Implementing Distributed Randomness

Adapting a random beacon protocol to run in practice

Master Thesis

Marius Asadauskas

Faculty of Science at the University of Bern
Cryptology and Data Security Research Group

Supervised by
Prof. Dr. Christian Cachin
Dr. Orestis Alpos & Mariarosaria Barbaraci

February 2024

Abstract

Creating randomness inside deterministic machines comes with many difficulties, especially when doing so in a distributed environment. However, there exist many random beacon protocols that accomplish just that. These protocols all vary in runtime, communication complexity, cryptographic primitives, and many other aspects. Despite the topic being heavily researched, distributed randomness still struggles to find mainstream adoption. In this thesis, we will examine how distributed randomness functions. We will also implement and improve upon an existing random beacon protocol and transform it from a synchronous to an asynchronous setting. Our work aims at finding reasons why distributed randomness has not found success in practice.

Contents

1 Introduction

The usefulness of randomness in our digital world cannot be overstated. From the most simple coin-tossing algorithm to the most complex neural network, algorithms require a certain amount of randomness to function. This stands in contrast to the deterministic nature of computers. Hence, it has been widely researched on how to overcome this hurdle and create randomness inside deterministic machines. Locally, this question has been mostly solved. Ranging from insecure solutions such as linear-feedback shift registers, where one simply shifts bits and uses linear operations to get randomness, to secure solutions such as pseudorandom generators, entropy collection, and many more. Furthermore, solutions inside hardware exist, such as trusted execution environments [1], chemical reactions, and others that guarantee randomness through aspects outside of software. These methods are all great for private randomness, where one creates and consumes a random value internally. However, when it comes to public randomness, these methods do not suffice.

Contrary to private randomness, public randomness is needed when the public has to be able to trust a random value and verify its authenticity. An old-fashioned example of public randomness would be the lottery. The public needs to be able to trust the randomness of the lottery, since lottery companies have a significant financial incentive to rig the numbers. If the lottery were to simply create a private random value and tell it to the public, there would be no reason to believe its authenticity. Hence, to calm the doubts of the public, companies normally put on a show. A common way they do so is by recreating the urn model. They toss balls with numbers into an urn, mix the balls, and then randomly pick out balls from the urn to create the final lottery numbers. Recording the process and then putting it on television for anybody to see. In doing so, they give the public a way to verify the authenticity of the random value, and transform private randomness into public randomness. While a great solution in practice, it does not translate well into our digital world. Usually, a computer can not wait a week for a random value to be created, as is the case with most lotteries.

Furthermore, public randomness is not only needed in cases such as lotteries and other forms of gambling. Another example would be random inspections. Multiple studies have found that police searches are unfair and target minorities disproportionately [2]. By using public randomness, one could ensure the honesty of random inspections and prove that they are not biased by race or other factors.

As of the writing of this thesis, there are only a couple of public randomness providers. One of which is a service known as DRand [3], which provides a verifiable random number every 30 seconds for the

public to use as they see fit. DRand does so by running a cryptographic protocol on a distributed network of nodes that creates a random number. A cryptographic protocol that creates randomness is also known as a random beacon protocol, and the process can be seen as creating distributed randomness.

1.1 Motivation

Although there are only a couple of random beacons running in practice, the topic of distributed randomness is heavily researched. As seen in Table 1.1, there exist a plethora of random beacon protocols that all vary in cryptographic primitive, communication complexity, unpredictability, and many other aspects. This leaves us with many options to choose from when trying to implement distributed randomness. We aim to pick out one or multiple random beacon protocols and implement them ourselves, as to find out why such a researched topic remains somewhat unused in practice.

Protocol	Fault Tolerance	Comm. Compl.		Unpredict-ability	Cryptographic		Network
		Best	Worst		Primitive	Assumption	
Commit-Reveal	$f = 0$	$O(n^2)$	$O(n^2)$	1	Commit	DDH	Async.
Albatross [4]	$n > 2f$	$O(n^3)$	$O(n^4)$	1	PVSS	DDH	Sync.
Bicorn [5]	$n > 2f$	$O(n^2)$	$O(n^3)$	1	Timed commit	RSW	Sync.
Drand [6]	$n > 2f$	$O(n^2)$	$O(n^3)$	1	Thr. BLS	uniq. Sig, CDH	Sync.
Dfinity [7]	$n > 2f$	$O(n^2)$	$O(n^3)$	$f + 1$	Thr. BLS	uniq. Sig, CDH	Sync.
GULL [8]	$n > 2f$	$O(n^3)$	$O(n^3)$	1	PVSS	DDH	Sync.
GRandPiper [9]	$n > 2f$	$O(n^2)$	$O(n^2)$	$f + 1$	PVSS	q-SDH, SXDH	Sync.
BRandPiper [9]	$n > 2f$	$O(n^2)$	$O(n^3)$	1	(P)VSS	q-SDH, SXDH	Sync.
HERB [10]	$n > 3f$	$O(n^2)$	$O(n^3)$	1	Thr. ElGamal	DDH	Sync.
HydRand [11]	$n > 3f$	$O(n^2)$	$O(n^3)$	$f + 1$	PVSS	DDH	Sync.
OptRand [12]	$n > 2f$	$O(n^2)$	$O(n^2)$	1	PVSS	q-SDH, SXDH	Sync.
RandHerd [13]	$n > 3f$	$O(c^2n)$	$O(n^4)$	1	Thr. Schnorr	DL	Sync.
RandHound [13]	$n > 3f$	$O(c^2n)$	$O(c^2n^2)$	1	PVSS	DL	Sync.
RandShare [13]	$n > 3f$	$O(n^3)$	$O(n^4)$	1	(P)VSS	DL	Async.
RandRunner [14]	$n > 2f$	$O(n)$	$O(n^2)$	$f + 1$	Trapdoor VDF	tVDF, DL	Sync.
SPURT [15]	$n > 3f$	$O(n^2)$	$O(n^2)$	1	PVSS	DBS	Psync.
SCRAPE [16]	$n > 2f$	$O(n^3)$	$O(n^4)$	1	PVSS	DDH/DBS	Sync.
STROBE [17]	$n > 2f$	$O(n^2)$	$O(n^3)$	1	VSS	RSA, DL	Sync.
Unicorn [18]	$n > 2f$	$O(n^2)$	$O(n^3)$	1	VDF	RSW	Sync.

Table 1.1: Existing random beacon protocols

In Table 1.1, we documented existing random beacon protocols and their characteristics. With a majority of the data being gathered from a summary of knowledge on distributed random beacons by Choi et al. [19] and the rest being extracted from the individual random beacon research papers. This table serves to emphasize the amount of research that has been done in the field of distributed randomness and acts as a guide when choosing which protocols to implement.

The first column of Table 1.1 contains the names of the random beacon protocols and the papers in which they were first described. The second column describes the fault tolerance of each protocol. It is important to note that n defines the network size, while f describes the number of adversaries the network can tolerate. For example, $n > 2f$ is known as an honest majority assumption and implies that a verifiable random output will be created, as long as more than half the network is honest. The third and fourth columns describe the communication complexity of each protocol with regard to the network size n. We differentiate between the best-case scenario, where there is no adversarial behavior, and the worst-case

scenario, where the maximum tolerated amount of adversarial behavior is present. Sending a bit via a point to point message has a complexity of $O(1)$, while a broadcast has one of $O(n)$. Furthermore, a client specified parameter c is relevant in RandHerd [13] and RandHound [13]. The fifth column describes the unpredictability of a protocol. With 1 being the optimal value, it refers to the number of protocol iterations a rushing adversary can predict into the future. A rushing adversary is one that can see and send all messages before every other participant. This value is at least 1, since a rushing adversary can always see the output of the protocol before every other participant. However, in some protocols, this value lies at $f + 1$ since they contain leadership elections and a rushing adversary can control up to f leaders in a row. Hence, to ensure that a random value is unknown, a user must wait $f + 1$ iterations in some protocols and only a single one in others. The sixth column describes the cryptographic primitive that is used in the protocol, such as verifiable secret sharing (VSS), and publicly verifiable secret sharing (PVSS) using secret sharing, verifiable delay functions (VDF), trapdoor VDF, and timed commitments using some form of proof of work, and finally threshold BLS, ElGamal, and Schnorr using threshold signatures. The seventh column describes the cryptographic assumption that must hold for the randomness generation to be secure, with some assumptions being more strict compared to others. The final column describes the network setting required for the protocol to function. This can be either synchronous, partially synchronous [20], or asynchronous. A synchronous network is one where messages are guaranteed to arrive after a finite time bound. In asynchronous networks, there exists no time bound, and messages may arrive after any finite amount of time. Finally, partially synchronous networks combine both modes by being asynchronous until a so-called global stabilization time (GST) happens, after which they are synchronous.

2
Background

In this chapter, we present the background knowledge needed to understand how random beacons function. This includes cryptographic concepts and assumptions, commit schemes, secret sharing, and further topics.

2.1 Cyclic groups

To understand how random beacons function, we must first understand what kind of output we can expect from them. After all, the statement "Our random beacon protocol output 1" does not deliver any sort of meaning until we know what other outputs were possible. Assuming the output was uniformly chosen from $\{0, 1\}$, then 1 had a chance of $1/2$, but from the set $\{0, \ldots, n-1\}$, it had a chance of $1/n$. To more clearly define the space in which random beacons function, we must take a closer look at group theory.

In group theory [21], a group $\mathbb{G}$ is defined as a set of elements together with an operation $< \mathbb{G}, \cdot >$, for which the following four properties hold

1. **Closure:** $\forall a, b \in \mathbb{G} : a \cdot b \in \mathbb{G}$
2. **Associativity:** $\forall a, b, c \in \mathbb{G} : a \cdot (b \cdot c) = (a \cdot b) \cdot c$
3. **Identity element:** $\exists e \in \mathbb{G}, \forall a \in \mathbb{G} : e \cdot a = a \cdot e = a$
4. **Inverse element:** $\forall a \in \mathbb{G}, \exists b \in \mathbb{G} : a \cdot b = b \cdot a = e$.

The simplest, yet one of the most interesting groups, is the cyclic group. Which is defined by a single element g of order q and an operation $\cdot$, such that

$$\mathbb{G} = \{g^0 = e, g, g^2, \ldots, g^{q-1}\}, |\mathbb{G}| = q.$$

If we define our outputs to be uniformly chosen from a cyclic group, statements such as "Our random beacon protocol output 1" suddenly gain meaning. Furthermore, the element g is known as a generator and acts as an important cryptographic tool.

2.2 Cryptographic assumptions

To prove the security of a cryptographic protocol, one must often make some underlying assumptions. These assumptions allow for authentication schemes, zero-knowledge proofs, and many other protocols to be provably secure as long as the underlying assumptions hold. In the following section, we will take a closer look at some standard assumptions that are important when generating randomness.

2.2.1 Decisional Diffie-Hellman assumption

A commonly used assumption when generating randomness is the decisional Diffie-Hellman assumption. It states that given a cyclic group $\mathbb{G}$ of order q, a generator g, and two triples

$$(g^a, g^b, g^{ab}), (g^a, g^b, g^c),$$

it is hard to know which triple was created using two random variables $a, b \in \mathbb{Z}_q$, and which one was created using three random variables $a, b, c \in \mathbb{Z}_q$. In other words, the decisional Diffie-Hellman assumption states that g^{ab} seems like a random value, even if one knows g^a and g^b. This assumption is crucial when generating random values. It is also important to note that a, b, c are finite field elements in $\mathbb{Z}_q$ and g^a, g^b, g^c, g^{ab} are group elements in $\mathbb{G}$.

The decisional Diffie-Hellman assumption builds upon the discrete logarithm problem, which states that given two group elements x and y in $\mathbb{G}$, it is hard to find $k \in \mathbb{Z}_q$ such that

$$x = y^k.$$

2.2.2 Random oracle model

Another commonly used assumption when generating randomness is the access to a random oracle. An oracle being an entity that provides wisdom, and the randomness property being the lack of predictability in events. In other words, a random oracle provides an unpredictable answer to any question. Furthermore, once a random oracle provides a value, they remember the randomness they provided and will deliver the same output to a specific input. If one party asks a random oracle a question and another party asks the exact same question, they will both receive the exact same answer. An ideal random oracle can be seen in Algorithm 1, if we assume the question to be an input of any length and the answer to be an output of a fixed length. Furthermore, with $:=$ we define values, with $=$ we compare values, and with $\leftarrow$ we randomly pick a value from a set.

Algorithm 1 Ideal random oracle

```
state
    A := [ ]                                  //Initialize an empty directory
H(x)
    if A[x] = ⊥ then
        A[x] ← {0,1}^l                        //Create a random value and save it under x
    return A[x]
```

In practice, random oracles are realized using cryptographic hash functions. A famous example, and the hash function used in our thesis, being SHA-256 [22]. Using only bit shifts and logical operations, a hash function deterministically turns any input into a seemingly random output. The use cases for such a function are vast and include fingerprinting, prime number generation, improving storage, and many more. When generating randomness, hash functions can also be used in multiple ways.

The simplest way would be to decouple the output from the input. Assume we have a function $f(x) := x$. A malicious party could choose the exact output of a function by changing x. However, turning the function into $g(x) := H(f(x))$ would not allow an adversary to determine the exact output. It would only allow an adversary to pick a preferable output. Hence, adding a hash function on top of a random number generator provides an extra layer of security. Another use case for random oracles lies in the creation of non-interactive zero-knowledge (NIZK) proofs.

2.3 Non-interactive zero-knowledge proofs

To understand NIZK proofs, we must first understand zero-knowledge proofs in general. Hence, in the following section, we will first go into detail on zero knowledge proofs by looking at a concrete example that is used in multiple random beacon protocols. Namely, zero-knowledge proofs of equality, which were first described by Chaum and Pedersen [23]. A zero-knowledge proof of equality is used to prove that two values are equal in their exponent. In other words, given two generators g_1, g_2 and two values y_1, y_2 one can prove that

$$\log_{g_1}(y_1) = \log_{g_2}(y_2) = x$$

without revealing the secret value x. How this is done can be seen in Figure 2.1.

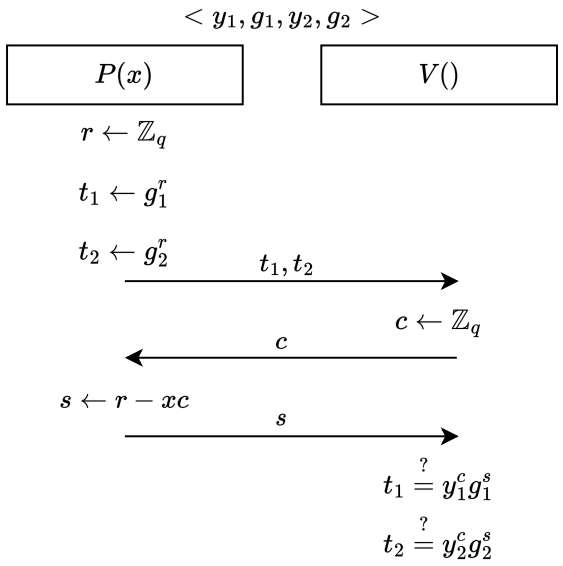


Figure 2.1: Illustration of a zero knowledge proof of equality

In Figure 2.1 our protocol consists of two parties, the prover and the verifier. Furthermore, two generators g_1, g_2, two values y_1, y_2, and the order of the underlying group q are assumed to be known. In this zero-knowledge proof, the prover wants to show that $\log_{g_1}(y_1) = \log_{g_2}(y_2) = x$ without revealing x. If they could reveal x, it would be as simple as sending x to the verifier, who could then check that $g_1^x = y_1$ and $g_2^x = y_2$. What the prover does instead is, firstly, they create a random value $r \leftarrow \mathbb{Z}_q$. This random value is also known as the blinding factor. The prover then raises both generators to the power of the blinding factor $t_1 \leftarrow g_1^r$ and $t_2 \leftarrow g_2^r$ and then sends both to the verifier. Once the verifier receives these values, they create a challenge by picking a random value $c \leftarrow \mathbb{Z}_q$. The prover then responds to

the challenge by sending back $s \leftarrow (r - xc)$. Once the verifier receives s and checks that $t_1 = y_1^c g_1^s$ and $t_2 = y_2^c g_2^s$ the protocol is over and the prover has shown to the verifier that $\log_{g_1}(y_1) = \log_{g_2}(y_2) = x$.

