MSE | MASTER OF SCIENCE IN ENGINEERING

Hes·so
Haute Ecole Spécialisée de Suisse occidentale
Fachhochschule Westschweiz
University of Applied Sciences and Arts Western Switzerland

Master of Science HES-SO in Engineering
Av. de Provence 6
CH-1007 Lausanne

Master of Science HES-SO in Engineering

Orientation: Information and Communication Technologies (ICT)

NEW METHODS FOR TRANSACTIONS IN BLOCKCHAIN SYSTEMS

Author:
Joël Gugger

Under the direction of:
Prof. Alexandre Karlov
HEIG-VD

External expert:
Pierre-Mathieu Alamy
Bity

Lausanne, HES-SO//Master, February 8, 2018

Information about this report

Contact information

Author: Joël Gugger

MSE Student

HES-SO//Master

Switzerland

Email: *joel.gugger@master.hes-so.ch*

Declaration of honor

I, undersigned, Joël Gugger, hereby declare that the work submitted is the result of a personal work. I certify that I have not resorted to plagiarism or other forms of fraud. All sources of information used and the author quotes were clearly mentioned.

Place, date: _____

Signature: _____

Validation

Accepted by the HES-SO//Master (Switzerland, Lausanne) on a proposal from:

Prof. Alexandre Karlov, Thesis project advisor
Pierre-Mathieu Alamy, Bity, Main expert

Place, date: _____

Acknowledgments

I especially want to acknowledge Thomas Shababi for his careful proofreading of this thesis and his helpful contribution in the discussions related to this work. Thanks to Daniel Lebrecht for his valuable contribution in collecting data for transaction optimization and his effective collaboration. Acknowledgement to Prof. Alexandre Karlov and Pierre-Mathieu Alamy for their precious feedback on this work. And thank you to my wife for supporting me during the writing of this thesis.

Abstract

Bitcoin is a decentralized peer-to-peer currency that allow users to to pay for things electronically. Bitcoin was created by a pseudonymous software developer going by the name of Satoshi Nakamoto in 2008, as an electronic payment system based on mathematical proof. Yet the largest challenge in Bitcoin for the coming years is scalability. Currently, Bitcoin can only handle a few transactions per second on the network. This is not sufficient in comparison to large payment infrastructures, which allow tens of thousands of transactions per second. As a potential scalability solution, the idea of payment channels was suggested by Satoshi in an email to Mike Hearn. A one-way payment channel specific for retail commercial transactions is presented, analyzed and optimized with threshold cryptography. The threshold scheme selected has been adapted and implemented into the Bitcon cryptographic library to compute a special two-party threshold ECDSA signature.

Keywords: Crypto-currencies, Bitcoin, Payment channels, Cryptography, Threshold ECDSA signatures, Curve secp256k1, Elliptic Curve Cryptography

Contents

Acknowledgements **v**

Abstract **vii**

1 Introduction **1**

2 Bitcoin, a peer-to-peer payment network **3**
2.1 The blockchain 4
2.2 Transactions 5
2.3 Scalability of Bitcoin 10

3 Payment channels, a micropayment network **11**
3.1 Types of payment channel 12
3.2 One-way channel (Shababi-Gugger-Lebrecht) 14
3.3 Optimizing payment channels 14

4 ECDSA asymmetric threshold scheme **17**
4.1 Reminder 18
4.2 Threshold scheme 21
4.3 Threshold Hierarchical Determinitic Wallets 29
4.4 Threshold deterministic signatures 34

5 Implementation in Bitcoin-core secp256k1 **35**
5.1 Configuration 37
5.2 `DER` parser-serializer 39
5.3 Paillier cryptosystem 41
5.4 Zero-knowledge proofs 45
5.5 Threshold module 49

6 Further research **55**
6.1 Side-channel attack resistant implementation and improvements . . . 55
6.2 Hardware wallets 56
6.3 General threshold scheme 56
6.4 Schnorr signatures 57

7 Conclusions **59**

A Experimental implementation in Python **61**

List of Figures **87**

Contents

List of Tables **89**

List of Sources **91**

Bibliography **93**

Glossary **97**

1 Introduction

Bitcoin is a decentralized peer-to-peer currency that allows users to pay for things electronically. Thousands of other cryptocurrencies exist, but only some of them are really interesting from a political, economical or technical point of view. Bitcoin was created by a pseudonymous software developer going by the name of Satoshi Nakamoto in 2008, as an electronic payment system based on mathematical proof. The idea was to produce a means of exchange, independent of any central authority and censorship-resistant, which could be transferred electronically in a secure, verifiable and immutable way. The blockchain is the output of this secure, verifiable and immutable mathematical proof.

The most significant challenge in Bitcoin for the coming years is scalability. Currently, Bitcoin enforces a block-size limit which is equivalent to only a few transactions per second on the network. This amount is not sufficient in comparison to large payment infrastructures, which allow tens of thousands of transactions per second and even more at peak times such as Christmas. To address this there are some proposals to modify the transaction structure (SegWit [1]), some to modify the block-size limit (SegWit2x) and others to create a second layer on top of the Bitcoin protocol (Lightning Network [2]). In the same idea of a second layer, this thesis explores implementation of a unidirectional payment channel for retail commercial transactions that allows two parties to transact over the blockchain while minimizing the number of transactions needed on the blockchain in a secure and trustless way. Every kind of channel needs multi-signature addresses to secure the funds. A cryptographic threshold scheme might improve these schemes significantly. Finding such a threshold scheme that fulfills the requirements is not trivial. The threshold scheme selected for this work is adapted to the needs of channels. And is implemented into the Bitcoin cryptographic library to compute a particular two-party threshold ECDSA signature.

2 Bitcoin, a peer-to-peer payment network

The Bitcoin ecosystem is composed of multiple actors. Users of the network access information via software on their laptop or mobile phone. These users can see the amounts present in their addresses. An address is the digest of a public key, itself being the representation of a private key. An address is owned by a user if this user has the associated private key in his possession. Users can transfer funds from some of their addresses to other addresses owned by other users or themselves. When funds are transferred a transaction is created and broadcast to the network. The network is composed of nodes, and these nodes take care of its proper functioning. Some of these nodes are called miners, they listen to new transactions and try to include them into the blockchain. This blockchain is the output, the necessary result, of the Bitcoin protocol and can be compared to a distributed public ledger. Nodes are software running all over the world. This software is maintained and improved by a group of developers present all over the world and for Bitcoin, the original and reference implementation is Bitcoin-core (previously referred to as bitcoind). Bitcoin-core allows interacting with the blockchain, and it is possible to retrieve information such as current *unconfirmed* transactions, information present in the blockchain, the amount available for an address, and more. *Unconfirmed* transactions are transactions that have not been yet included in the blockchain but have already been broadcast to the network.

In the following, some building blocks needed to figure out how payment channels work and how we can improve them, with some cryptography, are explored. If you are a master of Bitcoin and you already know how blocks are created, how transactions are structured, how fees are calculated and how segregated witness works, this chapter will just be a reminder. For further explanation, the best resource today is the book "Mastering Bitcoin" by Andreas Antonopoulos [3].

Contents

2.1 The blockchain 4
2.1.1 A chain of blocks 4
2.1.2 A list of transactions 4
2.2 Transactions 5
2.2.1 A list of inputs & outputs 5
2.2.2 Transaction fees 6
2.2.3 Scripting language 7
2.2.4 Segregated witness 9
2.2.5 Transaction malleability 9
2.3 Scalability of Bitcoin 10
2.3.1 Layer-two applications 10

2.1 The blockchain

The blockchain, as indicated by the name, is a chain of blocks. Blocks are created by miners in a race to find the next valid block. A block is valid if its identifier, i.e., the double hash of its header, is lower than the current difficulty target. The validity of a block is based on several other criteria which are not mentioned here. For further information, please refer to the book "Mastering Bitcoin". The header of a block is composed of a version number, a creation timestamp, a *nonce*, an other required information.

The difficulty target is adjusted every 2016 blocks, so that on average, a valid block is found in the network every ten minutes. The probability of finding a block can be modelled as a Poisson process, i.e., the probability of a given number of events occurring in a fixed interval of time or space, if these events occur with a known constant rate, is independent of the time since the last event. A miner will create a candidate block and compute its identifier if this identifier is lower than the current difficulty target then the block is valid and the miner notifies the network that he found the next block. Then the process starts again. If the block identifier is not valid, the miner can change the *nonce* value in the header and check with the new identifier. Enumerating these identifiers to find the next valid block requires an enormous amount of power. All of the miners, round the clock, keep searching for the next valid block by brut forcing these identifiers with the *nonce*.

2.1.1 A chain of blocks

As mentioned before, the blockchain is a chain which must be secure, verifiable, and immutable. To achieve immutability, modification of previous blocks must invalidate the chain. The block identifier is affected by information like the creation timestamp or the *nonce* used to adapt the modifier but also from the previous block identifier in the chain. That means that if the previous block identifier is changed for example because its content changed, the child block will become invalid as well as its child, and so on.

Modifying the blockchain without invalidating the chain requires recomputing all the block identifiers after the changed block with the same difficulty target. It requires a quantity of power that can be estimated and for which the costs represent a certain safety threshold. It is established that a transaction included in a block can be considered as safe after six child blocks. The amount of power needed to erase this transaction became statistically too high to be probable, but it does not mean that it will never happen since the mining process is modelled as a Poisson process.

2.1.2 A list of transactions

To be useful a block needs content. In Bitcoin, transactions compose the content of a block. As mentioned before, a transaction is called *confirmed* when it is included in a block. The number of confirmations, also called *depth*, is related to the number of blocks mined after the inclusion of the transaction.

A Merkle tree is created to keep track of all the transactions included in a block. This Merkle tree, or hash tree, is a is a structure in which every leaf node is labeled with the hash of a data block, and every non-leaf node is labeled with the cryptographic hash of the labels of its child nodes.

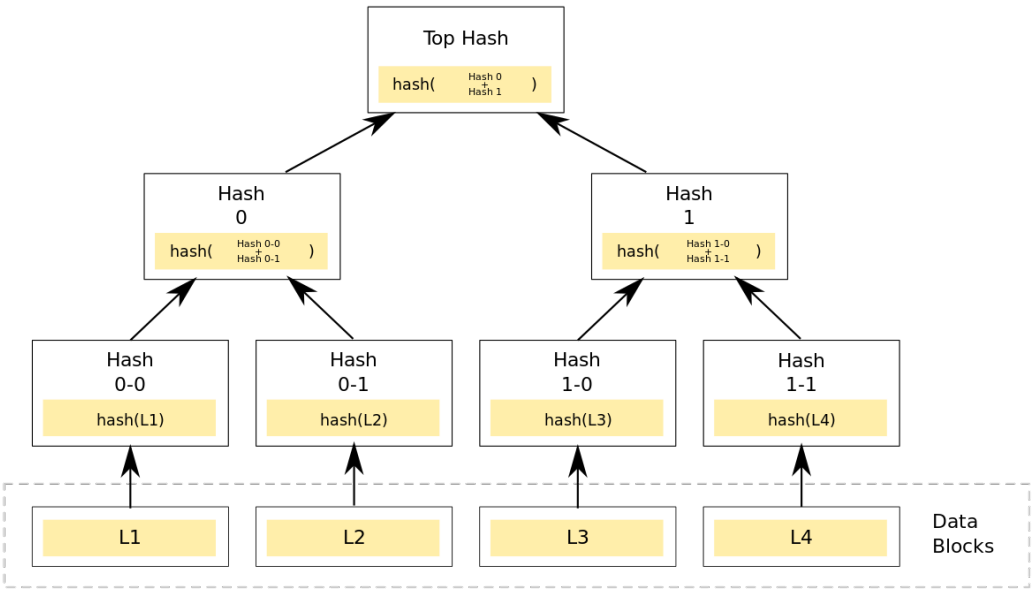

Figure 2.1 *Merkle tree construction*
Source: *https://en.wikipedia.org/wiki/Merkle_tree*

Given the top hash, known as the Merkle root, and a leaf, it is possible to prove the membership by giving the path for each complementary hash. E.g given the Merkle root and `L1`, the proof is `Hash 0-1` and `Hash 1`. The verifier can then compute the hash of `L1`, the result of this hash with `Hash 0-1`, and then with `Hash 1`. If the result is the same as the Merkle root, then `L1` is a part of the tree.

In a block, the miner creates a Merkle tree of all included transaction identifiers and puts the Merkle root into the header of the block. To validate if a transaction is included in a block the path must be provided. Then the resulting hash is compared to the Merkle root registered in the block's header. Simplified Payment Verification (SPV) nodes, nodes without the full blockchain, download only block headers and ask other nodes to provide partial views of relevant parts of the blockchain.

2.2 Transactions

Transactions allow users to move bitcoins from one address to another and are the content of Bitcoin's blockchain. In Bitcoin, the blockchain does not store a balance for each user; the blockchain keeps only the history of all transactions made since the beginning.

2.2.1 A list of inputs & outputs

A transaction is composed of a list of inputs and a list of outputs. In other words, where the bitcoins come from and where they go. An input refers to an unspent output at the address from where funds will be spent, and output refers to an address where the funds will go. To spend funds the user needs to control the addresses where unspent outputs are present. These unspent outputs are called `UTXOs` and, combined, represent the total amount owned by a user.

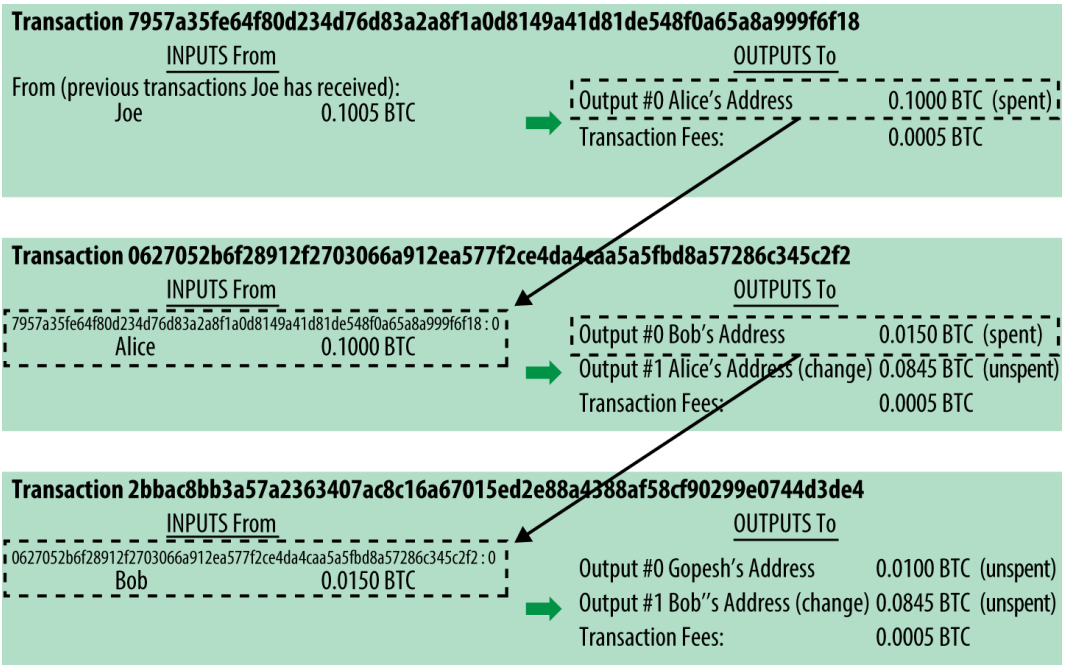

Figure 2.2 *A chain of transactions where inputs and outputs are linked*
Source: *https://github.com/bitcoinbook/bitcoinbook/blob/second_edition/ch02.asciidoc*

Each input has a value, the value specified in the output to which the input points. The sum of the value of all inputs in a transaction must be less than the sum of the values specified in the outputs, with the difference being the fee, meaning an input must be spent entirely.

The most straightforward transaction is composed of one input and one output with the same amount of money in and out if we include the fee. However, the most typical transaction is composed of one input, referring to where the funds come from, and two outputs as it is rare to have the right amount available in one `UTXO`. In this case, the first output is the user who will receive the funds, with the amount transferred, and the second output is another address owned by the sender to gather the change, i.e., the remaining amount.

As with blocks, transactions also have an identifier. These identifiers are created in the same way as blocks, by taking the double hash of the data, i.e., the whole transaction. This means that, in the original design, a transaction does not have its final transaction identifier (TXID) before it is wholly signed, i.e., every input.

2.2.2 Transaction fees

The sum of all outputs of a transaction is constrained by the sum of the inputs, but the difference is implicitly considered as a fee (as shown in Figure 2.2.) Fees were not required in the beginning, but today a transaction will not be relayed, nor included in a block without paying fees. A miner, when he finds a block, can create the first transaction without inputs, where a fixed amount of new coins is created plus the total amount of fees collected in all the included transactions. A miner will, therefore, select the transactions that pay the most fees given the space they consume

in the block. Fees are calculated with the weight of the transaction, the weight is equal to the transaction size in bytes. A ratio of fee per byte is then selected to find the fee for a transaction.

2.2.3 Scripting language

As described before, outputs or `UTXOs` are related to addresses and proof of ownership is required to spend them. To spend a `UTXO`, the Bitcoin protocol uses digital signatures, a valid signature for the address from which the it is being spent is required and, to sign, the private key is required. Thus, while signing a transaction corresponding to the right address, it is possible to prove that the user owns the address. However, the protocol does not just require signatures and public keys, conditions to *unlock* a `UTXO` are structured in scripts. Bitcoin has a stack-based script language called "Bitcoin Script".

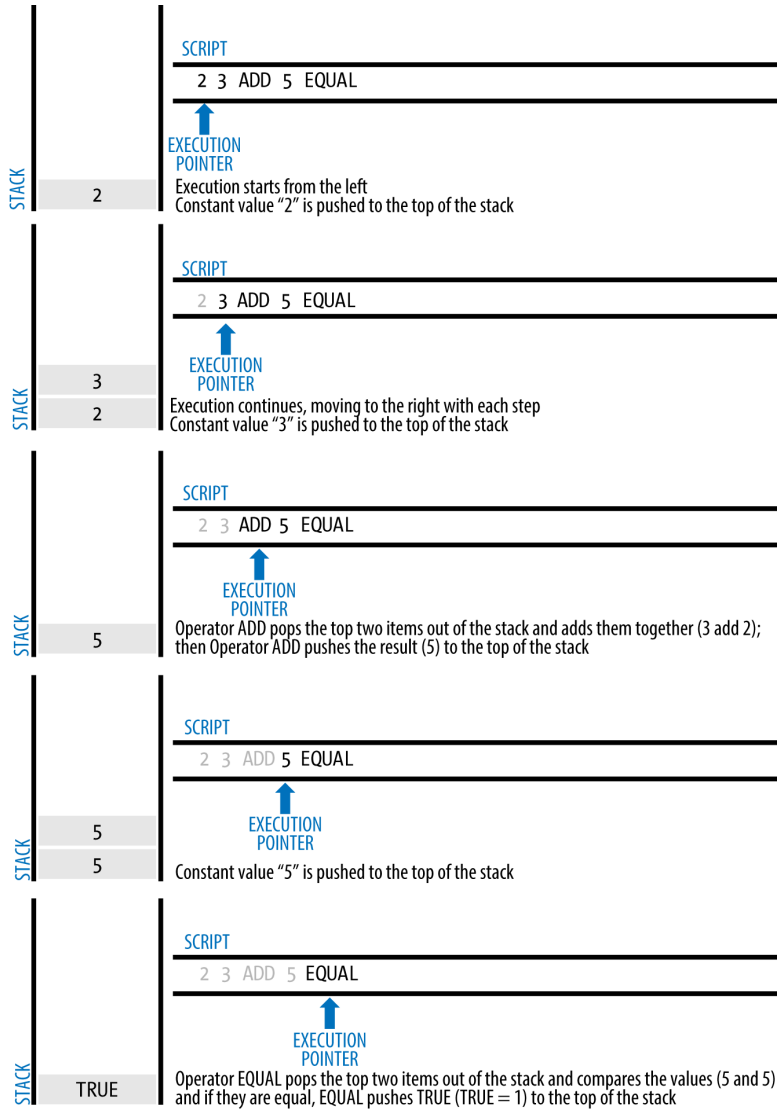

Figure 2.3 Example of simple Bitcoin script program execution
Source: *https://github.com/bitcoinbook/bitcoinbook/blob/second_edition/ch06.asciidoc*

The list of all the `OP_CODES` available in Bitcoin's scripting language is in the documentation. Among them are `OP_CHECKSIG`, which verifies a given signature with a public key provided on top of the stack, `OP_IF, OP_ELSE, OP_ENDIF` create execution branches with a boolean on top of the stack, `OP_DUP` duplicates the value on top of the stack or `OP_HASH160, OP_SHA256` which compute hashes of values on top of the stack.

Each input and output have a script. For outputs, the script establishes the requirement to be fulfilled in order to be able to spend it. An address is the result of the public key hashed with a `SHA256` and then hashed with a `RIPEMD160` encoded with a checksum in a more human-readable format. A user can decode the human-readable format of an address to retrieve the hash and create an output script called Pay To Public Key Hash (P2PKH). With the script, the address is retrieved, and given the address and the script, only the user who holds the private key of that address will be able to sign the transaction and spend the funds. The user controlling that address can create a transaction where that input points to the `UTXO` and, to unlock the funds, the user needs to sign the transaction and give the signature with the public key in the input's unlocking script. Before including a transaction in a candidate block, a miner validates all the inputs. He needs to check if the outputs are in fact `UTXOs`, so ensuring the outputs are *unspent*, and execute the unlocking script with the locking script. Both scripts are concatenated to validate input; the unlocking script first, as shown in Figure 2.4.

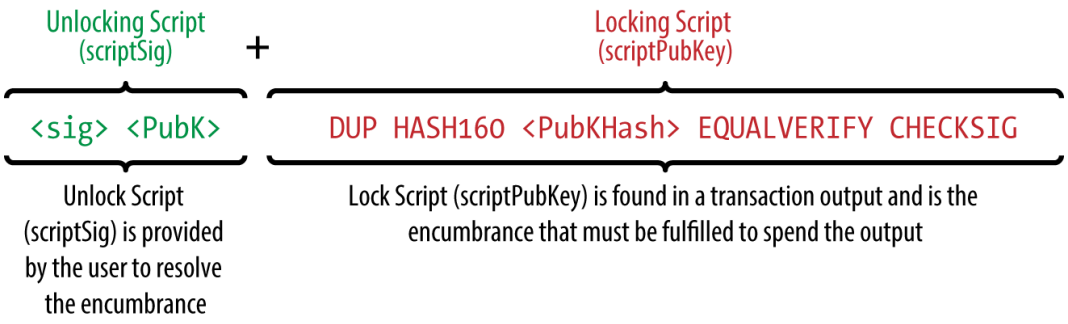

Figure 2.4 *Example of pay to public key hash locking script with unlocking script*
Source: *https://github.com/bitcoinbook/bitcoinbook/blob/second_edition/ch06.asciidoc*

The script in Figure 2.4, when executed, put the signature on top of the stack, then the public key. The public key is duplicated, hashed, and the public key hash present in the locking script is put on top of the stack, then the two first elements on top of the stack are then compared. If the comparison fails, the script fails and the transaction is rejected. If the test passes, the signature will be checked with the two remaining parameters on the stack (i) the public key and (ii) the signature. If the signature is valid, the value `True` is put on top of the stack. Otherwise, the value `False` put on top of the stack. If the value `True` is present on top of the stack at the end of the script the transaction is valid, otherwise the transaction is invalid.

Pay To Script Hash (P2SH) are the second standard script mostly used. It provides the ability to send outputs to a custom script without knowing the content of the script. A user can create a custom script and give the hash of this script, in an address, to another user. When the output is spent the whole script must be provided in the unlocking script. P2SH are used, e.g., to create multi-signature scripts.

2.2.4 Segregated witness

Segregated Witness (SegWit) is the Bitcoin Improvement Proposal (BIP) 141 that proposed changing the transaction structure to fix transaction malleability, add script versioning, and improve other aspects [4, 1]. In fact, SegWit changed the way outputs are structured. The malleability is fixed if all inputs use SegWit only. A transaction can have SegWit inputs and non-SegWit inputs at the same time. The BIP abstract explains its purpose:

> This BIP defines a new structure called a "witness" that is committed to blocks separately from the transaction merkle tree. This structure contains data required to check transaction validity but not required to determine transaction effects. In particular, scripts and signatures are moved into this new structure.
>
> The witness is committed in a tree that is nested into the block's existing merkle root via the coinbase transaction for the purpose of making this BIP soft fork compatible. A future hard fork can place this tree in its own branch.

With SegWit, a transaction has two TXIDs. The first is determined without all the witness data, so it is deterministic at transaction creation. The second one is related to the witness data and changes when signatures appear. This separation fixes the transaction malleability issue. The second big change is the way the weight is calculated to determine the fees. The transaction weight `Tx Weight` becomes $\texttt{Base Tx} * 3 + \texttt{Total Size}$ where `BaseTx` is the size without the witness data and `Total Size` is the serialized transaction with all the data, including the witness data. This new structure introduces a virtual transaction size such as virtual size is equal to $\texttt{Tx Weight}/4$. Thus, SegWit reduces the weight of the witness data in the calculus of the fees.

2.2.5 Transaction malleability

The fact that the transaction identifier depends on the hash of the whole serialized transaction while the signature does not currently cover all the data in a transaction introduces what is called transaction malleability. A miner could tweak the transaction to change is identifier before including it into a block without invalidating it nor changing the claiming output conditions. This malleability means that an unconfirmed chain of transactions must not be trusted (because the following transactions will depend on the hashes of the previous transactions.)

The first way to achieve malleability is to tweak the signatures themselves. For every signature (r, s), the signature $(r, -s \pmod n)$ is a valid signature for the same message. As mentioned in the Bitcoin wiki about transaction malleability:

> As of block 363724, the BIP66 soft fork has made it mandatory for all new transactions in the block chain to strictly follow the DER-encoded ASN.1 standard. Further efforts are still under way to close other possible malleability within DER signatures.

