



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:

firstname lastname  
lab

Lausanne, HES-SO//Master, January 26, 2018



# Information about this report

## Contact information

Author: JoëlGugger  
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  
firstname lastname, lab, Main expert

Place, date: \_\_\_\_\_

Prof. Alexandre Karlov  
Advisor

Prof. Fariba Moghaddam Bützberger  
Dean, HES-SO//Master



# Acknowledgments

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetur id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.



# Preface

A preface is not mandatory. It would typically be written by some other person (eg your thesis director).

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetur id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

*Lausanne, 12 Mars 2011*

T. D.





# Abstract

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

**Keywords:** keyword1, keyword2, keyword3



# Contents

<b>Acknowledgements</b>	<b>v</b>
<b>Preface</b>	<b>vii</b>
<b>Abstract</b>	<b>ix</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 Bitcoin, a peer-to-peer payment network</b>	<b>3</b>
2.1 Actors . . . . .	4
2.2 Blockchain . . . . .	4
2.3 Transaction . . . . .	5
2.4 Scalability of Bitcoin . . . . .	5
<b>3 Payment channels, a micro-transaction network</b>	<b>7</b>
3.1 Types of payment channel . . . . .	8
3.2 Our one-way channel . . . . .	8
3.3 Optimizing channels . . . . .	8
<b>4 (1,2)-threshold optimal ECDSA</b>	<b>11</b>
4.1 Recap . . . . .	12
4.2 Threshold scheme . . . . .	15
4.3 Threshold Hierarchical Deterministic Wallets . . . . .	18
4.4 Threshold deterministic signature . . . . .	20
<b>5 Implementation in Bitcoin-core secp256k1</b>	<b>21</b>
5.1 Configuration . . . . .	23
5.2 DER parser-serializer . . . . .	25
5.3 Paillier cryptosystem . . . . .	28
5.4 Zero-knowledge proofs . . . . .	31
5.5 Threshold module . . . . .	35
<b>6 Further research</b>	<b>43</b>
6.1 Hardware wallets . . . . .	43
<b>7 Conclusions</b>	<b>45</b>
<b>A Docker Configuration</b>	<b>47</b>
<b>List of Figures</b>	<b>49</b>

## Contents

---

List of Tables	51
List of Sources	53
Bibliography	55
Glossary	57





# 1 | Introduction

What is Bitcoin? why do we need it? [1, 2, 3, 4, 5]

ECDSA is the signature scheme used by Bitcoin to sign transactions. A standard transaction is constituted of a single signature corresponding to the address where the Bitcoins come from. But sometimes we need more complex management for locking funds. To address the limitation of a single signature, Bitcoin introduced a new OP\_CODE named CHECKMULTISIG with a new standard script. With this standard script, it is now possible to spend Bitcoin to an address that requires a minimum of  $m$  signatures in  $n$  authorised signatories and extend the capability of Bitcoin to lock funds in a more complex way.

However, some issues appear. The way the script works requires exposing all the public keys when an output is signed and this increases the transaction size enormously, which implies bigger fees. All the signatures are, obviously, present with the public keys in the transaction script, which implies that we can know which public keys signed the transaction. And there is some limitation, due to the script size limit, the maximum number of authorised signatories is 15. All these limitations mean that we cannot imagine a complex organization nor structure with the multi-signature script for the moment.

To address this limitation, a group of researchers published a first paper in 2015 and a second one in 2016 describing the way to achieve a threshold scheme with DSA and ECDSA. Today, there is no well-known implementation ready for production purposes even though industries need it. The principal purpose of this thesis is to provide a clear, well documented C library, based on the internal ECDSA library present in bitcoin-core.

The largest challenge in Bitcoin for the coming years is scalability. Currently, Bitcoin enforces a block-size limit which is equivalent to only some transactions per second on the network. This is not sufficient in comparison to big payment infrastructures such as VISA, which allows tens of thousands of transactions per second and even more in peak times such as Christmas. To address this, some proposals modifying the transaction structure (like SegWit), some proposals modifying the block-size limit (such as SegWit2x) and others creating a second layer based on top of the Bitcoin protocol (like Lightning Network) exist. In the same idea of the Lightning Network, Bity is working on an implementation of a one-way payment channel. A one-way 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. This kind of channel needs multi-signature addresses which might be improved with the threshold scheme. The second part of the thesis is to co-write the channel white paper and add a chapter of how to improve it with the threshold scheme (better privacy, cheaper transaction, less limitations).





## 2 | Bitcoin, a peer-to-peer payment network

Concepts Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetur id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

### Contents

<b>2.1</b>	<b>Actors</b>	<b>4</b>
2.1.1	Users	4
2.1.2	Miners	4
2.1.3	Developpers	4
<b>2.2</b>	<b>Blockchain</b>	<b>4</b>
2.2.1	Public ledger	5
2.2.2	Speed	5
<b>2.3</b>	<b>Transaction</b>	<b>5</b>
2.3.1	Scripting language	5
2.3.2	Transaction Fees	5
<b>2.4</b>	<b>Scalability of Bitcoin</b>	<b>5</b>
2.4.1	On-chain improvements	6
2.4.2	Layer-two applications	6

### 2.1 Actors

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetur id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

type of nodes in the peer-to-peer network [https://github.com/bitcoinbook/bitcoinbook/blob/second\\_edition/ch08.asciidoc](https://github.com/bitcoinbook/bitcoinbook/blob/second_edition/ch08.asciidoc)

#### 2.1.1 Users

#### 2.1.2 Miners

#### 2.1.3 Developpers

### 2.2 Blockchain

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetur id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

**2.2.1 Public ledger****2.2.2 Speed**

Mining difficulty

**2.3 Transaction**

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetur id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

**2.3.1 Scripting language**

Locking script, unlocking script

**2.3.2 Transaction Fees****2.4 Scalability of Bitcoin**

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetur id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa.

Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

### **2.4.1 On-chain improvements**

### **2.4.2 Layer-two applications**

### 3 | Payment channels, a micro-transaction network

Concept Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetur id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

#### Contents

<b>3.1</b>	<b>Types of payment channel</b>	<b>8</b>
<b>3.2</b>	<b>Our one-way channel</b>	<b>8</b>
<b>3.3</b>	<b>Optimizing channels</b>	<b>8</b>

### 3.1 Types of payment channel

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetur id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

### 3.2 Our one-way channel

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetur id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

### 3.3 Optimizing channels

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetur id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus

vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.





## 4 | (1,2)-threshold optimal ECDSA

### Contents

<b>4.1</b>	<b>Recap</b>	<b>12</b>
4.1.1	Elliptic curves	12
4.1.2	Paillier cryptosystem	13
4.1.3	Digital Signature Algorithm	13
4.1.4	Elliptic Curve Digital Signature Algorithm	14
<b>4.2</b>	<b>Threshold scheme</b>	<b>15</b>
4.2.1	Adapting zero-knowledge proofs to ECDSA	15
<b>4.3</b>	<b>Threshold Hierarchical Deterministic Wallets</b>	<b>18</b>
4.3.1	Private parent key to private child key	19
4.3.2	Public parent key to public child key	19
4.3.3	Child key share derivation	20
<b>4.4</b>	<b>Threshold deterministic signature</b>	<b>20</b>

## 4.1 Recap

Before introducing the threshold scheme, a recap of basic components used after in the scheme is presented. The recap is composed of Elliptic Curves (EC) basic mathematics, Paillier homomorphic encryption scheme, and digital signature—in particular Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (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.

#### 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  $b = 7$ .

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

```

p =  FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFC2F
G =      (79BE667E F9DCBBAC 55A06295 CE870B07 029BFCDB 2DCE28D9 59F2815B 16F81798,
      483ADA77 26A3C465 5DA4FBFC 0E1108A8 FD17B448 A6855419 9C47D08F FB10D4B8)
n =  FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF BAAEDCE6 AF48A03B BFD25E8C D0364141

```

The curve order  $n$  define the number of element (points) generated by the generator  $G$  on the curve. An exponentiation of the generator  $g^a \bmod p$  become a point multiplication with the generator  $a \cdot G$ .

#### Points addition

With two distinct point  $P$  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  $dP = P_1 + P_2 + \dots + P_d$ .

**Lemma 4.1.1** *Given a multiple  $Q$  of  $P$  where  $Q = nP$  it is infeasible to find  $n$  if  $n$  is large.*

### 4.1.2 Paillier cryptosystem

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  $c = g^m \cdot r^n \pmod{n^2}$  to encrypt the plaintext.

#### Decryption

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

### 4.1.3 Digital Signature Algorithm

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  $r = (g^k \bmod p) \bmod q \wedge r \neq 0$
- Calculate  $s = k^{-1}(\text{hash}(m) + xr) \bmod q \wedge s \neq 0$
- The signature is  $(r, s)$

### Verifying

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

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

### 4.1.4 Elliptic Curve Digital Signature Algorithm

ECDSA is a variant of DSA which uses elliptic curve cryptography.

#### 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  $r = x_1 \bmod n \wedge r \neq 0$
- Calculate  $s = k^{-1}(\text{hash}(m) + xr) \bmod n \wedge s \neq 0$
- 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  $w = s^{-1} \bmod n$
- Calculate  $u_1 = \text{hash}(m) \cdot w \bmod n$
- Calculate  $u_2 = rw \bmod n$

- 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  $r \equiv x_1 \pmod{n}$

## 4.2 Threshold scheme

### 4.2.1 Adapting zero-knowledge proofs to ECDSA

Initially the proofs have been designed 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 function (one different per proof.) For ECDSA some of these values are now points and some equation need to be adapted. Points are serialized in long form of 65 bytes, starting with 0x04 and two 32 bytes number for the coordinate x and y.

The first zero-knowledge proof  $\Pi$  is created by Alice to prove to Bob that she act correctly and have encrypted coherent data with Paillier encryption.

$\alpha \xleftarrow{R} \mathbb{Z}_{n^3}$	$\delta \xleftarrow{R} \mathbb{Z}_{n^3}$
$\beta \xleftarrow{R} \mathbb{Z}_N^*$	$\mu \xleftarrow{R} \mathbb{Z}_N^*$
$\gamma \xleftarrow{R} \mathbb{Z}_{n^3 \tilde{N}}$	$\nu \xleftarrow{R} \mathbb{Z}_{n^3 \tilde{N}}$
$\rho_1 \xleftarrow{R} \mathbb{Z}_{n \tilde{N}}$	$\rho_2 \xleftarrow{R} \mathbb{Z}_{n \tilde{N}}$
	$\rho_3 \xleftarrow{R} \mathbb{Z}_n$
	$\epsilon \xleftarrow{R} \mathbb{Z}_n$
$z_1 \leftarrow (h_1)^{x_1} (h_2)^{\rho_1} \pmod{\tilde{N}}$	$z_2 \leftarrow (h_1)^{x_2} (h_2)^{\rho_2} \pmod{\tilde{N}}$
$u_1 \leftarrow c \cdot \alpha$	$y \leftarrow d \cdot (x_2 + \rho_3)$
$u_2 \leftarrow g^\alpha \beta^N \pmod{N^2}$	$v_1 \leftarrow d \cdot (\delta + \epsilon)$
$u_3 \leftarrow (h_1)^\alpha (h_2)^\gamma \pmod{\tilde{N}}$	$v_2 \leftarrow w_2 \cdot \alpha + d \cdot \epsilon$
	$v_3 \leftarrow g^\delta \mu^N \pmod{N^2}$
	$v_4 \leftarrow (h_1)^\delta (h_2)^\nu \pmod{\tilde{N}}$
$e \leftarrow \text{hash}(c, w_1, d, w_2, m_1, m_2, z_1, u_1, u_2, u_3, z_2, y, v_1, v_2, v_3, v_4)$	
$s_1 \leftarrow ex_1 + \alpha$	$t_1 \leftarrow ex_2 + \delta$
$s_2 \leftarrow (r_1)^e \beta \pmod{N}$	$t_2 \leftarrow e\rho_3 + \epsilon \pmod{n}$
$s_3 \leftarrow e\rho_1 + \gamma$	$t_3 \leftarrow (r_2)^e \mu \pmod{N^2}$
	$t_4 \leftarrow e\rho_2 + \nu$
$\Pi \leftarrow \langle z_1, z_2, y, e, s_1, s_2, s_3, t_1, t_2, t_3, t_4 \rangle$	

