of the wallet as an independent module so that it can be integrated with any of the existing wallets with only minimal changes.

3. Include only the least required code within the trusted enclave and carry out all other operations that do not deal with any private key information outside the enclave. This is to minimize the number of calls to the enclave because every call to the enclave involves the overhead of data duplication (untrusted to trusted memory and vice-versa) and memory management given the enclaves have limited resources.

5.2 Transaction Manager and Copay

The transaction manager is responsible for executing user instructions and communicating with the Bitcoin network. One of the key decisions that influences the security guarantees of our system is selecting an existing wallet in line with our requirements. Based on the results from different sources[1], Copay, MyCelium, Trezor, JAXX and CoinBase wallets were considered. After a broader analysis of the features offered by these wallets and with our requirements in mind, we choose *Copay* wallet from Bitpay. Copay is a multi-signature HD wallet from Bitpay with secure-payment verification and is open-source. It supports multiple distinct wallets and provides isolation between them. The following are the major reasons for choosing Copay over other wallets.

1. Copay from Bitpay[2] is one of the most popular highly-rated bitcoin wallets with a large user base. This helps introduce the SGX wallet to a larger section of users.
2. Copay incorporates most of the latest bitcoin improvement suggestions and its operation is based on the HD protocol. Most importantly, the private keys are never accessed or stored on the Bitpay servers and are completely under the control of the user.
3. Copay code is open-source and the application is available on all the platforms including Android and iOS. The same code is used across both desktop and mobile applications.
4. Hardware-based wallets, namely Trezor and Ledger, are already integrated with Copay. This facilitates easier integration of the SGX enclave with Copay.
5. Also, Copay supports Testnet integration. Testnet [49] is a no-value bitcoin network used for testing by Bitcoin developers. The testnet transactions are recorded and tracked in a separate blockchain maintained by various developers.

---

[1] `https://99bitcoins.com/best-bitcoin-wallet-2015-bitcoin-wallets-comparison-review`
`https://bitcoin.org/en/choose-your-wallet`
`https://www.cryptocompare.com/wallets/#/overview`

[2] `https://copay.io/`

The user interface functionality of Copay is coded using HTML, CSS and AngularJS using the ionic framework. The NodeJS module bitcore-wallet-client is responsible for communicating with Bitcore-Wallet-Service (BWS) via simple asynchronous REST APIs and verifying the results and responses. BWS takes care of wallet creation, transaction broadcasting, data exchange with the peers, and communicating with the Bitpay servers to store the wallet identification information and other public data. Bitpay servers, however, do not exercise any control over the private or sensitive data. Copay code and further details are available at [8].

In our system, Copay provides the necessary networking, address generation and distribution functionality and acts as the transaction manager of the wallet for both payer and payee. In case of payer, it gets the remote attestation quote from the enclave and displays it as QR code to the payee. While, the payee wallet scans the QR code containing the sign-once guarantee and the signed transaction hash and verifies it using the IAS to get the necessary assurance.

5.3 SGX Enclave Library

The library consisting of two parts is implemented in C as a dynamic library on a 64-bit Linux system running on an SGX-enabled Intel Skylake processor. The untrusted and the trusted parts together are responsible for a number of functions. Many of these functions are standard Bitcoin procedures that are already implemented and being used in the existing wallets. We, therefore, decided to reuse one of these existing libraries and modify them to suit our requirements. We adapted the open-source crypto library from Trezor[3]. It is a highly optimized C library implementing HD protocol that includes mnemonic and seed generation, private and public key derivations, and transaction signing and serialization. We chose to modify the Trezor library over others because it is a C library and has already been integrated with Copay. The following subsections explain the untrusted and trusted modules of the library.

5.3.1 Untrusted Module

This provides the APIs for interaction with the external environment and encompasses a number of tasks. Listing 5.1 lists the two major APIs provided

[3]`https://github.com/trezor/trezor-crypto`

by the external interface of the untrusted module.

API `getAccountPublicKey` is used to get the parent public key of the wallet account from the trusted enclave and is invoked by Copay via the bridge. This key is fetched and stored on the Bitpay server and is used for the wallet identification and deriving the bitcoin addresses (by deriving the child public keys). It accepts the following arguments.

1. `hd_path` - An array of 32-bit unsigned integers with each number representing an index within the HD wallet path.

2. `hd_path_size` - is a 8-bit value that represents the actual length of the index array.

3. `coin_type` specifies the network type such as Bitcoin, Testnet or others.

4. `pub_key` is an OUT parameter into which the resultant public key fetched from the enclave is copied and returned to the caller.

After preparing the inputs, this API invokes the trusted enclave function to initialize the wallet and get the parent public key. In addition to the public key, this API also persists the sealed contents (described in Section 4.2.3) returned by the enclave.

```
uint8_t getAccountPublicKey(
              uint8_t hd_path_size, uint32_t* hd_path,
              uint8_t coin_type, PublicKey* pub_key);

uint8_t signTransaction(
    uint8_t tx_ips_len, TransactionInput* arrInputs,
    uint8_t tx_ops_len, TransactionOutput* arrOutputs,
    uint8_t tx_ref_len, RefTransaction* arrRefTx,
    uint8_t coinType, uint8_t* quote,
    SignedTx* signedResult, TxSignature* arrSignatures);
```

Listing 5.1: SGX Library External Interface

API `signTransaction` is invoked by Copay to get a transaction proposal signed. It requires the following arguments to be passed in by the caller.

