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 6, 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 and Daniel Lebrecht for their helpful contribution and rereading of this work.

Abstract

Bitcoin is a decentralized peer-to-peer currency that allows users to pay for things electronically. Created by a pseudonymous software developer going by the name of Satoshi Nakamoto in 2008, as an electronic payment system based on mathematical proof, Bitcoin can currently handle only some transactions per second on the network. Thus, the most substantial challenge for the coming years is scalability. This amount is not sufficient in comparison to big payment infrastructures, which allows 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. The paper present, analyze and optimize with threshold cryptography a unidirectional payment channel specific for retail, commercial transactions scenario. The threshold scheme selected is adapted and implemented into the Bitcoin cryptographic library to compute a particular two-party threshold ECDSA signature.

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

Contents

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 interesting from a political, economic or technical point of view. We can also mention Ethereum, ZCash, or Monero for example. Created by a pseudonymous software developer going by the name of Satoshi Nakamoto in 2008, as an electronic payment system based on mathematical proof, Bitcoin as the idea to produce a means of exchange, independent of any central authority, 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 in peak times such as 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 a second layer, this paper proposes a new implementation of a unidirectional payment channel for retail, commercial transactions scenario. A unidirectional payment channel 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 this schemes significantly. Finding such a threshold scheme that fulfill the requirements is not trivial. The threshold scheme selected is adapted and 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, herself 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 sent 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. 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 broadcasted.

In the following some building blocks needed to figure out how payment channels works and how we can improve them with some cryptography are traveled. 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 be just a reminder. For a further explanation, the best resource today is the book "Mastering Bitcoin" by Andreas Antonopoulos [1].

Contents

2.1 The blockchain

The blockchain, as indicated by his name, is a chain of blocks. Blocks are created by the miners in a race to find the next valid block called the mining process. A block is valid if its identifier, i.e., the double hash of its header, is lower to the current difficulty target. It is worth noting that the validity of a block is based on multiple other criteria which are not exposed 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, or the difficulty target used as the boundary.

The difficulty target is adjusted, so a valid block is found in the network every ten minutes on average. Poisson process models the mining probability, 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 notices 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. Finding the next valid block is a computation that requires an enormous amount of power. All the network, round the clock, keeps searching for the next valid block and the power of computation increase day after days.

2.1.1 A chain of blocks

As mentioned before, the blockchain is a chain who 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 into the chain 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. 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 there is the same probability of finding a valid block with the first nonce than with the thousandth.

2.1.2 A list of transactions

To be useful a block needs a content. In Bitcoin, transactions compose the content of a block. As mentioned before, a transaction is called *confirmed* when she is included in a block. The number of confirmation, 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.

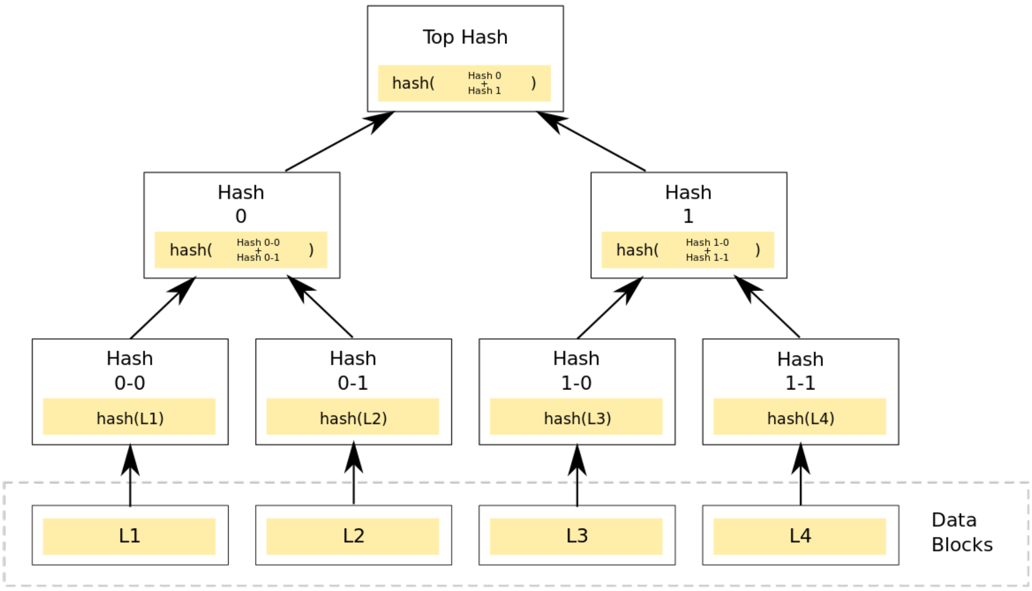

Figure 2.1 *Merkle tree construction*
Source: ***https://en.wikipedia.org/wiki/Merkle_tree***

Given the top hash or the Merkle root and a leaf, it is possible to prove the membership by giving the path for each complementary hashes. 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.

2.2 Transactions

Transactions allow users to move Bitcoins from an address to another and they create the content of the 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. Input refers to an address where the funds will be spent, and output refers to an address where the funds will go. An input points to another transaction output which has not been used already. Inputs and outputs are links, to spend funds the user needs to control addresses where unspent outputs are present. These unspent outputs are called `UTXOs` and represent the total amount own by a user.

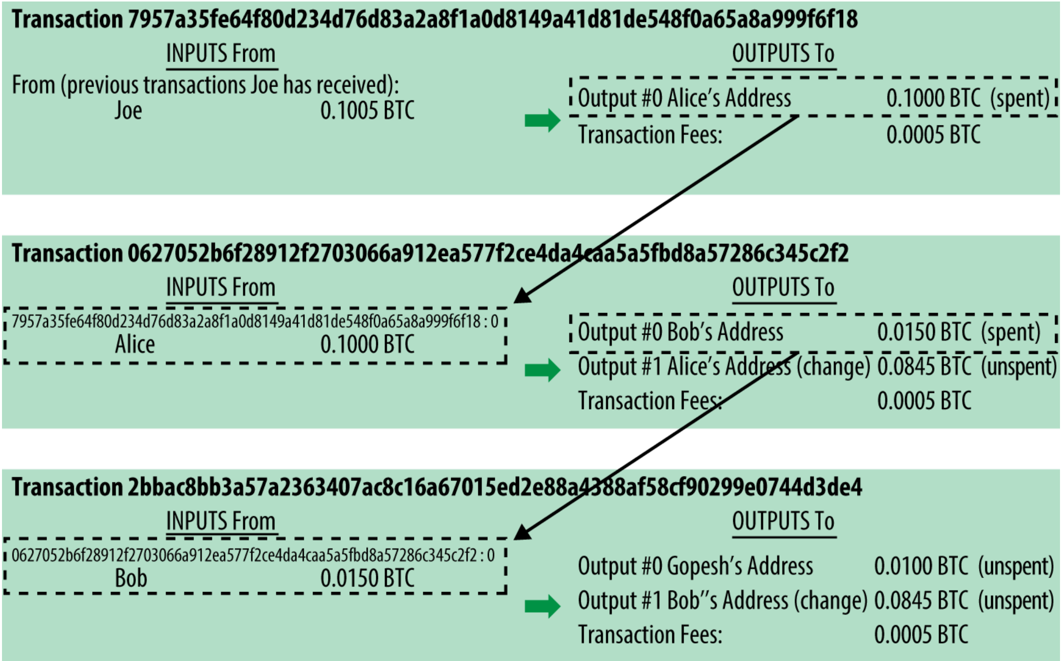

Figure 2.2 *A chain of transactions where inputs and outputs are linked*
Source: *https://github.com/bitcoinbook/bitcoinbook/blob/second_edition/ch02.asciidoc*

Each input is attached to a value (the value specified in the output to which the input points). The sum of all inputs in a transaction must correspond to the sum of all the values specified in the outputs, as in a dual entry accounting. So an input must be spent entirely.

So, the most straightforward transaction is composed of one input and one output with the same amount of money in and out. However, the most typical transaction is composed of one input, referring to where the funds come from, and two outputs. Indeed, it is scarce to have the right amount available in one `UTXO`. So, 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 blocks, transactions have an identifier. These identifiers are created in the same way as blocks, by taking the double hash of the data, i.e., the signed transaction. That means that, in the original design, a transaction does not have its final transaction identifier (TXID) before she is wholly signed (every input).

2.2.2 Transaction fees

The sum of all inputs in a transaction must correspond to the sum of all the values specified in the outputs, yes but if the sum of all outputs is lower than the sum of the inputs, the difference is implicitly considered as a fee (as shown in Figure 2.2.) No fee was required in the beginning, but today a transaction will not be 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 more fees concerning the amount of work needed to validate them. Fees are calculated with the size of the transaction in bytes, and a ratio of fee per byte 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 achieve the spending the Bitcoin protocol uses digital signatures. To spend a `UTXO`, a valid signature for to the address, and so the public key is required and to sign a transaction 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".

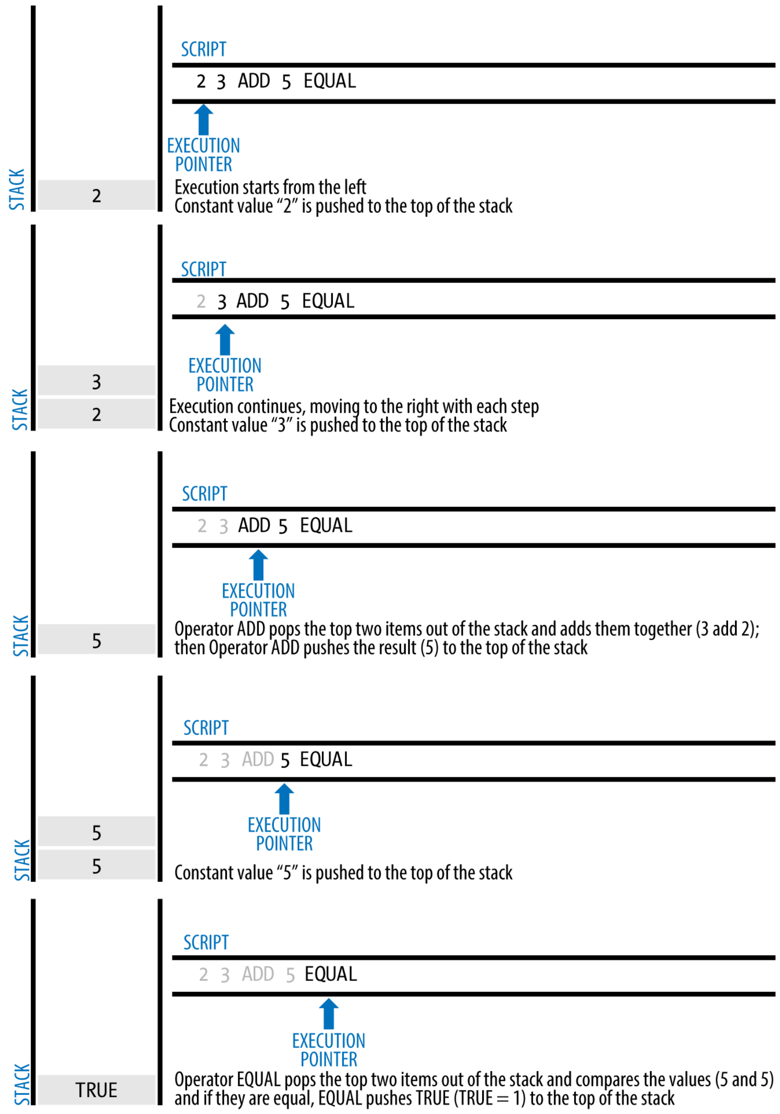

Figure 2.3 Example of simple Bitcoin script program execution
Source: *https://github.com/bitcoinbook/bitcoinbook/blob/second_edition/ch06.asciidoc*

The list of all the OP_CODES available in the Bitcoin script language is in the documentation. Among them, OP_CHECKSIG verify a given signature with a public key provided onto the stack, OP_IF, OP_ELSE, OP_ENDIF create execution branch with a boolean onto the stack, OP_DUP duplicates the value onto the stack or OP_HASH160, OP_SHA256 compute hashes of values onto the stack.

