Analysis and Applications of VRFs

Adrián Castro Lara

Resum– En aquest projecte analitzarem i definirem què són les VRFs (Verifiable Random Functions). A més a més, estudiarem la motivació que hi va haver darrere per tal de crear-les i començar a utilitzar-les. Després, estudiarem dues maneres de com s'implementen, mitjançant l'ús de criptografia de clau pública RSA i EC (Corbes El·líptiques). Dins les EC analitzarem les implementacions basades estrictament en corbes i les derivades d'elles mitjançant la utilització de *pairings*. A continuació, estudiarem diferents tipus d'implementacions i aplicacions que fan ús de les VRFs, com són les criptomonedes Cardano, Polkadot i Algorand i el criptosistema de NSEC5. Per acabar, implementarem amb Python un protocol de VRF mitjançant EC basat en l'esquema de signatura digital ECDSA (Elliptic Curve Digital Signature Algorithm) de Bitcoin per tal de solucionar el problema de l'*Anti-Exfil*.

Paraules clau– VRF, Clau Pública, RSA, Corbes El·líptiques (EC), Logaritme Discret, Pairings, ECDSA, Nonce, Polkadot, Cardano, Algorand, NSEC5, Anti-Exfil, Bitcoin.

Abstract–In this project we will analyze and define what VRFs (Verifiable Random Functions) are. In addition, we will study the motivation behind creating them and start using them. Then, we will study two ways of how they are implemented, by using RSA and EC (Elliptic Curves) public key cryptography. Within the EC we will analyze implementations strictly based on curves and those derived from them through the use of *pairings*. Next, we will study different types of implementations and applications that implement VRFs, such as Cardano, Polkadot and Algorand cryptocurrencies and the NSEC5 cryptosystem. Finally, we will implement in Python a VRF protocol using EC based on Bitcoin's ECDSA (Elliptic Curve Digital Signature Algorithm) digital signature scheme in order to solve the problem of *Anti-Exfil*.

Keywords– VRF, Public Key, RSA, Ellptic Curves (EC), Discrete Logarithm, Pairings, ECDSA, Nonce, Polkadot, Cardano, Algorand, NSEC5, Anti-Exfil, Bitcoin.

1 Introduction - Context of the work

A VRF is a pseudo-random function that, using public key cryptography, such as RSA or Elliptic Curves, generates and provides a value, which appears to be random, and a proof that this value has been generated correctly.

The mechanism allows the owner of the private key to compute and provide the random value and proof. Once delivered, anyone can verify using the owner's public key that this value was generated correctly. However, for the system to be secure and cryptographically useful, from the public key, value, and proof must not be able to obtain the owner's private key.

• Contact email: adrian.castrol@autonoma.cat
• Mention made: Information Technologies
• Work supervised by: Jordi Herrera Joancomarti (DEIC)
• Course 2022/23

Once the basis and operation of VRFs have been explained and conceptualized, we will then proceed to explain the motivation for which VRFs arose and the problems they solve. By definition, a pseudorandom oracle [GGM86] is not verifiable, unless the *seed*, s, is published, which is the *seed* responsible for generating the pseudorandom values. However, with the publication of the *seed* everyone could generate the pseudo-random values, making the oracle dispensable, since it would lose the property of unpredictability. Therefore, in order to trust the values generated by the oracle, without having to publish s, they must be fulfilled certain characteristics of mutual trust and reliability between the user generating the *seed* and the user who evaluates it. These problems are efficiently solved by generating a proof, together with the pseudorandom value, which allows the correctness of this value to be verified.

Next, we go on to define the basic characteristics that all systems that implement VRFs must have [GRPV22]:

- **Uniqueness**: For any fixed VRF public key and for any input, it is not possible to find evidence for more

than one VRF output.
- **Provability**: Given a value and its corresponding proof, we must be able to prove and verify the value correctly.
- **Pseudorandomness**: No value of the function can be distinguished from chance, even after seeing every other value of the function along with the proofs.
- **Full collision resistance**: As with cryptographic hash functions, VRFs must be collision resistant. This fact implies that it is infeasible to find two different inputs with the same output.

2 OBJECTIVES

In this section we will describe the project objectives which, more informally, have been discussed in the introduction. Basically, this project will consist of two main objectives:

- Study and theoretically analyze the functioning of VRFs and their two main implementation mechanisms, through EC and RSA.
- Implement a protocol in Python using EC that highlights the functionality of the VRFs and solves the *Anti-Exfil* problem.

However, we will also define specific objectives that will help us achieve our main objectives. These specific objectives are the following:

- Analyze and study the first manifesto and appearance of the VRFs by Silvio Micali, Michael Rabiny, Salil Vadhanz [MRV99].
- Understand Micali, Rabiny and Vadhanz's implementation of VRFs using RSA.
- Study the cryptographic structure and use of Elliptic Curves and *pairings*.
- Analyze the Dodis and Yampolskiy implementation using *pairings* [DY05].
- Examine the different practical applications that exist today on VRFs.
- Implement the VRF protocol using Python and integrating it with the ECDSA scheme.

3 METHODOLOGY

The next important point to highlight is the methodology we will use and follow to develop the project. Today, there are many different types of methodologies depending on the project or work to be done. In our case, as it is more of a project of research and subsequent development, the one that best fits our needs and objectives is the Scrum methodology [SS20].

This methodology is usually used in team projects to improve and streamline cooperation. In our case it is not a cooperative project, but it is true that many of the techniques of organization and detection of problems will be useful in our development. To begin, we will divide the project into *Sprints*, which consist of events of fixed duration that serve to determine the work that will be done in each of them in order to reach the final objectives. In our case, each *Sprint* will correspond to each of the partial and final deliveries and will have an approximate duration of one month each.

In addition, each *Sprint* must have the *Daily Scrums* which are daily sessions, in our case weekly, in which there is a review of how the *Sprint* is going. Then, we attempt to detect and solve errors or difficulties in order to be able to satisfactorily meet the dates and work dedicated to each *Sprint*. Finally, for each one once its delivery date arrives, an assessment session must be held to see if the objectives have been achieved. In addition, it must also be assessed if they should make modifications in the following *Sprints*. In order to plan it, we will use the Trello application that allows you to manage a project by creating tasks, to which you can define subsections and classify them by categories. In addition, this online application allows us to add another program called TeamGantt with which we can represent all tasks and categories in a Gantt chart. This Gantt diagram will allow us to see more graphically and visually the distribution of tasks over time and in the different *Sprints*.

4 THEORETICAL STUDY

4.1 History

The first appearance of VRFs was due to the authors Silvio Micali, Michael Rabiny and Salil Vadhanz in their paper *Verifiable Random Functions* in 1999 [MRV99]. In this paper we are introduced to the concept in a very theoretical way with which the authors intend to effectively combine the unpredictability and verifiability of pseudorandom functions by means of a secret *seed*, s. This construction that the authors will make is based on the implementation of pseudorandom functions described by Goldreich, Goldwasser and Micali himself in their document [GGM86].

The goal of the 1999 paper is that by knowing s we can evaluate a point x and provide a Nondeterministic Polynomial Time (NP) proof so that it can be shown that $f_s(x)$ has been generated correctly. Furthermore, the goal is also not to compromise the unpredictability of f_s at any other point for which no proof is provided. However, what this initial version intended was first of all to generate a VUF (Verifiable Unpredictable Function) and later through a *mapping* system of bits based on the generation of trees to transform it into a VRF. This implementation was quite complex, inefficient and difficult to apply.

In order to solve this, in 2005, Dodis and Yampolskiy in their paper *A Verifiable Random Function with Short Proofs and Keys* [DY05] presented, as the title indicates, a more optimal way and efficient to generate VRFs by using bilinear maps constructed from *pairings*.