1. `arrInputs` is the list of inputs included in the transaction. Each input structure contains various members as discussed in Section 2.1.2. The size of the array is represented by `tx_ips_len` argument.

2. `arrOutputs` is the list of transaction outputs with each output paying a certain amount of bitcoins to a particular address. Argument `tx_ops_len` indicates the number of outputs in the list.

3. `arrRefTx` is the list of previous transactions that are used for validating the inputs and outputs. `tx_ref_len` parameter represents the number of transactions in the list.

4. `coinType` determines the network and cryptocurrency coin type.

5. The OUT parameter `signedResult` stores the resultant serialized signed transaction, its length, number of signatures and the length of the remote attestation quote. The actual list of signatures and the quote are stored in `arrSignatures` and `quote` parameters respectively.

Apart from invoking the enclave functions for signing the transaction, this API is also responsible for the following functions described below.

1. ***Transaction Validation***: For each input, it retrieves and hashes the previous transaction data (inputs and outputs) from `arrRefTxs` and compares this hash with the `Previous tx_hash` parameter value included in that particular input. This verifies if each input included in the transaction is referring to a valid UTXO.

2. ***Transaction Fees***: It also verifies if the sum of all the inputs (UTXOs) is greater than or equal to the sum of total spending (all the outputs) and the standard transaction fee (collected by the miner).

3. ***Transaction Scripts***: Scripts are created to be included in the outputs and inputs of the transaction after validation. They are signature script `scriptSig` and the verification script `pubKey` for different script types described in Section 2.1.3.

4. ***Hashing and Serialization***: Bitcoin transaction is signed by signing the hash of its contents. The major algorithms used at various instances of hashing are SHA256, SHA512, RIPEMD-160 and HMAC with SHA256/SHA512. Also after signing, the transaction is serialized to be sent back to the caller.

5. ***Storage and Retrieval***: This API also manages the storage and retrieval of the sealed contents obtained from the enclave. In the event of error at any stage, the API exits and an appropriate error message is returned to the caller.

5.3.2 Trusted Enclave Module

The enclave is responsible for performing various trusted operations such as seed generation, private key derivation, signing transactions, sealing information and getting the remote attestation quote. The enclave defines the interface for these operations enabling interaction between the trusted and the untrusted modules. Such a trusted interface in SGX is known as Enclave Definition Language (EDL) file and consists of the various enclave calls (ECALL - untrusted to trusted) and output calls (OCALL - trusted to untrusted). Listing 5.2 lists the ECALLs of our trusted enclave.

```
trusted {

    public uint8_t ecall_init_wallet(
       [in] GetPublicKey* reqInputs,
       [out] HDPublicKeyNode* parentNodeInfo,
       [out,size=1024] uint8_t* sealed_buf,
       [out] uint32_t* sealed_data_size,
       [out] PublicKey* resp);

    public uint8_t ecall_sign_transaction(
       [in,size=32]uint32_t* address_n,uint32_t addr_len,
       [in,size=32] uint8_t* hash,
       [in,out,size=buf_len] uint8_t* sealed_buf,
       uint32_t buf_len,uint32_t ip_seal_data_len,
       [out] uint32_t* op_seal_data_len,
       [out,size=64] uint8_t* signature,
       [out] uint8_t* quote, [out] uint32_t* quote_len);
};
```

Listing 5.2: Trusted Enclave Calls (EDL)

ECALL ***ecall_init_wallet*** API initializes the wallet and performs the various tasks as explained in Section 4.2.3. It accepts the following list of arguments.

1. Argument `reqInputs` is a structure containing the HD wallet path representing the account and the coin type. The HD wallet path is used for parent key-pair derivation for the specified coin type.

2. All other parameters are OUT parameters that store the results returned by the function back to the untrusted caller. `parentNodeInfo` and `resp` contains the parent public key information derived inside the

enclave. `sealed_buf` holds the sealed seed, monotonic counter and the unused address range (initialized to 0 to 2^{31}) whose length is assigned to another OUT parameter `sealed_data_size`.

To generate a random number, create a new monotonic counter for the enclave and seal the data within the enclave, the following functions from the SGX SDK are used.

1. `sgx_read_rand` from `libsgx_trts` library - Generates a new random number.

2. `sgx_create_monotonic_counter_ex` - Creates a new monotonic counter with its value initialized to zero.

3. `sgx_seal_data_ex` - Encrypts the input data using the sealing key specific to the enclave.

ECALL ***ecall_sign_transaction*** API signs the supplied transaction digest. The enclave executes the series of steps as explained in Section 4.2.3. This API accepts the following list of arguments.

1. Argument `address_n` is an array of 32-bit unsigned integers with each number representing an index within the HD wallet path. This path is used to derive the corresponding child private key later used for signing. The actual length of this array is represented by `addr_len`.

2. `hash` is a 32-byte SHA256 hash of the content of the transaction that needs to be signed.

3. `sealed_buf` is a static buffer of fixed length `buf_len`. It is an IN-OUT parameter that carries the previously returned sealed data of length `ip_seal_data_len` as input to the enclave. It is used to verify that the key derived using the input HD wallet path (`address_n`) is not being reused for signing. If the signing succeeds, the buffer is flushed and is filled with the new set of sealed data. This new sealed data consists of the seed, updated version (or monotonic counter), and the updated unused address range. The length of the sealed data thus obtained is assigned to the `op_seal_data_len` OUT parameter.

