RSA and DSA : MTH4022 Project

Samuel Thompson

March 3, 2025

CONTENTS

1. INTRODUCTION

RSA encryption and DSA (the Digital Signature Algorithm) have been at the forefront of secure communications technology for decades. This paper aims to conduct a review of RSA and its weaknesses, detail DSA with a proof of correctness and finally give a discussion on the implementation of hash functions and DSA and of their potential weaknesses. We make some clarifying remarks.

1. NIST (National Institute for Standards and Technology) is a body that provides recommended practices for implementing cryptographic protocols.
2. Supplementary .ipynb files include code for all examples, algorithms and benchmarks.
3. We adopt the convention that we denote m *is congruent to* n *modulo* n by $m \equiv_p n$. If n is some function to be taken modulo q, we write $m \equiv_p (n \mod q)$.

2. RSA

The RSA encryption and decryption functions collectively referred to as the RSA cryptographic function[1], was the first example of a practical public key cryptographic function. The algorithm is attributed to Ron Rivest, Adi Shamir and Leonard Adleman in 1978 [RSA78] however an equivalent algorithm was discovered at GCHQ in 1973 but remained classified until 1997. More recently, lightweight cryptographic functions such as ECC [LNDPMS18] have largely replaced RSA in many applications.[MY17]. RSA still remains widely used in web authentication where a fast signature verification time is required [MY17, Case Study 1](see Section 3 for digital signatures).

2.1. THE RSA CRYPTOGRAPHIC FUNCTION

We introduce the steps for key generation, encryption and subsequent decryption below.

Algorithm 2.1 (RSA Key Generation)

1. ***Choose*** p *and* q *such that* $p \neq q$ *and* p, q *prime.*
2. ***Calculate*** $n = pq$*.*
3. ***Calculate*** $\Phi(n) = (p-1)(q-1)$ *where* Φ *is the Euler totient function.*
4. ***Choose*** *an integer* e *such that* $\gcd(\Phi(n), e) = 1$ *and* $1 < e < \Phi(n)$*.*
5. ***Calculate*** $d \equiv_{\Phi(n)} e^{-1}$*.*

This yields the public key (e, n) *and the private key* (d, n)*.*

Algorithm 2.2 (RSA Encryption)
To encrypt a plaintext message unit $m < n$*:*

$$c \equiv_n m^e$$

Using the public key (e, n)*.*

Algorithm 2.3 (RSA Decryption)
To decrypt a cyphertext message unit $c < n$*:*

$$m \equiv_n c^d$$

Using the private key (d, n)*.*

The RSA cryptographic function relies on the trapdoor function of modular exponentiation. A common attack involves factoring n to recover its prime factors p and q. This enables the calculation of the private key via steps 3-5 of Algorithm 2.1. In the next section we discuss methods that could be used to perform such an attack.

[1] In this paper we focus mainly on the RSA cryptographic function. A full cryptographic scheme must use techniques such as padding [BR94] to resist more subtle attacks than those we will discuss. This is known as *semantic security*. [Gol90].

2.2. FACTORISATION ALGORITHMS

For large numbers, an algorithm is said to take polynomial time if its running time scales polynomially with increasing input size. In computational complexity theory, polynomial time is considered the fastest scaling that is practical for algorithms. Technical definitions can be found in [Kal11].

The integer factorization problem has been known since Eratosthenes circa 200 BC yet a polynomial time algorithm has illuded mathematicians. It has been shown that *Shor's factorisation algorithm* runs in polynomial time on a quantum computer [MLLL$^+$12] [Sho97] however, due to the current limitations of quantum computers, this has not yet been successfully carried out at a scale that threatens RSA.

Algorithm 2.4 (Shor's Factorisation Algorithm)
We factorise $n \in \mathbb{Z}$.

1. ***choose*** *$y \in \mathbb{Z}$ such that $\gcd(n, y) = 1$.*
2. ***Calculate*** *the smallest possible integer r such that $y^r \equiv_n 1$.*
3. ***If*** *r is odd* ***repeat*** *from step 1 ensuring different choice of y.*
4. ***Calculate*** *$x = y^{\frac{r}{2}}$.*
5. ***Calculate*** *$\gcd(x - 1, x + 1)$, these are 2 factors of n.*

Note 2.4.1
It has been shown empirically in [MLLL$^+$12] that if remove step 3 for certain choices of y we find non trivial factors. A full description of how to implement this extension can be found in [Law15].