Each input and output have a script. For output, the script corresponds to the requirement to be fulfilled to be allowed 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 data 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 corresponding will be able to sign the transaction and spend the funds. The user owning the address can create a transaction such as input points to the UTXO. 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 pointed output is a UTXO, so if the pointed output is *unspent*, and execute the locking script with the unlocking script. Both scripts are concatenated to validate input; the unlocking script first (as shown in Figure 2.4).

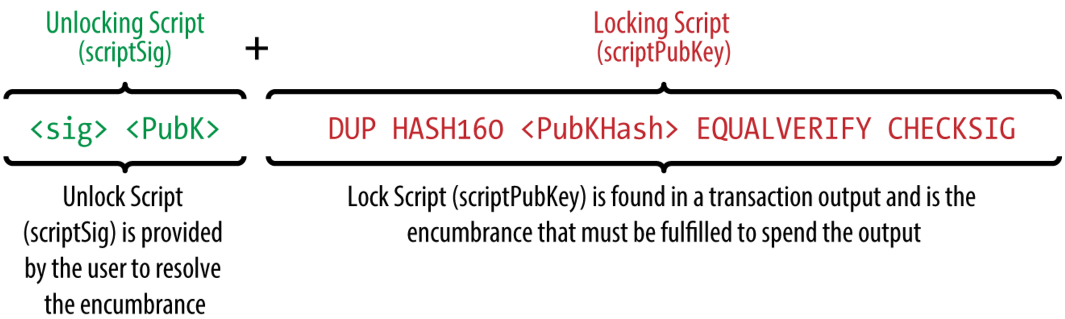

Figure 2.4 *Example of pay to public key hash script*
Source: *https://github.com/bitcoinbook/bitcoinbook/blob/second_edition/ch06.asciidoc*

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, and hashed, the public key hash present in the locking script is put on top of the stack, and the two first element on top of the stack are then compared. If the comparison failed, the script would fail and the transaction is rejected. If the test pass, the signature will be checked with the two remaining parameters onto the stack: the public key and the signature. If the signature is valid, the value True is put on top of the stack. Otherwise, the script put the value False 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.

2.2.4 Segregated witness

Segregated Witness (SegWit) is the Bitcoin Improvement Proposal (BIP) 141 that propose to change the transaction structure to fix transaction malleability, add script versioning, and improve other aspects [2, 3]. In fact, SegWit change 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 the purpose of SegWit:

> 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 TXID. The first is determine without all the witness data, so it is deterministic at the transaction creation. The second one is related to the witness data and change when signatures appear. This separation fixes the transaction malleability. The second big change is the way the size is calculated to determine the fees. The transaction weight `Tx Weight` becomes `Base Tx` $* 3 +$ `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. With this new structure the fees calculation change. The new structure introduce a virtual transaction size such as virtual size is equal to `Tx Weight`/4. Thus, SegWit reduce the weight of the witness data in the calculus of the fees.

2.2.5 Transaction malleability

The fact that transaction identifier depends on the hash of the serialized transaction while the signature does not currently cover all the data in a transaction introduce what is called transaction malleability. Indeed, a miner can 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:

> 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 on 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 the process can be painful. 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 often exchange.

3 Payment channels, a micropayment network

Payment channels or micropayment channels, as mentioned previously, are one part of the scalability solution. Thus, it exists various propositions to construct such structures. To have a better understanding of differences, strengths, and weaknesses of some of them the author proposes a classification and definitions 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 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 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* $\mathcal{P}_1 \rightarrow \mathcal{P}_2$*, 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* $\mathcal{P}_1 \rightarrow \mathcal{P}_2$ *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. It is worth noting that the list does not contain all the payment channel propositions and maybe some of them are missing. However, the list contains a fairly good representation of the different types existing.

Contents

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 into a channel and the bidirectional channel that allow two users to exchange into a channel. Usually, a bidirectional channel is more optimized than two unidirectional channels.

3.1.1 Unidirectional

In unidirectional channels, there is a payer, in the future also player-one or client, and a payee, after this also player-two or provider, and it is not possible to transfer money back in the reverse direction into the channel.

Spilman-style payment channels

Spilman-style payment channels, proposed by Jeremy Spilman in 2013 [4], 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 its 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 his funds, the canal is closing. Nor the payer or the receiver 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 [3, 5, 6, 7], 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 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` latter or equal to the specified value in the script [8].

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 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.

CLTV-style payment channels have the same properties than 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 exchange 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.

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

Listing 3.1 Locking script (scriptPubKey) with `CHECKLOCKTIMEVERIFY`

Decker-Wattenhofer duplex payment channels

Decker-Wattenhofer duplex payment channels [9], also called Duplex Micropayment Channels (DMC), propose in 2015 bidirectional channels based on pairs of Spilman-style unidirectional channels. The construction has a finite lifetime predefined at the setup phase but can be refresh 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. They are not pulseless (it requires a refresh transaction when the lifetime run over to keep the channel open) and not undelayed (without 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 [10].

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	Pulseless	Undelayed
Spilman-style	Uni	Yes	No	No	No
CLTV-style	Uni	Yes	No	No	No
Decker-Wattenhofer DMC	Bi	No	Yes	No	No
Poon-Dryja	Bi	No	Yes	Yes	Yes
Shababi-Gugger-Lebrecht	Uni	No	Yes	Yes	Yes

Table 3.1 Summary of different payment channels

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

3.2 Our one-way channel (Shababi-Gugger-Lebrecht)

Our one-way Payment Channel for Bitcoin is a simplified version of the Lightning Network and “Yours Lightning Protocol” [10, 11]. The scheme is specially designed for a scenario clients to a provider, where the provider has multiple clients through multiple channels. The core design aims to be cheaper as possible for the provider while being flexible for settlement funds. The white paper describing more in-depth the core design is following the appendices.

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, and we fixed it.

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 repose 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 script with a standard P2PKH the gains should be attractive.

3.3 Optimizing payment channels

		Non-SegWit		SegWit		
		R-Size	O	R-Size	V-Size	O
Refund transaction	SH	100	0%	110	80	0%
	PKH	100	0%	110	80	0%

Table 3.2 Summary of transaction size optimization

4 ECDSA asymmetric threshold scheme

Threshold cryptography has been discussed for a long time already, many cryptographic schemes like RSA or Paillier [12, 13] exists, but the difficulty is to use them in real case application. Since Bitcoin become famous, people lose funds because they lost 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 slow down the adoption of threshold cryptosystem is the complexity of creating an efficient and flexible scheme for Elliptic Curve Digital Signature Algorithm (ECDSA). Recently, researchers focused on finding more efficient and more generic system, but fortunately, a protocol fulfilling perfectly the needs described in the previous chapter required to improve payment channel in Bitcoin already exists. Nevertheless, this scheme explains how to perform a Digital Signature Algorithm (DSA) threshold signature and not 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 cited before. They base their construction of a threshold signature scheme with DSA based on a simple multiplicative shared secret and homomorphic encryption to keep the individual values unknown by the other player. The homomorphic encryption used as an example in the paper and chose for the implementation is the Paillier cryptosystem [17]. The following chapter describes how to adapt the scheme form DSA to ECDSA and introduce some fundamental building blocks needed for a real case scenario like hierarchical deterministic threshold wallet or deterministic signatures.

Contents

4.1 Reminder

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

4.1.1 Elliptic curves

Bitcoin use EC cryptography for securing his transaction. 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 define 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 $b = 7$.

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

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

The curve order n define the number of elements (points) generated by the generator G on the curve. Exponentiation of the generator $g^a \mod p$ become 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 $\mathcal{O}$ represent 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, the tangent to the curve $\mathcal{E}$ at point P determine 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 multiply 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 $s \equiv k^{-1}(\mathtt{hash}(m) + xr) \pmod q : s \neq 0$
- 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 $u_1 \equiv \mathtt{hash}(m) \cdot w \pmod q$
- Calculate $u_2 \equiv rw \pmod q$
- Calculate $v \equiv (g^{u_1} y^{u_2} \pmod p) \pmod q$
- The signature is valide iff $v = r$

Elliptic Curve Digital Signature Algorithm

ECDSA is a variant of DSA which uses elliptic curve cryptography and require 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 $s \equiv k^{-1}(\mathtt{hash}(m) + xr) \pmod n : s \neq 0$
- 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 valide iff $r \equiv x_1 \pmod n$

Schemes' analysis

In (r, s) the computation of the part s remain the same in each signature scheme, the only difference for s is the modulus applied. In DSA the modulus q is took, i.e. the order of the generator g modulo p, while in ECDSA the modulus n is took, 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 *This statement* $a \equiv g^b \pmod p$ *is equivalent in term 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 researched this postulate.

4.2 Threshold scheme

The "Two-party generation of DSA signatures" scheme presented by MacKenzie and Reiter, as mentioned before, 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 on 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.

As also mentioned before, the scheme corresponds to the same requirement of a Bitcoin 2-out-of-2 multi-signatures script. That means that it is possible to use it to improve the payment channels. However, it is necessary to adapt the scheme and particularly the zero-knowledge proofs construction. The following explains the chosen adaptation.

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 it is worth noting that others 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 amount of work on the signing protocol and because the implementation is not directly part of the cryptographic `C` library. This initialization can be further research for another thesis.

4.2.1 Adapting the scheme

Except for the zero-knowledge proofs, the adaptation is trivial and require just 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 remain 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 EC.

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 $x_2 z_2$.

