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, February 2, 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	Prof. Fariba Moghaddam Bützberger
Advisor	Dean, HES-SO//Master

Acknowledgments

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer 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, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer 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

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, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer 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

2.1 Actors

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer 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`

2.1.1 Users

2.1.2 Miners

2.1.3 Developpers

2.2 Blockchain

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer 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, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer 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, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer 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 improvments

2.4.2 Layer-two applications

3 Payment channels, a micro-transaction network

Concept Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer 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

3.1 Types of payment channel

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer 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, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer 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, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer 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 ECDSA asymmetric threshold scheme

Threshold cryptography has been discussed since a long time already, many cryptographic scheme like RSA or Paillier [6, 7] exists but sometines it is difficult to use them in real case application. Since Bitcoin become popular, people lose funds because they lost keys or their keys have been hacked. And since then researches have been done to secure Bitcoin wallets [4, 2]. But the biggest problem today in Bitcoin that slow down the adoption of threshold cryptosystem is the complexity of creating an efficient and flexible scheme for Elliptic Curve Digital Signature Algorithm (ECDSA). Most recent researches are focused on finding more efficient and more generic scheme, but fortunately a scheme fulfilling perfectly the needs described into the previous chapter required to improve payment channel in Bitcoin already exists. Nevertheless this scheme describe how to perform a Digital Signature Algorithm (DSA) threshold signature and so needs to be adapted.

The scheme analysed, transformed, and implemented in the following has been presented by MacKenzie and Reiter in their paper "Two-Party Generation of DSA Signatures" [1]. This scheme is also the basis of several other papers cited before. Their construction of a threshold signature scheme with ECDSA is based on a simple multiplicative shared secret and homomorphic encryption to keep the private values unknown by the other player. The homomorphic encryption used as exemple in the paper and choosed for the implement is the Paillier cryptosystem [3]. The following chapter describe how to adapt the scheme form DSA to ECDSA and introduce some fundamental building blocks needed for a real case scenario like hierarchical deterministic threshold wallet or deterministic signatures.

Contents

4.1 Reminder

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

4.1.1 Elliptic curves

Bitcoin use EC cryptography for securing his transaction. ECDSA—based on the DSA proposal by the National Institute of Standards and Technology (NIST)—over the curve secp256k1—proposed by the Standards for Efficient Cryptography Group (SECG)—is used.

Secp256k1 curve

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

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

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

The curve order n define the number of element (points) generated by the generator G on the curve. An exponentiation of the generator $g^a \mod p$ 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 $dP = P_1 + P_2 + \cdots + P_d$.

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

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

4.1.2 Paillier cryptosystem

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

Encryption

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

Decryption

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

4.1.3 Signature schemes

Digital Signature Algorithm

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), $\texttt{hash}$ the hashing function, m the message, and $x \in \mathbb{Z}_q$ the private key.

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

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

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

Elliptic Curve Digital Signature Algorithm

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

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

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

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

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

Schemes' analysis

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

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

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

The previous postulate is used to adapt zero-knowledge proofs from DSA to ECDSA hereafter. This postulate has not been further researched by lack of time.

4.2 Threshold scheme

The "Two-party generation of DSA signatures" scheme presented by MacKenzie and Reiter, as mentionned before, is an asymmetric scheme, i.e. at the end of the protocol only the initiator can retreive the full signature. The scheme proposed is a cryptographic (1,2)-threshold, i.e. one player can be corrupted on the two players and the scheme remain safe. It is worth noting that this is qualified as an optimal (t, n)-threshold scheme, i.e. $t = n - 1$, because if only one honest player remain the safety is guarantee.

As also mentionned before, the scheme correspond to the same requirement of a Bitcoin 2-out-of-2 multi-signatures script. This means that it is possible to use it to improve the payment channels. However, it is necessary to adapt the scheme and particulary the zero-knowledge proofs construction to ECDSA. The approach is explained in the following.

The presented scheme use a multiplicative shared secret and a multiplicative shared private random value. The secret x is shared between Alice and Bob, so that Alice holds the secret value $x_1 \in \mathbb{Z}_q$ and Bob $x_2 \in \mathbb{Z}_q$ such that $x \equiv x_1 x_2 \pmod q$.