4.2 VRF proposal by Micali, Rabiny, Vadhanz (1999)

This study, as we mentioned above, sought to solve the problem faced by Goldreich, Goldwasser and Micali. This problem is that a pseudorandom oracle is not verifiable, unless the *seed s* is published which is responsible for generating the values. However, publishing the *seed* would cause everyone to be able to generate the pseudo-random values and the oracle functionality would be dispensable, as it would lose the property of unpredictability. This fact causes that in order to trust the values generated by the or-

acle, without having to publish s, the following characteristics must be met:

- The user who generates the *seed* and evaluates it using the oracle is completely trustworthy.
- The user who evaluates the *seed* gains a profit for generating the values correctly, or gains nothing for generating the values dishonestly.

For this reason, Micali, Rabiny and Vadhanz propose to solve these problems efficiently by generating a proof, together with the pseudorandom value, which allows to verify the correctness of this value. In order to solve this, the first thing they propose is to generate a zero-knowledge proof that allows the value to be authenticated. To do this, a Bellare and Goldwasser [BG89] signature scheme is generated, where the owner of the *seed*, *s*, through a pseudorandom oracle f_s can publish a commitment, c. Later, whenever the owner wanted to prove that *v* is its value at a point *x* to a verifier *V*, it can prove with zero-knowledge in *V* that $v = f_s(x)$ and that *c* is a compromise of *s*.

However, as the authors themselves assure, this approach provides us with a weak solution, as it presents some problems. The main drawback of this scheme is that it requires interaction. Nevertheless, they claim that this problem can be solved by using non-interactive zero-knowledge proofs (NIZK) as Bellare and Goldwasser did [BG89]. This approach suffers from another drawback, which is that this type of proofs presupposes that the verifier and the generator share a string of bits that is guaranteed to be random. However, the following problems arise here regarding the choice and generation of the *R* bit string:

1. The owner of the *seed* selects *R*: If he selects the shared random string incorrectly, the robustness and reliability of the NIZK system is no longer guaranteed, since there may be many values *v* that are verifiable as $f_s(x)$.
2. Verifier selects *R*: If he selects the shared random string incorrectly, the NIZK property is no longer guaranteed. In this way, the owner by proving that $f_s(x) = v$ with respect to the incorrectly chosen *R* may cause the verifier to leak knowledge about the *seed s*.
3. The owner of the *seed* and the verifier jointly select *R* using a *coinflipping* protocol: This method is invalid as it requires interaction and we are trying to avoid it.
4. A trusted third party selects *R*: We do not want to assume the existence of a trusted third party.

In order to solve these drawbacks and problems, what Micali, Rabiny and Vadhanz propose is to use a random number generation scheme based on the RSA public key cryptosystem.

4.2.1 RSA Public Key Scheme

The RSA cryptographic system was designed and published in 1978 by R. Rivest, A. Shamir, and L. Adleman [RSA78]. The security and robustness of the algorithm is based on the problem of integer factorization. This scheme is one of the most widespread in public key cryptography

Although RSA can be used for encryption/decryption schemes and for digital signature schemes. We will analyze only the signatures, since they are the ones used by Micali, Rabiny and Vadhanz. The RSA Digital Signature algorithm consists of the following phases:

Key Generation:

1. We choose p and q (large prime numbers).
2. We calculate n as $n = p \cdot q$.
3. We calculate $\phi(n) = (p-1) \cdot (q-1)$.
4. We choose e such that $1 < e < \phi(n)$ and $mcd(e, \phi(n)) = 1$.
5. We calculate d as $d \equiv e^{-1} \pmod{\phi(n)}$.
6. We obtain $PK(e, n)$ and $SK(d, n)$.

Digital Signatures: In order to explain the process of creating digital signatures and their corresponding verification, we will define a scenario where there are 2 users, one called Alice and one called Bob.

1. We convert the message M to an integer m.
2. We calculate the value of the signature $s(m)$ as $s \equiv m^d \pmod{n}$ where d and n are Bob's private keys.
3. Alice receives s and checks the signature by doing $m' \equiv s^e \pmod{n} \equiv m^{e \cdot d} \pmod{n}$ where e and n are Bob's public keys.

4.2.2 VRF implementation using RSA

Once we have seen the digital signature scheme with RSA and its operation we will explain and evaluate the operation of the VRFs proposed in 1999 through the use of RSA. The algorithm they proposed can be separated into the following functions:

-PrimeSeq($a, Q, coins$)
Inputs: A value *a* that corresponds to the length of the prime numbers to be generated, and a polynomial *Q* in *GF(2^k)* where *k* is the number of bits of *a* and finally a string of *l*-bits called *coins* which can be interpreted as the random *seed*.
Outputs: A prime number p_x of length *k+1*.
Procedure:

1. We generate a vector y of values of length j.
2. We pass the vector y to the $PrimalyTest()$ function as a parameter, which is an external function that generated the value of $coins$, and selects the first number among all y_j which is prime.
3. We return this prime number as p_x.

-G(1^k)
Inputs: A security parameter 1^k.
Outputs: A public key $PK = (m, r, Q, coins)$ and a private key $SK(PK, \phi(m))$ where *m* is the RSA modulus, *r* is a random number such that $r \in Z_m^*$, *coins* is the *seed* mentioned above, and *Q* is the polynomial in $GF(2^k)$ where *k* is the number of bits.
Procedure:

1. We use the $PrimeSeq()$ function that will return prime numbers and through the trial and error mechanism, we choose two prime numbers q_1 and q_2 that have length $(k-1)/2$ and calculate $m = q_1 \cdot q_2$.
2. We calculate $\phi(m) = (q_1 - 1) \cdot (q_2 - 1)$.
3. We choose from the variable $coins$ a value $r \in Z_m^*$.
4. We choose a polynomial in $GF(2^k)$ of degree k that is irreducible which we call Q.
5. We define $PK = (m, r, Q, coins)$, $SK = (PK, \phi(m))$.

-F(SK, x)
Inputs: A private key $SK(PK, \phi(m))$ where $PK = (m, r, Q, coins)$ and a value *x* of length *a*-bits.
Outputs: A value $v \in Z_m^*$ which is the proof itself.
Procedure:

1. We run the $PrimeSeq()$ function so that we get $p_x = PrimeSeq(x, Q, coins)$.
2. We calculate $v \equiv r^{1/p_x} \pmod m$ which can be calculated simply, since we know $\phi(m)$.

-V(PK, x, v)
Inputs: A public key $PK = (m, r, Q, coins)$ and a value *x* of length *a*-bits and the value to be verified *v*.
Outputs: $1/True$ in case it is verified, otherwise $0/False$.
Procedure:

1. We run the $PrimeSeq()$ function so that we get $p_x = PrimeSeq(x, Q, coins)$.
2. We check that p_x is greater than m and that it is a prime number using the $PrimalyTest()$ function.
3. We check that $v \in Z_m^*$ and that $r \equiv v^{p_x} \pmod m$.
4. If all the checks are true, we return a $True$, if any of them fail, we stop execution and return a $False$.

Nevertheless, this algorithm presents the problem that the inputs are limited to k-bits. In Appendix A.1 we can find the tree-based method presented by the authors to solve this problem. On the other hand, in Appendix A.3 we can find the cryptographic security properties and characteristics of the system.

4.3 VRF proposal by Yevgeniy Dodis and Aleksandr Yampolskiy (2005)

Having seen the first and original implementation of VRFs, in 2005 the authors Yevgeniy Dodis and Aleksandr Yampolskiy present a new method to implement VRFs based on elliptic curves and *pairings* (bilinear groups). The authors define their proposal as a simple and efficient construction of a verifiable random function using *pairings*.

