Implementation and Analysis of an Elliptic Curve Cryptography System

Student Name: T. Butterfield

Supervisor Name: M. Bordewich

Submitted as part of the degree of BSc Computer Science to the

Board of Examiners in the Department of Computer Sciences, Durham University

April 15, 2021

Abstract —

Context/Background - Diffie and Hellman invented the idea of establishing a secret shared key using a public channel, RSA was the first implementation of this idea. ECC is a more efficient version of RSA, requiring both smaller key sizes and less computation to achieve the same security level.

Aims - The aim of this project was to create a working Elliptic Curve Cryptography system which facilitated secure communication over an open channel, and to analyse this system in terms of security and efficiency. This security analysis includes both theoretical difficulty and real-world security of the system, while the efficiency analysis compares ECC to RSA as well as a selection of elliptic curves to each other.

Method - Methods used in the system include Elliptic Curve point multiplication, achieved through repeated doubling and addition. Euclid's extended algorithm is also used for modular division.

Results - The result of this project was an end-to-end encrypted messaging system using ECC for key establishment and digital signatures. Results show that my ECC system is orders of magnitude more efficient than an RSA system of equal security level.

Conclusions - The system allows for secure and efficient communication due to smaller key sizes, making it well suited for use in mobile and low power devices. However, secure implementations are imperative for production usage to prevent side-channel attacks.

Keywords — Cryptography, Encryption, Elliptic Curve Cryptography (ECC), Elliptic Curve Diffie-Hellman (ECDH), Elliptic Curve Digital Signature Algorithm (ECDSA), Elliptic Curve Discrete Logarithm Problem (ECDLP)

I INTRODUCTION

A Background

Any cryptography system tries to solve two kinds of security problems: privacy and authentication. Imagine that Alice and Bob are trying to communicate, while Eve, the eavesdropper, intercepts and tries to read their messages. In Elliptic Curve Cryptography, privacy is solved with the Elliptic Curve Diffie-Hellman (ECDH) key exchange algorithm, which allows Alice and Bob to communicate securely without Eve being able to read their messages. Authentication is solved with the Elliptic Curve Digital Signature Algorithm (ECDSA), which allows Bob to verify that Alice was truly the sender of a message, and ensures that Eve cannot impersonate Alice.

These two algorithms are very similar to their RSA counterparts. While RSA squares and multiplies numbers repeatedly in order to achieve exponentiation, ECC doubles and adds points on a curve repeatedly to achieve multiplication.

An elliptic curve is defined by an equation of the form $y^2 = x^3 + ax + b$ and has a special property that the line joining any two points on the curve intersects at exactly one other point. The red line in Figure 1 and Figure 2 is a very simple elliptic curve with the equation $y^2 = x^3 - 3x + 3$.

With reference to Figure 1, in order to add two points A and B together, the straight line which passes through both points is drawn. The only other point at which the line intersects the curve is then found. This point C is reflected in the x-axis to form D. The point D is the sum of points A and B in Figure 1, so $A + B = D$.

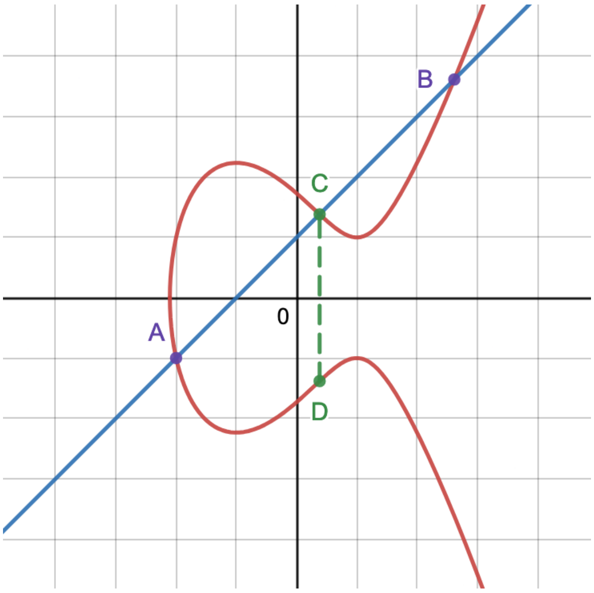

Figure 1: Point Addition $(A + B = D)$

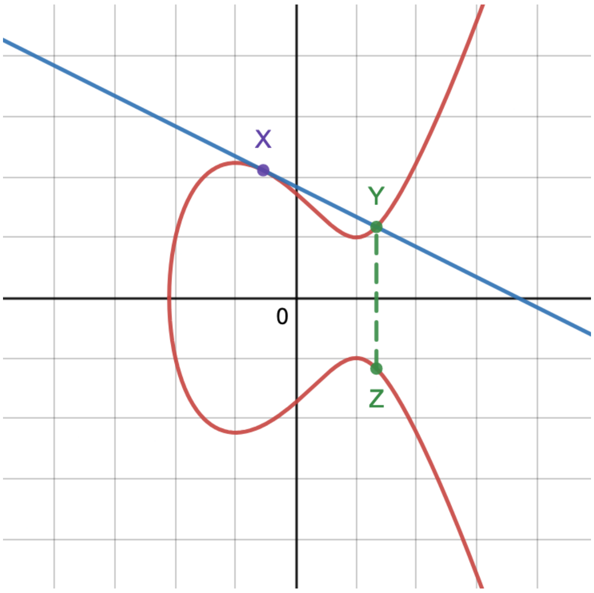

Figure 2: Point Doubling $(2 \cdot X = Z)$

The same principle applies for doubling a point, as Figure 2 illustrates. If instead $A = B = X$, then the line passing through both points is instead a tangent to the curve at X. The only other point at which the line intersects the curve in then Y in Figure 2. When reflected in the x-axis this becomes Z. The point Z is the double of point X in Figure 2, so $2 \cdot X = Z$. This allows any point on the curve to be multiplied by an integer, using repeated doubling and adding.

Although ECC is perhaps not as well-known as RSA, it is much more efficient. The problem with RSA is that for it to be considered secure by modern standards, the keys used must be at least 2048 bits in size. This makes computation slow and battery draining for mobile devices. However, an equivalent level of security can be achieved using ECC with a key which is only 224 bits in size, which means less computation, less data to be transferred, and greater efficiency for mobile devices. This is because the integer factorisation problem, on which the security of RSA is based, can be solved in sub-exponential running time. Whereas the Elliptic Curve Discrete

Logarithm Problem (ECDLP), on which the security of ECC is based, can only be solved in fully exponential running time. Meaning that, in order to increase security levels, RSA key sizes must increase exponentially while ECC key sizes can increase linearly.