2.3. ENCODING

In order to encrypt messages in a non-integer plaintext alphabet we must encode this alphabet. We will use ASCII which is an 8 bit code on 128 characters. Most coding languages run ASCII (or a type of extension of ASCII called UTF-8 [Con10]) internally, so implementation is as straight forward as telling the compiler (or interpreter depending on coding language) not to convert an ASCII integer to its corresponding symbol.

Example 2.5
We want to convert the message

Hello world!

into ASCII as described:

$$72 \;\; 101 \;\; 108 \;\; 108 \;\; 111 \;\; 32 \;\; 119 \;\; 111 \;\; 114 \;\; 108 \;\; 100 \;\; 33$$

These decimal numbers correspond to the ASCII codes for each character in ***Hello world!****. Since ASCII uses 8-bit binary encoding, each of these decimal values can be represented as an 8-bit binary string[2], so for example 72 would be represented as:*

$$01001000$$

This is how the codeword is actually transmitted and worked with by the computer and it is in this form that ASCII is instantaneous. Note that how this representation is obtained in python is not actually equivalent to reading the raw binary in the computers memory as most modern programming languages assume you don't want to work with raw binary and so abstract it away.

[2] ASCII encodes 128 symbols which can be accomplished by a 7 bit code however as more computers began to work with 8 bit groups of data, ASCII was extended to 8 bits where the leading bit is used for parity checks. [Mac03, Ch.11].

2.4. Implementation of the RSA Cryptographic Function

2.4.1. Encryption

We now have the tools to implement RSA encryption. Assuming that we have the public key corresponding to our intended recipient (e, n), we proceed by Algorithm 2.2 using modular exponentiation. Calculating $c \equiv_n m^e$ directly is extremely slow for large numbers, we outline a faster method below.

Algorithm 2.6 (Left to Right Binary Modular Exponentiation)
Here, we calculate $c \equiv_n m^e$.

1. ***Initialise*** *$c \equiv_n m$ and convert e to its binary representation $(b_k b_{k-1} \dots b_0)_2$ where $b_k = 1$.*
2. ***Iterate*** *over each bit $b_i : i = k-1, \dots, 0$ in the binary representation of e from left to right starting from b_{k-1} and do:*
    1. ***Re-calculate*** *$c \equiv_n c^2$.*
    2. ***If*** *$b_i = 1$ do:*
        1. ***Recalculate*** *$c \equiv_n m * c$*
3. *Once this process terminates $c \equiv_n m^e$.*

For proof of correctness see [Knu97, Sec.4.6.3 Algorithm A]. For an explicit walk through of the algorithm see Appendix B.

A breakdown of empirical time comparisons for a standard RSA encryption setup for different techniques can be found in Appendix A Table 1. We will now implement RSA encryption in the following example using toy variables for clarity.

Example 2.7
*We choose to encrypt the message **Hello Johnny!** that we converted into ASCII in Example 2.5. Our recipient is using the public key $(e, n) = (17, 1363)$. We encrypt the first integer 72 using Algorithms 2.2 and 2.6 as follows.*

$$c \equiv_{1363} 72^{17} \equiv_{1363} 504$$

Performing this for all characters, we get the encrypted message below.

$$504, 852, 686, 686, 977, 582, 1162, 977, 791, 686, 1153, 818$$

It is worth noting that this specific implementation would be very susceptible to a factorisation attack due to the small key sizes. We will see more on this in Section 2.4.3.

2.4.2. Decryption

Like the encryption process, RSA decryption involves modular exponentiation except working with much larger numbers. It is standard practice in RSA decryption to leverage the Chinese remainder theorem (CRT) as was first described in [Tak98]. Instead of directly calculating $m \equiv_n c^d$ we calculate

$$m_p \equiv_p c^{d_p} \quad \text{and} \quad m_q \equiv_q c^{d_q} \quad : d_p \equiv_{p-1} d, d_q \equiv_{q-1} d$$

using Algorithm 2.6. We then recover m through the system

$$m \equiv_p m_p$$
$$m \equiv_q m_q$$

which can be solved using CRT. We implement Garner's algorithm for CRT as follows.