Their proposal arose, since the VRFs of Micali, Rabiny and Vadhanz operate on a multiplicative group Z_n^* which must be very large in order to be able to achieve reasonable security due to being based on RSA . Another reason for the emergence of Dodis and Yampolskiy's implementation is the fact that the previous proposal with RSA is complex, inefficient, and the inputs must be fixed at k-bit length. It is for this reason that in the 2005 proposal for inputs of any size their implementation of VRF generates proofs of constant size. In order to achieve this, they use a collision-resistant hash function which ensures that VRFs with arbitrary inputs can be used.

As we mentioned above, the construction of Dodis and Yampolskiy is based on bilinear groups which are based on elliptic curve cryptography. This type of group has the properties of Diffie-Hellman decision-based assumption (DDH) and also Computational Diffie-Hellman assumption (CDH) which we will see in more detail, but which basically offer many useful properties such as verifiability.

4.3.1 Elliptic Curves Cryptography

The use of elliptic curves in the design of public-key cryptosystems was first proposed in 1985 by Neal Koblitz [NK87] and Victor Miller [VM85] in separate studies. The way in which they proposed to use elliptic curves within the world of cryptography is based on using the group of points of an elliptic curve defined on a finite body. The security of this cryptosystem is based on the discrete logarithm problem on elliptic curves, thus shedding light on elliptic curve cryptography (ECC).

The main advantage that elliptic curve cryptography provides is that we can use smaller keys than other traditional public key cryptography algorithms but obtain the same level of security. Decreasing the size of the key makes the computation and execution speed much faster and more efficient in some of the basic primitives, as well as saving resources in both computing capacity and storage capacity. The security level of an algorithm is a measure created to compare the security offered by different cryptographic algorithms when used with various key sizes. The level is measured by a value n which is equivalent to the best known attack against the algorithm when it requires 2^n steps. Table 1 shows a comparison of key sizes for different cryptosystems, as well as the level of security provided and the problems faced by each cryptosystem. So we can see that for a security level of 128 with RSA we need keys of size 3072 bits while with elliptic curves we get the same security level with 256 bit keys.

	Cryptographic Algorithms			
Security Level	Symmetric Key AES,3DES	Integers Factorization RSA	Discrete Logarithm DSA, DH, ElGamal	Elliptic Curves ECDSA, ECDH
80	80	1024	1024	160
112	112	2048	2048	224
128	128	3072	3072	256
192	192	7680	7680	384
256	256	15360	15360	512

TABLE 1. SECURITY LEVEL AGAINST KEY SIZE (BITS) FOR DIFFERENT ALGORITHMS

Having contextualized elliptic curves and analyzed the motivation behind their use, the first thing to do is to determine what an elliptic curve is and what shape does it have.

We can define an elliptic curve E/Z_p (with $p > 3$) as the set of all pairs $(x, y) \in Z_p$ such that they follow the following equation: $y^2 = x^3 + ax + b \bmod p$, together with an imaginary point at infinity ∂. One of the conditions that elliptic curves must fulfill is that $a, b \in Z_p$ and $\Delta = -16(4a^3 + 27b^2) \neq 0$ mod p. This Δ value is known as the discriminant and determines the necessary condition to prevent the curve from having a vertex or from crossing itself. If the condition is not met, the curve is not suitable for use in cryptographic algorithms. The expression of the equation of the curve specified above is known as the Weiertrass short form.

Once we have seen its form, what we will do next will be to define its group operation, which later is the one we will use in each protocol. In the case of elliptic curves we will define the sum group operation which will allow us to define the concept of scalar multiplication. We define the sum operation as the sum of two points that originate a third and the scalar multiplication as $nP = P + P + P...$ up to n times.

To finish, it is necessary to determine what are the parameters of a curve for cryptographic uses:

- **Base point G:** is a point that allows generating all the other points of the curve from itself and from the scalar multiplication of different values.
- **A prime number p:** specifies the dimension of the finite field.
- **The coefficients a and b:** define the curve E/Z_p, $y^2 = x^3 + ax + b \bmod p$.
- **The prime order n** of the base point G.
- **Cofactor $f = \#E/n$** where $\#E$ is the number of points on the curve in Z_p.

Regarding the algorithms and protocols that use elliptic curves there are the Diffie-Hellman Key Exchange using Elliptic Curves (ECDH), the Elliptic Curve Integrated Encryption Scheme (ECIES) and the Elliptic Curve Digital Signature Algorithm (ECDSA), among others. However, we will only analyze ECDSA, since as we mentioned above, for the practical part we will implement a VRF solution in the Bitcoin ECDSA protocol.

Elliptic Curve Digital Signature Algorithm (ECDSA)

In the same way that happened with the RSA signature protocol, there is a protocol based on elliptic curves that allows us to sign messages that can be verified using the public keys of the signatories. The algorithm is made up of three blocks: the first is in charge of key generation, the second of the signature and the last of the signature verification.

-Key Generation:

1. We choose a random integer $k_{priv} = d \in (1 \dots, n)$.
2. We calculate $k_{pub} = B = d \cdot G$.

-Signature of messages:

The variables we need to perform the signature are the message m, the private key d and the parameters (p, a, b, G, n) of the elliptic curve. Once the parameters are defined, the algorithm has the following steps:

1. We choose an ephemeral key $k_e \in (1 \dots, n)$.
2. We calculate $R = k_e \cdot G$ where R is a point on the curve (x_R, y_R).
3. We define $r = x_R \bmod n$. If $r = 0$ we return to step 1.
4. We calculate $e = H(m)$.
5. We calculate $s = (e + d \cdot r) \cdot k_e^{-1} \bmod n$. If $s = 0$ we return to step 1.
6. We define the signature as $S = (r, s)$.

-Verification of message signatures:

From a signature $S = (r, s)$, a message m, the corresponding public key B and the parameters (p, a, b, G, n) of the elliptic curve we can check a signature as follows:

1. We verify that $r \in (0 \dots, n)$.
2. We calculate $e = H(m)$.
3. We calculate $w = s^{-1} \bmod n$.
4. We calculate $u_1 = w \cdot e \bmod n$ and $u_2 = w \cdot r \bmod n$.
5. We calculate $P = u_1 \cdot G + u_2 \cdot B$, where P is a point on the curve (x_P, y_P).
6. If $x_P = r \bmod n$ the signature is valid, otherwise it is invalid.

4.3.2 Bilinear Groups (Pairings)

Having seen elliptic curves and their cryptographic characteristics and applications, we will next look at bilinear groups also known as *pairings*. The concept of *pairing* arises from some elliptic curves that present an additional structure that allows us to obtain new cryptographic applications. Since the mathematics behind *pairings* are very complex and there are a large number of calculations and operations that are difficult to understand and interpret, we will define the inner workings of a *pairing* as a *black box*. What we will do is define how they work and what properties and characteristics they present.

To be able to define a *pairing*, we will first define the cyclic groups G_0, G_1 and G_T of prime order q with $g_0 \in G_0$, $g_1 \in G_1$ and $g_T \in G_T$ which are generating elements of each group. We will call G_0 and G_1, as the source groups and G_T as the target group. Then we can define a *pairing* as an application $e : G_0 \text{ x } G_1 \rightarrow G_T$ which must satisfy the following properties:

- **Bilinearity**, that is, for all $P_0, Q_0 \in G_0$ and $P_1, Q_1 \in G_1$:
  $e(P_0, P_1 + Q_1) = e(P_0, P_1) \cdot e(P_0, Q_1)$
  $e(P_0 + Q_0, P_1) = e(P_0, P_1) \cdot e(Q_0, P_1)$
- **Non-degenerate**, that is, $g_T = e(g_0, g_1)$ is a generator of G_T and that $e(g, g) \neq 1$.
- In addition, it must also satisfy the following property which is derived from bilinearity: $e(\alpha P_0, \beta P_1) = e(P_0, P_1)^{\alpha\beta} = e(\beta P_0, \alpha P_1)$

Finally, it should be added that there are different types of *pairings*, but not all of them have cryptographic applications, the only *pairings* known to have cryptographic applications are the Tate and Weil *pairings*.