4. The resultant signature along with the attestation quote from the quoting enclave are copied into the `signature` and `quote` OUT parameters respectively. The length of the quote is then assigned to the `quote_len` OUT parameter.

Unused key range tracking - To verify that a particular key is not used more than once for signing, we devised an algorithm to store and track the index ranges of unused addresses. As explained in Section 4.2.3, it is sufficient to just track the indexes representing the signing keys. The aim of this algorithm is to process and store these ranges in a storage-efficient manner. When signing a transaction, the algorithm is invoked to check if the new index is within the unused address range. If yes, it is excluded from the unused range after signing, otherwise the signing terminates with an appropriate error. All these operations happen in a single ECALL to ensure atomicity. Alternatively, the algorithm can also be modified to track the used address indices. Listing 5.3 represents the pseudocode of the algorithm and also includes the structure used to store the range of indices.

The key reuse tracking algorithm used in our approach is space-efficient due to the following reasons.

1. The system only stores the 4-byte index of the address rather than storing the entire 20-byte address.

2. Further, it only stores the ranges of unused indices. This eliminates the necessity to store the indices of all the unused addresses (or alternatively used addresses). For example, to store the unused addresses corresponding to indices 1, 2, 3, 4 and 5 (20 bytes), we only store 1 (minimum) and 5 (maximum) thus saving 12 bytes of space.

3. One of the naive approaches to track unused indices is to store all the unused indices. Given the range of key indices is `0` to 2^{31}, such an approach requires `4 x` 2^{31} bytes of storage space. Our algorithm does not require more than this amount of space even in the worst-case wherein it is necessary to store every alternate index in the form of (1,1), (3,3), (5,5) till 2^{31}.

The following trusted functions from the Intel crypto library are used to unseal, read and increment monotonic counter, sign transaction and get the remote attestation quote within the enclave. The procedure for getting the remote attestation quote is yet to be implemented.

1. `sgx_unseal_data` - Decrypts the input sealed content and returns the unsealed buffer provided that the sealed content passed as input is not tampered.

2. `sgx_read_monotonic_counter` - Reads the value of the counter persisted within the enclave. The inputs to be passed in are counter UUID

```
Struct IndexRange {

   uint32_t min;
   uint32_t max;
}

Algorithm ProcessIndex
Inputs: newIndex, prevRangeArray, arrayLength
Outputs: newRangeArray, newLength

set j <- 0
set k <- 0
set indexProcessed <- false
loop j <- 0 to (arrayLength-1) do
   if indexProcessed = false AND prevRangeArray[j].min
                   <= newIndex <= prevRangeArray[j].max then
       if newIndex = prevRangeArray[j].min = prevRangeArray[j].max then
           set indexProcessed <- true
       else
           if newIndex = prevRangeArray[j].min then
               set prevRangeArray[j].min <- (prevRangeArray[j].min+1)
               set newRangeArray[k].min <- prevRangeArray[j].min
               set newRangeArray[k].max <- prevRangeArray[j].max
               set k <- (k+1)
           else if newIndex = prevRangeArray[j].max then
               set prevRangeArray[j].max <- (prevRangeArray[j].max-1)
               set newRangeArray[k].min <- prevRangeArray[j].min
               set newRangeArray[k].max <- prevRangeArray[j].max
               set k <- (k+1)
           else
               set newRangeArray[k].min <- prevRangeArray[j].min
               set newRangeArray[k].max <- (newIndex-1)
               set k <- (k+1)
               set newRangeArray[k].min <- (newIndex+1)
               set newRangeArray[k].max <- prevRangeArray[j].max
               set k <- (k+1)
           endif
           set indexProcessed <- true
       endif
   else
       set newRangeArray[k].min <- prevRangeArray[j].min
       set newRangeArray[k].max <- prevRangeArray[j].max
       set k <- (k+1)
end loop

set newLength <- k
if indexProcessed = true then
   return 0
else
   return 1
endif
```

Listing 5.3: Pseudocode of Unused Key Index Range Tracking Algorithm

and the nonce unique to that particular counter. These values can be retrieved from the unsealed buffer.

3. `sgx_increment_monotonic_counter` - Increments the value of the monotonic counter. The inputs to be passed in are counter UUID and the nonce.

4. `sgx_ecdsa_sign` - Signs the transaction with the private signing key derived using the HD protocol.

5. `sgx_get_quote` - Get the verifiable quote of the enclave (measurement) to be returned to the caller.

5.4 Integration Overview

The SGX enclave library is integrated and its functions are invoked from Copay using a proxy bridge, ***sgx-connector***. It is a NodeJS module coded to use ***node-ffi***[4]. *node-ffi* is a NodeJS addon capable of loading and calling the native C functions within the dynamic libraries using pure JavaScript. The sgx-connector bridge component takes care of the following functionalities.

1. It validates and sanitizes the inputs received from Copay.

2. Parses and converts the inputs passed to appropriate structures that emulate the C-style structures used within the enclave library.

3. Synchronously invokes the enclave APIs and fetches the results and error conditions. Converts these results to appropriate structures to be passed back to Copay.