This interactive protocol can be considered a zero-knowledge proof, as it fulfills the three properties that are required to be a zero-knowledge proof.

1. **Completeness**: If the statement is true, the verifier will be completely convinced by its truthfulness.
2. **Soundness**: If the statement is false, no prover can convince a verifier that it is true, except with a negligible probability.
3. **Zero-knowledge**: If a statement is true, a verifier learns no information other than the fact that it is true.

Completeness holds, since for $i \in \{1, 2\}$

$$\begin{aligned} y_i^c g_i^s &= y_i^c g_i^{r-xc} \\ &= y_i^c g_i^r (g_i^x)^{-c} \\ &= t_i y_i^c y_i^{-c} \\ &= t_i. \end{aligned}$$

Soundness holds, since given two valid transcripts $(t_1, t_2, c, s), (t_1, t_2, c', s')$ with $c \neq c'$ one can extract x by calculating

$$\begin{aligned} t_i = y_i^c g_i^s &= y_i^{c'} g_i^{s'} \\ y_i^{c-c'} &= g_i^{s'-s} \\ g_i^{x(c-c')} &= g_i^{s'-s} \\ x(c-c') &= s' - s \pmod q \\ x &= \frac{s'-s}{c-c'} \pmod q. \end{aligned}$$

Furthermore, the proof is **zero-knowledge**, since one can create transcripts (t_1, t_2, c, s) of real proofs with the same distribution as the real proof by computing

$$\begin{aligned} c &\leftarrow \mathbb{Z}_q \\ s &\leftarrow \mathbb{Z}_q \\ t_1 &\leftarrow y_1^c g_1^s \\ t_2 &\leftarrow y_2^c g_2^s. \end{aligned}$$

To extend a zero knowledge proof and make it non-interactive, we must add one further property.

4. **Non-interactiveness**: The statement can be proven without the verifier having to interact with the prover.

To accomplish this, we must remove any interactions between the prover and the verifier. With the only interaction being when the verifier creates a challenge $c \leftarrow \mathbb{Z}_q$. We do so by using a random oracle instead. The idea being that instead of requesting a challenge from a verifier, we ask the oracle to create a challenge by calculating $c \leftarrow H(t_1||t_2)$ and $s \leftarrow (r - xc)$ and then sending (c, s) to the verifier. Once the verifier receives (c, s) they can calculate t_1 and t_2 and then check that the challenge was created correctly. An illustration of a non-interactive zero-knowledge proof of equality can be found in Figure 2.2.

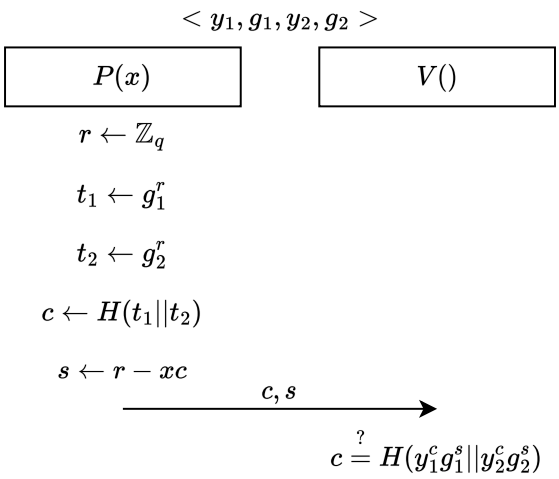


Figure 2.2: Illustration of a non-interactive zero-knowledge proof of equality

2.4 Commits and reveals

There are two main phases that appear in most distributed randomness protocols. The commit phase and the reveal phase. The commit phase allows participants to lock in their values without revealing them. Meanwhile, the reveal phase does the opposite. It allows participants to open up their committed values once they do not have to be kept secret anymore.

These phases appear in most distributed randomness protocols since distributed randomness should be created by all participants equally, however, each participant should choose a random value independently of all other participants. Hence, participants first commit to their values, and only once everyone has committed to a value, can they reveal them, and the final randomness gets calculated.

2.4.1 Commit phase

As mentioned before, the commit phase is where participants lock in their secret values. This is done with a one-way commit function, $Com(r, s)$. As an input, a commit function takes a secret value s and some randomness r, also known as the blinding factor. These inputs then get put through a one-way function, meaning that given $Com(r, s)$ there exists no efficient way to calculate $Com^{-1}()$, that leads to the inputs (r, s). The only way to find out the inputs of a commit function is to wait for the commit to be revealed or to test all possible inputs. Furthermore, testing all possible inputs should be computationally infeasible or mathematically impossible.

This leads us to the different commit schemes that exist. A commit function can either have perfect binding or it can have perfect hiding. With perfect binding, the secret value s cannot be changed once it has been committed to. With perfect hiding, the secret value cannot be extracted from the commit before it has been revealed. It is impossible to have both, since perfect binding implies computational hiding and perfect hiding implies computational binding [24]. To illustrate the difference, we shall take a common example of a commit function using a random oracle,

$$Com(r, s) := H(r||s).$$

If we set the length of the blinding factor r to zero and assume $s \in \{0, 1\}$ to be the only meaningful

input $Com(r, s) := H(s)$, then the commit function has perfect binding and computational hiding. The function has computational hiding since, given a commit $H(s)$ an adversary could compute all possibilities $H(0)$ and $H(1)$, as to find out which value s was committed to. Furthermore, the commit function has perfect binding since a committed value s cannot be changed, assuming $Com(r, 0) = H(0) \neq H(1) = Com(r', 1)$.

If we instead set the length of the blinding factor r to be unlimited and assume $s \in \{0, 1\}$, then we have perfect hiding and computational binding. The function has computational binding, since an adversary could compute new blinding factors r' until they found a collision $Com(r, s) = Com(r', \neg s)$. This collision exists, since the length of r is unlimited and the output length of the random oracle is limited. Once a collision is found, an adversary could reveal $(r', \neg s)$ after committing to (r, s), meaning that the commit function has computational binding. However, it has perfect hiding since, given $Com(r, s)$ both s and $\neg s$ are possible. Hence, the secret value s can never be found out.

Summarizing, perfect binding protects the integrity of the commit function, while perfect hiding protects the secret value s. When generating randomness, both schemes are valid. If an adversary could figure out all other commits, they could choose a fitting value to commit to, which would lead to a desired output. If an adversary could change their commit, they could wait for all commits to be revealed and reveal a value that would lead to a desired output. In both cases, an adversary with infinite computational power could influence the outcome, however, an adversary with limited computational power could influence neither. But perfect binding forces an adversary to calculate all other commits, while perfect hiding only forces them to change a single commit. Hence, perfect binding imposes a higher computational toll on an adversary.

2.4.2 Reveal phase

In the reveal phase, a party simply has to open their commit. This is usually done by broadcasting the values used inside the commit function. A reveal looks as follows,

$$Open(r, s) := (r, s).$$

Once received, participants can compare the reveal to the previous commit $Com(Open(r, s)) = Com(r, s)$. Furthermore, the timing of a reveal is important. A reveal should only be done once enough participants have committed to their values. An early reveal would make the previous commit ineffective. A late reveal would slow down the flow of a protocol. Lastly, a reveal that never arrives can cause issues. As mentioned, if a party does not decide to reveal a commit themselves, it is impossible or computationally infeasible to reveal it in their stead. Hence, the protocol could lose liveness. For this reason, some commit schemes implement a recovery option, where parties can work together to recover a value.

2.5 Two-party coin flip

To give a concrete example of distributed randomness, we shall take a closer look at one of the simplest examples. A two party coin flip. In other words, a protocol where two parties interactively generate and agree upon a random value in $\{0, 1\}$.

The protocol illustrated in Figure 2.3 consists of two phases, as mentioned in Section 2.4. A phase in which the participants commit to their random values, and a phase in which they reveal their committed values. The protocol starts with both Alice and Bob flipping a random coin themselves. We will later add these coin flips together modulo 2, as to get the resulting random coin flip. In the commit phase, Alice and Bob both lock in their coin flips with the help of a random oracle and a blinding factor, e.g. by computing $Com(r, s) := H(r||s)$. This ensures that the committed coin flips cannot later be changed. Since the blinding factor has variable length $r \leftarrow \{0, 1\}^*$, we have perfect hiding and computational binding. Once

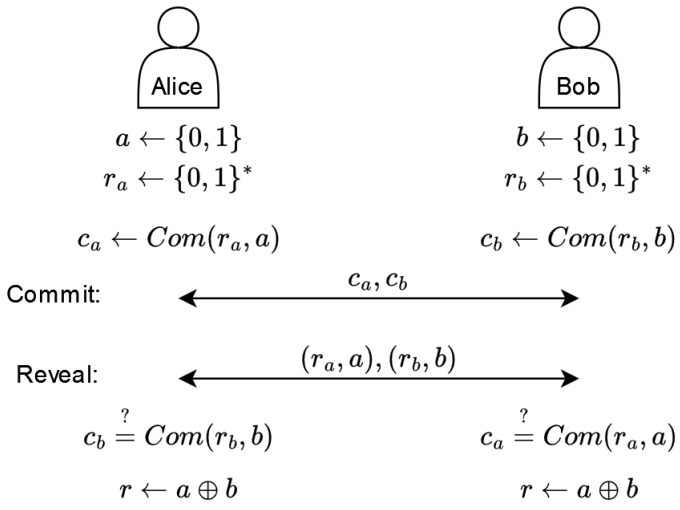


Figure 2.3: Illustrating example of distributed randomness

Alice and Bob have committed to their randomness, both know that neither can change their coin flips a and b anymore. Hence, the reveal phase begins, where both parties reveal the values that were used to create the commit. They do so by sending (r_a, a) and (r_b, b) to each other. To ensure that nothing was changed, Alice must check that the commit of (b, r) is equal to the value previously received, and Bob must do the same. Once the verification is complete, both Alice and Bob can be sure that a and b are independent of each other. The final randomness then gets calculated by taking both values and adding them together, modulo 2. In other words, XORing both values together.

This protocol guarantees security as long as a single party is honest. Even if the other party behaves maliciously, the resulting coin flip will still be random. To show this, we first define two random variables, $A :=$ "The coin flip of Alice" and $B :=$ "The coin flip of Bob". Furthermore, we define $Z := A \oplus B$. W.l.o.g. we assume that the coin flip of Alice is honest. It follows that $P[A = 0] = P[A = 1] = \frac{1}{2}$ and

$$\begin{aligned} P[Z = 0] &= P[A = 0 \cap B = 0] + P[A = 1 \cap B = 1] \\ &= P[A = 0]P[B = 0] + P[A = 1]P[B = 1] \\ &= \frac{1}{2}P[B = 0] + \frac{1}{2}P[B = 1] \\ &= \frac{1}{2}(P[B = 0] + P[B = 1]) = \frac{1}{2}. \\ P[Z = 0] = \frac{1}{2} &\implies P[Z = 1] = \frac{1}{2}. \end{aligned}$$

We see that no matter what distribution Bob chooses, the resulting coin flip remains fair.

The main issue, however, with this type of randomness generation protocol is that it does not guarantee liveness. When a single party behaves maliciously, they can prevent the protocol from ever finishing. This stems from the fact that we cannot force a party to open their commit unless they desire to do so. If a participant does not open their commit, the reveal phase never ends, and the protocol does not lead to a result. Hence, for this protocol to work, we need both parties to participate in it and a single party to honestly generate a random coin flip.

This example provides good insight into how distributed randomness generates values. The protocol

itself does not create randomness, instead, it takes values from multiple parties and adds them together into a completely new random value. However, we must ensure liveness, since doing so ensures that the protocol has a guaranteed random output. For this, secret sharing is a viable option.

2.6 Secret sharing

When keeping a secret, such as a password or any other type of sensitive information, storing it on a single device might seem like the safest option. Only having to worry about a single point of failure makes a secret more manageable. However, this single point of failure also implies that only one device must fail for the secret to be lost forever. A possible solution would be to create backups of the secret by copying it onto multiple devices. This would ensure liveness, but it would decrease security since there would now be multiple possibilities for the secret to be revealed. To solve these issues, secret sharing can be applied.

In $(f+1)$-of-n secret sharing, one can convert a secret into n secret shares and then distribute these among n participants. To then recover the secret, it takes any set of $f+1$ distinct shares. In the case of $(f+1)$-of-n secret sharing, the number $f+1$ is also known as a quorum, as it describes the minimum number of nodes needed to reconstruct the secret. In other words, even if $n-(f+1)$ shares get lost, as long as a quorum of nodes still exists, one can recover the secret. Furthermore, an adversary with f or fewer shares cannot create a quorum and should know nothing about the secret.

For this reason, the number of participants n inside a network is often chosen depending on how many adversaries one can tolerate. For example, $n > 2f$ implies that one has an honest majority, since f is the number of adversarial or faulty nodes and n is the network size.

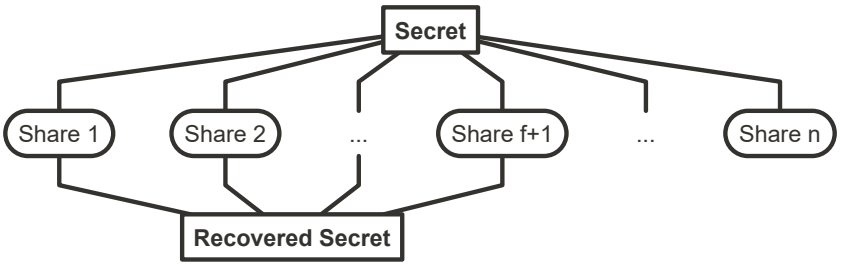


Figure 2.4: Example of $(f+1)$-of-n secret sharing

2.6.1 Polynomial secret sharing

A concrete example of secret sharing is polynomial secret sharing, also known as Shamir's secret sharing [25]. With the main idea being that a polynomial of degree f is clearly defined by any $f+1$ distinct points. Hence, one can put a secret at position zero of a polynomial and share the values $p(i)$ with nodes $i \in \{1,\dots,n\}$. To then reconstruct the secret at position zero, $f+1$ nodes must work together and interpolate the point at position zero. Since we assume that there are at most f adversaries, we can be sure that the secret can only be reconstructed with the permission of an honest party. More clearly, to do $(f+1)$-of-n polynomial secret sharing, one must follow the following steps.

1. Create f random coefficients $\{c_1,\dots,c_f\}$ from a finite field e.g. $\mathbb{Z}_q$. Together with the secret s the coefficients define the polynomial at position zero. Said polynomial looks as follows, $p(x) := s + c_1x + x_2x^2 + \cdots + c_fx^f$ and also results in a value from the finite field.

2. Send secret shares $p(i)$ to participants $i \in \{1,\dots,n\}$ over secure channels.

3. Delete the secret s yourself, as to avoid a single point of failure.

To then recover the secret, one has to collect $f + 1$ secret shares and use Lagrange interpolation to reconstruct the secret from its shares.

2.7 Theoretical lower limit

When talking about distributed or public randomness, it is important to understand, that it is not here to replace private randomness. Firstly, to create a distributed random value, one often needs private randomness. Furthermore, there exists a theoretical lower limit that comes with distributed randomness and distributed computing in general. The limit arises when taking physical distance into consideration. If we take a look at the two-party coin flip from Section 2.5 and imagine one party being in Switzerland and the other party being in New Zealand, which is practically on the other side of the world. We know that information cannot travel faster than the speed of light. Assuming we do not dig a hole through the center of the earth, the minimum amount of time for any information to reach New Zealand from Switzerland lies at around $\frac{19'000'000m}{300'000'000m/s} \approx 0.063sec.$ This can be seen as a minimal communication delay between the two parties. Furthermore, the protocol consists of two phases, so the shortest runtime for the protocol lies at around $0.126sec.$

Even with it being a minimal time, it is orders of magnitude slower than what it takes a computer to generate a random coin flip locally. Furthermore, there is no clear solution to this issue. One way would be to reduce the distance between participants, however, this would lead to the system being less distributed and more restrictive. Another solution would be to generate multiple numbers in batches, however, getting multiple random values at once rarely replaces getting multiple random values in succession. Hence, it should be expected that distributed randomness will take a longer amount of time to generate a random value. It is not meant to replace all the random values we consume. Instead, it provides a form of verifiable randomness for the public to believe in.