4.3.3 VRF implementation using Pairings

In this section, having already seen the operation of elliptic curves and *pairings*, we will proceed to describe the construction presented by Yevgeniy Dodis and Aleksandr Yampolskiy [DY05]. The characteristics and properties of the resulting VRF are the same as the original Micali group version, but with a different implementation. The protocol consists of three parts:

-G(1^k)

Inputs: A g value that is the generator of the G group.
Outputs: A random value SK that corresponds to the private key and a value PK that is the public key both of k-bits.
Procedure:

1. We choose a random value $s \in Z_p$ and define $SK = s$.
2. We calculate $PK = g^s$.

-Prove$_{SK}$(x)

Inputs: A value x of k bits, SK which is the private key, and the generator g of the group G.
Outputs: A y value of k-bits which corresponds to the value generated with the VRF and a π proof.
Procedure:

1. We calculate $y = e(g, g)^{1/(x+SK)}$.
2. We calculate $\pi = g^{1/(x+SK)}$.

-Ver$_{PK}$(x,y,π)
Inputs: A k-bit value x, PK which is the public key, the generated value y, the corresponding proof π, and the generator g of the group G .
Outputs: 1 if it was successfully validated or 0 otherwise.

1. We check if $e(g^x \cdot PK, \pi) = e(g, g)$. We develop thanks to the properties of bilinearity and non-degeneracy as follows: $e(g^x \cdot PK, \pi) = e(g^x \cdot g^s, g^{1/(x+SK)}) = e(g^{x+s}, g^{1/(x+s)}) = e(g, g)^{(x+s)/(x+s)} = e(g, g)$.
2. We check if $y = e(g, \pi)$. In the same way as we did in the previous step, thanks to the properties of bilinearity and non-degeneracy as follows: $y = e(g, \pi) \rightarrow e(g, g)^{1/(x+SK)} = e(g, g^{1/(x+SK)}) \rightarrow e(g, g)^{1/(x+SK)} = e(g, g)^{1/(x+SK)}$.
3. If both conditions are met, we return a 1, since the verification will be valid, otherwise we return a 0.

In the same way as with the proposal with the RSA, this algorithm presents the problem that the entries are limited to k-bits. In Appendix A.2 we can find the properties of the hash functions presented by the authors to solve this problem. On the other hand, in Appendix A.4 we can find the cryptographic security properties and characteristics of the system based on the Diffie-Hellman assumptions.

5 State of the Art and Current Applications

Although at the time when VRFs appeared there were not many ideas or applications where they could be used and implemented, today thanks to new advances, discoveries and improvements in the VRFs protocol itself, it has allowed these to be applied and used.

As we mentioned, in the beginning their implementation was expensive and complex, but thanks to the improvement of Dodis and Yampolskiy VRFs can be built efficiently and guaranteeing a good level of security. It is for this reason that most VRF applications today are implemented with elliptic curves or *pairings*, although there are also schemes implemented with RSA.

Among the wide variety of applications where they are or can be used, those applications where they generate the most renown and impact are cryptocurrencies. It is not surprising that a world as widespread and current as that of cryptocurrencies would find an application for VRFs. Some cryptocurrencies that use them are Polkadot, Cardano and Algorand. On the other hand, in the field of internet security schemes and protocols, they are also used to improve existing ones, such as DNSSEC.

Finally, another field where VRFs can be of interest is the world of online gambling such as poker, roulette, black jack... In these types of applications there is a hallmark called *Provably Fair* which is a credibility factor given to the page as long as its data and results are proven to be equally verifiable and randomly and correctly generated. Today this distinction is achieved in many different ways, however a very good practice would be to obtain it through the implementation of a VRF. An example is Chainlink [CHL] which is a web service that offers the generation of proofs and random values using VRFs. Chainlink's primary goal is, through the use of a VRF, to enable and accelerate the development of smart contracts focused on blockchain gaming, security, layer two protocols, and other use cases. This fact allows well-made systems that rely on chance to be fair/equal and uncertain for all contract participants, while successfully reducing the risk that an adversary can exploit your contract by predicting its outcomes.

5.1 Cryptocurrencies

In this section we will describe how VRFs are used in some cryptocurrencies. Specifically, we will study the use made by Polkadot, Cardano and Algorand.

5.1.1 Polkadot

In the case of the Polkadot [AB22] cryptocurrency, the VRF is used in the block production algorithm. This algorithm is known as BABE (Blind Assignment for Blockchain Extension) and is executed between the validating nodes to determine the authors of new blocks. In this mechanism, time is divided into *epochs* that at the same time are also divided into *slots*, where each *slot* has a duration of 6 seconds. An *epoch* is composed of 2400 *slots* making the duration of each *epoch* 4 hours. For each *slot* a block is produced and at the beginning of each *epoch*, all selected validators participate in a random selection process to determine which *slot* will be responsible for producing the block. Next, each validator will be responsible for using the VRF to generate a value and a proof of the correctness of this value. The parameters used by this VRF are the following:

- The secret key sk_i. Each validator has its signature key pair created specifically for this process.
- A random value r associated with the corresponding *epoch* and which corresponds to the hash of the values obtained from the execution of the VRF of the blocks in the previous *epoch* up to $(N - 2)$ blocks. This dependency causes this pseudorandom value r to depend on the pseudorandomness of previous N blocks.
- The number of *slot* sl_k.

Once the VRF is executed we obtain the two values discussed above, the result we will call y which corresponds to the randomly generated value and its corresponding proof p. Since this process is deterministic, relying on blockchain data from previous *epochs*, other validators must be able to verify this, and they can do so thanks to the p proof generated together with the validator's public key. Next, the value y is compared to a threshold that has been previously defined in the protocol implementation. If the y value is less than this threshold, then this validator will become a viable candidate for block production for that *slot*. Finally, the validator will create a block and send it to the network along with (y, p). It should be noted that these allocations are initially secret and only known to the allocated validators and it is at the moment they publicly claim that *slot* to produce a block that they become public

5.1.2 Cardano

Regarding Cardano's cryptocurrency [CAR] in the same way that Polkadot does, it also divides by *slots* the creation of blocks. In their case these *slots* are called *slots* of Ouroboros. The Cardano protocol also divides time into

epochs and *slots*. Each *epoch* is divided into N *slots*. These parameters are fixed by the developers of the protocol and are encoded directly in the genesis block of the *epoch*. For example, during the Cardano Incentivized Testnet (ITN), an *epoch* was exactly 24 hours. During each *epoch* a special random process is run from the first *slot* of an *epoch* where each *staking pool* group and individual validator starts processing a secret and secure random number. This is where the VRF intervenes, whose inputs are as follows:

- The identifier of the *slot*.
- The validator's VRF signing key.
- *Nonce*, which is derived from a hash made of 2/3 blocks of the previous *epoch*.

Once the pseudorandom value has been generated with the VRF this value will allow the validator to create a new block whenever and wherever it meets the requirements to produce a new block in that *slot*. Thanks to this pseudorandom generation there is no possibility to find out in advance which node will be entitled to produce the next block. When the *epoch* ends, a special map is generated for the next *epoch*. This map contains a list of pairs like [(number1, group1), (number2, group2), ..., (numberN, groupN)]. Numbers are *slots* identifiers and groupN are *staking* validator identifiers. So, exactly in *slot* number N, a validator will create a block and issue it to the Cardano network. The entire network will then verify whether this validator had the right to produce a block in *slot* N . If positive, the block is verified and then accepted or rejected.

5.1.3 Algorand

Within the Algorand protocol [AL], at the core of the blockchain there is a fast Byzantine agreement protocol. However, this agreement is not made among all users of the network but is limited to a small committee of users chosen randomly for each round.