The sgx-connector bridge component uses *express* framework in NodeJS to provide the REST interface APIs for invocation from Copay. These APIs are invoked asynchronously from Copay to fetch the public key of the account and the signatures of the transaction. The inputs and results are shared between the components using JSON objects[5].

[4] `https://github.com/node-ffi/node-ffi`
`https://www.npmjs.com/package/node-ffi`

[5] The implementation is complete and the integration testing is in progress

Chapter 6

Evaluation

In this chapter, we assess and analyze the functional and security aspects of the proposed solution and the prototype implementation. First, we evaluate the conformance of the proposed solution to the requirements. Second, we explain how our system is resistant to side-channel attacks and why the side-channel analysis does not affect our design. Third, we quantify the TCB of our system and the associated benefits of minimizing the TCB.

Our evaluation environment consists of a 64-bit Linux machine running Ubuntu 14.04 LTS with an SGX-enabled Intel Skylake processor. We integrated our prototype SGX enclave C library with the Copay Chrome App version. The following sections discuss the details of our evaluation.

6.1 Requirements Conformance

One of the key approaches for security assessment of a system is to determine the relevance of the proposed solution and implementation with respect to the defined set of requirements. The following paragraphs describe the conformance to various requirements outlined in Chapter 3.

6.1.1 Security Conformance

The following list presents the evaluation of the system with respect to the security requirements (Section 3.2).

1. ***S1-Sign-once semantics:*** To enforce the sign-once rule and prevent key reusage, the trusted enclave tracks the key usage for signing by storing the unused address indices. During initialization of the wallet, the enclave initializes this unused address range and seals this data along with the other parameters. At this point in time, the unused

range will be 0 to 2^{31} as no keys have yet been used for signing. Every time a transaction has to be signed, the trusted enclave ensures that the key index to be used for deriving the corresponding signing key is checked against the unused address range using the algorithm in Listing 5.3. These trusted operations deter key reusage and impose sign-once restriction.

2. ***S2-Signing keys protection:*** To impede an attacker from attempting to access and modify the signing and private key information, keys needs to be safeguarded. This is achieved by sealing two important data items a) root seed that is the source for all the key derivation b) index range of unused addresses (or keys). The sealing is done using the enclave-specific keys that are hardware-protected and are accessible only within the enclave. Sealing provides the necessary confidentiality and integrity protection. Also, all the operations that deal with the private key information are performed inside the enclave.

3. ***S3-Sign-once Guarantee:*** To induce the confidence of the payee that the payer is using the SGX wallet and cannot double-spend the coins, the enclave generates a verifiable sign-once guarantee. This guarantee is then returned to the payer as a *quote* from the enclave after signing every transaction (refer to Section 5.2). The payer displays it to the payee who can verify its trustworthiness by using the IAS.

6.1.2 Usability Conformance

The following list presents the evaluation of the system with respect to usability requirements (Section 3.3).

1. ***U1-No Additional Devices:*** Our solution does not require any additional external devices to store and protect the private wallet information apart from a wallet-installed device supported by the off-the-shelf hardware. This eliminates the necessity of the wallet user to carry other external devices for signing the transactions unlike some of the existing hardware-based wallets.

6.1.3 Deployability Conformance

As far as deployment is concerned, the proposed wallet is currently integrated with Copay bitcoin wallet. Given the independent design of the SGX enclave library, it can be integrated with most other wallets in place of Copay. The only requirement for its integration with a particular wallet is to replace the

enclave bridge component with a wallet-specific proxy component. Currently, most of the wallets, including Copay, do not provide any hardware-backed guarantees for storing and protecting the seed and key information. The following list presents the evaluation of the system with respect to deployability requirements (Section 3.4).

1. ***D1-No Protocol Modifications:*** Our solution involves only client-side (user) wallet modifications to share the verifiable guarantee as a QR code in case of a payer and to verify the remote attestation quote in case of the payee. It requires only a wallet upgrade, provided the underlying hardware provides a TEE, and does not require any changes or upgrades to the base Bitcoin protocol, miners or nodes in the Bitcoin network. It is therefore compatible with the exiting Bitcoin deployments.

2. ***D2-Off-the-shelf Hardware:*** Our proposal works using the TEE provided by the underlying hardware and our prototype implementation can be run on any SGX-enabled Intel processor.

3. ***D3-Incremental Deployment:*** Copay supports the integration of several distinct wallets such as Trezor, Ledger and other multisignature wallets. By integrating our solution, the other wallets operating using the existing mechanisms are not disrupted. Our wallet operates independently in the Copay using the underlying TEE.

6.2 Attacks and Countermeasures

While evaluating the security of the system, it is important to analyze it from the attackers perspective and determine if the system is capable of withstanding plausible attacks. Table 6.1 maps the attempts by adversaries to the corresponding countermeasures.

6.3 Side Channel Resistance

SGX enclaves are prone to side channel attacks. The type of attack determines the results retrieved by the attacker and its impact. Deterministic controlled-channel attacks [56] use input-dependent page faults to infer sensitive information using memory access patterns. This does not affect our proposed solution because the process of key derivation and signing the transactions are neither secret nor input-dependent, but rather remains the same for any input.