ECC is a very efficient method for modern public-key cryptography, but there are many different elliptic curves which can be used in an ECC system and they each vary in both speed and strength. As with any public-key cryptosystem, each user has both a public key and a private key. The public key is made public while the private key is kept private, as the names suggest. The strength of any system like this can be measured by how difficult it would be to find the user's private key, given their public key. In ECC this is known as the Elliptic Curve Discrete Logarithm Problem (ECDLP).

B Objectives

The aim of this project was to implement an Elliptic Curve Cryptography system, and to then analyse this system. This system allows for secure communication over a public channel between users. The analysis then compares both the efficiency and security aspects of the system. This means comparing it to a traditional RSA system, as well as looking at the differences between a selection of elliptic curves.

There were a total of nine objectives for this project, split equally into three sections: basic, intermediate, and advanced. The first basic objective was to create a basic working ECC system which computed the essential functions required. The second and third basic objectives were to create a client application that can conduct ECDH over a network connection, and to create a UI allowing a user to easily make use of the system and to decide what level of complexity is revealed to them.

The intermediate objectives revolved around improving the functionality and security of the system, with the first objective being to add secure random private key generation to the system. The other intermediate objectives were to implement the ECDSA and to enable the exchange of secure signed files via the UI.

The first advanced objective was to make improvements to the UI and to reveal the workings of the system. The penultimate objective was to analyse the code for efficiency, including how the time required scales with the curve and key size, making any necessary changes to the system to improve the efficiency. The final objective was to analyse the code for vulnerabilities, such as side-channel attacks, and to fix any such weaknesses.

II RELATED WORK

Diffie and Hellman (1976) defined cryptography as the study of mathematical systems for solving two kinds of security problems: privacy and authentication. A privacy system prevents the extraction of information by unauthorised parties from messages transmitted over a public channel, assuring it is read only by the intended recipient. An authentication system prevents the unauthorised injection of messages into a public channel, assuring the legitimacy of the sender. They also proposed that it was possible to develop systems in which two parties communicating solely over a public channel and using only publicly known techniques could create a secure connection. This was the first public discovery of public-key cryptography, although it was later revealed that researchers at GCHQ had already made this discovery in 1969.

While Diffie and Hellman presented the concept of a public-key cryptosystem, they did not present any practical implementation of such a system. Rivest, Shamir, and Adleman (1978) presented a new encryption method which did just that, it was the first public implementation of public-key cryptography. This method had the novel property that publicly revealing an encryption key did not thereby reveal the corresponding decryption key. This is now known as RSA public-key cryptography, although it was later revealed that a researcher at GCHQ had described an equivalent system in 1973.

Miller (1986) and Koblitz (1987) discussed an analogue of the Diffie-Hellman key exchange protocol based on elliptic curves over finite fields which used the multiplicative group of a finite field. The advantage of this system was that it appeared to be immune from attacks of the style of Adleman (1979) and Miller (1986). This was the foundation of Elliptic Curve Cryptography (ECC).

A Attacks

The elliptic curve parameters for cryptographic schemes should be carefully chosen in order to resist all known attacks on the ECDLP.

A.1 Exhaustive Search

The most naive algorithm for solving the ECDLP is exhaustive search whereby one computes the sequence of points $P, 2P, 3P, 4P, ...$ until Q is encountered, where Q is the user's public key. The running time for this approach is approximately n steps in the worst case and $n/2$ steps on average, where n is a property of the curve. Therefore, exhaustive search can be circumvented by selecting elliptic curve parameters with n sufficiently large to represent an infeasible amount of computation (e.g., $n \geq 2^{224}$) (Hankerson et al. 2003).

A.2 Pohlig-Hellman and Pollard's rho algorithm

The best general-purpose attack known on the ECDLP is the combination of the Pohlig-Hellman algorithm and Pollard's rho algorithm, which has a fully-exponential running time of $O(\sqrt{p})$, where p is the largest prime divisor of n. To resist this attack, the elliptic curve parameters should be chosen so that n is divisible by a prime number p sufficiently large so that $\sqrt{p}$ steps is an infeasible amount of computation (e.g., $p > 2^{227}$). If, in addition, the elliptic curve parameters are carefully chosen to defeat all other known attacks (e.g. Isomorphism attacks), then the ECDLP is believed to be infeasible given the state of today's computer technology (Hankerson et al. 2003).

B Implementation Issues

In 2010, it was discovered that Sony had made a basic and critical mistake in the implementation of their Elliptic Curve Cryptography system. In general, a digital signature in cryptography allows a user to verify that a file they downloaded was definitely created by a particular person/company. Sony's PlayStation 3 console uses digital signatures to verify that a game is authentic, this prevents users from running their own programs on the console. The process of digital signature generation in ECC involves picking a random integer k to be used later in the next step of the process. It is imperative that k is truly random and not predictable in any way. However, Sony wrote their own software to generate k, which used a constant number for each signature. This allowed hackers to compute their private key, from the public key, using simple algebra (Hotz 2010). Once hackers acquired Sony's private key, they could use it to generate their own digital signatures which impersonated Sony. This meant they could run their own programs on the console since the files appeared to be authentic. The lesson learnt here, in terms of my own implementation, is that a secure implementation of an ECC system requires a good random number generator.

III SOLUTION

This section presents the solutions to the problems in detail. Included here are details of the architecture and design of the system (Section A), the algorithms (Section B) and tools used (Section C), as well as an outline of the implementation issues faced (Section D), and finally a demonstration to verify and validate the system (Section E).