To decide which users participate in the agreement, a VRF is used. Each user in the Algorand network has a secret key SK and a verification key VK that is publicly known to everyone. Then what each user participating in the Algorand network does to decide if he is going to be part of the committee to execute the Byzantine agreement for block r is the following: Computes the function $Evaluate(SK, Qr)$ where Qr is a *seed* that is available to everyone in the system. The result of the function execution is a pair of values (Y, p) where Y is the random value generated with the VRF and p its proof. Once Y is obtained with its corresponding prrof p, it checks that Y is within a certain interval $[0, P]$ that depends on the user's participation in the system.

If the above check passes, then the user has a proof consisting of (Y, p) values that validates his commission membership for block r. Given (Y, p) and the user's verification key VK, anyone can verify that Y is the unique valid output and that it lies within a desired range, thus validating that the user who has VK has been chosen to be part of the r block committee. It should be noted that Algorand is committed to decentralization, and it is for this reason that they have published their source code and any details of the VRF implementation can be consulted at the following link:

`https://github.com/algorand/libsodium/tree/draft-irtf-cfrg-vrf-03`

The implementation they use for VRF is based on the standardization by Sharon Goldberg, Moni Naor, Dimitris Papadopoulos, Leonid Reyzin, and Jan Včelák [GRPV22].

5.2 Improvements in the DNS Protocol

After looking at the different cryptocurrencies that use VRFs, we'll explain how VRFs are used to add security to the DNS protocol. Specifically, we will see how the DNSSEC protocol, which adds security to DNS, incorporates this security through an implementation through VRFs called NSEC5. This proposal has been published by Dimitrios Papadopoulos, Duane Wessels, Shumon Huque, Moni Naor, Jan Včelák, Leonid Reyzin and Sharon Goldberg [G17]. The VRF algorithm that we use in the practical part together with ECDSA is the same one that they propose in their paper on NSEC5. Therefore, later in the practical part we will go into detail about its implementation. The purpose of NSEC5 is to guarantee two security properties:

1. Privacy against offline zone enumeration.
2. Integrity of zone content, even if an adversary compromises the authoritative DNS server responsible for answering DNS queries for that zone.

In order to achieve these objectives, the authors propose to include a new functionality by using a VRF based on elliptic curve cryptography. This VRF is also of special interest as it is being standardized by the IETF and used by several projects. In their paper they show how to integrate NSEC5 using EC-based VRF into DNSSEC. To do this, they take advantage of precomputation to improve performance and some optimizations at the DNS protocol level to shorten responses.

6 Practical Implementation

In this work, we have focused the practical side on the implementation of a VRF and its application in the world of cryptocurrencies, more specifically on the process used by cryptocurrency *wallets* for signature of the transactions. The specific case we will focus on is Bitcoin, but the solution can be applied to any cryptocurrency that uses ECDSA as a transaction signature. Today, many people know about the cryptocurrency Bitcoin (BTC) and all the importance it has gained since its appearance, but not everyone knows about the security and high level of cryptography behind it.

It should be noted that most transactions are carried out from *wallets* and are delicate operations that need to be very secure as there are BTC at stake which can mean a lot of money. It is for this reason that the cryptographic operations behind it must be secure and convey trust to both end users of the transaction. Currently, to carry out these transactions, what is used to make the Digital Signature is the ECDSA scheme. However, this scheme can present certain weaknesses because depending on how the random value used to sign is generated, the security of the cryptographic scheme can be reached and compromised.

Bad random value generation can lead to different problems. Some of them are well known and mechanisms to prevent them already exist, others are newer and are applied in more specific environments. In this work we will focus on an attack known as *Anti-Exfil* [AE] and consists in the fact that a *wallet* can leak the user's secret key by

carefully constructing its signatures. Since the commitment occurs during signature generation rather than key generation, even if the user verifies that their keys were generated from strong randomness, for example by generating their *seed* words by rolling dice and then importing them into multiple *wallets* to ensure they produce the same addresses, it is still possible to execute the attack.

The way the attack works is that the *wallet* generates a *nonce*, a unique one-time use number which in Section 4.3.1 we called k_e. This *nonce* makes up one half of an ECDSA signature, in a way that appears to be uniformly random, but actually isn't. There are many ways to do this: the *wallet* could produce *nonces* that are known to an attacker, allowing him to find the secret key used; could take a byte or two of the user's master *seed* in each *nonce*, allowing it to find each secret key with several hundred signatures, even from different keys; could subtly bias each *nonce* (even with a single bit) so that the keys could be recovered using the hidden numbers problem [BS17]. Regardless of the technique, these malicious signatures can be impossible to detect.

Currently, these *nonces* are being generated following the RFC6979 standard by creating deterministic *nonces*. However, using deterministic *nonces* cannot protect against biased attacks. Although RFC6979 guarantees that *nonces* are uniformly pseudorandom, it is impossible for a user to verify that it has actually been used since the generation of the *nonce* following RFC6979 involves the use of the private key, and in the case at hand, whoever must verify the *nonce* does not have access to this key. There are several known ways to prevent or detect such an attack:

- Users can re-derive their keys in a trusted *wallet*, calculate signatures using RFC6979, and determine whether the other *wallet* did the same or not. However, this practice is dangerous and impractical.
- The *wallet* could provide a zero-knowledge proof that it has successfully produced its *nonces*. This calculation is complex and computationally intensive.
- The *wallet* could do a multisignature between its own key and a user's key, which is derived from a password and need not be particularly secure. This would be an effective defense, but complicates the backup and recovery of coins, since *wallets* would have to implement the same protocol to recognize each other's coins, and losing the password would loss of funds.
- The user could make a multisignature between several *wallets* from different providers. This solution would require additional steps for the user when generating keys and signing.

It should be emphasized that all these measures end up being complex and do not really demonstrate their effectiveness against the main problem of *Anti-Exfil*. Currently, the solution that exists and has been implemented against this problem consists of the following: we will define two users, the *Hardware Wallet* which is responsible for generating the signature and the *Watch-Only Wallet* which is what asks for it and verifies it. Once the scenario is defined, the solution is to use the *sign-to-contract* [STC] to ask the *Hardware Wallet* to commit, using its *nonce* signature, with some random data provided by the *Watch-Only Wallet*. This causes the commit to re-randomize the *nonce*, removing any information it may have contained, and the *Watch-Only Wallet* discards the random data to ensure that no one can re-engineer it the result.

The steps performed by this *sign-to-contract* protocol are the following:

1. The *Watch-Only Wallet* chooses uniformly random data called b that it wants the *Hardware Wallet* to commit to and sends a hash of that data to the *Hardware Wallet*.
2. The *Hardware Wallet* calculates its *nonce* as it normally would; if using deterministic randomization ensure that the hash data of the *Watch-Only Wallet* is part of its randomization. Send this *nonce* R to the *Watch-Only Wallet*.
3. The *Watch-Only Wallet* then responds with the unfragmented random data b it wants to commit to.
4. The *Hardware Wallet* calculates $R + H(R||b) \cdot G$ and signs the transaction using this value as the *nonce*.

The *Hardware Wallet* must send its *nonce* R to the *Watch-Only Wallet* before receiving the random data b , otherwise it could choose R after so that the final *nonce* is biased. However, the *Watch-Only Wallet* must send a commitment b to the *Hardware Wallet* before it reveals R, or the *Watch-Only Wallet* could cheat it to create three signatures with the same R, but different b.

However, this protocol produces that instead of having two communications, one to send the transaction to be signed and another with the result of the transaction, there are four communications. For this reason, we will implement a protocol based on VRF that will allow the number of communications to be reduced, minimizing the risk of *Tampering* or *Spoofing*. Despite this, we will increase the size of the messages as there will be more values apart from the signature which will consist of the VRF proofs and the VRF value.