Algorithm 2.8 (Garner's Algorithm For CRT)
For a positive integer $M = \Pi_{i=1}^{t} m_i > 1 : m_i \in \mathbb{Z}$ with $gcd(m_i, m_j) = 1 \; \forall i, j \in \{1, \dots, t\}$ and residues of the target x modulo (m_i) denoted $(v_1, \dots, v_t)$ do the following.

1. ***Iterate*** *over* i *from 2 to* t *and do:*
    1. ***Set*** $C_i = 1$
    2. ***Iterate*** *over* j *from 1 to* $i - 1$ *and do:*
        1. ***Calculate*** $u \equiv_{m_i} m_j^{-1}$
        2. ***Re-calculate*** $C_i \equiv_{m_i} uC_i$
2. ***Set*** $u = v_1$ *and* $x = u$
3. ***Iterate*** *over i from 2 to* t *and do:*
    1. ***Re-calculate*** $u \equiv_{m_i} (v_i - x)C_i$
    2. ***Re-calculate*** $x = x + u \prod_{j=1}^{i-1} m_j$

At termination x *is the solution to the given system. For full details see [Knu97, Sec.4.3.2].*

Note that we compute the modular inverse using an implementation of the extended Euclidean algorithm. We also note that our private key in Algorithm 2.3 is extended to (p, q, d, n) when using this decryption method. A full list of decryption times on a standard RSA setup can be seen in Appendix A Table 2.

Example 2.9
We want to decrypt the cyphertext message

$$504, 852, 686, 686, 977, 582, 1162, 977, 791, 686, 1153, 818$$

from Example 2.7. Recall we used public key $(e, n) = (17, 1363)$*. To calculate* n*, we have that* $pq = n : p = 29, q = 47$ *was used. We implement Algorithm 2.1 to compute the private key.*

$$\begin{aligned} \phi(n) =& (p-1)(q-1) = 28 * 46 = 1288 \\ d \equiv& _{\phi(n)} 17^{-1} \equiv_{1288} 985 \end{aligned}$$

Note that we use an implementation of the extended euclidean algorithm to calculate d*. We now have the extended private key* $(p = 29, q = 47, d = 985, n = 1363)$*. To decrypt the first cyphertext message unit 504 we implement Algorithm 2.3 using the method described in the first part of this section.*

$$\begin{aligned} d_p &\equiv_{p-1} 985 \equiv_{28} 5 & \qquad d_q &\equiv_{q-1} 985 \equiv_{46} 19 \\ m_p &\equiv_p 504^{d_p} \equiv_{29} 14 & \qquad m_q &\equiv_q 504^{d_q} \equiv_{47} 25 \end{aligned}$$

Here we have implemented Algorithm 2.6, an explicit walk through of the process to find m_p *can be found in Appendix B. We recover* m *using CRT on the system:*

$$\begin{aligned} m &\equiv_{29} 14 \\ m &\equiv_{47} 25 \end{aligned}$$

Implementing CRT as in Algorithm 2.8 we recover:

$$m \equiv_{1363} 72$$

By performing this on all cyphertext message units we recover the ASCII code for our original message.

2.4.3. Performing a Factorisation Attack

We will we will perform a factorisation attack on the encrypted message in *'RSA-encrypted-1.txt'* that has public key $(e, n) = (289, 99157)$. We use Shore's algorithm 2.4 to factorise 99157. We find the factors

$$p = 229$$
$$q = 433$$

We can use these along with the public exponent to calculate the extended private key by Algorithm 2.1.

$$(p = 229, q = 433, d = 20449, n = 99157)$$

Implementing the decryption method from Section 2.4.2 and converting the ASCII to regular text we get the following message.

The fundamental problem of communication is that of reproducing at one point either exactly or ap-\nproximately a message selected at another point. Frequently the messages have meaning; that is they refer\nto or are correlated according to some system with certain physical or conceptual entities. These semantic\naspects of communication are irrelevant to the engineering problem. The significant aspect is that the actual\nmessage is one selected from a set of possible messages. The system must be designed to operate for each\npossible selection, not just the one which will actually be chosen since this is unknown at the time of design. (From C. Shannon, A Mathematical Theory of Communication, The Bell System Technical Journal,\nVol. 27, pp. 379–423, 623–656, July, October, 1948)

Note that $\backslash n$ is a control character that refers to a new line.