A Architecture and Design

At a high level, my system is a client-server network designed to allow multiple instances of a client program to connect to a single server program. The clients can communicate with each other, via the server, using end-to-end ECC encryption. In the description and explanation of my system I will personalise the participants; Alice and Bob want to communicate while Eve, the eavesdropper, intercepts and tries to read their messages. Figure 3 illustrates the protocol for Alice to send a message to Bob:

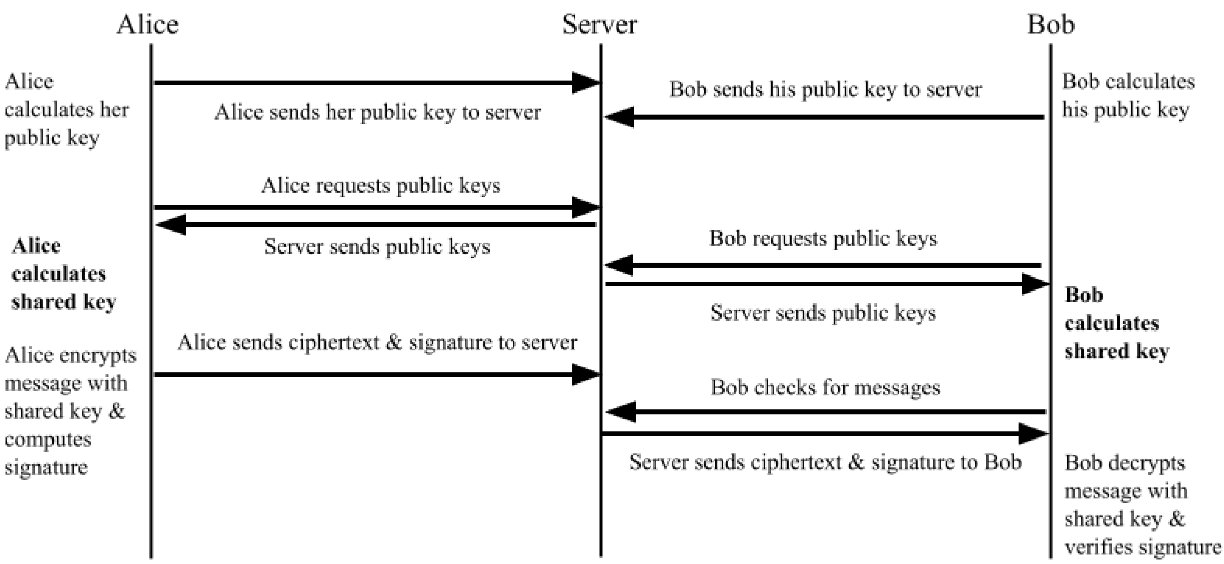

Figure 3: Message Exchange Diagram

A message here can be either a text message or a file from Alice's device. The server stores a list of the public keys of every client who is connected, and will send this to any client who requests it in order for them to establish a shared key with other clients. The server also holds the encrypted data and the accompanying signature for messages which have yet to be requested by the recipient. After the recipient has asked the server if there are any messages waiting for them, the server has sent the relevant message(s), and the recipient has decrypted and verified the message(s), the server then deletes the message(s). The only data stored by the client is a list of their shared keys with other clients. As shown in Figure 3, all of the computation is done by the client and the server does not perform any calculations; it only receives, stores, and sends client data. There are two ECC specific algorithms performed by each client in the exchange shown in Figure 3, namely the Elliptic Curve Diffie-Hellman (ECDH) key exchange algorithm and the Elliptic Curve Digital Signature Algorithm (ECDSA).

B Algorithms

This section outlines the algorithms which are used by my system and how I implemented them. The two ECC specific algorithms mentioned previously (ECDH and ECDSA) differ from their RSA counterparts (DH and DSA) only in that the operation of exponentiation of an integer is instead replaced by the operation of multiplying a point on an elliptic curve. The naive approach to this of repeatedly adding a point G to itself k times in order to calculate kG would take prohibitively long. Therefore I implemented an adapted version of the repeated squares method (squaring and multiplying integers) used by RSA in order to calculate large exponents efficiently. The method I implemented uses doubling and adding of points on an elliptic curve in order to achieve multiplication of a point. While the naive approach would take $O(k)$ time to calculate kG, this approach takes $O(log(k))$ time, making it practical for my system since k will be a very large number (of the order 2^{250}). An elliptic curve (Section B.1), along with the operations of adding (Section B.2) and doubling (Section B.3) points on the curve, are formally defined below.

B.1 Elliptic Curve

Let p be a prime number, and let F_p denote the field of integers modulo p. An elliptic curve E over F_p is defined by an equation of the form:

$$y^2 = x^3 + ax + b, \tag{1}$$

where $a,\ b \in F_p$ satisfy $4a^3 + 27b^2 \neq 0 \pmod{p}$.

B.2 Point Addition

Let $P = (x_1, y_1) \in E(F_p)$ and $Q = (x_2, y_2) \in E(F_p)$, where $P \neq Q$. Then $P + Q = (x_3, y_3)$, where:

$$x_3 = \lambda^2 - x_1 - x_2, \quad y_3 = \lambda(x_1 - x_3) - y_1, \quad and \quad \lambda = \frac{y_2 - y_1}{x_2 - x_1}. \tag{2}$$

B.3 Point Doubling

Let $P = (x_1, y_1) \in E(F_p)$. Then $P + P = 2P = (x_3, y_3)$, where:

$$x_3 = \lambda^2 - 2x_1, \quad y_3 = \lambda(x_1 - x_3) - y_1, \quad and \quad \lambda = \frac{3x_1^2 + a}{2y_1} \tag{3}$$

(Hankerson et al. 2003, López & Dahab 2000).

In my system when I want to multiply a point G by an integer k, I take the point G and double it $\lfloor log_2(k) \rfloor$ times, storing each new point in an array where index i holds $2^i \cdot G$. The final value is then calculated by adding the points in the array whose index corresponds to a 1 in the binary expansion of k. My system actually implements this using bitwise operations on k for efficiency, however it is more intuitive to imagine it as a binary expansion and an array of equal length as described. That is how I implement point multiplication using point addition and point doubling. Things are complicated further however due to the fact that an elliptic curve is defined over a field of integers modulo p (B.1), meaning that these operations are all carried out in modulo p and can only involve integers. Since the equations for both point addition (B.2) and point doubling (B.3) involve division, it is clear that a naive implementation of division cannot be used here since it would produce non-integer results. Therefore I implemented the Extended Euclidean algorithm (EEA) in order to ensure integer results of these division calculations. For example, in order to calculate the gradient $\lambda = \frac{y_2-y_1}{x_2-x_1}$ in point addition (B.2), I use the EEA to calculate $(x_2 - x_1)^{-1}$ mod p and then multiply this by $(y_2 - y_1)$. This ensures that λ is an integer, and therefore that the coordinates (x_3, y_3) as defined in B.2 are also integers.

Once I had a function for efficiently computing point multiplication, I was able to move on to the functions for ECDH and ECDSA.

B.4 ECDH

The *Elliptic Curve Diffie-Hellman* key agreement protocol allows Alice and Bob to securely agree upon the value of a point on an elliptic curve, although neither of them initially knows the value of the point. This point becomes their shared key, and the algorithm for generating it is formally defined below.