Figure 4.1 Adaptation of  $\Pi$  to ECDSA

$$\begin{array}{l}
 \langle z_1, z_2, y, e, s_1, s_2, s_3, t_1, t_2, t_3, t_4 \rangle \leftarrow \Pi \\
 \\
 \begin{array}{ll}
 \text{Verify } s_1, t_1 \in \mathbb{Z}_{n^3} & v_1 \leftarrow d \cdot (t_1 + t_2) + y \cdot (-e) \\
 u_1 \leftarrow c \cdot s_1 + w_1 \cdot (-e) & v_2 \leftarrow w_2 \cdot s_1 + d \cdot t_2 + y \cdot (-e) \\
 u_2 \leftarrow g^{s_1}(s_2)^N(m_1)^{-e} \bmod N^2 & v_3 \leftarrow g^{t_1}(t_3)^N(m_2)^{-e} \bmod N^2 \\
 u_3 \leftarrow (h_1)^{s_1}(h_2)^{s_3}(z_1)^{-e} \bmod \tilde{N} & v_4 \leftarrow (h_1)^{t_1}(h_2)^{t_4}(z_2)^{-e} \bmod \tilde{N}
 \end{array} \\
 \\
 \text{Verify } e = \mathbf{hash}(c, w_1, d, w_2, m_1, m_2, z_1, u_1, u_2, u_3, z_2, y, v_1, v_2, v_3, v_4)
 \end{array}$$

Figure 4.2 Adaptation of  $\Pi$  verification to ECDSA

Second zero-knowledge proof created by Bob to prove to Alice that he acted correctly according the protocol.

$\alpha \xleftarrow{R} \mathbb{Z}_{n^3}$	$\delta \xleftarrow{R} \mathbb{Z}_{n^3}$
$\beta \xleftarrow{R} \mathbb{Z}_{N'}^*$	$\mu \xleftarrow{R} \mathbb{Z}_N^*$
$\gamma \xleftarrow{R} \mathbb{Z}_{n^3 \tilde{N}}$	$\nu \xleftarrow{R} \mathbb{Z}_{n^3 \tilde{N}}$
$\rho_1 \xleftarrow{R} \mathbb{Z}_{n \tilde{N}}$	$\rho_2 \xleftarrow{R} \mathbb{Z}_{n \tilde{N}}$
	$\rho_3 \xleftarrow{R} \mathbb{Z}_n$
	$\rho_4 \xleftarrow{R} \mathbb{Z}_{n^5 \tilde{N}}$
	$\epsilon \xleftarrow{R} \mathbb{Z}_n$
	$\sigma \xleftarrow{R} \mathbb{Z}_{n^7}$
	$\tau \xleftarrow{R} \mathbb{Z}_{n^7 \tilde{N}}$
$z_1 \leftarrow (h_1)^{x_1} (h_2)^{\rho_1} \pmod{\tilde{N}}$	$z_2 \leftarrow (h_1)^{x_2} (h_2)^{\rho_2} \pmod{\tilde{N}}$
$u_1 \leftarrow c \cdot \alpha$	$y \leftarrow d \cdot (x_2 + \rho_3)$
$u_2 \leftarrow (g')^\alpha \beta^{N'} \pmod{(N')^2}$	$v_1 \leftarrow d \cdot (\delta + \epsilon)$
$u_3 \leftarrow (h_1)^\alpha (h_2)^\gamma \pmod{\tilde{N}}$	$v_2 \leftarrow w_2 \cdot \alpha + d \cdot \epsilon$
	$v_3 \leftarrow (m_3)^\alpha (m_4)^\delta g^{n\sigma} \mu^N \pmod{N^2}$
	$v_4 \leftarrow (h_1)^\delta (h_2)^\nu \pmod{\tilde{N}}$
	$z_3 \leftarrow (h_1)^{x_3} (h_2)^{\rho_4} \pmod{\tilde{N}}$
	$v_5 \leftarrow (h_1)^\sigma (h_2)^\tau \pmod{\tilde{N}}$
$e \leftarrow \text{hash}'(c, w_1, d, w_2, m_1, m_2, z_1, u_1, u_2, u_3, z_2, z_3, y, v_1, v_2, v_3, v_4, v_5)$	
$s_1 \leftarrow ex_1 + \alpha$	$t_1 \leftarrow ex_2 + \delta$
$s_2 \leftarrow (r_1)^e \beta \pmod{N'}$	$t_2 \leftarrow e\rho_3 + \epsilon \pmod{n}$
$s_3 \leftarrow e\rho_1 + \gamma$	$t_3 \leftarrow (r_2)^e \mu \pmod{N}$
$s_4 \leftarrow ex_1 x_4 + \alpha$	$t_4 \leftarrow e\rho_2 + \nu$
	$t_5 \leftarrow ex_3 + \sigma$
	$t_6 \leftarrow e\rho_4 + \tau$
	$t_7 \leftarrow ex_2 x_5 + \delta$
$\Pi' \leftarrow \langle z_1, z_2, z_3, y, e, s_1, s_2, s_3, s_4, t_1, t_2, t_3, t_4, t_5, t_6, t_7 \rangle$	