However, even if the format is standardized and enforced, signatures can still be changed by anyone who has access to the corresponding private keys. When the user signs a transaction, the unlocking script (scriptSig) contains, e.g., for a standard P2PKH, the signature and the public key. These data are present in the signed transaction, but cannot be present during the signing process because they do not yet exist. This presence of signatures in the script means that the content of unlocking scripts are not part of the signing data, but part of the hashing data for the TXID. E.g., by introducing an additional `OP_CODE`, it is possible to change the TXID without invalidating the signature.

Nevertheless, with SegWit activated, now the transaction malleability with signatures and scripts is no longer possible. As mentioned in the BIP 141:

> It allows creation of unconfirmed transaction dependency chains without counterparty risk, an important feature for offchain protocols such as the Lightning Network.

2.3 Scalability of Bitcoin

Improvements in the consensus layer have been made and will continue to appear to answer the problem of scalability. However, modifying the consensus layer is not easy, usually soft forks are needed, and in some cases hard forks, both of which can be a lengthy and painful process as majority adoption of the changes is required. With SegWit, the latest significant improvement on-chain, all the prerequisites to construct a robust layer-two application, such as fixing malleability, are fulfilled.

2.3.1 Layer-two applications

The layer-two is an architectural concept where a blockchain is used as a source of truth and only for resolving disagreements. Usually, the construction that uses this architectural concept is called payment channels or micropayment channels. These channels enable scalability because they reduce the number of transactions needed on the blockchain if two users exchange often. Channels allow more than scalability, with payment channel transactions are instant and can be accepted as *confirmed* without latency. Latency is a consequence of the mechanism that resolves the *double-spending* problem when funds are spent twice to different users. Payment channels create a structure that ensures that no double-spending is possible for the funds locked in the channel.

3 Payment channels, a micropayment network

Payment channels or micropayment channels, as mentioned previously, are one part of the scalability solution. There are various propositions to construct such structures. To have a better understanding of the differences, strengths, and weaknesses of some of them the author proposes a classification and definitions present in the white paper following appendices. Those definitions are repeated starting now.

Definition 3.0.1 (Trustless) *A channel is trustless if and only if the safety of funds for every player $p_i \in \mathcal{P} = \{\mathcal{P}_0, \ldots, \mathcal{P}_n\}$ at each step $\mathcal{S}$ of the protocol does not depend on player $\Delta p = \mathcal{P} - p_i$'s behavior.*

Definition 3.0.2 (Optimal) *A channel is optimal if and only if the number of transactions $\mathcal{T}(\mathcal{C})$ needed to claim the funds for a given constraint $\mathcal{C}$ is equal to the number of moves $\mathcal{M}(\mathcal{C})$ needed to satisfy the constraint at any time without breaking the first definition.*

For a constraint $\mathcal{C}$ in a channel $\mathcal{P}_1 \to \mathcal{P}_2$, refunding $\mathcal{P}_1$ requires $\mathcal{M}(\mathcal{C}) = 1$, thus an optimal scheme requires $\mathcal{T}(\mathcal{C}) = \mathcal{M}(\mathcal{C}) = 1$. Note: in a channel $\mathcal{P}_1 \to \mathcal{P}_2$ refund and settlement both require $\mathcal{M}(\mathcal{C}) = 1$.

Definition 3.0.3 (Endless) *A channel is endless if and only if there is no predetermined lifetime at the setup.*

Definition 3.0.4 (Pulseless) *A channel is pulseless if and only if there is no need to refresh or close the channel on-chain while at least one player $p_i \in \mathcal{P} = \{\mathcal{P}_0, \ldots, \mathcal{P}_n\}$ where the available amount to send is $A(p_i) > 0$. By definition a pulseless channel must be also endless.*

Definition 3.0.5 (Undelayed) *A channel is undelayed if and only if each player $p_i \in \mathcal{P} = \{\mathcal{P}_0, \ldots, \mathcal{P}_n\}$ can trigger the process to get their money back at any time.*

These definitions are used in the following to compare different commonly exposed payment channel constructions. The list does not contain all the payment channel propositions and some of them might be missing. However, the list contains a fairly good representation of the different constructions existing.

Contents

3.1 Types of payment channel 12
3.1.1 Unidirectional 12
3.1.2 Bidirectional 13
3.1.3 Summary 13
3.2 One-way channel (Shababi-Gugger-Lebrecht) 14
3.3 Optimizing payment channels 14

3.1 Types of payment channel

We can distinguish two type of channels, the unidirectional channel that allows one user to send money to another user in a channel and the bidirectional channel that allows two users to send in either direction in a channel. Usually, a bidirectional channel is more optimal than two unidirectional channels but introduced other constraints.

3.1.1 Unidirectional

In unidirectional channels, there is a payer, later referred to as player-one or client, and a payee, later referred to as player-two or provider. It is not possible to transfer money back in the reverse direction in the channel.

Spilman-style payment channels

Spilman-style payment channels, proposed by Jeremy Spilman in 2013 [5], are the most simple construction of a unidirectional payment channel. They have a finite lifetime predefined at the setup phase and the client, i.e., the payer, cannot trigger their refund before the end of the channel lifetime (but he can receive his funds back if the payee settles the channel before the end of the lifetime.) The channel is one-time use. When the payer or the payee get their funds, the channel is closing. Neither the payer nor the payee need to watch the blockchain to react to events during the lifetime of the channel because only the payee can broadcast a transaction, so both do not need to watch the blockchain to be safe. It is worth noting that, without a proper fix to transaction malleability [1, 6, 7, 8], this scheme is not secure.

According to the previous definitions, Spilman-style payment channels are trustless (assuming that a suitable solution for transaction malleability has been implemented), and optimal but not endless nor undelayed.

`CLTV`-style payment channels

Introduced in 2015, `CLTV`-style payment channels are a solution to the malleability problem in Spilman-style payment channels. With the new `OP_CODE` check locktime verify (`OP_CHECKLOCKTIMEVERIFY`), redefining the `OP_NOP2`, it is possible to enforce the non-spending of a transaction output until some time in the future. With `OP_CHECKLOCKTIMEVERIFY` a transaction output can enforce the spending transaction to have a `nLockTime` later or equal to the specified value in the script [9].

Instead of creating a funding transaction and a refund transaction vulnerable to transaction malleability attacks, the client creates the funding transaction output with a script (Listing 3.1) that allows the provider and the client to spend the funds with co-operation or after a lock time the client can spend the funds without the co-operation of the provider.

```
IF
    <provider pubkey> CHECKSIGVERIFY
ELSE
    <expiry time> CHECKLOCKTIMEVERIFY DROP
ENDIF
<client pubkey> CHECKSIG
```

Listing 3.1 Locking script (scriptPubKey) with `CHECKLOCKTIMEVERIFY`

`CLTV`-style payment channels have the same properties as Spilman-style payment channels following the previous definition but are not subject to transaction malleability attacks.

3.1.2 Bidirectional

In a bidirectional channel $\mathcal{C}$, the player A and the player B can send funds in direction $\mathcal{C}_{AB}$ and $\mathcal{C}_{BA}$. A bidirectional channel can be a specific scheme or a pairing of existing unidirectional channels.

Decker-Wattenhofer duplex payment channels

Decker-Wattenhofer duplex payment channels [10], also called Duplex Micropayment Channels (DMC), proposed in 2015, are bidirectional channels based on pairs of Spilman-style unidirectional channels. The construction has a finite lifetime predefined at the setup phase but can be refreshed on-chain to keep the channel open with an updated state. During the refresh process, it is possible to refill the channel, and the scheme allows payment routing with Hashed Timelock Contracts (HTLC).

Duplex payment channels are trustless and endless, but not optimal. The uncooperative close of the channel requires $d + 2$ transactions (where d is equal to the revocation tree depth). They are not pulseless (it requires a refresh transaction when the lifetime runs over to keep the channel open) and not undelayed (without other players cooperation the funds are recovered after `nLockTime` values.)

Poon-Dryja payment channels

Poon-Dryja payment channels, also called Lightning Network, is a proposed implementation of HTLC with bidirectional payment channels which allow payments to be securely routed across multiple peer-to-peer payment channels [2].

Their scheme is trustless (assuming that SegWit has been implemented), endless, pulseless, and undelayed but not optimal when the channel closes without co-operation.

3.1.3 Summary

Channel	Type	Optimal	Endless	Undelayed
Spilman-style	Uni	Yes	No	No
CLTV-style	Uni	Yes	No	No
Decker-Wattenhofer DMC	Bi	No	Yes	No
Poon-Dryja	Bi	No	Yes	Yes
Shababi-Gugger-Lebrecht	Uni	No	Yes	Yes

Table 3.1 Summary of different payment channels

This table summarizes the different properties of the proposed definitions of common channel schemes. The last row refers to the next presented scheme.

3.2 One-way channel (Shababi-Gugger-Lebrecht)

Our one-way payment channel for Bitcoin is a modified version of other layer-two applications, such as "Yours Lightning Protocol" or Lightning Network [2, 11]. The scheme is specially designed for a client to provider scenario, where the provider has multiple clients through multiple channels. The core design aims to be as cheap as possible for the provider while being flexible for settlement. The white paper describing the core design more in-depth is in the following appendices.

Our payment channel requires SegWit, and is trustless, endless, pulseless, and undelayed but not optimal when the channel closes without co-operation.

A part of this thesis was devoted to writing the white paper describing our channel scheme while working on the scheme itself. During this work we found a possible attack described in the white paper which we fixed.

The next step has been to analyze how it is possible to optimize the channel with threshold cryptography. As it is possible to see, every channel construction depends on a funding transaction that locks funds in a 2-out-of-2 multi-signature script. This funding transaction is always on-chain, so if it is possible to replace this P2SH with a standard P2PKH output the savings should be attractive.

3.3 Optimizing payment channels

Three transactions are compared with SegWit[1] and without. Optimization is expressed in percentage of size or virtual-size economized. The Script Hash (SH) consumes a multi-signature script, and the Public Key Hash (PKH) consumes a standard public key. Noted that size can vary few bytes with SegWit.

		Non-SegWit		SegWit		
		R-Size	O	R-Size	V-Size	O
First Refund	SH	302	36.75%	340	174	22.99%
	PKH	191		216	134	
Refund Normal	SH	335	32.54%	372	207	20.29%
	PKH	226		246	165	
Settlement	SH	335	32.54%	372	207	20.29%
	PKH	226		246	165	

Table 3.2 Summary of transaction size optimization

The average fee per byte in the last three months is around 292 Satoshis. This optimization allows saving until 32,412 Satoshis for the first refund transaction without SegWit, and 12,264 Satoshis for a refund or a settlement transaction with SegWit. At the current price, these amounts represent between USD $1.31 and USD $3.47[2]. If the channel is used for micropayments such as a couple of cents each time, this optimization makes the difference and lower the required threshold for rentability. The first refund transaction being less expensive, this also makes the client commitment easier.

[1] The transaction size is calculated with nested-SegWit and not with native mode.

[2] Average price of Bitcoin in the last 3 months, around $10,700 USD

Requirements need to be defined to be able to replace the multi-signature script with a threshold scheme. Analysis of the protocol and the signing process for a multi-signature script allows one to define these requirements. A multi-signature script can be unlocked with two different public keys and theire signature. The signers order does not matter, signatures are sorted by numerical value in the script. The protocol takes advantage of this fact. A transaction is usually held fully signed only by one player. The threshold scheme must follow these requirements (i) 2 players need to co-operate to generate a valid signature, (ii) both must be able to start the signing process, and (iii) only one player must be able to retrieve the signature at the end of the process. If both need the signature it is always feasible to share, meaning the current protocol is not better in this case.

4 ECDSA asymmetric threshold scheme

Threshold cryptography has been discussed for a long time already, many cryptographic schemes like RSA or Paillier [12, 13] exist, but others are less suitable to port. Since Bitcoin become famous, people have lost funds because they lose keys or get hacked. Since then research has been done to secure Bitcoin wallets [14, 15], however, the most significant problem today in Bitcoin that slows down the adoption of a threshold cryptosystem is the complexity of creating an efficient and flexible scheme for Elliptic Curve Digital Signature Algorithm (ECDSA). Recently, researchers have focused on finding more efficient and more generic systems, but fortunately, a protocol perfectly fulfilling the needs described in the previous chapter required to improve payment channels in Bitcoin already exists. Nevertheless, this scheme explains how to perform a threshold Digital Signature Algorithm (DSA) and not a threshold ECDSA. So the protocol needs to be adapted.

The scheme analyzed, transformed, and implemented in the following has been proposed by MacKenzie and Reiter in their paper "Two-Party Generation of DSA Signatures" [16]. This scheme is also the basis of several other papers previously cited. They base their construction of a threshold signature scheme a simple multiplicative shared secret and homomorphic encryption to keep the individual values unknown by the other signer. The homomorphic encryption used as an example in the paper and chosen for the implementation is the Paillier cryptosystem [17]. The following chapter describes how to adapt the scheme from DSA to ECDSA and introduces some fundamental building blocks needed for a real case scenario like hierarchical deterministic threshold wallet or deterministic signatures.

Contents

4.1 Reminder . . . 18
- 4.1.1 Elliptic curves . . . 18
- 4.1.2 Paillier cryptosystem . . . 19
- 4.1.3 Signature schemes . . . 19

4.2 Threshold scheme . . . 21
- 4.2.1 Adapting the scheme . . . 22
- 4.2.2 Adapting zero-knowledge proofs . . . 23

4.3 Threshold Hierarchical Determinitic Wallets . . . 29
- 4.3.1 Private parent key to private child key . . . 29
- 4.3.2 Public parent key to public child key . . . 30
- 4.3.3 Child key share derivation . . . 30
- 4.3.4 Proof-of-concept implementation . . . 31

4.4 Threshold deterministic signatures . . . 34

4.1 Reminder

Before introducing the threshold scheme, reminders of basic components used in the scheme are presented. These reminders are composed of Elliptic Curves (EC) general mathematics, Paillier homomorphic encryption scheme, and digital signature protocols, in particular DSA and ECDSA and their differences.

4.1.1 Elliptic curves

Bitcoin uses EC cryptography for securing its transactions. ECDSA, based on the DSA proposal by the National Institute of Standards and Technology (NIST), over the curve secp256k1, proposed by the Standards for Efficient Cryptography Group (SECG), is used [18].

Secp256k1 curve

The curve secp256k1 is defined over the finite field $\mathbb{F}_p$ of 2^{256} bits with a Koblitz curve $y^2 = x^3 + ax + b$ where $a = 0$ and $b = 7$. The equation is

$$y^2 = x^3 + 7$$

with parameters in hexadecimal

$$\begin{aligned} p = &\ \text{FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE FFFFFC2F} \\ G = &\ (\text{79BE667E F9DCBBAC 55A06295 CE870B07 029BFCDB 2DCE28D9 59F2815B 16F81798,} \\ &\ \text{483ADA77 26A3C465 5DA4FBFC 0E1108A8 FD17B448 A6855419 9C47D08F FB10D4B8}) \\ n = &\ \text{FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE BAAEDCE6 AF48A03B BFD25E8C D0364141} \end{aligned}$$

The curve order n defines the number of elements (points) generated by the generator G on the curve. Exponentiation of the generator $g^a \mod p$ becomes a point multiplication with the generator $a \cdot G$.

Points addition

With two distinct points P and Q on the curve $\mathcal{E}$, geometrically the resulting point of the addition is the inverse point, $(x, -y)$ of the intersection point with a straight line between P and Q. An infinity point $\mathcal{O}$ represents the identity element in the group. Algebraically the resulting point is obtained with:

$$\begin{aligned} P + Q = Q + P = P + Q + \mathcal{O} &= R \\ (x_p, y_p) + (x_q, y_q) &= (x_r, y_r) \\ x_r &\equiv \lambda^2 - x_p - x_q \pmod p \\ y_r &\equiv \lambda(x_p - x_r) - y_p \pmod p \\ \lambda &= \frac{y_q - y_p}{x_q - x_p} \\ &\equiv (y_q - y_p)(x_q - x_p)^{-1} \pmod p \end{aligned} \tag{4.1}$$

Point doubling

For P and Q equal, the formula is similar and can be simplified, the tangent to the curve $\mathcal{E}$ at point P determines R.

$$\begin{aligned} P + P &= 2P = R \\ (x_p, y_p) + (x_p, y_p) &= (x_r, y_r) \\ x_r &\equiv \lambda^2 - 2x_p \pmod{p} \\ y_r &\equiv \lambda(x_p - x_r) - y_p \pmod{p} \\ \lambda &= \frac{3x_p^2 + a}{2y_p} \\ &\equiv (3x_p^2 + a)(2y_p)^{-1} \pmod{p} \end{aligned} \tag{4.2}$$

Point multiplication

A point P can be multiplied by a scalar d. The straightforward way of computing a point multiplication is through repeated addition where $dP = P_1 + P_2 + \cdots + P_d$.

Lemma 4.1.1 (Elliptic Curve Discrete Logarithm Problem) *Given a multiple Q of P where $Q = nP$ it is infeasible to find n if n is large.*

Lemma 4.1.2 (Point Order) *A point P has order 2 if $P + P = \mathcal{O}$, and therefore $P = -P$. A point Q has order 3 if $Q + Q + Q = \mathcal{O}$, and therefore $Q + Q = -Q$.*

4.1.2 Paillier cryptosystem

> Wikipedia: The Paillier cryptosystem, invented by and named after Pascal Paillier in 1999, is a probabilistic asymmetric algorithm for public key cryptography. The problem of computing n-th residue classes is believed to be computationally difficult. The decisional composite residuosity assumption is the intractability hypothesis upon which this cryptosystem is based.

Encryption

With a public key (n, g) and a message $m < n$, select a random $r < n$ and compute the ciphertext $c = g^m \cdot r^n \mod n^2$ to encrypt the plaintext.

Decryption

With a private key (n, g, λ, μ) and a ciphertext $c \in \mathbb{Z}_{n^2}^*$ compute the plaintext as $m = L(c^\lambda \mod n^2) \cdot \mu \mod n$ where $L(x) = \frac{x-1}{n}$.

4.1.3 Signature schemes

Digital Signature Algorithm

> Wikipedia: The Digital Signature Algorithm (DSA) is a Federal Information Processing Standard for digital signatures. In August 1991 the National Institute of Standards and Technology (NIST) proposed DSA for use in their Digital Signature Standard (DSS) and adopted it as FIPS 186 in 1993.

Signing With public parameters (p, q, g), `hash` the hashing function, m the message, and $x \in \mathbb{Z}_q$ the private key.

- Generate a random $k \in \mathbb{Z}_q$
- Calculate $r \equiv (g^k \pmod p) \pmod q : r \neq 0$
- Calculate $s \equiv k^{-1}(\texttt{hash}(m) + xr) \pmod q : s \neq 0$
- The signature is (r, s)

Verifying With public parameters (p, q, g), `hash` the hashing function, m the message, (r, s) the signature, and $y = g^x \mod p$ the public key.

- Reject the signature if $r, s \notin \mathbb{Z}_q$
- Calculate $w \equiv s^{-1} \pmod q$
- Calculate $u_1 \equiv \texttt{hash}(m) \cdot w \pmod q$
- Calculate $u_2 \equiv rw \pmod q$
- Calculate $v \equiv (g^{u_1} y^{u_2} \pmod p) \pmod q$
- The signature is valid if $v = r$

Elliptic Curve Digital Signature Algorithm

ECDSA is a variant of DSA which uses elliptic curve cryptography and requires a different set of parameters and smaller keys.

Signing With public parameters $(\mathcal{E}, G, n)$, `hash` the hashing function, m the message, and $x \in \mathbb{Z}_n$ the private key.

- Generate a random $k \in \mathbb{Z}_n$
- Calculate $(x_1, y_1) = k \cdot G$
- Calculate $r \equiv x_1 \pmod n : r \neq 0$
- Calculate $s \equiv k^{-1}(\texttt{hash}(m) + xr) \pmod n : s \neq 0$
- The signature is (r, s)

Verifying With public parameters $(\mathcal{E}, G, n)$, `hash` the hashing function, m the message, (r, s) the signature, and $Q = x \cdot G$ the public key.

- Reject the signature if $r, s \notin \mathbb{Z}_n$
- Calculate $w \equiv s^{-1} \pmod{n}$
- Calculate $u_1 \equiv \texttt{hash}(m) \cdot w \pmod{n}$
- Calculate $u_2 \equiv rw \pmod{n}$
- Calculate the curve point $(x_1, y_1) = u_1 \cdot G + u_2 \cdot Q$ if $(x_1, y_1) = \mathcal{O}$ then the signature is invalid
- The signature is valid if $r \equiv x_1 \pmod{n}$

Analysis of the schemes

In (r, s) the computation of the part s remains the same in each signature scheme, the only difference for s is the modulus applied. In DSA modulo q is used, i.e., the order of the generator g modulo p, while in ECDSA modulo n is used, i.e., the order of the generator G on the curve $\mathcal{E}$.

The biggest adaptation is on how to calculate the part r from the private random k. In DSA the generator g is used with at first modulo p and then modulo q while in ECDSA the curve is used. The signer generates a point and uses the coordinate x_1 modulo n.

Postulate 4.1.3 *The statement* $a \equiv g^b \pmod{p}$ *is equivalent in terms of security to* $a = b \cdot G$ *and* $a \equiv (g^b \pmod{p}) \pmod{q}$ *is equivalent to* $a \equiv x \pmod{n} : (x, y) = b \cdot G$.

The previous postulate is used to adapt zero-knowledge proofs from DSA to ECDSA hereafter. Lack of time has not permitted further research into this postulate.

4.2 Threshold scheme

The "Two-party generation of DSA signatures" scheme presented by MacKenzie and Reiter [16], as mentioned before at the end of the chapter 3, is an asymmetric scheme, i.e., at the end of the protocol only the initiator can retrieve the full signature. The scheme proposed is a cryptographic (1,2)-threshold, i.e., one corrupted player can occur out of the two players, and the scheme remains safe. It is worth noting that this is qualified as an optimal (t, n)-threshold scheme, i.e., $t = n - 1$, because if only one honest player remains the safety is guaranteed.

The presented scheme in the original paper uses a multiplicative shared secret and a multiplicative shared private random value. The secret x is shared between Alice and Bob, so that Alice holds the secret value $x_1 \in \mathbb{Z}_q$ and Bob $x_2 \in \mathbb{Z}_q$ such that $x \equiv x_1 x_2 \pmod{q}$. Along with the public key y, $y_1 \equiv g^{x_1} \pmod{p}$ and $y_2 \equiv g^{x_2} \pmod{p}$ are public. Alice holds a private key, from now on mentioned as sk, corresponding to a public encryption key pk. Alice also knows a public encryption key pk' for which she does not know the private key sk'. Here the Paillier homomorphic cryptosystem

is used as the encryption scheme, but other homomorphic encryption systems can be used to implement the scheme. Alice and Bob know a set of parameters used for both zero-knowledge proofs.

Starting now the initialization is not taken into account, and the author assumes that reader has a good understanding of the original scheme [16]. The choice was made to focus the work on the signing protocol and because the implementation is not directly part of the cryptographic `C` library. This initialization can be further researched.

4.2.1 Adapting the scheme

Except for the zero-knowledge proofs, the adaptation is trivial and just requires the same adaptation of the DSA signature scheme and ECDSA signature scheme, i.e., compute the r value with the curve. The following figures describe the adapted scheme. The adapted protocol keeps the same messages, so they are not repeated. The postulate 4.1.3 is used to perform the adaptation.

The secret remains shared multiplicatively so that so that Alice holds the secret value $x_1 \in \mathbb{Z}_n$ and Bob $x_2 \in \mathbb{Z}_n$ such that $x \equiv x_1 x_2 \pmod{n}$. Alice holds her public key $Q_1 = x_1 \cdot G$ and Bob $Q_2 = x_2 \cdot G$ such that $Q = x_1 \cdot Q_2$ for Alice or $Q = x_2 \cdot Q_1$ for Bob. The notation $\cdot$ is used to denote the point multiplication over the curve.

All the random values k are chosen in $\mathbb{Z}_n$ instead of $\mathbb{Z}_q$, also in the case of deterministic signature. All the computation modulo q is replaced by modulo n, as shown in the previous digital signature recap. Values R_2 and R become points. Verifications of values R_2 and R become point verifications on the curve and r' is calculated by Alice and Bob as shown in the other reminder. The value cq added to the homomorphic encrypted signature is transformed into cn to hide values z_2 and $x_2 z_2$ in $\mathbb{Z}_n$.

The author noticed an error of notation in the original paper. On the second line Alice computes $z_1 \equiv (k_1)^{-1} \pmod{n}$ and the original paper uses the random value selection $\overset{R}{\leftarrow}$ instead of a standard assignation $\leftarrow$. This error is corrected in the following version of the protocol and does not affect the security.