B.5 Shared Key Establishment

Alice and Bob must agree on a finite field F_p, an elliptic curve E/F_p, and a base point $G \in E(F_p)$ of (prime) order n. Both Alice and Bob have a key pair (d, Q) consisting of a private key d (a randomly selected integer less than n) and a public key $Q = d \cdot G$.

0. Let (d_A, Q_A) be the key pair of Alice and (d_B, Q_B) be the key pair of Bob.
1. Alice computes $K = (x_K, y_K) = d_A \cdot Q_B$
2. Bob computes $L = (x_L, y_L) = d_B \cdot Q_A$
3. Since $d_A \cdot Q_B = d_A \cdot d_B \cdot G = d_B \cdot d_A \cdot G = d_B \cdot Q_A$
4. Therefore $K = L,$ hence $x_K = x_L$
5. The shared secret is x_K

Since it is practically impossible to find the private key d_A or d_B from the public key Q_A or Q_B, it is not possible for Eve to obtain the shared secret (Jurišic & Menezes 1997, Anoop 2007, Brown 2009, Silverman 2009).

In my implementation of this algorithm, all clients are automatically using the same base point which lies on the same elliptic curve, defined over the same finite field. This is because every client is an instance of the same program and this is hard-coded. The advantage of this is that Alice and Bob don't have to worry about agreeing upon all of the curve parameters before they can establish a shared key.

When each client instance is started, I generate the random private key for each client using Python's "secrets" module. The public key is then calculated using the point multiplication function mentioned previously. This public key is sent to the server along with the user's name (e.g. Alice) which they have entered via the command line. Any other user (e.g. Bob) can then get Alice's public key from the server and establish a secret shared key with her. In my system, Bob (or any other client) calculates their shared key with Alice by first asking the server for any available public keys. Bob then uses the point multiplication function to multiply Alice's public key by his private key. The shared keys that each client has established with their contacts are stored locally and used to encrypt future messages with the Advanced Encryption Standard (AES), a symmetric-key algorithm.

B.6 ECDSA

The *Elliptic Curve Digital Signature Algorithm* allows Alice to sign a message or file in such a way that Bob is able to then verify that Alice must have been the sender and that the message/file has not been altered. Alice does this by generating a signature using the hash of the message and her private key. She then sends this to Bob alongside her message. Bob then uses Alice's public key and the hash of the message to check that the signature must have been generated on the same message and by Alice. The two parts of this algorithm are formally defined below.

B.7 Signature Generation

For signing a message m sent by Alice, using Alice's private key d_A.

1. Calculate $e = HASH(m)$, where $HASH$ is a cryptographic hash function, such as SHA
2. Select a random integer k from $[1, n-1]$
3. Calculate $r = x1$, where $(x1, y1) = k \cdot G$ (mod n)
4. Calculate $s = k^{-1}(e + d_A \cdot r)$ (mod n)
5. The signature is the pair (r, s)

B.8 Signature Verification

For Bob to authenticate Alice's signature, Bob must have Alice's public key Q_A.

1. Calculate $e = HASH(m)$
2. Calculate $w = s^{-1}$ (mod n)
3. Calculate $u_1 = e \cdot w$ (mod n) and $u_2 = r \cdot w$ (mod n)
4. Calculate $(x_1, y_1) = u_1 \cdot G + u_2 \cdot Q_A$
5. The signature is valid if $x_1 = r$ (mod n), invalid otherwise

(Jurišic & Menezes 1997, Koblitz et al. 2000, Hankerson et al. 2003, Anoop 2007, Silverman 2009, Brown 2009).

In my implementation of this algorithm, Alice and Bob first agree upon all of the same initial curve parameters mentioned in Section B.4 by default since again they are all hard-coded in to every client. The advantage of this is that Alice and Bob don't have to worry about agreeing upon all of these parameters before they generate and verify signatures.

When Alice wants to send a signed message or a file to Bob, she enters the message or the name of the file via the command line, and generates a hash of the message/file using SHA-512 from Python's "hashlib" module. This hash is converted to an integer and truncated to be smaller than n, which means that this truncated hash e is less than n. A random number k is then generated using the same "secrets" module previously mentioned and the point G is multiplied by k using the point multiplication function, with the x-coordinate taken as r and the y-coordinate discarded. The integer r is multiplied by Alice's private key (also an integer) and added to the truncated hash of the message (another integer), finally this is divided by k to generate s. This division is done by multiplying by the inverse of k modulo n, calculated with the EEA function mentioned earlier. Alice sends both r and s to the server, along with her AES encrypted message.

On the receiving end of my implementation, after Bob receives all of this data from the server, he first decrypts the message itself using the shared key he has already established with Alice. He uses the plaintext message and the same hashing algorithm to generate a hash of the message, which he converts to an integer and truncates in the same way that Alice did, calling it e. This truncation ensures that the number e is smaller than n. The multiplicative inverse of s modulo n is then calculated using the EEA function and called w. Bob then does two integer multiplications, the first is the $e \cdot w \pmod n$, and the second is $r \cdot w \pmod n$, these new integers are called u_1 and u_2 respectively. Next, two lots of point multiplication are carried out and their results are summed by point addition. The first point multiplication is $u_1 \cdot G$ where G is the base point mentioned previously, and the second is $u_2 \cdot Q_A$ where the point Q_A is Alice's public key. After adding these two points together using the point addition function to create a new point, Bob takes just the x-coordinate of this new point and compares it to the r from Alice's signature. If these are equal $\pmod n$ then the signature is valid and was definitely generated by Alice using the same message which Bob received. So now Bob has the plaintext message and is assured that it is authentic.

C Tools used

The system was created entirely in Python. I chose this language because of both its ability to handle large integers with ease, and its large number of well-established modules for computing generic cryptographic functions. Since my system carries out operations involving huge integers, other less abstracted languages would have required much more involvement at the lower level which would only detract from my progress with the actual aims of the project.

The system can be started and interacted with via a command line interface. This was chosen since it allowed me to focus more on the protocols and back-end of the system rather than on designing a visually appealing graphical user interface (GUI). A GUI was not relevant to the mathematical-based aims of this project.

The seven Python modules used in my system are (in alphabetical order): "base64", "hashlib", "os", "PyCryptodome", "Pyro4", "secrets", and "uuid". Only two of these modules are not part of The Python Standard Library, and thus may require installation: "PyCryptodome" and "Pyro4". To implement the functions that I use from these modules was either not possible within the time constraints, or would have consumed time better spent on other aspects of the project more pertinent to my objectives. The server program only uses "Pyro4" and "uuid", whereas the client program uses every listed module apart from "uuid" (six total).