3
Design

3.1 SCRAPE explained

For our random beacon implementation, we chose to implement the protocol defined by Cascudo and David in their 2017 paper titled "SCRAPE: Scalable Randomness Attested by Public Entities" [16]. There were multiple reasons for this choice. Firstly, while researching other papers, we found that many led back to SCRAPE. Famous examples include Albatross [4] by the same authors and OptRand [12] by Bhat et al. Albatross takes SCRAPE and applies packed Shamir secret sharing to it. This allows for the generation of $O(n^2)$ uniformly random values instead of a single value. OptRand improves upon SCRAPE by having an epoch leader that handles most of the network traffic and processing. Both of these protocols improve upon SCRAPE, but they still stem from it. Furthermore, they add complexity to the protocol and are more difficult to implement. For our research, a simple, state-of-the-art protocol sufficed.

Objective reasons for choosing SCRAPE were its cryptographic primitive, unpredictability, honest majority assumption, reusable setup, the fact that all it needs, communication-wise, is a public ledger, its linear number of exponentiations per verification, and many more.

Regarding its cryptographic primitive, SCRAPE uses PVSS, which stands for publicly verifiable secret sharing. Allowing for any entity, even ones not involved in the protocol, to verify and validate that all computations were done correctly. This seemed the most fitting for generating public randomness. Furthermore, SCRAPE has a perfect unpredictability of 1, meaning that the next random value is not known by any party. SCRAPE also has an honest majority assumption. As long as $n > 2f$, with n being the number of nodes and f the number of malicious or faulty nodes, SCRAPE will deliver an honest output. Furthermore, the setup of SCRAPE is reusable, meaning that it only has to be done once in the first iteration. In later iterations, parties can reuse the public keys exchanged in the setup phase. As long as a public ledger or a blockchain exists, where users can post data, SCRAPE can function. It does not rely on point to point links, broadcasts, or any other means of communication other than a public ledger. SCRAPE was also the first PVSS based protocol to only require a linear number of exponentiations per verification, meaning that it is more efficient than its predecessor's computation wise.

All of these factors are what set SCRAPE apart from other works and led to our choice. However, SCRAPE was also problematic in certain aspects compared to other works. One aspect being that with a maximum amount of malicious behavior, SCRAPE has a communication complexity of $O(n^4)$, as

can be seen in Table 1.1, and requires a network to do $O(n^2)$ Lagrange interpolations, to calculate the final randomness. Lastly, SCRAPE requires a synchronous network to function, while asynchronous assumptions simplify implementation efforts and allow for more natural and robust protocols. We will address these issues in Section 3.2 after explaining SCRAPE.

3.1.1 Phases of SCRAPE

The random beacon protocol described in SCRAPE is made up of four phases. The setup phase consists of waiting for n public keys. This only happens in the first iteration and in later iterations, the protocol consists of the remaining three phases. The commit phase is where the protocol waits for $f + 1$ commits. The reveal phase is where the protocol waits for all commits to be opened. Lastly, the recovery phase is where the protocol waits for $f + 1$ recovery shares for each unopened commit.

It is important to note that the setup and the reveal phases of SCRAPE are what make the protocol synchronous. Waiting for n public keys and waiting for all commits to be opened is only possible in a synchronous network setting, since one has to know when to stop waiting. In the following subsection, we will go into detail on all four phases.

3.1.1.1 Setup phase

Before a party is allowed to participate in the random beacon protocol, their public key must be known to all other participants. For this, each party must complete a setup where they create a public and secret key pair. The public key must then be shared with all other participants. To do this, a party must first pick a random value from $\mathbb{Z}_q$ which is then set as their secret key

$$sk \leftarrow \mathbb{Z}_q.$$

Afterward, the public key gets derived from the secret key. This is done by calculating

$$pk := h^{sk}.$$

With h being a generator of the underlying cyclic group $\mathbb{G}$. The sharing of the public keys is then done through the public ledger. A participant must simply post their public key to take part in the protocol. The setup in pseudocode form can be seen in Algorithm 2. With Post(val) being the function to post values onto the public ledger and Receive(val) being the function invoked after a value is posted onto the public ledger. Furthermore, we represent each participant with a unique process inside $\{p_1, \ldots, p_n\}$.

Algorithm 2 Setup (process p_i).

1: **state**
2: $sk := 1$
3: $pks := [\perp]^n$ //Empty array of public keys

4: **upon event** INIT **do**
5: $sk \leftarrow \mathbb{Z}_q$ //Create a random secret key
6: **invoke** Post(h^{sk})

7: **upon** Receive(pk_j) from p_j **such that** $pks[j] = \perp$ **do**
8: $pks[j] := pk_j$

As mentioned before, this process must only be done once when starting the protocol. Afterward, all participants know each other's public keys, and the protocol may be run as many times as needed.

Furthermore, the setup can be implemented in such a way that nodes can be replaced, removed, and added at will, with only the affected nodes having to run the setup again. However, proactive key refreshes should still be possible in the case that a key has been used too often.

3.1.1.2 Commit phase

Once the setup is complete, all n public keys are known, and the protocol moves on to the commit phase. This can also be seen as the secret sharing phase, with randomness being the secret. Each participant starts by picking a random value $s \leftarrow \mathbb{Z}_q$ and uses $(f+1)$-of-n polynomial secret sharing to share this value among all other participants, as explained in Section 2.6. For this, a party defines a random polynomial $p(x) := s + c_1 x^1 + \cdots + c_f x^f$ which has degree f and the secret random value s at position zero. SCRAPE then defines the commit of a participant to consist of three parts, encrypted shares, commitments, and NIZK proofs.

1. Encrypted shares $\{\hat{s}_1, \ldots, \hat{s}_n\}$ exists to make the recovery of a secret possible, in case a party does not reveal a secret themselves. Furthermore, the shares are encrypted with the public keys of the participants $\hat{s}_i = pk_i^{p(i)}$, so that one party alone could not figure out the secret behind them. Instead, at least $f+1$ parties must work together to recover the secret.

2. Commitments $\{v_1, \ldots, v_n\}$ with $v_i = g^{p(i)}$ exist to allow figuring out the degree of the secret sharing polynomial $p(x)$. The degree of the polynomial $p(x)$ should not be higher than f, otherwise it would not be $(f+1)$-of-n secret sharing.

3. NIZK proofs $\{e, z_1, \ldots, z_n\}$ ensure that the commitments and the encrypted shares are created by the same polynomial $p(x)$. This is done with a NIZK proof of equality, which we described in Section 2.3. With e being the challenge created by the SHA-256 [22] hash function and $\{z_1, \ldots, z_n\}$ being the responses to the challenge.

Once the commit is shared and verified, a party has committed to their randomness. Furthermore, if an adversary decides not to reveal their commit, $f+1$ parties may work together to recover the secret instead. This property is what allows SCRAPE to have a guaranteed output. As long as a quorum of honest parties exists, commits can be opened. Lastly, a plain commit to the secret $Com(s, r)$ also gets published. This exists, so that one could simply open their secret instead of having to interpolate over all encrypted shares. What kind of commitment scheme to use is open to the reader. Since we have two generators, a Pedersen commitment scheme [26] would be possible. Here a party publishes $Com(s, r) := g^s h^r$, which has perfect hiding and computational binding. However, we could also use a perfect binding scheme in which one publishes $Com(s) := g^s$. This would be in line with all of our other commitments. Furthermore, g^s can be calculated by any public entity using Lagrange interpolation, since $\{g^{s_1}, \ldots, g^{s_n}\}$ are known. Hence, the plain commit is already included with the previous commit. The pseudocode form of how to produce a commit can be seen in Algorithm 3.

With the words commit, commitment, and plain commit being used inside the SCRAPE protocol and causing confusion, Table 3.1 differentiates the words.

Algorithm 3 Commit function

9: **Commit(** pks **)**
10: $s \leftarrow \mathbb{Z}_q$ //Pick a random polynomial of degree f with $p(0) = s$
11: **for** $i \in 1, \ldots, f$ **do**
12: $c_i \leftarrow \mathbb{Z}_q$
13: $p(x) := s + c_1 x^1 + \cdots + c_f x^f$

14: **for** $i \in 1, \ldots, n$ **do**
15: $v_i := g^{p(i)}; \hat{s}_i := pks[i]^{p(i)}$ //Create commitments v and encrypted shares $\hat{s}$
16: $w_i \leftarrow \mathbb{Z}_q$ //Create a NIZK proof using a random oracle
17: $\alpha_{1,i} := g^{w_i}; \alpha_{2,i} := pks[i]^{w_i}$
18: $e := H(v_1, \hat{s}_1, \ldots, v_n, \hat{s}_n, \alpha_{1,1}, \alpha_{2,1}, \ldots, \alpha_{1,n}, \alpha_{2,n})$
19: **for** $i \in 1, \ldots, n$ **do**
20: $z_i := w_i - p(i) \cdot e$
21: **return** $(\hat{s}_1, \ldots, \hat{s}_n, v_1, \ldots, v_n, e, z_1, \ldots, z_n)$

Type	Meaning
Commitment	Refers to the group elements $v_i = g^{p(i)}$ that allow for the degree of the polynomial $p()$ to be checked.
Commit	Refers to the vector that participants share during the commit phase. The vector consists of encrypted shares, commitments, and NIZK proofs and looks as follows, $(\hat{s}_1, \ldots, \hat{s}_n, v_1, \ldots, v_n, e, z_1, \ldots, z_n)$.
Plain commit	Refers to the value $Com(r, s)$ that gets posted onto the public ledger. It allows for $s = p(0)$ to be easily revealed, without the recovery having to take place.

Table 3.1: Different types of commits

3.1.1.3 Reveal phase

Once $f + 1$ parties have committed to their randomness and the commit vectors have been validated, the output of the random beacon protocol is determined, and the protocol moves on to the reveal phase. We wait for $f + 1$ commits, since we can then be sure that at least one commit comes from an honest party. Similarly to the two-party coin flip described in Figure 2.3, we can be sure that the output is truly random, as long as one of the committed secrets is truly random. Since the output is determined, the protocol may not accept any further commits once the reveal phase has begun. Furthermore, to get the output of the protocol, all commits must be opened.

The reveal phase consists of publishing the respective (s, r) values, which were used to produce the plain commit. Once received, parties must verify that $Com(s, r)$ is equal to the previously sent commit. This involves performing Lagrange interpolation on $\{v_1, \ldots, v_{f+1}\}$ to calculate g^s and ensuring that s was revealed correctly. Applying Lagrange interpolation on the commitments entails $O(n)$ exponentiations. It follows that with $O(n)$ reveals, each node performs $O(n^2)$ exponentiations, and the entire network performs $O(n^3)$ exponentiations.

If all commits get opened, the protocol finds itself in the best-case scenario of SCRAPE. We may skip the recovery phase and calculate the final randomness. However, if even a single commit is not revealed, the protocol moves on to the recovery phase.

3.1.1.4 Recovery phase

For every party that does not open their commit, the remaining participants must recover it on their behalf. This is possible, since $(f + 1)$-of-n secret sharing was used during the commit phase. Hence, a quorum can work together by decrypting their encrypted shares and reconstructing the secret. Although decrypting might not be the most fitting word, considering that a decrypted share looks as follows,

$$\tilde{s}_i := \hat{s}_i^{\frac{1}{sk_i}} = (pk_i^{p(i)})^{\frac{1}{sk_i}} = (h^{sk_i p(i)})^{\frac{1}{sk_i}} = h^{\frac{sk_i p(i)}{sk_i}} = h^{p(i)}.$$

Hence, decrypted shares $\tilde{s}_i$ are marked with a tilde, as they are the generator h raised to the power of the secret share $p(i)$. Furthermore, the recovery vector, which participants must post, is made up of two parts. The decrypted share $\tilde{s}_i$, and a NIZK proof of equality $\{e, z\}$, which proves that the share was decrypted correctly, by showing $\log_h(pk_i) = log_{\tilde{s}_i}(\hat{s}_i) = sk_i$. The pseudocode of the decryption process can be seen in Algorithm 4.

Algorithm 4 Decryption

22: **Decrypt(** sk_i**,** $(\hat{s}_1, \ldots, \hat{s}_n)$ **)**
23: $\tilde{s}_i := \hat{s}_i^{sk_i^{-1}}$ //Decrypt the share
24: $w \leftarrow \mathbb{Z}_q$ //Create a NIZK proof using a random oracle
25: $\alpha_1 := h^w; \alpha_2 := \tilde{s}_i^w$
26: $e := H(pk_i, \hat{s}_i, \alpha_1, \alpha_2)$
27: $z := w - e \cdot sk_i$
28: **return** $(\tilde{s}_i, e, z)$

Once $f + 1$ encrypted shares have been decrypted, we make use of an adapted version of Lagrange interpolation to reconstruct the original secret. The only difference being that the Lagrange coefficients are used in the exponent. Hence, $O(n)$ exponentiations have to be done. The pseudocode of the adapted version of Lagrange interpolation can be seen in Algorithm 5.

In SCRAPE, the recovery of a commit can happen at most f times, since we assume at least one commit to be from an honest party. However, SCRAPE also functions if all $f + 1$ commits have to be

Algorithm 5 Recovery

29: **Recover(** $\tilde{s}_{\alpha_1}, \ldots, \tilde{s}_{\alpha_{f+1}}$ **)**
30: $val := 1$
31: **for** $j \in \{1, \ldots, f+1\}$ **do**
32: $\lambda_j := \prod_{k \neq j} \frac{\alpha_k}{\alpha_k - \alpha_j}$ //Calculate Lagrange coefficients
33: $val := val \cdot \tilde{s}_{\alpha_j}^{\lambda_j}$
34: **return** val

recovered. Although, this would be the worst-case scenario of SCRAPE, since recovering a commit is more complex compared to revealing a commit, considering that a quorum has to work together, instead of a single party.

3.1.2 Randomness calculation

For simplicity reason, we define the secrets of the $f+1$ nodes, that participated in the commit phase, as $\{s_1, \ldots, s_{f+1}\}$. Assuming all secrets have been revealed, then the values $\{s_1, \ldots, s_{f+1}\}$ are known. To get a random value r' that follows from all other random values, one can calculate

$$r' := \sum_{i=1}^{f+1} s_i.$$

However, r' can only be calculated when all commits get revealed. If we assume commit $i \in \{1, \ldots, f+1\}$ does not get revealed and has to be recovered, then h^{s_i} is known instead of s_i. Since we assume the discrete logarithm problem to be hard, one cannot get s_i from h^{s_i} and one cannot calculate r' if a single s_i is unknown. To solve this issue, SCRAPE does not calculate r' but instead calculates

$$r := \prod_{i=1}^{f+1} h^{s_i},$$

which is equivalent to

$$r = \prod_{i=1}^{f+1} h^{s_i} = h^{\sum_{i=1}^{f+1} s_i} = h^{r'}.$$

And since r' is random, it follows that $h^{r'}$ is random as well. The only difference being that $r' \in \mathbb{Z}_q$, while $r \in \mathbb{G}$. However, the problem of differing groups can be solved with the help of a random oracle. Calculating $H(r')$ or $H(r)$ both result in a random value in $\{0,1\}^l$. This way, SCRAPE outputs the same random value no matter how many commits get recovered or revealed.

3.1.3 NIZK verification

As mentioned in the previous subsection, during two phases of SCRAPE, a NIZK proof is used. Once, during the commit phase and once during the recovery phase. In both instances, the purpose is similar. The first NIZK proof is used to prove that $\log_{pk_i}(\hat{s}_i) = \log_g(v_i) = p(i)$. The second NIZK proof is used to prove that $\log_h(pk_i) = log_{\tilde{s}_i}(\hat{s}_i) = sk_i$. These zero-knowledge proofs are known as proofs of equality, and they are explained in more detail in Section 2.3. The pseudocode form of the NIZK proof verification, can be found in Algorithm 6.