3. DSA

While RSA allows for sending encrypted messages to be read by the receiver only, it does not allow for the receiver to authenticate the sender. The Digital Signature Algorithm DSA is a method for the sender to create a unique signature for each message that the receiver can authenticate. In this section, we introduce a key component of DSA known as hash functions. We will define their role, step through the DSA process, and prove its correctness. Finally, we will explore how DSA and hash functions enable secure transactions in a basic cryptocurrency model and examine potential security vulnerabilities.

3.1. Hash Functions

Definition 3.1
A hash function H is a deterministic algorithm that takes as input a string in the class of all possible binary strings x and outputs a string $H(x)$ in the set of all possible binary strings of some fixed length denoted $|H|$.

It is important for cryptographic purposes that hash functions are *collision resistant*. We define a hash function H as collision resistant if it is infeasible for a polynomial time algorithm to find two strings $x \neq x' : H(x) = H(x')$. We note that such an x and x' certainly exist since the domain is much larger than the finite range. A hash function that is collision resistant is called a *cryptographic hash function*. [KL20, Chapter 6]

3.2. The Algorithm

The first step to formulating the DSA algorithm is parameter generation. These parameters do not change and any user in a given DSA system must use the same set of parameters. [KL20, Sec.13.5.3]

Algorithm 3.2 (Parameter Generation)

1. ***Choose*** *a cryptographic hash function* H*.*
2. ***Choose*** *a key length* L*.*
3. ***Choose*** *the modulus* N *such that* $N < L$ *and* $N \leq |H|$*. Note that if* $N < |H|$ *we only use the left most* N *bits of the hash.*
4. ***Choose*** *an* N *bit prime* q*.*
5. ***Choose*** *an* L *bit prime* p *such that* $q|(p-1)$*.*
6. ***Choose*** *a random integer* h *from the set* $\{2, \ldots p-2\}$*.*
7. ***Compute*** $g \equiv_p h^{\frac{p-1}{q}}$*. In the rare case that* $g = 1$ *try again with different* h*.*

This yields public parameters (p, q, g)*.*

Now for the key generation that describes how a user can generate their own public and private key in the system defined by parameters p, q and g.

Algorithm 3.3 (User Key Generation)

1. ***Choose*** *a random integer* x *from the set* $\{1, \ldots, q-1\}$*.*
2. ***Calculate*** $y \equiv_p g^x$

This yields the public key y *and the private key* x*.*

If a user with private key x wants to sign a message m they implement the following procedure.

Algorithm 3.4 (Signing)

1. ***Choose*** *a random integer* k *from the set* $\{1, \ldots, q-1\}$*.*
2. ***Compute*** $r \equiv_q (g^k \mod p)$*. In the rare case that* $r = 0$ *try again with different* k*.*
3. ***Compute*** $s \equiv_q k^{-1}(H(m) + xr)$*. In the rare case that* $s = 0$ *try again with different* k*.*

This yields the signature (r, s)*.*

Now that we are able to sign a message, we want to be able to verify a signature is from a particular sender. If we have received a message m signed (r, s) from a user with public key y we verify using the following.

Algorithm 3.5 (Signature Verification)

1. ***Verify*** $0 < r < q$ *and* $0 < s < q$*.*
2. ***Compute*** $w \equiv_q s^{-1}$*.*
3. ***Compute*** $v_1 \equiv_q H(m)w$*.*
4. ***Compute*** $v_2 \equiv_q rw$*.*
5. ***Compute*** $v \equiv_q (g^{v_1} y^{v_2} \mod p)$*.*
6. ***Verify*** $v = r$*.*

The message is valid if and only if $v = r$*.*

We say the DSA algorithm is correct if the user verifying a given message will only accept a genuine signature. We prove this using the notation above.

First, since $g \equiv_p h^{\frac{p-1}{q}}$ it follows that $g^q \equiv_p h^{p-1} \equiv 1$ by Fermat's little Theorem. Since $g > 0$ and q is prime, g has order q.