The "base64" module is used for decoding AES objects into bytes, "hashlib" for generating secure hashes, and "os" for file management. The "Cipher" package from "PyCryptodome" is used for encryption and decryption with AES, "Pyro4" is used for remote object invocation between the client and server, and the "secrets" module is for generating secure random numbers. In the server program, the "uuid" module is used for generating unique identifiers of every communication.

Initially I used the “random” module for random number generation, until I learned that this module is not cryptographically secure so I switched to using “secrets” instead. My own attempt at implementing file management for users to send and receive files using the system encountered problems when running on different operating systems. Therefore I decided to instead use the “os” module, which can handle miscellaneous operating system interfaces. This made the system far more portable since it was no longer operating system dependent.

I only used long standing and well used Python modules in order to avoid supply chain issues, specifically the kind recently highlighted by Ducklin (2021). Fake Python packages with misleading names can be uploaded to PyPi, where users in a hurry might download and install them by mistake, potentially infecting their project with malicious Python code. This is clearly an issue for any Python project which relies upon external modules, let alone a security-based one such as this.

D Implementation Issues

After I successfully implemented client-client communication with separate instances of the client program running on the same machine, I then attempted to adapt the system for use with the client instances on separate machines on the same network. However, I encountered issues with this due to the security protocols in place on networks. I decided against attempting to circumvent this potentially complex and time consuming issue, since network communication over the Internet is not strictly necessary for the purposes of this academic project, I am not aiming to replace Signal or WhatsApp.

E Verification and Validation

This subsection will run through a real interaction between Alice and Bob using the system, specifically the exchange shown in Figure 3. This will demonstrate the calculations performed by the system and the validity of these calculations. All large numbers here are shown in hexadecimal, for brevity. This interaction uses an elliptic curve known as Curve25519 (Bernstein 2006), it can be expressed by the equation:

$$y^2 = x^3 + 486662x^2 + x \pmod{2^{255} - 19}. \tag{4}$$

This curve is most often expressed in the Montgomery (1987) form as above ($By^2 = x^3 + Ax^2 + x$), but can be converted to the more traditional short Weierstrass form shown earlier ($y^2 = x^3 + ax + b$) through substitution. Curve25519 has the base point $g =$
$(0x9,\ 0x20ae19a1b8a086b4e01edd2c7748d14c923d4d7e6d7c61b229e9c5a27eced3d9)$,
which has prime order $n = 2^{252} +\ 0x14def9dea2f79cd65812631a5cf5d3ed$. The curve’s cofactor is $h = 8$. I use Curve25519 for this example since it is very commonly used in other similar implementations.

E.1 Key Generation

Alice’s randomly generated private key is
$0x69837a193b0f43bac8b32a30396a189c51706d49fcbb7c0e099b8f1b4bcb969$.
When she multiplies the base point g by this number, she obtains the point

$(0x67bf8995372ae1a9329f441d955193623aedf68a01ee5e2af7c33eabc27d9fd,$
$0xbc9a2fe41909056a46927d7c3af1ffed8e802854a32378df1845f819c016bdb)$
as her public key.

Generated in the same way, Bob's private key is
$0xf7d16d30314975879241ab2a17fdb1194fa2a7319b0b3331590c8482437f0b3$.
Therefore his public key is
$(0x2ffeaa851900d44817e3888854d958a91b9e1127ed0057a2e6796f3a1115cda,$
$0x3fd4639dbd63076c19aeb51816711d7de4331cb75ec3c89ea1c387297660809c)$.

E.2 Shared Key Establishment

Alice calculates the shared key by $h \cdot d \cdot Q$, where h is the cofactor, d is Alice's private key, and Q is Bob's public key.
Firstly, $d \cdot Q =$
$(0x7bb294204beba6ba902f5a21e14c4d5abc953d52b79be70377104f72de310a4a,$
$0x6d9d2ec13fc729a629259a1fcdf96d4a66e7e4e1365f54ed5fa56e235bc74b5)$
Then multiplying by $h = 8$ gives $h \cdot d \cdot Q =$
$(0x37ae8d3aa1495b39ead29b8abe24e7d8d7f2fb05186216a37c4b55708ead1fd,$
$0x29aecf95ba814e3e52866f5daa243d3366e8c8d0255117ee047822319b44222c)$
Therefore, Alice calculates the shared key as:

$$0x37ae8d3aa1495b39ead29b8abe24e7d8d7f2fb05186216a37c4b55708ead1fd. \tag{5}$$

Bob also calculates $h \cdot d \cdot Q$, except here d is his private key and Q is Alice's public key.
$d \cdot Q =$
$(0x7bb294204beba6ba902f5a21e14c4d5abc953d52b79be70377104f72de310a4a,$
$0x6d9d2ec13fc729a629259a1fcdf96d4a66e7e4e1365f54ed5fa56e235bc74b5)$
Then multiplying by $h = 8$ gives $h \cdot d \cdot Q =$
$(0x37ae8d3aa1495b39ead29b8abe24e7d8d7f2fb05186216a37c4b55708ead1fd,$
$0x29aecf95ba814e3e52866f5daa243d3366e8c8d0255117ee047822319b44222c)$
Bob's calculation of the shared key is:

$$0x37ae8d3aa1495b39ead29b8abe24e7d8d7f2fb05186216a37c4b55708ead1fd. \tag{6}$$

This demonstrates that the ECDH part of the system works, since Alice and Bob each calculate the same shared key (Key (5) = Key (6)). Critically, from a security perspective, only Alice and Bob can possibly know the value of this shared secret key because it was calculated using their respective private keys (which only they have access to). This makes it suitable for use as a symmetric AES key for end-to-end-encrypted communication between Alice and Bob.

E.3 Encryption

The system allows Alice to send a message or file to Bob which is encrypted using the shared key that they have just established (through ECDH). The following subsections (E.3, E.4, E.5, E.6) show an example message sent by Alice to Bob, as in Figure 3.

Alice wants to send the message "Hello, Bob!" to Bob. Their shared key is first converted from an integer to byte format, which is:

$b'\backslash x03z\backslash xe8\backslash xd3\backslash xaa\backslash x14\backslash x95\backslash xb3\backslash x9e\backslash xad)\backslash xb8\backslash xab$
$\backslash xe2N\}\backslash x8d\backslash x7f/\backslash xb0Q\backslash x86!j7\backslash xc4\backslash xb5W\backslash x08\backslash xea\backslash xd1\backslash xfd'$
(in big-endian byte order).