Attack	Countermeasure
Attempting to reuse keys	Key usage tracking and sign-once verification
Read or modify key information	Confidentiality and integrity protection with sealing
Decrypt key information by adversary	SGX sealing key unavailable outside the enclave
Decrypt key information with different enclave to recover seed and private key information	Sealing key specific to enclave and cross-enclave unsealing prevented
Replay old key information and attempt to reuse keys for signing	Tracking changes to sealed information via version parameter and SGX monotonic counter
Invoke initialize wallet, get initial unused address range and use it as sealed input to sign transaction. Cross wallet key information usage	Possible when seed and unused address range are sealed separately. Averted by sealing all of them to a single blob.
Replay same remote attestation quote for different transactions	Quote contains the hash of the transaction (Transaction ID)

Table 6.1: Attacks and Countermeasures

The side-channel attack proposed by Brasser et al. [12] is capable of extracting the private key information by exploiting the vulnerabilities in the existing implementation of modular exponentiation algorithms. However, sources from Intel [30] state that the issue is fixed in the Intel IIP crypto library and the corresponding functions are hardened against such attacks. Usage of such hardened library ensures protection of keys used for signing transactions and makes our solution resistant against potential side-channel attacks.

6.4 Minimizing Trusted Computing Base

The core set of firmware, hardware and software components that work together to enforce the security policy of a system is called the Trusted Computing Base (TCB) [11]. These components protect and ensure the security of the entire system. It is therefore important to ascertain the correctness of various components of the TCB. The larger the size of TCB, the greater the risk of bugs and security vulnerabilities.

In our implementation, the trusted enclave is the only software component part in the TCB. The enclave performs a number of security critical operations and for these it uses the functions provided by the SGX SDK with the aim of minimizing the TCB footprint. Currently, the trusted interface exposes only two ECALLs. The number of lines of logical source code (SLOC-L) including all the helper functions used for the enclave operations is **1560** in `C` excluding the Intel library functions[1].

6.5 Limitations

Though the proposed wallet successfully prevents double-spending, it does have some limitations as explained in the following paragraphs.

6.5.1 Resilience to Crashes

In our solution, the enclave signing operation is not fully resilient to crashes resulting from hardware or power failure in the following cases. Both situations could render the associated bitcoins unspendable leading to bitcoin-locking since the enclave does not allow re-signing with the same keys.

1. If the system crashes after signing the transaction and incrementing the monotonic version counter inside the enclave but before exiting from the enclave and returning the sealed data, signature and quote back to the untrusted caller.

2. If the system crashes after signing the transaction and exiting the enclave but before returning the signature and quote back to the transaction manager.

To address the first issue, the system is designed to reduce the window of crash by incrementing the monotonic counter only after getting the signature

[1]The process of further minimizing the TCB is a work in progress

and the remote attestation quote as depicted in Figure 4.3. This increases the resilience of the system even though all the operations still happen in the same enclave invocation. The second issue can be addressed by temporarily backing up the signature and quote as soon as it is returned from the enclave. In case of a crash, the same can be retrieved from the back-up and returned to the transaction manager. The back-ups can be periodically removed.

6.5.2 Backup and Recovery

Currently, our solution does not take into account the wallet recovery and restoration of bitcoins associated with it. This is necessary in situations such as accidental deletion of the wallet file or loss or corruption of the wallet data. The existing wallets facilitate import, backup and recovery by revealing the mnemonic word phrase to the users. However in our threat model, the wallet user is a potential adversary and revealing the mnemonic phrase risks double-spending. Hence, it is necessary to devise a solution that provides the restore functionality without revealing any sensitive information to the wallet user. Chapter 7 discusses a potential solution to address this limitation.

Chapter 7

Variations and Extensions

In this chapter, we discuss the possible enhancements and extensions to our system.

7.1 Multiple Accounts

The current prototype implementation tracks a single account wallet. However, the proposed solution can be used even when multiple accounts are derived from the same seed. To accommodate this use case, the untrusted module needs to pass the previously saved seed data during wallet initialization for the subsequent accounts. Also, the algorithm (refer to Listing 5.3) devised to store and track the unused address ranges needs to be modified to keep track of multiple accounts. The structure must store account-specific unused address ranges. This allows the user to have multiple accounts in the same wallet with all the keys being derived from the single seed.

7.2 Remote Attestation Verification

In the current integration with the Copay wallet, the remote attestation quote is shared with the payee or merchant and he/she is responsible for validating the trustworthiness of the quote by contacting the IAS. Our solution can be extended by having one of the Copay servers verify the quote and send the payee a confirmation stating that the transaction is safe from double-spending. By doing this, the wallet users (payee in this case) are not burdened with the verification process and related data exchange or network consumption.

7.3 Bloom Filters for Storage

To track the key usage, we store the ranges of indexes of the unused addresses. Though we are storing only the indices, in case of multiple accounts this might be a cause of concern given the fact that SGX enclaves have limited resources. Instead of using arrays, we could employ a bloom filter to store this since bloom filters are highly storage efficient. However, one disadvantage of using bloom filters is the false positives that result in bitcoin locking. To tackle this, we could employ two bloom filters as explained below.

1. One bloom filter stores the key indices of the addresses containing the spendable bitcoins in the untrusted part of the system. This is called the ***Load Address Filter***. Every time a new external address is generated, the wallet user sends the specific index to be stored in the load filter. If a collision occurs during storage, the load filter throws a false positive error. In such an event, the user generates a new address and repeats the process to check for false positives. If not, the index is inserted into the load filter and the address is shared with the external party or the user herself loads the bitcoins to this address from other wallets. Otherwise, it is skipped and a new address is generated. Thus, the load filter stores the indices that do not cause false positives.