Along with the public key y, $y_1 \equiv g^{x_1} \pmod{p}$ and $y_2 \equiv g^{x_2} \pmod{p}$ are public. Alice holds a private key, hereinafter mentionned as sk, corresponding to a public encryption key pk. Alice also knows a public encryption key pk' for which she does not know the private key sk'. Here the Paillier homomorphic cryptosystem is used as the encryption scheme, but it is worth noting that others homomorphic encryption systems can be use to implement the scheme. Alice and Bob know a set of parameters used for both zero-knowledge proofs.

Hereinafter, the initialisation is not taken into account. The choice was made to decrease the amount of work and because the implementation is not part of the cryptographic `C` library. This part can be a further research for an other thesis.

4.2.1 Adapting the scheme

Except for the zero-knowledge proofs, the adaptation is trivial and require just the same adaptation from DSA signature scheme and ECDSA signature scheme, i.e. compute the r value with the curve. The following figure describe the adapted scheme. Messages stay the same and are not repeated. The postulate 4.1.3 is used to perform the adaptation.

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

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

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

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

Figure 4.1 Adapted protocol for ECDSA

4.2.2 Adapting zero-knowledge proofs

Initialy 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 (a different one per proof.) For ECDSA some of these values become points and some equations need to be adapted. Points are serialized in the long form, 65 bytes starting with `0x04` and two 32 bytes number for the coordinates (x, y). As mentionned is the original paper, the variables names are not consistent with the first part of the paper. Hereinafter the variable names follow the same notation as the original paper and are therefore no longer consistent with the previous pages.

Zero-knowledge proof Π

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

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

Figure 4.2 The proof Π

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

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

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

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

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

Figure 4.3 Adaptation of Π*'s verification in ECDSA*

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

Figure 4.4 Adaptation of Π's construction in ECDSA

Zero-knowledge proof Π'

The second zero-knowledge proof is created by Bob to prove to Alice that he acted honestly according to the protocol. The proof states that the point $\mathcal{R}_2$ is generated accordingly to the value z_2 and then to the value k_2, the public key $\mathcal{Q}_2$ is related to the values z_2 and $x_2 z_2$, and the encrypted values μ and μ' are correctly composed.

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

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

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

Figure 4.5 *The proof* Π'

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

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

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

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

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

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

$$
\begin{array}{ll}
s_1 \leftarrow e x_1 + \alpha & t_1 \leftarrow e x_2 + \delta \\
s_2 \leftarrow (r_1)^e \beta \mod N' & t_2 \leftarrow e\rho_3 + \epsilon \mod n \\
s_3 \leftarrow e\rho_1 + \gamma & t_3 \leftarrow (r_2)^e \mu \mod N \\
s_4 \leftarrow e x_1 x_4 + \alpha & t_4 \leftarrow e\rho_2 + \nu \\
 & t_5 \leftarrow e x_3 + \sigma \\
 & t_6 \leftarrow e\rho_4 + \tau \\
 & t_7 \leftarrow e x_2 x_5 + \delta
\end{array}
$$

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

Figure 4.6 Adaptation of Π''s construction in ECDSA

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

Verify $s_1, t_1 \in \mathbb{Z}_{n^3}$	$\mathcal{V}_1 \leftarrow (t_1 + t_2) \cdot \mathcal{D} + (-e) \cdot \mathcal{Y}$
Verify $t_5 \in \mathbb{Z}_{n^7}$	$\mathcal{V}_2 \leftarrow s_1 \cdot \mathcal{W}_2 + t_2 \cdot \mathcal{D} + (-e) \cdot \mathcal{Y}$
$\mathcal{U}_1 \leftarrow s_1 \cdot \mathcal{C} + (-e) \cdot \mathcal{W}_1$	$v_3 \leftarrow (m_3)^{s_4} (m_4)^{t_7} g^{nt_5} (t_3)^N (m_2)^{-e} \mod N^2$
$u_2 \leftarrow (g')^{s_1} (s_2)^{N'} (m_1)^{-e} \mod (N')^2$	$v_4 \leftarrow (h_1)^{t_1} (h_2)^{t_4} (z_2)^{-e} \mod \tilde{N}$
$u_3 \leftarrow (h_1)^{s_1} (h_2)^{s_3} (z_1)^{-e} \mod \tilde{N}$	$v_5 \leftarrow (h_1)^{t_5} (h_2)^{t_6} (z_3)^{-e} \mod \tilde{N}$

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

Figure 4.7 Adaptation of Π' *verification to ECDSA*

4.3 Threshold Hierarchical Determinitic Wallets

Hierarchical deterministic wallets are sophisticated wallets in wich fresh keys can be generated from a previous key. Adapting hierarchical deterministic wallets with a threshold scheme can be achieve 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 player publish his own public share point, every one can compute the master public key. The multiplicative sharing is more communication applicant because the computation of the public key is sequential instead of parallel.