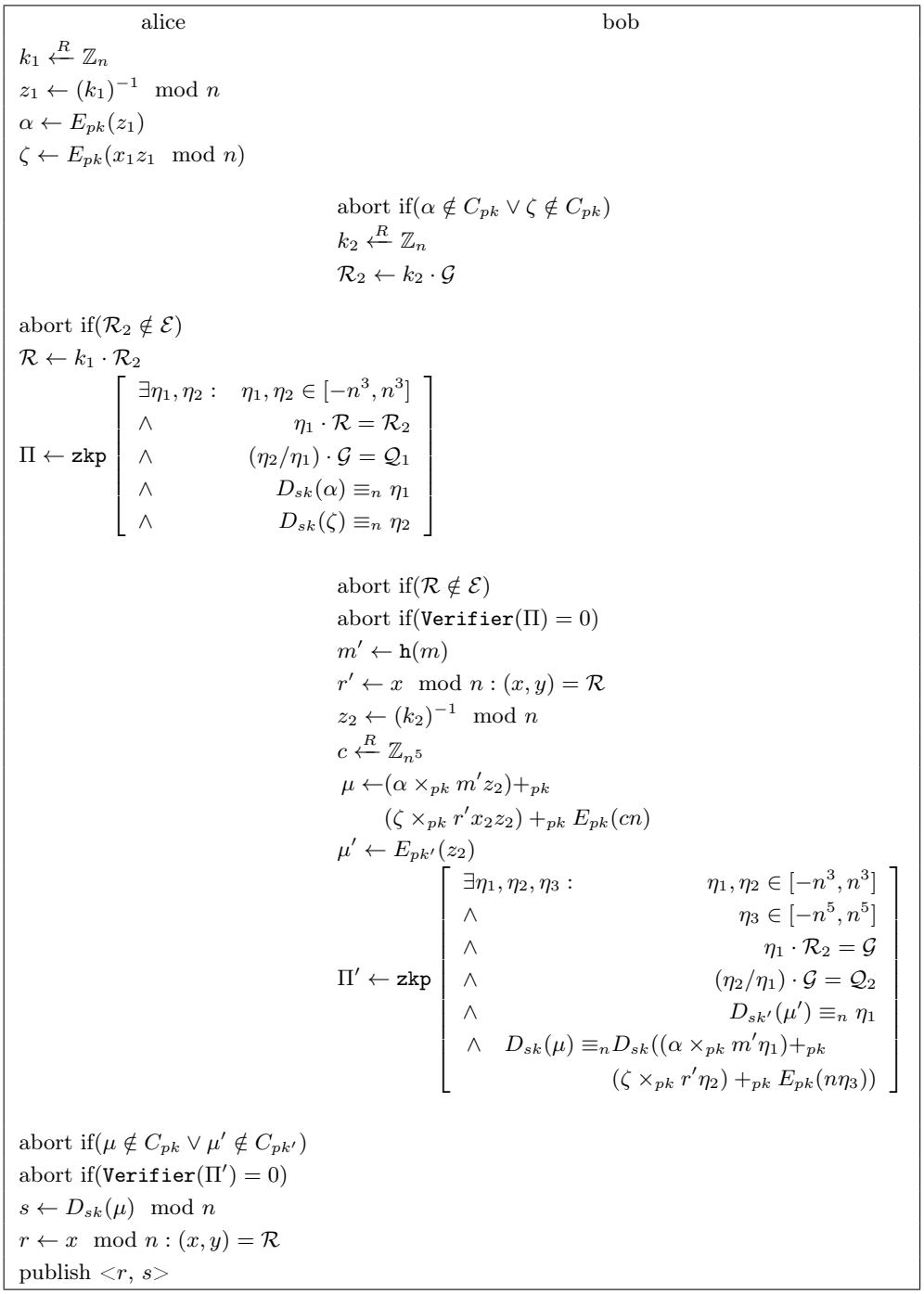

Figure 4.1 Adapted protocol for ECDSA

4.2.2 Adapting zero-knowledge proofs

Because the protocol designs proofs for the DSA architecture the values tested in the proofs are values in $\mathbb{Z}_q$. These values are used to create a challenge value e with two hash functions, a different one per proof. For ECDSA some of these values become points, so some equations need to be adapted. The adapted protocol serializes

points in the long form, 65 bytes starting with `0x04` and two 32 byte coordinates for (x, y). As mentioned in the original paper, the variables names are not consistent with the first part of the paper. Starting now the variable names follow the same notation as the original paper and are therefore no longer consistent with the previous pages.

Zero-knowledge proof Π

The first zero-knowledge proof Π is created by Alice to prove to Bob that she acted correctly and has encrypted coherent data with Paillier encryption, proving the ownership and the validity of the two encrypted values in relation to the public address Q_1 with $(x_1 z_1 / z_1) \cdot G = Q_1$. The proof states that the encrypted value α is related to R and R_2 such that $(k_1)^{-1} \cdot R = ((k_1)^{-1} k_1 k_2) \cdot G = k_2 \cdot G = R_2$.

$$\Pi \leftarrow \texttt{zkp} \left[\begin{array}{ll} \exists x_1, x_2 : & x_1, x_2 \in [-n^3, n^3] \\ \wedge & x_1 \cdot \mathcal{C} = \mathcal{W}_1 \\ \wedge & (x_2 / x_1) \cdot \mathcal{D} = \mathcal{W}_2 \\ \wedge & D_{sk}(m_1) \equiv_n x_1 \\ \wedge & D_{sk}(m_2) \equiv_n x_2 \end{array} \right]$$

Figure 4.2 The proof Π adapted

To help the reader a mapping of old variable names and the new variable name is done.

$x_1 = z_1$	$\mathcal{C} = \mathcal{R}$
$x_2 = x_1 z_1 \mod n$	$\mathcal{D} = \mathcal{G}$
$m_1 = \alpha$	$\mathcal{W}_1 = \mathcal{R}_2$
$m_2 = \zeta$	$\mathcal{W}_2 = \mathcal{Q}_1$

Table 4.1 Mapping between the protocol's variable names and the ZKP Π

$$\langle z_1, z_2, \mathcal{Y}, e, s_1, s_2, s_3, t_1, t_2, t_3, t_4 \rangle \leftarrow \Pi$$

Verify $s_1, t_1 \in \mathbb{Z}_{n^3}$	$\mathcal{V}_1 \leftarrow (t_1 + t_2) \cdot \mathcal{D} + (-e) \cdot \mathcal{Y}$
$\mathcal{U}_1 \leftarrow s_1 \cdot \mathcal{C} + (-e) \cdot \mathcal{W}_1$	$\mathcal{V}_2 \leftarrow s_1 \cdot \mathcal{W}_2 + t_2 \cdot \mathcal{D} + (-e) \cdot \mathcal{Y}$
$u_2 \leftarrow g^{s_1} (s_2)^N (m_1)^{-e} \mod N^2$	$v_3 \leftarrow g^{t_1} (t_3)^N (m_2)^{-e} \mod N^2$
$u_3 \leftarrow (h_1)^{s_1} (h_2)^{s_3} (z_1)^{-e} \mod \tilde{N}$	$v_4 \leftarrow (h_1)^{t_1} (h_2)^{t_4} (z_2)^{-e} \mod \tilde{N}$

Verify $e = \texttt{hash}(\mathcal{C}, \mathcal{W}_1, \mathcal{D}, \mathcal{W}_2, m_1, m_2, z_1, \mathcal{U}_1, u_2, u_3, z_2, \mathcal{Y}, \mathcal{V}_1, \mathcal{V}_2, v_3, v_4)$

Figure 4.3 Adaptation of the verification of Π in ECDSA

$\alpha \xleftarrow{R} \mathbb{Z}_{n^3}$	$\delta \xleftarrow{R} \mathbb{Z}_{n^3}$
$\beta \xleftarrow{R} \mathbb{Z}_N^*$	$\mu \xleftarrow{R} \mathbb{Z}_N^*$
$\gamma \xleftarrow{R} \mathbb{Z}_{n^3\tilde{N}}$	$\nu \xleftarrow{R} \mathbb{Z}_{n^3\tilde{N}}$
$\rho_1 \xleftarrow{R} \mathbb{Z}_{n\tilde{N}}$	$\rho_2 \xleftarrow{R} \mathbb{Z}_{n\tilde{N}}$
	$\rho_3 \xleftarrow{R} \mathbb{Z}_n$
	$\epsilon \xleftarrow{R} \mathbb{Z}_n$
$z_1 \leftarrow (h_1)^{x_1}(h_2)^{\rho_1} \mod \tilde{N}$	$z_2 \leftarrow (h_1)^{x_2}(h_2)^{\rho_2} \mod \tilde{N}$
$\mathcal{U}_1 \leftarrow \alpha \cdot \mathcal{C}$	$\mathcal{Y} \leftarrow (x_2 + \rho_3) \cdot \mathcal{D}$
$u_2 \leftarrow g^\alpha \beta^N \mod N^2$	$\mathcal{V}_1 \leftarrow (\delta + \epsilon) \cdot \mathcal{D}$
$u_3 \leftarrow (h_1)^\alpha (h_2)^\gamma \mod \tilde{N}$	$\mathcal{V}_2 \leftarrow \alpha \cdot \mathcal{W}_2 + \epsilon \cdot \mathcal{D}$
	$v_3 \leftarrow g^\delta \mu^N \mod N^2$
	$v_4 \leftarrow (h_1)^\delta (h_2)^\nu \mod \tilde{N}$
$e \leftarrow \texttt{hash}(\mathcal{C}, \mathcal{W}_1, \mathcal{D}, \mathcal{W}_2, m_1, m_2, z_1, \mathcal{U}_1, u_2, u_3, z_2, \mathcal{Y}, \mathcal{V}_1, \mathcal{V}_2, v_3, v_4)$	
$s_1 \leftarrow ex_1 + \alpha$	$t_1 \leftarrow ex_2 + \delta$
$s_2 \leftarrow (r_1)^e \beta \mod N$	$t_2 \leftarrow e\rho_3 + \epsilon \mod n$
$s_3 \leftarrow e\rho_1 + \gamma$	$t_3 \leftarrow (r_2)^e \mu \mod N^2$
	$t_4 \leftarrow e\rho_2 + \nu$
$\Pi \leftarrow \langle z_1, z_2, \mathcal{Y}, e, s_1, s_2, s_3, t_1, t_2, t_3, t_4 \rangle$	

Figure 4.4 Adaptation of the construction of Π *in ECDSA*

Zero-knowledge proof Π'

The second zero-knowledge proof is created by Bob to prove to Alice that he acted honestly according to the protocol. The proof states that the point $\mathcal{R}_2$ is generated accordingly to the value z_2 and so to the value k_2. That the public key $\mathcal{Q}_2$ is related to the values z_2 and $x_2 z_2$, and that the encrypted values μ and μ' are correctly composed. Again a mapping between the old names and the new names is done to help the reader.

$x_1 = z_2$	$\mathcal{C} = \mathcal{R}_2$
$x_2 = x_2 z_2 \mod n$	$\mathcal{D} = \mathcal{G}$
$x_3 = c \mod n$	$\mathcal{W}_1 = \mathcal{G}$
$m_1 = \mu'$	$\mathcal{W}_2 = \mathcal{Q}_2$
$m_2 = \mu$	$m_3 = \alpha$
$m_4 = \zeta$	

Table 4.2 Mapping between the protocol's variable names and the ZKP Π'

$$\Pi' \leftarrow \texttt{zkp}\left[\begin{array}{ll} \exists x_1, x_2, x_3 : & x_1, x_2 \in [-n^3, n^3] \\ \wedge & x_3 \in [-n^5, n^5] \\ \wedge & x_1 \cdot \mathcal{C} = \mathcal{W}_1 \\ \wedge & (x_2/x_1) \cdot \mathcal{D} = \mathcal{W}_2 \\ \wedge & D_{sk'}(m_1) \equiv_n x_1 \\ \wedge & D_{sk}(m_2) \equiv_n D_{sk}((m_3 \times_{pk} m' x_1) +_{pk} \\ & \qquad (m_4 \times_{pk} r' x_2) +_{pk} E_{pk}(n x_3)) \end{array}\right]$$

Figure 4.5 The proof Π' adapted

In the first Python implementation, the ZKP Π' did not pass the validation. To investigate from where the problem comes all equations from the validation was done by hand. By expanding the equation v_3, we can see that the result does not match the expected value.

Correcting the verification of Π' If $x_1 = z_2$, $x_2 = x_2 z_2$, $x_3 = c$, and $m_2 = \mu$ such that $\mu = (\alpha)^{m' x_1} (\zeta)^{r' x_2} g^{n x_3} (r_2)^N$, then the equation v_3 in the validation process does not correspond to construction of v_3 in the original paper. The result in the verification process Π' need to match $v_3 \leftarrow (m_3)^{\alpha} (m_4)^{\delta} g^{n\sigma} \mu^N \mod N^2$. The original equation proposes $v_3 \leftarrow (m_3)^{s_1} (m_4)^{t_1} g^{n t_5} (t_3)^N (m_2)^{-e} \mod N^2$ which does not include m' and r' present in μ, so m_2 cannot be used correctly as exposed next.

$$\begin{aligned} v_3 &\equiv (m_3)^{s_1} (m_4)^{t_1} g^{n t_5} (t_3)^N (m_2)^{-e} \pmod{N^2} \\ &\equiv (m_3)^{e x_1 + \alpha} (m_4)^{e x_2 + \beta} g^{n(e x_3 + \sigma)} \left((r_2)^e \mu\right)^N \left((m_3)^{m' x_1} (m_4)^{r' x_2} g^{n x_3} (r_2)^N\right)^{-e} \\ &\equiv (m_3)^{e x_1 + \alpha} (m_4)^{e x_2 + \beta} g^{n(e x_3 + \sigma)} (r_2)^{eN} \mu^N (m_3)^{-e m' x_1} (m_4)^{-e r' x_2} g^{-e n x_3} (r_2)^{-eN} \\ &\equiv (m_3)^{e x_1 + \alpha - e m' x_1} (m_4)^{e x_2 + \beta - e r' x_2} g^{e n x_3 + n\sigma - e n x_3} (r_2)^{eN - eN} \mu^N \\ &\equiv (m_3)^{e x_1 + \alpha - e m' x_1} (m_4)^{e x_2 + \beta - e r' x_2} g^{n\sigma} \mu^N \end{aligned} \tag{4.3}$$

The equation v_3 needs to be adapted to include $x_4 = m'$ and $x_5 = r'$ (m' and r' cannot be include directly in x_1 and x_2 without breaking equations u_1, u_2, u_3, v_2.) Two new parameters $s_4 \leftarrow e x_1 x_4 + \alpha$ and $t_7 \leftarrow e x_2 x_5 + \delta$ are added into the proof to correct the equation.

$$\begin{aligned} v_3 &\equiv (m_3)^{s_4} (m_4)^{t_7} g^{n t_5} (t_3)^N (m_2)^{-e} \pmod{N^2} \\ &\equiv (m_3)^{e x_1 x_4 + \alpha} (m_4)^{e x_2 x_5 + \beta} g^{n(e x_3 + \sigma)} \left((r_2)^e \mu\right)^N \left((m_3)^{x_1 x_4} (m_4)^{x_2 x_5} g^{n x_3} (r_2)^N\right)^{-e} \\ &\equiv (m_3)^{e x_1 x_4 + \alpha} (m_4)^{e x_2 x_5 + \beta} g^{n(e x_3 + \sigma)} (r_2)^{eN} \mu^N (m_3)^{-e x_1 x_4} (m_4)^{-e x_2 x_5} g^{-e n x_3} (r_2)^{-eN} \\ &\equiv (m_3)^{e x_1 x_4 + \alpha - e x_1 x_4} (m_4)^{e x_2 x_5 + \beta - e x_2 x_5} g^{e n x_3 + n\sigma - e n x_3} (r_2)^{eN - eN} \mu^N \\ &\equiv (m_3)^{\alpha} (m_4)^{\beta} g^{n\sigma} \mu^N \end{aligned} \tag{4.4}$$

Further research needs to be done to validate that the introduction of these two new parameters s_4 and t_7 does not break the security model.

$\alpha \xleftarrow{R} \mathbb{Z}_{n^3}$	$\delta \xleftarrow{R} \mathbb{Z}_{n^3}$
$\beta \xleftarrow{R} \mathbb{Z}^*_{N'}$	$\mu \xleftarrow{R} \mathbb{Z}^*_N$
$\gamma \xleftarrow{R} \mathbb{Z}_{n^3\tilde{N}}$	$\nu \xleftarrow{R} \mathbb{Z}_{n^3\tilde{N}}$
$\rho_1 \xleftarrow{R} \mathbb{Z}_{n\tilde{N}}$	$\rho_2 \xleftarrow{R} \mathbb{Z}_{n\tilde{N}}$
	$\rho_3 \xleftarrow{R} \mathbb{Z}_n$
	$\rho_4 \xleftarrow{R} \mathbb{Z}_{n^5\tilde{N}}$
	$\epsilon \xleftarrow{R} \mathbb{Z}_n$
	$\sigma \xleftarrow{R} \mathbb{Z}_{n^7}$
	$\tau \xleftarrow{R} \mathbb{Z}_{n^7\tilde{N}}$
$z_1 \leftarrow (h_1)^{x_1}(h_2)^{\rho_1} \mod \tilde{N}$	$z_2 \leftarrow (h_1)^{x_2}(h_2)^{\rho_2} \mod \tilde{N}$
$\mathcal{U}_1 \leftarrow \alpha \cdot \mathcal{C}$	$\mathcal{Y} \leftarrow (x_2 + \rho_3) \cdot \mathcal{D}$
$u_2 \leftarrow (g')^{\alpha}\beta^{N'} \mod (N')^2$	$\mathcal{V}_1 \leftarrow (\delta + \epsilon) \cdot \mathcal{D}$
$u_3 \leftarrow (h_1)^{\alpha}(h_2)^{\gamma} \mod \tilde{N}$	$\mathcal{V}_2 \leftarrow \alpha \cdot \mathcal{W}_2 + \epsilon \cdot \mathcal{D}$
	$v_3 \leftarrow (m_3)^{\alpha}(m_4)^{\delta} g^{n\sigma}\mu^N \mod N^2$
	$v_4 \leftarrow (h_1)^{\delta}(h_2)^{\nu} \mod \tilde{N}$
	$z_3 \leftarrow (h_1)^{x_3}(h_2)^{\rho_4} \mod \tilde{N}$
	$v_5 \leftarrow (h_1)^{\sigma}(h_2)^{\tau} \mod \tilde{N}$
$e \leftarrow \texttt{hash'}(\mathcal{C}, \mathcal{W}_1, \mathcal{D}, \mathcal{W}_2, m_1, m_2, z_1, \mathcal{U}_1, u_2, u_3, z_2, z_3, \mathcal{Y}, \mathcal{V}_1, \mathcal{V}_2, v_3, v_4, v_5)$	
$s_1 \leftarrow ex_1 + \alpha$	$t_1 \leftarrow ex_2 + \delta$
$s_2 \leftarrow (r_1)^e\beta \mod N'$	$t_2 \leftarrow e\rho_3 + \epsilon \mod n$
$s_3 \leftarrow e\rho_1 + \gamma$	$t_3 \leftarrow (r_2)^e\mu \mod N$
$s_4 \leftarrow ex_1x_4 + \alpha$	$t_4 \leftarrow e\rho_2 + \nu$
	$t_5 \leftarrow ex_3 + \sigma$
	$t_6 \leftarrow e\rho_4 + \tau$
	$t_7 \leftarrow ex_2x_5 + \delta$
$\Pi' \leftarrow \langle z_1, z_2, z_3, \mathcal{Y}, e, s_1, s_2, s_3, s_4, t_1, t_2, t_3, t_4, t_5, t_6, t_7 \rangle$	

Figure 4.6 Adaptation of the construction of Π' in ECDSA

$$\langle z_1, z_2, z_3, \mathcal{Y}, e, s_1, s_2, s_3, s_4, t_1, t_2, t_3, t_4, t_5, t_6, t_7 \rangle \leftarrow \Pi'$$

Verify $s_1, t_1 \in \mathbb{Z}_{n^3}$	$\mathcal{V}_1 \leftarrow (t_1 + t_2) \cdot \mathcal{D} + (-e) \cdot \mathcal{Y}$
Verify $t_5 \in \mathbb{Z}_{n^7}$	$\mathcal{V}_2 \leftarrow s_1 \cdot \mathcal{W}_2 + t_2 \cdot \mathcal{D} + (-e) \cdot \mathcal{Y}$
$\mathcal{U}_1 \leftarrow s_1 \cdot \mathcal{C} + (-e) \cdot \mathcal{W}_1$	$v_3 \leftarrow (m_3)^{s_4} (m_4)^{t_7} g^{n t_5} (t_3)^N (m_2)^{-e} \mod N^2$
$u_2 \leftarrow (g')^{s_1} (s_2)^{N'} (m_1)^{-e} \mod (N')^2$	$v_4 \leftarrow (h_1)^{t_1} (h_2)^{t_4} (z_2)^{-e} \mod \tilde{N}$
$u_3 \leftarrow (h_1)^{s_1} (h_2)^{s_3} (z_1)^{-e} \mod \tilde{N}$	$v_5 \leftarrow (h_1)^{t_5} (h_2)^{t_6} (z_3)^{-e} \mod \tilde{N}$

$$\text{Verify } e = \texttt{hash'}(\mathcal{C}, \mathcal{W}_1, \mathcal{D}, \mathcal{W}_2, m_1, m_2, z_1, \mathcal{U}_1, u_2, u_3, z_2, z_3, \mathcal{Y}, \mathcal{V}_1, \mathcal{V}_2, v_3, v_4, v_5)$$

Figure 4.7 Adaptation of the verification of Π' *in ECDSA*

4.3 Threshold Hierarchical Determinitic Wallets

Hierarchical deterministic wallets are sophisticated wallets in which new keys can be generated from a previous key. Adapting hierarchical deterministic wallets with a threshold scheme can be achieved by sharing the private key additively among $P = \{p_1, \ldots, p_i\}$ players

$$\begin{aligned} pk_i &= sk_i \cdot G \\ sk_{mas} &= \sum_{j=1}^{i} sk_j \bmod n \\ pk_{mas} &= \left[\sum_{j=1}^{i} sk_j \bmod n\right] \cdot G \\ &= \sum_{j=1}^{i} (sk_j \cdot G) = \sum_{j=1}^{i} pk_i \end{aligned}$$

or multiplicatively

$$\begin{aligned} sk_{mas} &= \prod_{j=1}^{i} sk_j \bmod n \\ pk_{mas} &= \left[\prod_{j=1}^{i} sk_j \bmod n\right] \cdot G \\ &= \Big(\big((G \cdot sk_1) \cdot sk_2\big) \ldots\Big) \cdot sk_j \end{aligned}$$

In the additive case, the master public key pk_{mas} is also the sum of all the points pk_i, which means that if each player publishes his share point, everyone can compute the master public key. The multiplicative sharing is more communication intensive because the computation of the public key is sequential instead of parallel.

An extended private key share is a tuple of (sk_i, c) with sk_i the regular private key and c the chain code, such that c is the same for each player. In the following, it is assumed that the private key is shared multiplicatively.

4.3.1 Private parent key to private child key

The function `CKDpriv` computes a child extended private key from the parent extended private key. The derivation can be *hardened*. This proposal differs from the BIP32 [19] standard in the chain derivation process. The `ser` function and `point` function are the same as described in the BIP.

$$f(l) = \begin{cases} \texttt{HMAC-SHA256}(c_{par}, \texttt{0x00} \mid\mid \texttt{ser}_{256}(sk_i^{par}) \mid\mid \texttt{ser}_{32}(k)) & \text{if } k \geq 2^{31} \\ \texttt{HMAC-SHA256}(c_{par}, \texttt{ser}_p(\texttt{point}(sk_{mas}^{par})) \mid\mid \texttt{ser}_{32}(k)) & \text{if } k < 2^{31} \end{cases}$$

$$sk_i \equiv l \cdot sk_i^{par} \pmod{n}$$

The function $f(l)$ computes the partial share l at index k, such that multiplied with the parent private key share sk_i^{par} for the player i the result is sk_i.

4.3.2 Public parent key to public child key

The function `CKDpub` computes a child extend public key from the parent extended public key. It is worth noting that it is not possible to compute a *hardened* derivation without the private parent key, and that every player updates the master public key for the threshold not their public key share.

$$f(l) = \begin{cases} \texttt{failure} & \text{if } k \geq 2^{31} \\ \texttt{HMAC-SHA256}(c_{par}, \texttt{ser}_p(pk_{mas}^{par}) \,||\, \texttt{ser}_{32}(k)) & \text{if } k < 2^{31} \end{cases}$$

$$\begin{aligned} pk_{mas} &= l \cdot pk_{mas}^{par} \\ &= l \cdot (sk_{mas}^{par} \cdot G) \\ &= (l \cdot sk_{mas}^{par} \bmod n) \cdot G \end{aligned}$$

4.3.3 Child key share derivation

The protocol assumes that one of the players $p_i \in P = \{p_1, \ldots, p_i\}$ is designated as the leader L. The function `CKSD` computes a threshold child extended key share from the threshold parent extended key share for the derivation index k. It is worth noting that only the leader L uses `CKDpriv` and if the derivation is *hardened*, i.e., if $k \geq 2^{31}$, a special case occurred and a round of communication is needed. Let's define `CKSD` for $k < 2^{31}$

$$\forall p_i \in P : f(t) = \begin{cases} \texttt{CKDpriv}(k) & \text{if } p_i = L \\ \texttt{CKDpub}(k) & \text{if } p_i \neq L \end{cases} \tag{4.5}$$

such that

$$\begin{aligned} sk_i &= \begin{cases} sk_i^{par} \cdot t & \text{if } p_i = L \\ sk_i^{par} & \text{if } p_i \neq L \end{cases} \\ sk_{mas} &= \left[\prod_{j=1}^{i} sk_j^{par} \right] \cdot t \\ &= sk_{mas}^{par} \cdot t \end{aligned} \tag{4.6}$$

and then $\forall p_i \in P$

$$\begin{aligned} pk_{mas} &= pk_{mas}^{par} \cdot t \\ &= (sk_{mas}^{par} \cdot G) \cdot t \\ &= \left[\prod_{j=1}^{i} sk_j^{par} \right] \cdot G \end{aligned} \tag{4.7}$$

At each derivation index each player updates their chain code. The derivation does not depend on the secret key because the chain code must remain deterministic and have the same value for each player without requiring communication round.

$$c_i = \texttt{HMAC-SHA256}(c_i^{par}, \texttt{ser}_{32}(k)) \tag{4.8}$$

If the index $k \geq 2^{31}$ the new master public key, only calculable by the master player L, must be revealed to other players. A round of communication is then needed to continue derivation.

In this threshold HD scheme only one private share changes at each derivation. In other words, the master private share is derived either with public information or with private information. If the derivation is private, i.e., *hardened* derivation, then a communication round between the players is necessary, more specifically we assume that a secure broadcast channel is open from the master player to the other players.