 Figure 4.3 Adaptation of  $\Pi'$  to ECDSA

$\langle z_1, z_2, z_3, y, e, s_1, s_2, s_3, s_4, t_1, t_2, t_3, t_4, t_5, t_6, t_7 \rangle \leftarrow \Pi'$	
Verify $s_1, t_1 \in \mathbb{Z}_{n^3}$	$v_1 \leftarrow d \cdot (t_1 + t_2) + y \cdot (-e)$
Verify $t_5 \in \mathbb{Z}_{n^7}$	$v_2 \leftarrow w_2 \cdot s_1 + d \cdot t_2 + y \cdot (-e)$
$u_1 \leftarrow c \cdot s_1 + w_1 \cdot (-e)$	$v_3 \leftarrow (m_3)^{s_4} (m_4)^{t_7} g^{nt_5} (t_3)^N (m_2)^{-e} \pmod{N^2}$
$u_2 \leftarrow (g')^{s_1} (s_2)^{N'} (m_1)^{-e} \pmod{(N')^2}$	$v_4 \leftarrow (h_1)^{t_1} (h_2)^{t_4} (z_2)^{-e} \pmod{\tilde{N}}$
$u_3 \leftarrow (h_1)^{s_1} (h_2)^{s_3} (z_1)^{-e} \pmod{\tilde{N}}$	$v_5 \leftarrow (h_1)^{t_5} (h_2)^{t_6} (z_3)^{-e} \pmod{\tilde{N}}$
Verify $e = \mathbf{hash}'(c, w_1, d, w_2, m_1, m_2, z_1, u_1, u_2, u_3, z_2, z_3, y, v_1, v_2, v_3, v_4, v_5)$	

 Figure 4.4 Adaptation of  $\Pi'$  verification to ECDSA

If  $x_1 = z_2$ ,  $x_2 = x_2 z_2$ ,  $x_3 = c$ , and  $m_2 = \mu$  with  $\mu = (\alpha)^{m'x_1} (\zeta)^{r'x_2} g^{nx_3} (r_2)^N$  then the equation  $v_3$  in the original paper doesn't work. The result in the verification process  $\Pi'$  need to match  $v_3 \leftarrow (m_3)^\alpha (m_4)^\delta g^{n\sigma} \mu^N \pmod{N^2}$ . The original equation proposed  $v_3 \leftarrow (m_3)^{s_1} (m_4)^{t_1} g^{nt_5} (t_3)^N (m_2)^{-e} \pmod{N^2}$  doesn't include  $m'$  and  $r'$  present in  $\mu$ , so  $m_2$  cannot be used correctly as showed next.

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

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 other equations.) Two new parameters  $s_4 \leftarrow ex_1 x_4 + \alpha$  and  $t_7 \leftarrow ex_2 x_5 + \delta$  are added into the proof to correct the equation.

$$\begin{aligned}
 v_3 &\equiv (m_3)^{s_4} (m_4)^{t_7} g^{nt_5} (t_3)^N (m_2)^{-e} \pmod{N^2} \\
 &\equiv (m_3)^{ex_1 x_4 + \alpha} (m_4)^{ex_2 x_5 + \beta} g^{n(ex_3 + \sigma)} ((r_2)^e \mu)^N ((m_3)^{x_1 x_4} (m_4)^{x_2 x_5} g^{nx_3} (r_2)^N)^{-e} \\
 &\equiv (m_3)^{ex_1 x_4 + \alpha} (m_4)^{ex_2 x_5 + \beta} g^{n(ex_3 + \sigma)} (r_2)^{eN} \mu^N (m_3)^{-ex_1 x_4} (m_4)^{-ex_2 x_5} g^{-enx_3} (r_2)^{-eN} \\
 &\equiv (m_3)^{ex_1 x_4 + \alpha - ex_1 x_4} (m_4)^{ex_2 x_5 + \beta - ex_2 x_5} g^{enx_3 + n\sigma - enx_3} (r_2)^{eN - eN} \mu^N \\
 &\equiv (m_3)^\alpha (m_4)^\beta g^{n\sigma} \mu^N
 \end{aligned} \tag{4.4}$$

### 4.3 Threshold Hierarchical Deterministic Wallets

Hierarchical deterministic wallets are sophisticated wallets in which fresh 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 public points  $pk_i$ , which means that if each one publish his own public share point, every one can compute the master public key.

#### 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 derivatioin can be *hardened*. This proposal differ from the BIP32 standard in the chain derivation proccess. The threshold scheme require the same chain on all participants, so the proccess cannot rely on the private key share of any participant.

$$\begin{aligned}
f(l) &= \begin{cases} \text{HMAC-SHA256}(c_{par}, 0x00 \parallel \text{ser}_{256}(k_{par}) \parallel \text{ser}_{32}(i)) & \text{if } i \geq 2^{31} \\ \text{HMAC-SHA256}(c_{par}, \text{ser}_p(\text{point}(k_{par})) \parallel \text{ser}_{32}(i)) & \text{if } i < 2^{31} \end{cases} \\
k_i &\equiv l \cdot k_{par} \pmod{n}
\end{aligned}$$

#### 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 than it is not possible to compute an *hardened* derivation without the parent private key.

$$\begin{aligned}
f(l) &= \begin{cases} \text{failure} & \text{if } i \geq 2^{31} \\ \text{HMAC-SHA256}(c_{par}, \text{ser}_p(K_{par}) \parallel \text{ser}_{32}(i)) & \text{if } i < 2^{31} \end{cases} \\
K_i &= l \cdot K_{par} \\
&= l \cdot (k_{par} \cdot G) \\
&= (l \cdot k_{par} \bmod n) \cdot G
\end{aligned}$$

