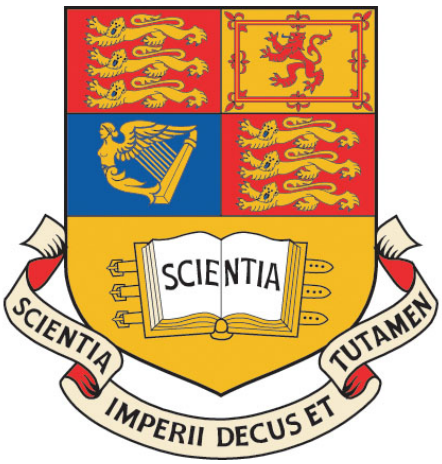
Imperial College London

Department of Electrical and Electronic Engineering

Final Year Project Report 2017

---




| | |
|---|---|
| Project Title: | **Anonymous Bitcoin Transactions via Homomorphic Encryption** |
| Student: | **Ayah Kajouk** |
| CID: | **00825291** |
| Course: | **EEE4** |
| Project Supervisor: | **Dr Wei Dai** |
| Second Marker: | **Dr Bruno Clerckx** |

**Abstract**

Bitcoin transactions are public and can be seen by anyone in the blockchain. For this reason, bitcoins are not fungible. This means that some coins which can be linked to specific wallet addresses are less valuable than others, such as the coins used in the Silk Road black market. As a consequence, some users are willing to pay extra for "clean coins". However, fungibility is important in a decentralized payment network. This issue can be solved by introducing private transactions in Bitcoin. This project implements an alternative design of a Bitcoin-compatible anonymous payment hub based on the recently released TumbleBit. In TumbleBit, an untrusted, unlinkable and anonymous middleman is used to link a payer and payee. This way, the system is kept decentralized while allowing for private transactions. The main focus of this project is to compare three encryption systems, Elgamal, ECC and RSA where the latter is the underlying encryption algorithm of the already existing TumbleBit. All three encryption algorithms share the multiplicative homomorphic property and this allows us to introduce a proof of concept of a similar design of TumbleBit, using Elgamal and ECC instead of RSA. Time complexity features of the new systems are also analyzed.

# Contents

# Chapter 1

# Introduction

Bitcoin is a peer-to-peer electronic cash system; it allows online payments to be sent from one person to another without the need for a trusted third party. It achieves this by timestamping the bitcoin transactions and thus forming a public record that cannot be changed. These records are hashed into an ongoing chain, known as the blockchain. The blockcahin will include all the details of these transactions making them available to any bitcoin user.

There have been many discussions regarding the economic concept of fungibility. This refers to the interchangeability property of a bitcoin [1] and the idea that no bitcoin should be deemed less valuable due to its past use or ownership. As Blockstream's CEO Adam Back has remarked: "Some of the exchanges and wallets are using tracing services, and [if there is a connection to illicit activity] up to four hops deep away from you, they will freeze your account." [1]. He has also concluded that "a lack of fungibility, if allowed to continue, could ultimately 'leak' into the system, affecting bitcoin's 'permissionlessness', or the ability of any parties to join and use the technology without censorship." [1]

However, due to the design of blockchain and the fact that the transaction histories of all bitcoins are traceable, all bitcoins' fungibility are at risk. Blockchain analysis can also link bitcoin addresses, i.e. users to their transactions and thus harming their privacy.

TumbleBit was proposed in 2016 by Ethan Heilman, Foteini Baldimtsi, Sharon Goldberg and Alessandra Sacfuro and it is a tool to ensure anonymity within Bitcoin. It allows many users to set up payment channels and send payments to one another using an untrusted intermediary.

TumbleBit applies cryptographic puzzles to replace the direct bitcoin transactions be-

tween the intermediary and users. [2] describes the mechanism for a scenario where Person A sends money to Person B: "If Person B can provide the solution to the puzzles, he can claim a bitcoin. Person A buys the answers for these puzzles from the intermediary for a bitcoin. Person A then sends the answer to Person B as payment, which Person B accepts since he can claim a bitcoin with it."

TumbleBit's puzzles are RSA encryptions of secret messages chosen by the intermediary. The aim of this project is to replace the homomorphic RSA encryption scheme with another widely known homomorphic algorithm, the Elgamal encryption. The motivation for eliminating RSA from TumbleBit is to enhance the security of the system and to increase the speed of the private transactions. The reasons for choosing Elgamal will be discussed in section 4.1. The main focus of this paper is to design a new TumbleBit system with Elgamal and determine if it performs better than the original system. For this, we will evaluate the computational complexity features of both and compare.

This paper includes a large section of technical background material on the mechanism of TumbleBit and homomorphic functions. This is followed by the Analysis and Design chapter where we capture the requirements of the new system, separate the process into a number of stages and propose a high-level design for each stage. In the Implementation chapter, we describe our proposed system that will satisfy the criteria stated previously. We also implement a third version of TumbleBit which will be analogous to the Elgamal algorithm, but based on elliptic curves. Finally, we evaluate the overall performance and compare the original and new TumbleBits.

# Chapter 2

# Background: TumbleBit

TumbleBit is an untrusted, Bitcoin-compatible anonymous payment hub. It was developed by a group of researchers in Boston University and the product was released in the second half of 2016. As opposed to other privacy-focused cryptocurrency products, TumbleBit is not a separate, individual cryptosystem; it is a tool to introduce private transactions to the already existing and widely known Bitcoin cryptosystem.

## 2.1 TumbleBit payment phases

TumbleBit achieves privacy by replacing the on-blockchain public transactions with off-blockchain private puzzle solving schemes. This allows a payer to send $Q$ bitcoins to an untrusted middleman, the Tumbler who will pass on the $Q$ coins to the payee. This method avoids recording the direct transaction between the payer and payee on the blockchain and thus achieves full anonymity.

Since this project is a proof of concept, we will be using an example where Alice will pay only 1 bitcoin to Bob. Using TumbleBit, both of them interact only with the Tumbler and not with each other. The puzzle $z$ is generated through Tumbler-Bob and solved through Alive-Tumbler interactions. One solved puzzle $z$ will result in the payment of one bitcoin from Alice to the Tumbler and then from the Tumbler to Bob. A TumbleBit transaction has three phases which are briefly presented in figure 2.1.

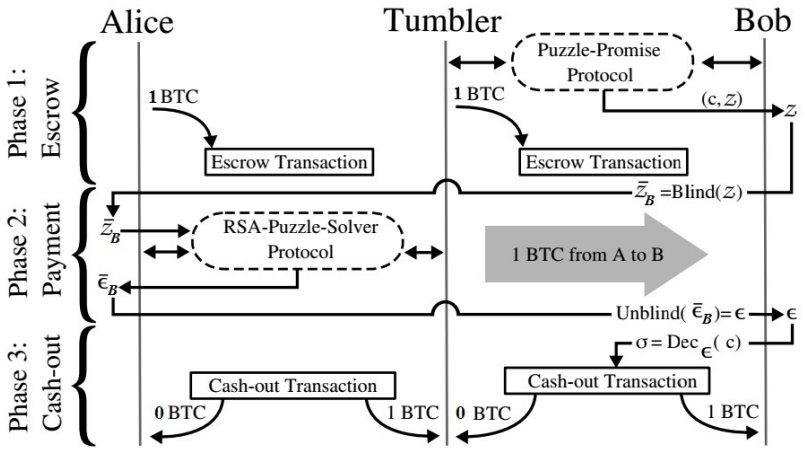



Figure 2.1: Overview of TumbleBit design[3].

### 2.1.1 Escrow phase

The first phase, the Escrow Phase implements an on-blockchain transaction. Firstly, two payment channels are set up: one between Alice and the Tumbler and another between the Tumbler and Bob. Secondly, a so-called puzzle-promise protocol is implemented.

#### 2.1.1.1 Payment channels

Bob asks the Tumbler to set up an escrow transaction[1] to escrow 1 bitcoin to Bob, as shown in figure 2.1. We denote this transaction as $T_{escr}(T, B)$. When Alice sets up a payment channel with the Tumbler, she also escrows 1 bitcoin in an escrow transaction denoted as $T_{escr}(A, T)$.

#### 2.1.1.2 Puzzle-promise protocol

This protocol is based on a cryptographic puzzle-solving scheme. The promiser generates a promise and a puzzle and he will pay one bitcoin to the receiver in exchange for the solution to this puzzle [3]. In TumbleBit, Bob and the Tumbler engage in a puzzle-promise protocol where the Tumbler promises to pay 1 bitcoin to Bob if Bob solves puzzle $z$ given by the Tumbler.

[1] An escrow transaction is a conditional transaction that can only be claimed by another transaction if the latter is signed by two previously specified bitcoin addresses [3].

The promise $c$ is an encryption of the Tumbler's *ECDSA-Secp256k1* signature[2] on the cash-out transaction $T_{cash}(T, B)$. This transaction is set up by Bob and will fulfill $T_{escr}(T, B)$ when signed by both Bob and the Tumbler. The reason for choosing ECDSA-Secp256k1 is to ensure compatibility with the BitCoin system. The signature is denoted as $\sigma$. Figure 2.1 shows the output of this protocol $(c, z)$ being sent to Bob.

The puzzle is an RSA cryptographic encryption scheme where the following equation encrypts a secret message $\epsilon$. This is shown in (2.1)

$$z = f_{RSA}(\epsilon, pk, N) = \epsilon^{pk} \bmod N \tag{2.1}$$

where $f_{RSA}$ is the RSA encryption function with secret key $sk$, public key $pk$ and large prime number $N$.

Bob needs $\epsilon$ because it is the secret key to $c$, i.e. it is needed to decrypt $c$ and get $\sigma$. After the puzzle-promise protocol, the escrow transactions are established on the blockchain.

### 2.1.2 Payment phase

The payment phase is an off-blockchain phase. Bob interacts with Alice to find the solution to puzzle $z$. In order to do that, he sends $z$ to Alice who will engage in a second protocol with the Tumbler, called the puzzle-solver protocol. However, before sending $z$ to Alice, Bob blinds $z$ by picking a random blinding factor $r_B \in Z_N^*$ [3] and using (2.2) to receive the blinded version of $z$, denoted by $\overline{z}_B$. Blinding is used to hide ciphertexts while allowing decryption using same the secret key. It is discussed in more details in section 4.2.

$$\overline{z}_B = r_B^{pk} \times z \bmod N \tag{2.2}$$

When Alice receives $\overline{z}_B$ she solves the blinded puzzle via the puzzle-solver protocol, as shown in figure 2.1. This is a fair exchange ensuring that Alice transfers one bitcoin to the Tumbler if and only if he gives a valid solution to the puzzle $\overline{z}_B$ [3]. After the Tumbler solves the puzzle, Alice sends the blinded solution $\overline{\epsilon}_B$ to Bob, who uses his secret blinding factor $r_B$ to unblind it and receive $\epsilon$:

$$\epsilon = \frac{\overline{\epsilon}_B}{r_B} \bmod N \tag{2.3}$$

[2]Elliptic Curve Digital Signature Algorithm and it uses Bitcoin's elliptic curve Secp256k1 [4]

[3]$Z_N^*$ denotes the set of $Z_N$ without zero, where $Z_N$ is the set of all integers mod $N$

Bob accepts Alice's solution only after successful verification:

$$\epsilon^{pk} = z \bmod N \tag{2.4}$$

Another way to think of this phase is to imagine that the Tumbler's signature $\sigma$ is the payment Bob is working for. $T_{cash}(T, B)$ can only be claimed by Bob if the Tumbler has his signature on it too, which is $\sigma$. He will need to solve $z$ to get $\epsilon$, use $\epsilon$ to decrypt $c$ and decrypt $c$ to get $\sigma$.

### 2.1.3 Cash-out phase

After Bob receives $\epsilon$, decrypts $c$ and receives $\sigma$, he can post $T_{cash}(T, B)$ on the blockchain to receive one bitcoin from the Tumbler, as seen in figure 2.1.

## 2.2 Puzzle-promise protocol

The puzzle-promise protocol has been introduced briefly in section 2.1.1.2. Figure 2.2 presents a more detailed walk through.

Recall that the aim of this protocol is to provide Bob with a puzzle-promise pair $(c, z)$ where the puzzle's solution will eventually allow Bob to acquire the Tumbler's signature $\sigma$ on $T_{cash}(T, B)$. Since TumbleBit is an untrusted system, we need to take every possible scenario into account.

- If Bob was simply given $(c, z)$, he would have no guarantee that the promise $c$ is encrypting the correct signature $\sigma$.
- However, the Tumbler cannot prove the correctness of $c$ by revealing the value of $\sigma$ to Bob because Bob could use $\sigma$ to claim one bitcoin from the escrow transaction $T_{escr}(T, B)$ without being paid by Alice.

As a solution to this problem, the protocol designed by TumbleBit exploits a so-called *cut and choose technique*[4]. This term is explained in the following step-by-step analysis of the protocol in figure 2.2.

[4]"A cut-and-choose protocol is a two-party protocol in which one party tries to convince another party that some data he sent to the former was honestly constructed according to an agreed-upon method."[5]

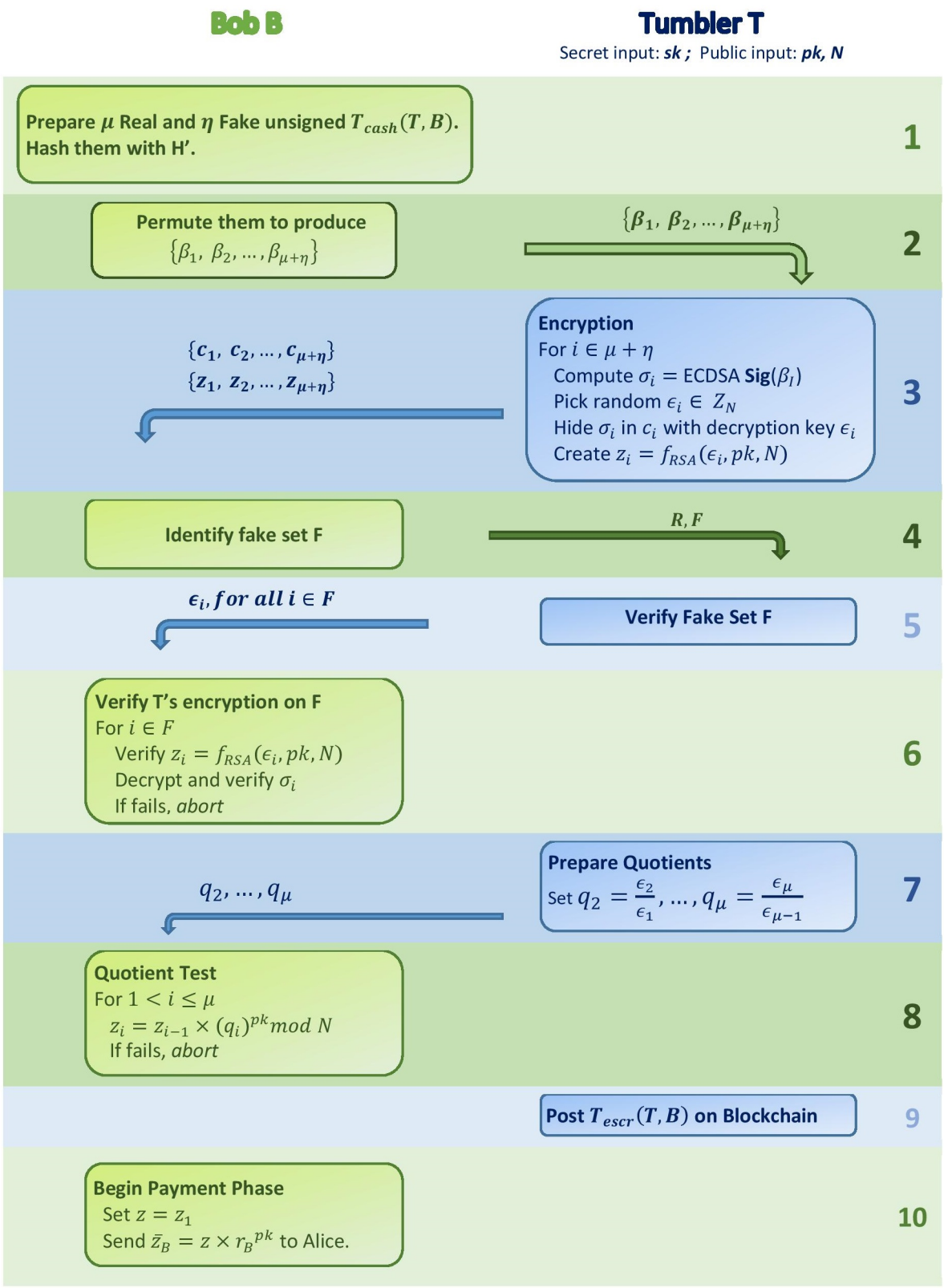



Figure 2.2: Puzzle-promise protocol: simplified version of [3].

The basic idea is to have the Tumbler compute $(\mu + \eta)$ many $(c, z)$ pairs and ask him to reveal the puzzle solution for $\eta$ of them so Bob can verify if those $\eta$ $(c, z)$ pairs are computed correctly. The Tumbler will not know which $\eta$ pairs will be verified by Bob, therefore, according to [3], the probability for the Tumbler to cheat by creating corrupt puzzle-promise pairs is equal to $\frac{1}{\binom{\mu+\eta}{\eta}}$. However, the $\eta$ puzzle-promise pairs have to be encryptions of $\eta$ fake transactions, otherwise Bob would gain the solution without Alice paying one bitcoin to the Tumbler. Steps 1 and 2 of figure 2.2 show Bob creating $\mu$ real and $\eta$ "fake" cash-out transactions, permuting and sending them over to the Tumbler.