Algorithm 6 NIZK check

CheckCommitNIZK($\hat{s}_1, \ldots, \hat{s}_n, v_1, \ldots, v_n, e, z_1, \ldots, z_n$)
35: **for** $j \in 1, \ldots, n$ **do**
36: $\alpha_{1,j} := g^{z_j} v_j^e; \alpha_{2,j} := pk_j^{z_j} \hat{s}_j^e$
37: **return** $e = H(v_1, \hat{s}_1, \ldots, v_n, \hat{s}_n, \alpha_{1,1}, \alpha_{2,1}, \ldots, \alpha_{1,n}, \alpha_{2,n})$

CheckRevealNIZK($\tilde{s}_j, e, z$)
38: $\alpha_1 := h^z pk_j^e; \alpha_2 := \tilde{s}_j^z \hat{s}_j^e$
39: **return** $e = H(pk_j, \hat{s}_j, \alpha_1, \alpha_2)$

3.1.4 Degree verification

An important, if not the most important, contribution of SCRAPE was its degree verification algorithm, which can be run with a linear number of exponentiations. SCRAPE accomplished this by simplifying the problem of checking the degree of the underlying polynomial. Since given a set of commitments $\{g^{p(1)}, g^{p(2)}, \ldots, g^{p(n)}\}$, one must check that the underlying polynomial $p(x)$ has a degree of $\leq f$. Not doing so would leave an obvious attack possibility. An adversary could share a polynomial of degree $> f$ and then interpolating different points would lead to different results. To illustrate such an attack, we take the simplest example: $f = 1$, $n = 3$, $p(x) = x^2$, and an order q. This leads to the secret shares $\{(1, 1), (2, 4), (3, 9)\}$. Taking the points $(1, 1), (2, 4)$ and interpolating them, gives us the secret value $q - 2$. However, taking the points $(1, 1), (3, 9)$ leads to the value $q - 3$. This allows an adversary to choose which randomness they would prefer by withholding their share. Hence, it is important to ensure that $p(x)$ has a degree $\leq f$.

To check the degree of $\{g^{p(1)}, \ldots, g^{p(n)}\}$ let us first talk about checking the degree of $\{p(1), \ldots, p(n)\}$. We know that given $f + 1$ points, we are able to uniquely identify a polynomial of degree f. One option would be to Lagrange interpolate all possible sets of $f + 1$ points and ensure that they all result in the same value. However, this method would be inefficient, as it would require $O(n!)$ Lagrange interpolations. A more efficient method would be to reconstruct the polynomial $p(x)$ given $f + 1$ points and then evaluate $p(x)$ on the remaining $n-(f+1)$ points, ensuring that all values stem from the same polynomial of degree f. However, polynomial reconstruction requires $O(n^3)$ multiplications, making this method inefficient as well. SCRAPE solves this problem in a way that follows from results in error-correcting code theory [27]. Firstly, a verifier creates a second random polynomial p' with $deg(p') = n - f - 2$. Multiplying both polynomials together should result in a polynomial with $deg(p'p) = (n - f - 2) + (f) = n - 2$. For a polynomial of degree $n - 2$ it holds that the coefficient of x^{n-1} is equal to 0. Furthermore, from Lagrange interpolation it follows that $p(x)p'(x) = \sum_{i=1}^{n} (\prod_{j \neq i} \frac{x-j}{i-j}) p(i)p'(i)$. Lastly, equating the coefficients of x^{n-1} from both sides results in the following equation

$$0 = \sum_{i=1}^{n} \prod_{j \neq i} \frac{1}{i - j} p(i)p'(i).$$

SCRAPE uses this equation to verify that the degree of a polynomial is $\leq f$. While this method seems to require $O(n^2)$ multiplications, one can reduce this to $O(n)$ by precomputing the coefficients $\lambda_i := \prod_{j \neq i} \frac{1}{i-j}$, which are the same in each execution of the verification algorithm. Furthermore, since the shares are in the exponent of the generator, the verification requires a linear number of exponentiations and looks as follows,

$$1 = g^0 = g^{\sum_{i=1}^{n} \lambda_i p(i)p'(i)} = \prod_{i=1}^{n} g^{p(i)} g^{\lambda_i p'(i)}.$$

The pseudocode for what has been described may be seen in Algorithm 7.

Algorithm 7 Degree verification

40: **CheckDegree**$(v_1, \ldots, v_n)$
41: **for** $i \in 0, \ldots, n-f-2$ **do**
42: $a_i \leftarrow \mathbb{Z}_q$
43: $p'(x) := a_0 + a_1 x^1 + \cdots + a_{n-f-2} x^{n-f-2}$ //Pick a random polynomial of degree $n-f-2$
44: **for** $i \in 1, \ldots, n$ **do** //Verify, the commitments are of degree f
45: $\lambda_i := \prod_{j \neq i}^{n} \frac{1}{i-j}$
46: $c_i^{\perp} := \lambda_i p'(i)$
47: **return** $1 = \prod_{i=1}^{n} v_i^{c_i^{\perp}}$

3.2 Changes to SCRAPE

While analyzing SCRAPE we decided to expand upon the existing protocol, as many other papers do [4], [12]. Doing so allowed us to improve upon SCRAPE and make it more simple, efficient, and asynchronous.

3.2.1 Adding shares

The largest change we made to SCRAPE, was to add all encrypted shares together once the underlying commits were validated. Doing so allowed us to exploit an interesting property of polynomial secret sharing. Interpolating all secret shares and then adding all secrets together, is the same as adding all secret shares together and then interpolating the added secret. We show this via a direct proof.

Given a quorum $\mathbb{Q}$ of size $f+1$, it holds that

$$\begin{aligned} \forall s_1, s_2 \in \mathbb{Z}_q : s_1 + s_2 &= \prod_{i \in \mathbb{Q}} \lambda_i p_1(i) + \prod_{i \in \mathbb{Q}} \lambda_i p_2(i) \\ &= \prod_{i \in \mathbb{Q}} \lambda_i p_1(i) + \lambda_i p_2(i) \\ &= \prod_{i \in \mathbb{Q}} \lambda_i (p_1(i) + p_2(i)). \end{aligned}$$

This is essential to our changes in SCRAPE, since originally we would have interpolated all secrets from the shares and then added the secrets together. With $f+1$ secrets, this implies doing Lagrange Interpolation $f+1$ times in the worst-case. To avoid this worst-case scenario, we instead add all shares together and interpolate only once. Hence, even if all commits do not get revealed, we only need to recover a single secret.

We begin our protocol with an array containing n neutral elements $C := [1]^n$. Once a valid commit arrives, we add the encrypted shares to our array by calculating $C[i] \cdot = \hat{s}_i$ for all $i \in \{1, \ldots, n\}$. Once $f+1$ valid commits have arrived, we are left with the following array

$$C = [pk_1^{p_1(1)+\cdots+p_{f+1}(1)}, \ldots, pk_n^{p_1(n)+\cdots+p_{f+1}(n)}].$$

Decrypting $f+1$ shares and interpolating them at position zero leaves us with $r := h^{p_1(0)+\cdots+p_{f+1}(0)}$. This is the same result as in the original SCRAPE protocol.

Since we add all encrypted shares together, we have to do $O(n^2)$ multiplications. However, we only have to do a single Lagrange interpolation, which requires $O(n)$ exponentiations. Meanwhile, in SCRAPE, we need to do $f + 1$ Lagrange interpolations, which amounts to $O(n^2)$ exponentiations per node. Afterward, SCRAPE adds the secrets together, causing $O(n)$ multiplications. Since exponentiations are more expensive compared to multiplications, we expect adding the shares together to be more efficient. Furthermore, only $O(n)$ decrypted shares have to be posted instead of $O(n^2)$, which adds more reasons as to why our protocol should perform better.

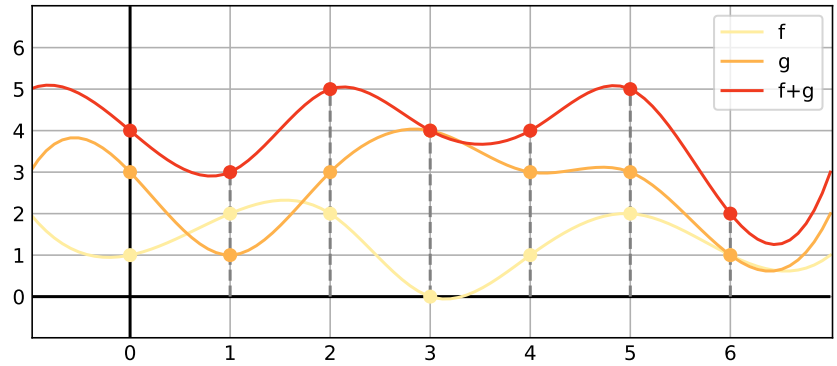


Figure 3.1: Illustration of adding shares

In Figure 3.1, we illustrate a simple example of adding two polynomials together. The beige polynomial f and the orange polynomial g get added together and result in the red polynomial. We see that adding the functions together trivially causes the secrets at $x = 0$ and all secret shares to be added. However, as shown previously, the statement also works in the opposite direction. Adding all secret shares together causes the resulting polynomial to be the addition of f and g.

3.2.2 Asynchronous SCRAPE

SCRAPE is almost asynchronous. However, two aspects make it synchronous. Firstly, in the setup phase, we wait for n public keys. The idea being that after a certain amount of time, we stop waiting for public keys, and that defines the n of the protocol. Doing this is only possible in a synchronous network, otherwise, we would not know when to stop waiting for public keys. This problem can be solved by assuming that $n = 3f + 1$ and waiting for $2f + 1$ public keys instead. There are at most f dishonest nodes, it follows that $2f + 1$ public keys will always arrive at some point.

The other problem is located in the reveal phase. There, we wait for all plain commits to be opened. In an asynchronous network, this could lead to waiting forever if a party does not reveal their commit. To avoid this, we must have an upper bound on message delays. However, having an upper bound on message delays is equivalent to being in a synchronous network. This issue can be avoided altogether by skipping the reveal phase. Instead, we move straight to the recovery phase, once $f + 1$ valid commits have arrived. These changes would make SCRAPE asynchronous. However, SCRAPE does not do this, since it would result in the worst-case scenario, where all $f + 1$ commits have to be recovered. Fortunately, in Section 3.2.1 we simplified the recovery process. Instead of having to recover $f + 1$ commits, we only have to recover a single added commit. Hence, once $f + 1$ commits have been validated, we move on to the recovery phase and wait for any set of $f + 1$ shares to be decrypted, and then we interpolate the added secrets using Lagrange interpolation.

In summary, the idea behind asynchronous SCRAPE is to not wait for every party to open their plain commit, but instead recover the secret by default. This allows us to skip the reveal phase entirely.

Furthermore, without a reveal phase, we can also skip posting plain commits, since they would not be used. What remains is the modified setup, commit, and recovery phase. All of which can be run asynchronously. Adding all of our changes together, we are left with the following pseudocode that can be seen in Algorithm 8.

Algorithm 8 Asynchronous SCRAPE (process p_i).

48: **state**
49: $sk_i := 1$ //Secret key
50: $pks := [\perp]^n$ //Array of Public keys
51: $Commits := [\perp]^n$ //Array of Commits
52: $Recoveries := [\perp]^n$ //Array of Recovery shares
53: $AddedCommit := [1]^{2f+1}$ //Array with $2f + 1$ many neutral elements

54: **upon event** INIT **do**
55: $sk_i \leftarrow \mathbb{Z}_q$
56: **invoke** Post(h^{sk})

57: **upon** Receive(pk_j) from p_j **such that** $|pks \neq \perp| \leq 2f \wedge pks[j] = \perp$ **do**
58: $pks[j] := pk_j$

59: **upon** $|pks \neq \perp| > 2f \wedge pks[i] \neq \perp$ **do** //$2f + 1$ keys, start commit phase
60: $C_i :=$ Commit(pks)
61: **invoke** Post(C_i)

62: **upon** Receive(C_j) from p_j **such that** $|Commits \neq \perp| \leq f \wedge Commits[j] = \perp$ **do**
63: **if** C_j **is valid then**
64: $AddedCommit.add(C_j)$
65: $Commits[j] := C_j$

66: **upon** $|Commits \neq \perp| > f \wedge pks[i] \neq \perp$ **do** //$f + 1$ commits, start recovery phase
67: $R_i :=$ Decrypt(sk_i, $AddedCommit$)
68: **invoke** Post(R_i)

69: **upon** Receive(R_j) from p_j **such that** $|Recoveries \neq \perp| \leq f \wedge Recoveries[j] = \perp$ **do**
70: **if** R_j **is valid then**
71: $Recoveries[j] := R_j$

72: **upon** $|Recoveries \neq \perp| > f$ **do** //$f + 1$ recovery shares, calculate final randomness
73: **return** Recover($Recoveries$)

4 Implementation

In this chapter, we discuss the implementation of our random beacon protocol.

4.1 Thetacrypt

For the implementation of our protocol, we chose to use Rust, which is a general-purpose programming language that enforces memory safety [28]. Furthermore, we wrote the code in Rust using the Tokio library [29] which provides an asynchronous runtime for Rust. Although it is possible to write protocols without Tokio and with standard runtimes, we found it better to make every aspect of our implementation asynchronous.

To simplify the implementation process, we developed our protocol on top of Thetacrypt [30], which is a threshold cryptography library written in Rust. Thetacrypt was developed by the Cryptology and Data Security research group at the University of Bern and allowed us to implement our cryptographic protocol without having to worry about networking, handling datatypes, finite field operations, and many other aspects that come with writing cryptographic protocols.

Figure 4.1 shows an illustration of the architecture of Thetacrypt taken from its GitHub page [31]. We can see that the architecture of Thetacrypt is made up of three distinct layers.

1. Firstly, the service layer, which consists of the RPC handler. The RPC handler is responsible for handling the API and managing other requests from users.

2. The core layer implements the logic, the primitives, and the orchestration code that is needed to run the different cryptographic protocols that exist within Thetacrypt.

3. The network layer implements the modules to exchange peer-to-peer messages between the participating parties. It also offers an interface to post messages onto a public ledger.

As mentioned, we did not have to make changes to every layer. Inside the service layer, we made no meaningful changes, since we made use of the API without adapting it. Furthermore, we also made no meaningful changes in the network layer, since an interface to post messages onto a public ledger was already provided, and that was all that SCRAPE needed communication-wise. The only changes we made

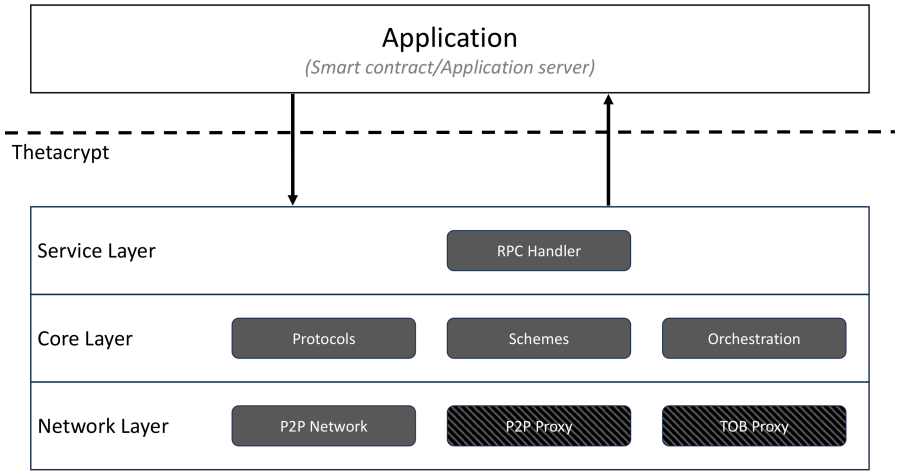


Figure 4.1: Illustration of the architecture of Thetacrypt [31]

were inside the core layer. Inside the core layer, we adapted the protocols module and the schemes module. Inside the protocols module, we wrote high-level code that handled incoming and outgoing messages, and determined which protocol phase one found themselves in. The code which we added to the protocols module, was very similar to the pseudocode seen in Algorithm 8. However, it still took 544 lines of code to transform the pseudocode into its real world equivalent. Furthermore, writing only high-level code did not suffice for our implementation. For the low-level code that handled the underlying cryptographic functions, we had to modify the schemes module. The code inside the schemes module was similar to the remaining pseudocode and took up 612 lines of code. The protocols module would then communicate with the schemes module through an interface, which we also expanded.