2. The other bloom filter, ***Used Address Filter***, stores only those indices that have *already been used* for signing the transactions. This prevents false positives completely given that the transaction is signed with a key that is loaded with some bitcoins and the load filter stores only those indices that do not cause false positives. This filter is sealed using the enclave-specific key and hence cannot be tampered or accessed by the adversary.

This approach prevents bitcoin locking without affecting the security of the system since the used addresses are still protected by the enclave. Even then, it is necessary to evaluate the performance overhead of using the bloom filters and the amount of time taken for signing a transaction.

7.4 Backup and Recovery

The wallet needs to be protected against unanticipated system crashes to prevent the users from losing bitcoins. Currently, our solution does not support recovery in case of losing or deletion of the wallet data file. One trivial

solution to tackle such problems is to take backups which is not practically convenient.

To facilitate recovery in case of complete system failures, we propose a potential solution using a central server, similar to Copay server, with an SGX TEE. The server is responsible for securely storing the mnemonic phrases of the wallet users. The data is confidential and integrity-protected by the server enclave and is only accessible to the enclave. Also, all the sensitive data exchange happens directly between the two enclaves (server enclave and wallet-user enclave) over a secure channel. The following steps are involved in a typical recovery scenario.

1. When a wallet is initialized for the first time, the user enclave retrieves the mnemonic phrase and the remote attestation quote.
2. Next, the user and the server enclaves mutually authenticate each other using their respective attestation quotes.
3. Then, both the enclaves establish a temporary secure channel using the Sigma protocol [18] based on Ephemeral Diffie-Hellman Key Exchange.
4. The user enclave shall then send the unique wallet identifier, remote attestation quote and the mnemonic phrase all encrypted with the session key over the secure channel.
5. The server enclave decrypts the data. It then encrypts the quote and the mnemonic phrase using the server enclave-specific sealing key and stores it along with the user-wallet identifier.
6. When a system fails, it can be revoked using Intel SGX revocation service. This revocation prevents the user enclave from providing valid quotes to verifiers. establishes a proof that the system has failed for real and thus the wallet user cannot deceit or attempt to double-spend.
7. After revocation, the user can now create a new enclave, mutually authenticate with the server enclave, establish a secure communication channel and request for the mnemonic phrase.
8. The server enclave shall then check that the previous enclave was indeed revoked by using the earlier stored remote attestation quote.
9. After successful verification, the server enclave then shares the mnemonic phrase, encrypted using the session key, with the user enclave.
10. The user enclave decrypts the data. It then restores the lost wallet with the mnemonic phrase using the existing Bitcoin recovery mechanisms.

Chapter 8

Conclusions and Future Work

The growing demand for financial privacy led to the proliferation of cryptocurrencies that provide secure and decentralized execution of monetary transactions. The recent introduction of Bitcoin accelerated the adoption and usage of these cryptocurrencies. However, the inherent vulnerability of double-spending in digital currencies and the long delay in transaction confirmation makes the blockchain-based cryptocurrencies unsuitable for real-world trading. Such an impending *double-spending attack* also poses serious risks to the payee (or merchant). He incurs financial losses if accepting unconfirmed payments, or makes the payer wait which may result in losing the customer. This prevents instant bitcoin payments and hinders the wider acceptance of blockchain-based cryptocurrencies in society.

In this thesis, we propose a solution to address the double-spending attack and thus enable instant bitcoin payments. Our system prevents double-spending by ensuring that the payer cannot use the same bitcoins in more than one transaction. We do this by enforcing *sign-once semantics* that prevent the payer from reusing the keys to sign multiple transactions. For tracking the unused keys, we devised an algorithm that efficiently processes and stores the unused index ranges. In order to prove that the transacting payer cannot double-spend, the payer also provides a reliable and verifiable *sign-once guarantee* to the payee. The security of our system arises from the assurances of trusted execution environments. We implemented a proof-of-concept using Intel SGX technology.

Unlike other solutions, our proposal only involves changes to the wallet application and does not require any modifications to the cryptocurrency networking ecosystem (nodes and miners) or protocol. Our system is therefore simpler and compatible across existing deployments of any blockchain-based cryptocurrency. Also, the verifiable guarantee by the payer allows

the payee to accept the unconfirmed payments without the risk of losing bitcoins. This encourages faster bitcoin payments similar to the traditional credit-card based payments wherein the merchant still receives the actual funds only after the transaction processing or confirmation. In exchange, the payers (or buyers) could avail discounts from the merchants who are keen on increasing their customer base. Thus, our solution eliminates the financial losses incurred by the payees and significantly reduces the transaction times of bitcoin payments, paving the way for increased adoption of Bitcoin and blockchain-based cryptocurrencies.

8.1 Future Work

The previous chapter described the various extensions of our system. In the following paragraphs, we discuss other possible areas of exploration to improve the security and efficiency of blockchain-based cryptocurrencies.

Using other TEEs: We have implemented our proof-of-concept using the Intel SGX TEE. One possible direction for future work could be to implement the proposed solution using other TEEs such as ARM TrustZone. TrustZone provides complete isolation of the secure world from the non-secure world each having its own hardware, software and operating system components. This makes the system completely side-channel resistant. However, TrustZone does not provide any in-built secure storage and relies on other secure elements [50]. Also, TrustZone does not by default provide remote attestation. While attestation services are usually provided by trusted OS vendors, these may not be standardized and may also incur usage charges.