Our implementation consists of an ECDSA Signature together with a VRF for the generation of random numbers and their corresponding proofs. In addition, for key generation we will use BIP32 [BIP], since from an entropy we can generate a master key that will allow us to derive public and private keys in different *paths*. One *path* will be for the signatures and another for the VRF. Finally, those values that should be randomly generated within the VRF will be generated using the RFC6979 standard. The VRF we will use is the same one used by Chainlink and NSEC5 which was presented by Goldberg [G17] and is based on EC. Next, we will describe the different steps of the algorithm in order to generate the signatures and the integration of the VRFs within the signatures. It should be noted that for the notation of the algorithm lowercase characters have been used for numerical values and capital letters for points on the elliptic curve.

User A wants to receive the signature of the message m by user B:

1. User A generates a random value α.
2. User A requests the corresponding signature on the message m from user B. User A sends m, α, P where P is the public key of user B and its private key is d, both derived with BIP32.

User B receives m, α, P and makes the signature:

1. User B calculates $k_o = H(d, m)$.
2. Calculates $R_o = k_o \cdot G$.
3. User B proceeds to calculate a pseudorandom value and the corresponding proofs applying the VRF.
4. Derives a pair of public and private keys with BIP32 where PK is the public key and x is the private key.
5. Applies the $H_1(\alpha)$ function which returns a H value that corresponds to a point on the curve, since H_1 is a "hash to curve" function.
6. Calculates $\gamma = x \cdot H$.
7. Picks a random value $k \in (0 \ldots, n)$ with RFC6979.
8. Calculates $c = H_3(G, H, x \cdot G, x \cdot H, k \cdot G, k \cdot H)$ where H_3 is a function that performs the hash of the six points passed as a parameter and returns a integer of size l, in our case 256 bits.
9. Calculates $s' = k - c \cdot x \bmod n$.
10. User B calculates $t = H_2(f \cdot \gamma)$, where f is the cofactor of the curve on which we work and H_2 is a function that performs the hash of a point and reduces the number of bits to the required size, in our case 256 bits.
11. Finally user B has the proof $\pi = (\gamma, c, s')$, and the VRF value $t = H_2(f \cdot \gamma)$.
12. Calculates $k_e = (t + k_o)$.
13. Calculates $R = t \cdot G + R_o$, this step and the 12th are the modifications with respect to the original protocol that allows the VRF to be integrated.
14. Defines $r = x_R \bmod n$.
15. Calculates $e = H(m)$.
16. Calculates $s = (e + d \cdot r) \cdot k_e^{-1} \bmod n$. If $s = 0$ we return to step 3.
17. Send to user A the set of $R_o, R, s, (\gamma, c, s')$.

User A receives the signature which corresponds to the proofs and values R_o, R, s, (γ, c, s') and checks the signature and the values generated with the VRF:

1. Calculates $U = c \cdot PK + s' \cdot G$.
2. Calculates $H = H_1(\alpha)$.
3. Checks that γ belongs to the curve.
4. Calculates $V = c \cdot \gamma + s' \cdot H$.
5. Checks that $c = H_3(G, H, PK, \gamma, U, V)$.
6. User A calculates $t = H_2(f \cdot \gamma)$.
7. Calculates $R = t \cdot G + R_o$.
8. Defines $r = x_R \bmod n$.
9. Calculates $e = H(m)$.
10. Calculates $w = s^{-1} \bmod n$.
11. Calculates $u_1 = w \cdot e \bmod n$ and $u_2 = w \cdot r \bmod n$.
12. Calculates $Z = u_1 \cdot G + u_2 \cdot B$, where Z is a point on the curve (x_Z, y_Z).
13. Checks if $x_Z = r \bmod n$.
14. If all the steps have been checked and validated correctly, the signature is valid.

It should be emphasized that in order to use the ECDSA we had to make some adaptations in the protocol to be able to integrate the VRF. This change is due to the fact that in this protocol the random value k_e, also know as *nonce*, is always secret, and then we encounter the problem that we cannot generate proofs to prove a value that the other user cannot know. It is for this reason that thanks to the modifications made on the protocol we can combine the two functionalities. To make the ECDSA signature we used the elliptic curve of Bitcoin, the *secp256k1*. On the other hand, for the integration of BIP32 we had to use Base58 to integer conversion functions, since the private keys provided by BIP32 are in WIF (Wallet Import Format) format. Additionally, we implemented a decompression function, since public keys were made up of a single value instead of a dot. This format is that the value provided is the x coordinate of the point with the prefix 03 or 02 depending on whether it is one value of y or the other.

For the implementation we used the Python programming language and the Spyder interpreter together with the Miniconda development environments. In addition, we used Github to do code version control and to be able to perform synchronizations and backups more efficiently. The implementation code can be found in the following Github repository:

`https://github.com/AdrianCL0/TFG_VRF_BTC_ECDSA_schema`

Finally, regarding the results of the execution and functionality of the code we managed to integrate the VRF with the ECDSA protocol and that the signatures are correctly verified. In Appendix A.5 we can find the results of running the algorithm for different keys generated with BIP32 and different messages.

7 Conclusions

In this work we have studied what VRFs are and how they arose and we have analyzed the different implementations that have emerged over time as well as the cryptographic schemes and operations behind each one. Next, we looked at various protocols, applications, and cryptocurrencies that make use of VRFs and how they apply and integrate them into their schemes. Finally, we have implemented a protocol with Python that efficiently solves the *Anti-Exfil* problem in the ECDSA scheme by integrating a VRF based on elliptic curves.

On the other hand, with regard to the conclusions of the work, first of all it should be noted that we have been able to verify that nowadays the majority of VRF implementations are with elliptical curves, since, as we commented previously in Table 1, to obtain the same level of security as elliptic curves the RSA keys must be much larger. This difference means that when working with large numbers the calculations are slower and, in addition, the amount of storage required is also much higher.

Secondly, with regard to the main objectives defined at the beginning of the project, we have been able to achieve them all, both theoretical and practical. On the practical side, we were able to successfully implement a protocol with Python-based VRF that more efficiently solves the *Anti-Exfil* problem of Bitcoin ECDSA signatures. We measure this efficiency in the number of messages and not in size, since in this type of transaction the important thing is the number of communications. In addition, it should be emphasized that with this project we have been able to appreciate the importance, applicability and scalability of

VRFs. Nevertheless, when implementing them in security or cryptographic protocols, one must be careful how they are integrated. This fact is due to having to generate random values and their corresponding proofs that prove them, if we do it incorrectly we can be showing values that must remain secret, thus compromising the security of the scheme.

To finish, the next horizons that this project raises for us is the fact of publishing it in an official way to see what impact and what reactions it has in the Bitcoin community as a possible implementation for the solution of the *Anti -Exfil* problem. In addition, the project can be adapted to different types of signatures such as Schnorr or other types of cryptocurrencies simply by changing the path chosen with BIP32. This versatility gives it a lot of scalability in the cryptocurrency world.

ACKNOWLEDGEMENTS

First of all, I want to thank my project tutor, Jordi Herrera, for helping me throughout the research and project development processes. I felt very supported and listened to, because at all times, he motivated me and advised me what was most beneficial for the project and me. Secondly, and not least, I want to thank my family, for standing by me when I found everything very difficult to achieve and for advising and supporting me during all the stages of the project. Thirdly, to my colleague and great friend Guillem for giving me all the attention and help possible in certain aspects of the project to be able to make it as correct as possible. Finally, I want to thank my partner Anna, for the patience she has been able to have and the encouragement and ideas she has given me at every moment, so that I am motivated and focused to make this project possible.

REFERENCES

[RSA78] R.L. Rivest, A. Shamir, $ L. Adleman, (1978). *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems*, pp. 120-126, Commun. ACM, vol. 21, no. 2, Feb. 1978.