The signer computes $s \equiv_q k^{-1}(H(m) + xr)$. So we have:

$$\begin{aligned} k &\equiv_q H(m)s^{-1} + xrs^{-1} \\ &\equiv_q H(m)w + xrw \end{aligned}$$

Since g has order q we have:

$$\begin{aligned} g^k &\equiv_p g^{H(m)w} g^{xrw} \\ &\equiv_p g^{H(m)w} y^{rw} \\ &\equiv_p g^{v_1} y^{v_2} \end{aligned}$$

Finally, the correctness of DSA follows from:

$$\begin{aligned} r &\equiv_q (g^k \bmod p) \\ &\equiv_q (g^{v_1} y^{v_2} \bmod p) \\ &\equiv_q v \end{aligned}$$

3.3. IMPLEMENTATION

One application of DSA and cryptographic hash functions is securing cryptocurrency systems such as bitcoin. In this example, we leverage DSA and hash functions to add security to a simple cryptocurrency system.

Example 3.6
Our cryptocurrency system has 4 users, Alice, Bob, Charlie and Derek. Each user starts with a publicly agreed number of tokens. If Alice sends 5 tokens to Bob, this transaction will be recorded on a 'block'. All transactions (blocks) over history are recorded on a public ledger known as the block chain. In theory it is then possible to work out how many tokens each user has.

We address two main security concerns, we want to prevent transactions that are unauthorized by the sender and prevent changes being made to past blocks on the blockchain.

To prevent unauthorized transactions we implement DSA using toy variables for the sake of the example. Whenever the initial token distribution is agreed upon, public DSA parameters are generated.

$$p = 47681 \qquad q = 149$$

With p, q prime and $149|(47680)$ as required. We select $h \in \{2, \ldots 47679\}$ to be 18174.

$$g \equiv_p h^{320} \equiv_p 26140$$

We use a toy hash function that outputs 8 bits (q is an 8 bit prime). Our hash function H will encode the input string in ASCII, sum these integers then output the 8 least significant bits of this sum (with 0's at the start if the sum is less than 8 bits). Of course H is not collision resistant however this will suffice for illustrating this example.

Each user generates a public and private key as in Section 3.2. Alice wants to send 5 tokens to Bob. To do this Alice should sign the following standardised string.

$$\texttt{Alice||Bob||5}$$

First choosing $k \in \{1, \ldots, q-1\}$ *to be* $k = 100$*. She computes:*

$$r \equiv_{149} (g^{100} \mod p) \equiv_{149} 32499 \equiv_{149} 17$$

Using the extended Euclidean algorithm, she finds $k^{-1} \equiv_q 76$ *and hashes:*

$$H(\texttt{Alice||Bob||5}) = 65 + 108 + 105 + 99 + 101 + 124 + 124 + 66 + 111 + 98 + 124 + 124 + 53 = 1302$$

taking the last 8 bits we get the hash as follows

$$(00010110)_2 = 22$$

She calculates s*:*

$$s \equiv_{149} 76(22 - x_a r) \equiv_{149} 30$$

where $x_a = 70$ *is Alice's private key. Alice now creates a block in standardised form and appends it to the blockchain.*

$$\texttt{Alice||Bob||5||17||30||PrevHash=0000}$$

Note that the signature is contained in the block so anyone can verify that Alice sent the transaction. The final part

$$\texttt{PrevHash=0000}$$

can be overlooked for now since this is the first or genesis block. Now Bob performs signature verification as outlined in Section 3.2, he ensures that $0 < r, s < q$ *and calculates:*

$$\begin{aligned} w &\equiv_{149} 30^{-1} \equiv_{149} 5 \\ v_1 &\equiv_{149} H(\texttt{Alice||Bob||5})w \equiv_{149} 110 \\ v_2 &\equiv_{149} rw \equiv_{149} 85 \end{aligned}$$

Bob then calculates v*:*

$$v \equiv_{149} (g^{v_1} y_a^{v_2} \mod p) \equiv_{149} 17$$

Where y_a *is Alice's public key. Since* $v = r$*, Bob knows that the transaction is legitimate.*