The author noticed an error of notation in the original paper. On the second line Alice compute $z_1 \equiv (k_1)^{-1} \pmod n$ and the original paper use the random value selection $\xleftarrow{R}$ instead of a standard assignation $\leftarrow$. This error is corrected in the following version of the protocol.

alice	bob
$k_1 \xleftarrow{R} \mathbb{Z}_n$	
$z_1 \leftarrow (k_1)^{-1} \mod n$	
$\alpha \leftarrow E_{pk}(z_1)$	
$\zeta \leftarrow E_{pk}(x_1 z_1 \mod n)$	
	abort if$(\alpha \notin C_{pk} \vee \zeta \notin C_{pk})$
	$k_2 \xleftarrow{R} \mathbb{Z}_n$
	$\mathcal{R}_2 \leftarrow k_2 \cdot \mathcal{G}$
abort if$(\mathcal{R}_2 \notin \mathcal{E})$	
$\mathcal{R} \leftarrow k_1 \cdot \mathcal{R}_2$	
$\Pi \leftarrow \mathtt{zkp} \begin{bmatrix} \exists \eta_1, \eta_2 : & \eta_1, \eta_2 \in [-n^3, n^3] \\ \wedge & \eta_1 \cdot \mathcal{R} = \mathcal{R}_2 \\ \wedge & (\eta_2/\eta_1) \cdot \mathcal{G} = \mathcal{Q}_1 \\ \wedge & D_{sk}(\alpha) \equiv_n \eta_1 \\ \wedge & D_{sk}(\zeta) \equiv_n \eta_2 \end{bmatrix}$	
	abort if$(\mathcal{R} \notin \mathcal{E})$
	abort if$(\mathtt{Verifier}(\Pi) = 0)$
	$m' \leftarrow \mathtt{h}(m)$
	$r' \leftarrow x \mod n : (x, y) = \mathcal{R}$
	$z_2 \leftarrow (k_2)^{-1} \mod n$
	$c \xleftarrow{R} \mathbb{Z}_{n^5}$
	$\mu \leftarrow (\alpha \times_{pk} m' z_2) +_{pk}$ $(\zeta \times_{pk} r' x_2 z_2) +_{pk} E_{pk}(cn)$
	$\mu' \leftarrow E_{pk'}(z_2)$
	$\Pi' \leftarrow \mathtt{zkp} \begin{bmatrix} \exists \eta_1, \eta_2, \eta_3 : & \eta_1, \eta_2 \in [-n^3, n^3] \\ \wedge & \eta_3 \in [-n^5, n^5] \\ \wedge & \eta_1 \cdot \mathcal{R}_2 = \mathcal{G} \\ \wedge & (\eta_2/\eta_1) \cdot \mathcal{G} = \mathcal{Q}_2 \\ \wedge & D_{sk'}(\mu') \equiv_n \eta_1 \\ \wedge & D_{sk}(\mu) \equiv_n D_{sk}((\alpha \times_{pk} m' \eta_1) +_{pk} \\ & (\zeta \times_{pk} r' \eta_2) +_{pk} E_{pk}(n \eta_3)) \end{bmatrix}$
abort if$(\mu \notin C_{pk} \vee \mu' \notin C_{pk'})$	
abort if$(\mathtt{Verifier}(\Pi') = 0)$	
$s \leftarrow D_{sk}(\mu) \mod n$	
$r \leftarrow x \mod n : (x, y) = \mathcal{R}$	
publish $<r, s>$	

Figure 4.1 Adapted protocol for ECDSA

4.2.2 Adapting zero-knowledge proofs

Initially, the protocol designs proofs for the DSA architecture, so the values tested in the proofs are values in $\mathbb{Z}_q$. These values are used to create a challenge 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 serialize points in the long form, 65 bytes starting with `0x04`, and two 32 bytes 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 act correctly and have 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 $(k_1)^{-1} \cdot R = ((k_1)^{-1} k_1 k_2) \cdot G = k_2 \cdot G = R_2$.

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

Figure 4.2 *The proof* Π

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

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* Π*'s verification in ECDSA*

$$
\begin{array}{ll}
\alpha \xleftarrow{R} \mathbb{Z}_{n^3} & \delta \xleftarrow{R} \mathbb{Z}_{n^3} \\
\beta \xleftarrow{R} \mathbb{Z}_N^* & \mu \xleftarrow{R} \mathbb{Z}_N^* \\
\gamma \xleftarrow{R} \mathbb{Z}_{n^3\tilde{N}} & \nu \xleftarrow{R} \mathbb{Z}_{n^3\tilde{N}} \\
\rho_1 \xleftarrow{R} \mathbb{Z}_{n\tilde{N}} & \rho_2 \xleftarrow{R} \mathbb{Z}_{n\tilde{N}} \\
 & \rho_3 \xleftarrow{R} \mathbb{Z}_n \\
 & \epsilon \xleftarrow{R} \mathbb{Z}_n \\
 & \\
z_1 \leftarrow (h_1)^{x_1}(h_2)^{\rho_1} \mod \tilde{N} & z_2 \leftarrow (h_1)^{x_2}(h_2)^{\rho_2} \mod \tilde{N} \\
\mathcal{U}_1 \leftarrow \alpha \cdot \mathcal{C} & \mathcal{Y} \leftarrow (x_2 + \rho_3) \cdot \mathcal{D} \\
u_2 \leftarrow g^{\alpha}\beta^{N} \mod N^2 & \mathcal{V}_1 \leftarrow (\delta + \epsilon) \cdot \mathcal{D} \\
u_3 \leftarrow (h_1)^{\alpha}(h_2)^{\gamma} \mod \tilde{N} & \mathcal{V}_2 \leftarrow \alpha \cdot \mathcal{W}_2 + \epsilon \cdot \mathcal{D} \\
 & v_3 \leftarrow g^{\delta}\mu^{N} \mod N^2 \\
 & v_4 \leftarrow (h_1)^{\delta}(h_2)^{\nu} \mod \tilde{N}
\end{array}
$$

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

$$
\begin{array}{ll}
s_1 \leftarrow ex_1 + \alpha & t_1 \leftarrow ex_2 + \delta \\
s_2 \leftarrow (r_1)^{e}\beta \mod N & t_2 \leftarrow e\rho_3 + \epsilon \mod n \\
s_3 \leftarrow e\rho_1 + \gamma & t_3 \leftarrow (r_2)^{e}\mu \mod N^2 \\
 & t_4 \leftarrow e\rho_2 + \nu
\end{array}
$$

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

Figure 4.4 *Adaptation of* Π*'s construction 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_2z_2, and that the encrypted values μ and μ' are correctly composed.

$x_1 = z_2$	$\mathcal{C} = \mathcal{R}_2$
$x_2 = x_2z_2 \mod n$	$\mathcal{D} = \mathcal{G}$
$x_3 = c \mod n$	$\mathcal{W}_1 = \mathcal{G}$
$m_1 = \mu'$	$\mathcal{W}_2 = \mathcal{Q}_2$
$m_2 = \mu$	$m_3 = \alpha$
$m_4 = \zeta$	

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

$$\Pi' \leftarrow \texttt{zkp}\left[\begin{array}{ll} \exists x_1, x_2, x_3 : & x_1, x_2 \in [-n^3, n^3] \\ \wedge & x_3 \in [-n^5, n^5] \\ \wedge & x_1 \cdot \mathcal{C} = \mathcal{W}_1 \\ \wedge & (x_2/x_1) \cdot \mathcal{D} = \mathcal{W}_2 \\ \wedge & D_{sk'}(m_1) \equiv_n x_1 \\ \wedge \quad D_{sk}(m_2) \equiv_n D_{sk}((m_3 \times_{pk} m'x_1) +_{pk} & \\ \qquad (m_4 \times_{pk} r'x_2) +_{pk} E_{pk}(nx_3)) & \end{array}\right]$$

Figure 4.5 The proof Π'

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 $\mu = (\alpha)^{m'x_1}(\zeta)^{r'x_2} g^{nx_3}(r_2)^N$, then the equation v_3 in the validation proccess does not correspond to construction of v_3 in the original paper. The result in the verification proccess Π' need to match $v_3 \leftarrow (m_3)^\alpha (m_4)^\delta g^{n\sigma} \mu^N \mod N^2$. The original equation proposed $v_3 \leftarrow (m_3)^{s_1}(m_4)^{t_1} g^{nt_5}(t_3)^N (m_2)^{-e} \mod N^2$ does not include m' and r' present in μ, so m_2 cannot be used correctly as showed next.

$$\begin{aligned} v_3 &\equiv (m_3)^{s_1}(m_4)^{t_1} g^{nt_5}(t_3)^N (m_2)^{-e} \pmod{N^2} \\ &\equiv (m_3)^{ex_1+\alpha}(m_4)^{ex_2+\beta} g^{n(ex_3+\sigma)}\big((r_2)^e \mu\big)^N \big((m_3)^{m'x_1}(m_4)^{r'x_2} g^{nx_3}(r_2)^N\big)^{-e} \\ &\equiv (m_3)^{ex_1+\alpha}(m_4)^{ex_2+\beta} g^{n(ex_3+\sigma)}(r_2)^{eN}\mu^N (m_3)^{-em'x_1}(m_4)^{-er'x_2} g^{-enx_3}(r_2)^{-eN} \\ &\equiv (m_3)^{ex_1+\alpha-em'x_1}(m_4)^{ex_2+\beta-er'x_2} g^{enx_3+n\sigma-enx_3}(r_2)^{eN-eN}\mu^N \\ &\equiv (m_3)^{ex_1+\alpha-em'x_1}(m_4)^{ex_2+\beta-er'x_2} g^{n\sigma}\mu^N \end{aligned} \tag{4.3}$$

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 $s_4 \leftarrow ex_1x_4 + \alpha$ and $t_7 \leftarrow ex_2x_5 + \delta$ are added into the proof to correct the equation.