[VM85] R.L. Rivest, A. Shamir, $ L. Adleman, (1985). *Use of Elliptic Curve in Cryptography*, Exploratory Computer Science, IBM Research,P.O. Box 218, Yorktown Heights, NY 1985.

[GGM86] Oded Goldreich, Shafi Goldwasser, & Silvio Micali, (1986). *How to construct random functions*, pp. 120-126, Journal of the ACM, 33(4):792–807, 1986.

[NK87] Neal Koblitz, (1987). *Elliptic Curve Cryptosystems*, (1987) Mathematics of Computation. Volume 48. number 177, pp 203-208, January 1987.

[BG89] Mihir Bellare & Shafi Goldwasser, (1989). *New paradigms for digital signatures and message authentication based on non-interactive zero knowledge proofs*, In G. Brassard, editor, Advances in Cryptology- CRYPTO '89, volume 435 of Lecture Notes in Computer Science, pages 194–211. Springer-Verlag, 1990, 20–24 August 1989.

[MRV99] Silvio Micali, Michael Rabiny & Salil Vadhanz, (1999). *Verifiable Random Functions*, pp. 120-130, 40th Annual Symposium on Foundations of Computer Science, 1999.

[DY05] Yevgeniy Dodis & Aleksandr Yampolskiy, (2005). *A Verifiable Random Function with Short Proofs and Keys*, Vaudenay, S. (eds) Public Key Cryptography - PKC 2005. PKC 2005. Lecture Notes in Computer Science, vol 3386. Springer, Berlin, Heidelberg.

[VL13] Fré Vercauteren & K. U. Leuven, (2013).*Final Report on Main Computational Assumptions in Cryptography*, ECRYPT II, European Network of Excellence in Cryptology II, ICT-2007-216676.

[G17] Dimitrios Papadopoulos, Duane Wessels, Shumon Huque, Moni Naor, Jan Včelák, Leonid Reyzin & Sharon Goldberg. (2017).*Making NSEC5 Practical for DNSSEC*, Cryptology ePrint Archive, Paper 2017/099.

[BS17] Barak Shani, (2017). *Hidden Number Problems*, The University of Auckland. 2017.

[GM18] Vipul Goyal and Francisco Maturana, (2018). *Introduction to Cryptography*, Lecture 11: Key Agreement.

[SS20] Ken Schwaber & Jeff Sutherland, (2020). *The Scrum Guide. The Definitive Guide to Scrum: The Rules of the Game*, Scrum.Org, 2020.

[AB22] Alex Binué Sampedro, (2022). *Estudi de la blockchain de Polkadot*, Universitat Autònoma de Barcelona (UAB).

[GRPV22] S. Goldberg, L. Reyzin, D. Papadopoulos, & J. Vcelak, (2022). *Verifiable Random Functions (VRFs)*, Internet Research Task Force (IRTF).

[AE] Anti-Exfil: Stopping Key Exfiltration. [Online], Avalaible: *https://medium.com/blockstream/anti-exfil-stopping-key-exfiltration/-589f02facc2e*, (visited on 21.01.2023).

[AL] Algorand Releases First Open-Source Code of Verifiable Random Function. [Online], Avalaible: *https://www.algorand.com/resources/algorand-announcements/algorand-releases-first-open-source-code-of-verifiable-random-function*, (visited on 21.01.2023).

[BIP] The math behind BIP-32 child key derivation. [Online], Avalaible: *https://medium.com/@robbiehanson15/the-math-behind-bip-32-child-key-derivation-7d85f61a6681*, (visited on 21.01.2023).

[CAR] Generating randomness in Cardano: Verifiable Random Function. [Online], Avalaible: *https://cardanojournal.com/generating-randomness-in-cardano-verifiable-random-function-121*, (visited on 21.01.2023).

[CHL] Chainlink VRF: On-Chain Verifiable Randomness. [Online], Avalaible: *https://blog.chain.link/chainlink-vrf-on-chain-verifiable-randomness/*, (visited on 21.01.2023).

[STC] Sign-to-contract. How to timestamp with zero marginal cost. [Online], Avalaible: *https://blog.eternitywall.com/2018/04/13/sign-to-contract/*, (visited on 21.01.2023).

APPENDIX

A.1 Extend Length Through Tree

A problem we encounter when implementing the 1999 and 2005 algorithms is the fact that the input and output lengths are fixed at k-bits. What this limitation requires is that all input values must have k-bits and if they do not have them, their length must be adapted. In order to make this adaptation, what Micali, Rabiny and Vadhanz propose is to create a method by using a tree in which one more bit is added at each level. This representation can be seen in Figure 1. Regarding the operation of the tree we will not go into detail, since it is a complex and inefficient process that in practice is not implemented due to these complexities and inefficiencies.

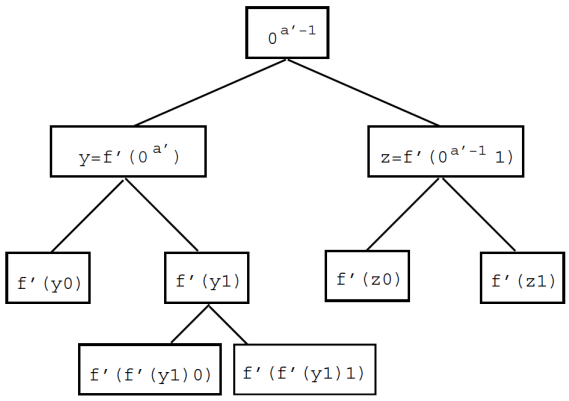

Fig. 1: Tree-shaped structure in order to increase the length [MRV99]

A.2 Extend Length Using Hash Function

The authors Yevgeniy Dodis and Aleksandr Yampolskiy propose to take advantage of the cryptographic hash functions properties in order to be able to adapt the length of the inputs. This method also allows you to do it in a more efficient and less complex way than with the tree method. The fact that we can use the hash function as a method to extend the size of values is due to the fact that we assume that they satisfy the following cryptographic properties:

- **Preimages resistance:** Given a hash value h, it must be difficult if not nearly impossible to find any message m such that $h = hash(m)$. This concept is related to the one-way function that hashes have since they are not invertible. Hash functions that do not have this property are vulnerable to preimaging attacks.
- **Second preimage resistance:** Given an input m_1, it must be hard to find an input m_2 different from m_1 such that $hash(m_1) = hash(m_2)$. This property is often also known as weak resistance to collisions. Functions that do not have this property are vulnerable to second preimage attacks.
- **Collision resistance:** It must be hard to find two different messages $m1$ and $m2$ such that $hash(m_1) = hash(m_2)$. This property is also known as strong collision resistance.

A.3 RSA Security Hardness Assumptions

To demonstrate the security of the cryptographic scheme, we must be able to demonstrate that there is some calculation whose efficiency is asymmetric since in one direction it is easy to calculate while in the other it has a high cost and complexity. However, this fact is usually a very difficult task to prove, so what we often do is to rely on hardness assumptions for our cryptographic applications. These assumptions are based on long-running problems for which no efficient algorithms are known and are therefore widely known to be difficult. Next we will express the assumption about the RSA [GM18]:

We define n as $p \cdot q$ where p and q are prime numbers. Then, as we defined before, we get that $\phi(n) = (p-1) \cdot (q-1)$. Next we choose $e \in 1, 2, , ...\phi(n) - 1$ so that it is relatively prime with $\phi(n)$. We then calculate d as $d \equiv e^{-1} \pmod{\phi(n)}$. Once the scheme is described, we can define the RSA assumption as:

- Given e and n it is difficult to calculate the e-th roots. More formally we can express it as:

RSA Assumption: We define $\prod n$ as the set of primes of length n, then we have:

$$\Pr\left[\begin{array}{c} p,q \leftarrow \Pi_n,\ p \neq q,\ N = pq, \\ e \leftarrow \mathbb{Z}^*_{\Phi(N)},\ x \leftarrow \mathbb{Z}^*_N \end{array} : A(x^e \bmod N) = x\right] \leq \text{negl}(n)$$

Another important aspect to consider is the fact of inverting the RSA algorithm, since if we have $x^e \pmod{n}$ we can calculate $x \equiv x^d \pmod{n} \equiv x^{e \cdot d} \pmod{n}$. The fact of being able to reverse the operation is because if we have $\phi(n)$ it is very easy to calculate d from e using the Euclidean algorithm. Therefore we can define that calculating $\phi(n)$ must be complicated. Then we can determine that the integer factorization method must be difficult and is a necessary condition for the assumption of RSA. However, today, it is not known for sure if this is a sufficient condition, since there could be other ways to calculate $\phi(n)$ that do not involve factorization. Finally, we can add to the RSA assumption that the RSA function is hard to invert when we have any additional information.