In step 3, the Tumbler creates $(\mu + \eta)$ different $\sigma$'s to sign all of Bob's transactions without knowing which ones are fake or real. He creates $(\mu+\eta)$ promises to hide all $\sigma$'s with randomly chosen encryption keys $\epsilon$'s and sends these over to Bob together with the corresponding RSA puzzles.

Next, Bob wants to verify the $\eta$ puzzle-promise pairs. To do that, he needs to identify the "fake" set of transactions (step 4), wait for the Tumbler to verify that the $\eta$ transactions are indeed fake (step 5) and then receive the $\eta$ values of $\epsilon$ to verify the corresponding $(c, z)$ pairs (step 6).

According to [3], the cut-and-choose algorithm guarantees that $B$ knows that *at least one* of the $\mu$ puzzle-promise pairs is correctly formed. However, he cannot know which one exactly. Additionally, he can only send one puzzle to Alice to solve through the puzzle-solver protocol. For this reason, a *quotient-chain method* is introduced in step 7. The tumbler sends the quotients, defined in (2.5), to Bob. Note, knowing the value of quotients does not reveal the puzzle solutions. They ensure that solving one puzzle will give the solutions to all the other $\mu - 1$ puzzles. This is shown in (2.6).

$$q_2 = \frac{\epsilon_2}{\epsilon_1}, \quad \ldots \quad , q_\mu = \frac{\epsilon_\mu}{\epsilon_{\mu-1}} \tag{2.5}$$

$$\epsilon_i = \epsilon_1 \times q_2 \times, ...., q_i \tag{2.6}$$

It is important to note that a cheating Bob will not be able to claim more than one bitcoins by obtaining more than one correct puzzle solution using the quotients defined above. This is due to the way $T_{escr}(T, B)$ is defined. It offers only 1 bitcoin in exchange for the ECDSA-Secp256k1 signature and can be used only once during this protocol execution [3].

At the end of the puzzle-promise protocol, Bob picks one of the $\mu$ puzzle-promise pairs, blinds it with his secret blinding factor $r_B$ and sends it over to Alice (step 10 in figure 2.2).

## 2.3 Puzzle-solver protocol

TumbleBit's puzzle-solver protocol ensures fair-exchange for both Alice and the Tumbler where Alice pays one bitcoin to the Tumbler in exchange for a valid solution to the RSA puzzle $\overline{z}_B$ (previously blinded by Bob). The step-by-step description of the protocol and the realization of fair exchange are explained in sections 2.3.1 and 2.3.2 respectively.

### 2.3.1 The protocol

Recall that the puzzle-promise protocol ends with Bob blinding his puzzle $z$ before sending it to Alice. Thus, in the puzzle-solver protocol the Tumbler will not be able to recognize $z$ from $\overline{z}_B$.

Figure 2.3 presents a detailed protocol walk through. The idea is to have the Tumbler solve the puzzle $\overline{z}_B$ to obtain $\overline{\epsilon}_B$, encrypt $\overline{\epsilon}_B$ to get $\mu^{prg}$ and send this to Alice. She will only be able to decrypt $\mu^{prg}$ if she commits to send the one bitcoin to the Tumbler. However, how can Alice be sure that $\mu^{prg}$ is the encryption of the correct solution $\overline{\epsilon}_B$?

The cut and choose method is used here as well. In steps 1 and 2 of figure 2.3 Alice blinds $\overline{z}_B$ $m$ times and also creates $n$ "fake" puzzles. She randomly permutes and sends them over to the Tumbler to solve. The idea here is similar to section 2.3.1; the probability for the Tumbler to cheat by providing a corrupt puzzle solution is $\frac{1}{\binom{m+n}{n}}$ [3].

In step 4, the Tumbler decrypts the received puzzles to obtain the double-blinded solution $\overline{\overline{\epsilon}}$ (blinded by Bob first then by Alice). He then hashes this under a randomly chosen key $k^{prg}$ and obtains the ciphertext $\mu^{prg}$. He also hashes $k^{prg}$ to obtain $h = H(k^{prg})$. The pair $(\mu^{prg}, h)$ is then sent to Alice. The hashing functions $H$ and $H^{prg}$ are defined in appendix A.

Alice will want to verify if the $n$ fake puzzles have been decrypted correctly. To do this, she will ask the Tumbler to reveal the $n$ values of $k^{prg}$. But first, she will have to identify the fake set which is done in step 5. This is necessary because if Alice received the key $k^{prg}$ of a correctly formed $(\mu^{prg}, h)$ pair, she would be able to obtain a puzzle solution without paying for it. As such, Alice can only have the "fake" puzzle solutions revealed. After the Tumbler verifies the fake set in step 6, he sends the preimages of $h$ of the fake puzzles (i.e. the corresponding $k^{prg}$ values).

Alice then verifies the solutions of the fake puzzles by recomputing the RSA encryption equation in step 7. If the verification process is successful it is very likely that the

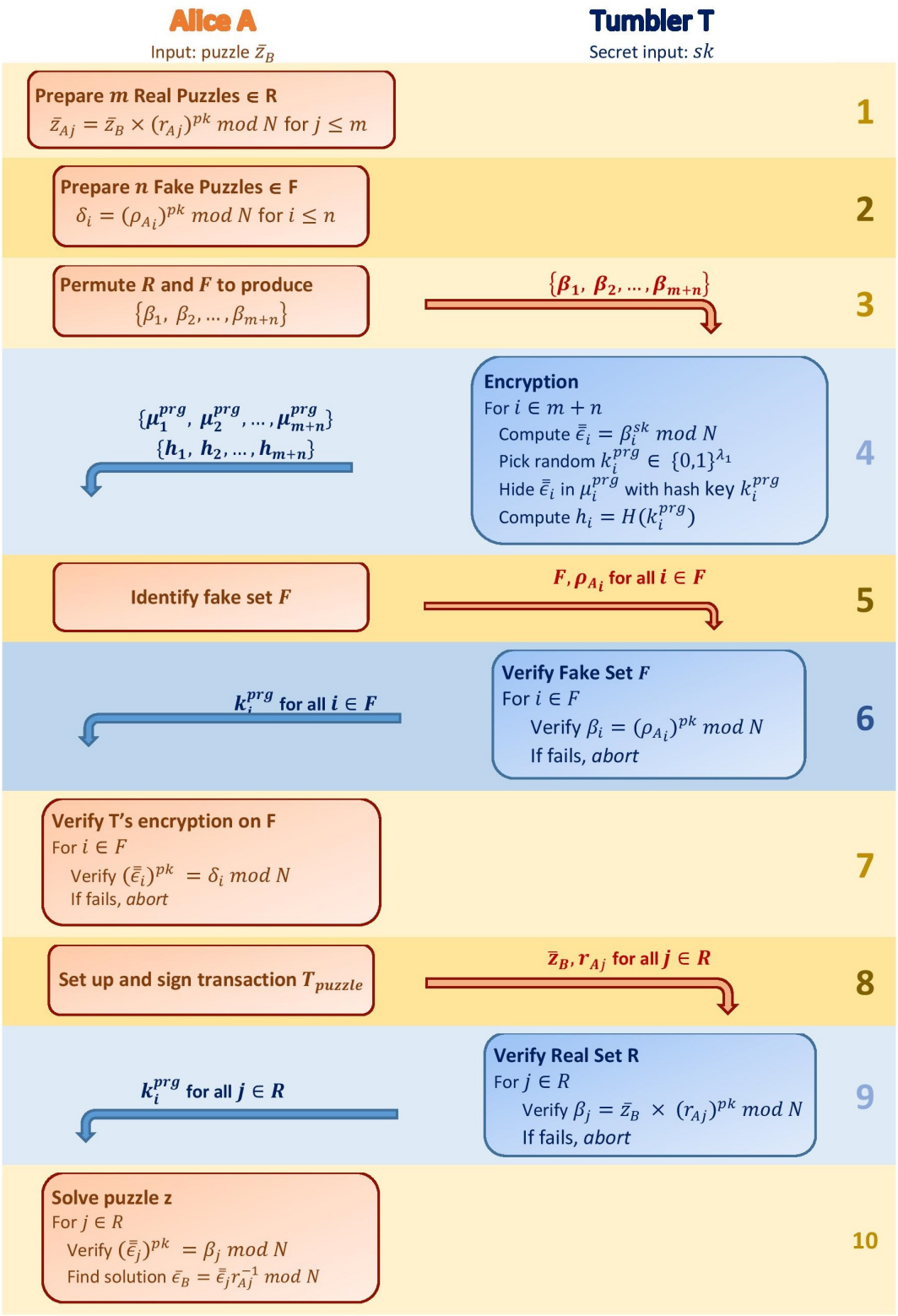



Figure 2.3: Puzzle-solver protocol: simplified version of [3].

Tumbler has calculated the correct solutions for all the $(m + n)$ puzzles since for a cheating Tumbler it is difficult to guess all the $n$ pairs and form them properly so he would not get caught, while providing corrupt solutions for all the $m$ unopened pairs, so that he can claim a bitcoin without actually solving the puzzle [3].

In step 8, Alice sets up an off-blockchain transaction, denoted as $T_{puzzle}$. This points to the escrow transaction $T_{escr}(A, T)$ which was set up in the escrow phase. It is a form of commitment from Alice and it offers one bitcoin if the following condition is met:"the fulfilling transaction is signed by the Tumbler and has all preimages $h_j$ for all $j \in R$" [3].

Afterwards, Alice sends $\overline{z}_B$ and her blinding factors $r_A$ for all the $m$ real puzzles to the Tumbler, so that he can verify that all $\overline{z}_{Ai}$ for all $i \in R$ are the blinded versions of the same puzzle $\overline{z}_B$. This is important because Alice could cheat and have each of the real $(\mu^{prg}, h)$ pairs open to the solution of a different puzzle thus allowing Alice to obtain several solutions for the price of one. If this verification stage is successful, the Tumbler sends all the $m$ preimages to Alice.

As mentioned earlier in section 2.3.1, the cut and choose technique ensures that at least one of the real $(\mu^{prg}, h)$ pairs is formed correctly. However, how can Alice know which one? Luckily, this can be verified in a simple way: she recomputes the RSA encryption equation for all the received puzzle solutions to find the one that is correctly formed. This is shown in step 10. Alice unblinds $\overline{\overline{\epsilon}}$ and sends $\overline{\epsilon}_B$ to Bob. Note, the cut and choose technique entails that the probability for an uncorrectly formed $(\mu^{prg}, h)$ pair to occur is very little, however, the protocol does not take chances and therefore checks before sending it over to Bob.

### 2.3.2 Fair exchange

The protocol has to ensure fairness for both Alice and the Tumbler. Fairness for Alice means that the Tumbler will earn 1 bitcoin if and only if she obtains a correct solution. Fairness for the Tumbler entails that after one execution of this protocol, Alice will learn the valid solution to at most one puzzle of her choice.

From the detailed description provided in 2.3.1 above, we can conclude that if the Tumbler agrees to solve puzzle $\overline{z}_B$ then he gets paid and Alice receives one solution. If the Tumbler refuses, Alice gets the 1 bitcoin back and the Tumbler gets nothing.

Fairness for the Tumbler is captured: Alice can only get solution if she sends one bitcoin. Fairness for Alice is also captured: the Tumbler can only receive 1 bitcoin if he solves a puzzle correctly. (Further proofs for the security of the protocol can be found in appendix of [3].)

# Chapter 3

# Background: Homomorphic Encryption

## 3.1 Homomorphic Encryption

This paper focuses on the analysis of homomorphic encryption. Here we briefly explain what is meant by homomorphism and give a general definition for multiplicatively homomorphic functions as this will be relevant to this project since both RSA and Elgamal are multiplicatively homomorphic.

Two algebraic groups[1] $(G_1, o_{G_1})$ and $(G_2, o_{G_2})$ are said to be homomorphic when the mapping from $G_1$ to $G_2$ is performed via function $f : G_1 \rightarrow G_2$, such that for both elements $g_1$ and $g_2$ in $G_1$, it follows that

$$f(g_1 \quad o_{G_1} \quad g_2) = f(g_1) \quad o_{G_2} \quad f(g_2) \tag{3.1}$$

If (3.2) holds then $f$ is additively homomorphic and if (3.3) is true then it is a multiplicatively homomorphic function.

$$f(g_1) \quad o_{G_2} \quad f(g_2) = f(g_1 \quad + \quad g_2) \tag{3.2}$$

$$f(g_1) \quad o_{G_2} \quad f(g_2) = f(g_1 \quad \times \quad g_2) \tag{3.3}$$

[1] "An algebraic group $G$ is a set with its own operation $o_G$ that combines its elements $g_1$ and $g_2$ to return $(g_1 o_G g_2)$ which will be a third element of $G$."[6]

## 3.2 RSA Encryption Algorithm

RSA was the first public-key cryptosystem that was practically applicable, developed by Ron Rivest, Leonard Adleman and Adi Shamir in 1977. Similarly to other public-key cryptosystems, the RSA algorithm consists of three main components: key generation, encryption and decryption. In this section these are going to be described separately. this will be helpful for later analysis because RSA is the heart of TumbleBit's Payment Phase and it is heavily integrated into the puzzle-solver protocol described in section 2.3.1.

### 3.2.1 The Algorithm

**Key generation.** In RSA, there will be a set of public keys available for everyone to share and a set of private keys to be kept hidden by the receiver of the message. The key generation process starts with picking random prime numbers $L$ and $s$. For security reasons, the bit-length of the two numbers should be the same. The modulus of the encryption function $n$ will be the product of the two prime numbers $n = Ls$. The algorithm also requires Euler's totient function of $n$ which is the number of positive integers less than $n$ and coprime with $n$ [6]. This is computed using (3.4).

$$\phi(n) = \phi(L)\phi(s) = (L-1)(s-1) \tag{3.4}$$

The public-key exponent of the encryption function will be $pk$, which is a randomly chosen integer in the interval $1 < pk < \phi(n)$ in a way so that $gcd(pk, \phi(n)) = 1$ is true [6]. For efficiency purposes $pk$ should have a short bit-length and small Hamming weight[2]. The private key $sk$ will be the modular multiplicative inverse of $pk$ mod $\phi(n)$, i.e.: $sk = pk^{-1} \bmod \phi(n)$ [6]. The set of public keys is $(n, pk)$, the set of private keys is $(n, sk)$ and the set of parameters to be kept secret is $(L, s, \phi(n))$.

**Encryption.** The sender will convert the message $\varepsilon$ into an integer $\epsilon$, which will have a magnitude within the interval $1 < \epsilon \ll n$. He computes the ciphertext $z$ using (3.5) [6].

$$z = \epsilon^{pk} \bmod n \tag{3.5}$$

**Decryption.** The receiver recovers the message $\epsilon$ using the private decryption exponent $sk$, such that

$$\epsilon = z^{sk} \bmod n = \epsilon^{sk \times pk \bmod \phi(n)} \bmod n = \epsilon^{1 \bmod \phi(n)} \bmod n = \epsilon \bmod n \tag{3.6}$$

[2]Hamming weight of x: $wt(x) = |\{i : x_i \neq 0\}|$ [7]

## 3.3 Elgamal Encryption Algorithm

The Elgamal cryptosystem is a multiplicatively homomorphic public-key encryption algorithm developed by Egyptian Taher El Gamal in 1985. Just as RSA, it also consists of three main components: the key generation, the message encryption and decryption. Elgamal is defined over a cyclic group $G$ which will influence its security properties. This is further analyzed in section 3.3.4. This background information will be useful for our analysis on how to implement TumbleBit with Elgamal in section 5.3.

### 3.3.1 The Algorithm

**Key generation.** The secret key $x$ will be chosen randomly by the receiver, so that $x \in (1, ..., ord - 1)$, where $ord$ is the order of $G$. The set of public keys to be published before encryption will consist of

- the cyclic group $G$
- $ord$, the order of $G$
- the prime modulo $p$
- $g$, one of the generators of $G$
- $y$ where $y = g^x \bmod p$

A general discussion is provided in section 3.3.2 to demonstrate how a generator can be found. In section 3.3.3 a more specific description is given to show how this is done for Elgamal. These will be referred to in later analysis of the proposed Elgamal-based TumbleBit protocols.

**Encryption.** To encrypt a message $\epsilon$, the sender will use the previously published keys to find the two ciphertexts $a$ and $b$ and send them to the receiver. A random value for the variable $k$ has to be chosen by the sender so that $k \in (1, ..., ord - 1)$ and it has to be kept secret. $a$ and $b$ can be computed using (3.7).

$$a = g^k \bmod p \; ; \quad b = \epsilon \times y^k \bmod p \tag{3.7}$$

**Decryption.** The receiver decrypts the ciphertexts $(a, b)$ using his private key $x$ and computing (3.8).

$$\begin{aligned} \epsilon = b \times a^{ord-x} \bmod p = \epsilon \times y^k \times g^{k(ord-x)} \bmod p \\ = \epsilon \times g^{kx} \times g^{k.ord} \times g^{-kx} \bmod p = \epsilon g^{kx-kx} \times \bmod p = \epsilon \end{aligned} \tag{3.8}$$

### 3.3.2 Finding the generator of a cyclic group $G$

The order of cyclic group $G$ and the order of all the subgroups of $G$ will be a factor of $p-1$. We introduce a simple example to demonstrate this as well as to explain what is meant by subgroup. Consider $Z_p$ where $p = 7$. Figure 3.1 shows a table of all the $g$ elements of $Z_p$ in each row and all $k$ values in each column. The expression $g^k \bmod p$ is evaluated for every $g$ and $k$.

| g \ k | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 2 | 1 | 2 | 4 | 1 | 2 | 4 | 1 |
| 3 | 1 | 3 | 2 | 6 | 4 | 5 | 1 |
| 4 | 1 | 4 | 2 | 1 | 4 | 2 | 1 |
| 5 | 1 | 5 | 3 | 6 | 2 | 4 | 1 |
| 6 | 1 | 6 | 1 | 6 | 1 | 6 | 1 |

Figure 3.1: Evaluating $g^k \bmod p$ in $Z_p$ where $p = 7$

The first value of $g$ is zero and normally this is excluded from the considered values of the generator. The set of elements of $Z_p$ without zero is denoted as $Z_p^*$. The next generator $g = 1$ always returns 1. The remaining elements: **2, 3, 4, 5 and 6** return sets of results for the $g^k \bmod p$ expression with set magnitudes: **3, 6, 3, 6 and 2** respectively. These magnitudes are all factors of 6, which is $p-1$, where $p$ is the modulo. Each of the smaller groups with number of elements less than $p-1$ are called the subgroups of $Z_p^*$. The generators of $Z_7^*$ are 3 and 5.