### 4.3.3 Child key share derivation

It is assume that one of the participants  $P_j$  is designated as the leader  $L$ . The function CKSD compute a child extended key share from the parent extended key share.

$$\begin{aligned}
 f(T) &= \begin{cases} \text{CKDpriv}(i) & \text{if } L = j \\ \text{CKDpub}(i) & \text{if } L \neq j \end{cases} \\
 sk_i &= \left[ \prod_{s=1}^j sk_{par}^s \right] \cdot T \\
 &= sk_{par} \cdot T \\
 pk_i &= pk_{par} \cdot T \\
 &= sk_{par} \cdot G \cdot T \\
 &= \left[ \prod_{s=1}^j sk_{par}^s \right] \cdot G \\
 c_i &= \text{HMAC-SHA256}(c_{par}, \text{ser}_{32}(i))
 \end{aligned}$$

If the index is greather or equal than  $2^{31}$  the public key share of the participants  $P_j = L$  need to be revealed in order to compute the public child key, a round of communication is needed.

## 4.4 Threshold deterministic signature

How to use an HMAC function in threshold mode to have deterministic  $k$  for signature.

# 5 | Implementation in Bitcoin-core secp256k1

As mentioned before, Bitcoin use elliptic 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

<b>5.1</b>	<b>Configuration</b>	<b>23</b>
5.1.1	Add new experimental module	23
5.1.2	Configure compilation	24
<b>5.2</b>	<b>DER parser-serializer</b>	<b>25</b>
5.2.1	Sequence	25
5.2.2	Integer	26
5.2.3	Octet string	26
<b>5.3</b>	<b>Paillier cryptosystem</b>	<b>28</b>
5.3.1	Data structures	28
5.3.2	Encrypt and decrypt	29
5.3.3	Homomorphism	31
<b>5.4</b>	<b>Zero-knowledge proofs</b>	<b>31</b>
5.4.1	Data structures	32
5.4.2	Generate proofs	33
5.4.3	Validate proofs	35

<b>5.5</b>	<b>Threshold module . . . . .</b>	<b>35</b>
5.5.1	Create call message . . . . .	36
5.5.2	Receive call message . . . . .	37
5.5.3	Receive challenge message . . . . .	38
5.5.4	Receive response challenge message . . . . .	39
5.5.5	Receive terminate message . . . . .	39

---

## 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`.

```

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

```

*Listing 5.1 Add argument 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.

```

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

```

*Listing 5.4 Set threshold module to experimental 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 functionalities 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.

```

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

```

*Listing 5.6 Specialized Makefile for threshold module*

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

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

## 5.2 DER parser-serializer

Transmit messages and retrieve keys are an 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  $\text{byte} = \{b_7, \dots, b_1, b_0\}$  indicate the number of followed bytes which are used for the lenght.

```

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

*Listing 5.7 Implementation of a DER 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 retrieved lenght result to the same amount of bytes passed in argument.

When a sequence holds other sequence, retrieve their total lenght (including header and content lenght bytes) is needed to recursively parse them. A specific function is created to retrieve 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.

```

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

```

*Listing 5.8 Implementation of a DER sequence parser*

The result of a content lenght serialization can be  $\geq 1$  byte-s. If the content is shorter than 0x80, then one byte is enough to store the lenght. Else multiple bytes ( $\geq 2$ ) 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.

```

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

```

*Listing 5.9 Implementation of a DER sequence serializer*

If the content 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 just 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 retrieve the lenght of the content,



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

*Listing 5.10 Implementation of a DER 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 content 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 transmitted. 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 exponent  $\lambda = \varphi(n) = (p-1)(q-1)$ , 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*

```
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.

## Chapter 5. Implementation in Bitcoin-core secp256k1

---

```
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  $m = L(c^\lambda \bmod n^2) \cdot \mu \bmod n$  where  $L(x) = (x - 1)/n$ . The cipher is raised to the lambda  $c^\lambda \bmod n^2$  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  $\mu$  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 component  $s = D_{sk}(\mu) \bmod q : \mu = (\alpha \times_{pk} m' z_2) +_{pk} (\zeta \times_{pk} r' x_2 z_2) +_{pk} E_{pk}(cn)$  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  $D_{sk}(E_{pk}(m_1, r_1) \cdot E_{pk}(m_2, r_2) \bmod n^2) = m_1 + m_2 \bmod n$  or  $D_{sk}(E_{pk}(m_1, r_1) \cdot g^{m_2} \bmod n^2) = m_1 + m_2 \bmod n$  where  $D_{sk}$  denotes decryption with private key  $sk$  and  $E_{pk}$  denotes encryption with public key  $pk$ . Only the first variant is implemented, where two ciphertexts are added together to result in a third ciphertext.

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

Listing 5.19 Implementation of homomorphic addition with Paillier cryptosystem

#### Multiplication

Multiplication  $\times_{pk}$  over ciphertexts can be performed with  $D_{sk}(E_{pk}(m_1, r_1)^{m_2} \bmod n^2) = m_1 m_2 \bmod n$ , 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 operations.

```
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  $\exists x, \log_x(h_1) = h_2 \bmod \tilde{N}$ . One function is provided in the module to parse a `ZKPPParameter` DER schema.