$$\begin{aligned} v_3 &\equiv (m_3)^{s_4}(m_4)^{t_7} g^{nt_5}(t_3)^N (m_2)^{-e} \pmod{N^2} \\ &\equiv (m_3)^{ex_1x_4+\alpha}(m_4)^{ex_2x_5+\beta} g^{n(ex_3+\sigma)}\big((r_2)^e \mu\big)^N \big((m_3)^{x_1x_4}(m_4)^{x_2x_5} g^{nx_3}(r_2)^N\big)^{-e} \\ &\equiv (m_3)^{ex_1x_4+\alpha}(m_4)^{ex_2x_5+\beta} g^{n(ex_3+\sigma)}(r_2)^{eN}\mu^N (m_3)^{-ex_1x_4}(m_4)^{-ex_2x_5} g^{-enx_3}(r_2)^{-eN} \\ &\equiv (m_3)^{ex_1x_4+\alpha-ex_1x_4}(m_4)^{ex_2x_5+\beta-ex_2x_5} g^{enx_3+n\sigma-enx_3}(r_2)^{eN-eN}\mu^N \\ &\equiv (m_3)^{\alpha}(m_4)^{\beta} g^{n\sigma}\mu^N \end{aligned} \tag{4.4}$$

$$
\begin{array}{ll}
\alpha \xleftarrow{R} \mathbb{Z}_{n^3} & \delta \xleftarrow{R} \mathbb{Z}_{n^3} \\
\beta \xleftarrow{R} \mathbb{Z}^*_{N'} & \mu \xleftarrow{R} \mathbb{Z}^*_{N} \\
\gamma \xleftarrow{R} \mathbb{Z}_{n^3\tilde{N}} & \nu \xleftarrow{R} \mathbb{Z}_{n^3\tilde{N}} \\
\rho_1 \xleftarrow{R} \mathbb{Z}_{n\tilde{N}} & \rho_2 \xleftarrow{R} \mathbb{Z}_{n\tilde{N}} \\
 & \rho_3 \xleftarrow{R} \mathbb{Z}_{n} \\
 & \rho_4 \xleftarrow{R} \mathbb{Z}_{n^5\tilde{N}} \\
 & \epsilon \xleftarrow{R} \mathbb{Z}_{n} \\
 & \sigma \xleftarrow{R} \mathbb{Z}_{n^7} \\
 & \tau \xleftarrow{R} \mathbb{Z}_{n^7\tilde{N}} \\
 & \\
z_1 \leftarrow (h_1)^{x_1}(h_2)^{\rho_1} \mod \tilde{N} & z_2 \leftarrow (h_1)^{x_2}(h_2)^{\rho_2} \mod \tilde{N} \\
\mathcal{U}_1 \leftarrow \alpha \cdot \mathcal{C} & \mathcal{Y} \leftarrow (x_2 + \rho_3) \cdot \mathcal{D} \\
u_2 \leftarrow (g')^{\alpha}\beta^{N'} \mod (N')^2 & \mathcal{V}_1 \leftarrow (\delta + \epsilon) \cdot \mathcal{D} \\
u_3 \leftarrow (h_1)^{\alpha}(h_2)^{\gamma} \mod \tilde{N} & \mathcal{V}_2 \leftarrow \alpha \cdot \mathcal{W}_2 + \epsilon \cdot \mathcal{D} \\
 & v_3 \leftarrow (m_3)^{\alpha}(m_4)^{\delta} g^{n\sigma}\mu^{N} \mod N^2 \\
 & v_4 \leftarrow (h_1)^{\delta}(h_2)^{\nu} \mod \tilde{N} \\
 & z_3 \leftarrow (h_1)^{x_3}(h_2)^{\rho_4} \mod \tilde{N} \\
 & v_5 \leftarrow (h_1)^{\sigma}(h_2)^{\tau} \mod \tilde{N}
\end{array}
$$

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

$$
\begin{array}{ll}
s_1 \leftarrow ex_1 + \alpha & t_1 \leftarrow ex_2 + \delta \\
s_2 \leftarrow (r_1)^{e}\beta \mod N' & t_2 \leftarrow e\rho_3 + \epsilon \mod n \\
s_3 \leftarrow e\rho_1 + \gamma & t_3 \leftarrow (r_2)^{e}\mu \mod N \\
s_4 \leftarrow ex_1x_4 + \alpha & t_4 \leftarrow e\rho_2 + \nu \\
 & t_5 \leftarrow ex_3 + \sigma \\
 & t_6 \leftarrow e\rho_4 + \tau \\
 & t_7 \leftarrow ex_2x_5 + \delta
\end{array}
$$

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

Figure 4.6 Adaptation of Π''s construction 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 $t_5 \in \mathbb{Z}_{n^7}$	$\mathcal{V}_2 \leftarrow s_1 \cdot \mathcal{W}_2 + t_2 \cdot \mathcal{D} + (-e) \cdot \mathcal{Y}$
$\mathcal{U}_1 \leftarrow s_1 \cdot \mathcal{C} + (-e) \cdot \mathcal{W}_1$	$v_3 \leftarrow (m_3)^{s_4} (m_4)^{t_7} g^{nt_5} (t_3)^N (m_2)^{-e} \mod N^2$
$u_2 \leftarrow (g')^{s_1} (s_2)^{N'} (m_1)^{-e} \mod (N')^2$	$v_4 \leftarrow (h_1)^{t_1} (h_2)^{t_4} (z_2)^{-e} \mod \tilde{N}$
$u_3 \leftarrow (h_1)^{s_1} (h_2)^{s_3} (z_1)^{-e} \mod \tilde{N}$	$v_5 \leftarrow (h_1)^{t_5} (h_2)^{t_6} (z_3)^{-e} \mod \tilde{N}$

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 Π' *verification to 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:

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

or multiplicatively:

$$\begin{aligned}
sk_{mas} &= \prod_{i=1}^{s} sk_i \bmod n \\
pk_{mas} &= \left[\prod_{i=1}^{s} sk_i \bmod n\right] \cdot G \\
&= (((G \cdot sk_1) \cdot sk_2) \dots) \cdot sk_i
\end{aligned}$$

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

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

The function $f(l)$ compute 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` compute 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. It is worth noting that every player updates the master public key for the threshold, not the public key share.

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

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

4.3.3 Child key share derivation

The protocol assumes that one of the players P_i is designated as the leader L. The function `CKSD` compute 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 use `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 $k < 2^{31}$:

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

such that:

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

and then $\forall i \in P_i$:

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

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 de derivation.

In this threshold HD scheme only one private share change at each derivation. In other words, the master private share is derived either with public information or with private information, i.e., *hardened* derivation. If the derivation is private, 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 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 (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 not anymore 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 proof-of-concept implemented in Python is made. A share can be tagged as master share as described previously. The result of the script is presented after that, three shares are created, and the first one is tagged as the master share. The root threshold public key `m/` is computed and display, then individual shares' addresses are displayed. The share s_1 is derived with and without *hardened* path, as expected the resulted address is different. The master public key resulting of each share derivations 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.

```
=== 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 using threshold HD wallet

A share is composed of four main 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 return a new share for a given index. The path derivation function `derive` take a path and generate the chain of shares. In this Implementation, if a share not tagged as master try 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.

```
if __name__ == "__main__":
    print("=== Threshold addresses ===")

    chain = ecdsa.gen_priv()
    # Shares
    s1 = Share(chain, True, ecdsa.gen_priv())
    s2 = Share(chain, False, ecdsa.gen_priv())
    s3 = Share(chain, False, ecdsa.gen_priv())

    sec = (s1.secret * s2.secret * s3.secret) % ecdsa.n
    pub = ecdsa.get_pub(sec)
    add = get(pub)
    print "Master root public key m/   :", add

    s1.set_master_pub(pub)
    s2.set_master_pub(pub)
    s3.set_master_pub(pub)

    print "\n*** Individual addresses m/ ***"
    print "s1:", s1.address()
    print "s2:", s2.address()
    print "s3:", s3.address()

    print "\n*** Hardened derivation for one share ***"
    print "s1 m/44/0/1  :", get(s1.derive("m/44/0/1").master_pub)
    print "s1 m/44/0/1' :", get(s1.derive("m/44/0/1'").master_pub)

    print "\n*** Master public key m/44/0/1 ***"
    s1 = s1.derive("m/44/0/1")
    s2 = s2.derive("m/44/0/1")
    s3 = s3.derive("m/44/0/1")
    print "s1:", get(s1.master_pub)
    print "s2:", get(s2.master_pub)
    print "s3:", get(s3.master_pub)

    sec = (s1.secret * s2.secret * s3.secret) % ecdsa.n
    pub = ecdsa.get_pub(sec)
    add = get(pub)
    print "\nMaster public key m/44/0/1 :", add

    print "\n*** Individual addresses m/44/0/1 ***"
    print "s1:", s1.address()
    print "s2:", s2.address()
    print "s3:", s3.address()
```

Listing 4.2 Demonstration of using threshold HD wallet

```python
class Share(object):
    def __init__(self, chain, master, secret=ecdsa.gen_priv()):
        super(Share, self).__init__()
        self.chain = chain
        self.master = master
        self.secret = secret
        self.master_pub = None

    def pub(self):
        return ecdsa.get_pub(self.secret)

    def address(self):
        return get(self.pub())

    def set_master_pub(self, pub):
        self.master_pub = pub

    def d_pub(self, i):
        if i >= pow(2, 31): # Only not hardened
            raise Exception("Impossible to hardened")
        k = "%x" % self.chain
        data = "00%s%08x" % (ecdsa.expand_pub(self.master_pub), i)
        hmac = hashlib.pbkdf2_hmac('sha256', k, data, 100)
        point = ecdsa.point_mult(self.master_pub, long(binascii.hexlify(hmac), 16))
        data = "%08x" % (i)
        hmac = hashlib.pbkdf2_hmac('sha256', k, data, 100)
        c = long(binascii.hexlify(hmac), 16)
        share = Share(c, self.master, self.secret)
        share.set_master_pub(point)
        return share

    def d_priv(self, i):
        k = "%x" % self.chain
        data = "%08x" % (i)
        hmac = hashlib.pbkdf2_hmac('sha256', k, data, 100)
        c = long(binascii.hexlify(hmac), 16)
        if i >= pow(2, 31): # Hardened
            data = "00%32x%08x" % (self.secret, i)
        else: # Not hardened
            data = "00%s%08x" % (ecdsa.expand_pub(self.master_pub), i)
        hmac = hashlib.pbkdf2_hmac('sha256', k, data, 100)
        key = long(binascii.hexlify(hmac), 16) * self.secret
        point = ecdsa.point_mult(self.master_pub, long(binascii.hexlify(hmac), 16))
        share = Share(c, self.master, key)
        share.set_master_pub(point)
        return share

    def d(self, index):
        if self.master:
            return self.d_priv(index)
        else:
            return self.d_pub(index)

    def derive(self, path):
        path = string.split(path, "/")
        if path[0] == "m":
            path = path[1:]
            share = self
            for derivation in path:
                if "'" in derivation:
                    i = int(derivation.replace("'", "")) + pow(2, 31)
                    share = share.d(i)
                else:
                    i = int(derivation)
                    share = share.d(i)
            return share
        else:
            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 worst, select a static value for k. This problem already appends to 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.

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:

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

However, this issue can be prevented by a deterministic generation of k, as described by the 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 we sign multiple times the same message, 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.

$$\begin{aligned}
k_1 &= \texttt{HMAC}(m, x_1) \\
k_2 &= \texttt{HMAC}(m, x_2) \\
k &= k_1 k_2 \mod n
\end{aligned}$$

The values k_1 and k_2 remain secret as well as the value x_1 and x_2 but the signature will always be the same for the given message and the threshold key.