### 3.3.3 Finding the generator for Elgamal

Here we discuss some properties of the generators used for Elgamal. This will help to understand some concepts described in section 5.3. For security reasons, both the magnitude of $p$ and $ord$, the order of the subgroup of $G$ must be large prime numbers. We pick the value of $p$ so that $p = 2ord + 1$, where $ord$ is also a randomly chosen large prime number. This way, the subgroups of $G$ will have orders $ord$, 2, 1 or $p-1$, since $p - 1 = 2ord$.

To find a generator of order $p-1$, we pick a random generator of $Z_p^*$ and test if it satisfies the equation $g^{\frac{p-1}{2}} = -1 \bmod p$. If yes, it will most likely be a generator of

order $p-1$ since it is very unlikely to find a generator of order 1 or 2. However, for safety reasons, we can check that $g^2$ is not equal to 1.

### 3.3.4 Security of Elgamal

**Discrete Logarithm Problem.** In cryptography, the discrete logarithm problem (DLP) is applied as a source for "one way functions". Informally, a one way function is defined as "a function $f : X \to Y$ where it is easy to compute $f(x)$ when $x \in X$ is given, however, it is difficult to find the value of $x$ when $y \in Y$ is given, where $f(x) = y$" [8].

Elgamal's encryption equations are considered to be one-way functions and their security is based on the discrete logarithm problem. Based on [8], the formal definition of DLP can be described the following way: "given $g \in G$ and $a \in \langle g \rangle$, finding an integer $x$ such that $g^x = a$ is computationally difficult"[8], where $g$ and $G$ are same parameters defined in section 3.3.1 and $\langle g \rangle$ is a cyclic subgroup, generated by $g$ in a similar way described in section 3.3.2.

The discrete logarithm problem will be referred to in section 4.1.2 as part of the security justification of the proposed Elgamal verification system for TumbleBit.

**Decisional Diffie-Hellman.** The decisional Diffie-Hellman (DDH) assumption is a test which determines if the Elgamal encryption scheme is secure. This depends on the properties of the cyclic group $G$ used for encryption.

It states that "considering a cyclic group $G$ of order $q$ and generator $g$, given $g^a$ and $g^b$ where $a$ and $b$ are chosen uniformly and independently from $Z_{ord}$, the value of $g^{ab}$ "looks like" a random element in $G$. In other words, the two probability distributions $(g^a, g^b, g^{ab})$ and $(g^a, g^b, g^c)$ are computationally indistiguishable, where $a, b, c$ are randomly and independently chosen from $Z_{ord}$" [9].

Elgamal has to hold this assumption to ensure it is polynomially secure which will be discussed in section 4.1.

## 3.4 Elgamal Encryption using Elliptic Curve Cryptography

It is possible to implement the Elgamal Encryption scheme using Elliptic Curve Cryptography (ECC) by replacing the above-mentioned cyclic group $G$ with an Elliptic

Curve (EC). This provides an alternative design for TumbleBit where the protocols can be implemented using an Elgamal Elliptic Curve Cryptosystem. In this section we are going to briefly discuss how elliptic curves should be formed and provide additional background information in appendix D. This information will be used to show how Elgamal algorithm is implemented using ECC in section 5.4.

### 3.4.1 Elliptic Curves over Finite Field

ECC is based on elliptic curves defined over finite fields $p$ and its equation is shown in (3.9). It defines a finite field which consists of the points satisfying (3.9) along with its identity element, infinity[3]. The total number of these points plus infinity is defined as the order of the elliptic curve $O_E$[4].

$$Y^2 = X^3 + \lambda_1 X + \lambda_2 \bmod p \tag{3.9}$$

$A$ and $B$ determine the shape of the curve and they have to satisfy equation (3.10) in order for the curve to be applicable in cryptography. (3.10) ensures that the EC does not have any repeated factors.[5]

$$4\lambda^3 + 27\lambda^3 \neq 0 \bmod p \tag{3.10}$$

### 3.4.2 The Algorithm

After choosing a prime modulus $p$ and forming a suitable elliptic curve EC mod $p$, we can implement the Elgamal ECC which will be analogous to the Elgamal algorithm described in section 3.3.1.

According to [10], "Modular multiplication and modular exponentiation in finite field cryptography are equivalent to ECC operations of addition of points on an elliptic curve and multiplication of a point on an elliptic curve by an integer respectively".

**Key generation.** The receiver chooses a random secret key $x_E$ from $\{1, ..., O_E - 1\}$ and a point $g_E$ which lies on the EC and has an order of $O_E$ [11]. These two parameters are the equivalents of secret key $x$ and generator $g$ of Elgamal.

A point $y_E$ is calculated using (3.11). This will also be a point on EC and it is analogous to Elgamal's public key $y$. The receiver publishes the public keys $(p, g_E, y_E)$.

$$y_E = x_E \times g_E \tag{3.11}$$

[3]Refer to appendix D.2 for more information on identity elements.
[4]Refer to appendix D.3 for more information on EC order.
[5]Refer to appendix D.1 for more information on repeated factors of EC.

**Encryption.** A function is used to map the message $\epsilon$ to a point $E$ on the EC [11]. Then the sender chooses a random integer $k_E$ from $\{1, ..., O_E - 1\}$ and computes two points on EC:

$$\begin{aligned} A &= k_E \times g_E \\ B &= E + k_E \times y_E \end{aligned} \tag{3.12}$$

where $a_E$ and $b_E$ are analogous to Elgamal's encryptions $a$ and $b$ respectively and they are sent to the receiver.

**Decryption.** The message is decrypted using (3.13). The original message $\epsilon$ can be calculated using the inverse of the previous mapping function.

$$\begin{aligned} B - x_E \times A = (E + k_E \times y_E) - x_E \times (k_E \times g_E) = \\ = E + k_E \times x_E \times g_E - k_E \times x_E \times g_E = E \end{aligned} \tag{3.13}$$

# Chapter 4

# Analysis and Design

## 4.1 Why Elgamal Encryption?

### 4.1.1 Homomorphism

The motivation for choosing Elgamal encryption algorithm was its multiplicative homomorphic property. This allows for a design similar to the already existing TumbleBit system. This is because the puzzle that needs to be solved by the receiver of the bitcoins in the original protocol is an RSA encryption equation. As already mentioned in section 3.1, RSA is also muliplicatively homomorphic.

### 4.1.2 Security

Elgamal has several advantages over RSA regarding its security. One of them is the polunomial security that is true for Elgamal but not for RSA and it is summarized in lemma 4.1.1. However, note that Elgamal is only conditionally secure, i.e. it has to satisfy the Diffie-Hellman assumption (DDH) which was explained in section 3.3.4.

**Lemma 4.1.1.** *"If the decisional Diffie-Hellman assumption (DDH) holds in the underlying cyclic group G then Elgamal is polynomially secure against any passive adversary whereas RSA can never be polynomially secure" [12].*

Polynomial security can be described the following scenario: we suppose we are given an arbitrary function $f$ and we choose two random messages $m_1$ and $m_2$. We are then given a ciphertext which could equal to either $f(m_1)$ or $f(m_2)$. A scheme is

polynomially secure if in polynomial time[1] you cannot decide which message is hidden in the ciphertext, with probability significantly greater than 0.5 [12].

Polynomial security is an evidently important property that every crypotsystem should hold. Deterministic encryptions cannot be polynomially secure therefore plain RSA is a polynomially insecure system. With suitable randomized padding scheme, RSA can be polynomially secure too and this is the reason why it is widely used. However, padding schemes increase key lengths significantly. We suspect that with the right design for the Elgamal TumbleBit system, we can achieve shorter key sizes, while still ensuring polynomial security. This will be determined in the Evaluation chapter.

### 4.1.3 Elliptic curves

Finally, the main advantage of Elgamal over RSA is that it allows the integration of elliptic curves. Elgamal with Elliptic Curve Cryptography (ECC) is known for faster encryption, better security and significantly smaller key sizes when compared to other cryptosystems, such as RSA [10]. This will be evaluated in section 6.5.

## 4.2 Blinding

Blinding is a technique of applying a function $f$ on an encrypted message to protect it against timing and side-channel attacks. The same operations can be performed on blinded and non-blinded versions of the ciphertext. The inverse of function $f$ can be performed on the blinded output in order to receive the original plaintext. This is demonstrated for both RSA and Elgamal encryptions where blinding is based on their multiplicative homomorphic property.

RSA is a multiplicatively homomorphic system which means that the product of two ciphertexts will be equal to the encryption of the product of the two corresponding messages. This is shown in (4.1).

$$m_1^{pk} \times m_2^{pk} \bmod n = (m_1 m_2)^{pk} \bmod n \tag{4.1}$$

Therefore, blinding of an RSA ciphertext is performed in a similar way. In TumbleBit, Bob selects a random blinding factor $\overline{r}_B$ and blinds puzzle $z$ as shown in (2.2).

Elgamal's multiplicative homomorphic property can be shown using two example en-

[1] "When the execution time of a computation, $m(n)$, is no more than a polynomial function of the problem size, $n$. More formally $m(n) = O(n^l)$ where l is a constant" [13].

cryptions:

$$(a_1, b_1) = (g^{r_1}, m_1 g^{r_1}) \text{ and } (a_2, b_2) = (g^{r_2}, m_2 g^{r_2}) \tag{4.2}$$

where $r_1$ and $r_2$ are randomly chosen from $\{1, ..., ord - 1\}$ and the other parameters correspond to section 3.3.1. The product of the two sets of ciphertexts result in the encryption of $m_1, m_2$:

$$(a_1, b_1) \times (a_2, b_2) = (g^{r_1} \times g^{r_2}, m_1 g^{r_1} \times m_2 g^{r_2}) = (g^{r_1+r_2}, m_1 m_2 g^{r_1+r_2}) \tag{4.3}$$

In the proposed TumbleBit design, Bob will select a random blinding factor $r_B$ and compute the blinded Elgamal puzzles $(\overline{a}_B, \overline{b}_B)$ the following way:

$$(\overline{a}_B, \overline{b}_B) = (a \times g^{r_B}, b \times r_B \times y^{r_B}) \tag{4.4}$$

Blinding is an important part of both TumbleBit protocols. This ensures that the untrusted Tumbler will not be able to link the payee Bob with payer Alice by comparing the puzzle he sent to Bob in the puzzle-promise protocol and the puzzle received from Alice in the puzzle-solver protocol.

## 4.3 Verification

In TumbleBit, Alice will need to verify if the puzzle solution she received from the Tumbler $\epsilon^*$ equals the correct $\epsilon$ that the Tumbler has encrypted in the puzzle-promise protocol. A cheating Tumbler would want to send a corrupt solution to Alice so when she sends $\epsilon^*$ to Bob after posting $T_{cash}(A, T)$ to the blockchain, he would not be able to claim his bitcoin. Therefore, verification is done before Alice submits $T_{cash}(A, T)$ to the blockchain.

### 4.3.1 Verification with RSA

In the original TumbleBit design, Alice verifies the Tumbler's solution by recomputing the RSA encryption equation (2.1) using the solution to be tested $\epsilon^*$ instead of $\epsilon$. Of course this is done before sending it to Bob, so in fact Alice is verifying the blinded version of the solution $\epsilon^*$. If the blinded solution is correct then the original solution has to be correct too. This is demonstrated in (4.5) where $\overline{\epsilon}^*_B$ and $\epsilon^*$ denoten the blinded and unblinded solutions to be verified respectively. If $\overline{\epsilon}^*_B$ satisfies the first equality in (4.5) then $\epsilon^*$ is the correct solution to puzzle $z$.

$$(\overline{\epsilon}^*_B)^{pk} \equiv \overline{z}_B = z \times r_B^{pk} = (\epsilon^*)^{pk} \times r_B^{pk} \tag{4.5}$$

This design is based on the deterministic property of RSA. Alice can verify the puzzle solution without the knowledge of the secret key. This is because RSA encryption of the same message $\epsilon$ will result in the same ciphertext $z$ since RSA has no randomness in its algorithm. However, this property implicates insecurity issues which were discussed in 4.1.2.

### 4.3.2 Issue with Elgamal verification

Since Elgamal is a probabilistic algorithm, verification cannot be done by recomputing the encryption equation. This is because the keys $x$ and $k$ have to be kept secret. $x$ is used for decryption; it can only be known by the Tumbler and is used during the entire transaction, i.e. the same $x$ value is used for all the $Q$ bitcoins. Whereas, the key $k$ has to be different for every puzzle since the solution $\epsilon$ can reveal the value of $g^{xk}$ by calculating $b \times \epsilon^{-1} \bmod p = g^{xk}$ and if the same k value is used for every puzzle, the next $\epsilon$ value will no longer be hidden since $\epsilon = b \times g^{-xk} \bmod p$.

Furthermore, note that the Tumbler randomly selects the value of $k$ in the puzzle-promise protocol, however, he will not know this value in the puzzle-solver protocol. In other words, if the Tumbler knew the value of $k$ for each blinded puzzle in the puzzle solver protocol, he can easily make the link between Alice and Bob and blinding would loose its importance.

It is important to emphasize that the verification process is part of the puzzle-solver protocol and it needs to check the correctness of the actual solution received from the Tumbler. It is not enough to verify if the Tumbler has the correct secret key; a cheating Tumbler could easily use the correct secret key in the verification process and at the same time send a corrupt solution to Alice. Figure 4.1 shows a high-level design of the required system. After Alice receives $\epsilon^*$ from the Tumbler, she inputs it into the verification system. She also inputs the necessary set of public parameters from the list shown in figure 4.1. At this point, $k$ is unknown to both Alice and the Tumbler and $x$ is only known by the Tumbler. The verification will be successful if the output matches the expected value which could either be a public parameter or an arbitrary random number.

In section 5.2, we propose a deign that satisfies all the criteria mentioned here.

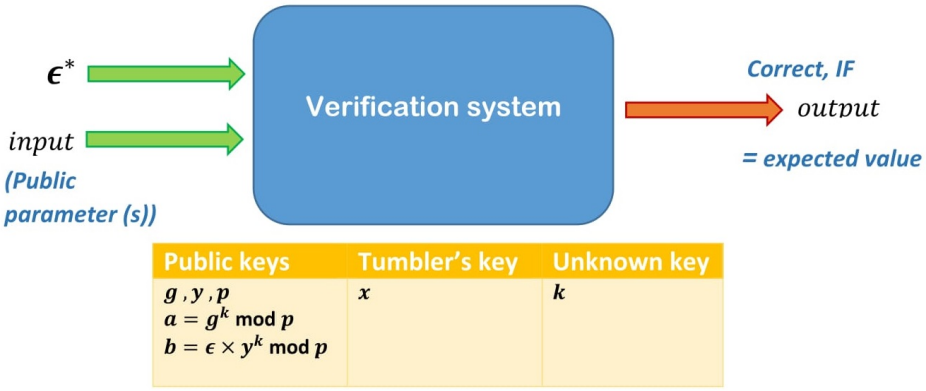



Figure 4.1: High-level system for verification of Elgamal-encrypted puzzle solution

# Chapter 5

# Implementation

In here we will show the low level design of the thingy with elgamal...

## 5.1 Quotient test

As we have mentioned earlier in section 2.2, the RSA quotient-chain technique ensures that Bob will receive the correct solution of at least one puzzle. Once he has received one puzzle solution from Alice, the quotients provided by the Tumbler will enable him to learn all the other puzzle solutions. We implement a similar design for the proposed Elgamal-based TumbleBit system. We need to ensure that the extra parameters provided to Bob will not reveal harmful information about the puzzle.

In the new puzzle-promise protocol, the tumbler will send $(\mu - 1)$ pair of parameters $(q_i, \Delta k_i)$ for $i \in \{2, ..., \mu\}$, where similarly to section 2.2, $\mu$ represents the number of real puzzles produced by the Tumbler. $q_i$ will be identical to the quotients used in the RSA implementation of TumbleBit, i.e. $q_i = \frac{\epsilon_i}{\epsilon_{i-1}}$ . $\Delta k_i$ will be defined as

$$\Delta k_2 = k_2 - k_1, \ \ldots \ , \Delta k_\mu = k_\mu - k_{\mu-1} \tag{5.1}$$

where $k_i$ represents the Elgamal random variables for each puzzle, previously defined in section 3.3.1. This additional set of parameters are needed for Bob to verify the quotients, i.e. to implement the Elgamal equivalent step of step 8 in figure 2.2.

Making all the $\Delta k_i$ values available to Bob cannot be harmful to Elgamal's security. This is because the Tumbler only provides $\mu - 1$ equations of (5.1) however there are $\mu$ unknown $k$ values to Bob.

When Bob receives all the $(q_i, \Delta k_i)$ pairs from $T$, he will verify them using the following

equation:

$$b_i = b_{i-1} \times q_i \times y^{\Delta k_i} \bmod p \tag{5.2}$$

Once Bob has the solution to one of the Elgamal puzzle pairs $(a, b)$ (earlier defined in equation 3.7), he will be able to obtain all the other puzzle solutions the same way as described in equation (2.6).