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

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

#### Zero-Knowledge Proof $\Pi$

Zero-Knowledge Proof  $\Pi$  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  $\Pi$  sequence*

#### Zero-Knowledge Proof $\Pi'$

Zero-Knowledge Proof  $\Pi'$  is composed of the same named values as ZKP  $\Pi$  plus five new ones. The construction of the proof is based on  $\Pi$  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  $\Pi'$  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_eczrp_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 $\Pi$

As shown in FIGURE REF, the proof states that: (i) it exists a known value by the proover that link  $r \rightarrow r_2$ , (ii) it exists a second known value by the proover that, related to the first one, link  $G \rightarrow y_1$ , (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  $\alpha$  and  $\zeta$ , scalar values  $sx_1$  and  $sx_2$  representing  $z_1 = (k_1)^{-1} \bmod n$  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  $\alpha$  and  $\zeta$ , 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  $\Pi$

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  $e = \text{hash}(v)$ .

### Zero-Knowledge Proof $\Pi'$

As shown in FIGURE REF, the proof states that: (i) it exists a known value by the proover  $x_1$  that link  $r_2 \rightarrow G$ , (ii) it exists a second known value by the proover that, related to the first one, link  $G \rightarrow y_2$ , (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  $\Pi'$

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  $e = \text{hash}'(v)$ .

It is worth noting that `hash` and `hash'` must be different hashing function to avoid reusing  $\Pi$  proofs, even not satisfying the predicate, to construct fraudulent  $\Pi'$  proofs.



### 5.4.3 Validate proofs

Validation of proofs  $\Pi$  and  $\Pi'$  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, retrieve the candidate value  $e'$ , and compare if  $e = e'$ . If the values match the proof is valid.

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

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

*Listing 5.27 Function signature to validate ZKP  $\Pi$  and  $\Pi'$*

## 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  $P_1$  retrieve 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 retrieved 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.

```
247 int secp256k1_threshold_call_create(const secp256k1_context *ctx,
    ↪ secp256k1_threshold_call_msg *callmsg, secp256k1_threshold_signature_params *params,
    ↪ const secp256k1_scalar *secshare, const secp256k1_paillier_pubkey *paillierkey, const
    ↪ unsigned char *msg32, const secp256k1_nonce_function noncefp, const
    ↪ secp256k1_paillier_nonce_function pnoncefp) {
248     int ret = 0;
249     int overflow = 0;
250     unsigned char nonce32[32];
251     unsigned char sec32[32];
252     unsigned int count = 0;
253     secp256k1_scalar privinv;
254
255     ARG_CHECK(ctx != NULL);
256     ARG_CHECK(callmsg != NULL);
257     ARG_CHECK(params != NULL);
258     ARG_CHECK(secshare != NULL);
259     ARG_CHECK(paillierkey != NULL);
260     ARG_CHECK(msg32 != NULL);
261     secp256k1_scalar_get_b32(sec32, secshare);
262     while (1) {
263         ret = noncefp(nonce32, msg32, sec32, NULL, NULL, count);
264         if (!ret) {
265             break;
266         }
267         secp256k1_scalar_set_b32(&params->k, nonce32, &overflow);
268         if (!overflow && !secp256k1_scalar_is_zero(&params->k)) {
269             secp256k1_scalar_inverse(&params->z, &params->k); /* z1 */
270             secp256k1_scalar_mul(&prinv, &params->z, secshare); /* x1z1 */
271             if (secp256k1_paillier_encrypt_scalar(callmsg->alpha, &params->z, paillierkey,
    ↪ pnoncefp)
272                 && secp256k1_paillier_encrypt_scalar(callmsg->zeta, &prinv, paillierkey,
    ↪ pnoncefp)) {
273                 break;
274             }
275         }
276         count++;
277     }
278     memset(nonce32, 0, 32);
279     memset(sec32, 0, 32);
280     secp256k1_scalar_clear(&prinv);
281     return ret;
282 }
```

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.

```

284 int secp256k1_threshold_call_received(const secp256k1_context *ctx,
    ↪ secp256k1_threshold_challenge_msg *challengemsg, secp256k1_threshold_signature_params
    ↪ *params, const secp256k1_threshold_call_msg *callmsg, const secp256k1_scalar *secshare,
    ↪ const unsigned char *msg32, const secp256k1_nonce_function noncefp) {
285     int ret = 0;
286     int overflow = 0;
287     unsigned int count = 0;
288     unsigned char k32[32];
289     unsigned char sec32[32];
290
291     ARG_CHECK(ctx != NULL);
292     ARG_CHECK(challengemsg != NULL);
293     ARG_CHECK(params != NULL);
294     ARG_CHECK(callmsg != NULL);
295     ARG_CHECK(secshare != NULL);
296     ARG_CHECK(msg32 != NULL);
297     secp256k1_scalar_get_b32(sec32, secshare);
298     while (1) {
299         ret = noncefp(k32, msg32, sec32, NULL, NULL, count);
300         if (!ret) {
301             break;
302         }
303         secp256k1_scalar_set_b32(&params->k, k32, &overflow);
304         if (!overflow && !secp256k1_scalar_is_zero(&params->k)) {
305             if (secp256k1_ec_pubkey_create(ctx, &params->r, k32)) {
306                 memcpy(&challengemsg->r2, &params->r, sizeof(secp256k1_pubkey));
307                 break;
308             }
309         }
310         count++;
311     }
312     memset(k32, 0, 32);
313     memset(sec32, 0, 32);
314     return ret;
315 }

```

Listing 5.29 Implementation of `call_received` function

### 5.5.3 Receive challenge message

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

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

*Listing 5.30 Implementation of `challenge_received` function*

#### 5.5.4 Receive response challenge message

The `response_challenge_received` function is called by  $P_2$  and validates the first Zero-Knowledge Proof,  $\Pi$ . The final ciphertext which contain the  $s$  part of the distributed signature is computed and the second Zero-Knowledge Proof  $\Pi'$  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  $x_2z_2$ . A value  $x_3$  where  $n|x_3$  is added to the cipher to hide information about the secret share and the secret random. In ECDSA  $s = k^{-1}(m + rx) \pmod n$ , so the ciphertext match the requirement as demonstrated below:

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

#### 5.5.5 Receive terminate message

The `terminate_received` function is called by  $P_1$  and validates the second Zero-Knowledge Proof,  $\Pi'$ . After validation of the proof, the ciphertext is decrypted and the signature is composed. The signature is then tested and the protocol ends. Only  $P_1$  can decrypt the ciphertext so the protocol is asymmetric. If  $P_2$  also needs the signature,  $P_1$  must share it. There is now way for  $P_2$  to know the signature without a cooperative  $P_1$ .

```

379     ret = secp256k1_eczkp_pi_verify(
380         ctx,
381         respmsg->pi,
382         zkp,
383         callmsg->alpha,
384         callmsg->zeta,
385         &respmsg->r,
386         &challengemsg->r2,
387         pairedshare,
388         pairedkey
389     );
390     if (ret) {
391         mpz_inits(m1, m2, c, n5, n, nc, m, z, rsig, inv, NULL);
392         secp256k1_scalar_inverse(&params->z, &params->k); /* z2 */
393         secp256k1_scalar_mul(&prinv, &params->z, secshare); /* x2z2 */
394         mpz_import(n, 32, 1, sizeof(n32[0]), 1, 0, n32);
395         secp256k1_scalar_set_b32(&msg, msg32, &overflow);
396         if (!overflow && !secp256k1_scalar_is_zero(&msg)) {
397             secp256k1_pubkey_load(ctx, &r, &respmsg->r);
398             secp256k1_fe_normalize(&r.x);
399             secp256k1_fe_normalize(&r.y);
400             secp256k1_fe_get_b32(b, &r.x);
401             secp256k1_scalar_set_b32(&sigr, b, &overflow);
402             /* These two conditions should be checked before calling */
403             VERIFY_CHECK(!secp256k1_scalar_is_zero(&sigr));
404             VERIFY_CHECK(overflow == 0);
405             mpz_import(rsig, 32, 1, sizeof(b[0]), 1, 0, b);
406             secp256k1_scalar_get_b32(b, &params->z);
407             mpz_import(z, 32, 1, sizeof(b[0]), 1, 0, b);
408             secp256k1_scalar_get_b32(b, &prinv);
409             mpz_import(inv, 32, 1, sizeof(b[0]), 1, 0, b);
410             secp256k1_scalar_get_b32(b, &msg);
411             mpz_import(m, 32, 1, sizeof(msg32[0]), 1, 0, msg32);
412             mpz_mul(m1, m, z); /* m'z2 */
413             mpz_mul(m2, rsig, inv); /* r'x2z2 */
414             mpz_pow_ui(n5, n, 5);
415             noncefp(c, n5);
416             mpz_mul(nc, c, n); /* cn */
417             secp256k1_paillier_mult(m3, callmsg->alpha, m1, pairedkey);
418             secp256k1_paillier_mult(m4, callmsg->zeta, m2, pairedkey);
419             secp256k1_paillier_add(m5, m3, m4, pairedkey);
420             ret = secp256k1_paillier_encrypt_mpz(enc, nc, pairedkey, noncefp);
421             secp256k1_scalar_get_b32(sec32, secshare);
422             if (ret && secp256k1_ec_pubkey_create(ctx, &y2, sec32)) {
423                 secp256k1_paillier_add(termmsg->mu, m5, enc, pairedkey);
424                 ret = secp256k1_paillier_encrypt_mpz(termmsg->mu2, z, p2, noncefp);
425                 VERIFY_CHECK(secp256k1_eczkp_pi2_generate(
426                     ctx, /* ctx */
427                     termmsg->pi2, /* pi2 */
428                     zkp, /* zkp */
429                     termmsg->mu2, /* m1 */
430                     termmsg->mu, /* m2 */
431                     callmsg->alpha, /* m3 */
432                     callmsg->zeta, /* m4 */
433                     enc, /* r */
434                     z, /* x1 */
435                     inv, /* x2 */
436                     c, /* x3 */
437                     m, /* x4 */
438                     rsig, /* x5 */
439                     &challengemsg->r2, /* c */
440                     &y2, /* w2 */
441                     pairedkey, /* pairedkey */
442                     p2, /* pubkey */
443                     rdnfp, /* rdnfp */
444                     ) == 1);
445             }
446         }