Transaction Verification: At present, the Bitcoin peers validate the transactions to be embedded in the blockchain by storing and tracking the UTXOs in an indexed set. This requires more storage space and involves a maintenance overhead at each node. A possible extension could be to verify the transactions using the security provisions of TEEs. Such an approach would eliminate the storage consumption and maintenance costs of the ever growing UTXO set and might also reduce the transaction verification time.

Bibliography

[1] 51% Attack in Bitcoin.
https://learncryptography.com/cryptocurrency/51-attack. Accessed 27 May 2017.

[2] Anati, I., Gueron, S., Johnson, S., and Scarlata, V. Innovative Technology for CPU-based Attestation and Sealing. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy* (2013), vol. 13.

[3] Antonopoulos, A. M. Mastering Bitcoin: Unlocking Digital Cryptocurrencies. First ed. O'Reilly Media, 2014, ch. 04: Keys, Addresses and Wallets, pp. 70–74.

[4] Antonopoulos, A. M. Mastering Bitcoin: Unlocking Digital Cryptocurrencies. First ed. O'Reilly Media, 2014, ch. 05: Transactions.

[5] Antonopoulos, A. M. Mastering Bitcoin: Unlocking Digital Cryptocurrencies. First ed. O'Reilly Media, 2014, ch. 04: Keys, Addresses and Wallets, pp. 84–86.

[6] Bamert, T., Decker, C., Elsen, L., Wattenhofer, R., and Welten, S. Have a Snack, Pay with bitcoins. In *Peer-to-Peer Computing (P2P), IEEE Thirteenth International Conference* (2013), IEEE, pp. 1–5.

[7] Bitcoin Market Price (USD), 2017.
https://blockchain.info/charts/market-price?timespan=all. Accessed 05 May 2017.

[8] Bitpay. Copay - A Bitcoin Wallet from BitPay.
https://github.com/bitpay/copay/. Accessed 16 May 2017.

[9] Block Hashing Algorithm.
https://en.bitcoin.it/wiki/Block_hashing_algorithm. Accessed 05 May 2017.

[10] Bonneau, J., Miller, A., Clark, J., Narayanan, A., Kroll, J. A., and Felten, E. W. SoK: Research Perspectives and Challenges for Bitcoin and Cryptocurrencies. In *Security and Privacy (SP), IEEE Symposium* (2015), IEEE, pp. 104–121.

[11] Brand, S. L. Department of Defense Standard - Trusted Computer System Evaluation Criteria (Orange Book). *Standard 5200.28,National Computer Security Center* (1985), 1–94.

[12] Brasser, F., Müller, U., Dmitrienko, A., Kostiainen, K., Capkun, S., and Sadeghi, A.-R. Software Grand Exposure: SGX Cache Attacks are Practical. *arXiv preprint arXiv:1702.07521* (2017).

[13] Brickell, E., and Li, J. Enhanced Privacy ID from Bilinear Pairing for Hardware Authentication and Attestation. *International Journal of Information Privacy, Security and Integrity 2 1*, 1 (2011), 3–33.

[14] Brodie, L. *Starting FORTH: An introduction to the FORTH language and Operating Systems*, first ed. Prentice-Hall, 1987.

[15] What is an Orphan Block?, Mar 2015. https://blog.cex.io/bitcoin-dictionary/what-is-an-orphan-block-9632. Accessed 05 May 2017.

[16] Chaum, D. Blind Signatures for Untraceable Payments. In *Advances in Cryptology* (1983), Springer, pp. 199–203.

[17] Cryptocurrency Market Capitalization, 2017.
https://coinmarketcap.com/all/views/all/. Accessed 05 May 2017.

[18] Costan, V., and Devadas, S. Intel SGX Explained. *IACR Cryptology ePrint Archive 2016* (2016), 86.

[19] Bitcoin Controlled Supply.
https://en.bitcoin.it/wiki/Controlled_supply. Accessed 27 June 2017.

[20] Dai, W. B-money: Anonymous Distributed Electronic Cash System. http://www.weidai.com/bmoney.txt. Accessed 05 May 2017.

[21] An Overview of Blockchain. https://bitcoin.org/en/developer-guide#block-chain-overview. Accessed 05 May 2017.

[22] Transactions in Bitcoin. https://bitcoin.org/en/developer-guide#transactions. Accessed 05 May 2017.

[23] Difficulty and Proof-of-Work in Blockchain. https://bitcoin.org/en/developer-guide#proof-of-work. Accessed 05 May 2017.

[24] Dmitrienko, A., Noack, D., and Yung, M. Secure Wallet-Assisted Offline Bitcoin Payments with Double-Spender Revocation. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security* (2017), ACM, pp. 520–531.

[25] Hardened keys in Bitcoin. https://bitcoin.org/en/developer-guide#hardened-keys. Accessed 05 May 2017.

[26] Hashcash: A Proof-of-Work Algorithm. http://www.hashcash.org/. Accessed 05 May 2017.

[27] ECC Key Creation: Hierarchical Deterministic Protocol. https://bitcoin.org/en/developer-guide#hierarchical-deterministic-key-creation. Accessed 05 May 2017.