5 Implementation in Bitcoin-core secp256k1

As mentionned before, Bitcoin use eliptic curve cryptography (ECC) for signing transactions. When the first release of Bitcoin core appeared in the early 2009, the cryptographic computations was performed with the OpenSSL library. Some years after a project started with the goal of replacing OpenSSL and creating a custom and minimalistic `C` library for cryptography over the curve secp256k1. This library is now available on GitHub at `bitcoin-core/secp256k1` project and it is one of the most optimized, if not the most optimized, library for the curve secp256k1. It is worth noting that this library is also used by other major crypto-currencies like Ethereum, so extending the capabilities of this library is a good choice to attract other cryptographer to have a look and increase the amount 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) an 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 need to be reviewed and more tested before been used in production. It is also worth noting that this library doesn't 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 when this lines are wrote. Note that the sources can evolve after that this report is written, to be sure to read the latest version of the code check out the sources directly on GitHub.

Contents

5.1 Configuration

The library use `autotools` to manage the compilation, installation and uninstallation. A system of module is already present in the structure with an ECDH experimental module for shared secret computation and a recovery module for recover ECDSA public key. A module can be flag as experimental, then, at the 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 a m4 macro defined by autoconf in the `configure.ac` file with the argument `--enable-module-threshold`. The default value is set to `no`.

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

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

If the variable `$enable_module_recovery` is set to yes into `configure.ac` (lines 443 to 445) a compiler constant is declared, again with a 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 is run 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 to experimental to avoid enabling it without explicitly agree to build experimental code. If experimental is set to yes a warning is display during the configuration process, if experimental is not set and any experimental module is enable an error message is display and the process failed.

```
if test x"$enable_experimental" = x"yes"; then
  AC_MSG_NOTICE([******])
  AC_MSG_NOTICE([WARNING: experimental build])
  AC_MSG_NOTICE([Experimental features do not have stable APIs or properties, and may not be safe for production use.])
  AC_MSG_NOTICE([Building ECDH module: $enable_module_ecdh])
  AC_MSG_NOTICE([Building Threshold module: $enable_module_threshold])
  AC_MSG_NOTICE([******])
else
  if test x"$enable_module_ecdh" = x"yes"; then
    AC_MSG_ERROR([ECDH module is experimental. Use --enable-experimental to allow.])
  fi
  if test x"$enable_module_threshold" = x"yes"; then
    AC_MSG_ERROR([Threshold module is experimental. Use --enable-experimental to allow.])
  fi
  if test x"$set_asm" = x"arm"; then
    AC_MSG_ERROR([ARM assembly optimization is experimental. Use --enable-experimental to allow.])
  fi
fi
```

Listing 5.4 Set threshold module to experimental into `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 functions, these headers are copied in the right folders when `sudo make install` command is run. The file `Makefile.am` define which headers need to be installed, which not and how to compile the project. This file is parsed by autoconf to generate the final `Makefile` with all the fonctionalities expected.

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

```
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 enable

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

```
include_HEADERS += include/secp256k1_threshold.h
noinst_HEADERS += src/modules/threshold/der_impl.h
noinst_HEADERS += src/modules/threshold/paillier.h
noinst_HEADERS += src/modules/threshold/paillier_impl.h
noinst_HEADERS += src/modules/threshold/paillier_tests.h
noinst_HEADERS += src/modules/threshold/eczkp.h
noinst_HEADERS += src/modules/threshold/eczkp_impl.h
noinst_HEADERS += src/modules/threshold/eczkp_tests.h
noinst_HEADERS += src/modules/threshold/threshold_impl.h
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 retreive keys are an important part of the scheme. Because between all steps a communication on the network is necessary, a way to export and import data is required. Bitcoin private key are simple structures because of the fixed curve and their intrinsic nature, a single 2^{256} bits value. Threshold private key are composed of multiple parts like: (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 has been choosed. 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 start with the constant `0x30` and is followed by the content lenght and the content itself. A lenght could be in the short form or the long form. If the content number of bytes is shorter to `0x80` the lenght byte indicate the lenght, if the content is equal or longer than `0x80` the seven lower bits 0 to 6 where $\texttt{byte} = \{\texttt{b}_7, \ldots, \texttt{b}_1, \texttt{b}_0\}$ indicate the number of followed bytes which are used for the lenght.

```c
void secp256k1_der_parse_len(const unsigned char *data, unsigned long *pos, unsigned long
 ↪  *lenght, unsigned long *offset) {
    unsigned long op, i;
    op = data[*pos] & 0x7F;
    if ((data[*pos] & 0x80) == 0x80) {
        for (i = 0; i < op; i++) {
            *lenght += data[*pos+1+i]<<8*(op-i-1);
        }
        *offset = op + 1;
    } else {
        *lenght = op;
        *offset = 1;
    }
    *pos += *offset;
}
```

Listing 5.7 Implementation of a `DER` *lenght parser*

The sequence parser check the first byte with the constant `0x30` and extract the content lenght. Position in the input array are holds in the `*pos` variable, extracted lenght is stored in `*lenght`, and the offset holds how many bytes in the data are used for the header and the lenght. A coherence check is performed to ensure that the current offset and the retreived lenght result to the same amount of bytes passed in argument.

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

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

```c
int secp256k1_der_parse_struct(const unsigned char *data, size_t datalen, unsigned long *pos, unsigned long *lenght, unsigned long *offset) {
    unsigned long loffset;
    if (data[*pos] == 0x30) {
        *pos += 1;
        secp256k1_der_parse_len(data, pos, lenght, &loffset);
        *offset = 1 + loffset;
        if (*lenght + *offset != datalen) { return 0; }
        else { return 1; }
    }
    return 0;
}
```

Listing 5.8 *Implementation of a `DER` sequence parser*

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

```c
unsigned char* secp256k1_der_serialize_sequence(size_t *outlen, const unsigned char *op, const size_t datalen) {
    unsigned char *data = NULL, *len = NULL;
    size_t lensize = 0;
    len = secp256k1_der_serialize_len(&lensize, datalen);
    *outlen = 1 + lensize + datalen;
    data = malloc(*outlen * sizeof(unsigned char));
    data[0] = 0x30;
    memcpy(&data[1], len, lensize);
    memcpy(&data[1 + lensize], op, datalen);
    free(len);
    return data;
}
```

Listing 5.9 *Implementation of a `DER` sequence serializer*

If the content lenght is longer than `0x80`, then `mpz` is used to serialize the lenght into a bytes array in big endian most significant byte first. The lenght of this serialization is stored into `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 the 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 start with `0x02`, followed by the lenght of the data. Parsing and serializing integer are already implemented in `libgmp`, functions are juste wrapper to extract information from the header and start the `mpz` importation at the right offset.

5.2.3 Octet string

Octet strings are used to holds serialized data like points/public keys. An octet string is an arbitrary array of bytes. The header start with `0x04` followed by the size of the content. The serialization implementation retreive the lenght of the content,

```c
unsigned char* secp256k1_der_serialize_len(size_t *datalen, size_t lenght) {
    unsigned char *data = NULL; void *serialize; size_t longsize; mpz_t len;
    if (lenght >= 0x80) {
        mpz_init_set_ui(len, lenght);
        serialize = mpz_export(NULL, &longsize, 1, sizeof(unsigned char), 1, 0, len);
        mpz_clear(len);
        *datalen = longsize + 1;
    } else {
        *datalen = 1;
    }
    data = malloc(*datalen * sizeof(unsigned char));
    if (lenght >= 0x80) {
        data[0] = (uint8_t)longsize | 0x80;
        memcpy(&data[1], serialize, longsize);
        free(serialize);
    } else {
        data[0] = (uint8_t)lenght;
    }
    return data;
}
```

Listing 5.10 *Implementation of a **DER** lenght serializer*

copy the header and the octet string into a new memory space, and return the pointer with the total lenght. The parser implementation copy the content and set the conent lenght, 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 need to be reviewed.

5.3.1 Data structures

Encrypted message, public and private keys are transmited. As mentionned before, the `DER` standard format is used to parse and serialize data. `DER` schema for all data structures are defined to ensure portability over different implementations.

Public keys

The public key is composed of a public modulus and a generator. The implementation data structure add 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 Paillier implementation, all numbers are stored in `mpz_t` type. The parser take in input an array of bytes with a lenght and the public key to fill.

```
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 exponant $\lambda = \varphi(n) = (p-1)(q-1)$, and a private coefficient $\mu = \varphi(n)^{-1} \bmod n$. Again, a version number is added for future compatibility purposes.

The parser take in input an array of bytes with a lenght and the private key to fill. The big modulus is computed after the parsing to accelerate encryption and decryption.

```
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*

```c
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 structure contain a nonce value that could be set to 0 to stores 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 messages 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 fonction pointer `noncefp` and set into the nonce value `res->nonce`. This nonce is stored because his value is needed to create Zero-Knowledge Proofs. Then, the cipher $c = g^m \cdot r^n \mod n^2$ is putted into `res->message` to complete the encryption process. All intermediray states are wipe out 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 failed the encryption fail also. The random function of type `secp256k1_paillier_nonce_function` must use a good CPRNG and his 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 nonces generation

To decrypt the cipher $c \in \mathbb{Z}^*_{n^2}$ with the private key, the function compute $m = L(c^\lambda \mod n^2) \cdot \mu \mod n$ 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 putted to an intermediray state variable. Then the $L(x)$ function is applied on the intermediray 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 composant $s = D_{sk}(\mu) \mod q : \mu = (\alpha \times_{pk} m'z_2) +_{pk} (\zeta \times_{pk} r'x_2z_2) +_{pk} E_{pk}(cn)$ 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 descryption 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 resulted encrypted message for 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 generate a proof and validates the other one. A proof is generated and verified under some ZKP parameters, these parameters are fixed at the initialization time and don't change over the 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 messages 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 inside the implementation and is exported with the secp256k1 library as a 65 bytes 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.

```
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*

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 than values to express all the proven statements. Again, the point y is a point serialized as 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 mathematic computations are needed with two `hash` functions.

A CPRNG function is required to generate both proofs. This function generate random number in $\mathbb{Z}_{max}$ and $\mathbb{Z}^*_{max}$. The `flag` argument indicate which case is treated, `STD` or `INV`. If the function have not 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) it exists a known value by the proover that link $r \to r_2$, (ii) it exists a second known value by the proover that, related to the first one, link $G \to y_1$, (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 need to be passed in argument, 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 splited 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) it exists a known value by the proover x_1 that link $r_2 \to G$, (ii) it 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) it exists a third known value by the proover x_3 and the result of $D_{sk}(\mu)$ is the homomorphic operation of $(\alpha \times x_1) + (\zeta \times x_2) + x_3$.

```
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'

The function implementation can also be splited 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 $e = \texttt{hash'}(v)$.

It is worth noting that `hash` and `hash'` must be different hashing function to avoid reusing Π proofs, even not 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 splited in three steps: compute the proof values, retreive 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 usefull to create an application that wants to use the distributed signature protocol. The public API includes all the function needed to parse-serialize keys, messages, and signature parameters. Signature parameters holds the values k, $z = k^{-1}$, and $r = k \cdot G$, these values are—in a normal signature mode—computed, used, and destroy in one time. However, a mechanisme to save et restore these values is required in the distributed mode because the context can be destroy and re-created between each steps.

The public API also includes the five functions that implement the protocol. One function is one step in the protocol and between two 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 messages names and actions is done in this report. Players P_1 and P_2 represent the initiator and collaborator. Player P_1 initialize the communication and ask P_2 to collaborate on a signature, if P_2 collaborates and the protocol end successfully P_1 retreive the signature.

Four messages are necessary between the five steps. In order, the proposed name 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 his name, create 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 bytes array and the nonce (k) is retreived with the `noncefp` function pointer. It is worth noting that this function could be call multiple times until a nonce that is not zero and which doesn't overflow is found. However, this function as a limited number of calls and if the limit is reached the function will fail. The signatures 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 signature mode. If the deterministic mode is choosed, the counter indicates the number of round done by the function.