## 5.2 Verification with Elgamal

Based on the high-level design presented in section 4.3.2, an Elgamal verification system is proposed here. Figure 5.1 summarizes the verification process. The step-by-step breakdown of the algorithm is also provided followed by justification for design validity.

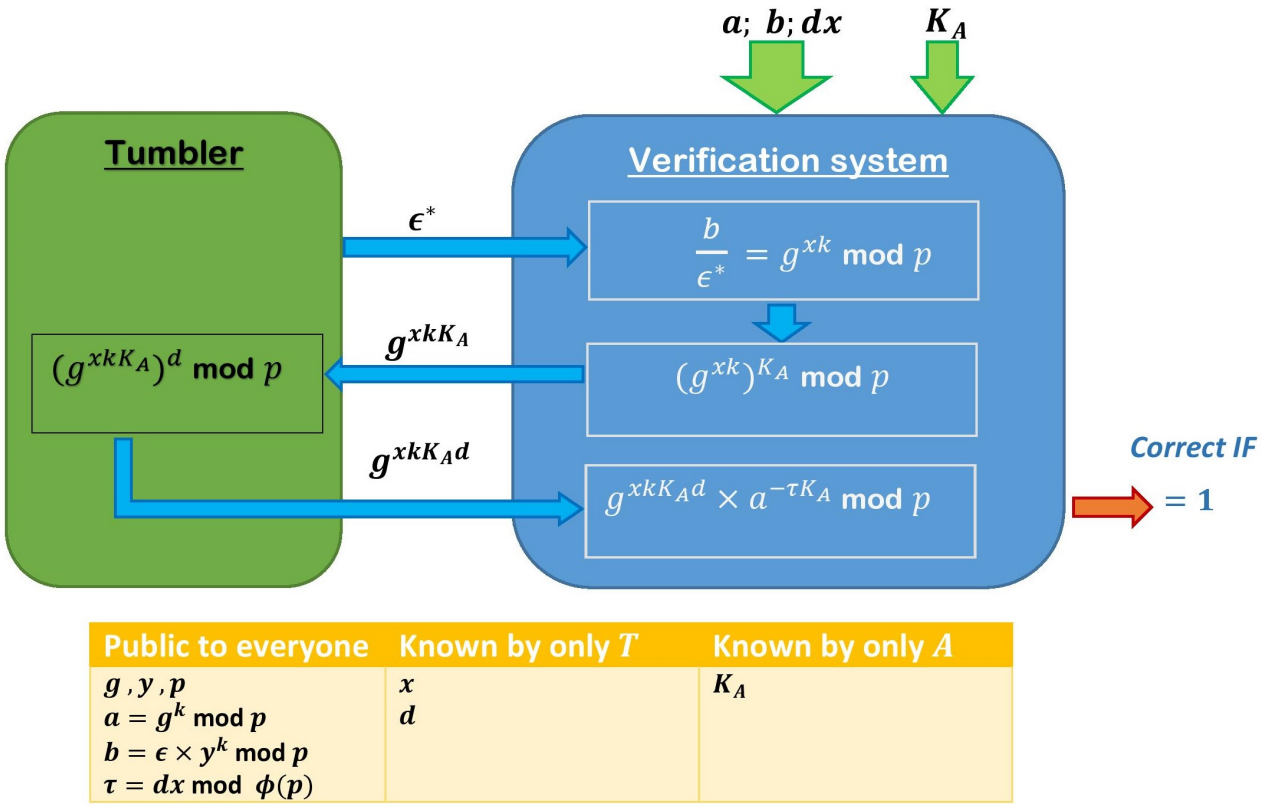



Figure 5.1: Design proposal of Elgamal verification process

This algorithm requires us to introduce additional public and private parameters to TumbleBit's system. These are

- Verifier's new secret key $K_A$ where $K_A \in \{1, ..., ord - 1\}$
- Tumbler's new secret key $d$ where $d \notin \{(p-1), ord, 2, 1, 0\}$
- Tumbler's new public key $\tau = dx \bmod \phi(p)$, where $\phi(p)$ is the totient function defined in (3.4)

The input parameters to the verification system will include the Elgamal puzzle-pair $(a, b)$, the new public key $\tau$ and the verifier's secret key $K_A$. The verification process will follow the steps shown below.

1. The Tumbler will send the decrypted message $\epsilon^*$ to the verifier.
2. The verifer calculates $\omega = \frac{b}{\epsilon^*}$ which should return $g^{xk} \bmod p$ if $\epsilon^*$ is valid, i.e. if $\epsilon^* = \epsilon$.
3. The verifier blinds $\omega$ the following way: $(\frac{b}{\epsilon^*})^{K_A} \bmod p = g^{xkK_A} \bmod p$
4. The verifier sends this to the Tumbler.
5. The Tumbler raises this to $d$, his secret key: $(g^{xkK_A})^d \bmod p = g^{xkK_Ad} \bmod p$
6. The Tumbler sends this to the verifier.
7. The verifier calculates: $g^{xkK_Ad} \times a^{-\tau K_A} = g^{xkK_Ad} \times g^{-kdxK_A} \bmod p \equiv 1 \bmod p$

### 5.2.1 Mathematical explanation for the Elgamal verification system

This method relies on two identities:

1. $a^x \bmod p = (a \bmod p)^x \bmod p$
2. Euler's totient theorem: $a^x \bmod p = a^{x \bmod \phi(p)} \bmod p$

When the tumbler raises $\omega^{K_A}$ to the power $d$, he receives $(\omega^{K_A} \bmod p)^d \bmod p = \omega^{dK_A} \bmod p$ due to the first identity. Additionally, $(a^{dx \bmod \phi(p)} \bmod p)^{K_A} \bmod p = a^{dxK_A} \bmod p$, due to the second identity.

### 5.2.2 Justification

Note that the values of $x$ and $d$ secret keys do not change throughout the entire system, i.e. they will keep their values for all the $Q$ bitcoins. Therefore, it is essential for the algorithm not to reveal any information about these tow parameters.

Anyone other than the Tumbler will not find out the value of $x$ or $d$ from the public key $\tau$ since they are both kept secret. Furthermore, anyone other than the Tumbler cannot find out the value of $d$ from $g^{dxk}$ by just knowing $g^{xk}$ due to the discrete logarithm problem and since the values of $x$ and $k$ are only known by the Tumbler. To be more precise, in the puzzle-solver protocol the value of $k$ is unknown by the Tumbler too. Finally, a cheating verifier cannot find out the value of $d$ from $g^{xdkK_A} \bmod p$ since the values of $x$ and $k$ are unknown to him.

A cheating Tumbler would want to send a corrupt $\epsilon^*$ to the verifier, i.e. $\epsilon^* \neq \epsilon$. This could not work due to the following reasons:

- The verifier computes $\frac{b}{\epsilon^*} \bmod p = \theta \times g^{xk} \bmod p$, where $\theta = \frac{\epsilon}{\epsilon^*} \bmod p$
- Assume the Tumbler knows the value of $\theta$ since he knows both original and corrupt messages $\epsilon$ and $\epsilon^*$ respectively.
- The verifier sends the blinded $\frac{b}{\epsilon^*}$ to the Tumbler: $(\frac{b}{\epsilon^*})^{K_A} = \theta^{K_A} \times g^{xkK_A} \bmod p$
- A cheating Tumbler would want to extract $\theta$ from the verifier's returned number so that at the end of verification he would receive 1 and not find out that the solution to the puzzle is corrupt.
- However, due to the verifier's blinding, the Tumbler will not be able to do this. He cannot extract $\theta$ from $\theta^{K_A} \times g^{xkK_A} \bmod p$ due to the discrete logarithm problem, even though he knows the value of $\theta$.
- The Tumbler raises this to power of $d$: $\theta^{dK_A} \times g^{xdkK_A} \bmod p$.
- A cheating Tumbler would want to pick values $\epsilon^*$, $\theta$ and $d$ in a way so that $\theta^{dK_A} \equiv 1 \bmod p$.
- He could pick $\epsilon^*$ so that $\frac{\epsilon}{\epsilon^*} = 1 \bmod p$, however that is only possible if $\epsilon^* = \epsilon$.
- He could also pick $\epsilon^*$ in a way so that $\theta \bmod p$ is a random element of an arbitrary subgroup in the cyclic group $G$ picked by the cheating Tumbler. He could raise

$\theta^{dK_A} \times g^{xdkK_A} \bmod p$ to the power $ord$, where $ord$ is the order of this subgroup and $\theta$ is a random element of this subgroup. He would receive

$$\theta^{ord.dK_A} \times g^{ord.xdkK_A} \bmod p = 1 \times g^{ord.xdkK_A} \bmod p$$

However, he cannot get back from this $g^{xdkK_A}$ since

$$g^{xdkK_A} \bmod p \neq (g^{ord.xdkK_A} \bmod p^{ord^{-1} \bmod p)} \bmod p$$

- He also cannot use Euler's totient theorem. This is because

  $g^{ord.xdkK_A \times (ord^{-1} \bmod \phi(p))} \bmod p \neq g^{(ord.xdkK_A.ord^{-1}) \bmod \phi(p))} \bmod p = g^{xdkK_A} \bmod p$

- So, when the corrupt $(\theta^{K_A} \times g^{xkK_A})^d \bmod p$ is returned to the verifier, he will compute

  $(\theta^{dK_A} \times g^{xkdK_A}) \times a^{-\tau K_A} \bmod p = \theta^{dK_A xkdK_A} \times g^{-kdxK_A}) \bmod p = \theta^{dK_A} \neq 1 \bmod p$

## 5.3 TumbleBit protocols using Elgamal

We implemented the two TumbleBit protocols using Elgamal and provided a detailed walk-through for the new puzzle-promise and puzzle-solver algorithms. These are shown in figures 5.2 and 5.3 respectively.

## 5.4 TumbleBit using Elgamal with Elliptic Curves

Based on the background information given in section 3.4 on Elgamal Elliptic Curve Cryptography, we implement TumbleBit using this third cryptosystem. Its protocols will be almost identical to the Elgamal version described in section 5.3. The main difference will be that the Elgamal exponentiation and multiplication will be replaced by point multiplication and point addition on the EC. For this reason, we only provide a brief description of TumbleBit's four main cryptographic operations: encryption, decryption, blinding and verification.

For the new puzzle-promise protocol, the Tumbler's public input will include $(g_E, y_E, p)$ and the secret inputs will consist of $(x_E, k_E)$, where $p$ is the prime modulus, $g_E$ is a point on EC and $y_E$ was calculated in (3.11). $x_E$ and $k_E$ are randomly chosen values from $\{1, ..., O_E - 1\}$.

For the new puzzle-solver protocol, the Tumbler's public input will be $(g_E, y_E, p, d_E x_E)$ and his secret inputs $(x_E, d_E)$. $d_E$ is the Tumbler's secret parameter that he will use for verification. It is randomly chosen from $\{1, ..., O_E - 1\}$. Alice will have $(\overline{A}_B, \overline{B}_B, \kappa_A)$ as her secret inputs, where $\kappa_A$ is her secret exponent used for verification, also chosen from $\{1, ..., O_E - 1\}$.

### 5.4.1 Encryption

In the puzzle-promise protocol, before the Tumbler encrypts the $\epsilon$ parameters, he uses a mapping function to convert them to points on the EC, denoted as $E$ and earlier described in section 3.4.2. He then encrypts these to obtain the $(\mu + \eta)$ Elgamal ECC puzzles $(A, B)$ . This is shown in (3.12).

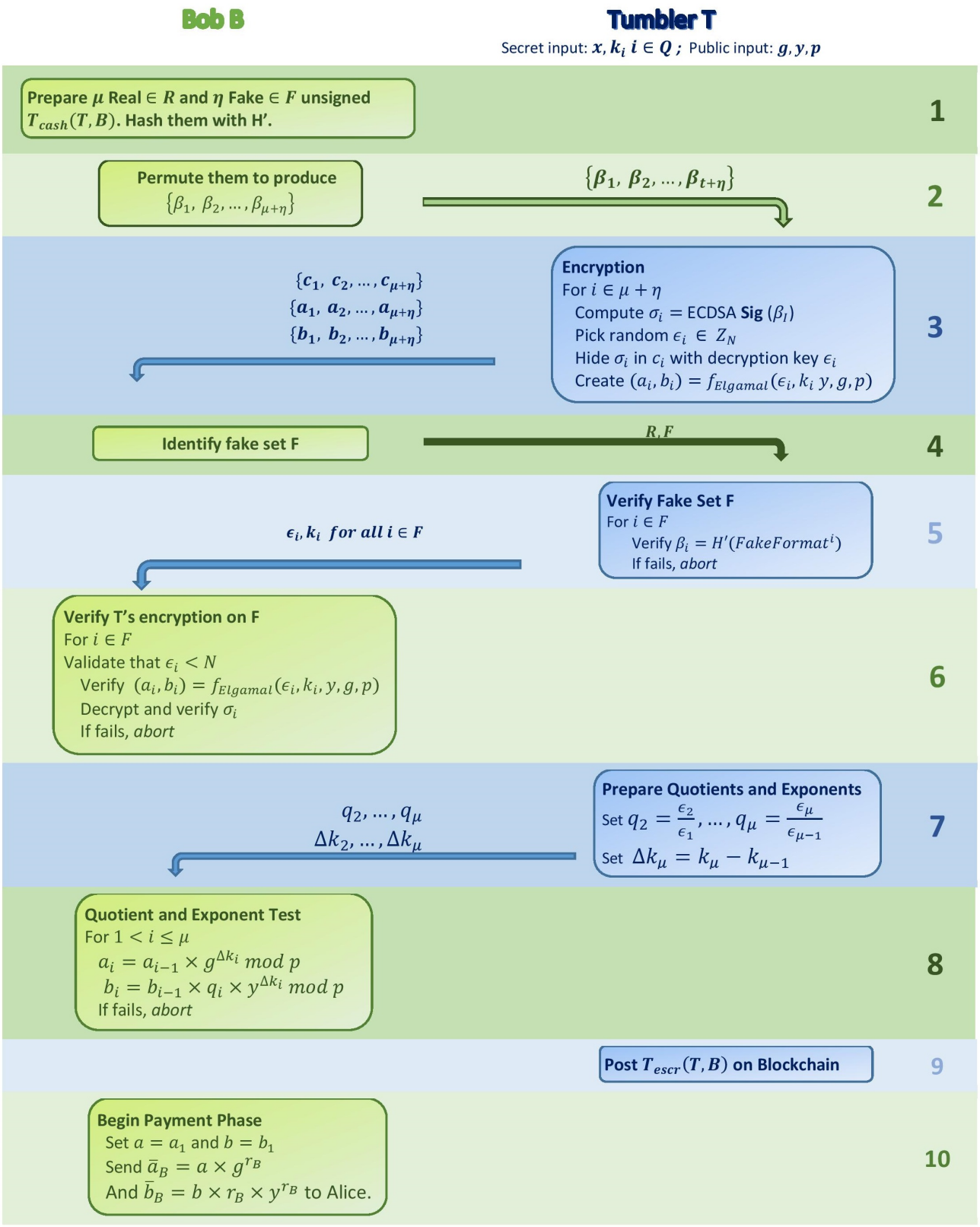



Figure 5.2: Proposal for Elgamal-based puzzle-promise protocol

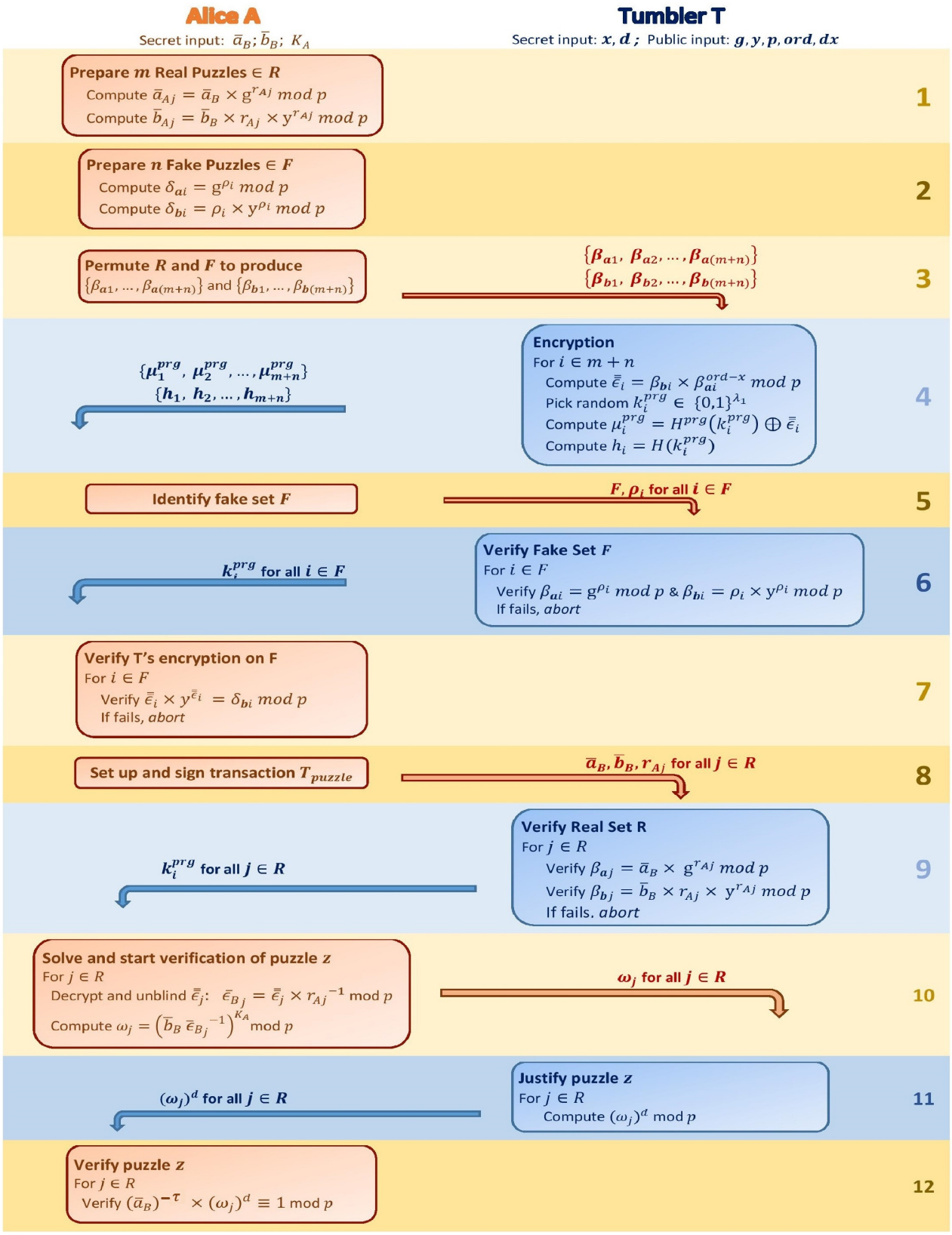



Figure 5.3: Proposal for Elgamal-based puzzle-solver protocol

### 5.4.2 Decryption

Decryption of ECC Elgamal ciphertexts is analogous to the Elgamal decryption and it is presented in (3.13).