This scheme is sufficient for the payment channels. Players negotiate a threshold key used for the $\texttt{Multisig}_i$ address with a root derivation path `m/44'/0'/a'/0'` at the opening of the channel (with variable `a` is related to the channel account number between the client and the provider as shown in the paper.) Then the index `i` in the paper is used to derive each address without requiring any communication. It is worth noting that the root derivation path can also be simplified at `m/a'` or even `m/` because the compatibility with a standard wallet is no longer a requirement. Noted that the version `m/a'` is more flexible and allows multiple channels between a client and a provider with only one threshold key.

4.3.4 Proof-of-concept implementation

A Python proof-of-concept has been coded. A share can be tagged as master share as previously described. The result of the script is presented below, three shares are created, and the first one is tagged as the master share. The root threshold public key `m/` is computed and displayed, then individual shares' addresses are displayed. The share s_1 is derived with and without *hardened* path, as expected the resulting address is different. The master public key resulting from each share derivation for the path `m/44/0/1` is the same as computing the private key with all individual secret shares and getting the associated address, as expected. To note that only the master individual address for `m/` and `m/44/0/1` has changed. The complete implementation is shown in the appendices.

```
=== Threshold addresses ===
Master root public key m/   : 1BF5ZpQMCg3eGDEm51rkiwcKR12UnFu

*** Individual addresses m/ ***
s1: 1tRFxbAfKKowtqrSC3bVUi491hTXqg1
s2: 16uCytSc9oAJyi5FbxmH6NyTJuYkCLj
s3: 1TcYLZUZYd86AFaT58tzFGBW1BVVw7K

*** Hardened derivation for one share ***
s1 m/44/0/1  : 128PvDGSbZuNpz1zG1Mh1fjJFN3eNaTb
s1 m/44/0/1' : 12883vUsA2gyCAcSNogGUMFuCJsrj58

*** Master public key m/44/0/1 ***
s1: 128PvDGSbZuNpz1zG1Mh1fjJFN3eNaTb
s2: 128PvDGSbZuNpz1zG1Mh1fjJFN3eNaTb
s3: 128PvDGSbZuNpz1zG1Mh1fjJFN3eNaTb

Master public key m/44/0/1 : 128PvDGSbZuNpz1zG1Mh1fjJFN3eNaTb

*** Individual addresses m/44/0/1 ***
s1: 1nNL1gozCk4J1agV667kJFmsyu4RvF5
s2: 16uCytSc9oAJyi5FbxmH6NyTJuYkCLj
s3: 1TcYLZUZYd86AFaT58tzFGBW1BVVw7K
```

Listing 4.1 Result of Python proof-of-concept threshold HD wallet

A share is composed of four main pieces of information (i) the secret share, (ii) the chain code, (iii) the tag for the master share, and (iv) the threshold public key. Players can set the threshold public key address after computation. The derive function `d` derives with `CKDpub` or `CKDpriv` depending on the master tag and returns a new share for a given index. The path derivation function `derive` takes a path and generates the chain of shares. In this Implementation, if a share not tagged as master tries to derive a path with a `hardened` index, an exception is raised, and the process stops. However, in a real-world case, a communication process must take place to complete the derivation with an external input to update the master player public key share.

```
252 if __name__ == "__main__":
253     print("=== Threshold addresses ===")
254 
255     chain = ecdsa.gen_priv()
256     # Shares
257     s1 = Share(chain, True, ecdsa.gen_priv())
258     s2 = Share(chain, False, ecdsa.gen_priv())
259     s3 = Share(chain, False, ecdsa.gen_priv())
260 
261     sec = (s1.secret * s2.secret * s3.secret) % ecdsa.n
262     pub = ecdsa.get_pub(sec)
263     add = get(pub)
264     print "Master root public key m/   :", add
265 
266     s1.set_master_pub(pub)
267     s2.set_master_pub(pub)
268     s3.set_master_pub(pub)
269 
270     print "\n*** Individual addresses m/ ***"
271     print "s1:", s1.address()
272     print "s2:", s2.address()
273     print "s3:", s3.address()
274 
275     print "\n*** Hardened derivation for one share ***"
276     print "s1 m/44/0/1  :", get(s1.derive("m/44/0/1").master_pub)
277     print "s1 m/44/0/1' :", get(s1.derive("m/44/0/1'").master_pub)
278 
279     print "\n*** Master public key m/44/0/1 ***"
280     s1 = s1.derive("m/44/0/1")
281     s2 = s2.derive("m/44/0/1")
282     s3 = s3.derive("m/44/0/1")
283     print "s1:", get(s1.master_pub)
284     print "s2:", get(s2.master_pub)
285     print "s3:", get(s3.master_pub)
286 
287     sec = (s1.secret * s2.secret * s3.secret) % ecdsa.n
288     pub = ecdsa.get_pub(sec)
289     add = get(pub)
290     print "\nMaster public key m/44/0/1 :", add
291 
292     print "\n*** Individual addresses m/44/0/1 ***"
293     print "s1:", s1.address()
294     print "s2:", s2.address()
295     print "s3:", s3.address()
```

Listing 4.2 Demonstration of using threshold HD wallet

```
163 class Share(object):
164     def __init__(self, chain, master, secret=ecdsa.gen_priv()):
165         super(Share, self).__init__()
166         self.chain = chain
167         self.master = master
168         self.secret = secret
169         self.master_pub = None
170 
171     def pub(self):
172         return ecdsa.get_pub(self.secret)
173 
174     def address(self):
175         return get(self.pub())
176 
177     def set_master_pub(self, pub):
178         self.master_pub = pub
179 
180     def d_pub(self, i):
181         if i >= pow(2, 31): # Only not hardened
182             raise Exception("Impossible to hardened")
183         k = "%x" % self.chain
184         data = "00%s%08x" % (ecdsa.expand_pub(self.master_pub), i)
185         hmac = hashlib.pbkdf2_hmac('sha256', k, data, 100)
186         point = ecdsa.point_mult(self.master_pub, long(binascii.hexlify(hmac), 16))
187         data = "%08x" % (i)
188         hmac = hashlib.pbkdf2_hmac('sha256', k, data, 100)
189         c = long(binascii.hexlify(hmac), 16)
190         share = Share(c, self.master, self.secret)
191         share.set_master_pub(point)
192         return share
193 
194     def d_priv(self, i):
195         k = "%x" % self.chain
196         data = "%08x" % (i)
197         hmac = hashlib.pbkdf2_hmac('sha256', k, data, 100)
198         c = long(binascii.hexlify(hmac), 16)
199         if i >= pow(2, 31): # Hardened
200             data = "00%32x%08x" % (self.secret, i)
201         else: # Not hardened
202             data = "00%s%08x" % (ecdsa.expand_pub(self.master_pub), i)
203         hmac = hashlib.pbkdf2_hmac('sha256', k, data, 100)
204         key = long(binascii.hexlify(hmac), 16) * self.secret
205         point = ecdsa.point_mult(self.master_pub, long(binascii.hexlify(hmac), 16))
206         share = Share(c, self.master, key)
207         share.set_master_pub(point)
208         return share
209 
210     def d(self, index):
211         if self.master:
212             return self.d_priv(index)
213         else:
214             return self.d_pub(index)
215 
216     def derive(self, path):
217         path = string.split(path, "/")
218         if path[0] == "m":
219             path = path[1:]
220             share = self
221             for derivation in path:
222                 if "'" in derivation:
223                     i = int(derivation.replace("'", "")) + pow(2, 31)
224                     share = share.d(i)
225                 else:
226                     i = int(derivation)
227                     share = share.d(i)
228             return share
229         else:
230             return False
```

Listing 4.3 Construction of a share for a threshold HD wallet

4.4 Threshold deterministic signatures

One of the simplest ways to compromise the private key in ECDSA, or in DSA, is to select a weak pseudo-random number generator for k or even worse, select a static value for k. This problem already affected Sony in December 2010 when a group of hackers calling itself *fail0verflow* announced the recovery of the ECDSA private key used to sign software for the PlayStation 3. The recovery process is simple.

Given two signatures (r, s) and (r, s') employing the same unknown k for different messages m and m'. Let's define x as the private key, z as the hash of m and z' of m', an attacker can calculate

$$\begin{aligned} s &\equiv k^{-1}(z + rx) \pmod{n} \\ s' &\equiv k^{-1}(z' + rx) \pmod{n} \\ s - s' &\equiv k^{-1}(z + rx) - k^{-1}(z' + rx) \pmod{n} \\ &\equiv k^{-1}(z - z') \pmod{n} \\ k &\equiv \frac{z - z'}{s - s'} \pmod{n} \\ x &\equiv \frac{sk - z}{r} \pmod{n} \end{aligned} \tag{4.9}$$

However, this issue can be prevented by a deterministic generation of k, as described by RFC 6979 [20]. The random value k can be generated deterministically by using a `HMAC` function such that the parameters are the private key and the message to sign.

The other positive side is that signatures for the same key pair and the same message are deterministic, i.e., if the same message multiple is signed times, the signature remains the same. This determinism is also a significant advantage in Bitcoin to reduce transaction malleability (nevertheless, the signer can still choose to sign with non-deterministic nonce.) The threshold scheme can also enjoy the same properties through a deterministic signature system.

$$\begin{aligned} \forall p_i : k_i &= \texttt{HMAC}(m, x_i) \\ k &= \prod_{i=1}^{i} k_s \bmod n \end{aligned} \tag{4.10}$$

The values k_i remain secret as well as the value x_i but the signature will always be the same for the given message and threshold key.

5 Implementation in Bitcoin-core secp256k1

As previously mentioned, Bitcoin uses elliptic curve cryptography (ECC) for signing transactions. When the first release of Bitcoin core appeared in early 2009, OpenSSL library was used to perform the cryptographic computations. Several years later, a project started with the goal of replacing OpenSSL and creating a custom and minimalistic `C` library for cryptography over the secp256k1 curve. This library is now available on GitHub at `bitcoin-core/secp256k1` and it is one of the most optimized libraries, if not the most, for the secp256k1 curve. It is worth noting that other significant crypto-currencies like Ethereum also use this library, so extending the capabilities of this library is an excellent choice to attract other cryptographers to have a look and increase the number of reviews for this thesis.

The implementation is spread into four main components (i) a `DER` parser-serializer, (ii) a textbook implementation of Paillier homomorphic cryptosystem, (iii) implementation of the Zero-Knowledge Proofs adaptation, and (iv) the threshold public API. It is worth noting that the current implementation is NOT production ready and NOT side-channel attack resistant. Paillier and ZKP are not constant time computation and use `libgmp` for all arithmetic computations, even when secret values are used. This implementation is a textbook implementation of the scheme and needs to be reviewed and tested more thoroughly before being used in production. It is also worth noting that this library does not implement the functions needed to initialize the setup. Only the functions needed to parse existing keys and compute a distributed signature are implemented.

This chapter refers to the implementation available on GitHub at `https://github.com/GuggerJoel/secp256k1/tree/threshold` at the time of writing. Note that the sources may evolve after writing this report, to be sure to read the latest version of the code, check out the sources directly on GitHub.

Contents

5.1 Configuration 37
5.1.1 Add new experimental module 37
5.1.2 Configure compilation 38
5.2 `DER` parser-serializer 39
5.2.1 Sequence 39
5.2.2 Integer 40
5.2.3 Octet string 41
5.3 Paillier cryptosystem 41
5.3.1 Data structures 41
5.3.2 Encrypt and decrypt 43
5.3.3 Homomorphism 44
5.4 Zero-knowledge proofs 45
5.4.1 Data structures 45
5.4.2 Generate proofs 46
5.4.3 Validate proofs 48

5.5 Threshold module 49
5.5.1 Create call message 49
5.5.2 Receive call message 50
5.5.3 Receive challenge message 51
5.5.4 Receive response challenge message 51
5.5.5 Receive terminate message 53

5.1 Configuration

The library uses `autotools` to manage the compilation, installation, and uninstallation. A system of modules is already present in the structure with an ECDH experimental module for shared secret computation and a recovery module for recovering ECDSA public keys. A module can be flagged as experimental, then, at configuration time, an explicit parameter enabling experimental modules must be passed, and a warning is shown to warn that the build contains experimental code.

5.1.1 Add new experimental module

In this structure, the threshold extension is all indicated to be an experimental module also. A new variable `$enable_module_recovery` is declared with an m4 macro defined by autoconf in the `configure.ac` file with the argument `--enable-module-threshold`. The default value is set to `no`.

```
137 AC_ARG_ENABLE(module_threshold,
138     AS_HELP_STRING([--enable-module-threshold],[enable Threshold ECDSA computation with
        ↪  Paillier homomorphic encryption system and zero-knowledge proofs (experimental)]),
139     [enable_module_threshold=$enableval],
140     [enable_module_threshold=no])
```

Listing 5.1 Add argument in `configure.ac` to enable the module

If the variable `$enable_module_recovery` is set to yes in `configure.ac` (lines 443 to 445), a compiler constant is declared, again with an m4 marco defined by autoconf, and set to 1 in `libsecp256k1-config.h` (lines 20 and 21). This header file is generated when `./configure` script runs and is included in the library.

```
443 if test x"$enable_module_threshold" = x"yes"; then
444   AC_DEFINE(ENABLE_MODULE_THRESHOLD, 1, [Define this symbol to enable the threshold module])
445 fi
```

```
20 /* Define this symbol to enable the threshold module */
21 #define ENABLE_MODULE_THRESHOLD 1
```

Listing 5.2 Define constant `ENABLE_MODULE_THRESHOLD` if module enable

The main file `secp256k1.c` (lines 586 to 590) and the tests file `tests.c` include headers based on the compiler constant definition.

```
586 #ifdef ENABLE_MODULE_THRESHOLD
587 # include "modules/threshold/paillier_impl.h"
588 # include "modules/threshold/eczkp_impl.h"
589 # include "modules/threshold/threshold_impl.h"
590 #endif
```

Listing 5.3 Include implementation headers if `ENABLE_MODULE_THRESHOLD` is defined

The module is set as experimental to avoid enabling it without explicitly agreeing to build experimental code. If the experimental parameter is set to yes a warning is displayed during the configuration process to warn the user. If the experimental parameter is not set and an experimental module is enabled an error message is displayed, and the process fails.

```
465 if test x"$enable_experimental" = x"yes"; then
466   AC_MSG_NOTICE([******])
467   AC_MSG_NOTICE([WARNING: experimental build])
468   AC_MSG_NOTICE([Experimental features do not have stable APIs or properties, and may not be
      ↪  safe for production use.])
469   AC_MSG_NOTICE([Building ECDH module: $enable_module_ecdh])
470   AC_MSG_NOTICE([Building Threshold module: $enable_module_threshold])
471   AC_MSG_NOTICE([******])
472 else
473   if test x"$enable_module_ecdh" = x"yes"; then
474     AC_MSG_ERROR([ECDH module is experimental. Use --enable-experimental to allow.])
475   fi
476   if test x"$enable_module_threshold" = x"yes"; then
477     AC_MSG_ERROR([Threshold module is experimental. Use --enable-experimental to allow.])
478   fi
479   if test x"$set_asm" = x"arm"; then
480     AC_MSG_ERROR([ARM assembly optimization is experimental. Use --enable-experimental to
      ↪  allow.])
481   fi
482 fi
```

Listing 5.4 Set threshold module to experimental in `configure.ac`

5.1.2 Configure compilation

A module is composed of one or many `include/` headers that contain the public API with a small description of each function, these headers are copied in the right folders when `sudo make install` command is run. The file `Makefile.am` defines which headers need to be installed, which don't and how to compile the project. This file is parsed by Autoconf to generate the final `Makefile`.

Each module has its `Makefile.am.include` which describes what to do with all the files in the module folder. This file is included in the main `Makefile.am` (lines 179 to 181) if the module is enabled.

```
179 if ENABLE_MODULE_THRESHOLD
180 include src/modules/threshold/Makefile.am.include
181 endif
```

Listing 5.5 Include specialized Makefile if threshold module is enabled

The specific `Makefile.am.include` declares the requisite header to be included and declares the list of all the headers that must not be installed on the system when `sudo make install` command is run.

```
1  include_HEADERS += include/secp256k1_threshold.h
2  noinst_HEADERS += src/modules/threshold/der_impl.h
3  noinst_HEADERS += src/modules/threshold/paillier.h
4  noinst_HEADERS += src/modules/threshold/paillier_impl.h
5  noinst_HEADERS += src/modules/threshold/paillier_tests.h
6  noinst_HEADERS += src/modules/threshold/eczkp.h
7  noinst_HEADERS += src/modules/threshold/eczkp_impl.h
8  noinst_HEADERS += src/modules/threshold/eczkp_tests.h
9  noinst_HEADERS += src/modules/threshold/threshold_impl.h
10 noinst_HEADERS += src/modules/threshold/threshold_tests.h
```

Listing 5.6 Specialized Makefile for threshold module

It is possible to build the library and enable the threshold module with the command below.

```
./configure --enable-module-threshold --enable-experimental
```

5.2 DER parser-serializer

Transmit messages and retrieve keys are an essential part of the scheme. Because between each step communication on the network is necessary, a way to export and import data is required. Bitcoin private keys are a simple structure because of the fixed curve and their intrinsic nature, a single 2^{256} bits value. Threshold private keys are composed of multiple parts, (i) the private share, (ii) a Paillier private key, (iii) a Paillier public key, and (iv) Zero-Knowledge Proof parameters. To serialize these complex structures the DER standard is chosen. Three simple data types are implemented in the library (i) sequence, (ii) integer, and (iii) octet string.

5.2.1 Sequence

The sequence data structure holds a sequence of integers and/or octet strings. The sequence starts with the constant 0x30 and is followed by the content length and the content itself. A length could be in the short form or the long form. If the content number of bytes is shorter than 0x80 the length byte indicates the length, if the content is equal or longer than 0x80 the seven lower bits 0 to 6 where byte $= \{b_7, \ldots, b_1, b_0\}$ indicate the number of following bytes which are used for the length.

```
10 void secp256k1_der_parse_len(const unsigned char *data, unsigned long *pos, unsigned long
    ↪  *lenght, unsigned long *offset) {
11     unsigned long op, i;
12     op = data[*pos] & 0x7F;
13     if ((data[*pos] & 0x80) == 0x80) {
14         for (i = 0; i < op; i++) {
15             *lenght += data[*pos+1+i]<<8*(op-i-1);
16         }
17         *offset = op + 1;
18     } else {
19         *lenght = op;
20         *offset = 1;
21     }
22     *pos += *offset;
23 }
```

Listing 5.7 Implementation of a DER length parser

The sequence parser checks the first byte with the constant 0x30 and extracts the content length. Positions in the input array are held in the *pos variable, extracted length is stored in *length, and the offset holds how many bytes of the data are used for the header and the length. A coherence check is performed to ensure that the current offset and the retrieved length result in the same number of bytes passed in argument.

When a sequence holds other sequences, retrieving their total length (including header and content length bytes) is needed to parse them recursively. A specific function is created to retrieve the total length of a struct given a pointer to its first byte.

```
25 int secp256k1_der_parse_struct(const unsigned char *data, size_t datalen, unsigned long *pos,
    ↪  unsigned long *lenght, unsigned long *offset) {
26     unsigned long loffset;
27     if (data[*pos] == 0x30) {
28         *pos += 1;
29         secp256k1_der_parse_len(data, pos, lenght, &loffset);
30         *offset = 1 + loffset;
31         if (*lenght + *offset != datalen) { return 0; }
32         else { return 1; }
33     }
34     return 0;
35 }
```

Listing 5.8 Implementation of a DER sequence parser

The serialization of a sequence is implemented as a serialization of an octet string with the sequence header `0x30` without an integrity check of the content. The content length is serialized first, then the header is added.

The result of content length serialization can be ≥ 1 bytes. If the content is shorter than `0x80`, then one byte is enough to store the length. Otherwise multiple bytes (≥ 2) are used. Because the number of bytes is undefined before the computation, a memory allocation is necessary and a pointer is returned with the length of the array.

```
155 unsigned char* secp256k1_der_serialize_sequence(size_t *outlen, const unsigned char *op,
    ↪  const size_t datalen) {
156     unsigned char *data = NULL, *len = NULL;
157     size_t lensize = 0;
158     len = secp256k1_der_serialize_len(&lensize, datalen);
159     *outlen = 1 + lensize + datalen;
160     data = malloc(*outlen * sizeof(unsigned char));
161     data[0] = 0x30;
162     memcpy(&data[1], len, lensize);
163     memcpy(&data[1 + lensize], op, datalen);
164     free(len);
165     return data;
166 }
```

Listing 5.9 Implementation of a DER sequence serializer

If the content length is longer than `0x80`, then `mpz` is used to serialize the length into a bytes array in big-endian most significant byte first. The length of this serialization is stored in `longsize` and is used to create the first byte with the most significant bit set to 1 (line 93).

5.2.2 Integer

Integers are used to store most values in the keys and Zero-Knowledge Proofs. An integer can be positive, negative or zero and are represented in the second complement form. The header starts with `0x02`, followed by the length of the data. Parsing and serializing integers is already implemented in `libgmp`, functions are just wrappers to extract information from the header and start the `mpz` importation at the right offset.

```
81  unsigned char* secp256k1_der_serialize_len(size_t *datalen, size_t lenght) {
82      unsigned char *data = NULL; void *serialize; size_t longsize; mpz_t len;
83      if (lenght >= 0x80) {
84          mpz_init_set_ui(len, lenght);
85          serialize = mpz_export(NULL, &longsize, 1, sizeof(unsigned char), 1, 0, len);
86          mpz_clear(len);
87          *datalen = longsize + 1;
88      } else {
89          *datalen = 1;
90      }
91      data = malloc(*datalen * sizeof(unsigned char));
92      if (lenght >= 0x80) {
93          data[0] = (uint8_t)longsize | 0x80;
94          memcpy(&data[1], serialize, longsize);
95          free(serialize);
96      } else {
97          data[0] = (uint8_t)lenght;
98      }
99      return data;
100 }
```

Listing 5.10 Implementation of a DER length serializer

5.2.3 Octet string

Octet strings are used to hold serialized data like points/public keys. An octet string is an arbitrary array of bytes. The header starts with `0x04` followed by the size of the content. The serialization implementation retrieves the length of the content, copies the header and the octet string into a new memory space, and returns the pointer with the total length. The parser implementation copies the content and sets the content length, the position index, and the offset.

5.3 Paillier cryptosystem

Homomorphic encryption is required in the scheme and Paillier is proposed in the white paper. Paillier homomorphic encryption is simple to implement in a textbook way, this implementation is functional but not optimized and needs to be reviewed.

5.3.1 Data structures

Encrypted messages, public keys and private keys are transmitted. As mentioned before, the `DER` standard format is used to parse and serialize data. A `der` schema for all data structures is defined to ensure portability over different implementations.

Public keys

The public key is composed of a public modulus and a generator. The implementation's data structure adds a big modulus corresponding to the square of the modulus. A version number is added for future compatibility purposes.

```
HEPublicKey ::= SEQUENCE {
    version           INTEGER,
    modulus           INTEGER,  -- p * q
    generator         INTEGER
}
```

Listing 5.11 DER schema of a Paillier public key

`libgmp` is used for all the arithmetic in the Paillier implementation, all numbers are stored in `mpz_t` type. The parser takes as input, an array of bytes with a length and the public key to create.

```
typedef struct {
    mpz_t modulus;
    mpz_t generator;
    mpz_t bigModulus;
} secp256k1_paillier_pubkey;

int secp256k1_paillier_pubkey_parse(
    secp256k1_paillier_pubkey *pubkey,
    const unsigned char *input,
    size_t inputlen
);
```

Listing 5.12 DER parser of a Paillier public key

Private keys

The private key is composed of a public modulus, two primes, a generator, a private exponent $\lambda = \varphi(n) = (p-1)(q-1)$, and a private coefficient $\mu = \varphi(n)^{-1} \mod n$. Again, a version number is added for future compatibility purposes.

```
HEPrivateKey ::= SEQUENCE {
    version           INTEGER,
    modulus           INTEGER,  -- p * q
    prime1            INTEGER,  -- p
    prime2            INTEGER,  -- q
    generator         INTEGER,
    privateExponent   INTEGER,  -- (p - 1) * (q - 1)
    coefficient       INTEGER   -- (inverse of privateExponent) mod (p * q)
}
```

Listing 5.13 DER schema of a Paillier private key

The parser takes as input, an array of bytes with a length and the private key to create. The big modulus is computed after parsing to accelerate encryption and decryption.

```
typedef struct {
    mpz_t modulus;
    mpz_t prime1;
    mpz_t prime2;
    mpz_t generator;
    mpz_t bigModulus;
    mpz_t privateExponent;
    mpz_t coefficient;
} secp256k1_paillier_privkey;

int secp256k1_paillier_privkey_parse(
    secp256k1_paillier_privkey *privkey,
    secp256k1_paillier_pubkey *pubkey,
    const unsigned char *input,
    size_t inputlen
);
```

Listing 5.14 DER parser of a Paillier private key

Encrypted messages

An encrypted message with Paillier cryptosystem is a big number $c \in \mathbb{Z}_{n^2}^*$. No version number is added in this case. The implementation's structure contains a nonce value which could be set to 0 to store the nonce used during encryption.

```
HEEncryptedMessage ::= SEQUENCE {
    message           INTEGER
}
```

Listing 5.15 *DER schema of an encrypted message with Paillier cryptosystem*

An encrypted message can be serialized and parsed and they are used in message exchange during the signing protocol by both parties.