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

4.3.1 Private parent key to private child key

The function `CKDpriv` compute a child extended private key from the parent extended private key. The derivation can be *hardened.* This proposal differ from the `BIP32` [8] standard in the chain derivation proccess. The `ser` function and `point` function are the same as described in `BIP32`.

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

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

4.3.2 Public parent key to public child key

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

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

4.3.3 Child key share derivation

It is asume that one of the players P_i is designated as the leader L. The function `CKSD` compute a threshold child extended key share from the threshold parent extended key share for the derivation index k. It is worth noting that only the leader L use `CKDpriv` and if the derivation is *hardened*, i.e. if $k \geq 2^{31}$, a special case occured and a round of communication is needed. Let's define `CKSD` for $k < 2^{31}$:

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

such that:

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

and then $\forall i \in P_i$:

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

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

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

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

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

This scheme is sufficient for the payment channels, a threshold key used for the $\mathtt{Multisig}_i$ address with a root derivation path `m/44'/0'/a'/0'` is negociated at the opening of the channel (variable `a` is related to the channel account number between the client and the provider as shown in the paper). Then the index `i` in the paper is used to derive each addresses without require any communication. It is worth noting that the root derivation path can also be simplify at `m/a'` or even `m/` because the compatibility with a standard wallet is not anymore a requirement. Noted that the version `m/a'` is more flexible and allows multiple channels between a client and a provider with only one threshold key.

4.3.4 Proof-of-concept implementation

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

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

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

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

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

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

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

Listing 4.1 Result of using threshold HD wallet

A share is composed of four main information: (i) the secret share, (ii) the chain code, (iii) the tag for the master share, and (iv) the threshold public key. The threshold public key address can be set after computation. The derive function `d`

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

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

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

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

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

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

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

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

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

Listing 4.2 Demonstration of using threshold HD wallet

derives with `CKDpub` or `CKDpriv` depending on the master tag and return a new share for a given index. The path derivation function `derive` take a path and generate the chain of shares. In this Implementation, if a share not tagged as master try to derive a path with an `hardened` index, an exception is raised and the proccess stops. But in real world case, a communication process must take place to complete the derivation.

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

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

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

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

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

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

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

    def derive(self, path):
        path = string.split(path, "/")
        if path[0] == "m":
            path = path[1:]
            share = self
            for derivation in path:
                if "'" in derivation:
                    i = int(derivation.replace("'", "")) + pow(2, 31)
                    share = share.d(i)
                else:
                    i = int(derivation)
                    share = share.d(i)
            return share
        else:
            return False
```

Listing 4.3 Construction of a share for a threshold HD wallet

4.4 Threshold deterministic signatures

One of the simplest way to compromise the private key in ECDSA is to select a weak pseudo random number generator for k or even worst, select a static value for k. This problem already append to Sony in December 2010 when a group of hacker calling itself *fail0verflow* announced recovery of the ECDSA private key used to sign software for the PlayStation 3.

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

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

But this issue can be prevented by deterministic generation of k, as described by the RFC 6979 [9]. The random value k can be generated deterministicly by using an `HMAC` function such that the parameters are the private key and the message to sign.

The other positive point is that signatures for the same key paire and the same message are deterministic, i.e. if we sign multiple times the same message, the signature remain the same. This is aslo a big advantage in Bitcoin to help preventing transaction malleability. The deterministic signature construction can also be applied to the threshold scheme with the same properties.

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

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

5 Implementation in Bitcoin-core secp256k1

As mentionned before, Bitcoin use eliptic curve cryptography (ECC) for signing transactions. When the first release of Bitcoin core appeared in the early 2009, the cryptographic computations was performed with the OpenSSL library. Some years after a project started with the goal of replacing OpenSSL and creating a custom and minimalistic `C` library for cryptography over the curve secp256k1. This library is now available on GitHub at `bitcoin-core/secp256k1` project and it is one of the most optimized, if not the most optimized, library for the curve secp256k1. It is worth noting that this library is also used by other major crypto-currencies like Ethereum, so extending the capabilities of this library is a good choice to attract other cryptographer to have a look and increase the amount of reviews for this thesis.

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

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

Contents

5.1 Configuration

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

5.1.1 Add new experimental module

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

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

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

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

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

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

Listing 5.2 Define constant `ENABLE_MODULE_THRESHOLD` if module enable

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

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

Listing 5.3 Include implementation headers if `ENABLE_MODULE_THRESHOLD` is defined

The module is set to experimental to avoid enabling it without explicitly agree to build experimental code. If experimental is set to yes a warning is display during the configuration process, if experimental is not set and any experimental module is enable an error message is display and the process failed.

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

Listing 5.4 Set threshold module to experimental into ***configure.ac***

5.1.2 Configure compilation

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

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

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

Listing 5.5 Include specialized Makefile if threshold module is enable

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