### 5.4.3 Blinding

In the puzzle-promise protocol, Bob blinds the puzzles $(A, B)$ using $R_B$ to obtain $\overline{A}_B, \overline{B}_B$. This is shown in (5.3). $R_B$ is his blinding factor randomly chosen from $\{1, ..., O_E - 1\}$. When the Tumbler decryptes this, Bob will receive $E \oplus R_B$ as calculated in (5.4).

$$\begin{aligned} \overline{A}_B &= A + (R_B \times g_E) \\ \overline{B}_B &= B + R_B + (R_B \times y_E) \end{aligned} \tag{5.3}$$

$$\begin{aligned} &\overline{B}_B - (x_E \times \overline{A}_B) = \\ &B + R_B + (R_B \times y_E) - x_E \times [A + (R_B \times g_E)] = \\ &E + (k_E \times x_E \times g_E) + R_B + (R_B \times x_E \times g_E) - x_E \times [(k_E \times g_E) + (R_B \times g_E)] = \\ &E + R_B + (k_E \times x_E \times g_E) + (R_B \times x_E \times g_E) - (x_E \circ k_E \times g_E) - (x_E \times R_B \times g_E) = \\ &E + R_B \end{aligned} \tag{5.4}$$

### 5.4.4 Verification

Verification of ECC Elgamal decryption is analogous to our proposal in section 5.2. Alice will compute $\omega_E \times \kappa_A$ using $E^*$, the received puzzle solution from the Tumbler. Then she sends this to the Tumbler who will multiply this $d_E$. Alice will verify this; $E^*$ is valid if it equals $E$ and the result of the verification is zero. The process is shown below.

1. $\omega_E = B - E^*$
2. $\omega_E \times \kappa_A = \kappa_A \times B - \kappa_A \times E^*$
3. $\omega_E \times \kappa_A \times d_E = d_E \times \kappa_A \times B - d_E \times \kappa_A \times E^*$

4. $-(d_E \times x_E \times \kappa_A) \times A + d_E \times \kappa_A \times B - d_E \times \kappa_A \times E^* =$
   $-[d_E \times x_E \times \kappa_A \times k_E \times g_E] + [d_E \times \kappa_A \times (E + (k_E \times y_E))] - [d_E \times \kappa_A \times E^*] =$
   $-[d_E \times x_E \times \kappa_A \times k_E \times g_E] + [d_E \times \kappa_A \times E] + [d_E \times \kappa_A \times k_E \times x_E \times g_E] - [d_E \times \kappa_A \times E^*] = 0$

# Chapter 6

# Evaluation

## 6.1 Modular exponentiation

Modular exponentiation is at the heart of both RSA and Elgamal encryption schemes. In this section we review the algorithmic and mathematical aspects of the software implementation of modular exponentiation. This information will be useful when comparing the computational complexities of the original and proposed TumbleBit systems in section 6.3.

We use the modular exponentiation $z = \epsilon^{pk} \bmod N$ in this section as an example to demonstrate certain rules. Firstly, note "$z$ cannot be computed by first exponentiating $\epsilon^{pk}$ and then dividing to obtain the remainder, i.e. $z = \epsilon^{pk}$ % $N$" [14]. The result of each step of the exponentiation process has to be reduced by modulo $N$. This is because for large exponents the result of $\epsilon^{pk}$ would have very large space requirement. In TumbleBit, the 2048-bit RSA encryption is used where the size of the modulus $N$ is 2048 bits [15] and $\epsilon \in Z_N$, so the size of $\epsilon$ is at most 2048 bits too. The public exponent $pk$ is equal to 65537[1], which is 17 bits long. Therefore, $\epsilon^{pk}$ would need

$$log_2(\epsilon^{pk}) = pk \times log_2(\epsilon) \approx 2^{17} \times 2048 = 2^{28} \tag{6.1}$$

bits to be stored.

The naive way to compute $z = \epsilon^{pk} \bmod N$ is to perform $pk-1$ modular multiplications, i.e. $\epsilon \bmod N \rightarrow \epsilon^2 \bmod N \rightarrow \epsilon^3 \bmod N \rightarrow ... \rightarrow \epsilon^{pk} \bmod N$. However, there are several more sophisticated ways to compute $z$ without having to calculate all powers of $\epsilon$.

[1]This is the largest known prime number of the form $2^{2^n} + 1$ for $n = 4$. It is used as the public exponent in the proof of concept Github project of TumbleBit. [16]

In this paper we will analyze the *multiply-and-add* algorithm and determine its computational complexity. This method is also known as the binary exponentiation scheme where each key bit is processed sequentially with a modular square operation and then a conditional modular multiplication [17]. Figure 6.1 presents the pseudo code of the multiply-and-add algorithm.

```
z:=1                                    //set z to initial value
for i from |pk|-1 down to 0 do:
    z:=z*z mod N                        //square
    if (pk_i= 1), then z:=z*ϵ mod N//multiply
return z
```

Figure 6.1: multiply-and-add algorithm from left to right evaluating $z = \epsilon^{pk} \bmod N$ [17]

It is clear that the number of modular squarings that the algorithm uses is

$$(\text{no. of bits of pk}) - 1 = (\lfloor log_2(pk) \rfloor + 1) - 1 = \lfloor log_2(pk) \rfloor \tag{6.2}$$

[2] and the number of modular multiplications will equal the hamming weight[3] of $pk$, $wt(pk)$ which will have a range of $0 < wt(pk) < \lfloor log_2(pk) \rfloor$.

Therefore, the total number of modular and modular squarings in a modular exponentiation will be

- a maximum value of $\lfloor log_2(e) \rfloor + \lfloor log_2(e) \rfloor$,
- a minimum value of $\lfloor log_2(e) \rfloor + 0$
- hence, an average value of $\frac{3}{2} \lfloor log_2(e) \rfloor$

- Total number of modular squarings is $\lfloor log_2(pk) \rfloor$.
- Total number of modular multiplications is between 0 and $\lfloor log_2(pk) \rfloor$
- an average number of modular multiplications is$\frac{1}{2} \lfloor log_2(pk) \rfloor$

## 6.2 Point multiplication

In this paper, we analyze the software implementation of point multiplication based on a method called double-and-add. It is analogous to the multiply-and-add algorithm used to implement modular exponentiation ins section 6.1.

[2] $\lfloor . \rfloor$ represents the floor function
[3] Hamming weight of x: $wt(x) = |\{i : x_i \neq 0\}|$ [7]

We take $A = k_E \times g_E$ as an example for point multiplication where the point $g_E$ on the EC is multiplied by the integer $k_E$. This is implemented by adding the point to itself repeatedly. However, instead of performing $k_E - 1$ additions, we show why the double-and-add algorithm is more efficient in figure 6.2.

```
A:=0                                    //set A to initial value
for i from |kE|-1 down to 0 do:
    A:= A + A                           //point doubling
    if (kEi = 1), then A:= A + gE       //point addition
return A
```

Figure 6.2: Double-and-add algorithm to compute $A = k_E \times g_E$ from left to right. [18]

Similarly to the multiply-and-square algorithm, we can approximate the average number of point additions and point doublings required to implement a point multiplication. We get the following results:

- Total number of point doublings is $\lfloor log_2(k_E) \rfloor$.
- Total number of point additions is between 0 and $\lfloor log_2(k_E) \rfloor$
- hence, an average number of point additions is $\frac{1}{2} \lfloor log_2(K_E) \rfloor$

Appendix D.4 shows the calculations required to perform both point addition and point doubling. This allows us to find the number of modular arithmetic operations required which are shown in table 6.1.

| | **Point addition** | **Point doubling** |
|---|---|---|
| **Modular multiplication** | 2 | 2 |
| **Modular squaring** | 1 | 2 |
| **Modular inverse** | 1 | 1 |

Table 6.1: Number of modular operations needed for point addition and point doubling.

Figure 6.3 shows experimental results of [18] for the speed of the above-mentioned three modular arithmetic operations. We can observe that modular inverse is much more expensive than multiplication and squaring. We can conclude that on average, modular multiplication is approximately 2 times and 27 times faster than modular

squaring and modular inverse respectively. This information will be used in section 6.5 when comparing the performance of the three TumbleBit implementations.

| | multiplication | mod p | square | mod p | mod inverse |
|---|---|---|---|---|---|
| MPIR | 0.07 us | 0.15 us | 0.13 us | 0.15 us | 1.8 us |
| openssl | 0.08 us | 0.43 us | 0.06 us | 0.43 us | 18.0 us |

Figure 6.3: Experimental results for speed of modular multiplication, squaring and inverse using two libraries, openssl and MPIR.[18]

## 6.3 TumbleBit with Elgamal Performance

In this section we are going to analyze and compare the computational complexities of encryption, decryption blinding and verification for the RSA and Elgamal-based Tumblebit systems. Figure 6.4 summarizes the number of modular exponentiation and multiplications required for these cryptogrpahic operations.

### 6.3.1 Determining number of modular exponentiations and multiplications

| | RSA | | | Elgamal | | | |
|---|---|---|---|---|---|---|---|
| | Expression | Modular exponentiation | Modular multiplication | Expression | Modular exponentiation | Modular multiplication | Modular inverse |
| Encryption | $z = \epsilon^{pk} \bmod N$ | 1 | | $a = g^k \bmod p$ | 1 *CB* | | |
| | | | | $b = \epsilon \times y^k \bmod p$ | 1 *CB* | 1 | |
| Decryption | $\epsilon = z^{sk} \bmod N$ | 1 | | $b \times a^{ord-x} \bmod p$ | 1 | 1 | |
| Blinding | $\bar{z}_B = z \times {r_B}^{pk} \bmod N$ | 1 *CB* | 1 | $\bar{a}_B = a \times g^{r_B} \bmod p$ | 1 *CB* | 1 | |
| | | | | $\bar{b}_B = b \times r_B \times y^{r_B} \bmod p$ | 1 *CB* | 1 *CB* + 1 | |
| Verification | $\epsilon^{pk} = z \bmod N$ | 1 | | $\omega = b \times \epsilon^{-1} \bmod p$ | | 1 | 1 |
| | | | | ${\omega^{K_A}} \bmod p$ | 1 | | |
| | | | | $(\omega^{K_A})^d \bmod p$ | 1 | | |
| | | | | $a^{-\tau K_A} \times (\omega^{K_A})^d \bmod p$ | 1 *CB* | 1 *CB* + 1 | 1 *CB* |

Figure 6.4: Modular exponentiation and multiplications for RSA and Elgamal operations. *CB* stands for "computable beforehand".

The expressions marked with "CB" can be computed in the one-time-only setup phase which is defined in appendix A. This reduces the computational complexity and time length of the two protocols.

This way, the Elgamal encryption of the $(\mu + \eta)$ transactions in the puzzle-promise protocol (step 3 of figure 5.2) can be reduced to $(\mu + \eta)$ modular multiplications in the escrow phase and $2(\mu + \eta)$ modular exponentiations in the setup phase.

Decryption does not have any mathematical expressions marked with *CB* since it is directly dependent on the puzzles the Tumbler receives from Alice in the puzzle-solver protocol. Therefore it cannot be pre-computed in the setup phase.

The $m$ Elgamal blinding operations performed by Alice in the puzzle-solver protocol (step 1 of figure 5.3) can be reduced to $2m$ modular multiplications during the payment phase and $m$ multiplications $+2m$ modular exponentiations in the setup phase.

Finally, the verification of the $m$ Elgamal puzzle solutions can only be simplified in its final stage (step 12 of figure 5.3) by pre-computing $a^{-dxK_A} \bmod p$ in the setup phase and leaving 2 multiplications and exponentiations to be performed in the payment phase.

### 6.3.2 Determining the allowable security strength

Now we want to determine the minimum allowed key-size for Elgamal. For this we use two tables in [19] which is a document published by the National Institute of Standards and Technology (NIST), a US measurement standards laboratory in 2016. This is shown in figures 6.5 and 6.6.

<table>
<tr><th colspan="2">Security Strength</th><th>2011 through 2013</th><th>2014 through 2030</th><th>2031 and Beyond</th></tr>
<tr><td rowspan="2">80</td><td>Applying</td><td>Deprecated</td><td colspan="2">Disallowed</td></tr>
<tr><td>Processing</td><td colspan="3">Legacy use</td></tr>
<tr><td rowspan="2">112</td><td>Applying</td><td rowspan="2">Acceptable</td><td rowspan="2">Acceptable</td><td>Disallowed</td></tr>
<tr><td>Processing</td><td>Legacy use</td></tr>
<tr><td>128</td><td rowspan="3">Applying/Processing</td><td>Acceptable</td><td>Acceptable</td><td>Acceptable</td></tr>
<tr><td>192</td><td>Acceptable</td><td>Acceptable</td><td>Acceptable</td></tr>
<tr><td>256</td><td>Acceptable</td><td>Acceptable</td><td>Acceptable</td></tr>
</table>

Figure 6.5: Comparable security strengths for algorithms approved by NIST.[19]

| Bits of security | Symmetric key algorithms | FFC (e.g., DSA, D-H) | IFC (e.g., RSA) | ECC (e.g., ECDSA) |
|---|---|---|---|---|
| 80 | 2TDEA[18] | $L = 1024$<br>$N = 160$ | $k = 1024$ | $f = 160\text{-}223$ |
| 112 | 3TDEA | $L = 2048$<br>$N = 224$ | $k = 2048$ | $f = 224\text{-}255$ |
| 128 | AES-128 | $L = 3072$<br>$N = 256$ | $k = 3072$ | $f = 256\text{-}383$ |
| 192 | AES-192 | $L = 7680$<br>$N = 384$ | $k = 7680$ | $f = 384\text{-}511$ |
| 256 | AES-256 | $L = 15360$<br>$N = 512$ | $k = 15360$ | $f = 512+$ |

Figure 6.6: Time frames of security strengths, where $L$ and $N$ represent the size of public and private keys in finite-field cryptography (FFC) respectively, $k$ is the key size in integer-factorization cryptography (IFC) and $f$ is the key size in ECC.[19].

The table in 6.5 shows NIST's approved security strengths of cryptographic algorithms for the years between 2011 and 2030. The column of 'Security strength' refers to the bits of security of the algorithms listed next to it and according to [20], this means "an estimation of the amount of work required to defeat a cryptographic algorithm, and therefore the higher the value, the better". The terms 'Applying' and 'Processing' refer to encryption and decryption respectively.

We can see that starting form 2014, a security strength of 80 bits has been disallowed for encryption purposes and only allowed for legacy use when it comes to decryption. The term 'Legacy use' means that "a digital signature that provides 80 bits of security could be processed (i.e., verified) after 2013 as indicated by the "legacy use" indication", according to [19]. However, this is of no concern to TumbleBit since all its cryptographic operations will have a security strength of 112 (after year of 2014).

From figure 6.6, we can see that an RSA scheme with security strength of 112 bits will require a key size of 2048 bits. This is consistent with the original 2048-bit RSA-based TumbleBit in [3]. An Elgamal algorithm with the same security level will need a public key size of 2048 bits and a private key size of 224 bits. In the context of the Elgamal-

based TumbleBit, this means that the modulus $p$ has to be 2048-bit and the private keys $x$ and $d$ 224-bit long.

### 6.3.3 Determining number of modular arithmetic operations

For the 2048-bit RSA scheme used in TumbleBit, the public exponent $pk = (65537)_{10} = (10000000000000001)_2$ has a hamming weight $wt(pk)$ of 2. Therefore, according to figure 6.1, $\epsilon^p k$ mod $N$ will have $\lfloor log_2(pk) \rfloor = 16$ modular squaring operations and 2 modular multiplications.

From the proof of concept code of TumbleBit [16], we use the generated private keys found in the '*keys*' folder to estimate the bit length of the RSA secret key. These are created using the OpenSSL library. There are two files containing two different secret keys and they are called '*private_2048_test.pem*' and '*private_2048_test.pem*'. They include the base64 encoding of the original secret keys. For our analysis, we require only the size of the original keys and this can be computed using the size of their base64 encoded versions. The process is described in appendix C and the result is 9536 bits. Hence, we will be calculating with an RSA secret key of size 9536 bits.