5.3.2 Encrypt and decrypt

Like all other encryption schemes in public key cryptography, the public key is used to encrypt and the private key to decrypt. To encrypt the message `mpz_t m` where $m < n$, a random value r where $r < n$ is selected with the function pointer `noncefp` and set in the nonce value `res->nonce`. This nonce is stored because its value is needed to create Zero-Knowledge Proofs. Then, the cipher $c = g^m \cdot r^n \mod n^2$ is put in `res->message` to complete the encryption process. All intermediary states are erased before returning the result.

```
int secp256k1_paillier_encrypt_mpz(secp256k1_paillier_encrypted_message *res, const mpz_t m,
↪   const secp256k1_paillier_pubkey *pubkey, const secp256k1_paillier_nonce_function
↪   noncefp) {
    mpz_t l1, l2, l3;
    int ret = noncefp(res->nonce, pubkey->modulus);
    if (ret) {
        mpz_inits(l1, l2, l3, NULL);
        mpz_powm(l1, pubkey->generator, m, pubkey->bigModulus);
        mpz_powm(l2, res->nonce, pubkey->modulus, pubkey->bigModulus);
        mpz_mul(l3, l1, l2);
        mpz_mod(res->message, l3, pubkey->bigModulus);
        mpz_clears(l1, l2, l3, NULL);
    }
    return ret;
}
```

Listing 5.16 *Implementation of encryption with Paillier cryptosystem*

If the random value selection process fails the encryption also fails. The random function of type `secp256k1_paillier_nonce_function` must use a good CPRNG and its implementation is not part of the library.

```
typedef int (*secp256k1_paillier_nonce_function)(
    mpz_t nonce,
    const mpz_t max
);
```

Listing 5.17 *Function signature for Paillier nonce generation*

To decrypt the cipher $c \in \mathbb{Z}_{n^2}^*$ with the private key, the function computes $m = L(c^\lambda \mod n^2) \cdot \mu \mod n$ where $L(x) = (x-1)/n$. The cipher is raised to the lambda $c^\lambda \mod n^2$ in line 4 and the result is put in an intermediary state variable. Then the $L(x)$ function is applied to the intermediary state in lines 5-6. Finally, the multiplication with μ and the modulo of n are taken (lines 7-8) to lead to the result. It is worth noting that, in line 6, only the quotient of the division is recovered.

```
1  void secp256k1_paillier_decrypt(mpz_t res, const secp256k1_paillier_encrypted_message *c,
   ↪  const secp256k1_paillier_privkey *privkey) {
2      mpz_t l1, l2;
3      mpz_inits(l1, l2, NULL);
4      mpz_powm(l1, c->message, privkey->privateExponent, privkey->bigModulus);
5      mpz_sub_ui(l2, l1, 1);
6      mpz_cdiv_q(l1, l2, privkey->modulus);
7      mpz_mul(l2, l1, privkey->coefficient);
8      mpz_mod(res, l2, privkey->modulus);
9      mpz_clears(l1, l2, NULL);
10 }
```

Listing 5.18 Implementation of decryption with Paillier cryptosystem

5.3.3 Homomorphism

The choice of this scheme is not hazardous, homomorphic addition and multiplication are used to construct the signature $s = D_{sk}(\mu) \mod q : \mu = (\alpha \times_{pk} m'z_2) +_{pk} (\zeta \times_{pk} r'x_2z_2) +_{pk} E_{pk}(cn)$ where $+_{pk}$ denotes homomorphic addition over the ciphertexts and $\times_{pk}$ denotes homomorphic multiplication over the ciphertexts.

Addition

Addition $+_{pk}$ over ciphertexts is computed with $D_{sk}(E_{pk}(m_1, r_1) \cdot E_{pk}(m_2, r_2) \mod n^2) = m_1 + m_2 \mod n$ or $D_{sk}(E_{pk}(m_1, r_1) \cdot g^{m_2} \mod n^2) = m_1 + m_2 \mod n$ where D_{sk} denotes decryption with private key sk and E_{pk} denotes encryption with public key pk. Only the first variant is implemented, where two ciphertexts are added together to result in a third ciphertext.

```
void secp256k1_paillier_add(secp256k1_paillier_encrypted_message *res, const
↪  secp256k1_paillier_encrypted_message *op1, const secp256k1_paillier_encrypted_message
↪  *op2, const secp256k1_paillier_pubkey *pubkey) {
    mpz_t l1;
    mpz_init(l1);
    mpz_mul(l1, op1->message, op2->message);
    mpz_mod(res->message, l1, pubkey->bigModulus);
    mpz_clear(l1);
}
```

Listing 5.19 Implementation of homomorphic addition with Paillier cryptosystem

Multiplication

Multiplication $\times_{pk}$ over ciphertexts can be performed with $D_{sk}(E_{pk}(m_1, r_1)^{m_2} \mod n^2) = m_1 m_2 \mod n$, the implementation is straight forward in this case. The nonce value from the ciphertext is copied in the resulting encrypted message to not lose information after opperations.

```
void secp256k1_paillier_mult(secp256k1_paillier_encrypted_message *res, const
 ↪  secp256k1_paillier_encrypted_message *c, const mpz_t s, const secp256k1_paillier_pubkey
 ↪  *pubkey) {
    mpz_powm(res->message, c->message, s, pubkey->bigModulus);
    mpz_set(res->nonce, c->nonce);
}
```

Listing 5.20 Implementation of homomorphic multiplication with Paillier cryptosystem

5.4 Zero-knowledge proofs

Two Zero-Knowledge Proofs are used in the scheme, each party generates a proof and validates the other one. A proof is generated and verified under some ZKP parameters, these parameters are fixed at initialization time and don't change over time.

5.4.1 Data structures

Three data structures are created, one for each ZKP and one for storing the parameters. Zero-Knowledge Proofs are composed of big numbers and points, and need to be serialized and parsed to be included in the message exchange protocol.

Zero-Knowledge Parameters

Zero-Knowledge parameter is composed of three numeric values (i) $\tilde{N}$ a public modulus, (ii) h_2 a value selected randomly $\in \mathbb{Z}^*_{\tilde{N}}$, and (iii) h_1 a value where $\exists x, \log_x(h_1) = h_2 \mod \tilde{N}$. One function is provided in the module to parse a `ZKPParameter DER` schema.

```
ZKPParameter ::= SEQUENCE {
    modulus             INTEGER,
    h1                  INTEGER,
    h2                  INTEGER
}
```

Listing 5.21 `DER` schema of a Zero-Knowledge parameters sequence

Zero-Knowledge Proof Π

Zero-Knowledge Proof Π is composed of numeric values and one point. The point is stored in a public key internal structure within the implementation and is exported through the secp256k1 library as a 65 byte uncompressed public key. The uncompressed public key is then stored as an octet string in the schema. A version number is added for future compatibility purposes. Two functions are provided in the module to parse and serialize a `ECZKPPi DER` schema.

Zero-Knowledge Proof Π'

Zero-Knowledge Proof Π' is composed of the same named values as ZKP Π plus five new ones. The construction of the proof is based on Π but needs more equations to express all the proven statements. Again, the point y is a point serialized as

```
ECZKPPi ::= SEQUENCE {
    version             INTEGER,
    z1                  INTEGER,
    z2                  INTEGER,
    y                   OCTET STRING,
    e                   INTEGER,
    s1                  INTEGER,
    s2                  INTEGER,
    s3                  INTEGER,
    t1                  INTEGER,
    t2                  INTEGER,
    t3                  INTEGER,
    t4                  INTEGER
}
```

Listing 5.22 *DER schema of a Zero-Knowledge* Π *sequence*

an uncompressed public key in an octet string and a version number is added for future compatibility purposes. Two functions are provided in the module to parse and serialize a `ECZKPPiPrim` `DER` schema.

```
ECZKPPiPrim ::= SEQUENCE {
    version             INTEGER,
    z1                  INTEGER,
    z2                  INTEGER,
    z3                  INTEGER,
    y                   OCTET STRING,
    e                   INTEGER,
    s1                  INTEGER,
    s2                  INTEGER,
    s3                  INTEGER,
    s4                  INTEGER,
    t1                  INTEGER,
    t2                  INTEGER,
    t3                  INTEGER,
    t4                  INTEGER,
    t5                  INTEGER,
    t6                  INTEGER,
    t7                  INTEGER
}
```

Listing 5.23 *DER schema of a Zero-Knowledge* Π' *sequence*

5.4.2 Generate proofs

Proofs are generated in relation to a specific setup and a specific in progress signature, which makes them linked to a large number of values (points, encrypted messages, secrets, parameters, etc.) The complexity of these constructions is strongly felt in the code. Heavy mathematical computations are needed with two `hash` functions.

A CPRNG function is required to generate both proofs. This function generates random numbers in $\mathbb{Z}_{max}$ and $\mathbb{Z}^*_{max}$. The `flag` argument indicate which case is treated, `STD` or `INV`. If the function has no access to a good source of randomness or cannot generate a good random number a zero is returned, otherwise a one is returned.

```
typedef int (*secp256k1_eczkp_rdn_function)(
    mpz_t res,
    const mpz_t max,
    const int flag
);

#define SECP256K1_THRESHOLD_RND_INV 0x01
#define SECP256K1_THRESHOLD_RND_STD 0x00
```

Listing 5.24 Function signature for ZKP CPRNG

Zero-Knowledge Proof Π

As shown in figure 4.2, the proof states that (i) there exists a known value by the proover that links $r \to r_2$, (ii) there exists a second known value by the proover that, related to the first one, links $G \to y_1$, (iii) the result of $D_{sk}(\alpha)$ is this first value, and (iv) the result of $D_{sk}(\zeta)$ is this second value.

To do computation on the curve a context object needs to be passed in arguments, then the ZKP object to fill, the ZKP parameters, the two encrypted messages α and ζ, scalar values sx_1 and sx_2 representing $z_1 = (k_1)^{-1} \mod n$ and $x_1 z_1$, then the point r, the point r_2, the partial public key y_1, the proover Paillier public key which has been used to encrypt α and ζ, and finally a pointer to a CPRNG function used to generate all needed random values.

```
int secp256k1_eczkp_pi_generate(
    const secp256k1_context *ctx,
    secp256k1_eczkp_pi *pi,
    const secp256k1_eczkp_parameter *zkp,
    const secp256k1_paillier_encrypted_message *m1,
    const secp256k1_paillier_encrypted_message *m2,
    const secp256k1_scalar *sx1,
    const secp256k1_scalar *sx2,
    const secp256k1_pubkey *c,
    const secp256k1_pubkey *w1,
    const secp256k1_pubkey *w2,
    const secp256k1_paillier_pubkey *pubkey,
    const secp256k1_eczkp_rdn_function rdnfp
);
```

Listing 5.25 Function signature to generate ZKP Π

The function implementation can be split in four main parts (i) generate all the needed random values v, (ii) compute the challenge values, (iii) compute the `hash` of these values v, and (iv) compute the ZKP values with $e = \texttt{hash}(v)$.

Zero-Knowledge Proof Π′

As shown in figure 4.5, the proof states that (i) there exists a known value by the proover x_1 that link $r_2 \to G$, (ii) there exists a second known value by the proover that, related to the first one, link $G \to y_2$, (iii) the result of $D_{sk'}(\mu')$ is this first value, and (iv) there exists a third known value by the proover x_3 and the result of $D_{sk}(\mu)$ is the homomorphic operation of $(\alpha \times x_1) + (\zeta \times x_2) + x_3$.

The function implementation can also be split in four main parts (i) generate all the needed random values v, (ii) compute the proof values, (iii) compute the `hash'` of these values v, and (iv) compute the ZKP values with $e = \texttt{hash'}(v)$.

```
int secp256k1_eczkp_pi2_generate(
    const secp256k1_context *ctx,
    secp256k1_eczkp_pi2 *pi2,
    const secp256k1_eczkp_parameter *zkp,
    const secp256k1_paillier_encrypted_message *m1,
    const secp256k1_paillier_encrypted_message *m2,
    const secp256k1_paillier_encrypted_message *m3,
    const secp256k1_paillier_encrypted_message *m4,
    const secp256k1_paillier_encrypted_message *r,
    const mpz_t x1,
    const mpz_t x2,
    const mpz_t x3,
    const mpz_t x4,
    const mpz_t x5,
    const secp256k1_pubkey *c,
    const secp256k1_pubkey *w2,
    const secp256k1_paillier_pubkey *pairedkey,
    const secp256k1_paillier_pubkey *pubkey,
    const secp256k1_eczkp_rdn_function rdnfp
);
```

Listing 5.26 Function signature to generate ZKP Π'

It is worth noting that `hash` and `hash'` must be different hashing functions to avoid reusing Π proofs, even without satisfying the predicate, to construct fraudulent Π' proofs.

5.4.3 Validate proofs

Validation of proofs Π and Π' can be done with (i) the Paillier public keys, (ii) the ZKP parameters, and (iii) the exchanged messages. The process can be split into three steps: compute the proof values, retrieve the candidate value e', and compare if $e = e'$. If the values match the proof is valid.

```
int secp256k1_eczkp_pi_verify(
    const secp256k1_context *ctx,
    secp256k1_eczkp_pi *pi,
    const secp256k1_eczkp_parameter *zkp,
    const secp256k1_paillier_encrypted_message *m1,
    const secp256k1_paillier_encrypted_message *m2,
    const secp256k1_pubkey *c,
    const secp256k1_pubkey *w1,
    const secp256k1_pubkey *w2,
    const secp256k1_paillier_pubkey *pubkey
);

int secp256k1_eczkp_pi2_verify(
    const secp256k1_context *ctx,
    secp256k1_eczkp_pi2 *pi2,
    const secp256k1_eczkp_parameter *zkp,
    const secp256k1_paillier_encrypted_message *m1,
    const secp256k1_paillier_encrypted_message *m2,
    const secp256k1_paillier_encrypted_message *m3,
    const secp256k1_paillier_encrypted_message *m4,
    const secp256k1_pubkey *c,
    const secp256k1_pubkey *w2,
    const secp256k1_paillier_pubkey *pubkey,
    const secp256k1_paillier_pubkey *pairedkey
);
```

Listing 5.27 Function signature to validate ZKP Π *and* Π'

5.5 Threshold module

The threshold module exposes the public API used to create an application that wants to use the distributed signature protocol. The public API includes all functions needed to parse-serialize keys, messages, and signature parameters. Signature parameters hold the values k, $z = k^{-1}$, and $r = k \cdot G$, these values are—in a normal signature mode—computed, used, and destroyed in one go. However, a mechanism to save and restore these values is required in distributed mode because the context can be destroyed and re-created between each step.

The public API also includes the five functions that implement the protocol. One function is one step in the protocol and in-between functions, the generated message is serialized by the caller and parsed by the sender. The signature parameters could also be serialized and parsed during the response waiting time.

Nomenclature

A proposal for exchanged message names and actions is done in this report. Players P_1 and P_2 represent the initiator and collaborator. Player P_1 initializes the communication and asks P_2 to collaborate on a signature, if P_2 collaborates and the protocol ends successfully P_1 retrieves the signature.

Four messages are necessary between the five steps. In order, the proposed names are (i) call message, (ii) challenge message, (iii) response challenge, and (iv) terminate message. The functions are named after the corresponding action and message name.

5.5.1 Create call message

The `call_create` function, as indicated by its name, creates the call message. Arguments are checked to be non-null. If one of them is the function will fail. The secret share is loaded in a 32 byte array and the nonce (k) is retrieved with the `noncefp` function pointer. It is worth noting that this function could be called multiple times until a nonce that is not zero and which doesn't overflow is found. However, this function has a limited number of calls and if the limit is reached the function will fail. The signature parameters are then set and encrypted in the call message. The parameters k and z are set for P_1. The `noncefp` can point to an implementation of a deterministic signature mode or a random [20] signature mode. If the deterministic model is chosen, the counter indicates the number of rounds done by the function.

```
247 int secp256k1_threshold_call_create(const secp256k1_context *ctx,
    ↪  secp256k1_threshold_call_msg *callmsg, secp256k1_threshold_signature_params *params,
    ↪  const secp256k1_scalar *secshare, const secp256k1_paillier_pubkey *paillierkey, const
    ↪  unsigned char *msg32, const secp256k1_nonce_function noncefp, const
    ↪  secp256k1_paillier_nonce_function pnoncefp) {
248     int ret = 0;
249     int overflow = 0;
250     unsigned char nonce32[32];
251     unsigned char sec32[32];
252     unsigned int count = 0;
253     secp256k1_scalar privinv;
254
255     ARG_CHECK(ctx != NULL);
256     ARG_CHECK(callmsg != NULL);
257     ARG_CHECK(params != NULL);
258     ARG_CHECK(secshare != NULL);
259     ARG_CHECK(paillierkey != NULL);
260     ARG_CHECK(msg32 != NULL);
```

```
261     secp256k1_scalar_get_b32(sec32, secshare);
262     while (1) {
263         ret = noncefp(nonce32, msg32, sec32, NULL, NULL, count);
264         if (!ret) {
265             break;
266         }
267         secp256k1_scalar_set_b32(&params->k, nonce32, &overflow);
268         if (!overflow && !secp256k1_scalar_is_zero(&params->k)) {
269             secp256k1_scalar_inverse(&params->z, &params->k); /* z1 */
270             secp256k1_scalar_mul(&privinv, &params->z, secshare); /* x1z1 */
271             if (secp256k1_paillier_encrypt_scalar(callmsg->alpha, &params->z, paillierkey,
                 ↪  pnoncefp)
272                 && secp256k1_paillier_encrypt_scalar(callmsg->zeta, &privinv, paillierkey,
                     ↪  pnoncefp)) {
273                 break;
274             }
275         }
276         count++;
277     }
278     memset(nonce32, 0, 32);
279     memset(sec32, 0, 32);
280     secp256k1_scalar_clear(&privinv);
281     return ret;
282 }
```

Listing 5.28 Implementation of `call_create` function

5.5.2 Receive call message

The `call_received` function sets the parameter k and r of P_2 and prepares the challenge message with r. Again, the pointer can point to a deterministic implementation for generating the nonce.

```
284 int secp256k1_threshold_call_received(const secp256k1_context *ctx,
    ↪  secp256k1_threshold_challenge_msg *challengemsg, secp256k1_threshold_signature_params
    ↪  *params, const secp256k1_threshold_call_msg *callmsg, const secp256k1_scalar *secshare,
    ↪  const unsigned char *msg32, const secp256k1_nonce_function noncefp) {
285     int ret = 0;
286     int overflow = 0;
287     unsigned int count = 0;
288     unsigned char k32[32];
289     unsigned char sec32[32];
290
291     ARG_CHECK(ctx != NULL);
292     ARG_CHECK(challengemsg != NULL);
293     ARG_CHECK(params != NULL);
294     ARG_CHECK(callmsg != NULL);
295     ARG_CHECK(secshare != NULL);
296     ARG_CHECK(msg32 != NULL);
297     secp256k1_scalar_get_b32(sec32, secshare);
298     while (1) {
299         ret = noncefp(k32, msg32, sec32, NULL, NULL, count);
300         if (!ret) {
301             break;
302         }
303         secp256k1_scalar_set_b32(&params->k, k32, &overflow);
304         if (!overflow && !secp256k1_scalar_is_zero(&params->k)) {
305             if (secp256k1_ec_pubkey_create(ctx, &params->r, k32)) {
306                 memcpy(&challengemsg->r2, &params->r, sizeof(secp256k1_pubkey));
307                 break;
308             }
309         }
310         count++;
311     }
312     memset(k32, 0, 32);
313     memset(sec32, 0, 32);
```

```
314     return ret;
315 }
```

Listing 5.29 Implementation of `call_received` function

5.5.3 Receive challenge message

The `challenge_received` function is called by P_1 to compute the final public point r of the signature and create the first Zero-Knowledge Proof.

```
317 int secp256k1_threshold_challenge_received(const secp256k1_context *ctx,
    ↪  secp256k1_threshold_response_challenge_msg *respmsg,
    ↪  secp256k1_threshold_signature_params *params, const secp256k1_scalar *secshare, const
    ↪  secp256k1_threshold_challenge_msg *challengemsg, const secp256k1_threshold_call_msg
    ↪  *callmsg, const secp256k1_eczkp_parameter *zkp, const secp256k1_paillier_pubkey
    ↪  *paillierkey, const secp256k1_eczkp_rdn_function rdnfp) {
318     int ret = 0;
319     unsigned char k32[32];
320     secp256k1_pubkey y1;
321     secp256k1_scalar privinv;
322
323     ARG_CHECK(ctx != NULL);
324     ARG_CHECK(respmsg != NULL);
325     ARG_CHECK(params != NULL);
326     ARG_CHECK(challengemsg != NULL);
327     secp256k1_scalar_get_b32(k32, &params->k);
328     memcpy(&respmsg->r, &challengemsg->r2, sizeof(secp256k1_pubkey));
329     ret = secp256k1_ec_pubkey_tweak_mul(ctx, &respmsg->r, k32);
330     secp256k1_scalar_get_b32(k32, secshare);
331     if (ret && secp256k1_ec_pubkey_create(ctx, &y1, k32)) {
332         memcpy(&params->r, &respmsg->r, sizeof(secp256k1_pubkey));
333         secp256k1_scalar_mul(&privinv, &params->z, secshare);
334         VERIFY_CHECK(secp256k1_eczkp_pi_generate(
335             ctx,
336             respmsg->pi,
337             zkp,
338             callmsg->alpha,
339             callmsg->zeta,
340             &params->z,
341             &privinv,
342             &params->r,
343             &challengemsg->r2,
344             &y1,
345             paillierkey,
346             rdnfp
347         ) == 1);
348     }
349     memset(k32, 0, 32);
350     secp256k1_scalar_clear(&privinv);
351     return ret;
352 }
```

Listing 5.30 Implementation of `challenge_received` function

5.5.4 Receive response challenge message

The `response_challenge_received` function is called by P_2 and validates the first Zero-Knowledge Proof, Π. The final ciphertext which contains the s part of the distributed signature is computed and the second Zero-Knowledge Proof Π' is created.

The point r is normalized and the coordinate $r.x$ is obtained (modulo n). The `hash` is multiplied with z_2 and the coordinate $r.x$ is multiplied with x_2z_2. A value x_3 where $n|x_3$ is added to the cipher to hide information about the secret share and the secret random. In ECDSA $s = k^{-1}(m + rx) \mod n$, so the ciphertext matches the requirement as demonstrated below:

$$\begin{aligned} D_{sk}(\mu) &\equiv (\alpha \times mz_2) + (\zeta \times rx_2z_2) + (x_3) \pmod n \\ &\equiv (z_1 \times mz_2) + (x_1z_1 \times rx_2z_2) \pmod n \\ &\equiv (z_1z_2m) + (x_1z_1rx_2z_2) \pmod n \\ &\equiv z_1z_2(m + rx_1x_2) \pmod n \\ &\equiv z(m + rx) \pmod n \\ &\equiv k^{-1}(m + rx) \pmod n \end{aligned}$$

```
379     ret = secp256k1_eczkp_pi_verify(
380         ctx,
381         respmsg->pi,
382         zkp,
383         callmsg->alpha,
384         callmsg->zeta,
385         &respmsg->r,
386         &challengemsg->r2,
387         pairedshare,
388         pairedkey
389     );
390     if (ret) {
391         mpz_inits(m1, m2, c, n5, n, nc, m, z, rsig, inv, NULL);
392         secp256k1_scalar_inverse(&params->z, &params->k); /* z2 */
393         secp256k1_scalar_mul(&privinv, &params->z, secshare); /* x2z2 */
394         mpz_import(n, 32, 1, sizeof(n32[0]), 1, 0, n32);
395         secp256k1_scalar_set_b32(&msg, msg32, &overflow);
396         if (!overflow && !secp256k1_scalar_is_zero(&msg)) {
397             secp256k1_pubkey_load(ctx, &r, &respmsg->r);
398             secp256k1_fe_normalize(&r.x);
399             secp256k1_fe_normalize(&r.y);
400             secp256k1_fe_get_b32(b, &r.x);
401             secp256k1_scalar_set_b32(&sigr, b, &overflow);
402             /* These two conditions should be checked before calling */
403             VERIFY_CHECK(!secp256k1_scalar_is_zero(&sigr));
404             VERIFY_CHECK(overflow == 0);
405             mpz_import(rsig, 32, 1, sizeof(b[0]), 1, 0, b);
406             secp256k1_scalar_get_b32(b, &params->z);
407             mpz_import(z, 32, 1, sizeof(b[0]), 1, 0, b);
408             secp256k1_scalar_get_b32(b, &privinv);
409             mpz_import(inv, 32, 1, sizeof(b[0]), 1, 0, b);
410             secp256k1_scalar_get_b32(b, &msg);
411             mpz_import(m, 32, 1, sizeof(msg32[0]), 1, 0, msg32);
412             mpz_mul(m1, m, z); /* m'z2 */
413             mpz_mul(m2, rsig, inv); /* r'x2z2 */
414             mpz_pow_ui(n5, n, 5);
415             noncefp(c, n5);
416             mpz_mul(nc, c, n); /* cn */
417             secp256k1_paillier_mult(m3, callmsg->alpha, m1, pairedkey);
418             secp256k1_paillier_mult(m4, callmsg->zeta, m2, pairedkey);
419             secp256k1_paillier_add(m5, m3, m4, pairedkey);
420             ret = secp256k1_paillier_encrypt_mpz(enc, nc, pairedkey, noncefp);
421             secp256k1_scalar_get_b32(sec32, secshare);
422             if (ret && secp256k1_ec_pubkey_create(ctx, &y2, sec32)) {
423                 secp256k1_paillier_add(termsg->mu, m5, enc, pairedkey);
424                 ret = secp256k1_paillier_encrypt_mpz(termsg->mu2, z, p2, noncefp);
425                 VERIFY_CHECK(secp256k1_eczkp_pi2_generate(
426                     ctx,                    /* ctx */
427                     termsg->pi2,            /* pi2 */
```

```
428                     zkp,                    /* zkp */
429                     termsg->mu2,            /* m1 */
430                     termsg->mu,             /* m2 */
431                     callmsg->alpha,         /* m3 */
432                     callmsg->zeta,          /* m4 */
433                     enc,                    /* r */
434                     z,                      /* x1 */
435                     inv,                    /* x2 */
436                     c,                      /* x3 */
437                     m,                      /* x4 */
438                     rsig,                   /* x5 */
439                     &challengemsg->r2,      /* c */
440                     &y2,                    /* w2 */
441                     pairedkey,              /* pairedkey */
442                     p2,                     /* pubkey */
443                     rdnfp                   /* rdnfp */
444                 ) == 1);
445             }
446         }
```