Another transaction yields the second block:

$$\texttt{Bob||Charlie||3||33||136||PrevHash=11100111}$$

The element `PrevHash=11100111` *is determined by* $H(\texttt{Alice||Bob||5||17||30||PrevHash=0000})$*, that is the hash of the previous block. To prevent blockchain tampering, each block includes the hash of the previous block. If Derek modifies Alice's transaction, changing 5 to 10, the blockchain becomes:*

$$\begin{aligned} &\texttt{Alice||Bob||10||17||30||PrevHash=0000} \\ &\texttt{Bob||Charlie||3||33||136||PrevHash=11100111} \end{aligned}$$

Bob audits the blockchain by taking the hash of the first block.

$$H(\texttt{Alice||Bob||10||17||30||PrevHash=0000}) = 00010011$$

Bob checks and can see that this does not match `PrevHash=11100111` *on the second block. At this point Bob knows that the blockchain has been tampered with.*

Although this example is of a heavily simplified cryptocurrency, it illustrates the role that hash functions and the DSA algorithm play in real world cryptocurrency schemes. Full scale cryptocurrencies such as bitcoin utilise 2048 bit parameters and use hash functions such as SHA-256 that are believed to be collision resistant. The complications that arise whenever we try to decentralise the blockchain are extremely interesting and more on cryptocurrencies can be found in bitcoins foundational paper. [Nak08]

The DSA algorithm relies on the hardness of the discrete logarithm problem, we cannot find the private key x from the public key

$$y \equiv_p g^x$$

using classical methods assuming that we are using NIST recommended key sizes. In Example 3.6, we used a key length of 16 bits for p whereas a secure implementation requires p of at least 2048 bits. We also used a weak hash function which exposes vulnerability to another kind of attack.[KL20, Sec.6.4]

Example 3.7
Using the setup in Example 3.6, a transaction occurs where Alice sends Derek 10 tokens and we get the block:

$$\texttt{Alice||Derek||10||75||47||PrevHash=01001110}$$

Derek writes a script that iterates over the amount of tokens sent in the above block and hashes it in an attempt to find a collision. He finds that

$$H(\texttt{Alice||Derek||10||75||47||PrevHash=01001110}) = 10111011$$

and

$$H(\texttt{Alice||Derek||1000080||75||47||PrevHash=01001110}) = 10111011$$

Derek still has a problem, since Alice signed the message and not the whole block, the signature will now be invalid. So Derek must rewrite his algorithm to find a value i such that

$$H(\texttt{Alice||Derek||}i\texttt{||75||47||PrevHash=01001110}) = 10111011$$

and

$$H(\texttt{Alice||Derek||}i) = H(\texttt{Alice||Derek||10}) = 00011010$$

Even with our toy setup, it is computationally intensive to find such a value. Note that many real cryptocurrencies use SHA-256 where the odds of finding any collision, never mind a similar block, is in theory approximately 1 in 2^{128}. It should be noted that although this is widely assumed to be the case, there does not exist a proof that there is not a more efficient algorithm to find SHA-256 collisions. [KL20, Sec.6.4.1]

Another weakness of the DSA algorithm comes from poor implementation. If Alice signs 2 different messages, reusing the nonce k, an attacker can easily retrieve the private key x_a by the following procedure using notation as in Section 3.2.

$$s_1 \equiv_q k^{-1}(H(m_1) + xr_1)$$
$$s_2 \equiv_q k^{-1}(H(m_2) + xr_2)$$

Since $H(m_1)$, $H(m_2)$, s_1, s_2, r_1 and r_2 are known, an attacker can solve for k and recover x.

Due to advanced lattice based attacks, it is possible to recover k even if k is only *partially leaked*. This means that an attacker is able to retrieve some of the bits that make up k. This could be achieved through various means such as exploiting a random number generator that is not totally random or attacking the hardware that is used to generate values for k. We outline a simplified version of one of these attacks in the following example.