As mentioned in section 3.3.1, Elgamal's public keys include the modulus $p$, generator $g$ and $y = g^x$ mod $p$. These will each have a bit length of 2048 bits. The private keys $x$ and $k$ will be 224 bits long. Therefore, based on section 6.1, we can calculate the computational complexity of $y = g^x$ mod $p$. This will involve

$$(224 - 1) = 223 \tag{6.3}$$

modular squarings and an average number of

$$\frac{1}{2}(224 - 1) \approx 112 \tag{6.4}$$

modular multiplications. Figure 6.7 summarizes the computational complexity of the four operations of figure 6.4 for both RSA and Elgamal encryptions.

| | Expression | Total no. of | | Expression | (no CB) Total no. of | | | (with CB) Total no. of | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | mod squ. | mod mul. | | mod squ. | mod mul. | mod inv. | mod squ. | mod mul. | mod inv. |
| Encryption | $z = \epsilon^{pk} \bmod N$ | 16 | 2 | $a = g^k \bmod p$<br>$b = \epsilon \times y^k \bmod p$ | 446 | 225 | | | 1 | |
| Decryption | $\epsilon = z^{sk} \bmod N$ | 9535 | 4768 | $b \times a^{ord-x} \bmod p$ | 223 | 113 | | 223 | 113 | |
| Blinding | $\bar{z}_B = z \times {r_B}^{pk} \bmod N$ | 16 | 3 | $\bar{a}_B = a \times g^{r_B} \bmod p$<br>$\bar{b}_B = b \times r_B \times y^{r_B} \bmod p$ | 446 | 227 | | | 2 | |
| Verification | $\epsilon^{pk} = z \bmod N$ | 16 | 2 | $\omega = b \times \epsilon^{-1} \bmod p$<br>$\omega^{K_A} \bmod p$<br>$(\omega^{K_A})^d \bmod p$<br>$a^{-\tau K_A} \times (\omega^{K_A})^d \bmod p$ | 892 | 451 | 2 | 669 | 338 | 1 |

Figure 6.7: Average number of modular arithmetic operations for RSA and Elgamal operations. 'No CB' and 'With CB' denote no and with calculations beforehand respectively.

We can observe that all RSA operations are faster than their equivalent operations in Elgamal apart from decryption which consumes less computational time in Elgamal than for RSA.

## 6.4 TumbleBit with Elgamal Encryption using ECC

In section 5.4, TumbleBit was implemented using Elgamal Elliptic Curve Cryptosystem. The number of modular arithmetic operations required to perform its encryption, decryption, blinding and verification are presented in figure 6.8. Similarly to Elgamal, some of the parameters can be pre-calculated in the setup phase in order to reduce the overall runtime of the four operations.

Figure 6.6 shows that using ECC we can achieve a security level of 112 bits when the key size is only 255 bits long. This means that the modulus $p$ will be 255 bits long. According to appendix D.3, the order of the elliptic curve $O_E$ also has to be 255 bits long. Since all $k_E, x_E, R_B, \kappa_A, d_E, d_E x_E$ parameters are chosen from $\{1, ..., O_E\}$, their bit length will be at most 255.

As a consequence, all the point multiplications in figure 6.8 will need 254 point additions and 127 point doublings each on average. This translates to 762 modular multiplications, 508 modular squarings and 381 modular inverses for one point multiplication on average.

The modular operations marked with CB in figure 6.8 can be computed in the setup phase. This reduces the speed significantly.

| Elgamal using ECC | | | | | | |
|---|---|---|---|---|---|---|
| | Expression | EC point multiplication | EC point addition | Total no. of modular multiplications | Total no. of modular squarings | Total no. of modular inverse |
| Encryption | $A = k_E \times g_E$ | 1 *CB* | | 1524 *CB* + 2 | 1016 *CB* + 1 | 762 *CB* + 1 |
| | $B = E + k_E \times y_E$ | 1 *CB* | 1 | | | |
| Decryption | $E = B - x_E \times A$ | 1 | 1 | 764 | 509 | 382 |
| Blinding | $\bar{A}_B = A + R_B \times g_E$ | 1 *CB* | 1 | 1526 *CB* + 4 | 1017 *CB* + 2 | 763 *CB* + 2 |
| | $\bar{B}_B = B + R_B + R_B \times y_E$ | 1 *CB* | 1 *CB* + 1 | | | |
| Verification | $\omega_E = B - E$ | | 1 | 1524 *CB* + 1528 | 1016 *CB* + 1018 | 762 *CB* + 764 |
| | $\kappa_A \times \omega_E$ | 1 | | | | |
| | $d_E \times (\kappa_A \omega_E)$ | 1 | | | | |
| | $-(d_E x_E \times \kappa_A) \times A + d_E \kappa_A \omega_E$ | 2 *CB* | 1 | | | |

Figure 6.8: Total number of EC point multiplications for Elgamal oeprations using ECC.

## 6.5 Overall performance evaluation

When comparing the three implementations of TumbleBit (RSA, Elgamal and Elgamal ECC), we observe that all the RSA cryptographic operations take significantly more time to calculate than the other two, apart from verification. For that, Elgamal performs the worst, followed by Elgamal ECC and then RSA with a significantly lower runtime. For the other three cryptographic operations, Elgamal ECC highly outperforms both Elgamal and RSA.

This observation is firstly based on the analysis presented in appendix E, where we estimated the speed of the 256-bit modular multiplication[4] to be 50 times faster than the 2048-bit one. Secondly, in section 6.2 we have calculated the speed of modular multiplication to be 2 times and 27 times faster than modular squaring and modular inverse respectively. Thirdly, the number of modular multiplications required for the four main cryptographic operations were computed for RSA, Elgamal and Elgamal ECC and presented in figures 6.7 and 6.8.

Using this information, we were able to compare the three TumbleBit algorithms and present the results in a table in figure 6.9. This shows the number of units required to do the corresponding computations, where one unit is the time required to complete one 256-bit modular multiplication. Note, the Elgamal and Elgamal ECC algorithm are assumed to be using the setup phase to pre-compute the CB marked equations in figures 6.7 and 6.8.

[4]In this paper, we assume that the speed of the 256-bit modular multiplication is almost identical to the 255-bit one which is used in the Elgamal ECC implementation of TumbleBit. This can also be seen in figure E.1 where the two values for speed are very close to each other.

| | RSA | Elgamal (with CB) | Elgamal ECC |
|---|---|---|---|
| Encryption | 1700 | 50 | 31 |
| Decryption | 1191900 | 27950 | 12096 |
| Blinding | 1750 | 100 | 62 |
| Verification | 1700 | 85150 | 24192 |

Figure 6.9: Speed comparison of RSA, Elgamal and Elgamal ECC cryptographic operations. One unit is equal to the time to compute one 256-bit modular multiplication.

The original objectives of this project have been fulfilled since we have managed to replace the RSA puzzle scheme with Elgamal and Elgamal ECC. This way, TumbleBit protocols achieve polynomial security as described in section 4.1.2 and their encryption, decryption and blinding runtime has been significantly reduced too.

It is true that the proposed Elgamal verification system is more expensive than the RSA version and the reason for this is that the RSA verification algorithm that is used in the original TumbleBit design relies on the deterministic property of RSA. However, one of the motivations of this project was to eliminate this property from the new implementation of TumbleBit in order to ensure polynomial security. As a consequence, although the proposed system has increased the number of modular arithmetic operations and reduced the verification speed, it has also managed to maintain the required secrecy for both parties.

# Chapter 7

# Conclusions

The main focus of the report was to prove that extracting RSA from TumbleBit's design and replacing it with different multiplicatively homomorphic encryption scheme, such as Elgamal would improve the system. From our evaluation we saw that this is indeed possible with a suitable design and pre-calculations.

We have provided general background on the mechanism of TumbleBit; firstly, gave a brief introduction on its algorithm, secondly, described in detail both of its protocols. We then moved on to homomorphic encryption and discussed both RSA and Elgamal encryption systems in detail. We mentioned some advantages of Elgamal over RSA and presented the three main motivations why this could be a better implementation for TumbleBit. These included its homomorphic property, its better security and the fact that it can be integrated with elliptic curves.

We gave a general analysis on the properties that the new system must hold and discussed the main issue that was faced in this project. This was the creation of the Elgamal verification system and it consumed the most amount of time. We believe this novel design for Elgamal encryption verification could be used in other systems, projects and scenarios. We focused on trying to keep the system as general and non-specific to TumbleBit as possible.

Implementation covered the Elgamal realization of the four main cryptographic operations that TumbleBit relies on and these are: encryption, decryption, blinding and verification. The reason for emphasizing these was because the rest of the algorithm of the new implementation is identical to the ones described in the background chapter. We only changed the RSA-related stages of the protocols. The Elgamal ECC implementation was also described here.

Evaluation was the second most difficult part of the project, following the verification

system. It was challenging to find a way to compare all the three cryptosystems. It is important to emphasize that these comparisons are based on many approximations and assumptions; the aim was to justify the efficiency of the new TumbleBit algorithms.

We conclude that the project was successful; the new systems have increased the security of TumbleBit, the length of the key sizes decreased and therefore the runtime of three of the cryptographic operations has significantly shortened.

Although there are many features of TumbleBit that requires further research in the future, but we can say that this project serves as a step towards introducing private transactions within Bitcoin.

## 7.1 Future Work

Future reasearch is required to focus on reducing the runtime of the Elgamal verification system. The computational complexity is quite high in its current form. This is due to the large number of modular exponentiations required to be computed during the protocol runtime. The aim is to increase its overall speed, perhaps by enabling more calculations to be performed in the setup phase while maintaining the same level of security and satisfying the criteria discussed in this project.

Other future work related to this specific project could be targeted to implement the Elgamal TumbleBit protocols that allow payers to send more than one bitcoin. In the original white paper of TumbleBit, [3] this implementation was included in the appendix. Parallel transactions were used to send several bitcoins from a payer to a payee. Due to time constraints this research was neglected int this project.

Another issue with TumbleBit that is more specific to the original design was summarized in [21]: “There is one big drawback to TumbleBit in its current state. The system only works with a fixed denomination. Anyone who wanted to pay an amount larger than that denomination will need to make multiple payments, which is not exactly convenient.”. This will require further research int the future.

# Bibliography


[1] P. Rizzo, "What if one bitcoin isn't like the others? fungibility debated at scaling conference", 2016. [Online]. Available: http://www.coindesk.com/fungability-bitcoin-scaling-milan-one-bitcoin-not-like-others/.

[2] L. Coleman, "How tumblebit builds on coinswap to improve bitcoin privacy and fungibility", 2016. [Online]. Available: https://www.cryptocoinsnews.com/how-tumblebit-builds-on-coinswap-to-improve-bitcoin-privacy-and-fungibility/.

[3] E. Heilman, L. Alshenibr, F. Baldimtsi, A. Scafuro, and S. Goldberg, "Tumblebit: An untrusted bitcoin-compatible anonymous payment hub", Cryptology ePrint Archive, Report 2016/575, Tech. Rep., 2016. [Online]. Available: https://eprint.iacr.org/2016/575.pdf.

[4] A. G. Malvik and B. Witzoee, "Elliptic curve digital signature algorithm and its applications in bitcoin", 2015. [Online]. Available: http://cs.ucsb.edu/~koc/ecc/project/2015Projects/Malvik+Witzoee.pdf.

[5] C. Crépeau, "Cut-and-choose protocol", in *Encyclopedia of Cryptography and Security*, H. C. A. Van Tilborg and S. Jajodia, Eds. Boston, MA: Springer US, 2011, pp. 290–291, ISBN: 978-1-4419-5906-5. [Online]. Available: http://dx.doi.org/10.1007/978-1-4419-5906-5_240.

[6] X. Yi, R. Paulet, and E. Bertino, *Homomorphic encryption and applications*. Springer, 2014, pp. 16–19.

[7] W. Dai, *Section 3: Error correcting codes (ecc): Fundamentals*, 2016.

[8] S. M. KEVIN, "The discrete logarithm problem", *Cryptology and computational number theory*, vol. 42, 1990. [Online]. Available: http://www.mccurley.org/papers/dlog.pdf.

[9] X. Yi, R. Paulet, and E. Bertino, *Homomorphic encryption and applications*. Springer, 2014, pp. 32–36.

[10] R. Sunuwar and S. K. Samal, "Elgamal encryption using elliptic curve cryptography", 2015. [Online]. Available: https://cse.unl.edu/~ssamal/crypto/EEECC.pdf.

[11] N. Koblitz, "Elliptic curve cryptosystems", *Mathematics of computation*, vol. 48, no. 177, pp. 203–209, 1987. [Online]. Available: http://www.ams.org/journals/mcom/1987-48-177/S0025-5718-1987-0866109-5/S0025-5718-1987-0866109-5.pdf.

[12] N. Smart, *Cryptography: An introduction*. McGraw-Hill, 2003, pp. 315–324.

[13] P. E. Black and V. Pieterse, *"polynomial time", in Dictionzary of Algorithms and Data Structures*. National Institute of Standards and Technology, 2004. [Online]. Available: https://www.nist.gov/dads/HTML/polynomialtm.html.

[14] C. K. Koc, "High-speed rsa implementation", RSA Laboratories, RSA Data Security, Inc, Tech. Rep., 1994, pp. 9–11. [Online]. Available: ftp://ftp.rsasecurity.com/pub/pdfs/tr201.pdf.

[15] P. M. C. Saraiva, "Openssl acceleration using graphics processing units", *PhD diss., TECNICO LISBOA*, pp. 9–10, 2013. [Online]. Available: https://fenix.tecnico.ulisboa.pt/downloadFile/395146019110/dissertacao.pdf.

[16] L. AlShenibr and E. Heilman, *Tumblebit/poc_code*, https://github.com/BUSEC/TumbleBit/tree/master/POC_code, 2016.

[17] M. F. Witteman, J. G. J. van Woudenberg, and F. Menarini, "Defeating rsa multiply-always and message blinding countermeasures", in *Topics in Cryptology – CT-RSA 2011: The Cryptographers' Track at the RSA Conference 2011, San Francisco, CA, USA, February 14-18, 2011. Proceedings*, A. Kiayias, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 77–88, ISBN: 978-3-642-19074-2. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-19074-2_6.

[18] N. Courtois, G. Song, and R. Castellucc, "Speed optimizations in bitcoin key recovery attacks", University College London, Tech. Rep. [Online]. Available: https://eprint.iacr.org/2016/103.pdf.

[19] E. Barker, "Recommendation for key management part 1: General", *NIST Special Publication*, vol. 800, pp. 64–67, 2015. [Online]. Available: http://csrc.nist.gov/publications/nistpubs/800-57/sp800-57_part1_rev3_general.pdf.

[20] A. Di Mauro, "A question of key length; does size really matter when it comes to cryptography?", *NIST Special Publication*, 2015. [Online]. Available: https://www.yubico.com/wp-content/uploads/2015/08/Yubico_WhitePaper_A_Question_Of_Key_Length.pdf.

[21] J. Buntinx, "Tumblebit aims to improve bitcoin transaction anonymity", 2017. [Online]. Available: https://themerkle.com/tumblebit-aims-to-improve-bitcoin-transaction-anonymity/.

[22] A. Corbellini, "Elliptic curve cryptography: Ecdh and ecdsa", 2015. [Online]. Available: http://andrea.corbellini.name/2015/05/30/elliptic-curve-cryptography-ecdh-and-ecdsa/.

[23] D. J. Bernstein, "Chacha, a variant of salsa20", in *Workshop Record of SASC*, vol. 8, 2008. [Online]. Available: `https://cr.yp.to/chacha/chacha-20080120.pdf`.

[24] *How to calculate the size of encrypted data?* [Online]. Available: `http://www.obviex.com/articles/CiphertextSize.pdf`.

[25] S. Josefsson, Ed., *The base16, base32, and base64 data encodings*, United States, 2006.

[26] S. Z. Haugnæss, "Generering av elliptiske kurver for kryptografiske anvendelser", Master's thesis, 2015. [Online]. Available: `https://www.duo.uio.no/bitstream/handle/10852/45384/thesis.pdf?sequence=1&isAllowed=y`.

[27] M. Rosing, *Implementing elliptic curve cryptography*. Greenwich, CT, USA: Manning Publications Co., 1999, pp. 104–125, ISBN: 1-884777-69-4.

[28] Y. Wang, J. Leiwo, and T. Srikanthan, "Efficient high radix modular multiplication for high-speed computing in re-configurable hardware [cryptographic applications]", in *2005 IEEE International Symposium on Circuits and Systems*, May 2005, 1226–1229 Vol. 2. [Online]. Available: `file://icnas3.cc.ic.ac.uk/ak6913/01464815.pdf`.

# Appendix A

# Protocols of RSA-based TumbleBit

We present the detailed versions of both puzzle-promise and puzzle-solver protocols for the original RSA-based TumbleBit. Figures A.1 and A.2 show a step-by-step description of the puzzle-promise and puzzle-solver protocols respectively.

## A.1 Puzzle-promise protocol

The Tumbler's secret inputs include

- the RSA secret key $sk$ and
- a freshly chosen ephemeral ECDSA-Secp256k1 secret key $SK_T^{eph}$

The public inputs will include

- the RSA public key $pk$,
- the modulus $N$,
- an ephemeral ECDSA-Secp256k1 public key $PK_T^{eph}$ and
- a proof parameter $\pi$

An ephemeral ECDSA-Secp256k1 signature scheme uses temporary key pairs $(SK_T^{eph}, PK_T^{eph})$, i.e the Tumbler generates the public-private key pair on the fly, before the start of the protocol [22]. The purpose of this signature scheme will be explained later in this section.