A.4 Diffie-Hellman Security Assumptions

As we mentioned with the RSA scheme, the security of cryptosystems is based on some calculation whose efficiency is asymmetric since in one direction it is easy to calculate while in the other it has a high cost and complexity. In this case, the security of the scheme is based on a set of assumptions about the Diffie-Hellman scheme. We remember that one of the main problems to be solved in this case was the discrete logarithm, therefore we can define the following assumptions [VL13]:

q-Diffie-Hellman Inversion Assumption (q-DHI): This assumption defines that there is no efficient algorithm that can calculate g^x given the values of $(g, g^x, ..., g^{xq})$.

q-Decisional Bilinear Diffie-Hellman Inversion Assumption(q-DBDHI): This assumption defines that there is no efficient algorithm that can distinguish $e(g,g)^{1/x}$ from random numbers despite having seen $(g, g^x, ..., g^{xq})$.

A.5 Results of the Execution of ECDSA Signatures with VRF

ECDSA:
Keys from path m/44'/10'/0'/0/0
Public Key: 02ffec56c8736c699b427691aef33644b4642cf094bbca9d58f06ed1eaebd3165f
Private Key WIF format: Kx2o5VCMyJSAcpq5LCiD4o86F8BpfAFVgwDKVDBJQRrZm1AXqBCp

VRF:
Keys from path m/44'/17'/0'/0/0
Public Key: 0335870f875dbb666fdc2a730039a67dd742f8835bcf1261a86c5580c2e016fe02
Private Key WIF format: L2CGB7pdk2XYNbR5ebKEepZvDHrAQZbN8Y4H3tiDCMRibtUM7att

Alice's public ECDSA key:
(115757351250351297023599834545792252529207263071997134313504216920487053432415,
85137358557283880708236127762603774765399049892229086247646851675756783015778)

Alice's public VRF key:
(24211212504317337465140951375952004896569759471901747637054608762821397642754,
30241116528185981386117041395397081851738188707379160220684591256248425344497)

Message: 2c9b9fd094bd6b16192e4b41fbff6d711fbd1dae9b808183571563a015485fe5

Signature: $S = (R, R_o, s)$:
[R=(233893332740200379647819228070094662694884266436018935064373136325730775149 54,
458411910033026803759864487962107667184279351569357042448708257326815895405 60),
R_o=(193845318576348888866093784451140829361399091677469341187120938026440019721 18,
112917893998461843020092123826783257784289733518519637410507181422489402179658),
s=3861156964091147600219971906134658176478090392972852054692508132994539537492 0]

VRF: $\pi = (\gamma, c, s')$:
[γ=(3545304531275172137410739748672435549601791272145212314242050918847262222694 8,
7863519277323745223886109200438559928023693100674092935309968702428104116602 2),
c=5425770586994874088383978440449624717021956475304983942371773645755990754748 3,
s'=1076005309318123370015914533829032443301019340652695037920808943079472114813 90]

————————————————-Signature has been verified correctly————————————————-

ECDSA:
Keys from path m/44'/10'/0'/0/1
Public Key: 026ae84a437814da29f31acd60646fb34b447d7af7b3112f907106178136914 12b
Private Key WIF format: Kyw6MW5eb6Ea1RfQt9UcWeYT8hiAFYDXwEBLnBfCckgYf1MaRR5Q

VRF:
Keys from path m/44'/17'/0'/0/1
Public Key: 03d8c7ec710a1acf04f1b987b28b514209171d7f8c961b792a98c3221061267f61
Private Key WIF format: KyafCNP31znK32NzYmxGHYpcdQWMKUxgTz3XXD6QjgYvkCmQ5zhp

Alice's public ECDSA key:
(48355583017046964631678020268274888243076783384257837712557846600865924858155,
79471022408486473964958929965547810598068685028718733439715173189172914362 38)

Alice's public VRF key:
(98052809719552405694820027517980781773760722818908654293081830804128268713825,
84702380404789486948421337640693720066543943772648720394157774482391748073137)

Message: 9b876d416424c070cfe3798a842da6866844e7b7bb1545937c0d7d853e909ea6

Signature: $S = (R, R_o, s)$:
[R=(516736921106143156809982948823996319450328716633022766243059365814389664 3144,
304451647263227559350387144399833759587769756485416754787385701496325510376 03),
R_o=(578352681140059973679598935243003970963167962519006165852254179823049634341 08,
956852102212599693459739763713916758361804279832093352488651043910728991882 74),

s=113772159199080592378077779528595232289830146185344739603762740203122946612 93]

VRF: $\pi = (\gamma, c, s')$:
[γ=(984108675861140131019015273058579828477224025727305106361501787731011024427 85,
805495692033741406910708308796485146813711231457753361857834043541580788451 2),
c=216748520077568400503761862143963964743927763625144831363640734835376643754 13,
s'=333208183752710186356402080426029397232824234453805492156331372969167784092 94]

————————————————-Signature has been verified correctly————————————————-

ECDSA:
Keys from path m/44’/10’/0’/0/2
Public Key: 0399439e3b183295d039d7d853ad27203754fae4815e216f0443b90c59a1aea1d2
Private Key WIF format: L3wbk3VKDTe5XBuJgWTzaf8sRiGr2GU26kV91Bh31MVd7ec7iRNa

VRF:
Keys from path m/44’/17’/0’/0/2
Public Key: 03d3f78241d9e94af71ae73e47be840febb4716e110bc634b402f321588e169dd0
Private Key WIF format: KyrixvmHnzBzAA2Uc7298NeHs5hriTq9Q6544qBDTj7ufciD9Sqt

Alice’s public ECDSA key:
(693233366556878599900027929042739399893236585993868389817035085632303854146 10,
103005206900623153567570191235190695365050700014657951031492372035447171258153)

Alice’s public VRF key:
(958753212784398234113223658768050601625749180609155489044819038217166666213 92,
480938036880870789166237290187126198107553753156270507233208900769455284097 47)

Message: c4e665a2fb388e4ad38f8f46ff7bb2946c93755f14597ecb0568aa56b4b530e7

Signature: $S = (R, R_o, s)$:
[R=(561933226832600376979806461133764849110824299092070990319015923059887465904 98,
141081191926697589709634154250906024162721466273964211869425326957631665725 77),
R_o=(758175908776698411300544587543440129768251136066012181710836525336126208493 74,
578308153656433741963429622858576123426109551426047895117579987057615295107 95),
s=716345845649971887452639464724087601504466437961148198998857096102310432370 68]

VRF: $\pi = (\gamma, c, s')$:
[γ=(228443646581595955657244927148228236763996856737915563978519981221376384194,
128650884467978727516515822379729053488251517661024792157668205886388222270 65),
c=371446257501655621333746518784007446615331821366245306803898175327333989405 78,
s'=642255517638899505498713183049418765227121429661278369923306942147464657323 30]

————————————————-Signature has been verified correctly————————————————-

————————————ALL 3 TRANSACTIONS HAVE BEEN SIGNED AND VERIFIED————————————