5 Evaluation

In this chapter, we firstly discuss the methodology used for testing and benchmarking our implementation. Afterward, we look at the results that follow from our simulations and research. Lastly, we observe the output of a statistical test suite which we ran on our random beacon protocol outputs.

5.1 Docker

To simulate nodes in our distributed randomness network, we made use of the Docker engine [32], which is a useful tool for OS-level virtualization. However, we shall talk about the other options we considered as well.

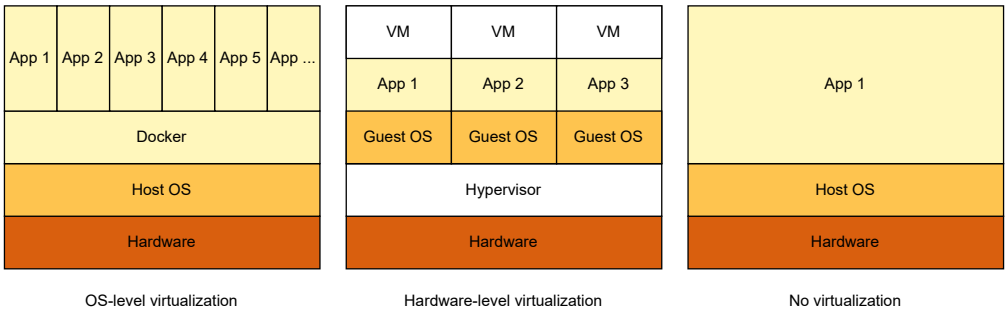


Figure 5.1: Illustrating example of virtualization

In Figure 5.1, one can find the three options we considered for running our protocol on top of hardware. The most intuitive option would be to use no virtualization. This can be seen on the right-hand side of Figure 5.1. No virtualization implies that each Thetacrypt instance would run on its own set of hardware. Hence, running a large-scale simulation would require numerous devices. The benefit of this method would be that it would provide the most authentic simulation, since a distributed randomness network can have multiple nodes located anywhere in the world. A way to deploy nodes across the world would be

to use cloud computing services such as AWS. We could use their elastic compute cloud [33] to borrow multiple machines in different locations around the globe. Furthermore, using no virtualization allows us to run as close to bare-metal performance as possible. However, the distributed randomness protocol is not nearly computationally heavy enough to warrant the use of a separate machine for each node. Furthermore, using services such as AWS can become costly, which we wanted to avoid.

The second option would be to use hardware-level virtualization. This can be seen in the middle of Figure 5.1. Virtualizing the hardware allows us to run multiple virtual machines on a single device. Each VM would then run its own instance of Thetacrypt and represent a node. However, here we run into the opposite problem. Running a VM incurs a large overhead, and after a certain number of VMs, the device would be overloaded. This would lead to needing multiple devices and, in turn, the same issues as with no virtualization.

The third option is OS-level virtualization, which can be seen on the left-hand side of Figure 5.1. This type of virtualization can be best described by a quote, from David Wheeler: "All problems in computer science can be solved by another level of indirection." Docker embodies this quote by putting software into containers and then running the containers on top of software. Doing so incurs a smaller overhead compared to hardware-level virtualization. The containers only load the minimum number of packages that are necessary for the software to run, instead of loading a fully fledged operating system as hardware-level virtualization does. For our work, we used Docker to containerize Thetacrypt and run multiple instances on top of a single device. This way, we could simulate an entire distributed randomness network on a single device. The only thing that should differ in a real-world setting would be the latency of the nodes and the variation in computational power between nodes.

5.2 Methodology

To test our protocol in a fair and reproducible manner, we required a reproducible setup. Hence, we decided on the following experimental setup. In each simulation, we firstly created n docker containers, which represented the nodes participating in the protocol. Each of the n docker containers ran an instance of Thetacrypt. To make communication between these nodes possible, we created a further container that acted as a blockchain stub. The only job of the blockchain stub was to receive messages from nodes and broadcast them back in total order. Lastly, we created a client that started the protocol and specified the number of iterations the protocol should run for. With the first iteration resulting in a single random value and containing the setup phase, the commit phase, and the recovery phase. Later iterations functioned the same, except they skipped the setup phase, as the public keys were known. Once the number of iterations was reached, the protocol terminated, and the client received a JSON with a log of the entire protocol and the amount of time that was spent in each iteration. The entire simulation was run locally with no message delays on an 8-core AMD machine with 16 GB of RAM. However, the specifics did not matter too much, as we simply needed a device that could run our simulations and had constant computational power, to standardize our tests and allow for the comparison of the results.

Inside each simulation, we recorded three metrics. The average runtime of the protocol, the number of messages that got posted to the blockchain, and the communication cost of the protocol. Measuring the runtime of a protocol acted as a substitute for the computational complexity of said protocol, since there were no message delays, the computational complexity should have been the only factor influencing runtime. Furthermore, we averaged the runtimes to make the data accurate. We also measured the number of messages posted to the blockchain, since each message to the blockchain incurred an overhead. Lastly, we measured the communication cost by measuring how many bytes the protocol required to function. The higher the communication cost, the more costly it would be to post the protocol onto a blockchain. Hence, it is important for a random beacon protocol to try to minimize the communication cost.

5.3 Findings

In the following section, we present the findings of our research where we implemented our adapted version of SCRAPE, which we will from here on refer to as async. SCRAPE, and the worst-case scenario of SCRAPE. We then ran experiments according to the methodology presented in Section 5.2 for both protocols and compared the results to each other. This was done to see whether our changes to SCRAPE improved the protocol in regard to the measured metrics. Furthermore, the simulations showed us the real-world cost of distributed randomness. We only compared async. SCRAPE with the worst-case scenario of SCRAPE, since the other scenarios could not be transformed into an asynchronous setting. However, async. SCRAPE is asynchronous by design, and the worst-case scenario of SCRAPE could be made asynchronous by waiting for $2f + 1$ instead of n keys in the setup phase, skipping the reveal phase, and recovering every commit instead. Recovering all commits is equivalent to the worst-case scenario of SCRAPE, since in the best-case scenario, all commits would be revealed and no recovery would have to be done.

5.3.1 Average runtime

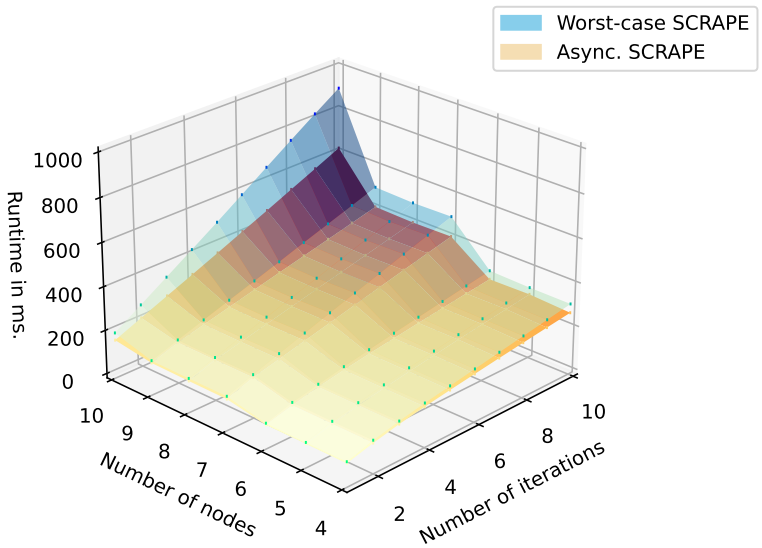


Figure 5.2: Runtime measurements with up to 10 nodes and 10 iterations

Figure 5.2 contains the results of a small-scale simulation, where we recorded the average runtime of the worst-case scenario of SCRAPE and the average runtime of async. SCRAPE. The small-scale simulation ran with up to 10 nodes and 10 iterations, as to highlight some significant properties of both protocols. The averages were calculated by running the simulation 100 times and then taking the average time from each iteration. The average runtime of async. SCRAPE can be seen on the orange-colored surface, while the exact values can be found in Table 6.1. The average runtime of the worst-case scenario of SCRAPE can be seen on the blue surface, and the exact values can be found in Table 6.2. Furthermore,

the x-axis represents the number of nodes that were used to run the protocol. This number ranges from 4 to 10, with 4 being the smallest number of nodes we could simulate, given $f = 1$ and $n > 3f = 3$, and 10 being the maximum number of nodes since we only wanted to run a small-scale simulation here. The y-axis represents the number of iterations the protocol was run for. This ranges from 1 to 10, since we told the protocol to stop after producing 10 random values. Lastly, the z-axis represents the average runtime of the protocol in milliseconds. This value ranges from 0 ms. to 1000 ms, with 1000 ms. being equivalent to 1 second. Hence, a lower value on the z-axis indicates a smaller runtime and is beneficial for a protocol.

Firstly, we observe that in this small-scale setting, the runtimes between both protocols do not differ by much. The biggest difference can be seen at the maximum simulation size. With 10 nodes and after 10 iterations, the average runtime of the worst-case scenario of SCRAPE lies at 897.3 ms, while the average runtime of async. SCRAPE lies at 615.8 ms. If we turn this into random values per second, we get that with 10 nodes async. SCRAPE produced 16.2 random values a second, while the worst-case scenario of SCRAPE produced 11.1 random values a second. It follows that async. SCRAPE was 45.7% faster. The smallest difference can be seen at the smallest simulation size. With 4 nodes and after 1 iteration, async. SCRAPE took 85.4 ms, while the worst-case scenario of SCRAPE took 92.8 ms. It follows that async. SCRAPE was only 8.7% faster than the worst-case scenario of SCRAPE. This stems from the fact that the first iteration, in both protocols, contains the setup phase, and it takes a while for the nodes to initialize and do further precomputation. It is important to note that in every simulation, async. SCRAPE outperformed the worst-case scenario in terms of runtime. We also see that the runtimes depended heavily on the number of tolerated adversaries f. With 4, 5, and 6 nodes, the average runtimes were practically equivalent. This comes from $n \in \{4, 5, 6\}$ only tolerating a single adversary, $f = 1$. We see the same trend with 7, 8, and 9 nodes as they could tolerate up to 2 adversaries, $f = 2$. 10 nodes was once again different since 10 nodes could tolerate up to 3 adversaries, $f = 3$. Another aspect we observe is that the runtime increased linearly with every iteration. Lastly, we see that the runtime increased polynomially when we increased the number of nodes. However, the experimental setup is still too small to make definitive conclusions.

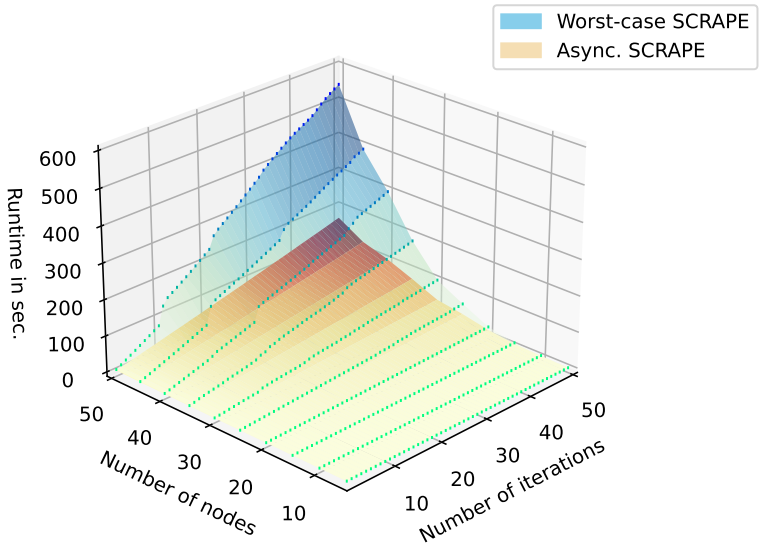


Figure 5.3: Runtime measurements with 50 nodes and 50 iterations

Figure 5.3 shows the results of a larger-scale simulation compared to the small-scale simulation seen in Figure 5.2. We recorded the average runtimes of both async. SCRAPE and the worst-case scenario of SCRAPE. However, here we ran the experiments for up to 50 nodes and 50 iterations. The averages were calculated by running the simulation 10 times and taking the average runtimes. The runtime of async. SCRAPE can be seen on the orange-colored surface, and the exact data for async. SCRAPE can be found in Table 6.3. The runtime of the worst-case scenario of SCRAPE can be seen on the blue-colored surface, while the exact data for the worst-case scenario of SCRAPE can be found in Table 6.4. The x-axis of Figure 5.3 represents the number of nodes that were used to run the protocol. This number ranges from 5 to 50 since we started with 5 nodes and increased the number by 5 until we reached 50 nodes. The y-axis represents the number of iterations the protocol ran for. This ranges from 1 to 50 iterations, since after 50 iterations, the protocol stopped. In other words, after producing 50 random values, the protocol stopped. Lastly, the z-axis represents the average runtime in seconds. The z-axis ranges from 0 to 600 seconds, with 600 seconds being equivalent to 10 minutes.

Similarly to the smaller scale simulation, we can see that our protocol was strictly faster than the worst-case scenario of SCRAPE. Furthermore, we see the difference more clearly now. The largest difference is observed at the maximum simulation size. With 50 nodes and after 50 iterations, the average runtime of async. SCRAPE lies at 164.2 seconds, while the worst-case scenario of SCRAPE lies at 544.7 seconds. It holds that async. SCRAPE produced 18.3 random values per minute, while the worst-case scenario of SCRAPE produced 5.5 random values per minute. It then follows that async. SCRAPE was 231.7% faster than the worst-case scenario of SCRAPE. We continue to see that the runtime increased linearly when raising the number of iterations. However, there were some exceptions in the worst-case scenario of SCRAPE. With 40, 45, and 50 nodes, the runtime increased drastically in multiple locations. This can be attributed to the system being overloaded when running too many docker containers. Furthermore, we see that increasing the number of nodes causes the runtime to grow polynomially.

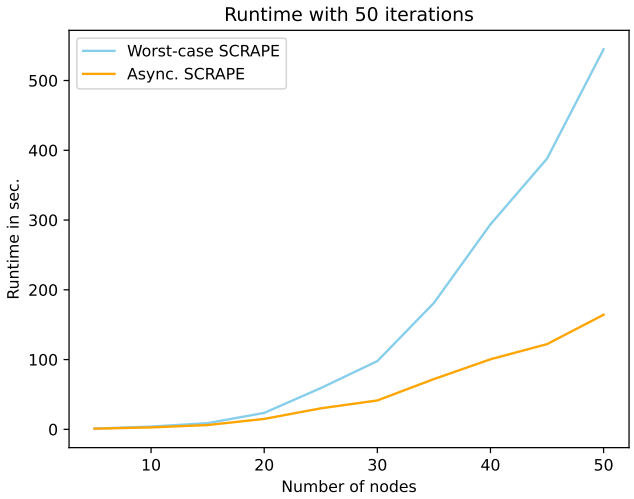


Figure 5.4: Runtime measurements with 50 iterations

In Figure 5.4, we take a closer look at how the increase of nodes influenced the average runtime of both protocols. We do so by only looking at the runtime after 50 iterations, where the runtimes are the highest, and leaving the number of nodes to be variable. This is the same as looking at the xz-plane at 50 iterations in Figure 5.3. Hence, the x-axis remains the number of nodes and the y-axis becomes the

runtime in seconds. Similarly, the average runtime of async. SCRAPE can be seen on the orange line, while that of the worst-case scenario of SCRAPE can be seen on the blue line. As before, we see that the worst-case scenario of SCRAPE and async. SCRAPE both scaled polynomially when increasing the network size. However, this graph emphasizes more clearly, that the exponent of async. SCRAPE was lower than the exponent of the worst-case scenario of SCRAPE.

5.3.2 Number of messages

In the following section, we compare the number of messages that got posted to the blockchain in async. SCRAPE with the number of messages that got posted in the worst-case scenario of SCRAPE. Since each message that gets posted to the blockchain incurs an overhead, this metric is important to measure and aim to reduce.