Listing 5.31 *Core function of* `response_challenge_received`

5.5.5 Receive terminate message

The `terminate_received` function is called by P_1 and validates the second Zero-Knowledge Proof, Π'. After validation of the proof, the ciphertext is decrypted and the signature is composed. The signature is then tested and the protocol ends. Only P_1 can decrypt the ciphertext so the protocol is asymmetric. If P_2 also needs the signature, P_1 must share it. There is no way for P_2 to know the signature without a cooperative P_1.

```
460     unsigned char n32[32] = {
461         0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
            ↪  0xff, 0xfe,
462         0xba, 0xae, 0xdc, 0xe6, 0xaf, 0x48, 0xa0, 0x3b, 0xbf, 0xd2, 0x5e, 0x8c, 0xd0, 0x36,
            ↪  0x41, 0x41
463     };
464     unsigned char b[32];
465     void *ser;
466     int ret = 0;
467     int overflow = 0;
468     size_t size;
469     mpz_t m, n, sigs;
470     secp256k1_ge sigr, pge;
471     secp256k1_paillier_pubkey *p1;
472     secp256k1_scalar r, s, mes;
473
474     ARG_CHECK(ctx != NULL);
475     ARG_CHECK(sig != NULL);
476     ARG_CHECK(termsg != NULL);
477     ARG_CHECK(params != NULL);
478     ARG_CHECK(p != NULL);
479     ARG_CHECK(pub != NULL);
480     ARG_CHECK(msg32 != NULL);
481     p1 = secp256k1_paillier_pubkey_get(p);
482     ret = secp256k1_eczkp_pi2_verify(
483         ctx,                    /* ctx */
484         termsg->pi2,            /* pi2 */
485         zkp,                    /* zkp */
486         termsg->mu2,            /* m1 */
487         termsg->mu,             /* m2 */
488         callmsg->alpha,         /* m3 */
489         callmsg->zeta,          /* m4 */
490         &challengemsg->r2,      /* c */
491         pairedpub,              /* w2 */
```

```
492         p1,                     /* pubkey */
493         pairedkey               /* pairedkey */
494     );
495     if (ret) {
496         secp256k1_scalar_set_b32(&mes, msg32, &overflow);
497         ret = !overflow && secp256k1_pubkey_load(ctx, &pge, pub);
498         if (ret) {
499             secp256k1_pubkey_load(ctx, &sigr, &params->r);
500             secp256k1_fe_normalize(&sigr.x);
501             secp256k1_fe_normalize(&sigr.y);
502             secp256k1_fe_get_b32(b, &sigr.x);
503             secp256k1_scalar_set_b32(&r, b, &overflow);
504             VERIFY_CHECK(!secp256k1_scalar_is_zero(&r));
505             VERIFY_CHECK(overflow == 0);
506             mpz_inits(m, n, sigs, NULL);
507             secp256k1_paillier_decrypt(m, termsg->mu, p);
508             mpz_import(n, 32, 1, sizeof(n32[0]), 1, 0, n32);
509             mpz_mod(sigs, m, n);
510             ser = mpz_export(NULL, &size, 1, sizeof(unsigned char), 1, 0, sigs);
511             secp256k1_scalar_set_b32(&s, ser, &overflow);
512             if (!overflow
513                 && !secp256k1_scalar_is_zero(&s)
514                 && secp256k1_ecdsa_sig_verify(&ctx->ecmult_ctx, &r, &s, &pge, &mes)) {
515                 secp256k1_ecdsa_signature_save(sig, &r, &s);
516             } else {
517                 memset(sig, 0, sizeof(*sig));
518             }
519         }
520         mpz_clears(m, n, sigs, NULL);
521         secp256k1_scalar_clear(&r);
522         secp256k1_scalar_clear(&s);
523         secp256k1_scalar_clear(&mes);
524     }
525     secp256k1_paillier_pubkey_destroy(p1);
526     return ret;
```

Listing 5.32 *Core function of* `terminate_received`

6 Further research

It is possible to list an enormous number of ideas for further research in a field like crypto-currencies or blockchain. But some of those more related to the work done in this paper are listed in the following. Some of them are improvements of the work already done, but not yet ready for production, and some of them are entirely exploratory.

6.1 Side-channel attack resistant implementation and improvements

The proposed implementation in the library `secp256k1` relies upon `libgmp` for all complicated mathematical calculus and `libgmp` is not robust against side-channel attacks, and this is normal, the library has not been developed for that particular purpose. Therefore, another implementation is needed to handle, in constant time and constant memory if possible, the mathematical calculus part. This is a significant improvement that can be done, or must be done, before hoping to use the module is some real case scenario.

6.1.1 Second hash function

The current implementation uses the hash function `SHA256` implemented in the library `secp256k1` for Π and Π'. This is not compliant with the orignial papers requirements, another hash function must be implemented and used for Π'.

6.1.2 Paillier cryptosystem

Two major improvements or modifications can be made specificaly on the Paillier cryptosystem implementation. As shown in the original paper, the Chinese Remainder Theorem can be used to optimize the decryption. In the standard approach, with a private key (n, g, λ, μ) and a ciphertext $c \in \mathbb{Z}_{n^2}^*$ it is possible to compute the plaintext $m = L(c^\lambda \mod n^2) \cdot \mu \mod n$ where $L(x) = \frac{x-1}{n}$. With the CRT two functions L_p and L_q are defined as

$$L_p(x) = \frac{x-1}{p} \quad \text{and} \quad L_q(x) = \frac{x-1}{q}$$

Decryption can, therefore, be performed with modulo p and modulo q and recombining modular residues afterward:

$$\begin{aligned} m_p &= L_p(c^{p-1} \mod p^2)\ h_p \mod p \\ m_q &= L_q(c^{q-1} \mod p^2)\ h_q \mod q \\ m &= \mathrm{CRT}(m_p, m_q) \mod pq \end{aligned}$$

with precomputations

$$\begin{aligned} h_p &= L_p(g^{p-1} \mod p^2)^{-1} \mod p \quad \text{and} \\ h_q &= L_q(g^{q-1} \mod p^2)^{-1} \mod q \end{aligned}$$

Paillier cryptosystem can be adapted to EC cryptography as shown in the paper "Trapdooring Discrete Logarithms on Elliptic Curves over Rings" by Pascal Paillier [21]. It is worth noting however that the curve construction is different from the curve used to sign and so the code base cannot necessarily be reused.

6.1.3 Zero-knowledge proofs

Non-interactive zero-knowledge proofs are a significant research field. The article "From Extractable Collision Resistance to Succinct Non-interactive Arguments of Knowledge, and Back Again" by Bitansky, Nir and Canetti, Ran and Chiesa, Alessandro and Tromer, and Eran [22] introduced the acronym zk-SNARK for zero-knowledge Succinct Non-interactive ARgument of Knowledge that is the backbone of the Zcash protocol [23]. In the recent paper "Bulletproofs: Efficient Range Proofs for Confidential Transactions" [24] a new non-interactive zero-knowledge proof protocol with concise proofs and without a trusted setup is proposed. Further research could be done to adapt the zero-knowledge proof construction and migrate to a more generic approach, keeping in mind that the zero-knowledge proof construction proposed in the original paper dates from the early 2000s and advancement has been made since then.

6.2 Hardware wallets

Hardware wallet devices have become increasingly popular. They promise to keep the keys safe and, at least, expose the keys less thanks to a dedicated and controlled environment. Thus, keys can be stored safely and, in an organization, for example, multiple hardware wallets can be used to create a multi-signature to control the funds.

The development of this threshold library, even if it is just a 2-out-of-2 multi-signature script equivalent, can be used to create threshold hardware wallet devices. Two hardware wallet devices can be set up together to create a multi-user setup, or a hardware wallet device can be coupled with a phone to secure a web-wallet.

Usually when a new Bitcoin wallet is created a list of words, called a mnemonic phrase, is shown to the user as a backup of his wallet key. The mnemonics are between twelve and twenty four words, and each word represents 11 bits of the initial seed [25]. For a threshold key, it is not possible to represent all the data in the same way given the size of the key (near 4.5 Kb). Another way to display and transmit this information is needed to improve usability. Further research could be done to find a better way to represent and display a threshold key.

The master tag is not included in the `DER` schema. Is the key itself responsible for storing this information or is this information part of the setup and can be stored elsewhere? This question can be explored.

6.3 General threshold scheme

The way multi-signature scripts work in Bitcoin requires exposing all public keys related to the signatures. That increases the transaction size, which implies more significant fees. Due to the script size limit of around 500 bytes, the maximum number of signatories is fifteen. The signatures are naturally present with the public keys in the script, which implies that it is possible to know which keys signed the transaction. That implies less anonymity on the blockchain. With a general threshold scheme, these limitations would be removed.

As previously mentioned, research has been done to generalize and find an optimal (t, n)-threshold in ECDSA [12, 13]. These papers base their work on the scheme chosen in this thesis, so a deeper analysis could be performed to assess the changes needed and adapt the current implementation to construct a generic threshold scheme.

6.4 Schnorr signatures

In the paper "Efficient Identification and Signatures for Smart Cards" published in CRYPTO 1989, C.P. Schnorr proposes the "Schnorr signature algorithm" [26]. The Schnorr signature is considered the simplest digital signature scheme to be provably secure in a random oracle model [27, 28]. Thus, Bitcoin developers and researchers have had a strong interest in this specific scheme for some years now. Schnorr signatures could greatly reduce the size of the signature from 65 bytes (ECDSA in `DER` format) to 40 bytes.

With the arrival of SegWit, script versioning was also introduced, making it is easier to introduce a new `OP_CODE` and so introduce a new signature validation scheme. However, this will not invalidate the present work and research because of the specific nature of its application.

Nevertheless, Schorr signatures are tipped to be the next scheme used in Bitcoin and maybe in other crypto-currencies. Further research could be done to find a protocol that fulfills the requirements defined for payment channel optimization.

7 Conclusions

Some mechanisms of Bitcoin have been explained to allow introducing two significant facts in Bitcoin today, i.e., the scalability problem and the latency due to the security model. The problem of scalability already has existing drafted solutions like consensus changes or layer-two applications such as payment channels. Payment channels are not new to Bitcoin. This idea was suggested by Satoshi in an email to Mike Hearn. A unidirectional payment channel with specific capabilities is proposed. This layer-two application can be improved with threshold cryptography by reducing the size of the transactions without changing the security model. This reduction is made by replacing the multi-signature script by a single signature computed with threshold cryptography. The threshold scheme is analyzed and adapted to ECDSA before being implemented in the existing library used in Bitcoin-core, the library `secp256k1`. Finally, further research about payment channels, Bitcoin in general, and threshold signature schemes are exposed.

The Bitcoin payment channel implementation will be released as open source software soon, and testing will begin in the coming months. The threshold implementation will be part of a current project comprising the creation of an open Bitcoin Teller Machine (BTM) using payment channels to allow instant withdraw cash without security risk for the BTM provider and threshold signature to secure the key on the machine by avoiding full private key.

A Experimental implementation in Python

A Python implementation fully working for testing threshold ECDSA signature and zero-knowledge proofs is available on GitHub at `https://github.com/GuggerJoel/poc-threshold-ecdsa-secp256k1`.

```
#!/usr/bin/env python
import hashlib
import paillier
import ecdsa
import eczkp
import eczkp_pem
import pem
import utils

def alice_round_1(m, x1, y1, ka_pub, ka_priv):
    k1 = utils.randomnumber(ecdsa.n-1, inf=1)
    z1 = utils.invert(k1, ecdsa.n)
    alpha, r1 = paillier.encrypt(z1, ka_pub)
    zeta, r2 = paillier.encrypt(x1 * z1 % ecdsa.n, ka_pub)
    return k1, z1, alpha, zeta, r1, r2

def bob_round_1(alpha, zeta):
    k2 = utils.randomnumber(ecdsa.n-1, inf=1)
    r2 = ecdsa.point_mult(ecdsa.G, k2)
    return k2, r2

def alice_round_2(alpha, zeta, r2, k1, y1, z1, x1, zkp, ka_pub, rr1, rr2):
    Ntild, h1, h2 = zkp
    eta1 = z1
    eta2 = (x1 * z1) % ecdsa.n
    r = ecdsa.point_mult(r2, k1)

    c = r # POINT
    d = ecdsa.G # POINT
    w1 = r2 # POINT
    w2 = y1 # POINT
    m1 = alpha
    m2 = zeta
    x1 = eta1
    x2 = eta2
    r1 = rr1 # RANDOM ALPHA ENC
    r2 = rr2 # RANDOM ZETA ENC

    pi = eczkp.pi(c, d, w1, w2, m1, m2, r1, r2, x1, x2, zkp, ka_pub)
    return r, pi

def bob_round_2(pi, m, alpha, zeta, r, k2, x2, r2, y1, y2, ka_pub, kb_pub, zkp):
    n, g = ka_pub
    n2 = n * n

    rq = r[0] % ecdsa.n
    if rq == 0:
        print("signature failed, retry")
        exit(1)

    z2 = utils.invert(k2, ecdsa.n)
    x2z2 = (x2 * z2) % ecdsa.n
    x3 = utils.randomnumber(pow(ecdsa.n, 5)-1, inf=1)

    if not eczkp.pi_verify(pi, r, ecdsa.G, r2, y1, alpha, zeta, zkp, ka_pub):
        print "Error: zkp failed"
        exit(1)

    mu1 = paillier.mult(alpha, m * z2, n2)
```

```
60      mu2 = paillier.mult(zeta, rq * x2z2, n2)
61      mu3, rnumb = paillier.encrypt(x3 * ecdsa.n, ka_pub)
62      mu = paillier.add(paillier.add(mu1, mu2, n2), mu3, n2)
63
64      muprim, rmuprim = paillier.encrypt(z2, kb_pub)
65
66      c = r2
67      d = ecdsa.G
68      w1 = ecdsa.G
69      w2 = y2
70      m1 = muprim # ENCRYPTED Z2
71      m2 = mu # ENCRYPTED RESULT
72      m3 = alpha # ENCRYPTED Z1
73      m4 = zeta # ENCRYPTED X1Z1
74      r1 = rmuprim
75      r2 = rnumb
76      x1 = z2
77      x2 = x2z2
78      x4 = m
79      x5 = rq
80
81      pi2 = eczkp.pi2(c, d, w1, w2, m1, m2, m3, m4, r1, r2, x1, x2, x3, x4, x5, zkp, ka_pub,
        ↪  kb_pub)
82      if not pi2:
83          print "Error: zkp failed"
84          exit(1)
85
86      return mu, muprim, pi2
87
88  def alice_round_3(pi2, r, r2, y2, mup, mu, alpha, zeta, zkp, ka_priv, kb_pub):
89      n, p, q, g, lmdba, mupaillier = ka_priv
90      ka_pub = (n, g)
91      rf = r[0] % ecdsa.n
92
93      c = r2
94      d = ecdsa.G
95      w1 = ecdsa.G
96      w2 = y2
97      m1 = mup
98      m2 = mu
99      m3 = alpha
100     m4 = zeta
101
102     if not eczkp.pi2_verify(pi2, c, d, w1, w2, m1, m2, m3, m4, zkp, ka_pub, kb_pub):
103         print "Error: zkp 2 failed"
104         exit(1)
105
106     s = paillier.decrypt(mu, ka_priv) % ecdsa.n
107     if s == 0:
108         print("signature failed, retry")
109         exit(1)
110
111     return rf, s
112
113
114 def run_secdsa():
115     # Aclice
116     x1 = utils.randomnumber(ecdsa.n, inf=2)
117     y1 = ecdsa.get_pub(x1)
118     ka_pub, ka_priv = paillier.gen_key()
119
120     # Bob
121     x2 = utils.randomnumber(ecdsa.n, inf=2)
122     y2 = ecdsa.get_pub(x2)
123     kb_pub, kb_priv = paillier.gen_key()
124
125     zkp = eczkp.gen_params(1024)
126
127     pub = ecdsa.get_pub(x1 * x2 % ecdsa.n)
```

```
128
129     # Message hash
130     message = "hello"
131     h = hashlib.sha256()
132     h.update(message.encode("utf-8"))
133     m = long(h.hexdigest(), 16)
134     print "Message to sign: ", message
135     print "Hash: ", m
136
137     # ALICE ROUND 1
138     k1, z1, alpha, zeta, rr1, rr2 = alice_round_1(m, x1, y1, ka_pub, ka_priv)
139     # BOB ROUND 1
140     k2, r2 = bob_round_1(alpha, zeta)
141     # ALICE ROUND 2
142     r, pi = alice_round_2(alpha, zeta, r2, k1, y1, z1, x1, zkp, ka_pub, rr1, rr2)
143     # BOB ROUND 2
144     mu, mup, pi2 = bob_round_2(pi, m, alpha, zeta, r, k2, x2, r2, y1, y2, ka_pub, kb_pub,
        ↪  zkp)
145     # ALICE ROUND 3 (final)
146     sig = alice_round_3(pi2, r, r2, y2, mup, mu, alpha, zeta, zkp, ka_priv, kb_pub)
147
148     print "Signature:"
149     print sig
150     r, s = sig
151     print "Sig status: ", ecdsa.verify(sig, m, pub, ecdsa.G, ecdsa.n)
152
153
154 if __name__ == "__main__":
155     print("S-ECDSA")
156     run_secdsa()
```

Listing A.1 Main file of threshold ECDSA proof-of-concept

Trustless Endless Pulseless and Undelayed One-way Payment Channel for Bitcoin

Thomas Shababi[1], Joël Gugger[2], and Daniel Lebrecht[1]

[1] DigiThink, Neuchâtel, Switzerland
info@digithink.ch
[2] HES-SO Master, Lausanne, Switzerland
joel.gugger@master.hes-so.ch

Abstract. The greatest challenge for Bitcoin in the coming years is scalability. Currently, Bitcoin enforces a 1 Megabyte block-size limit which is equivalent to ~7 transactions per second on the network. This is not sufficient in comparison to big payment infrastructure such as credit card processors, which allows tens of thousands of transactions per second and even more in peaks like Christmas. To address this, there are some proposals to modify the transaction structure (like SegWit), some to modify the block-size limit (such as SegWit2x) and others to create a second layer on top of the Bitcoin protocol (such as Lightning Network). In the same idea of the Lightning Network, we propose a one-way payment channel that allows two parties to transact off-chain while minimizing the number of transactions needed in the blockchain in a secure and trustless way.

Keywords: Crypto-currencies, Bitcoin, Payment channels, State channels, Threshold ECDSA signatures

1 Introduction

Decentralized crypto-currencies such as Bitcoin [5] and its derivatives employ a special decentralized public append-only log based on proof-of-work called the *blockchain* to protect against equivocation in the form of *double-spending*, i.e., spending the same funds to different parties. In a decentralized crypto-currency, users transfer their funds by publishing digitally signed transactions. Transactions are confirmed only when they are included in the blockchain, which is generated by currency miners who solve proof-of-work puzzles. Although a malicious owner can sign over the same funds to multiple receivers through multiple transactions, eventually only one transaction will be approved and added to the publicly verifiable blockchain.

But the blockchain is slow and potentially expensive in fees when the time comes to broadcast transactions. Scalability is one of the biggest challenges in blockchain systems these days, and as mentionned before, some proposals are focused on the blockchain data structures and the consensus layer, others such as the Lightning Network [6] are focused on a second layer of transactions where

transactions are created off-chain and the blockchain itself is used as a conflict resolution system and source of truth. These proposals are called payment channels and provide a wide number of advantages.

1.1 Our contribution

Most retail commecrial transactions are unidirectional, therefore bidirectional channels are not always necessary. Streaming payments in a consumer context are mostly unidirectional and bidirectional channels impose the burden upon both parties to police the channel by listening to the network and submitting transactions to enforce correct state, which greatly increases the complexity for participants. In this paper we propose a simplified scheme aimed to be used in a consumer context with a service provider offering goods or services to many clients and receiving the payments into payment channels for rapidity and convenience.

In our scheme, inspired by by the Lightning Network and "Yours Lightning Protocol" [6, 1], the client, hereinafter Carol, wants to buy goods or services from the provider Bob. Bob is likely to sell goods or services to Carol several times and wants to receive payments into a channel to minimize transaction costs and have instant transaction finality. For this scheme to be realistic, some requirements are assumed. The channel must stay open for an undefined amount of time and the client must not be obliged to stay online and watch the blockchain to be safe, only the receiver, i.e., the provider, must stay online to be safe. The provider does not want to lock any funds for the clients, if he needs to send money to some clients, it is asumed that these transactions are regular on-chain transactions or via other channels. When clients send money via channels, the provider must be able to manage when and how funds are settled from which channels, so he must be able to settle a channel without closing it. A client, who has blocked funds specifically for the provider, must be able to, with the providers cooperation, withdraw an arbitrary amount out of the channel without closing it.

We propose several definitions to qualify a payment channel and generalize the analysis of different implementations. These definitions are not standard nor official in any manners.

Definition 1 (Trustless). *A channel is trustless if and only if the funds' safety for every player $p_i \in \mathcal{P} = \{\mathcal{P}_0, \ldots, \mathcal{P}_n\}$ at each step $\mathcal{S}$ of the protocol does not depend on players' $\Delta p = \mathcal{P} - p_i$ behavior.*

Definition 2 (Optimal). *A channel is optimal if and only if the number of transactions $\mathcal{T}(\mathcal{C})$ needed to claim the funds for a given constraint $\mathcal{C}$ is equal to the number of moves $\mathcal{M}(\mathcal{C})$ needed to satisfy the constraint at any time without breaking the first definition.*

For a constraint $\mathcal{C}$ in a channel $\mathcal{P}_1 \to \mathcal{P}_2$, refunding $\mathcal{P}_1$ requires $\mathcal{M}(\mathcal{C}) = 1$, thus an optimal scheme requires $\mathcal{T}(\mathcal{C}) = \mathcal{M}(\mathcal{C}) = 1$. Note: in a channel $\mathcal{P}_1 \to \mathcal{P}_2$ refund and settlement both require $\mathcal{M}(\mathcal{C}) = 1$.

Definition 3 (Endless). *A channel is endless if and only if there is no predetermined lifetime at the setup.*

Definition 4 (Pulseless). *A channel is pulseless if and only if there is no need to refresh or close the channel on-chain while at least one player* $p_i \in \mathcal{P} = \{\mathcal{P}_0, \ldots, \mathcal{P}_n\}$ *where the available amount to send is* $A(p_i) > 0$*. By definition a pulseless channel must be also endless.*

Definition 5 (Undelayed). *A channel is undelayed if and only if each player* $p_i \in \mathcal{P} = \{\mathcal{P}_0, \ldots, \mathcal{P}_n\}$ *can trigger the process to get their money back at any time.*

2 Building Blocks

In the following, the concepts and sub-protocols used in this work are described in more detail.

2.1 Channel State

The channel state is expressed by two indexes i and n, hereinafter also $\texttt{Channel}_{i,n}$. Both indexes are independent and can only be positively incremented. Index i represents the offset of the multisig address where the channel's funds are locked. Index n represents the offset used to create the revocation secrets, this secret is used after in smart contracts.

A channel state always depends on an account a, this account is defined when the channel is created between the client and the server and never changes during its life. We need to share public hierachical deterministic addresses between the client and the server. Let's define the hierachical deterministic Bitcoin account path as:

$$\forall a \geq 2, \exists \texttt{xPriv}_a \mid \texttt{xPriv}_a = \texttt{m/44'/0'/a'}$$
$$\forall a \geq 2, \ \exists \texttt{xPub}_a \ \mid \texttt{xPub}_a = \texttt{m/44'/0'/a'}$$

For a given account a at $\texttt{Channel}_{i,n}$, the protocol and transactions depend on the private multi-signature node $\mathit{\Pi}$, the public multi-signature node π, the private revocation node Ω, the public revocation node ω, and the private secret node Θ. Let's define these nodes as:

$$\mathit{\Pi}_i = \texttt{xPriv}_a \texttt{/0/i}$$
$$\pi_i \ = \texttt{xPub}_a \ \texttt{/0/i}$$
$$\Omega_i = \texttt{xPriv}_a \texttt{/1/i}$$
$$\omega_i \ = \texttt{xPub}_a \ \texttt{/1/i}$$
$$\Theta_n = \texttt{xPriv}_a \texttt{/2'/n'}$$

It is worth noting that $\mathit{\Pi}_i$, π_i, Ω_i, and ω_i aren't hardened derivations, instead of Θ_n. This because we need to be able to compute the public keys π_i and ω_i from the $\texttt{xPub}_a$.

Channel Dimensions The channel dimension, noted $|\texttt{Channel}|$, depends of the number of indexes present in the state. Let's define the channel dimension:

$$N = |\texttt{Channel}_{i,n}| = 2$$

Revocation Secret The revocation sercret $\Phi_{i,n}$ corresponds to the state $\texttt{Channel}_{i,n}$ and depends on the secret Θ_n and the revocation key Ω_i.

$$\Phi_{i,n} = \texttt{HMAC}(\Theta_n, \Omega_i)$$

The secret is the HMAC of Θ_n and Ω_i. Both indexes are used to protect Carol from the Old Settlement Attack With Weak Secret (see 4.3).

2.2 Smart Contracts

Two types of smart contract are used in the payment channel scheme. The fisrt one is a standard 2-out-of-2 multi-signature script and the second is a custom script used to prevent the client from broadcasting old transactions.