There is a one-time-only setup phase performed before the start of the protocol where the above-mentioned parameter $\pi$ is created. As defined in [3], in the setup phase "the Tumbler $T$ announces its RSA public key $(pk, N)$ together with a non-interactive

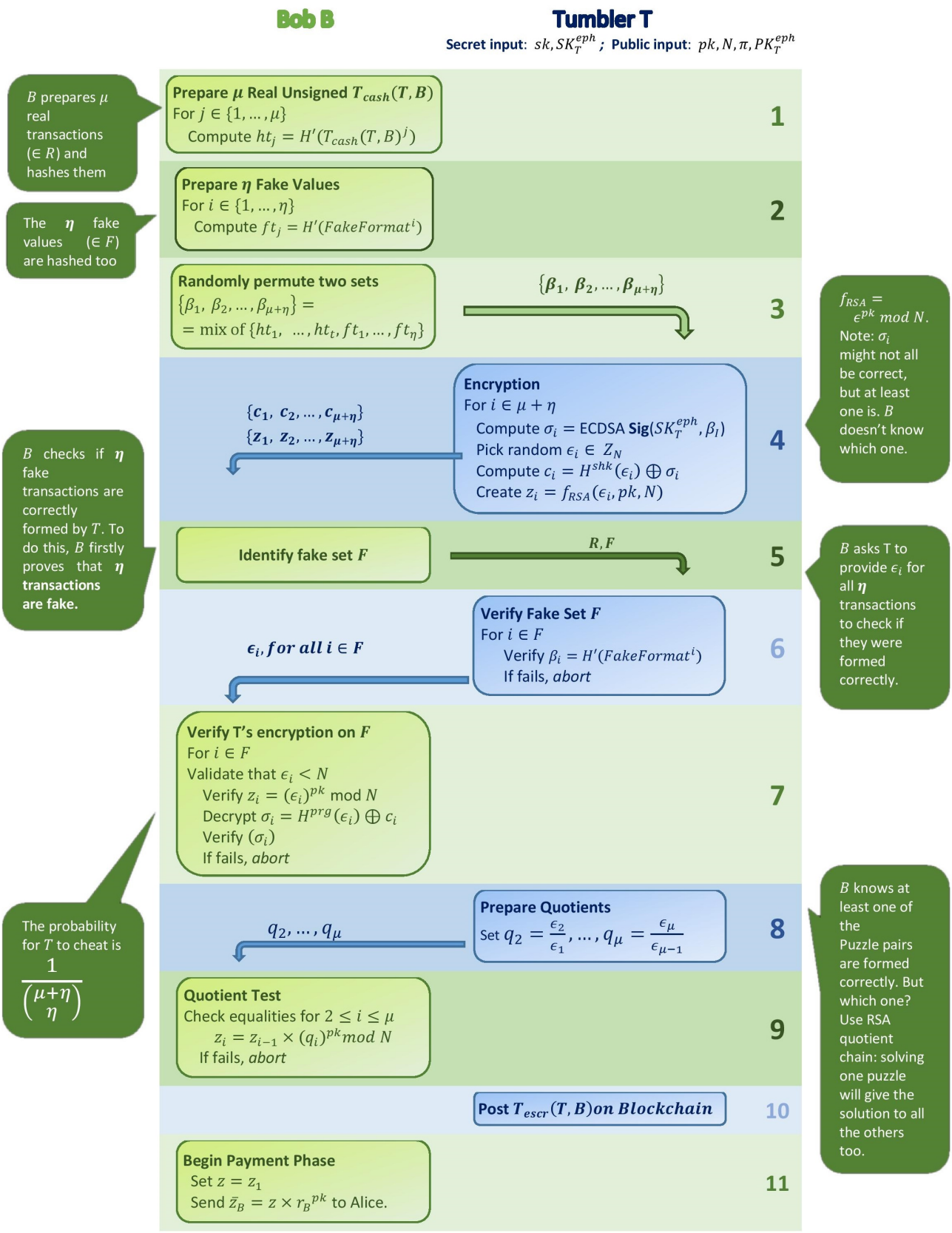



Figure A.1: Detailed puzzle-promise protocol for RSA-based TumbleBit.

zero-knowledge proof of knowledge $\pi$ of the corresponding RSA secret key $sk$. Every user of TumbleBit validates $(pk, N)$ using $\pi$." In other words, $\pi$ proves the validity of the RSA key pairs $(sk, pk)$ and the modulus $N$ in the setup phase. It is also marked as a public input parameter in figure A.2 to the puzzle-solver protocol, serving the same purpose.

In step 1 of figure A.1, Bob prepares $\mu$ unsigned real $T_{cash}(T, B)$ transactions and hashes them the following way:

1. he chooses a random pad $\rho_i \leftarrow \{0,1\}^\lambda$;
2. sets $T_{cash}(T, B)^i = CashOutFormat(\rho_i)$ and
3. calculates $ht_i = H'(T_{cash}(T, B)^i)$ for all $i \in \{1, ..., \mu\}$.

$H'$ is modeled as a random orcale and it is a 'Hash256'; in other words it is a SHA-256 cascaded with itself. SHA-256 is the hash function used in Bitcoin's 'hash-and-sign' paradigm with ECDSA-Secp256k1 [3], the elliptic curve digital signature algorithm (already defined in section 2.1.1).

$CashOutFormat$ is a format used to hide $T_{cash}(T, B)$. As [3] puts it, "it is a shorthand for the unsigned portion of a transaction that fullfils $T_{esc}(T, B)$". $\rho_i$ is used to introduce randomness to $CashOutFormat$ and therefore increase its entropy.

In step 2, Bob prepares $\eta$ fake values in the following way: (1) he randomly picks a pad $r_i \leftarrow \{0,1\}^\lambda$; (2) computes $ft_i = H'(FakeFormat||r_i)$, where $FakeFormat$ is a "distinguishable public string" [3] and $r_i$ is a parameter used to increase the randomness of $ft_i$.

Step 4 of figure A.1 shows that $SK_T^{eph}$ is used to sign $\beta_i$ with the Tumbler's ECDSA signature to receive $\sigma_i$. This is then hidden in $c_i$ using the SHA-512 hash function $H^{shk}$.

The Tumbler verifies the fake set $F$ by firstly recomputing all the above-mentioned $ft_i$ parameters using the $H'$ function and the revealed $r_i$ parameters (step 6). Secondly, in step 7, he validates the fake promise $c_i$ by (1) decrypting $\sigma_i$ using the $H^{prg}$ function and (2) verifying $\sigma_i$ using the public key $PK_T^{eph}$, i.e.

$$ECDSA - Ver(PK_T^{eph}, H(ft_i), \sigma_i) \equiv 1 \tag{A.1}$$

where ECDSA-Ver denotes the ephemeral verification function and $H(ft_i)$ represents the truncated message that was signed. In [3], $H$ is another form of the SHA256 hash function.

$H^{prg}$ is a stream cipher called Chacha20 which is a symmetric key cipher where each digit of the plaint text that is to be converted is encrypted separately [23].

## A.2 Puzzle-solver protocol

As mentioned in section 2.3.1, in step 1 and 2 Alice blinds $\overline{z}_B$ with her $m$ random blinding factors $r_{Ai}$ for $i \in \{1, ..., m\}$ and also creates $n$ fake puzzles using randomly chosen values for $\rho_{Aj}$ for $j \in \{1, ..., n\}$.

In steps 4 and 7, the same $H^{prg}$ and $H$ hashing functions are used which were defined earlier in section A.1. $H^{prg}$ is used to hide the decrypted solution of the puzzle in $\mu^{prg}$ using the randomly chosen key $k^{prg}$. The same hashing function will be used in steps 7 and 11 to decrypt $c$ and receive the puzzle solutions. $H$ is the function applied to hide the key $k^{prg}$ in $h$.

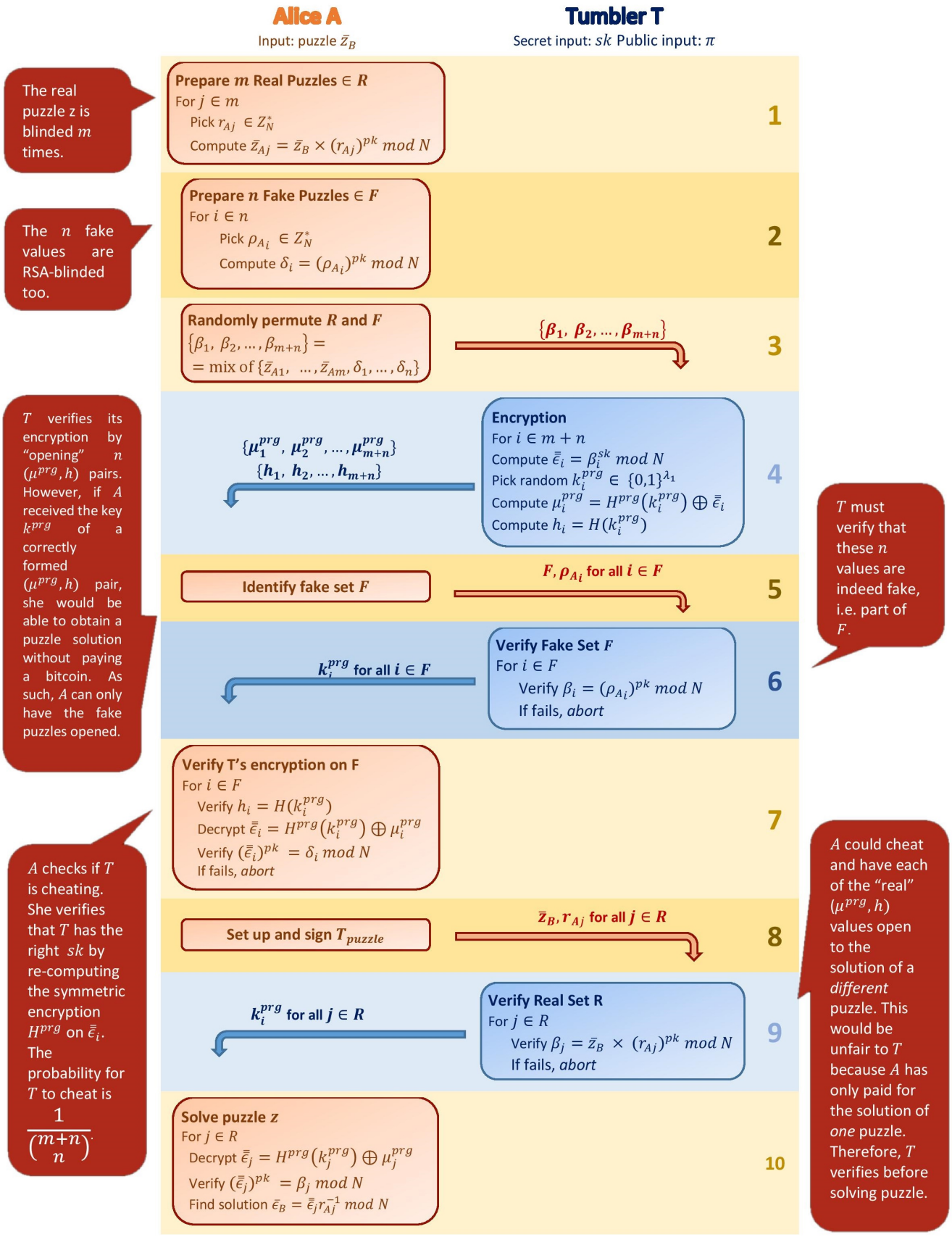



Figure A.2: Detailed puzzle-solver protocol for RSA-based TumbleBit.

# Appendix B

# Protocols of Elgamal-based TumbleBit

We present the detailed versions of both puzzle-promise and puzzle-solver protocols for the proposed Elgamal-based TumbleBit. Figures B.1 and B.2 show a step-by-step description of the puzzle-promise and puzzle-solver protocols respectively.

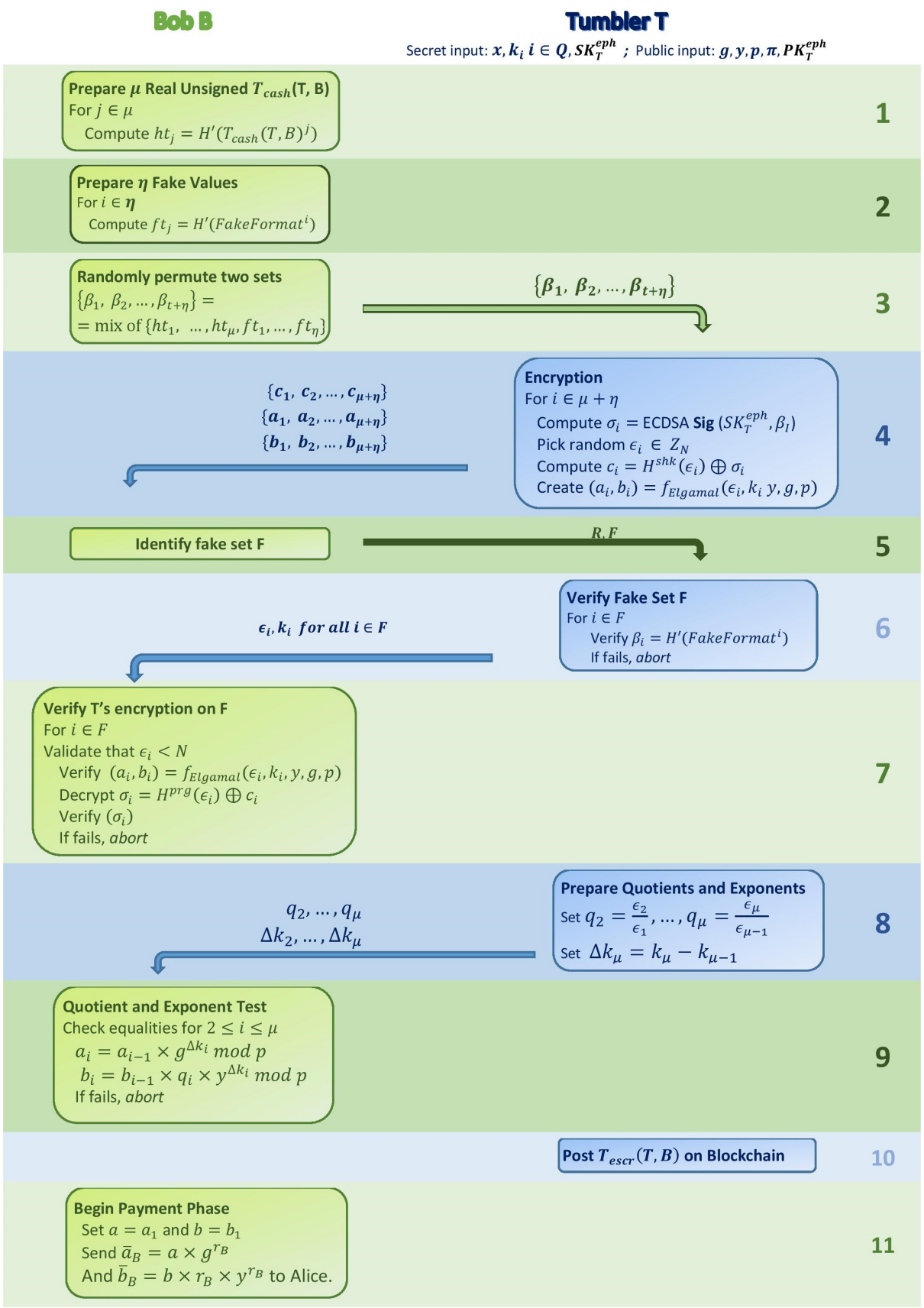



Figure B.1: Detailed puzzle-promise protocol for Elgamal-based TumbleBit.

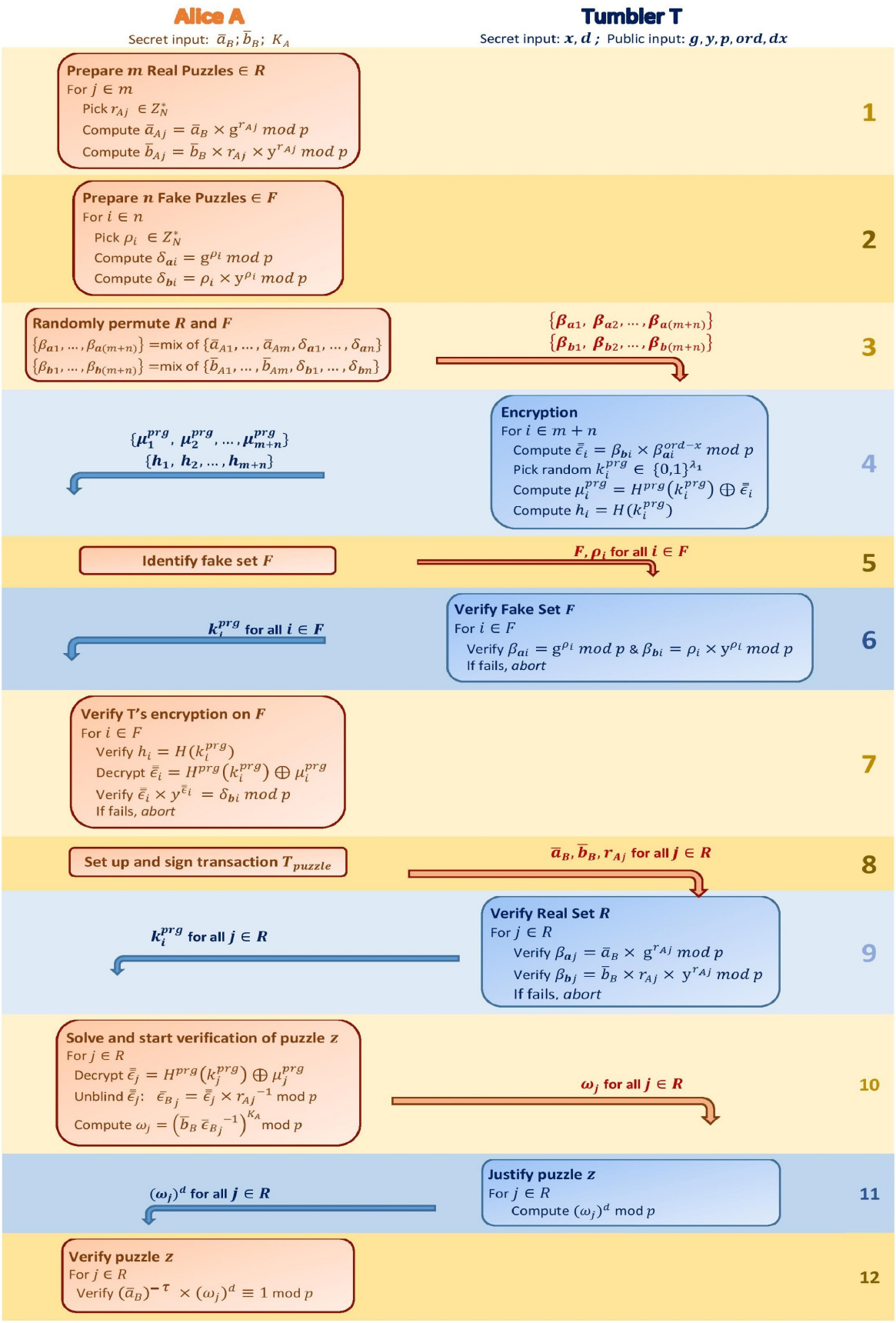



Figure B.2: Detailed puzzle-solver protocol for Elgamal-based TumbleBit.