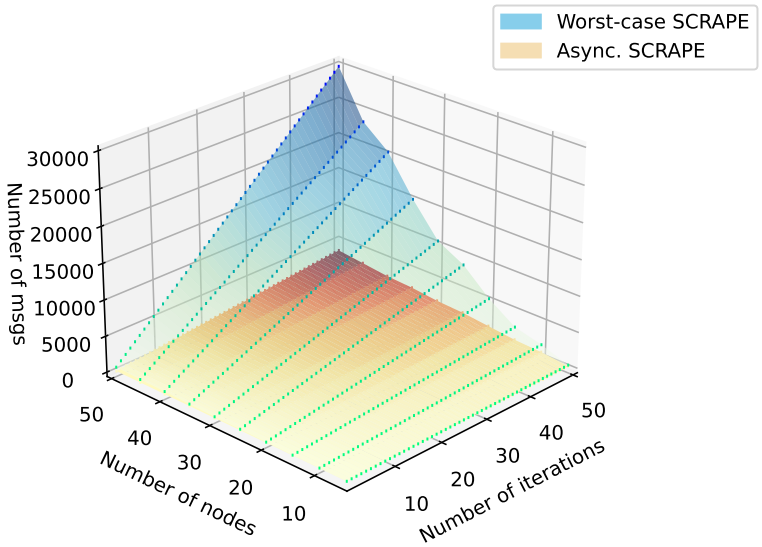


Figure 5.5: Message number with 50 iterations and 50 nodes

Figure 5.5 shows the number of messages that got posted to the blockchain in async. SCRAPE and the number of messages that got posted to the blockchain in the worst-case scenario of SCRAPE. We ran the experiments for up to 50 nodes and 50 iterations. We did not take averages, since the number of messages that got posted to the blockchain did not differ between simulations with the same parameters. The number of messages that async. SCRAPE posted can be seen on the orange-colored surface, while the exact data can be found in Table 6.5. The number of messages that the worst-case scenario of SCRAPE posted can be seen on the blue-colored surface, while the exact data can be found in Table 6.6. The graph seen in Figure 5.5 has the same dimensions, as the average runtime graph seen in Figure 5.3, with the only difference being the z-axis. Here, the z-axis represents the number of messages that got posted to the blockchain for a given simulation. This number ranges from 0 to 30'000, since with the largest simulation size, the worst-case scenario of SCRAPE posted slightly less than 30'000 messages.

Our experimental data shows that async. SCRAPE always outperformed the worst-case scenario of

SCRAPE by a wide margin. With the largest difference being at the maximum simulation size. With 50 nodes and after 50 iterations, the worst-case scenario of SCRAPE posted 29'783 messages to the blockchain, while async. SCRAPE posted 3'463 messages. Hence, async. SCRAPE posted only 11.6% as many messages as the worst-case scenario of SCRAPE did. The smallest difference can be seen at the smallest simulation size. With 5 nodes and after a single iteration, async. SCRAPE posted 14 messages, and the worst-case scenario of SCRAPE posted 17 messages to the blockchain. Hence, async. SCRAPE posted 82.4% as many messages as the worst-case scenario of SCRAPE did. We also see that the message number grew linearly when we increased the number of iterations. Furthermore, increasing the number of nodes caused the number of messages to grow quadratically in the worst-case scenario of SCRAPE, while increasing the number of nodes caused the number of messages in async. SCRAPE to grow linearly. This stems from the fact that the worst-case scenario of SCRAPE must post $(f+1)^2$ recovery messages, while async. SCRAPE only has to post $f+1$ many.

5.3.3 Communication cost

In the following section, we compare the communication cost of both protocols to each other. Doing so allows us to see how much space we should expect our random beacon protocols to take up on a blockchain. Since memory is valuable in public ledgers, it is important to try to minimize the communication cost size.

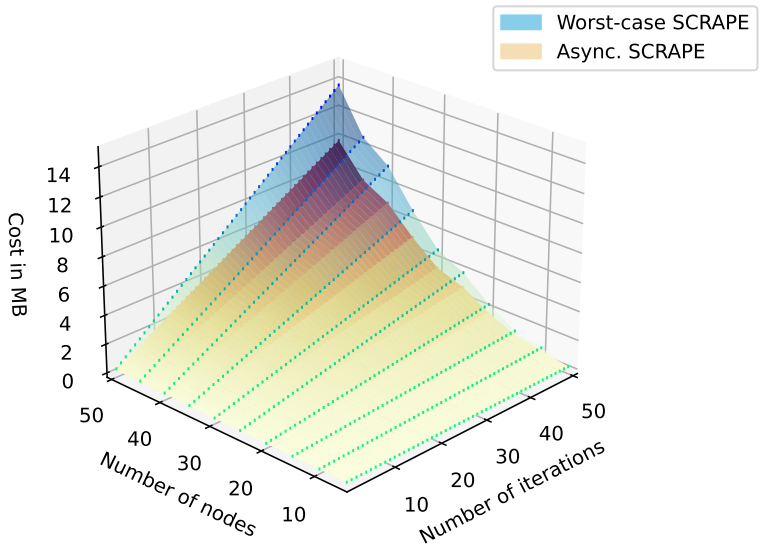


Figure 5.6: Communication cost with 50 iterations and 50 nodes

Figure 5.6 presents the number of bytes required by the async. SCRAPE protocol and the number of bytes required by the worst-case scenario of SCRAPE. We ran the experiments for up to 50 nodes and 50 iterations. Furthermore, we did not take averages, since the communication cost did not differ between different executions with the same parameters. The communication cost of async. SCRAPE can be seen on the orange-colored surface, and the exact data can be found in Table 6.7. The communication cost of the worst-case scenario of SCRAPE can be seen on the blue-colored surface, and the exact data can be found

in Table 6.8. The graph seen in Figure 5.6 has the same dimensions, as the average runtime graph seen in Figure 5.3, with the only difference being the z-axis. Here, the z-axis represents the amount of bytes needed for the protocol to function and be verifiably valid. This size is measured in megabytes, i.e., 10^6 bytes. The size ranges from 0 MB to 14 MB, since the maximum communication cost of all simulations was slightly less than 14 MB.

Firstly, we observe that the communication cost between async. SCRAPE and the worst-case scenario of SCRAPE did not differ by much. Although it is important to note that in all experiments, async. SCRAPE took up less space compared to the worst-case scenario of SCRAPE. The biggest difference can be seen at the maximum simulation size of 50 nodes and after 50 iterations. There async. SCRAPE took up 9.7 MB, while the worst-case scenario of SCRAPE took up 13.6 MB. It follows that async. SCRAPE required only 71.3% as much space as the worst-case scenario of SCRAPE did. The smallest difference can be seen at the smallest simulation size of 5 nodes and after a single iteration. There, we see that async. SCRAPE took up 3328 bytes, while the worst-case scenario of SCRAPE took up 3904 bytes. Hence, async. SCRAPE took up 85.2% as many bytes. Furthermore, we see that the communication cost grew linearly with the number of iterations and quadratically with the number of nodes for both protocols.

5.4 Results

The following sections summarizes the results of our findings that stem from implementing async. SCRAPE, the worst-case scenario of SCRAPE, and further research.

5.4.1 Average runtime

Firstly, when looking at the small scale simulation seen in Figure 5.2, we notice that the runtime of both protocols depends heavily on f. This makes sense if we highlight where throughout the protocols f is important. In the setup phase, both protocols wait for $2f + 1$ keys to be shared. Afterward, in the commit phase, each commit is made up of $2f + 1$ encrypted shares, commitments, and NIZK proofs. Before the recovery phase can start, each protocol waits for $f + 1$ commits to arrive. Then async. SCRAPE waits for $f + 1$ recovery shares, while the worst-case scenario of SCRAPE waits for $(f + 1)^2$ shares. All of these factors add together and explain why the runtimes depend so heavily on f. The only factor that depends directly on n is the responsiveness of the network and f itself, given that $n > 3f$. We see in the findings that with 6 nodes and after 10 iterations, async. SCRAPE was slightly faster than with 5 nodes and 10 iterations. This happens, since from 5 to 6 nodes f remains the same, however, the responsiveness of the network increases. In a non-local simulation, this difference would be more prominent.

Moving on to the larger scale simulation seen in Figure 5.3. We notice that the runtime increases polynomially when the number of nodes n increases. This is highlighted in Figure 5.4. While the runtime increase undoubtedly seems polynomial, the value in the exponent of the runtime is less clear. However, we do see that async. SCRAPE seems to have a lower exponent compared to the worst-case scenario of SCRAPE. This is to be expected since, in terms of exponentiations, a network running async. SCRAPE requires $O(n^2)$, while one running the worst-case scenario of SCRAPE requires $O(n^3)$. Furthermore, the number of messages sent by async. SCRAPE is $O(n)$, while the worst-case scenario of SCRAPE sends $O(n^2)$. Considering that each message is equivalent to a total order broadcast, it follows that async. SCRAPE has a communication complexity of $O(n^3)$ and the worst-case scenario of SCRAPE has one of $O(n^4)$. Since async. SCRAPE requires fewer exponentiations and has a lower communication complexity, it follows that the runtime should also have a lower exponent. One between $O(n^2)$ and $O(n^3)$, while the runtime of the worst-case scenario of SCRAPE should be between $O(n^3)$ and $O(n^4)$. Furthermore, Figure 5.4 support the claim that async. SCRAPE has an exponent of $O(n^2)$ while the worst-case scenario of SCRAPE has one of $O(n^3)$.

5.4.2 Number of messages

In the message graph, which can be found in Figure 5.5, we see the largest difference between both protocols. However, these differences did not translate into runtime or communication cost differences as directly as we expected. To understand why, we must first understand what kind of messages async. SCRAPE saves on. The worst-case scenario of SCRAPE sends more messages compared to async. SCRAPE, since in the recovery phase it has to send $(f+1)^2$ messages, while async. SCRAPE only has to send $f+1$ messages. Hence, the messages that async. SCRAPES saves on are recovery messages. These are made up of three values $(\tilde{s}, e, z)$, with each value being 256 bits long. Hence, a recovery message only takes up 768 bits, or 96 bytes. Considering that at the maximum simulation size, the worst-case scenario of SCRAPE posted 26'320 more messages, this equates to only a couple of megabytes, which is what we saw in the communication cost graph seen in Figure 5.6. The majority of the size is still taken up by the commits. Even removing all recovery shares would not change the communication cost too drastically.

In terms of computational complexity, a recovery message only contains a single NIZK proof that must be verified. This requires a constant number, $O(1)$, of computations, hence receiving a recovery message should not influence the runtime by much. However, receiving $f+1$ recovery shares causes the protocol to do a Lagrange interpolation. Hence, the worst-case scenario of SCRAPE requires f more Lagrange interpolations. This does influence the computational complexity of the protocol. Hence, we see a larger difference in the average runtime compared to the communication cost.

5.4.3 Communication cost

In the communication cost graph, which can be found in Figure 5.6, we see the smallest differences between both protocols. As mentioned, this comes from the fact that the commits are unchanged in both protocols, and the commits take up $O(n^2)$ bytes. Hence, without removing these commits or making them smaller, we cannot get below a communication cost of $O(n^2)$.

5.4.4 Summary

To summarize and explain our findings, we document them in Table 5.1. Furthermore, the best-case scenario data of SCRAPE stems from a paper written by Cascudo and David named "ALBATROSS: Publicly AttestabLe BATched Randomness Based On Secret Sharing" [4].

Protocol	Setup	Commit	Reveal	Recovery	Total exponentiations
Worst-case SCRAPE	$O(n)$	$O(n^2)$	-	$O(n^3)$	$O(n^3)$
Best-case SCRAPE	$O(n)$	$O(n^2)$	$O(n^3)$	-	$O(n^3)$
Async. SCRAPE	$O(n)$	$O(n^2)$	-	$O(n^2)$	$O(n^2)$

Table 5.1: Exponentiations per network

Table 5.1 contains the number of exponentiations required in the entire network, for each phase of SCRAPE. With exponentiations being the most computationally heavy operation in cyclic groups, it is worth comparing these values. The first column contains the protocol names. We compare async. SCRAPE, with the worst and best-case scenarios of SCRAPE. The second column describes the number of exponentiations needed in the setup phase. This is always $O(n)$, since n parties have to create a public key h^{sk}. The third column describes the commit phase and states that all protocols require $O(n^2)$ exponentiations in the commit phase. This follows from the fact that $f+1$ commits need to be made, with each commit requiring a linear number of exponentiations. Afterward, the reveal phase only pertains to the best-case scenario of SCRAPE since in both async. SCRAPE and the worst-case scenario of SCRAPE no reveals happen. The reveal phase requires $O(n^3)$ exponentiations, since each node has to do a

Lagrange interpolation for each reveal, as to verify it. It follows that $O(n)$ nodes have to do $O(n)$ Lagrange interpolations, with each interpolation needing $O(n)$ exponentiations. Hence, the complexity is $O(n^3)$. The recovery phase only applies to async. SCRAPE and the worst-case scenario of SCRAPE. Similarly to the reveal phase, in the worst-case scenario of SCRAPE $O(n)$ nodes need to perform $O(n)$ Lagrange interpolations. However, in async. SCRAPE $O(n)$ nodes have to perform a single Lagrange interpolation. Hence, async. SCRAPE has a complexity of $O(n^2)$, while the worst-case scenario of SCRAPE has one of $O(n^3)$. Lastly, the final column describes the total number of exponentiations required by each protocol. This value is calculated by summing everything together. It follows that both the worst-case and beast-case scenarios of SCRAPE require $O(n^3)$ exponentiations, while async. SCRAPE only requires $O(n^2)$.

Protocol	Number of messages	Communication cost	Network
Worst-case SCRAPE	$O(n^2)$	$O(n^2)$	Async.
Best-case SCRAPE	$O(n)$	$O(n^2)$	Sync.
Async. SCRAPE	$O(n)$	$O(n^2)$	Async.

Figure 5.7: Further comparisons between async. SCRAPE and SCRAPE

In Figure 5.7, one can find further comparisons that follow from our research and findings. We once again compare Async. SCRAPE, with the worst and best-case scenarios of SCRAPE. The second column contains the number of messages that need to be sent in each protocol. The best-case scenario of SCRAPE requires the same number of messages, as async. SCRAPE, which is $O(n)$. However, the worst-case scenario requires $O(n^2)$ messages, since it requires for $(f+1)^2$ recovery shares to be sent. Since each message is equivalent to a total order broadcast, the communication complexity follows by taking the number of messages and multiplying them by $O(n^2)$. Afterward, we see that the communication cost is $O(n^2)$ bytes in all three protocols. This holds since all protocols contain the same commits, which take up $O(n^2)$ bytes. However, since reveals are smaller than recovery shares, we can expect the best-case scenario of SCRAPE to be slightly smaller than async. SCRAPE, which is then again smaller than the worst-case scenario of SCRAPE in terms of communication cost. Lastly, the network assumption of the protocols can be found in the final column. As mentioned before, async. SCRAPE and the worst-case scenario of SCRAPE are asynchronous, while the best-case scenario of SCRAPE requires a synchronous setting.

To summarize, through our findings and research, we see that async. SCRAPE outperforms the worst-case scenario of SCRAPE. It follows that our changes had a positive effect in both theory and practice. Although we were not able to implement the best-case scenario of SCRAPE, we also expect our protocol to outperform the best-case scenario of SCRAPE in practice. We expect this, since the number of required exponentiations is lower in async. SCRAPE and the remaining parameters are equivalent.

5.5 Randomness test

Given a finite value, it is not possible to prove or disprove that the value was generated randomly. The only way to prove or disprove randomness is through mathematical analysis. Thankfully, the randomness of the SCRAPE random beacon protocol was mathematically proven in its paper [16]. Furthermore, our only major change to SCRAPE was to add commits together, which did not change the output of SCRAPE. This was shown in Section 3.2.1. Hence, the randomness of SCRAPE implies the randomness of our protocol.

Testing the randomness of our protocol would be a waste of resources, since it was already mathematically proven. However, testing the randomness of an implementation is different. A single bug could turn a random beacon protocol into a completely deterministic one. Hence, it makes sense to test the randomness of an implementation, as a precaution. For this, we made use of the NIST statistical test suite for random and pseudorandom number generators [34]. Created by Rukhin et al. for the U.S. Department

of Commerce, this test suite's purpose is to test whether values seem random or not. It does so by applying multiple mathematical tests that compare a bitstream to a bitstream created by random coin flips. To test our implementation, we firstly ran our protocol 10'000 times and saved the 256-bit outputs into a file. We then ran the test suite on said file.