Multisig Contract The multi-signature contract at $\texttt{Channel}_{i,n}$, hereinafter $\texttt{Multisig}_i$, can be constructed with Carol's π_i key, and Bob's π_i key. Let's define the $\texttt{Multisig}_i$ script:

OP_2 <π_i^{carol}> <π_i^{bob}> OP_2 OP_CHECKMULTISIG

Revocable PubKey Contract Bob and Carol may wish make an output to Carol which Carol can spend after a timelock and Bob can revoke if it is an old state. The next contract, for $\texttt{Channel}_{i,n}$, uses Carol's ω_i key, Bob's ω_i key, and Carol's secret $\Phi_{i,n}$.

```
OP_IF
  <ω_i^carol> OP_CHECKSIG
  <timelock> OP_CHECKSEQUENCEVERIFY OP_DROP
OP_ELSE
  <ω_i^bob> OP_CHECKSIGVERIFY
  OP_HASH160 <Hash160(Φ_{i,n})> OP_EQUAL
OP_ENDIF
```

With this contract Carol can spend this output after the timelock with the script signature:

<Ω_i^{carol} signature> OP_TRUE

In the case if Carol broadcasts an older transaction Bob can revoke it with the script signature:

<Carol's $\Phi_{i,n}$> <Ω_i^{bob} signature> OP_FALSE

Bob has a head start during which, if he knows the secret $\Phi_{i,n}$ generated by Carol, he can spend the money while Carol cannot. This mechanism prevents Carol from broadcasting older transactions which do not match the current $\texttt{Channel}_{i,n}$.

2.3 Transactions

A transaction is noted $\texttt{Transaction}^{i,n}_{<>}$ to denote the name of the transaction, on which indexes this transaction depends—here on indexes i and n—and who has already signed this transaction—denoted by the <>. If a transaction is signed by Carol the transaction is noted $\texttt{Transaction}^{i,n}_{<carol>}$. Transactions that appear in blue on figures are only owned fully signed by Carol and red ones only by Bob, i.e., only the owner can broadcast the transaction.

Funding Transaction The funding transaction, hereinafter $\texttt{FundingTx}^{i}_{<>}$, is the transaction sending funds to the first multisig address. This transaction depends only on the state index i used by the multisig contract and is fully signed as soon as Carol signs it.

A funding transaction is never broadcast by Carol until she possesses the corresponding refund transaction that allows her to get her money back off the channel. This refund transaction has only one output that goes to the revocation contract. To be able to revoke this contract Bob has to know the secret $\Phi_{i,n}$, so if no transactions have been made Bob cannot revoke the contract.

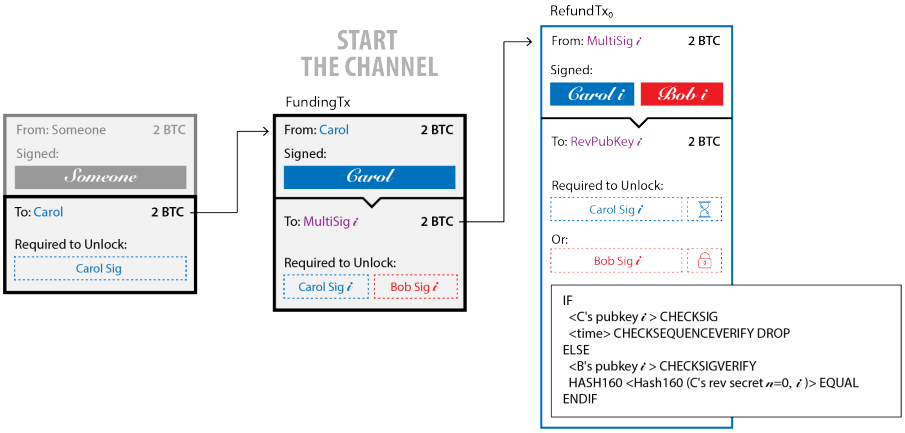

Fig. 1. Funding transaction that starts the channel by sending money in the first multisig address with the first refund transaction that allows Carol to close the channel if no transaction is made.

Refund Transaction The refund transaction, hereinafter $\texttt{RefundTx}^{i,n}_{<>}$, is a transaction that keeps track of the balances of Carol and Bob at $\texttt{Channel}_{i,n}$ and allows Carol to close the channel if Bob does not repsond or does not cooperate anymore. This transaction has one input or more—which come from the multisig address corresponding to the state index i—and two outputs. The first output represents the amount still owned by Carol, and the second—which can be non-present in the transaction if the balance is equal to zero—is the amount owned by Bob. This second output is not present when the channel is open and each time Bob settles the channel.

Carol's balance is sent to a revocation contract corresponding to the channel state. This prevents Carol from broadcasting an old refund transaction such as $\texttt{RefundTx}^{i,n-1}_{<>}$. The amount owned by Bob is sent directly to Bob's address. The refund transaction is broadcast by Carol so the fees are substracted from the first output, owned by Carol.

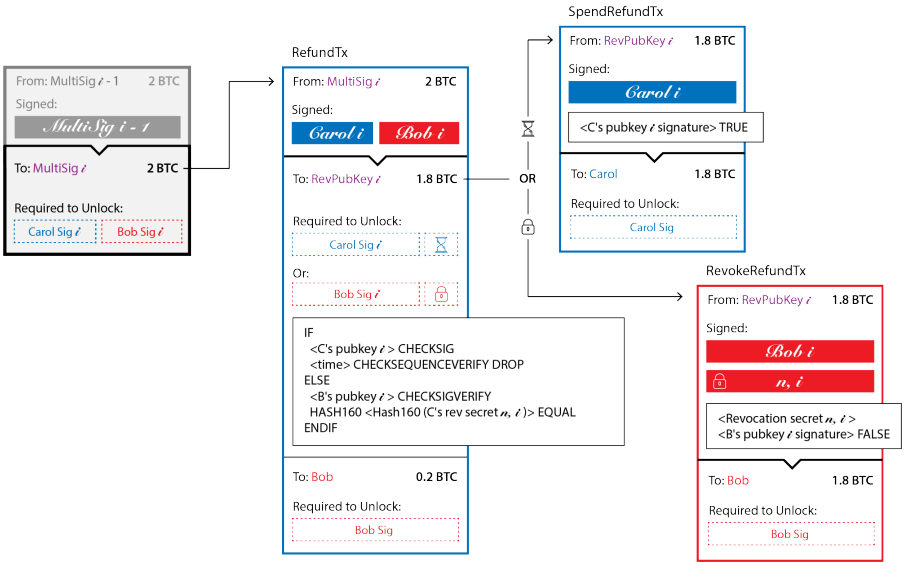

Fig. 2. Refund transaction based on the current multisig address with the associated spend and revoke transactions that allows Carol to get her money back and Bob to revoke the contract if he knows the secret.

Because a refund transaction spends funds from a multisig address, it must be signed by both Carol and Bob to be considered valid. The revocation contract used in the Carol's output can be spent with a spend refund transaction after a timelock delay. She just needs to sign the output with her Ω_i key to unlock the funds. Bob can directly revoke the contract, without delay, if he knows the secret $\Phi_{i,n}$ and sign with his Ω_i key.

Settlement Transaction The settlement transaction, hereinafter also mentioned as $\texttt{SettlementTx}^{i,n}_{<>}$, is a transaction that keeps track of Carol and Bobs balances at $\texttt{Channel}_{i,n}$ and allows Bob to settle the channel without closing it. Because the settlement transaction spends the funds from the multisig address, both Carol and Bob need to sign to consider the transaction as valid. Fees are substracted from Bob's output, because he is responsible for broadcasting the transaction and settling the channel.

A settlement transaction always has one output that sends Bob's balance directly to Bob's address and one output that send the remaining funds to the next multisig address $\texttt{Channel}_{i+1,n}$. Because the funds are sent to the next multisig address a post settlement refund transaction is created—Carol needs a way to get her money back off the channel. This transaction has the same structure as the first refund transaction—one output to the next revocation contract—because the funds owned by Bob, in this case, are already settled.

If Bob broadcasts the fully signed settlement transaction, Carol has two choices (i) continue to transact on the channel with the new multisig address and (ii) close the channel with her post settlement refund transaction. It is worth noting that the secret for the revocation contract is $\Phi_{i+1,n}$, so in the case of a new transaction $\texttt{Channel}_{i+1,n}$ becomes $\texttt{Channel}_{i+1,n+1}$ and then, the secret for $\texttt{Channel}_{i,n-1}$ is shared:

$$\Phi_{i,n-1}(\texttt{Channel}_{i+1,n+1}) = \Phi_{i+1,n}$$

Post Settlement Refund Transaction The post settlement refund transaction aims to spend funds from the next multisig address directly to a revocation contract. As explained before, this contract is not revocable by Bob if no transaction is made after the settlement transaction, but when Carol sends an amount to Bob she shares the secret needed to revoke the contract, thus she cannot broadcast this transaction that is now attached to an old state.

Withdraw Transaction The withdraw transaction, hereinafter $\texttt{WithdrawTx}_i$, is a transaction that allows Carol to take an arbitrary amount of money out the channel. This amount is sent to an arbitrary address specified by Carol. For this, she has to ask Bob for his cooperation. This transaction is not autogenerated when Carol sends money to Bob, both have to be online to create this transaction.

This transaction automatically settles the channel with the amount owned by Bob at $\texttt{Channel}_{i,n}$ and changes the remaining amount available for Carol for the state $\texttt{Channel}_{i+1,n}$, thus, remaining funds are moved to the next channel address.

Closed Channel Transaction The close channel transaction, hereinafter also mentioned as $\texttt{ClosedChannelTx}_i$, is also a cooperative transaction, that allows Carol or Bob to close the channel in the most effective way (less fee and quicker). This transaction has two outputs, one for Bob with the amount owned by Bob

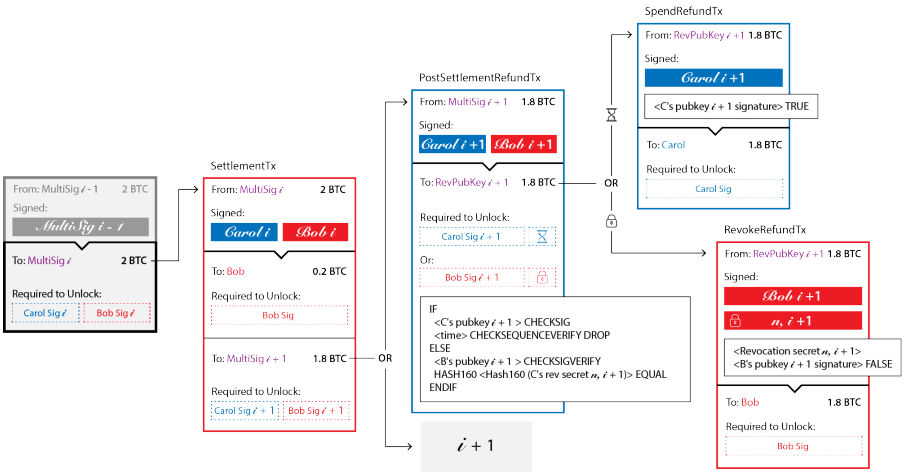

Fig. 3. Settlement transaction that allows Bob to settle the channel, moving the remaining funds to the next multisig address with the post settlement refund transaction that allows Carol to close the channel directly after the settlement.

to his address and a second one for Carol with the remaining amount of money in the channel. When a $\texttt{ClosedChannelTx}_i$ is created, no more transactions can be created or accepted on the channel.

Pay To Channel Transaction The pay to channel transaction, hereinafter $\texttt{PayToChannelTx}_i$, allows Carol or Bob to send money directly to the current channel address. This transaction is usefull for Carol if there is not enough money on the channel and she wants to send more money to Bob without opening another payment channel. It is also usefull for Bob in the case he wants to send money to Carol—he can send money directly to Carol's address, but the payment can be related to a channel event or action—and allows her to reuse it in the channel, e.g fidelity points.

Before broadcasting the pay to channel transaction or before accepting that payment as part of the usable funds for Carol, an interim refund transaction needs to be created. This interim refund transaction is a safty garauntee for Carol until the merge occures.

For a state $\texttt{Channel}_{i,n}$ without any pay to channel transaction, the multisig address has, normaly, one unspent output. This unspent output is used as an input for each transaction and these transactions split it to track the balances of each party. After a pay to channel transaction, the multisig address has more than one unspent output. When Carol sends money to Bob they have to check if a pay to channel transaction occured and if it is the case they need to merge the interim refund and use all the unspent outputs. It is worth noting that the more pay to channel transactions occur the more (fee) cost incurred.

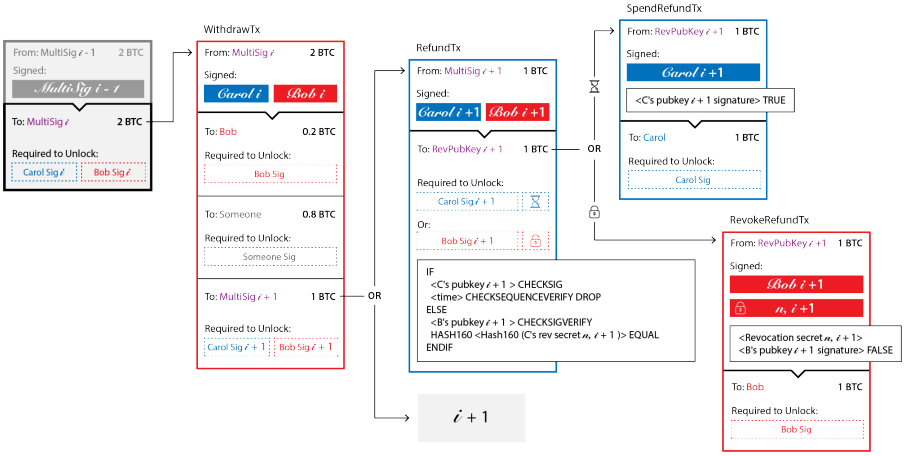

Fig. 4. Withdraw transaction that allows Carol to take money out the channel, pay Bob, and move the remaining funds into the next multisig address. A refund transaction is created to allow Carol to recover her funds after the withdraw if no transaction is made. The refund transaction can be spent by Carol with a spend refund transaction and cannot be contested with the revoke refund transaction if no other transaction is made. If the state moves to $\texttt{Channel}_{i+1,n+1}$, again, the secret for $\texttt{Channel}_{i,n-1}$ is shared, then Bob knows $\Phi_{i,n-1}(\texttt{Channel}_{i+1,n+1}) = \Phi_{i+1,n}$ and can revoke.

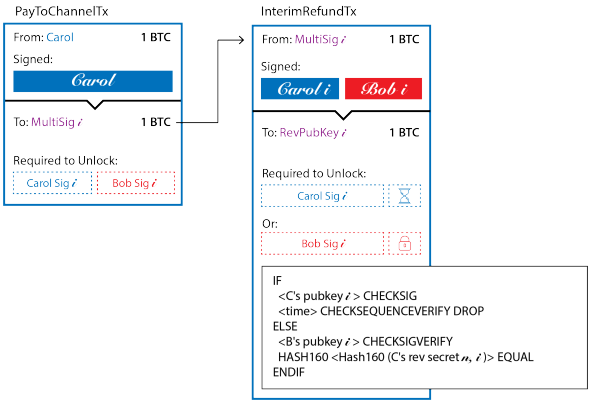

Fig. 5. Pay to channel transaction by Carol with the interim refund transaction. The interim refund transaction acts like a standard refund transaction but aims to be merged in the next round of transactions. The interim refund transaction has the same spending requirements as a standard refund transaction, if no transaction is made Carol can spend the interim refund, otherwise Bob can revoke the interim refund.

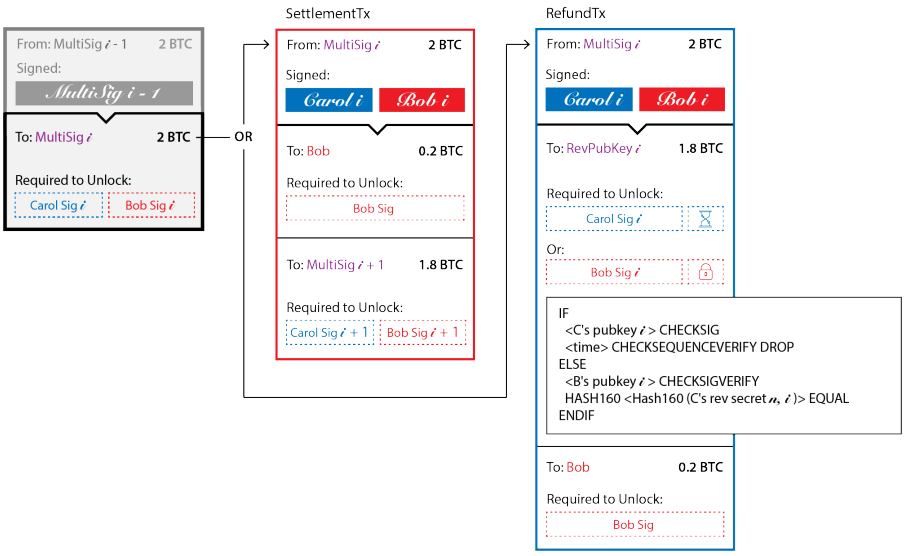

Fig. 6. Simplified view of possibilites for a standard state $\texttt{Channel}_{i,n}$ without second layer dependency transactions like spend and revoke. The content of a multisig can be settled by Bob or can be refunded to Carol.

Interim Refund Transaction The interim refund transaction, hereinafter $\texttt{InterimRefundTx}_{i,n}$, is a temporary transaction used by Carol to get her money out of the channel. This transaction is created to protect Carol from Bob invalidating the current refund transaction.

Definition 6. *A channel merge occurs each time an interim refund transaction is merged into the regular refund transaction and the regular settlement transaction.*

Definition 7. *A channel reduce occurs each time a non-closing channel transaction is broadcast and included in the blockchain. The channel is then in reduced mode when one and only one* `UTXO` *is available in the current multisig address.*

3 Trustless One-way Payment Channel

3.1 Channel Setup

Before opening the channel Carol and Bob need to exchange keys for the channel account a and negotiate the relative timelock value.

1. Carol:
   (a) sends a request to open a channel with:
      i. the account a

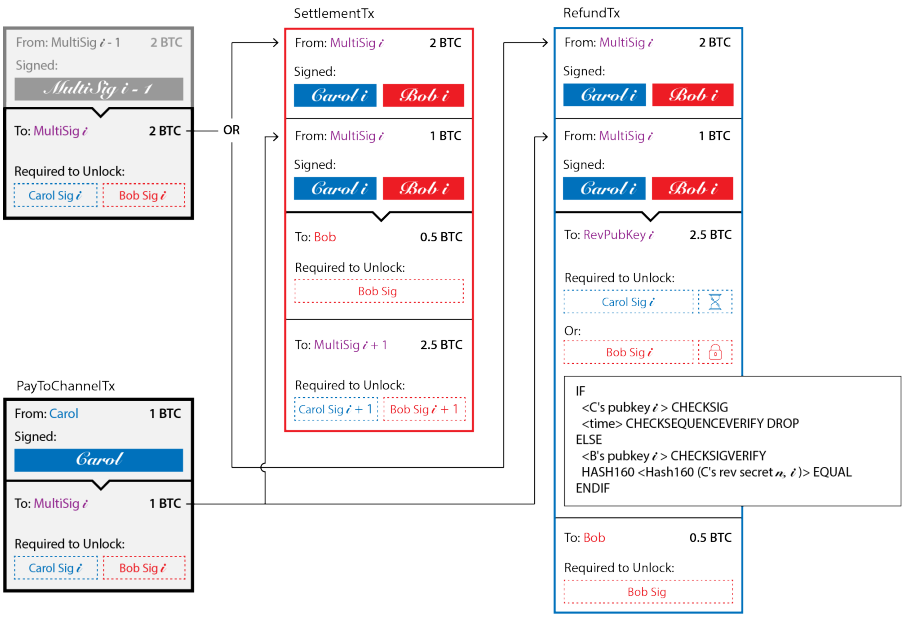

Fig. 7. Result of a merged pay to channel transaction after Carol sends 0.3 more Bitcoin to Bob. The $\texttt{Multisig}_i$ contains two `UTXOs`, (i) from the funding transaction or the last move from $\texttt{Multisig}_{i-1}$, and (ii) from the pay to channel transaction adding 1 BTC into the channel. Both the settlement transaction and refund transaction contain the two `UTXOs` as inputs to sign and spend the totality with the adjusted balances.

   ii. Carol's $\texttt{xPub}_a$

   iii. the relative timelock parameter

2. Bob:
   (a) if Bob agrees with the request and timelock is whithin acceptable range, respond with:
      i. Bob's $\texttt{xPub}_a$

3.2 Channel Opening

To open the channel, Carol and Bob must cooperate to generate and fund a multi-signature address, hereinafter also referred to as $\texttt{Multisig}_i$ address. This multi-signature address acts as the channel address and stores the totality of the channel's funds. This address holds normaly only one `UTXO`, but with pay to channel transactions, this could be different.

1. Bob:
   (a) generates $\texttt{Multisig}_i$ with Bob's Π_i and Carol's π_i and sends it
2. Carol:

(a) creates $\texttt{FundingTx}^{i}_{<>}$ that funds the $\texttt{Multisig}_i$ address

(b) generates $\Phi_{i,n}$

(c) creates $\texttt{RefundTx}^{i,n}_{<>}$ with $\texttt{Multisig}_i$ and $\Phi_{i,n}$ sending the full amount back to herself via the $\texttt{RevPubKey}_{i,n}$ contract

(d) initiates the channel by sending:
   i. $\texttt{hash}(\Phi_{i,n})$
   ii. $\texttt{RefundTx}^{i,n}_{<>}$

3. Bob:
   (a) receives $\texttt{RefundTx}^{i,n}_{<>}$, signs it and returns $\texttt{RefundTx}^{i,n}_{<bob>}$
4. Carol:
   (a) broadcasts $\texttt{FundingTx}^{i}_{<carol>}$
5. Bob:
   (a) waits for transaction's confirmations
   (b) consider the channel as open

If Bob stops responding after step 2, Carol has created transactions which she is unable to use. If Carol stops responding after step 3, Bob has signed a transaction which will probably never be used. After a while, Bob must consider the channel opening as failed. If Bob stops responding after step 4, Carol can broadcast her refund transaction and she is safe. If Carol stops responding after opening the channel Bob does not lose anything.

3.3 Transact

Carol to Bob The Carol to Bob protocol allows Carol to send an arbitrary amount of money throught the channel. Carol desires to authorize a payment of M satoshis to Bob at $\texttt{Channel}_{i,n}$ state.

If there is an unconfirmed Bob to Carol transaction and we need to use it because there are no more funds in the regular refund output, we have to merge that refund in that new $\texttt{Channel}_{i,n+1}$ state. Bob can trust this unconfirmed output because it comes from himself.

1. Carol:
   (a) derives $\Phi_{i,n+1}$ and $\Phi_{i+1,n+1}$
   (b) generates the $\texttt{RefundTx}^{i,n+1}_{<>}$ with two outputs:
      i. Refund Output: Carol's new balance to $\texttt{RevPubKey}_{i,n+1}$ contract
      ii. Settlement Output: Bob's new balance to settlement address
   (c) sends a message to Bob containing:
      i. $\texttt{RefundTx}^{i,n+1}_{<>}$
      ii. $\texttt{hash}(\Phi_{i,n+1})$ and $\texttt{hash}(\Phi_{i+1,n+1})$

iii. the amount of M satoshis being paid

2. Bob:
   (a) generates the $\texttt{SettlementTx}^{i}_{<>}$ with two outputs:
      i. Settlement Output: Bob's new balance
      ii. Change Output: Carol's new balance to $\texttt{Multisig}_{i+1}$ with Bob's Π_{i+1} and Carol's π_{i+1}
   (b) generates the $\texttt{PostSettlementRefundTx}^{i+1,n+1}_{<bob>}$ with:
      i. Refund Output: sends Carol's funds to the associated $\texttt{RevPubKey}_{i+1,n+1}$ contract with the secret = $\texttt{hash}(\Phi_{i+1,n+1})$
   (c) sends:
      i. $\texttt{RefundTx}^{i,n+1}_{<bob>}$
      ii. $\texttt{SettlementTx}^{i}_{<>}$
      iii. $\texttt{PostSettlementRefundTx}^{i+1,n+1}_{<bob>}$
3. Carol:
   (a) sends:
      i. $\texttt{SettlementTx}^{i}_{<carol>}$
      ii. the shared secret $\Phi_{i,n}$
4. Bob:
   (a) upates state channel to $\texttt{Channel}_{i,n} \Rightarrow \texttt{Channel}_{i,n+1}$ and the payment can now be considered as final

If Bob stops responding after step 2, Carol can broadcast the refund transaction but she has no incentives to do that because she will lose part of her balance compared to broadcasting the previous state refund transaction. Because she has not yet shared the secret $\Phi_{i,n}$, Bob cannot yet revoke the current $\texttt{Channel}_{i,n}$ state. If Carol does not respond at step 3, Bob can settle the current $\texttt{Channel}_{i,n}$ state, but cannot settle the $\texttt{Channel}_{i,n+1}$ state in negotiation, Carol is safe. After step 3, Carol can refund herself and Bob can revoke the old $\texttt{Channel}_{i,n}$ state and settle the new $\texttt{Channel}_{i,n+1}$ state, the transaction is complete.

Channel Topop The channel topop protocol allows Bob or Carol to send an output directly to the current channel multisig address and allows Carol to include this output as part of usable funds immediatley if it's from Bob. If the funds come from Carol, they can be immediatley used for a withdraw transaction only.

To protect Carol, the refund for this additional amount is separate to the existing refund output to prevent Bob from invalidating Carol's refund transaction by sending an output which becomes invalid or not accepted by the network (lower fee, double spend, invalid script, etc.)

In subsequent transactions, once this output has confirmed, the refund should be merged into a single refund output as before, to be more efficient with refund transaction size.