```
include_HEADERS += include/secp256k1_threshold.h
noinst_HEADERS += src/modules/threshold/der_impl.h
noinst_HEADERS += src/modules/threshold/paillier.h
noinst_HEADERS += src/modules/threshold/paillier_impl.h
noinst_HEADERS += src/modules/threshold/paillier_tests.h
noinst_HEADERS += src/modules/threshold/eczkp.h
noinst_HEADERS += src/modules/threshold/eczkp_impl.h
noinst_HEADERS += src/modules/threshold/eczkp_tests.h
noinst_HEADERS += src/modules/threshold/threshold_impl.h
noinst_HEADERS += src/modules/threshold/threshold_tests.h
```

Listing 5.6 Specialized Makefile for threshold module

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

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

5.2 DER parser-serializer

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

5.2.1 Sequence

The sequence data structure holds a sequence of integers and/or octet strings. The sequence start with the constant `0x30` and is followed by the content lenght and the content itself. A lenght could be in the short form or the long form. If the content number of bytes is shorter to `0x80` the lenght byte indicate the lenght, if the content is equal or longer than `0x80` the seven lower bits 0 to 6 where $\texttt{byte} = \{\texttt{b}_7, \ldots, \texttt{b}_1, \texttt{b}_0\}$ indicate the number of followed bytes which are used for the lenght.

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

Listing 5.7 Implementation of a `DER` lenght parser

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

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

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

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

Listing 5.8 Implementation of a `DER` sequence parser

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

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

Listing 5.9 Implementation of a `DER` sequence serializer

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

5.2.2 Integer

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

5.2.3 Octet string

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

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

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

copy the header and the octet string into a new memory space, and return the pointer with the total lenght. The parser implementation copy the content and set the conent lenght, the position index, and the offset.

5.3 Paillier cryptosystem

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

5.3.1 Data structures

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

Public keys

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

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

Listing 5.11 *`DER` schema of a Paillier public key*

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

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

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

Listing 5.12 *`DER` parser of a Paillier public key*

Private keys

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

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

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

Listing 5.13 `DER` *schema of a Paillier private key*

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

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

Listing 5.14 `DER` *parser of a Paillier private key*

Encrypted messages

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

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

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

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

5.3.2 Encrypt and decrypt

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

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

Listing 5.16 Implementation of encryption with Paillier cryptosystem

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

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

Listing 5.17 Function signature for Paillier nonces generation

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

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

Listing 5.18 Implementation of decryption with Paillier cryptosystem

5.3.3 Homomorphism

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

Addition

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

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

Listing 5.19 Implementation of homomorphic addition with Paillier cryptosystem

Multiplication

Multiplication $\times_{pk}$ over ciphertexts can be performed with $D_{sk}(E_{pk}(m_1, r_1)^{m_2} \mod n^2) = m_1m_2 \mod n$, the implementation is straight forward in this case. The nonce value from the ciphertext is copied in the resulted encrypted message for not lose information after opperations.

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

Listing 5.20 Implementation of homomorphic multiplication with Paillier cryptosystem

5.4 Zero-knowledge proofs

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

5.4.1 Data structures

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

Zero-Knowledge Parameters

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

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

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

Zero-Knowledge Proof Π

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

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

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

Zero-Knowledge Proof Π'

Zero-Knowledge Proof Π' is composed of the same named values as ZKP Π plus five new ones. The construction of the proof is based on Π but needs more than values to express all the proven statements. Again, the point y is a point serialized as an uncompressed public key in an octet string and a version number is added for future compatibility purposes. Two functions are provided in the module to parse and serialize a `ECZKPPiPrim` `DER` schema.

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

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

5.4.2 Generate proofs

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

A CPRNG function is required to generate both proofs. This function generate random number in $\mathbb{Z}_{max}$ and $\mathbb{Z}^*_{max}$. The `flag` argument indicate which case is treated, `STD` or `INV`. If the function have not access to a good source of randomness or cannot generate a good random number a zero is returned, otherwise a one is returned.

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

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

Listing 5.24 *Function signature for ZKP CPRNG*

Zero-Knowledge Proof Π

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

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

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

Listing 5.25 *Function signature to generate ZKP* Π

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

Zero-Knowledge Proof Π'

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

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

Listing 5.26 *Function signature to generate ZKP* Π'

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

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

5.4.3 Validate proofs

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

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

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

Listing 5.27 Function signature to validate ZKP Π and Π'

5.5 Threshold module

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

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

Nomenclature

A proposal for exchanged messages names and actions is done in this report. Players P_1 and P_2 represent the initiator and collaborator. Player P_1 initialize the communication and ask P_2 to collaborate on a signature, if P_2 collaborates and the protocol end successfully P_1 retreive the signature.

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

5.5.1 Create call message

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