```
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) {
    int ret = 0;
    int overflow = 0;
    unsigned char nonce32[32];
    unsigned char sec32[32];
    unsigned int count = 0;
    secp256k1_scalar privinv;

    ARG_CHECK(ctx != NULL);
    ARG_CHECK(callmsg != NULL);
    ARG_CHECK(params != NULL);
    ARG_CHECK(secshare != NULL);
    ARG_CHECK(paillierkey != NULL);
    ARG_CHECK(msg32 != NULL);
    secp256k1_scalar_get_b32(sec32, secshare);
    while (1) {
        ret = noncefp(nonce32, msg32, sec32, NULL, NULL, count);
        if (!ret) {
            break;
        }
        secp256k1_scalar_set_b32(&params->k, nonce32, &overflow);
        if (!overflow && !secp256k1_scalar_is_zero(&params->k)) {
            secp256k1_scalar_inverse(&params->z, &params->k); /* z1 */
            secp256k1_scalar_mul(&privinv, &params->z, secshare); /* x1z1 */
            if (secp256k1_paillier_encrypt_scalar(callmsg->alpha, &params->z, paillierkey, pnoncefp)
                && secp256k1_paillier_encrypt_scalar(callmsg->zeta, &privinv, paillierkey, pnoncefp)) {
                break;
            }
        }
        count++;
    }
    memset(nonce32, 0, 32);
    memset(sec32, 0, 32);
    secp256k1_scalar_clear(&privinv);
    return ret;
}
```

Listing 5.28 Implementation of `call_create` function

5.5.2 Receive call message

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

```c
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) {
    int ret = 0;
    int overflow = 0;
    unsigned int count = 0;
    unsigned char k32[32];
    unsigned char sec32[32];

    ARG_CHECK(ctx != NULL);
    ARG_CHECK(challengemsg != NULL);
    ARG_CHECK(params != NULL);
    ARG_CHECK(callmsg != NULL);
    ARG_CHECK(secshare != NULL);
    ARG_CHECK(msg32 != NULL);
    secp256k1_scalar_get_b32(sec32, secshare);
    while (1) {
        ret = noncefp(k32, msg32, sec32, NULL, NULL, count);
        if (!ret) {
            break;
        }
        secp256k1_scalar_set_b32(&params->k, k32, &overflow);
        if (!overflow && !secp256k1_scalar_is_zero(&params->k)) {
            if (secp256k1_ec_pubkey_create(ctx, &params->r, k32)) {
                memcpy(&challengemsg->r2, &params->r, sizeof(secp256k1_pubkey));
                break;
            }
        }
        count++;
    }
    memset(k32, 0, 32);
    memset(sec32, 0, 32);
    return ret;
}
```

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.

```c
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) {
    int ret = 0;
    unsigned char k32[32];
    secp256k1_pubkey y1;
    secp256k1_scalar privinv;

    ARG_CHECK(ctx != NULL);
    ARG_CHECK(respmsg != NULL);
    ARG_CHECK(params != NULL);
    ARG_CHECK(challengemsg != NULL);
    secp256k1_scalar_get_b32(k32, &params->k);
    memcpy(&respmsg->r, &challengemsg->r2, sizeof(secp256k1_pubkey));
    ret = secp256k1_ec_pubkey_tweak_mul(ctx, &respmsg->r, k32);
    secp256k1_scalar_get_b32(k32, secshare);
    if (ret && secp256k1_ec_pubkey_create(ctx, &y1, k32)) {
        memcpy(&params->r, &respmsg->r, sizeof(secp256k1_pubkey));
        secp256k1_scalar_mul(&privinv, &params->z, secshare);
        VERIFY_CHECK(secp256k1_eczkp_pi_generate(
            ctx,
            respmsg->pi,
            zkp,
            callmsg->alpha,
            callmsg->zeta,
            &params->z,
            &privinv,
            &params->r,
            &challengemsg->r2,
            &y1,
            paillierkey,
            rdnfp
        ) == 1);
    }
    memset(k32, 0, 32);
    secp256k1_scalar_clear(&privinv);
    return ret;
}
```

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 contain 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 get (modulo n). The `hash` is multiplied with z_2 and the coordinate $r.x$ is multiplied with $x_2 z_2$. 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 match the requirement as demonstrated below:

$$\begin{aligned}
D_{sk}(\mu) &\equiv (\alpha \times mz_2) + (\zeta \times rx_2 z_2) + (x_3) \pmod n \\
&\equiv (z_1 \times mz_2) + (x_1 z_1 \times rx_2 z_2) \pmod n \\
&\equiv (z_1 z_2 m) + (x_1 z_1 r x_2 z_2) \pmod n \\
&\equiv z_1 z_2 (m + r x_1 x_2) \pmod n \\
&\equiv z(m + rx) \pmod n \\
&\equiv k^{-1}(m + rx) \pmod n
\end{aligned}$$

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 asymetric. If P_2 also needs the signature, P_1 must share it. There is now way for P_2 to know the signature without a cooperative P_1.

```
    ret = secp256k1_eczkp_pi_verify(
        ctx,
        respmsg->pi,
        zkp,
        callmsg->alpha,
        callmsg->zeta,
        &respmsg->r,
        &challengemsg->r2,
        pairedshare,
        pairedkey
    );
    if (ret) {
        mpz_inits(m1, m2, c, n5, n, nc, m, z, rsig, inv, NULL);
        secp256k1_scalar_inverse(&params->z, &params->k); /* z2 */
        secp256k1_scalar_mul(&privinv, &params->z, secshare); /* x2z2 */
        mpz_import(n, 32, 1, sizeof(n32[0]), 1, 0, n32);
        secp256k1_scalar_set_b32(&msg, msg32, &overflow);
        if (!overflow && !secp256k1_scalar_is_zero(&msg)) {
            secp256k1_pubkey_load(ctx, &r, &respmsg->r);
            secp256k1_fe_normalize(&r.x);
            secp256k1_fe_normalize(&r.y);
            secp256k1_fe_get_b32(b, &r.x);
            secp256k1_scalar_set_b32(&sigr, b, &overflow);
            /* These two conditions should be checked before calling */
            VERIFY_CHECK(!secp256k1_scalar_is_zero(&sigr));
            VERIFY_CHECK(overflow == 0);
            mpz_import(rsig, 32, 1, sizeof(b[0]), 1, 0, b);
            secp256k1_scalar_get_b32(b, &params->z);
            mpz_import(z, 32, 1, sizeof(b[0]), 1, 0, b);
            secp256k1_scalar_get_b32(b, &privinv);
            mpz_import(inv, 32, 1, sizeof(b[0]), 1, 0, b);
            secp256k1_scalar_get_b32(b, &msg);
            mpz_import(m, 32, 1, sizeof(msg32[0]), 1, 0, msg32);
            mpz_mul(m1, m, z); /* m'z2 */
            mpz_mul(m2, rsig, inv); /* r'x2z2 */
            mpz_pow_ui(n5, n, 5);
            noncefp(c, n5);
            mpz_mul(nc, c, n); /* cn */
            secp256k1_paillier_mult(m3, callmsg->alpha, m1, pairedkey);
            secp256k1_paillier_mult(m4, callmsg->zeta, m2, pairedkey);
            secp256k1_paillier_add(m5, m3, m4, pairedkey);
            ret = secp256k1_paillier_encrypt_mpz(enc, nc, pairedkey, noncefp);
            secp256k1_scalar_get_b32(sec32, secshare);
            if (ret && secp256k1_ec_pubkey_create(ctx, &y2, sec32)) {
                secp256k1_paillier_add(termsg->mu, m5, enc, pairedkey);
                ret = secp256k1_paillier_encrypt_mpz(termsg->mu2, z, p2, noncefp);
                VERIFY_CHECK(secp256k1_eczkp_pi2_generate(
                    ctx,                /* ctx */
                    termsg->pi2,        /* pi2 */
                    zkp,                /* zkp */
                    termsg->mu2,        /* m1 */
                    termsg->mu,         /* m2 */
                    callmsg->alpha,     /* m3 */
                    callmsg->zeta,      /* m4 */
                    enc,                /* r */
                    z,                  /* x1 */
                    inv,                /* x2 */
                    c,                  /* x3 */
                    m,                  /* x4 */
                    rsig,               /* x5 */
                    &challengemsg->r2,  /* c */
                    &y2,                /* w2 */
                    pairedkey,          /* pairedkey */
                    p2,                 /* pubkey */
                    rdnfp               /* rdnfp */
                ) == 1);
            }
        }
```

Listing 5.31 Core function of `response_challenge_received`

```
    unsigned char n32[32] = {
        0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
         ↪   0xff, 0xfe,
        0xba, 0xae, 0xdc, 0xe6, 0xaf, 0x48, 0xa0, 0x3b, 0xbf, 0xd2, 0x5e, 0x8c, 0xd0, 0x36,
         ↪   0x41, 0x41
    };
    unsigned char b[32];
    void *ser;
    int ret = 0;
    int overflow = 0;
    size_t size;
    mpz_t m, n, sigs;
    secp256k1_ge sigr, pge;
    secp256k1_paillier_pubkey *p1;
    secp256k1_scalar r, s, mes;