C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	P-VALUE	PROPORTION	STATISTICAL TEST
8	11	9	10	14	10	9	9	8	12	0.955835	100/100	Frequency
12	12	18	6	7	8	12	5	12	8	0.129620	100/100	BlockFrequency
8	14	5	10	10	12	9	8	16	8	0.401199	100/100	CumulativeSums
7	14	10	8	7	9	10	11	12	12	0.851383	99/100	CumulativeSums
15	14	11	6	11	8	11	10	7	7	0.514124	100/100	Runs
13	13	7	9	10	10	15	12	5	6	0.366918	97/100	LongestRun
7	7	4	15	23	9	8	6	11	10	0.001399	100/100	Rank
19	6	9	9	6	14	9	12	8	8	0.108791	96/100	FFT
20	11	10	8	11	11	9	12	3	5	0.028817	99/100	ApproximateEntropy
19	4	19	8	12	9	5	10	9	5	0.002203	100/100	Serial
14	10	14	13	7	7	12	11	6	6	0.383827	100/100	Serial
14	4	9	12	9	11	10	10	10	11	0.739918	97/100	LinearComplexity

Table 5.2: Async. SCRAPE randomness report

Table 5.2 contains the findings from the NIST statistical test suite, which we ran on a file containing 10'000 values generated by our implementation of async. SCRAPE. Firstly, the test suite divided the file into 100 bitstreams, each bitstream containing 100 values. Each value was made up of 256 bits, hence each bitstream contained 25'600 bits, with NIST recommending more than 20'000 bits per bitstream. Afterward, the test suite was evaluated for each bitstream and the report was generated from these results. Inside the report the number of rows corresponds to the number of statistical tests applied. Furthermore, columns 1 to 10 correspond to the frequency of p-values distributed into 10 buckets arising from each statistical test. Column 11 contains a p-value that comes from applying a chi-square test on the previous 10 columns. If the resulting p-value is larger than 0.001, then the bitstreams differ from each other and are uniformly distributed. If the p-value is smaller than 0.001 then the bitstreams are too similar and the data is not random. Column 12 is the proportion of bitstreams that passed each test. As long as more than 95 out of 100 bitstreams passed the test, can the data that the bitstreams stem from be considered random. The 13th column contains the names of the corresponding statistical tests.

All the following tests took a bitstream and mathematically compared it to an ideal bitstream made up of random coin flips. Firstly, the frequency statistical test took each bitstream and checked whether the likelihood of a zero was the same as the likelihood of a one. 100 of the 100 bitstreams passed this test. Afterward, the block frequency test checked whether there were any local deviations in the frequency of ones and zeros by looking at smaller blocks instead of the entire bitstream. Here 100 out of 100 bitstreams passed as well. The cumulative sums test transformed zeros into minus ones and then checked what the highest sum of any subsection of a bitstream was. This was passed by 100 out of 100 bitstreams. Subsequently, the test was reversed, and the smallest cumulative sum was tested. Hence, the smallest subsection of the bitstream was checked. Here the first failure happened, however, the remaining 99 bitstreams passed. Hence, the statistical test was considered passing, since more than 95 bitstreams passed. The runs statistical test checked how long the longest sequence of identical bits was. Once again, 100 out of 100 bitstreams passed the test. Similarly to the block frequency test, the longest-run test looked at individual blocks of data instead of the entire bitstream. It then checked for the longest sequence of ones inside each block to spot local derivations. Here, 97 of 100 bitstreams passed. The rank test took

a bitstream and divided it into binary matrices. The binary matrices were then tested for their rank, to find linear dependencies inside the matrices. 100 of the 100 bitstreams passed. The FFT statistical test took each bitstream and performed a fast Fourier transform on it. The FFT transformed the bitstreams into Fourier series, which were then tested for emerging patterns. This test failed the most compared to all other tests. However, since 96 of the bitstreams passed, the statistical test was still considered a success. The approximate entropy test calculated the approximate entropy of a bitstream by checking how many bitstrings of length m appeared compared to how many of length $m + 1$ appeared. With m depending on the length of the bitstream. 99 of 100 bitstreams passed the approximate entropy test. The serial statistical test was similar to the approximate entropy test, since it also checked how many bitstrings of length m appeared and if this was expected from a random sequence. With $m = 1$ this test would be equivalent to the frequency test. Furthermore, the first serial test compared bitstrings of length m with strings of length $m - 1$. The second serial test compared bitstrings of lengths m, $m - 1$, and $m - 2$ with each other. In both cases, the tests passed 100 out of 100 times. Finally, the linear complexity test checked whether the data was created through a linear shift register. This was accomplished by calculating the minimum length of the linear shift register needed to create the bitstream. 97 out of 100 bitstreams passed the linear complexity test. Overall, all statistical tests passed both in terms of p-values and proportion. Hence, the NIST statistical test suite recognized the values created by async. SCRAPE as random.

C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	P-VALUE		PROPORTION		STATISTICAL TEST
56	8	3	7	4	5	2	2	8	5	0.000000	*	57/100	*	Frequency
86	4	0	3	2	1	1	1	1	1	0.000000	*	29/100	*	BlockFrequency
59	10	8	7	5	4	1	2	3	1	0.000000	*	56/100	*	CumulativeSums
64	8	3	4	7	2	6	1	2	3	0.000000	*	53/100	*	CumulativeSums
47	8	8	8	7	2	4	5	2	9	0.000000	*	72/100	*	Runs
36	13	8	7	9	4	5	7	6	5	0.000000	*	90/100	*	LongestRun
17	10	8	8	20	5	4	4	14	10	0.001399		94/100	*	Rank
11	15	7	9	9	10	9	15	4	11	0.350485		94/100	*	FFT
82	7	4	3	0	0	2	0	1	1	0.000000	*	38/100	*	ApproximateEntropy
77	5	6	2	2	0	5	1	2	0	0.000000	*	40/100	*	Serial
52	8	4	8	6	5	7	1	4	5	0.000000	*	71/100	*	Serial
13	6	8	8	9	10	7	5	19	15	0.042808		97/100		LinearComplexity

Table 5.3: Our thesis randomness report

To contrast the previous results seen in Table 5.2, we also ran the NIST statistical test suite on undoubtedly non-random data. For this, we chose an earlier version of our master thesis. The final report generated by the NIST statistical test suite can be seen in Table 5.3. The report structure is the same as seen in Table 5.2, with the only difference being the results. We see that the previous version of our master thesis failed every single test, other than the linear complexity test. It did not fail the linear complexity test, since the master thesis was not generated by a linear shift register. Furthermore, the bitstreams were deemed too similar to each other in all tests other than the rank test, the FFT test, and the linear complexity test. The bitstreams were deemed too similar, since they all contained English words and other similar information, hence they also contained similar bytes. We see that the NIST statistical test suite correctly deemed our master thesis to be non-random. This gives support to the previous claims, where the NIST statistical test suite deemed our random beacon implementation to be random. However, the possibility that a monkey with a typewriter typed a previous version of this thesis is not zero, and in that case, the NIST statistical test suite would have wrongly classified the data as non-random.

6 Conclusion

In the following chapter, we conclude our thesis. We do so by discussing distributed randomness in general and possible future work. We also discuss further considerations that appeared when analyzing distributed randomness.

6.1 Possible attack on SCRAPE

While analyzing our findings, we ran into a possible problem that comes with SCRAPE. Namely, that SCRAPE assumes the existence of a public ledger, or blockchain, to ensure total order. However, SCRAPE does not account for the presence of financially motivated miners inside the blockchain. These miners might behave maliciously to some extent when a financial reward can be extracted. This type of malicious behavior can be seen when miners themselves do front-running attacks [35].

A different type of financially motivated attack exists for SCRAPE inside blockchains. When a commit does not get opened $f + 1$ recovery shares have to be posted. This is a concern since it discourages miners from accepting reveals. If they accept a reveal, then a single value gets posted onto the blockchain. If they do not accept a reveal, then $f + 1$ parties have to post all the values necessary to recover a commit onto the blockchain. In the case of SCRAPE $3(f + 1)$ 256-bit values get posted onto the blockchain instead of a single 256-bit value, when a miner does not accept a reveal. Hence, miners are financially motivated to not accept reveals in the hopes that a recovery will take place. Especially with a blockchain running at full capacity, a miner suffers no loss from ignoring a reveal and instead putting a different transaction onto the blockchain. This reward for miners comes at the cost of participants and leads to the worst-case scenario of SCRAPE. However, what makes this attack less likely compared to front-running, is the fact that miners do not profit directly from refusing a reveal. Since the recovery would only happen in later blocks, so too would the profit only appear in later blocks. However, the blockchain in general would see more demand, hence it would still benefit a miner indirectly to refuse a reveal. An illustration of the described attack can be seen in Figure 6.1.

A possible counter to this type of attack is paying more to post reveals onto a blockchain compared to other data. If a party were to pay $3(f + 1)$ times the normal transaction fee for a reveal transaction, miners would lose their financial incentive and instead accept reveals right away. Furthermore, it might be enough to make a reveal slightly more expensive compared to a normal transaction. Since ignoring a reveal would

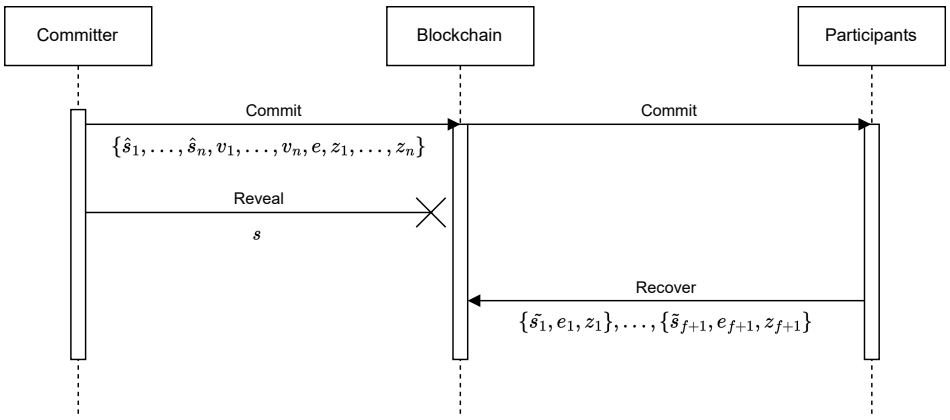


Figure 6.1: Illustration of financially incentivized attack on SCRAPE

then come at a cost for the miner. Another counter is to improve a blockchain and ensure that it does not run at full capacity. Or, the recovery of the random value could be done off chain. Furthermore, any defenses that apply to front-running should also apply to this type of attack, since preventing front-running usually entails obfuscating transactions.

It is still important to note that this type of attack might be transferable to all types of commit and reveal schemes running on a blockchain. As long as a recovery phase exists and requires more space or computational power than the reveal phase, this attack is viable. In async. SCRAPE this is not a problem since we always recover. Hence, miners cannot refuse reveals, since there are no reveals to be refused.

6.2 Scarcity of public randomness

This thesis set out to analyze and research existing random beacon protocols and pick one, or multiple, to implement. In doing so, we hoped to find out why random beacons have not found mainstream success. We read multiple papers and took a closer look at many protocols. In the end, we decided to implement SCRAPE and in doing so, we improved upon it. We made SCRAPE asynchronous by adding the shares together, which made the worst-case communication complexity decrease from $O(n^4)$ to $O(n^3)$ and the number of exponentiations decrease from $O(n^3)$ to $O(n^2)$. Since we were able to do all this, we found that existing protocols can still be made more efficient, robust, and versatile. We also assume that our version of SCRAPE can be further improved upon. This shows us that random beacons still have much space left to grow.

Space for improvement is a good thing when it comes to research. However, it makes distributed randomness difficult to trust. In cryptography, a protocol is trusted and accepted if it has withstood the test of time, and when it comes to random beacons, very few have done so. Every year, multiple new and improved protocols come out, and all of them differ in certain aspects from each other. Furthermore, the variation between the protocols makes comparing them more complex and leads to there being no definitive best random beacon protocol out there. It all depends on what is needed and what setting one is working in.

Lastly, if there is one thing we found out during development, it is that even the simplest random beacon protocol is still difficult to comprehend and implement. Even with Thetacrypt making the process more smooth, it still took multiple weeks to understand distributed randomness and multiple weeks to implement it. For this reason, it is understandable that developers would be hesitant to sink a lot of time into implementing a distributed random beacon. The scenario where all of their work becomes redundant

once the newest random beacon protocol gets released is not hard to imagine. However, as time moves on and a random beacon protocol exists uninterrupted for a longer time frame, more distributed randomness providers should appear.

6.3 Future work

As discussed, random beacons can still be improved upon. A possible next step would be to create a new random beacon protocol that does not depend on any previous work. While this does offer the possibility of discovering a groundbreaking random beacon protocol, the chances of doing so are slim considering how many different random beacon protocols already exist. Instead, we could listen to a common phrase in academia and try to stand on the shoulders of giants. Taking multiple protocols into consideration and building up on them instead. Since both Albatross and OptRand stem from SCRAPE, it should be possible to combine both works into one. Furthermore, we could also incorporate our work into the mix. Doing so would lead to, admittedly, more complexity, but it would also lead to a more robust and powerful protocol. This protocol would be stronger than its parts, and could probably be improved upon even further. This would lead to a new and improved random beacon protocol without luck being involved.

It is also worth looking into, generalizing the formula of async. SCRAPE, and transferring it to all random beacon protocols. Creating a general way to transform random beacons into an asynchronous setting by removing the reveal phase and going straight to the recovery phase. However, this would lead to problems when applied to proof of work based protocols, since the recovery there is much more computationally heavy compared to PVSS. However, removing the reveal phase would eliminate the possible attack on SCRAPE described in Section 6.1.

Lastly, while working on distributed randomness, and SCRAPE in general we made use of many cryptographic methods, i.e. secret sharing, hash functions, NIZK proofs, commitment schemes, and many more, all to create a single random value. It is not far-fetched that instead of sharing a random value, one could adapt SCRAPE to share more meaningful values with other participants. A possible adaptation of SCRAPE and async. SCRAPE could lead to a distributed key generation protocol. Another adaptation could lead to an anonymous voting system. It would be interesting to explore other options and see different possible use cases of SCRAPE.

6.3.1 Security

When it comes to random beacons, the security requirements are somewhat different compared to other fields of cryptography. One does not have to encrypt into the future, one simply has to encrypt for the present. Once the protocol is finished, so is the need for the values to be cryptographically secure. Modern cryptography has a security standard of 128 bits. This means that breaking an encryption would require at least 2^{128} computations, which is far beyond what any modern computer is capable of.

To show this, we shall take the currently fastest computer in the world, which is the frontier supercomputer [36]. It is capable of $1.1 \cdot 10^{18}$ computations per second. This is equal to around $3.5 \cdot 10^{25}$ computations a year. For the frontier supercomputer 128 bits of security would take

$$\text{years} \cdot 3.5 \cdot 10^{25} = 2^{128} \implies \text{years} \approx 2^{43.1}.$$

Hence, breaking 128 bits of security would take longer than the age of the universe. However, this calculation is flawed, since it neglects the fact that computers improve over time. Although it is no longer holds, we shall take Moore's law, which states that the number of transistors in an integrated circuit doubles about every two years, into consideration. We simplify Moore's law and assume that the computational

power doubles every two years. Solving the following equation leads to

$$3.5 \cdot 10^{25} \int_0^{\text{years}} 2^{\frac{x}{2}}\, dx = 2^{128} \implies \text{years} \approx 83.2$$

Hence, our current encryptions are safe for around 83.2 years according to Moore's law. This assumes that no other forces, such as quantum cryptography, come into play. Our random beacon protocol, however, does not need to be secure for 83 years. It only needs to be secure for the commit and reveal phase to pass without anyone being able to see the commits or falsify the reveals. Once those phases are done, the random value has already been generated. By setting a time to live (TTL) for our randomness, we can adapt the required security level. This could be done deterministically, like in the Bitcoin network, where the hash must have a certain number of zeros depending on the speed of the network. With a TTL of 10 minutes and the frontier supercomputer as our adversary, we would only need around 70 bits of security. Cutting down the message size by around 45.3%. This solution would benefit a blockchain, where every bit of data is costly. Furthermore, a lower security level simplifies the computations, which would make the computation cost lower.