This is then hashed for security purposes, which generates the AES key:
$b'\backslash x00\backslash x0b\backslash x0f\backslash xe8\backslash xbfl\backslash xa9\backslash xbb0\backslash x00(*\backslash x92\backslash x01\backslash xba\backslash xba$
$\backslash xadU\backslash xc2K8\backslash x08\backslash x1a\backslash xdbL\backslash xb8U\backslash xcf\backslash x15\backslash x94\backslash xcf\backslash xca'$.

AES is used in EAX mode (encrypt-then-authenticate-then-translate) to generate the triple $(nonce, ciphertext, tag)$, where:
$nonce = b'\backslash xe0\backslash x80\backslash x00\backslash x8e\backslash xd63rPd\backslash xa3\backslash x161\backslash xe5l\backslash xf0\backslash xe4'$,
$ciphertext = b'U\backslash x15\backslash xe9\backslash x17\backslash xce4\backslash xa7\backslash xd9\backslash xbc : \backslash xc6'$,
$tag = b'[V\backslash xb2R\backslash x85k\backslash xb0UZRW\backslash xf7?\backslash xf8\backslash x14G'$.

Nonce is short for "number used once", it is a random number used in the encryption process and is required for decryption. This $(nonce, ciphertext, tag)$ triple is then sent to Bob, along with the signature generated in E.4.

E.4 Signature Generation

As mentioned, Alice can send an encrypted message to Bob, but she can also digitally sign it in such a way that Bob can verify the authenticity of the message. Here the signature is generated using the procedure outlined in Section B.7.

The message ("Hello, Bob!") is hashed and then truncated to give the integer $e =$
$0x144b549f0522c7f6c224fe404e8200cfbe352c16883f16777d01ba8c81bce487$.
Alice generates a random number k and calculates $r = kG =$

$$0xa96542405f3dc83b44e45beaa40d911efe8c5fee82a9f087cce28882b0fca82. \quad (7)$$

Next, she calculates $s = k^{-1}(e + d \cdot r) =$
$0x237543088442522d4dfb6acb7126982df6d73473d1d89fe0d26f24226d79e82$, where d is her private key.

The signature that Alice sends, alongside the encrypted message triple, is the pair (r, s). Therefore, in total, Alice sends the quintuple $(nonce, ciphertext, tag, r, s)$.

E.5 Decryption

Bob receives the quintuple $(nonce, ciphertext, tag, r, s)$. He uses the triple ($nonce$, $ciphertext$, tag), along with his shared key, to decrypt the message. Bob converts the shared key from an integer to byte format, which he hashes to generate the same AES key as Alice did in Section E.3. This AES key, along with the nonce, is used to decrypt the ciphertext. The plaintext is: "Hello, Bob!". The tag is also used to verify the authenticity of the message, although this is not strictly necessary since additional ECC authentication is done in Section E.6.

E.6 Signature Verification

After decrypting the message with AES, Bob uses the signature pair (r, s) to verify Alice's ECC signature using the procedure outlined in Section B.8. He first hashes the plaintext and truncates it to get the integer $e =$
$0x144b549f0522c7f6c224fe404e8200cfbe352c16883f16777d01ba8c81bce487$.

Note that this is the same hash generated by Alice, but Bob doesn't yet know this since the hash itself is not sent. Bob calculates the x-coordinate of the point $v = (e \cdot s^{-1})G + (r \cdot s^{-1})Q =$

$$0xa96542405f3dc83b44e45beaa40d911efe8c5fee82a9f087cce28882b0fca82, \tag{8}$$

where Q is Alice's public key.

Since this v matches the r in Alice's signature ($v = r$), Bob is assured that Alice sent the message, and that the content of the message is unchanged, since both Alice's private key, and the same hash must have been used to generate the signature.

This demonstration both verifies and validates my system. The fact that both Alice and Bob calculate the same shared key, (5) and (6), verifies and validates that the ECDH algorithm for key establishment has been implemented correctly in my system. While the fact that Alice's r generated in (7) matches Bob's v generated in (8) verifies and validates that the ECDSA algorithm for both signature generation and verification has also been implemented correctly.

IV RESULTS

This section presents the results of various tests carried out on the system. These tests analyse both the efficiency (Section A) and the security (Section B) of my system. In terms of efficiency, this includes a comparison between my ECC system and a standard RSA implementation, as well as a comparison between various ECC curves which my system can use. When analysing the security of my system, I look at whether any information about the private key is leaked through side-channel attacks, specifically timing attacks. All tests were carried out using a machine with an AMD Ryzen 5 3600 processor (3.6 GHz base, 4.2 GHz boost).

A Efficiency

Perhaps the most significant claimed benefit of using ECC is its superior efficiency over RSA. Therefore I start here with a timing comparison between my ECC system and an RSA implementation from the "PublicKey" package of the "PyCryptodome" library. Specifically, the operation being timed here is the generation of a key pair, meaning both private and public key generation, of varying security strength. It is important to note that security strength is not the same as the key size, this is due to attacks on both RSA and ECC that provide computational advantages. As an example, a security strength of 112 bits requires an RSA key size of 2048 bits and an ECC key size of 224 bits (Barker et al. 2020, p54-55).

For my ECC system I am measuring the time taken to both choose a random private key of the given security strength, and to then multiply the base point of the curve (M-511) by this number to create the corresponding public key. I am using the curve M-511 because it is one of the largest elliptic curves, meaning it can use the largest range of private key values, generating the largest range of results here. For RSA I am measuring the time taken to return a key pair of a specified security strength by the library function "RSA.generate()". Figure 4 shows the results of this comparison for key pairs of 5 different security strengths (excluding 0). The top graph in Figure 4 shows the overall comparison for the entire range of both RSA and ECC (0-256 bits), while the bottom graph shows the same comparison except with a smaller range for RSA (0-112 bits).

Figure 4 illustrates that the time taken by my ECC system scales linearly as security strength increases, whereas the time taken by RSA scales exponentially. The second point to make about these results is that the bottom graph in Figure 4 shows RSA up to a security strength of 112 bits, which is today's minimum recommended level of security according to NIST (Barker et al. 2020, p54-55). Even at today's minimum security level RSA takes nearly an order of magnitude longer than ECC to generate a key pair (0.82s v 0.096s here). It is also important to consider that, due to the nature of computer advancement, this minimum security level will only increase over time. This means that the benefits of using ECC over RSA will also only increase over time, for example when the recommended security level moves to 128 bits, the time difference will widen further. In these results that difference is 4.9 seconds for RSA against 0.11 seconds for ECC. In general this is the nature of comparing a $O(n^2)$ system with a $O(n)$ system. ECC is faster than RSA today, and will get even faster (relatively) in the future.