[28] Bitcoin in India: Govt. forms committee to study Virtual Currency Framework, Apr 2017. https://www.cryptocoinsnews.com/bitcoin-india-govt-forms-committee-study-virtual-currency-framework/. Accessed 05 May 2017.

[29] Johnson, D., Menezes, A., and Vanstone, S. The Elliptic Curve Digital Signature Algorithm (ECDSA). *International journal of information security 1*, 1 (2001), 36–63.

[30] Johnson, S. Intel SGX and Side-Channels, May 2017. https://software.intel.com/en-us/articles/intel-sgx-and-side-channels. Accessed 27 May 2017.

[31] Kaliski, B. PKCS# 5: Password-based Cryptography Specification Version 2.0. RFC 2898: https://tools.ietf.org/html/rfc2898. Accessed 05 May 2017.

[32] Karame, G. O., Androulaki, E., and Capkun, S. Double-spending Fast Payments in Bitcoin. In *Proceedings of the 2012 ACM conference on Computer and Communications Security* (2012), ACM, pp. 906–917.

[33] Bitcoin Signing Key Reusage and Consequences. `https://bitcoin.org/en/developer-guide#avoiding-key-reuse`. Accessed 05 May 2017.

[34] Lind, J., Eyal, I., Pietzuch, P., and Sirer, E. G. Teechan: Payment Channels Using Trusted Execution Environments. *arXiv preprint arXiv:1612.07766* (2016).

[35] Lopez, J., and Dahab, R. An Overview of Elliptic Curve Cryptography (ECC).

[36] Merkle, R. C. Method of providing Digital Signatures, Jan 5 1982. US Patent 4,309,569. `https://patents.google.com/patent/US4309569A/en`.

[37] Nakamoto, S. Bitcoin: A Peer-to-Peer Electronic Cash System. `https://bitcoin.org/bitcoin.pdf`. Accessed 05 May 2017.

[38] Palatinus, M., and Rusnak, P. BIP Standard 44: Multi-Account Hierarchy for Deterministic Wallets, Apr 2014. `https://github.com/bitcoin/bips/blob/master/bip-0044.mediawiki`. Accessed 05 May 2017.

[39] Palatinus, M., Rusnak, P., Voisine, A., and Bowe, S. BIP Standard 39: Mnemonic codes for Generating Deterministic Keys, Sep 2013. `https://github.com/bitcoin/bips/blob/master/bip-0039.mediawiki`. Accessed 05 May 2017.

[40] Peck, M. Bitcoin: The Cryptoanarchists Answer to Cash, May 2012. `http://spectrum.ieee.org/computing/software/bitcoin-the-cryptoanarchists-answer-to-cash/0`. Accessed 05 May 2017.

[41] Podolanko, J. P., Ming, J., and Wright, M. Countering Double-Spend Attacks on Bitcoin Fast-Pay Transactions. *IEEE Security* (2017).

[42] Risk Steering Committee (RSC). DHS Risk Lexicon, September 2010.

https://www.dhs.gov/xlibrary/assets/dhs-risk-lexicon-2010.pdf. Accessed 16 May 2017.

[43] Rosenfeld, M. Analysis of Hashrate-based Double-spending. *arXiv preprint arXiv:1402.2009* (2014).

[44] Bitcoin Transaction Scripts. https://en.bitcoin.it/wiki/Script. Accessed 05 May 2017.

[45] Bitcoin Signature Hash Types. https://bitcoin.org/en/developer-guide#signature-hash-types. Accessed 05 May 2017.

[46] Szabo, N. Bit gold: Cryptocurrency, Dec 2008. http://unenumerated.blogspot.fi/2005/12/bit-gold.html. Accessed 05 May 2017.

[47] Tapscott, A. How Blockchain is Changing Finance, Mar 2017. https://hbr.org/2017/03/how-blockchain-is-changing-finance. Accessed 05 May 2017.

[48] Team, T. B. The New Intel◯ and BitPay Integration, Nov 2016. https://blog.bitpay.com/intel-and-bitpay/. Accessed 27 May 2017.

[49] Testnet Wiki. https://en.bitcoin.it/wiki/Testnet. Accessed 16 May 2017.

[50] TrustZone: ARM Developer. https://developer.arm.com/technologies/trustzone. Accessed 10 June 2017.

[51] Tschorsch, F., and Scheuermann, B. Bitcoin and Beyond: A Technical Survey on Decentralized Digital Currencies. *IEEE Communications Surveys & Tutorials 18*, 3 (2015), 2084–2123.

[52] Digital Currencies: Call for Information, Mar 2015. https://www.gov.uk/government/consultations/digital-currencies-call-for-information/digital-currencies-call-for-information. Accessed 05 May 2017.

[53] Unspent Transaction Outputs Chart. https://blockchain.info/charts/utxo-count. Accessed 19 May 2017.

[54] Vasudevan, A., Owusu, E., Zhou, Z., Newsome, J., and McCune, J. M. Trustworthy Execution on Mobile Devices: What Security Properties Can My Mobile Platform Give Me? In *International Conference on Trust and Trustworthy Computing* (2012), Springer, pp. 159–178.

[55] Wuille, P. Bitcoin Improvement Protocol Standard - bip0032, Feb 2012. `https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki`. Accessed 05 May 2017.

[56] Xu, Y., Cui, W., and Peinado, M. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *Security and Privacy (SP), 2015 IEEE Symposium on* (2015), IEEE, pp. 640–656.