```c
int secp256k1_threshold_call_create(const secp256k1_context *ctx, secp256k1_threshold_call_msg *callmsg, secp256k1_threshold_signature_params *params, const secp256k1_scalar *secshare, const secp256k1_paillier_pubkey *paillierkey, const unsigned char *msg32, const secp256k1_nonce_function noncefp, const secp256k1_paillier_nonce_function pnoncefp) {
    int ret = 0;
    int overflow = 0;
    unsigned char nonce32[32];
    unsigned char sec32[32];
    unsigned int count = 0;
    secp256k1_scalar privinv;

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

Listing 5.28 Implementation of `call_create` function

5.5.2 Receive call message

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

```
int secp256k1_threshold_call_received(const secp256k1_context *ctx, secp256k1_threshold_challenge_msg *challengemsg, secp256k1_threshold_signature_params *params, const secp256k1_threshold_call_msg *callmsg, const secp256k1_scalar *secshare, const unsigned char *msg32, const secp256k1_nonce_function noncefp) {
    int ret = 0;
    int overflow = 0;
    unsigned int count = 0;
    unsigned char k32[32];
    unsigned char sec32[32];

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

Listing 5.29 Implementation of `call_received` function

5.5.3 Receive challenge message

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

```c
int secp256k1_threshold_challenge_received(const secp256k1_context *ctx, secp256k1_threshold_response_challenge_msg *respmsg, secp256k1_threshold_signature_params *params, const secp256k1_scalar *secshare, const secp256k1_threshold_challenge_msg *challengemsg, const secp256k1_threshold_call_msg *callmsg, const secp256k1_eczkp_parameter *zkp, const secp256k1_paillier_pubkey *paillierkey, const secp256k1_eczkp_rdn_function rdnfp) {
    int ret = 0;
    unsigned char k32[32];
    secp256k1_pubkey y1;
    secp256k1_scalar privinv;

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

Listing 5.30 Implementation of `challenge_received` function

5.5.4 Receive response challenge message

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

The point r is normalized and the coordinate $r.x$ is get (modulo n). The `hash` is multiplied with z_2 and the coordinate $r.x$ is multiplied with $x_2 z_2$. A value x_3 where $n|x_3$ is added to the cipher to hide information about the secret share and the secret random. In ECDSA $s = k^{-1}(m + rx) \mod n$, so the ciphertext match the requirement as demonstrated below:

$$\begin{aligned} D_{sk}(\mu) &\equiv (\alpha \times mz_2) + (\zeta \times rx_2 z_2) + (x_3) \pmod n \\ &\equiv (z_1 \times mz_2) + (x_1 z_1 \times rx_2 z_2) \pmod n \\ &\equiv (z_1 z_2 m) + (x_1 z_1 r x_2 z_2) \pmod n \\ &\equiv z_1 z_2 (m + rx_1 x_2) \pmod n \\ &\equiv z(m + rx) \pmod n \\ &\equiv k^{-1}(m + rx) \pmod n \end{aligned}$$

5.5.5 Receive terminate message

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

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

Listing 5.31 Core function of `response_challenge_received`

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

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

Listing 5.32 Core function of ***`terminate_received`***

6 Further research

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

6.1 Side-channel attack resistant implementation and improvements

The proposed implementation into the library `secp256k1` rely on `libgmp` for all complex mathematical calculus and `libgmp` is not strong against side channel attacks, and it is normal, the library has not been developed for that particular purpose. Therefore, a other implementation need to take the place and handle, in constant time and constant memory if possible, the mathematical calculus part. This is a big improvement that can be done, or must be done, before hoping to use the module is some real case scenario.