    ARG_CHECK(ctx != NULL);
    ARG_CHECK(sig != NULL);
    ARG_CHECK(termsg != NULL);
    ARG_CHECK(params != NULL);
    ARG_CHECK(p != NULL);
    ARG_CHECK(pub != NULL);
    ARG_CHECK(msg32 != NULL);
    p1 = secp256k1_paillier_pubkey_get(p);
    ret = secp256k1_eczkp_pi2_verify(
        ctx,                        /* ctx */
        termsg->pi2,                /* pi2 */
        zkp,                        /* zkp */
        termsg->mu2,                /* m1 */
        termsg->mu,                 /* m2 */
        callmsg->alpha,             /* m3 */
        callmsg->zeta,              /* m4 */
        &challengemsg->r2,          /* c */
        pairedpub,                  /* w2 */
        p1,                         /* pubkey */
        pairedkey                   /* pairedkey */
    );
    if (ret) {
        secp256k1_scalar_set_b32(&mes, msg32, &overflow);
        ret = !overflow && secp256k1_pubkey_load(ctx, &pge, pub);
        if (ret) {
            secp256k1_pubkey_load(ctx, &sigr, &params->r);
            secp256k1_fe_normalize(&sigr.x);
            secp256k1_fe_normalize(&sigr.y);
            secp256k1_fe_get_b32(b, &sigr.x);
            secp256k1_scalar_set_b32(&r, b, &overflow);
            VERIFY_CHECK(!secp256k1_scalar_is_zero(&r));
            VERIFY_CHECK(overflow == 0);
            mpz_inits(m, n, sigs, NULL);
            secp256k1_paillier_decrypt(m, termsg->mu, p);
            mpz_import(n, 32, 1, sizeof(n32[0]), 1, 0, n32);
            mpz_mod(sigs, m, n);
            ser = mpz_export(NULL, &size, 1, sizeof(unsigned char), 1, 0, sigs);
            secp256k1_scalar_set_b32(&s, ser, &overflow);
            if (!overflow
                && !secp256k1_scalar_is_zero(&s)
                && secp256k1_ecdsa_sig_verify(&ctx->ecmult_ctx, &r, &s, &pge, &mes)) {
                secp256k1_ecdsa_signature_save(sig, &r, &s);
            } else {
                memset(sig, 0, sizeof(*sig));
            }
        }
        mpz_clears(m, n, sigs, NULL);
        secp256k1_scalar_clear(&r);
        secp256k1_scalar_clear(&s);
        secp256k1_scalar_clear(&mes);
    }
    secp256k1_paillier_pubkey_destroy(p1);
    return ret;
```

Listing 5.32 Core function of `terminate_received`

6 Further research

It is possible to list an enormous amount of idea or further research in a field like crypto-currencies or blockchain. But some of them 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 completely exploratory.

6.1 Side-channel attack resistant implementation and improvements

The proposed implementation into the library `secp256k1` rely on `libgmp` for all complex mathematical calculus and `libgmp` is not strong against side channel attacks, and it is normal, the library has not been developed for that particular purpose. Therefore, a other implementation need to take the place and handle, in constant time and constant memory if possible, the mathematical calculus part. This is a big 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 use the hash function `SHA256` implemented into the library `secp256k1` for Π and Π'. This is not complient with the orignial paper requirements, a other hash function must be implemented and used for Π'.

6.1.2 Paillier cryptosystem

Two major improvements or modifications could be performed 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 function L_p and L_q are define by

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

Decryption can therefore be perform over mod p and mod q and recombining modular residues afterwards:

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

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 nothing however that the curve construction is different than the curve used to sign and so the code base cannot can not necessarily be reused.

6.1.3 Zero-knowledge proofs

Non-interactive zero-knowledge proofs are a big 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 are 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 very short 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, to remember that the zero-knowledge proof construction proposed in the original paper dates from the early 2000s, progress has been made since.

6.2 Hardware wallets

Hardware wallet devices have become increasingly popular with people and society. They promise to keep the keys safe and, at least, expose less the keys thanks to a dedicated and controlled environment. Thus, keys can be stored safly and, in an organisation for exemple, multiple hardware wallets can be used to create a multi-signature and 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 real threshold hardware wallet devices. Two hardware wallet devices can be setup together to create a multi-user setup, or an hardware wallet device can be couple with a phone to secure a lightweight wallet.

Usually, when a new Bitcoin wallet is created, a list of words called mnemonic is shown to the user as a backup of his wallet key. The mnemonics are between twelve and twenty-four and each word represent 11 bits of the primary 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). A other way to display and transmit these information is needed to increase 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 to store this information or this information is a part of the setup and can be stored outside, this question can be deepened.

6.3 More generic threshold scheme

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

6.4 Schnorr signatures

In the paper "Efficient Identification and Signatures for Smart Cards" published in CRYPTO 1989, C.P. Schnorr propose 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 a strong interest for this specific scheme since some years now. Schnorr signatures could greatly reduce the size of the signaure from 65 bytes (ECDSA in `DER` format) to 40 bytes.

With the arrival of SegWit, it is now possible to have script version, thus it is more easy to indroduce new `OP_CODE` and so introduce a new signature validation scheme. However, this will not invalidate the present work and researches 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 fulfill the requirements defined for payment channels optimization.

7 | Conclusions

Basics mechanisms of Bitcoin have been explain to introduce two major issues in Bitcoin today, i.e. the scalability problem and the latency problem. These issues already have existing drafted solutions like consensus changes or payment channel. Payment channels are not new to Bitcoin and multiple implementations of the Lightning Network specification already exist. A other scheme, with different capabilities, is proposed to handle a specific context explain in the white paper. This scheme can be improve with threshold cryptography by reducing the size of the transactions. This reduction is done by replacing the multi-signature script in Bitcoin by a single signature computed with threshold cryptography. The threshold scheme is analysed and adapted to ECDSA before been implemented inside the existing library used in Bitcoin-core implementation, the library `secp256k1`. Finally, further research about payment channels, Bitcoin, and threshold signature scheme are exposed.

The Bitcoin payment channel implementation will be released in open source soon and testing will begin in the next months. The threshold implementation will be part of a current project comprising the creation of a open BTM machine using payment channel to withdraw cash money and threshold signature to buy crypto-coins to avoid storing full private keys on the machine.

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`.

```python
#!/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)
```

```
    mu2 = paillier.mult(zeta, rq * x2z2, n2)
    mu3, rnumb = paillier.encrypt(x3 * ecdsa.n, ka_pub)
    mu = paillier.add(paillier.add(mu1, mu2, n2), mu3, n2)

    muprim, rmuprim = paillier.encrypt(z2, kb_pub)

    c = r2
    d = ecdsa.G
    w1 = ecdsa.G
    w2 = y2
    m1 = muprim # ENCRYPTED Z2
    m2 = mu # ENCRYPTED RESULT
    m3 = alpha # ENCRYPTED Z1
    m4 = zeta # ENCRYPTED X1Z1
    r1 = rmuprim
    r2 = rnumb
    x1 = z2
    x2 = x2z2
    x4 = m
    x5 = rq

    pi2 = eczkp.pi2(c, d, w1, w2, m1, m2, m3, m4, r1, r2, x1, x2, x3, x4, x5, zkp, ka_pub,
     ↪  kb_pub)
    if not pi2:
        print "Error: zkp failed"
        exit(1)

    return mu, muprim, pi2

def alice_round_3(pi2, r, r2, y2, mup, mu, alpha, zeta, zkp, ka_priv, kb_pub):
    n, p, q, g, lmdba, mupaillier = ka_priv
    ka_pub = (n, g)
    rf = r[0] % ecdsa.n

    c = r2
    d = ecdsa.G
    w1 = ecdsa.G
    w2 = y2
    m1 = mup
    m2 = mu
    m3 = alpha
    m4 = zeta

    if not eczkp.pi2_verify(pi2, c, d, w1, w2, m1, m2, m3, m4, zkp, ka_pub, kb_pub):
        print "Error: zkp 2 failed"
        exit(1)

    s = paillier.decrypt(mu, ka_priv) % ecdsa.n
    if s == 0:
        print("signature failed, retry")
        exit(1)

    return rf, s


def run_secdsa():
    # Aclice
    x1 = utils.randomnumber(ecdsa.n, inf=2)
    y1 = ecdsa.get_pub(x1)
    ka_pub, ka_priv = paillier.gen_key()

    # Bob
    x2 = utils.randomnumber(ecdsa.n, inf=2)
    y2 = ecdsa.get_pub(x2)
    kb_pub, kb_priv = paillier.gen_key()

    zkp = eczkp.gen_params(1024)

    pub = ecdsa.get_pub(x1 * x2 % ecdsa.n)
```

```python

    # Message hash
    message = "hello"
    h = hashlib.sha256()
    h.update(message.encode("utf-8"))
    m = long(h.hexdigest(), 16)
    print "Message to sign: ", message
    print "Hash: ", m

    # ALICE ROUND 1
    k1, z1, alpha, zeta, rr1, rr2 = alice_round_1(m, x1, y1, ka_pub, ka_priv)
    # BOB ROUND 1
    k2, r2 = bob_round_1(alpha, zeta)
    # ALICE ROUND 2
    r, pi = alice_round_2(alpha, zeta, r2, k1, y1, z1, x1, zkp, ka_pub, rr1, rr2)
    # BOB ROUND 2
    mu, mup, pi2 = bob_round_2(pi, m, alpha, zeta, r, k2, x2, r2, y1, y2, ka_pub, kb_pub,
     ↪  zkp)
    # ALICE ROUND 3 (final)
    sig = alice_round_3(pi2, r, r2, y2, mup, mu, alpha, zeta, zkp, ka_priv, kb_pub)

    print "Signature:"
    print sig
    r, s = sig
    print "Sig status: ", ecdsa.verify(sig, m, pub, ecdsa.G, ecdsa.n)


if __name__ == "__main__":
    print("S-ECDSA")
    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 commercial 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 Π, 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:

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

It is worth noting that Π_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 <$\pi_i^{carol}$> <$\pi_i^{bob}$> 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
  <$\omega_i^{carol}$> OP_CHECKSIG
  <timelock> OP_CHECKSEQUENCEVERIFY OP_DROP
OP_ELSE
  <$\omega_i^{bob}$> OP_CHECKSIGVERIFY
  OP_HASH160 <Hash160($\Phi_{i,n}$)> OP_EQUAL
OP_ENDIF
```

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

```
<$\Omega_i^{carol}$ 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.

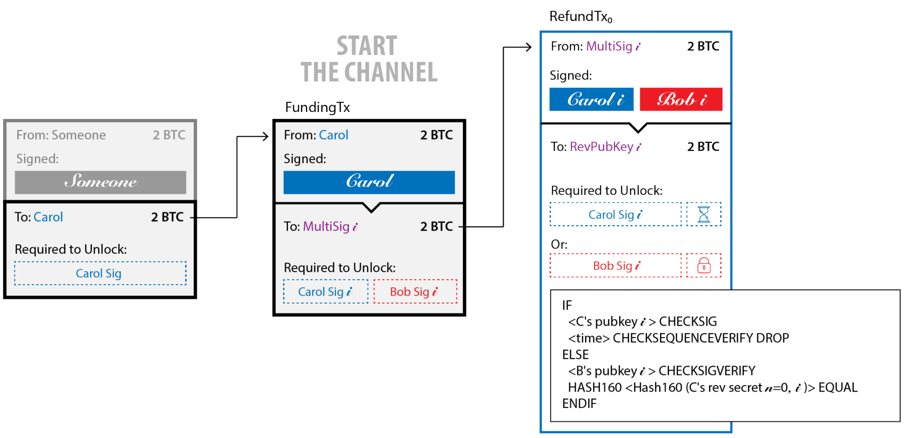

Fig. 1. Funding transaction that starts the channel by sending money in the first multisig address with the first refund transaction that allows Carol to close the channel if no transaction is made.