```

Listing 5.31 Core function of *response\_challenge\_received*

```

460 unsigned char n32[32] = {
461     0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
        ↪ 0xff, 0xfe,
462     0xba, 0xae, 0xdc, 0xe6, 0xaf, 0x48, 0xa0, 0x3b, 0xbf, 0xd2, 0x5e, 0x8c, 0xd0, 0x36,
        ↪ 0x41, 0x41
463 };
464 unsigned char b[32];
465 void *ser;
466 int ret = 0;
467 int overflow = 0;
468 size_t size;
469 mpz_t m, n, sigs;
470 secp256k1_ge sigr, pge;
471 secp256k1_paillier_pubkey *p1;
472 secp256k1_scalar r, s, mes;
473
474 ARG_CHECK(ctx != NULL);
475 ARG_CHECK(sig != NULL);
476 ARG_CHECK(msg != NULL);
477 ARG_CHECK(params != NULL);
478 ARG_CHECK(p != NULL);
479 ARG_CHECK(pub != NULL);
480 ARG_CHECK(msg32 != NULL);
481 p1 = secp256k1_paillier_pubkey_get(p);
482 ret = secp256k1_eczkp_pi2_verify(
483     ctx,                    /* ctx */
484     termmsg->pi2,           /* pi2 */
485     zkp,                    /* zkp */
486     termmsg->mu2,           /* m1 */
487     termmsg->mu,            /* m2 */
488     callmsg->alpha,         /* m3 */
489     callmsg->zeta,          /* m4 */
490     &challengemsg->r2,      /* c */
491     pairedpub,             /* w2 */
492     p1,                    /* pubkey */
493     pairedkey               /* pairedkey */
494 );
495 if (ret) {
496     secp256k1_scalar_set_b32(&mes, msg32, &overflow);
497     ret = !overflow && secp256k1_pubkey_load(ctx, &pge, pub);
498     if (ret) {
499         secp256k1_pubkey_load(ctx, &sigr, &params->r);
500         secp256k1_fe_normalize(&sigr.x);
501         secp256k1_fe_normalize(&sigr.y);
502         secp256k1_fe_get_b32(b, &sigr.x);
503         secp256k1_scalar_set_b32(&r, b, &overflow);
504         VERIFY_CHECK(!secp256k1_scalar_is_zero(&r));
505         VERIFY_CHECK(overflow == 0);
506         mpz_inits(m, n, sigs, NULL);
507         secp256k1_paillier_decrypt(m, termmsg->mu, p);
508         mpz_import(n, 32, 1, sizeof(n32[0]), 1, 0, n32);
509         mpz_mod(sigs, m, n);
510         ser = mpz_export(NULL, &size, 1, sizeof(unsigned char), 1, 0, sigs);
511         secp256k1_scalar_set_b32(&s, ser, &overflow);
512         if (!overflow
513             && !secp256k1_scalar_is_zero(&s)
514             && secp256k1_ecdsa_sig_verify(&ctx->ecmult_ctx, &r, &s, &pge, &mes)) {
515             secp256k1_ecdsa_signature_save(sig, &r, &s);
516         } else {
517             memset(sig, 0, sizeof(*sig));
518         }
519     }
520     mpz_clears(m, n, sigs, NULL);
521     secp256k1_scalar_clear(&r);
522     secp256k1_scalar_clear(&s);
523     secp256k1_scalar_clear(&mes);
524 }
525 secp256k1_paillier_pubkey_destroy(p1);
526 return ret;

```

Listing 5.32 Core function of *terminate\_received*





## 6 | Further research

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetur id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

### 6.1 Hardware wallets

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetur id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.



## 7 | Conclusions

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetur id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.



# A | Docker Configuration

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

```
1 version: '2.1'
2 services:
3   levee:
4     command: >
5       bash -c "echo Container ready! Sleep 100000000... && sleep 100000000"
6     privileged: true
7     container_name: levee
8     build: ./levee
9     image: levee/levee
10    volumes:
11      - ./levee/shared:/shared
```

*Listing A.1*    *caption*

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.



# List of Figures

4.1	Adaptation of $\Pi$ to ECDSA . . . . .	15
4.2	Adaptation of $\Pi$ verification to ECDSA . . . . .	16
4.3	Adaptation of $\Pi'$ to ECDSA . . . . .	17
4.4	Adaptation of $\Pi'$ verification to ECDSA . . . . .	18





# List of Tables



# List of sources

5.1	Add argument into <code>configure.ac</code> to enable the module . . . . .	23
5.2	Define constant <code>ENABLE_MODULE_THRESHOLD</code> if module enable . . . . .	23
5.3	Include implementation headers if <code>ENABLE_MODULE_THRESHOLD</code> is defined . . . . .	23
5.4	Set threshold module to experimental into <code>configure.ac</code> . . . . .	24
5.5	Include specialized Makefile if threshold module is enable . . . . .	24
5.6	Specialized Makefile for threshold module . . . . .	24
5.7	Implementation of a DER lenght parser . . . . .	25
5.8	Implementation of a DER sequence parser . . . . .	26
5.9	Implementation of a DER sequence serializer . . . . .	26
5.10	Implementation of a DER lenght serializer . . . . .	27
5.11	DER schema of a Paillier public key . . . . .	28
5.12	DER parser of a Paillier public key . . . . .	28
5.13	DER schema of a Paillier private key . . . . .	29
5.14	DER parser of a Paillier private key . . . . .	29
5.15	DER schema of an encrypted message with Paillier cryptosystem . . . . .	29
5.16	Implementation of encryption with Paillier cryptosystem . . . . .	30
5.17	Function signature for Paillier nonces generation . . . . .	30
5.18	Implementation of decryption with Paillier cryptosystem . . . . .	30
5.19	Implementation of homomorphic addition with Paillier cryptosystem . . . . .	31
5.20	Implementation of homomorphic multiplication with Paillier cryptosystem . . . . .	31
5.21	DER schema of a Zero-Knowledge parameters sequence . . . . .	32
5.22	DER schema of a Zero-Knowledge $\Pi$ sequence . . . . .	32
5.23	DER schema of a Zero-Knowledge $\Pi'$ sequence . . . . .	33
5.24	Function signature for ZKP CPRNG . . . . .	33
5.25	Function signature to generate ZKP $\Pi$ . . . . .	34
5.26	Function signature to generate ZKP $\Pi'$ . . . . .	34
5.27	Function signature to validate ZKP $\Pi$ and $\Pi'$ . . . . .	35
5.28	Implementation of <code>call_create</code> function . . . . .	36
5.29	Implementation of <code>call_received</code> function . . . . .	37
5.30	Implementation of <code>challenge_received</code> function . . . . .	38
5.31	Core function of <code>response_challenge_received</code> . . . . .	40
5.32	Core function of <code>terminate_received</code> . . . . .	41
A.1	caption . . . . .	47



# Bibliography

- [1] 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.
- [2] 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](https://doi.org/10.1007/978-3-319-39555-5_9).
- [3] 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>.
- [4] Steven Goldfeder et al. “Securing Bitcoin wallets via a new DSA/ECDSA threshold signature scheme”. In: 2015.
- [5] 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>.



# Glossary

**DSA** Digital Signature Algorithm. 12, 14

**EC** Elliptic Curves. 12

**ECDSA** Elliptic Curve Digital Signature Algorithm. 12, 14

**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.. 12

**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). 12