1. Initiator:
   (a) create the $\texttt{PayToChannelTx}^{i}_{<initiator>}$ that funds the $\texttt{Multisig}_i$ address
   (b) create $\texttt{InterimRefundTx}^{i,n}_{<>}$ with $\texttt{Multisig}_i$ and $\texttt{hash}(\Phi_{i,n})$ sending the full amount back to Carol via the $\texttt{RevPubKey}_{i,n}$ contract
   (c) sends:
      i. $\texttt{InterimRefundTx}^{i,n}_{<initiator>}$
2. Receiver:
   (a) validate $\texttt{InterimRefundTx}^{i,n}_{<initiator>}$
   (b) sends if the payment is accepted or not
3. Initiator:
   (a) if the payment is accepted broadcast $\texttt{PayToChannelTx}^{i}_{<initiator>}$
4. Receiver:
   (a) wait for $\texttt{PayToChannelTx}^{i}_{<initiator>}$ transaction's confirmations

If the receiver does not validate the payment the initiator has no incentive to broadcast the transaction, if it is accepted, then the initiator can send money into the channel safely because of the interim refund transaction. Without negotiating a new state, Bob cannot revoke $\texttt{InterimRefundTx}^{i,n}_{<initiator>}$ and Carol can spend the refund. When a new $\texttt{Channel}_{i,n+1}$ state is negotiated, Bob can revoke the $\texttt{InterimRefundTx}^{i,n}_{<initiator>}$ if Carol tries to broadcast it. At $\texttt{Channel}_{i,n+1}$, the refund transaction and the settlement transaction contain the merged refund transaction.

It is worth noting that the initiator does need to know the secret to create the pay to channel transaction and the interim refund transaction. An external player can ask the needed information to topop the channel knowing only public information.

Withdrawing The withdraw protocol allows Bob to authorize a withdrawal of M satoshis at Carol's request and with her cooperation for $\texttt{Channel}_{i,n}$ state. Bob needs to validate the withdrawal amount and can set up a set of rules internally to manage the channel economics. It is worth noting that when the withdraw takes place Bob funds get automatically settled without him paying fee.

1. Carol:
   (a) derives $\Phi_{i+1,n}$
   (b) generates the $\texttt{Multisig}_{i+1}$ address with Bob's π_{i+1} and Carol's Π_{i+1}
   (c) generates the $\texttt{WithdrawTx}^{i}_{<>}$ with:
      i. Settlement Output: sends Bob's funds to the settlement address
      ii. Withdraw Output: withdrawal amount M to specified address
      iii. Change Output: new balance into $\texttt{Multisig}_{i+1}$

(d) generates the $\texttt{RefundTx}^{i+1,n}_{<>}$ from address $\texttt{Multisig}_{i+1}$ with:
   i. Refund Output: Carol's new balance into $\texttt{RevPubKey}_{i+1,n}$ contract with the secret = $\texttt{hash}(\Phi_{i+1,n})$

(e) sends:
   i. $\texttt{RefundTx}^{i+1,n}_{<>}$
   ii. $\texttt{WithdrawTx}^{i}_{<>}$
   iii. $\texttt{hash}(\Phi_{i+1,n})$

2. Bob:
   (a) verifies, signs and returns:
      i. $\texttt{RefundTx}^{i+1,n}_{<bob>}$
      ii. $\texttt{WithdrawTx}^{i}_{<bob>}$
3. Carol:
   (a) shares:
      i. $\Phi_{i,n}$ to invalidate the current state
      ii. $\texttt{WithdrawTx}^{i}_{<bob,carol>}$
4. Bob:
   (a) broadcast $\texttt{WithdrawTx}^{i}_{<bob,carol>}$
   (b) upates state channel to $\texttt{Channel}_{i,n} \Rightarrow \texttt{Channel}_{i+1,n}$ and validate exchange

If Bob does not respond in step 2, Carol has not disclosed any important information. If Bob stops responding after step 2, Carol can withdraw the amount and safely refund her funds if no transaction is negotiated. If Carol does not respond after step 2, Bob must wait a while and if the withdraw transaction is not broadcasted, he must broadcast the settlement transaction to force the transition to the next $\texttt{Channel}_{i+1,n}$ state.

Settlement The settlement protocol allows Bob to broadcast at $\texttt{Channel}_{i,n}$ state the $\texttt{SettlementTx}_{i,n}$ to get the settlement output and move the remaining funds into the next $\texttt{Multisig}_{i+1}$ address. In this case the channel stays open and Carol can create new transactions or close the channel.

If the $\texttt{SettlementTx}_{i,n}$ is broadcast and Carol wants to close the channel, she can broadcast the $\texttt{PostSettlementRefundTx}_{i,n}$ and wait the timelock to get her money back. Carol has to query the network to know if the $\texttt{SettlementTx}_{i,n}$ is broadcast, she can only query the blockchain before each new transaction to be sure that the settlement transaction has not been broadcasted yet.

3.4 Channel Closing

Cooperative Closing the channel cooperatively allows Carol—or Bob if Carol is online—to ask if Bob agrees to close the channel efficiently, withdrawing the full remaining balance, at $\texttt{Channel}_{i,n}$ state. The following steps 3 and 4 can be merged and executed by the same player depending on the implementation.

1. Carol:
   (a) generates the $\texttt{ClosedChannelTx}^{n+1}_{<carol>}$ with:
      i. Settlement Output: sends Bob's funds to Bob address
      ii. Change Output: sends Carol's funds to Carol address
   (b) sends $\texttt{ClosedChannelTx}^{n+1}_{<carol>}$
2. Bob:
   (a) verifies and signs $\texttt{ClosedChannelTx}^{n+1}_{<carol>}$
   (b) sends $\texttt{ClosedChannelTx}^{n+1}_{<carol,bob>}$
3. Carol:
   (a) broadcasts $\texttt{ClosedChannelTx}^{n+1}_{<carol,bob>}$

Contentious The contentious channel closing protocol allows Carol to close the channel alone, i.e., without Bob's cooperation or response, at $\texttt{Channel}_{i,n}$ state. Carol can broadcast her fully signed refund transaction sending her funds to $\texttt{RevPubKey}_{i,n}$ address. Carol would then need to spend from the revocation public key contract after the timelock delay with the spend refund transaction.

It is worth noting that only Carol can close the channel, but Bob can get his money by broacasting his settlement transaction at any time.

4 Evidence of Trustlessness

In the following, axioms, possible edge-cases, and discovered attacks, with an evidence of trustlessness for the channel protocol, are exposed. *Liveness* in the blockchain, i.e. transactions can be included in the next blocks, is asumed to guarantee the security model.

4.1 Axioms

Refund Transaction For $\texttt{Channel}_{i,n}$ state, if Carol broadcasts the current refund transaction, Bob cannot revoke it without knowing $\Phi_{i,n}$. After the timelock, Carol can generate and broadcast a spend refund transaction.

$$\forall i \wedge n, \exists \Phi_{i,n} : \quad \forall \Phi_{i,n}, \text{bob knows } \Phi_{i,n-1} \wedge \texttt{hash}(\Phi_{i,n})$$

The same rule is applied to interim refund transactions, if Carol broadcasts the current interim refund transaction, Bob cannot revoke it without knowing $\Phi_{i,n}$, and Carol can spend the interim refund after the timelock.

Old Refund Transaction For $\texttt{Channel}_{i,n}$ state, if Carol broadcasts an old refund transaction, e.g. $n-1$, then Bob has the time during the timelock to generate and broadcast the revoke transaction for the state $\texttt{Channel}_{i,n-1}$ with $\Phi_{i,n-1}$ secret.

$$\forall 0 \leq x < n, \exists \Phi_{i,x} : \quad \forall \Phi_{i,x}, \exists \texttt{RevokeTx}^{i,x}_{<bob>}$$

The same rule is applied to old interim refund transactions, if Carol broadcasts an old interim refund transaction, e.g. $n-1$, Bob can revoke it with $\Phi_{i,n-1}$ secret.

Settlement Transaction For $\texttt{Channel}_{i,n}$ state, if Bob broadcasts his $\texttt{SettlementTx}_{i,n}$ transaction, Carol has the choice to close the channel or transact on top of the new $\texttt{Multisig}_{i+1}$ address.

$$\forall \mathcal{C} = \texttt{Channel}_{i,n}, \exists \Phi_{i,n} \wedge \Phi_{i+1,n} : \quad \text{bob knows } \Phi_{i+1,n} \text{ iff } \mathcal{C} = \texttt{Channel}_{i+1,n+1}$$

Contentious Channel Closing By contentious it means that all players are not communicating anymore and/or do not agree on a valid state. Let's define the way for Carol to close the $\texttt{Channel}_{i,n}$ state.

$$\forall \texttt{Channel}_{i,n}, \exists \texttt{RefundTx}^{i,n}_{<carol,bob>} \text{only owned by Carol}$$

and

$$\forall \texttt{RefundTx}^{i,n}_{<carol,bob>}, \exists \texttt{SpendRefundTx}^{i,n}_{<carol>} \text{only owned by Carol}$$

so:

$$\forall \texttt{Channel}_{i,n}, \exists \texttt{SpendRefundTx}^{i,n}_{<carol>} \text{only owned by Carol}$$

4.2 Edge Cases

Someone does not broadcast Cooperative Transaction If one player does not share a fully signed cooperative transaction and the secret $\Phi_{i,n}$ attached to the current $\texttt{Channel}_{i,n}$ state, then the other player need to force after a while the transition into the new $\texttt{Channel}_{i+1,n}$ state with his own fully signed transaction, i.e $\texttt{RefundTx}_{i,n}$ or $\texttt{SettlementTx}_{i,n}$ transaction.

4.3 Attacks

In this section, attack vectors discovered and fixes are discussed. Attacks exposed are no longer valid in the current scheme, but a deep analysis has been carried out to generalize the protocol construction and improve the scheme.

Old Settlement Attack With Weak Secret It is possible for Bob to lock the funds in the multisig or steal the money if the secret construction is too weak. For a channel at N dimensions the secret is considered weak if:

$$|\Phi| < N$$

Let's assume that the revocation secret Φ depends only on n and not on i for $\texttt{Channel}_{i,n}$. Thus, the secret can be expressed by:

$$|\texttt{Channel}_{i,n}| = N = 2 : |\Phi_n| = 1 \implies |\Phi_n| < N$$

Then, for $\texttt{Channel}_{i,n}$, if Bob broadcast an old settlement transaction, e.g. $n-1$, then Carol cannot use her post settlement refund transaction because she previously shared the secret Φ_{n-1}. So the remaining funds are blocked in the $\texttt{Multisig}_{i+1}$ address. To be able to get her funds back, Carol would have to transact with Bob, if Bob does not cooperate, Carol has no way to recover her funds. If she tries to refund the $\texttt{Multisig}_{i+1}$, then Bob can revoke with Φ_{n-1} secret.

If the secret dimension is equal to the channel dimension, i.e. $|\Phi_{i,n}| = |\texttt{Channel}_{i,n}|$, then the previous shared secret is $\Phi_{i,n-1}$ and the secret for refunding the $\texttt{Multisig}_{i+1}$ address at $\texttt{Channel}_{i,n-1}$ state is $\Phi_{i+1,n-1}$ and then:

$$\Phi_{i,n-1} \neq \Phi_{i+1,n-1}$$

Game theory is not sufficient to ensure the security of the channel if, when a player acts dishonestly, there exists an incentive to gain, even probabilistically, over the other player. In this case, the provider loses funds by broadcasting the $\texttt{Channel}_{i,n-1}$ state but can gain all funds if the client does not act correctly and does unlock his funds.

5 Further Improvements

Improvements can be done in two ways: (i) extending channel capabilities or (ii) optimizing the channel costs by reducing the transaction size or their number.

5.1 Threshold Signatures

The ability to settle and withdraw the channel without closing has a downside, each time, a transaction is broadcast on chain which costs fees. Optimizing the channel transaction size or the number of transactions needed is an area of further research.

The principal cost of a transaction comes from it inputs and their type. A channel transaction spends one or more `UTXOs` from the $\texttt{Multisig}_i$ address. These `UTXOs` are `P2SH` of a Bitcoin 2-out-of-2 multi-signature script that requires, obviously, two signatures. Knowing that a signature size is at least 64 bytes and an average transaction size (one simple input and one or two outputs) is a bit

more than 200 bytes, it is easy to see that replacing the `P2SH` with a 2-out-of-2 multisig `UTXOs` by `P2PKH` `UTXOs` is more efficient in any cases.

To achive this, an ECDSA threshold signature scheme, with the same requirements as the 2-out-of-2 multisig, is required. This scheme exists and can be adapted from the paper "Two-Party Generation of DSA Signatures" by MacKenzie and Reiter [4].

5.2 Pre-authorized Payments

Pre-authorized payments are required in other real case scenarios such as provider acting as a payment processor. The client must be able to set a limit within which the provider can take money during a given time period.

Further research can be done in this area to figure out the achievability and the most effective way to implement this feature in this scheme. Maybe a third layer, on top of layer two, is necessary and achievable, maybe the channel dimension can be increased.

6 Related Work

Simple micropayment channels were introduced by Hearn and Spilman [3]. The Lightning Network by Poon and Dryja [6], also creates a duplex micropayment channel. However it requires exchanging keying material for each update in the channels, which results in either massive storage or computational requirements in order to invalidate previous transactions. Finally, Decker and Wattenhofer introduced a payment network with duplex micropayment channels [2].

7 Conclusion

Trustless one-way payment channels for Bitcoin resolve many problems. Scalability is near infinite and costs of the channel decrease linearly with the number of transactions in the channel. Delays to consider a transaction as valid are brought back to network delay and minimal check time. Clients do not need to be online to keep their funds safe in the channel and can withdraw arbitrary amounts and refill the channel at any time. The provider does not need to lock funds to receive money and has no cost to setup a channel with a client.

We describe an unidirectional channel scheme trustless, endless, pulseless, and undelayed, but not optimal following the previous definitions. The lack of optimality is due to the intermediary revocation state, where refunding takes $\mathcal{T}(\mathcal{C}) = 2$ instead of $\mathcal{M}(\mathcal{C}) = 1$.

8 Acknowledgement

Loan Ventura and Thomas Roulin are acknowledged for their helpful contribution and comments during the completion of this work.

References

[1] Ryan X. Charles and Clemens Ley. *Yours Lightning Protocol*. 2016. URL: https://github.com/yoursnetwork/yours-channels/blob/master/docs/yours-lightning.md.

[2] Christian Decker and Roger Wattenhofer. "A Fast and Scalable Payment Network with Bitcoin Duplex Micropayment Channels". In: *Stabilization, Safety, and Security of Distributed Systems*. Ed. by Andrzej Pelc and Alexander A. Schwarzmann. Cham: Springer International Publishing, 2015, pp. 3–18. ISBN: 978-3-319-21741-3.

[3] Mike Hearn and Jeremy Spilman. *Bitcoin contracts*. URL: https://en.bitcoin.it/wiki/Contracts.

[4] Philip MacKenzie and Michael K. Reiter. "Two-Party Generation of DSA Signatures". In: *Advances in Cryptology — CRYPTO 2001*. Ed. by Joe Kilian. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 137–154. ISBN: 978-3-540-44647-7.

[5] Satoshi Nakamoto. *Bitcoin: A peer-to-peer electronic cash system*. 2009. URL: http://bitcoin.org/bitcoin.pdf.

[6] Joseph Poon and Thaddeus Dryja. "The bitcoin lightning network: Scalable off-chain instant payments". In: *draft version 0.5* 9 (2016).

List of Figures

2.1 Merkle tree construction 5
2.2 A chain of transactions where inputs and outputs are linked 6
2.3 Example of simple Bitcoin script program execution 7
2.4 Example of pay to public key hash script 8

4.1 Adapted protocol for ECDSA 23
4.2 The proof Π adapted 24
4.3 Adaptation of the verification of Π in ECDSA 24
4.4 Adaptation of the construction of Π in ECDSA 25
4.5 The proof Π′ adapted 26
4.6 Adaptation of the construction of Π′ in ECDSA 27
4.7 Adaptation of the verification of Π′ in ECDSA 28

List of Tables

3.1 Summary of different payment channels 13
3.2 Summary of transaction size optimization 14

4.1 Mapping between the protocol's variable names and the ZKP Π 24
4.2 Mapping between the protocol's variable names and the ZKP Π' 25

List of sources

3.1 Locking script (scriptPubKey) with `CHECKLOCKTIMEVERIFY` 12

4.1 Result of Python proof-of-concept threshold HD wallet 31
4.2 Demonstration of using threshold HD wallet 32
4.3 Construction of a share for a threshold HD wallet 33

5.1 Add argument in `configure.ac` to enable the module 37
5.2 Define constant `ENABLE_MODULE_THRESHOLD` if module enable 37
5.3 Include implementation headers if `ENABLE_MODULE_THRESHOLD` is defined 37
5.4 Set threshold module to experimental in `configure.ac` 38
5.5 Include specialized Makefile if threshold module is enabled 38
5.6 Specialized Makefile for threshold module 38
5.7 Implementation of a `DER` length parser 39
5.8 Implementation of a `DER` sequence parser 40
5.9 Implementation of a `DER` sequence serializer 40
5.10 Implementation of a `DER` length serializer 41
5.11 `DER` schema of a Paillier public key 41
5.12 `DER` parser of a Paillier public key 42
5.13 `DER` schema of a Paillier private key 42
5.14 `DER` parser of a Paillier private key 42
5.15 `DER` schema of an encrypted message with Paillier cryptosystem 43
5.16 Implementation of encryption with Paillier cryptosystem 43
5.17 Function signature for Paillier nonce generation 43
5.18 Implementation of decryption with Paillier cryptosystem 44
5.19 Implementation of homomorphic addition with Paillier cryptosystem . . 44
5.20 Implementation of homomorphic multiplication with Paillier cryptosystem 45
5.21 `DER` schema of a Zero-Knowledge parameters sequence 45
5.22 `DER` schema of a Zero-Knowledge Π sequence 46
5.23 `DER` schema of a Zero-Knowledge Π' sequence 46
5.24 Function signature for ZKP CPRNG 47
5.25 Function signature to generate ZKP Π 47
5.26 Function signature to generate ZKP Π' 48
5.27 Function signature to validate ZKP Π and Π' 48
5.28 Implementation of `call_create` function 50
5.29 Implementation of `call_received` function 51
5.30 Implementation of `challenge_received` function 51
5.31 Core function of `response_challenge_received` 53
5.32 Core function of `terminate_received` 54

A.1 Main file of threshold ECDSA proof-of-concept 63

Bibliography

[1] Eric Lombrozo, Johnson Lau, and Pieter Wuille. *Segregated Witness*. URL: https://github.com/bitcoin/bips/blob/master/bip-0141.mediawiki (visited on 02/02/2018).

[2] Joseph Poon and Thaddeus Dryja. "The bitcoin lightning network: Scalable off-chain instant payments". In: *draft version 0.5* 9 (2016).

[3] Andreas M. Antonopoulos. *Mastering Bitcoin: Unlocking Digital Crypto-Currencies*. 1st. O'Reilly Media, Inc., 2014. ISBN: 1449374042, 9781449374044.

[4] *SegWit*. URL: https://en.wikipedia.org/wiki/SegWit (visited on 02/02/2018).

[5] Jeremy Spilman. *[Bitcoin-development] Anti DoS for tx replacement*. 2013. URL: https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2013-April/002433.html (visited on 02/04/2018).

[6] Pieter Wuille. *Dealing with malleability*. 2014. URL: https://github.com/bitcoin/bips/blob/master/bip-0062.mediawiki (visited on 02/04/2018).

[7] Marcin Andrychowicz et al. "How to deal with malleability of BitCoin transactions". In: *CoRR* abs/1312.3230 (2013). arXiv: 1312.3230. URL: http://arxiv.org/abs/1312.3230.

[8] Christian Decker and Roger Wattenhofer. "Bitcoin Transaction Malleability and MtGox". In: *CoRR* abs/1403.6676 (2014). arXiv: 1403.6676. URL: http://arxiv.org/abs/1403.6676.

[9] Peter Todd. *CHECKLOCKTIMEVERIFY*. 2014. URL: https://github.com/bitcoin/bips/blob/master/bip-0065.mediawiki (visited on 02/05/2018).

[10] Christian Decker and Roger Wattenhofer. "A Fast and Scalable Payment Network with Bitcoin Duplex Micropayment Channels". In: *17th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS), Edmonton, Canada*. Aug. 2015.

[11] Ryan X. Charles and Clemens Ley. *Yours Lightning Protocol*. 2016. URL: https://github.com/yoursnetwork/yours-channels/blob/master/docs/yours-lightning.md.

[12] Dan Boneh and Matthew Franklin. "Efficient generation of shared RSA keys". In: *Advances in Cryptology — CRYPTO '97*. Ed. by Burton S. Kaliski. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 425–439. ISBN: 978-3-540-69528-8.

[13] Carmit Hazay et al. "Efficient RSA Key Generation and Threshold Paillier in the Two-Party Setting". In: *Topics in Cryptology – CT-RSA 2012*. Ed. by Orr Dunkelman. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 313–331. ISBN: 978-3-642-27954-6.

[14] Steven Goldfeder et al. "Securing Bitcoin wallets via a new DSA/ECDSA threshold signature scheme". In: 2015.

[15] Rosario Gennaro, Steven Goldfeder, and Arvind Narayanan. "Threshold-Optimal DSA/ECDSA Signatures and an Application to Bitcoin Wallet Security". In: *Applied Cryptography and Network Security - 14th International Conference, ACNS 2016, Guildford, UK, June 19-22, 2016. Proceedings*. 2016, pp. 156–174. DOI: 10.1007/978-3-319-39555-5_9. URL: https://doi.org/10.1007/978-3-319-39555-5_9.

[16] Philip D. MacKenzie and Michael K. Reiter. "Two-Party Generation of DSA Signatures". In: *Advances in Cryptology - CRYPTO 2001, 21st Annual International Cryptology Conference, Santa Barbara, California, USA, August 19-23, 2001, Proceedings*. Vol. 2139. Lecture Notes in Computer Science. Springer, 2001, pp. 137–154. DOI: 10.1007/3-540-44647-8_8.

[17] Pascal Paillier. "Public-key Cryptosystems Based on Composite Degree Residuosity Classes". In: *Proceedings of the 17th International Conference on Theory and Application of Cryptographic Techniques*. EUROCRYPT'99. Prague, Czech Republic: Springer-Verlag, 1999, pp. 223–238. ISBN: 3-540-65889-0. URL: http://dl.acm.org/citation.cfm?id=1756123.1756146.

[18] *SEC 2: Recommended Elliptic Curve Domain Parameters*. Version 2. 2010.

[19] Pieter Wuille. *Hierarchical Deterministic Wallets*. 2017. URL: https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki (visited on 02/01/2018).

[20] Thomas Pornin. *Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA)*. RFC 6979. Aug. 2013. DOI: 10.17487/RFC6979. URL: https://rfc-editor.org/rfc/rfc6979.txt.

[21] Pascal Paillier. "Trapdooring Discrete Logarithms on Elliptic Curves over Rings". In: *Advances in Cryptology — ASIACRYPT 2000*. Ed. by Tatsuaki Okamoto. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 573–584. ISBN: 978-3-540-44448-0.

[22] Nir Bitansky et al. "From Extractable Collision Resistance to Succinct Non-interactive Arguments of Knowledge, and Back Again". In: *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*. ITCS '12. Cambridge, Massachusetts: ACM, 2012, pp. 326–349. ISBN: 978-1-4503-1115-1. DOI: 10.1145/2090236.2090263. URL: http://doi.acm.org/10.1145/2090236.2090263.

[23] Eli Ben-Sasson et al. *Zerocash: Decentralized Anonymous Payments from Bitcoin*. Cryptology ePrint Archive, Report 2014/349. https://eprint.iacr.org/2014/349. 2014.

[24] Benedikt Bünz et al. *Bulletproofs: Efficient Range Proofs for Confidential Transactions*. Cryptology ePrint Archive, Report 2017/1066. https://eprint.iacr.org/2017/1066. 2017.

[25] Marek Palatinus et al. *Mnemonic code for generating deterministic keys*. 2013. URL: https://github.com/bitcoin/bips/blob/master/bip-0039.mediawiki (visited on 02/01/2018).

[26] C. P. Schnorr. "Efficient Identification and Signatures for Smart Cards". In: *Advances in Cryptology — CRYPTO' 89 Proceedings*. Ed. by Gilles Brassard. New York, NY: Springer New York, 1990, pp. 239–252. ISBN: 978-0-387-34805-6.

[27] Mihir Bellare and Phillip Rogaway. "Random Oracles Are Practical: A Paradigm for Designing Efficient Protocols". In: *Proceedings of the 1st ACM Conference on Computer and Communications Security*. CCS '93. Fairfax, Virginia, USA: ACM, 1993, pp. 62–73. ISBN: 0-89791-629-8. DOI: 10.1145/168588.168596. URL: http://doi.acm.org/10.1145/168588.168596.

[28] Yannick Seurin. "On the Exact Security of Schnorr-Type Signatures in the Random Oracle Model". In: *Advances in Cryptology – EUROCRYPT 2012*. Ed. by David Pointcheval and Thomas Johansson. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 554–571. ISBN: 978-3-642-29011-4.

Glossary

BIP Bitcoin Improvement Proposal. 9, 10, 29

DMC Duplex Micropayment Channels. 13

DSA Digital Signature Algorithm. 17, 18, 20–22, 34

EC Elliptic Curves. 18, 55

ECDSA Elliptic Curve Digital Signature Algorithm. 17, 18, 20–22, 34, 56, 57, 59

HTLC Hashed Timelock Contracts. 13

National Institute of Standards and Technology (NIST) is a unit of the U.S. Commerce Department. Formerly known as the National Bureau of Standards, NIST promotes and maintains measurement standards.. 18

P2PKH Pay To Public Key Hash. 8, 10, 14

P2SH Pay To Script Hash. 9, 14

SegWit Segregated Witness. 9, 10, 13, 14

SPV Simplified Payment Verification. 5

Standards for Efficient Cryptography Group (SECG) is an international consortium founded by Certicom in 1998. The group exists to develop commercial standards for efficient and interoperable cryptography based on elliptic curve cryptography (ECC). 18