Appendix

Nodes\Iterations	4	5	6	7	8	9	10
1	85.4	87.0	87.6	121.8	123.6	126.6	154.3
2	116.2	117.5	118.8	164.1	164.6	166.8	205.1
3	142.9	144.2	145.8	193.4	193.9	197.3	255.3
4	163.6	165.2	167.9	223.3	223.4	226.8	305.0
5	182.1	185.2	187.6	253.3	252.7	256.4	355.8
6	199.5	204.4	205.0	283.1	281.9	286.0	407.2
7	216.6	223.0	222.8	313.0	310.3	314.7	458.3
8	232.8	240.6	239.5	342.5	339.3	343.9	510.9
9	248.8	257.8	255.8	371.5	368.3	373.0	562.8
10	264.6	274.9	271.7	400.4	397.0	402.4	615.8

Table 6.1: Async. SCRAPE runtime in milliseconds, up to 10 nodes

Nodes\Iterations	4	5	6	7	8	9	10
1	92.8	93.2	95.4	134.7	135.4	137.4	187.7
2	129.1	129.5	132.3	177.1	176.2	179.0	265.1
3	156.3	156.2	158.3	218.1	215.9	218.8	342.3
4	180.0	180.5	183.1	258.5	255.1	258.3	420.4
5	200.8	202.5	206.3	298.4	294.2	296.9	498.6
6	221.9	224.0	228.5	337.6	332.4	336.4	579.1
7	243.0	245.4	249.5	376.8	372.3	377.1	660.3
8	263.6	265.6	270.4	416.4	411.0	416.6	740.6
9	284.5	286.3	291.2	455.3	449.2	455.9	820.2
10	305.0	306.2	311.6	493.4	489.4	495.5	897.3

Table 6.2: SCRAPE worst-case runtime in milliseconds, up to 10 nodes

Nodes\Iterations	1	10	20	30	40	50
5	0.092	0.281	0.448	0.614	0.771	0.963
10	0.164	0.634	1.155	1.676	2.234	2.789
15	0.233	1.31	2.537	3.743	4.925	6.15
20	0.443	3.167	6.049	9.066	12.051	14.958
25	0.792	6.182	12.109	18.104	24.202	30.075
30	1.046	8.41	16.544	24.834	33.037	41.402
35	1.59	14.423	28.285	42.688	57.118	72.033
40	2.25	19.623	39.338	59.151	79.448	100.4
45	2.521	23.495	47.283	71.703	96.496	122.127
50	3.756	32.135	63.829	96.516	129.786	164.212

Table 6.3: Async. SCRAPE runtime in seconds, up to 50 nodes

Nodes\Iterations	1	10	20	30	40	50
5	0.098	0.305	0.487	0.673	0.86	1.033
10	0.176	0.866	1.678	2.466	3.25	3.996
15	0.259	1.853	3.616	5.408	7.096	8.869
20	0.533	4.787	9.533	14.149	18.91	23.576
25	0.964	10.196	20.153	30.135	46.049	59.116
30	1.342	15.978	37.603	58.58	78.563	97.804
35	2.396	28.78	71.576	105.2	148.382	181.163
40	3.414	40.09	88.373	170.894	222.01	293.848
45	4.243	50.865	157.041	221.377	315.565	387.98
50	6.157	75.933	218.697	349.624	455.538	544.713

Table 6.4: SCRAPE worst-case runtime in seconds, up to 50 nodes

Nodes\Iterations	1	10	20	30	40	50
5	14	68	128	188	248	308
10	31	157	297	437	577	717
15	42	204	384	564	744	924
20	59	293	553	813	1073	1333
25	76	382	722	1062	1402	1742
30	87	429	809	1189	1569	1949
35	104	518	978	1438	1898	2358
40	111	565	1069	1573	2078	2582
45	121	654	1246	1838	2430	3022
50	131	743	1423	2103	2783	3463

Table 6.5: Async. SCRAPE number of messages, up to 50 nodes

Nodes\Iterations	1	10	20	30	40	50
5	17	98	188	278	368	458
10	52	367	717	1067	1417	1767
15	78	563	1101	1640	2179	2718
20	137	1073	2113	3153	4193	5233
25	212	1742	3442	5142	6842	8542
30	258	2139	4229	6319	8409	10499
35	357	3048	6038	9028	12018	15008
40	472	4117	8167	12217	16267	20317
45	538	4714	9354	13994	18634	23274
50	677	6023	11963	17903	23843	29783

Table 6.6: SCRAPE worst-case number of messages, up to 50 nodes

Nodes\Iterations	1	10	20	30	40	50
5	0.003	0.029	0.058	0.087	0.116	0.145
10	0.012	0.111	0.221	0.332	0.442	0.552
15	0.018	0.172	0.343	0.514	0.685	0.856
20	0.035	0.335	0.667	1.0	1.333	1.666
25	0.057	0.55	1.099	1.647	2.195	2.743
30	0.07	0.678	1.354	2.03	2.706	3.382
35	0.1	0.975	1.946	2.918	3.89	4.861
40	0.136	1.325	2.646	3.967	5.288	6.609
45	0.156	1.52	3.036	4.551	6.067	7.583
50	0.199	1.95	3.896	5.841	7.787	9.732

Table 6.7: Async. SCRAPE communication cost in megabytes, up to 50 nodes

Nodes\Iterations	1	10	20	30	40	50
5	0.004	0.035	0.07	0.104	0.139	0.173
10	0.015	0.146	0.291	0.435	0.58	0.725
15	0.024	0.23	0.458	0.687	0.916	1.144
20	0.047	0.456	0.909	1.363	1.817	2.27
25	0.078	0.758	1.513	2.269	3.024	3.78
30	0.096	0.938	1.873	2.808	3.743	4.678
35	0.138	1.355	2.707	4.059	5.41	6.762
40	0.188	1.849	3.694	5.539	7.384	9.23
45	0.216	2.125	4.245	6.366	8.486	10.607
50	0.277	2.733	5.462	8.191	10.92	13.649

Table 6.8: SCRAPE worst-case communication cost in megabytes, up to 50 nodes

Acknowledgments

Firstly, I would like to thank my supervisors Mariarosaria Barbaraci and Dr. Orestis Alpos for helping me through the difficulties that come with writing a master thesis. I would also like to thank Professor Christian Cachin for the many helpful inputs from his side and for giving me the idea to write about distributed randomness in the first place. Furthermore, I am also truly grateful towards my family and friends, who have supported me throughout my studies. Lastly, although I cannot put this part into words, I would like to thank my father, Svajus Joseph Asadauskas, who passed away last year. When I started writing this thesis, I was in a difficult period of my life, and without the help and kindness of everyone, I cannot imagine where I would be today.

Bibliography

[1] Victor Costan and Srinivas Devadas. Intel SGX explained. *IACR Cryptol. ePrint Arch.*, page 86, 2016.

[2] Fredrik H Leinfelt. Racial influences on the likelihood of police searches and search hits: A longitudinal analysis from an american midwestern city. *The Police Journal*, 79(3):238–257, 2006.

[3] Nicolas Gailly, Philipp Jovanovic, and Bryan Ford. Drand, distributed randomness beacon. `https://drand.love/`, 2019. Was accessed: 2023-09-21.

[4] Ignacio Cascudo and Bernardo David. ALBATROSS: publicly attestable batched randomness based on secret sharing. In *ASIACRYPT (3)*, volume 12493 of *Lecture Notes in Computer Science*, pages 311–341. Springer, 2020.

[5] Kevin Choi, Arasu Arun, Nirvan Tyagi, and Joseph Bonneau. Bicorn: An optimistically efficient distributed randomness beacon. In *FC (1)*, volume 13950 of *Lecture Notes in Computer Science*, pages 235–251. Springer, 2023.

[6] Ewa Syta, Philipp Jovanovic, Eleftherios Kokoris-Kogias, Nicolas Gailly, Linus Gasser, Ismail Khoffi, Michael J. Fischer, and Bryan Ford. Scalable bias-resistant distributed randomness. In *IEEE Symposium on Security and Privacy*, pages 444–460. IEEE Computer Society, 2017.

[7] Timo Hanke, Mahnush Movahedi, and Dominic Williams. DFINITY technology overview series, consensus system. *CoRR*, abs/1805.04548, 2018.

[8] Ignacio Cascudo, Bernardo David, Omer Shlomovits, and Denis Varlakov. Mt. random: Multi-tiered randomness beacons. In *ACNS*, volume 13906 of *Lecture Notes in Computer Science*, pages 645–674. Springer, 2023.

[9] Adithya Bhat, Nibesh Shrestha, Zhongtang Luo, Aniket Kate, and Kartik Nayak. Randpiper - reconfiguration-friendly random beacons with quadratic communication. In *CCS*, pages 3502–3524. ACM, 2021.

[10] Alisa Cherniaeva, Ilia Shirobokov, and Omer Shlomovits. Homomorphic encryption random beacon. *IACR Cryptol. ePrint Arch.*, page 1320, 2019.

[11] Philipp Schindler, Aljosha Judmayer, Nicholas Stifter, and Edgar R. Weippl. Hydrand: Efficient continuous distributed randomness. In *SP*, pages 73–89. IEEE, 2020.

[12] Adithya Bhat, Nibesh Shrestha, Aniket Kate, and Kartik Nayak. Optrand: Optimistically responsive reconfigurable distributed randomness. In *NDSS*. The Internet Society, 2023.

[13] Ewa Syta, Philipp Jovanovic, Eleftherios Kokoris-Kogias, Nicolas Gailly, Linus Gasser, Ismail Khoffi, Michael J. Fischer, and Bryan Ford. Scalable bias-resistant distributed randomness. In *IEEE Symposium on Security and Privacy*, pages 444–460. IEEE Computer Society, 2017.

[14] Philipp Schindler, Aljosha Judmayer, Markus Hittmeir, Nicholas Stifter, and Edgar R. Weippl. Randrunner: Distributed randomness from trapdoor vdfs with strong uniqueness. In *NDSS*. The Internet Society, 2021.

[15] Sourav Das, Vinith Krishnan, Irene Miriam Isaac, and Ling Ren. Spurt: Scalable distributed randomness beacon with transparent setup. In *SP*, pages 2502–2517. IEEE, 2022.

[16] Ignacio Cascudo and Bernardo David. SCRAPE: scalable randomness attested by public entities. In *ACNS*, volume 10355 of *Lecture Notes in Computer Science*, pages 537–556. Springer, 2017.

[17] Donald Beaver, Konstantinos Chalkias, Mahimna Kelkar, Lefteris Kokoris-Kogias, Kevin Lewi, Ladi de Naurois, Valeria Nikolaenko, Arnab Roy, and Alberto Sonnino. STROBE: streaming threshold random beacons. In *AFT*, volume 282 of *LIPIcs*, pages 7:1–7:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023.

[18] Arjen K. Lenstra and Benjamin Wesolowski. A random zoo: sloth, unicorn, and trx. *IACR Cryptol. ePrint Arch.*, page 366, 2015.

[19] Kevin Choi, Aathira Manoj, and Joseph Bonneau. Sok: Distributed randomness beacons. In *SP*, pages 75–92. IEEE, 2023.

[20] Cynthia Dwork, Nancy A. Lynch, and Larry J. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, 1988.

[21] Prof. Dr. James L. Massey. Applied digital information theory ii, 2008. `http://www.isiweb.ee.ethz.ch/archive/massey_scr/adit2.pdf`.

[22] Donald E. Eastlake III and Tony Hansen. US secure hash algorithms (SHA and sha-based HMAC and HKDF). *RFC*, 6234:1–127, 2011.

[23] David Chaum and Torben P. Pedersen. Wallet databases with observers. In *CRYPTO*, volume 740 of *Lecture Notes in Computer Science*, pages 89–105. Springer, 1992.

[24] Chunming Tang, Dingyi Pei, Zhuojun Liu, Zheng-an Yao, and Mingsheng Wang. Perfectly hiding commitment scheme with two-round from any one-way permutation. *IACR Cryptol. ePrint Arch.*, page 34, 2008.

[25] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.

[26] Torben P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *CRYPTO*, volume 576 of *Lecture Notes in Computer Science*, pages 129–140. Springer, 1991.

[27] Howard E. Brandt. Review of *Protecting Information: From Classical Error Correction to Quantum Cryptography* by susan loepp and william k. wootters. *Cryptologia*, 33(2):205–207, 2009.

[28] Graydon Hoare. Rust, a language empowering everyone to build reliable and efficient software. `https://www.rust-lang.org/`, 2015. Was accessed: 2024-01-03.

[29] Carl Lerche. Tokio, build reliable network applications without compromising speed. `https://tokio.rs/`, 2016. Was accessed: 2024-01-03.

[30] Orestis Alpos, Mariarosaria Barbaraci, Christian Cachin, Noah Schmid, and Michael Senn. Thetacrypt: A distributed service for threshold cryptography on-demand: Demo abstract. In *Proceedings of the 24th International Middleware Conference Demos, Posters and Doctoral Symposium, Bologna, Italy, December 11-15, 2023*, pages 33–34. ACM, 2023.

[31] Orestis Alpos, Mariarosaria Barbaraci, Christian Cachin, Noah Schmid, and Michael Senn. Github of thetacrypt - threshold cryptography library in rust. `https://github.com/cryptobern/thetacrypt`, 2021. Was accessed: 2024-01-15.

[32] Solomon Hykes. Docker, make better, secure software from the start. `https://www.docker.com/`, 2013.

[33] Adam Selipsky and C.J. Moses. Amazon ec2, secure and resizable compute capacity for virtually any workload. `https://aws.amazon.com/ec2/`, 2006. Was accessed: 2023-11-06.

[34] Andrew Rukhin, Juan Soto, James Nechvatal, Miles Smid, Elaine Barker, Stefan Leigh, Mark Levenson, Mark Vangel, David Banks, Alan Heckert, et al. *A statistical test suite for random and pseudorandom number generators for cryptographic applications*, volume 22. National Institute of Standards & Technology, 2001.

[35] Philip Daian, Steven Goldfeder, Tyler Kell, Yunqi Li, Xueyuan Zhao, Iddo Bentov, Lorenz Breidenbach, and Ari Juels. Flash boys 2.0: Frontrunning in decentralized exchanges, miner extractable value, and consensus instability. In *SP*, pages 910–927. IEEE, 2020.

[36] Georgia Tourassi Justin L Whitt and Bronson E Messer II. Frontier supercomputer debuts as world's fastest, breaking exascale barrier. `https://www.ornl.gov/news/frontier-supercomputer-debuts-worlds-fastest-breaking-exascale-barrier/`, 2022. Was accessed: 2023-11-06.

Declaration of consent

on the basis of Article 30 of the RSL Phil.-nat. 18

Name/First Name: Asadauskas, Marius Paulius

Registration Number: 18-106-575

Study program: MSc Computer Science

Bachelor ☐ Master ☑ Dissertation ☐

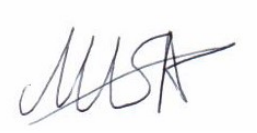
Title of the thesis: Implementing Distributed Randomness
Adapting a random beacon protocol to run in practice

Supervisor: Prof. Dr. Christian Cachin

I declare herewith that this thesis is my own work and that I have not used any sources other than those stated. I have indicated the adoption of quotations as well as thoughts taken from other authors as such in the thesis. I am aware that the Senate pursuant to Article 36 paragraph 1 litera r of the University Act of 5 September, 1996 is authorized to revoke the title awarded on the basis of this thesis.
For the purposes of evaluation and verification of compliance with the declaration of originality and the regulations governing plagiarism, I hereby grant the University of Bern the right to process my personal data and to perform the acts of use this requires, in particular, to reproduce the written thesis and to store it permanently in a database, and to use said database, or to make said database available, to enable comparison with future theses submitted by others.

Bern, 08.02.2024

Place/Date



Signature