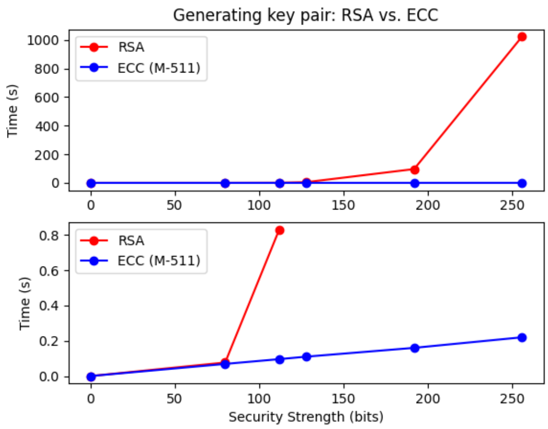

Figure 4: ECC more efficient than RSA

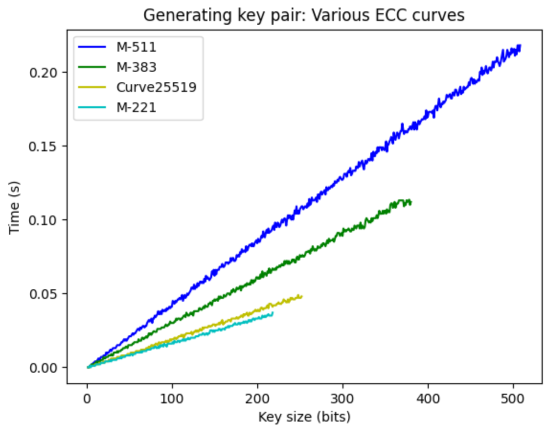

Figure 5: Some curves faster than others

After demonstrating the benefits of ECC over RSA, the focus then shifts to the benefits of some ECC curves over others. Figure 5 shows how the time taken to generate a key pair varies as the size of the private key increases, for 4 different curves. This, as before, means timing the operations of selecting a private key k of a given size and then generating the corresponding public key kG by multiplying the curve's base point G by this number.

Figure 5 illustrates that, while the time taken by every curve increases linearly, the speed of this increase (the gradient of the line) is greater for some curves than others. These differences can be explained by the properties of each curve. Recall from Section III.B.1 that an elliptic curve is defined over a finite field modulo p. This means that if the calculation of a coordinate produces a result greater than p, p is subtracted from this result to produce a much smaller number. Therefore no point can have an x or y-coordinate greater than p. If a curve is defined to have a larger p, the points on the curve can have larger coordinates, so calculations involving these points will take longer. This explains the differences in gradients of the 4 lines in Figure 5, since curve M-221 has $p \approx 2^{221}$ while curve M-511 has $p \approx 2^{511}$ (hence the names).

Another point to note about Figure 5 is that some lines on the graph are shorter than others. This is due to another property of each curve. Recall from Section III.B.5 that the base point G of every curve has an order n, where $nG = 0$. What this means in practice, as mentioned in

Section III.B.5, is that the private keys used with a given curve must be less than n. When it came to producing the results in Figure 5, the key sizes for each curve could only range from 1 to n, and n is different for every curve. For example, the order of curve M-221's base point is $n \approx 2^{218}$, while curve M-511's is $n \approx 2^{508}$. This explains the length of each line in Figure 5, the line for M-221 ends at 218 bits and the line for M-511 ends at 508 bits.

B Security

Here I analyse my system for security vulnerabilities, specifically whether my implementation leaks any secret data through side-channel attacks such as timing attacks. If there is a correlation between the properties of a secret value and the time taken by an operation involving this value, an attacker could use a timing attack to gain some information about the supposedly secret value. In terms of my system, the secret value is the private key k and the operation is the calculation of the public key kG, in which the base point G is multiplied by the private key k. This operation is carried out by repeated doubling and adding of the base point G, until reaching kG. If an attacker is able to isolate and time this operation, knowing that there was a correlation, they could infer information about the private key from measuring how long the operation took.

To check if my system leaked side-channel information in this way, I used the system to generate 300 random key pairs, timing the time taken by the process each time. I plotted these times against two different properties of the private key: the number of 1 bits in the binary expansion of the key (Figure 6), and the size of the key, measured by taking the log of the key (Figure 7).

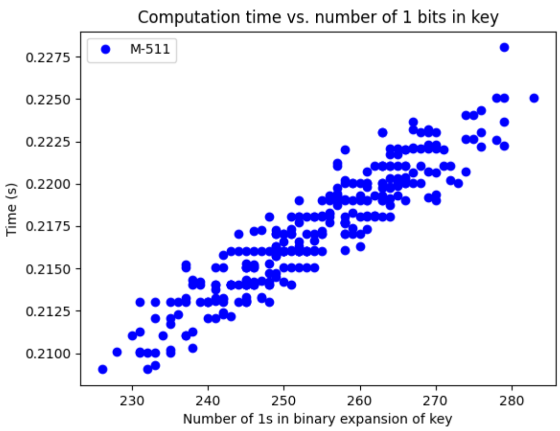

Figure 6: Correlation with number of 1s in key

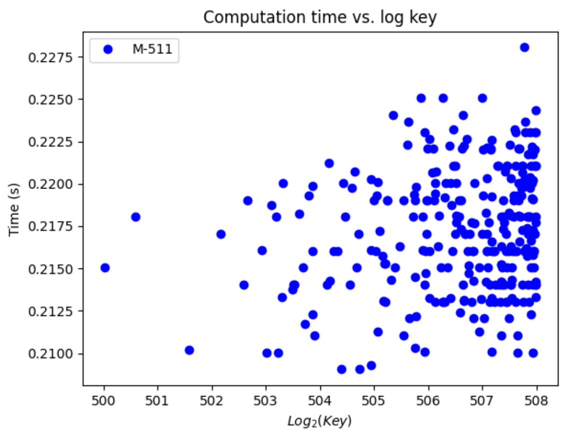

Figure 7: No strong correlation with key size

Figure 6 appears to show a positive correlation between the number of 1s in the binary expansion of the private key and the time taken to generate the key pair. However, Figure 7 appears to show no strong correlation between the value of the private key and the time taken to generate the key pair.