Example 3.8
We use the setup of Example 3.6. Alice adds 2 blocks to the block chain. The first sends 5 tokens to Bob, the second sends 10 tokens to Charlie as follows.

$$\texttt{Alice||Bob||5||99||108||PrevHash=01011010}$$
$$\texttt{Alice||Charlie||10||4||84||PrevHash=11101011}$$

Derek has ascertained through manipulation of the hardware Alice is using to generate k, that Alice is generating 8 bit k values and that the first 4 bits of k_1 used to sign the first transaction are 0110 and the first 4 bits of k_2 used to sign the second message are 0101. Derek then models k_1 and k_2 as follows:

$$k_1 = k_1' + \delta_1 = (01100000)_2 + \delta_1 = 96 + \delta_1$$
$$k_2 = k_2' + \delta_2 = (01010000)_2 + \delta_2 = 80 + \delta_2$$

Derek manipulates the two equations that where used to calculate s_1 and s_2 as follows. We say x_a is Alice's private key, H_1 and H_2 are the hashes of `Alice||Bob||5` and `Alice||Charlie||10` respectively.

$$s_1 \equiv_q (k_1 + \delta_1)^{-1}\big(H_1 + xr_1\big)$$
$$s_2 \equiv_q (k_2 + \delta_2)^{-1}\big(H_2 + xr_2\big)$$

Rearranging for x_a:

$$x_a \equiv_q \frac{s_1(k_1' + \delta_1) - H_1}{r_1} \tag{3.4.1}$$
$$x_a \equiv_q \frac{s_2(k_2' + \delta_2) - H_2}{r_2} \tag{3.4.2}$$

Equating the two equations:

$$\frac{s_1(k_1' + \delta_1) - H_1}{r_1} \equiv_q \frac{s_2(k_2' + \delta_2) - H_2}{r_2}$$

Expanding and isolating δ_1, δ_2:

$$s_1 r_2 \delta_1 - s_2 r_1 \delta_2 \equiv_q \big(H_1 r_2 - s_1 k_1' r_2 - H_2 r_1 + s_2 k_2' r_1\big)$$

Plugging in the values that are openly available to him and values calculated for k_1' and k_2':

$$134\delta_1 - 121\delta_2 \equiv_{149} 110 \tag{3.4.3}$$

It is at this stage that in a full scale DSA system (in which Derek would require as many signed messages with partial k leakage as possible) that lattice reduction techniques would be used to ascertain k and thus the private key. For details, see [HGS00]. Since our system is small Derek can implement the following.

Derek knows that δ_1 and δ_2 are both 4 bit numbers so they both must be between 0 and 15 inclusive. He writes a script that tests which possible combinations satisfy equation 3.4.3. He finds that $\delta_1 = 2$ and $\delta_2 = 5$ is a possible combination and so is $\delta_1 = 7$ and $\delta_2 = 13$. Derek can now take the two possible values for $k_1 = k_1' + \delta_1$ and plug them into Equation 3.4.1. He will then have two possible private keys for Alice. He can then sign an arbitrary message using each private key and verify them using Alice's public key. Whichever message is correctly verified has used Alice's real private key and thus Derek can forge Alice's signature.

4. Summary and Conclusions

In this paper, we outlined the RSA cryptographic function and its implementation, including a discussion of factorisation attacks. While such attacks are currently infeasible for full scale RSA schemes, advancements in quantum computing may change this in the future. We then introduced the Digital Signature Algorithm (DSA), proved its correctness, and examined its role alongside hash functions in securing cryptocurrency transactions. Finally, we highlighted how poor implementation of these protocols can compromise transaction integrity. Future work could explore the security requirements of a more complete cryptocurrency system and consider measures needed for post-quantum cryptographic resilience.

REFERENCES

[BR94] Mihir Bellare and Phillip Rogaway. Optimal asymmetric encryption—how to encrypt with rsa. In *Advances in Cryptology—EUROCRYPT '94*, pages 92–111. Springer, 1994.

[Con10] The Unicode Consortium. *The Unicode Standard*. The Unicode Consortium, Mountain View, California, US, 6.0 edition, 2010.

[Gol90] S. Goldwasser. The search for provably secure cryptosystems. In *Cryptology and Computational Number Theory*, volume 42 of *Proceedings of the 42nd Symposium in Applied Mathematics*. American Mathematical Society, 1990.

[HGS00] N. A. Howgrave-Graham and N. P. Smart. Lattice attacks on digital signature schemes. *Journal of Cryptology*, 2000. Communicated by: A. Menezes.