6.1.1 Second hash function

The current implementation use the hash function `SHA256` implemented into the library `secp256k1` for Π and Π'. This is not complient with the orignial paper requirements, a other hash function must be implemented and used for Π'.

6.1.2 Paillier cryptosystem

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

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

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

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

with precomputations

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

Paillier cryptosystem can be adapted to EC cryptography as shown in the paper "Trapdooring Discrete Logarithms on Elliptic Curves over Rings" by Pascal Paillier [10]. It is worth nothing however that the curve construction is different than the curve used to sign and so the code base cannot can not necessarily be reused.

6.1.3 Zero-knowledge proofs

Non-interactive zero-knowledge proofs are a big research field. The article "From Extractable Collision Resistance to Succinct Non-interactive Arguments of Knowledge, and Back Again" by Bitansky, Nir and Canetti, Ran and Chiesa, Alessandro and Tromer, and Eran [11] introduced the acronym zk-SNARK for zero-knowledge Succinct Non-interactive ARgument of Knowledge that are the backbone of the Zcash protocol [12]. In the recent paper "Bulletproofs: Efficient Range Proofs for Confidential Transactions" [13] a new non-interactive zero-knowledge proof protocol with very short proofs and without a trusted setup is proposed. Further research could be done to adapt the zero-knowledge proof construction and migrate to a more generic approach, to remember that the zero-knowledge proof construction proposed in the original paper dates from the early 2000s, progress has been made since.

6.2 Hardware wallets

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

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

Usually, when a new Bitcoin wallet is created, a list of words called mnemonic is shown to the user as a backup of his wallet key. The mnemonics are between twelve and twenty-four and each word represent 11 bits of the primary seed [14], for a threshold key it is not possible to represent all the data in the same way given the size of the key (near 4.5 Kb). A other way to display and transmit these information is needed to increase usability. Further research could be done to find a better way to represent and display a threshold key.

The master tag is not included in the `DER` schema. Is the key itself responsible to store this information or this information is a part of the setup and can be stored outside, this question can be deepened.

6.3 More generic threshold scheme

As previously mentionned, research have been done to generalize and find an optimal (t, n) threshold in ECDSA [6, 7]. These papers base their work on the scheme chosen by the implementation, so a deeper analysis could be performed to assess the needed changes to adapt the current implementation to a generic threshold.

6.4 Schnorr signatures

In the paper "Efficient Identification and Signatures for Smart Cards" published in CRYPTO 1989, C.P. Schnorr propose the "Schnorr signature algorithm" [15]. The Schnorr signature is considered the simplest digital signature scheme to be provably secure in a random oracle model [5, 16]. Thus, Bitcoin developers and researchers have a strong interest for this specific scheme since some years now. Schnorr signatures could greatly reduce the size of the signaure from 65 bytes (ECDSA in `DER` format) to 40 bytes.

With the arrival of SegWit, it is now possible to have script version, thus it is more easy to indroduce new `OP_CODE` and so introduce a new signature validation scheme. However, this will not invalidate the present work and researches because of the specific nature of its application.

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

7 Conclusions

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer 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.

```yaml
version: '2.1'
services:
  levee:
    command: >
        bash -c "echo Container ready! Sleep 100000000... && sleep 100000000"
    privileged: true
    container_name: levee
    build: ./levee
    image: levee/levee
    volumes:
        - ./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

List of Tables

List of sources

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.

[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.

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

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

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

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

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

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

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

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

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

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

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

Glossary

DSA Digital Signature Algorithm. 11, 12, 14–16

EC Elliptic Curves. 12, 16, 51

ECDSA Elliptic Curve Digital Signature Algorithm. 11, 12, 14–16, 28, 52, 53

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