# Appendix C

# Finding the size of the base64 encoded RSA secret keys

Base64 encoding converts byte arrays into ASCII strings. We present the algorithm in figure C.1 using the string "Hello, world!" as an example to be encoded. Its binary representation is

$$01001000011001010110110001101100011011110010110001110111011011110111001001101100011001000010000100001$$

The base64 index table in figure C.2 will have 64 characters, i.e. each character will consist of 6 bits. To encode "Hello, world!" we take 6 bits and convert it to the base64 character then we repeat. The result will be

$$SGVsbG8sIHdvcmxkIQ ==$$

This can also be viewed in a different way; every 3 octet[1] of the original data is converted into 4 sextet[2] of base64 characters. If the size of the input octets is not divisible by 3 then the base64-encoded string will be appended with one or two equal signs "=", depending on the number of the missing octets [24]. For example, "Hello, world!" is padded with two equal signs because it requires extra 16 bits to be added to its binary representation, as shown in figure C.1.

[1] 1 octet=8 bits
[2] 1 sextet=6 bits

| ASCII | H | e | l | l | o | , | [space] |
|---|---|---|---|---|---|---|---|
| Binary | 010010 00 | 0110 0101 | 01 101100 | 011011 00 | 0110 1111 | 00 101100 | 001000 00 |
| Base64 | S | G | V | s | b | G | 8 | s | l |

| ASCII | w | o | r | l | d | ! | | |
|---|---|---|---|---|---|---|---|---|
| Binary | 0111 0111 | 01 101111 | 011100 10 | 0110 1100 | 01 100100 | 001000 01 | **** **** | ** ****** |
| Base64 | H | d | v | c | m | x | k | l | Q | = | = |

Figure C.1: Algorithm of base64-encoding with example "Hello, world!"

The RSA secret keys found in the open-source github project in [16] are shown in figures C.3 and C.4. They are both 1592 characters long. Note, the first one has two paddings while the second one has only one. This means that the second secret key is 8 bits longer than the first one. Since our aim is to find only an estimation for the RSA secret keys size, we will randomly choose the size of the key shown in figure C.3 for further analysis. The original key size is

$$(1592 \times 6) - 16 = 9536 \text{ bits } = 1192 \text{ octets}$$

| Value | Encoding | Value | Encoding | Value | Encoding | Value | Encoding |
|---|---|---|---|---|---|---|---|
| 0 | A | 17 | R | 34 | i | 51 | z |
| 1 | B | 18 | S | 35 | j | 52 | 0 |
| 2 | C | 19 | T | 36 | k | 53 | 1 |
| 3 | D | 20 | U | 37 | l | 54 | 2 |
| 4 | E | 21 | V | 38 | m | 55 | 3 |
| 5 | F | 22 | W | 39 | n | 56 | 4 |
| 6 | G | 23 | X | 40 | o | 57 | 5 |
| 7 | H | 24 | Y | 41 | p | 58 | 6 |
| 8 | I | 25 | Z | 42 | q | 59 | 7 |
| 9 | J | 26 | a | 43 | r | 60 | 8 |
| 10 | K | 27 | b | 44 | s | 61 | 9 |
| 11 | L | 28 | c | 45 | t | 62 | + |
| 12 | M | 29 | d | 46 | u | 63 | / |
| 13 | N | 30 | e | 47 | v | | |
| 14 | O | 31 | f | 48 | w | (pad) | = |
| 15 | P | 32 | g | 49 | x | | |
| 16 | Q | 33 | h | 50 | y | | |

Figure C.2: Base 64 index table [25].

```
MIIEpAIBAAKCAQEArqDcYDN5G30tEr090DpwS7ZhASNTI/tHfZ/0qCHjjpQzKS5ocpgY6LGt9d
qXqmSb7J32Uu0kbp9yd3BU7rwdiin8n4ke18GnquCy+3LuomkSvvDHPoI4gfSa7pfaFwoeb/95
PA+1Z2WuIn6n7CICpzjNtMhrQUoUxKAxK/w2NjwjpXH+ypK6CyvkBOg+Jm5vLxUQPiwZWdemzn
IMmAT6rR1bMqnl5B17HQkRYlXrzVtWXH24dpWi8h3SXynNPa1je1XcUrY04VVj+tSh0zbF/+FG
vOu+3614sSgyymp/Y5GkdPhIcL/OamXtI7Dm1mf8JKQrS3YZiNWv8NzuC0cPDQIDAQABAoIBAQ
CqQXua75RsEKcZVhgmJFPH/ZCmI2Kz+hHBkWpfYmsSX7thszVyRIhElrcctHqaTm0XTQsT3tFe
aWaYvJSSDiZexoDeqjm/8bcfCuR186BmCDuWnfokH6IoUeucI8EmJV93ooT44ffjT2dvijz+JE
qh7ufgEmc/yJJnAifu0n+ob21EDGMROxhObzlvdCgAzKxTwFS7602eoFJEqHrzJW2MlCoIAsCV
5HvzN+jLvNFBUrclQkhcSPZTw8lVVij+7ygFmmk9LdiQDbx80prjXAHWdzDqdQ6dogSRP5wOFm
kWSpsfnotJJ8qr4qQU1ahhgI4uoueZrbxMCSq047W6H0WhAoGBANzUx40tmV1p37T56csZqkMJ
qEODctr5fNpHwIzMLPqBfa2BumTdY09vL/ven+9dFGpYcCdSndVmo6O5GqERpHLgaVJjZehVq+
rZP+tPyv1knBbIN4vDY8KJqpEx8u5TwqTFlftXfKunhw1Ued5cHl66OaEKGGFmzpc0/93X5L77
AoGBAMpwaCGvULKWbw6UWMY8usK5pbCy5xqhbbz1/83UHJbfC2EJs181cFjn4PwdWJkuRBFCrV
L3FvqnvR5cmRE6VFzTdECBUIhMjup1BCCMwKs0cWvzZdwsqfy4SQtooq4VBK4q8CbtBJ5ve1pI
VTSNe6ZD6tHPLNkTSz6QJpzB/uuXAoGAIhffrxYpJVjhJsmmpKqTtMHa7oFuzAvUkMafHZ2wHA
S96Hms00GywWLDH7mhK41NR5XbytZgc7/i57X+PHvCzdGDsSqTYylG330m9ydciX5s8r96g+No
lXI3mS8C+HSKCnJO030QTFgS66XtIr1KGwnZN9mdJfx/TKzsC833DN0CgYANyS/eM2EseEAh3w
pYnaQeRKQ670P7tUiS2LHN44b64z+UISP89NesxkW1BY6WB+eMwVyqTLdc5HVljurBmJZMJ/4/
sk64qXYGE2fvi4ZHYm/i6RVjtArzcd1PkJWbg0TpU3U9QWDSJUdiM5DR3lywSum/1fr44W57WT
e37yIcCwKBgQCtRICk3b7HJE+L9Mq7KCUUXC6WyCMIp7n6j84H6TjqeWTil4lT/pZKllVEDWx
IH9mtXgpVnj4RLvuk2jSC4TI/JecsQtqPi1raaZhkfiGu73HaesW2u6qzWtAq24EHkmMPyxjtP
1Gv2FXD3ksbC/1tZnKb4tFSmc8hBOKD3PbjQ==
```

Figure C.3: RSA secret key from github project in [16] in file "private_2048_test.pem"

```
MIIEpQIBAAKCAQEAtYjVMf/U6qibELdSoo9L6K1RKId9Bc0h6epIWIxfc5P4bXqVy65ISzKdF9
hu86LGUoGONyqmJV12Bs6AL59bTf6nBjuaz02T66jZ7+7PLRJEdcFfJgK5PeG1d9RVVUXiNvZ+
nA6X0TnTfWyNKPKEskXrAj56UfHefWucoW6nQrPeY+7idEw8BZ/WusFDJq5eFY7qKT2Pp2424h
uDPuQqfSBWMsQIvG26U9umYfBYEScwgOgpthXAqrEAvQW52QBFtLDG1gx+IXMRkur5b68c7ZNc
FYCuXd8XbPfDFQxfG4wkJ0MsAAtqR8nbfRG1x+/edhnAob5vsl+4jDjhp3E07QIDAQABAoIBAQ
Cux1FMVF4wl6jETVx4OrMBEG2FMf2DKl8ukWyaJHx6EMsszMuQUPo1vENXhAA+/kcnhKjiIGPa
scNDBmqQb0M/MqCo9ZY1ZSG30NgTk79QS7gyD72GITUsiTVoa7zE3wXPmT+JDNvKi3w909eMxw
JPidWcUX7/inmGE6e4HYtYQan8sJydB+rOm2TFers7JrOdinP6eGliCFwHDF/khUUtMX6QFlXS
/qWxBVwMRPUTx5SaoMgFRxCxLQeVKmpIdo3aU1pYCB/gZ0EtNm5b1496T+BJ/u/kvNwPX3o7Fc
zRG6FFyPzmOBM/zhIOkjG4vuhbOFiiq0Q/tFlH0C5+ezdZAoGBANt3XD+uxmrmX1++gQ2GuyjB
nT2v1bRG9apVX549P5F3/n0izElwIdPZ8ynBGdj6AS12Hcqk3k0Tt5Xmb16oTUK3W5zXSmyBWl
5Y0zG1TemptJjA1a9rC+sNimJ6NFXER1oBzygG5FEbSUhtzCoTGsi8M06mGLd86s2uxjFmKY+v
AoGBANPA/wnVYldv3f/hE+0wOUGpJxWbuEEvmWEl/xPhI+Wmc3zLsJvn6WvvOKkWnzMg/yR9G6
7IUiXcfbwzPLPu9h/Q+5hSDNOripACOiNcx+X8BSZNWATVMC1J6v9V8dlCdjA2C8aRJaG9MTNI
gmTmUb3IPlcxgLTL/e68JAbQ+nAjAoGBAKYk8HNNjuyyxpUnFW/2BY5PNQjUKsa1yZlP1HlpCM
zJuKFnTJx0BUfqgcmkZDr82RNfjiIxOhHD0YHr13gjlYniYbqUycTnEFDIkenZxcgVL6FMqviH
45fowlDXDvO3CgU7xWYaAZLdQ1dPt/ZKTEaXI4hw+dk++ksH+wa+vswFAoGANsNh8fQak8xdmJ
BoK95d4GpTrlXwaANcYnCGMGj4dKsAkRTInvlyN7TNbYVpNLri4Vftsd0iyaHlbqe9mjdBtebB
Opp1sMRbeHTq/jw/gm4UEtzL16we4oeMW+6pNmvmzv8bOoZNOjAIql+lQV0DZNaFsN0fkSJ0kK
pdktm2+wsCgYEAqoDuBlQGr8HY4Ln/xbRIXrmY0V5lndYcnfUqxzlmTAYPynxQ407qI02jEFuj
9HJKS71zZmNYaRBEv4P5f+RfpOXemGURjG6mcOSm12297ZU625nhHlKcZnYqxwLMgJCWrDkN8p
ulG5K3SCgFNI4TURmu3qJ35fNoC11T0ycJX20=
```

Figure C.4: RSA secret key from github project in [16] in file "private_2048_tumbler.pem"

# Appendix D

# Elliptic Curve Properties

A brief description of the Elliptic Curve was provided in section 3.4 along with its defining equation in (3.9). Here, we provide more background information on the discriminant, identity element and order of an EC. These are all required to gain a deeper understanding of the Elgamal ECC which will be used to implement an alternative design of the TumbleBit protocols in section 5.4.

## D.1 Discriminant of Elliptic Curves

Elliptic curves used in cryptography cannot have repeated factors for $x^3 + Ax + B$. In other words, the EC has to be non-singular which is true if and only if its discriminant is non-zero [10]. To achieve this, we define the discriminant [26] in (D.1).

$$\triangle = \text{ discriminant } = -16(4a^3 + 27b^2) \neq 0 \tag{D.1}$$

## D.2 Identity Element of Elliptic Curves

To understand the mathematics behind EC, we need to define the identity element $0_\infty$ which is also known as the point at infinity. It is the point that, added to any point $P$ on the curve gives the same point [27], i.e.

$$P + 0_\infty = P \tag{D.2}$$

## D.3 Order of Elliptic Curve

The order $O_E$ is the total number of points that satisfy the defining equation of the elliptic curve in (3.9). An EC defined over a finite field with modulus $p$ can only be applicable in cryptography if its order is comparable in size to $p$ [10]. In other words, the order of the curve $O_E$ has will satisfy the (D.3) [27].

$$|O_E - (p+1)| \leq 2\sqrt{p} \tag{D.3}$$

## D.4 Arithmetic operations in EC

In this section we define EC point addition. We use two points $P$ and $Q$ on an EC with coordinates $(X_P, Y_P)$ and $(X_Q, Y_Q)$ respectively and perform a point addition to obtain a third point $R$ with coordinates $(X_R, Y_R)$.

$$\begin{aligned} X_R &= S^2 - X_P - X_Q \bmod p \\ Y_R &= S(X_P - X_R) - Y_P \bmod p \end{aligned} \tag{D.4}$$

$S$ is the slope of the line passing through $P$ and $Q$ and its value will depend on $Q$. When $P$ and $Q$ are distinct, $S$ will be defined by (D.5), whereas when $P = Q$, $S$ is defined by (D.6). The latter process is also known as point doubling. Note, $A$ in (D.6) is a coefficient earlier defined in (3.9).

$$S = \frac{Y_Q - Y_P}{X_Q - X_P} \bmod p \tag{D.5}$$

$$S = \frac{3X_P^2 + A}{2Y_P} \bmod p \tag{D.6}$$

# Appendix E

# Speed of modular multiplication

We present a graph in figure E.1showing the results for the speed of modular multiplications with operand sizes ranging from 128 to 2048 bits. This graph is taken from [28].

The $k$ parameters stand for radix values, however for the TumbleBit evaluation section, we only require the approximate ratio of the speed of the 2048-bit and 256-bit operand modular multiplication. From this graph we can conclude that the 256-bit modular multiplication is approximately 50 times faster than the 2048-bit modular multiplication.

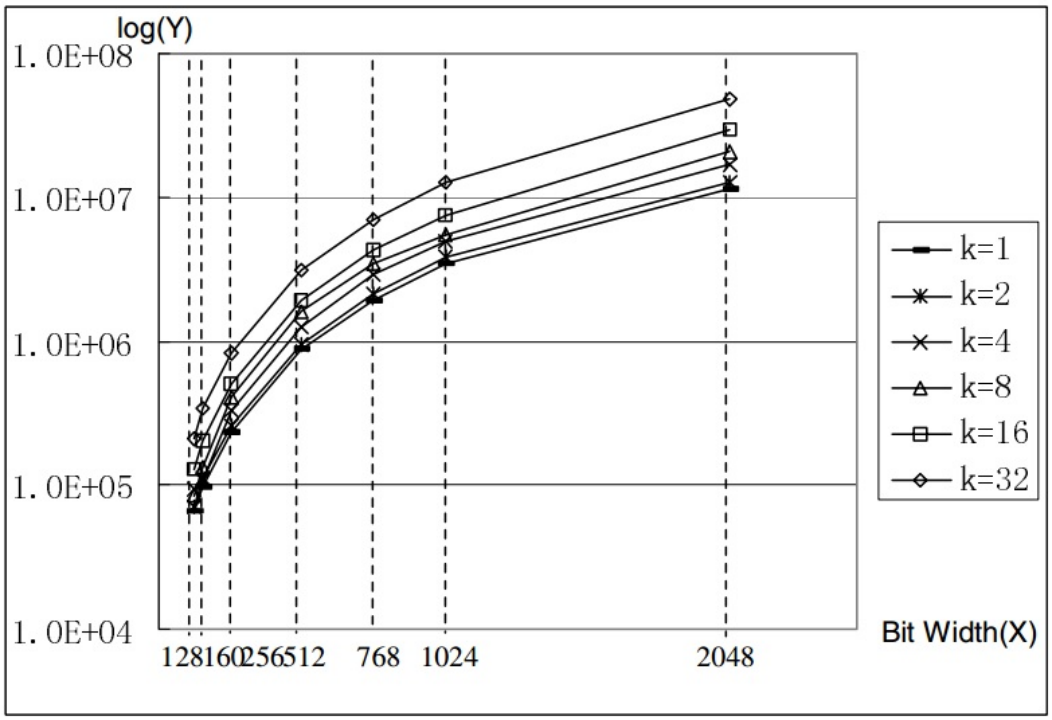



Figure E.1: Running time (ns) for a specific algorithm. [28]