[Kal11] Burt Kaliski. *Subexponential Time*, pages 1267–1267. Springer US, Boston, MA, 2011.

[KL20] J. Katz and Y. Lindell. *Introduction to Modern Cryptography*. Chapman & Hall/CRC Cryptography and Network Security Series. CRC Press, 2020.

[Knu97] D.E. Knuth. *The Art of Computer Programming: Fundamental Algorithms, Volume 1*. Pearson Education, 1997.

[Law15] Thomas Lawson. Odd orders in shor's factoring algorithm. *Quantum Information Processing*, 14(3):831–838, January 2015.

[LNDPMS18] Carlos Andres Lara-Nino, Arturo Diaz-Perez, and Miguel Morales-Sandoval. Elliptic curve lightweight cryptography: A survey. *IEEE Access*, 6, 2018.

[Mac03] D.J.C. MacKay. *Information Theory, Inference and Learning Algorithms*. Cambridge University Press, 2003.

[MLLL+12] Enrique Martin-Lopez, Anthony Laing, Thomas Lawson, Roberto Alvarez, Xiao-Qi Zhou, and Jeremy L. O'Brien. Experimental realisation of shor's quantum factoring algorithm using qubit recycling. *Nature Photonics*, 6(11):773–776, 2012.

[MY17] Dindayal Mahto and Dilip Kumar Yadav. Rsa and ecc: A comparative analysis. *International Journal of Applied Engineering Research*, 12(19):9053–9061, 2017.

[Nak08] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Online, 2008. `https://bitcoin.org/bitcoin.pdf`.

[RSA78] Ronald L. Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[Sho97] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26(5):1484–1509, October 1997.

[Tak98] Tsuyoshi Takagi. Fast rsa-type cryptosystem modulo pkq. In Hugo Krawczyk, editor, *Advances in Cryptology — CRYPTO '98*, pages 318–326, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

A. Encryption/Decryption Method Benchmarks

The following tables outline empirical time comparisons for different methods of modular exponentiation. The setup in Table 1 models a typical, full scale RSA encryption process. Table 2 does the same for decryption. Note that Table 2 does not contain timings for *Naive Modular Exponentiation* as it takes too long to test in a reasonable time for this setup. We refer to the method outlined in Section 2.4.2 as *Garner's CRT Exponentiation*.

Method	Mean	Standard Deviation	Loops per Test	Number of Tests
Naive Modular Exponentiation	$9.44ms$	$547\mu s$	100	7
Modular Exponentiation Algorithm	$76.2\mu s$	$1.58\mu s$	10000	7
Garner's CRT Exponentiation	$839\mu s$	$21.7\mu s$	1000	7

Table 1: RSA encryption technique comparison performing $m^e \mod n$ with a random 8 bit m (ASCII is 8 bit), random 2048 bit n (NIST standard public modulus) and $e = 65537$ (commonly used public exponent).

Method	Mean	Standard Deviation	Loops per Test	Number of Tests
Naive Modular Exponentiation	-	-	-	-
Modular Exponentiation Algorithm	$31.5ms$	$1.25ms$	10	7
Garner's CRT Exponentiation	$10.7ms$	$157\mu s$	100	7

Table 2: RSA decryption technique comparison performing $c^d \mod n$ with a random 2048 bit c (standard encrypted word size using 2048 bit modulus), random 2048 bit n (NIST standard public modulus) and a random 2048 bit d (standard private key size).

B. Explicit Modular Exponentiation

We explicitly calculate $m_p \equiv_{29} 504^5$ from Example 2.9 according to Algorithm 2.6.

1. We initialise $m_p \equiv_{29} 504 \equiv_{29} 11$ and note that $5 = (b_2 b_1 b_0)_2 = (101)_2$.
2. We now start with $b_1 = 0$. So we set $m_p \equiv_{29} 11^2 \equiv_{29} 5$.
3. For $b_0 = 1$, we first set $m_p \equiv_{29} 5^2 \equiv_{29} 25$ and since $b_0 = 1$, we set

$$m_p \equiv_{29} m(m_p) \equiv_{29} 504(25) \equiv_{29} 14$$

We conclude that $m_p \equiv_{29} 14$.

Word Count with Appendix : 3246