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.

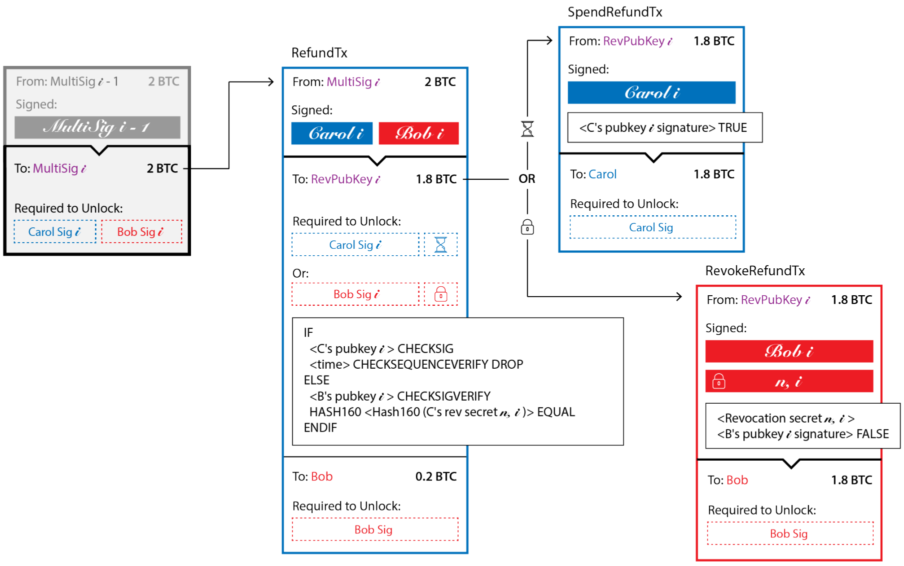

Fig. 2. Refund transaction based on the current multisig address with the associated spend and revoke transactions that allows Carol to get her money back and Bob to revoke the contract if he knows the secret.

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

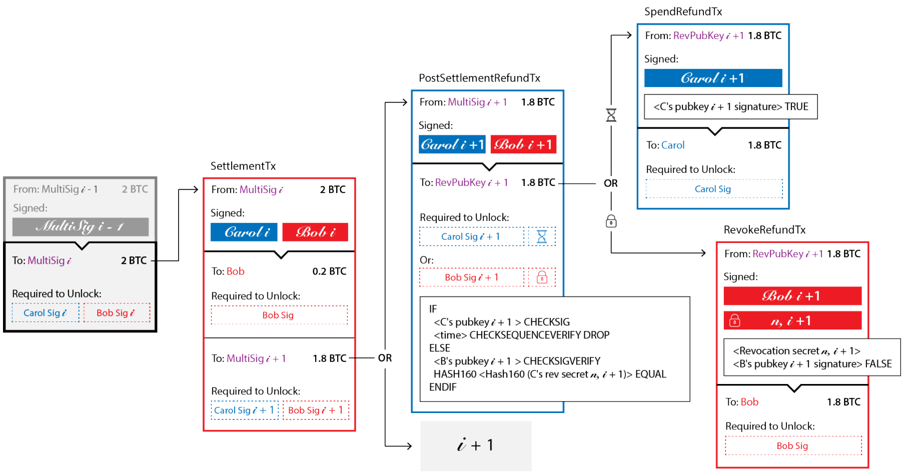

Fig. 3. Settlement transaction that allows Bob to settle the channel, moving the remaining funds to the next multisig address with the post settlement refund transaction that allows Carol to close the channel directly after the settlement.

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.

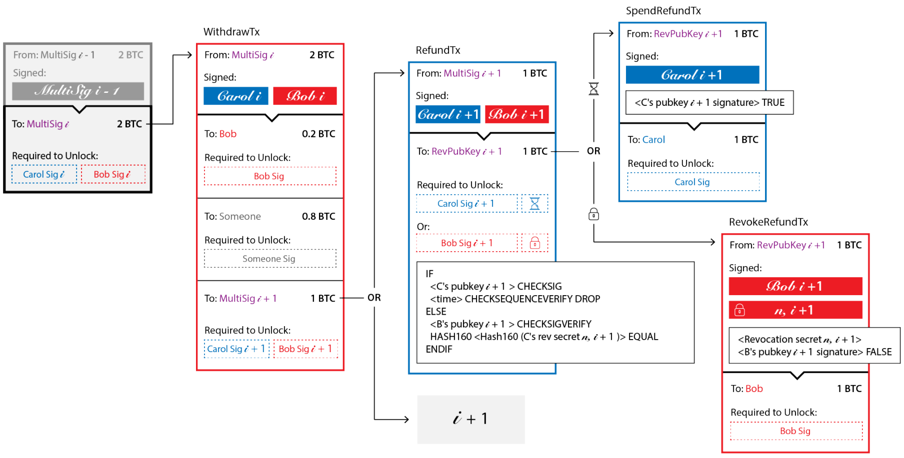

Fig. 4. Withdraw transaction that allows Carol to take money out the channel, pay Bob, and move the remaining funds into the next multisig address. A refund transaction is created to allow Carol to recover her funds after the withdraw if no transaction is made. The refund transaction can be spent by 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.

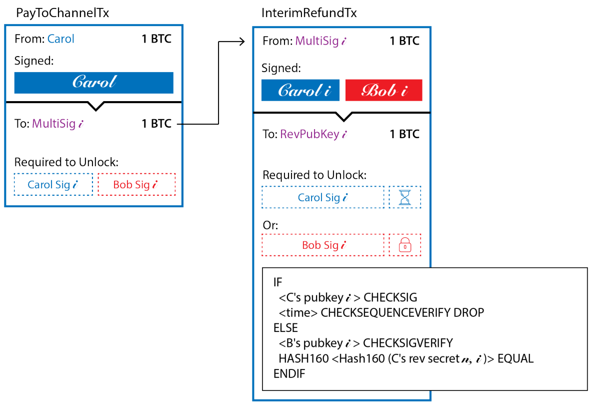

Fig. 5. Pay to channel transaction by Carol with the interim refund transaction. The interim refund transaction acts like a standard refund transaction but aims to be merged in the next round of transactions. The interim refund transaction has the same spending requirements as a standard refund transaction,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

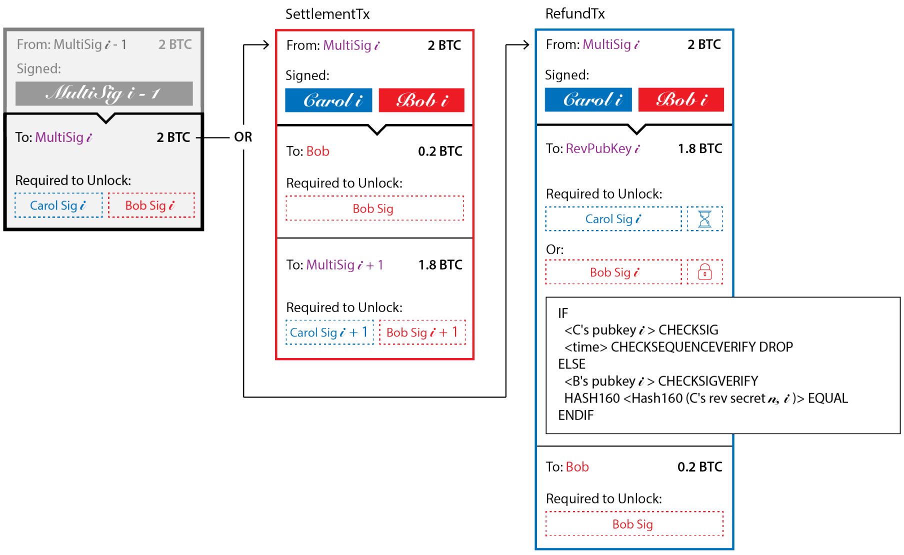

Fig. 6. Simplified view of possibilites for a standard state $\texttt{Channel}_{i,n}$ without second layer dependency transactions like spend and revoke. The content of a multisig can be settled by Bob or can be refunded to Carol.

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

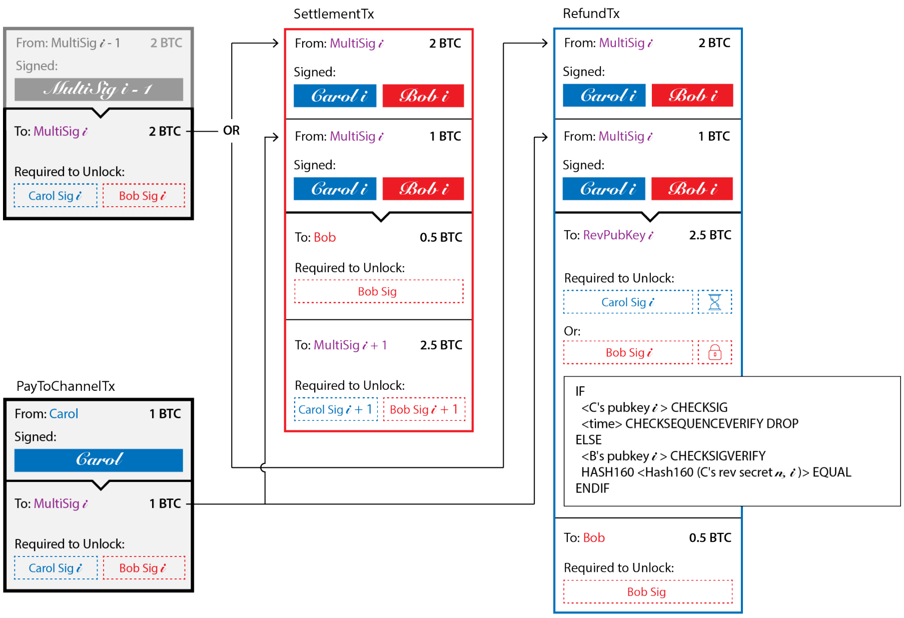

Fig. 7. Result of a merged pay to channel transaction after Carol sends 0.3 more Bitcoin to Bob. The $\texttt{Multisig}_i$ contains two `UTXOs`, (i) from the funding transaction or the last move from $\texttt{Multisig}_{i-1}$, and (ii) from the pay to channel transaction adding 1 BTC into the channel.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

List of Tables

List of sources

Bibliography

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

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

[3] 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).

[4] 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).

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

[6] 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`.

[7] 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`.

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

[9] 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.

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

[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. 8–10, 27

DMC Duplex Micropayment Channels. 13

DSA Digital Signature Algorithm. 15, 16, 18–20, 32

EC Elliptic Curves. 16, 20, 55

ECDSA Elliptic Curve Digital Signature Algorithm. 15, 16, 18–20, 32, 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.. 16

P2PKH Pay To Public Key Hash. 8, 9, 14

SegWit Segregated Witness. 8–10, 13, 14

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). 16