The correlation shown in Figure 6 can be explained by examining the doubling and adding procedure. To simplify, the system doubles the point G repeatedly to produce $2G$, $4G$, ..., $2^{log_2(k)} \cdot G$. Then, the relevant doublings are each added to produce the result, so if the i^{th} bit in the binary expansion of k is a 1, $2^i \cdot G$ is added. Therefore, the more 1 bits there are in k, the more point additions will have to be done. Since every operation of point addition takes some amount of time, the total time taken will increase as the number of point additions increases. This is why Figure 6 shows a positive correlation.

The lack of strong correlation shown in Figure 7 can be explained by considering the scale of the axis in comparison to that of Figure 5. Figure 5 shows that there is a strong positive correlation between private key size and time taken to generate a key pair, but that is over the entire range of key sizes 0 to 508. Since here in Figure 7 the private key is randomly chosen from the range $[1 - n]$ where $n \approx 2^{508}$, the most common key size is 508 bits ($2^{507} - 2^{508}$), with half as many keys for every bit decrease. This explains both why the spread of points in Figure 7 is so concentrated on the right side, and why the number of points in each interval approximately halves moving from left to right. This is representative of how the system performs in practice. The relatively small variation in both axes of Figure 7 means that random error (noise or natural variation) plays a much more significant role in the timing results and obscures the underlying positive correlation between key size and time taken. Therefore, my system does not show a strong correlation between the key size and the time taken in practice.

V EVALUATION

Regarding Figure 6, although this shows correlation between the number of 1 bits in the binary expansion of the key and the time taken, this is not a terminal problem for the system. Even if an attacker could infer that the number of 1 bits in the key was exactly x, this still leaves the number of possible keys as $\binom{508}{x}$. The attacker's best case scenario here is for x to be at either end of the x-axis in Figure 6. Suppose that in a rare case x is known to be as low as 220, this still gives $\binom{508}{220} \approx 2^{497}$. Therefore, the attacker has reduced the number of possible keys from $\sim 2^{508}$ to $\sim 2^{497}$ in a best case scenario for them. This is certainly not a crippling issue for the system.

The minimum recommended security requirements for ECC key size today is 224 bits (Barker et al. 2020, p54-55). This means that curve M-221 is not considered secure anymore, but the other three curves are secure since their $n < 2^{224}$. This minimum requirement for key size will increase in the future as computers get faster. Therefore, while Curve25519 is today's fastest secure curve, curves M-383 and M-511 will still be considered secure further into the future. Specifically, for 2031 and beyond the minimum security strength for encryption algorithms will move to 128 bits (Barker et al. 2020, p59). For ECC this means 256 bit keys, and for RSA this would be 3072 bit keys. So 2031 is when Curve25519 will no longer be used, while curves M-383 and M-511 will used for some time after this. This timeline does not take into account if/when large-scale quantum computers will become available, since this would/will require the usage of quantum-resistant algorithms. Neither ECC nor RSA are quantum-resistant since they rely on the difficulty of the discrete logarithm problem, which can be easily solved using Shor's algorithm for polynomial-time integer factorisation on a quantum computer.

VI CONCLUSIONS

ECC is more efficient than RSA, but not all ECC curves are equally strong or fast. The security of ECC is based on the hardness of the ECDLP, and currently the best algorithms known to solve the ECDLP have fully exponential running time. This is in contrast to the subexponential-time algorithms known for the integer factorisation problem, on which the security of RSA is based.

This means that, for example, a 224-bit ECC key provides the same level of security as a 2048-bit RSA key. ECC is a very strong encryption method, but only when implemented properly, using a secure curve and a good random number generator.

A Further Work

A possible solution to the issue shown in Figure 6 is to add a timing delay to the system such that the key pair generation always takes the same amount of time. The system would generate the key pair and then wait until a given time had elapsed before returned the result, meaning that an attacker could not infer any information about the private key.

References

Adleman, L. (1979), A subexponential algorithm for the discrete logarithm problem with applications to cryptography, *in* ‘20th Annual Symposium on Foundations of Computer Science (sfcs 1979)’, pp. 55–60.

Anoop, M. (2007), ‘Elliptic curve cryptography, an implementation guide’, *online Implementation Tutorial, Tata Elxsi, India* **5**.

Barker, E., Barker, W., Burr, W., Polk, W., Smid, M. et al. (2020), *Recommendation for key management: Part 1: General*, National Institute of Standards and Technology, Technology Administration.

Bernstein, D. J. (2006), Curve25519: New diffie-hellman speed records, *in* M. Yung, Y. Dodis, A. Kiayias & T. Malkin, eds, ‘Public Key Cryptography - PKC 2006’, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 207–228.

Brown, D. (2009), ‘Standards for efficient cryptography 1 (sec-1)’, *Standards for Efficient Cryptography, 2009* .

Diffie, W. & Hellman, M. (1976), ‘New directions in cryptography’, *IEEE Transactions on Information Theory* **22**(6), 644–654.

Ducklin, P. (2021), ‘Poison packages - ”supply chain risks” user hits python community with 4000 fake modules’.
URL: *https://nakedsecurity.sophos.com/2021/03/07/poison-packages-supply-chain-risks-user-hits-python-community-with-4000-fake-modules*

Hankerson, D., Menezes, A. J. & Vanstone, S. (2003), *Guide to elliptic curve cryptography*, Springer Science & Business Media.

Hotz, G. (2010), Console hacking 2010-ps3 epic fail, *in* ‘27th Chaos Communications Congress’.
URL: *https://fahrplan.events.ccc.de/congress/2010/Fahrplan/events/4087.en.html*

Jurišic, A. & Menezes, A. (1997), ‘Elliptic curves and cryptography’, *Dr. Dobb’s Journal* pp. 26–36.

Koblitz, N. (1987), ‘Elliptic curve cryptosystems’, *Mathematics of computation* **48**(177), 203–209.

Koblitz, N., Menezes, A. & Vanstone, S. (2000), 'The state of elliptic curve cryptography', *Designs, codes and cryptography* **19**(2-3), 173–193.

López, J. & Dahab, R. (2000), 'An overview of elliptic curve cryptography'.

Miller, V. S. (1986), Use of elliptic curves in cryptography, *in* H. C. Williams, ed., 'Advances in Cryptology — CRYPTO '85 Proceedings', Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 417–426.

Montgomery, P. L. (1987), 'Speeding the pollard and elliptic curve methods of factorization', *Mathematics of computation* **48**(177), 243–264.

Rivest, R. L., Shamir, A. & Adleman, L. (1978), 'A method for obtaining digital signatures and public-key cryptosystems', *Commun. ACM* **21**(2), 120–126.
URL: *https://doi.org/10.1145/359340.359342*

Silverman, J. H. (2009), *The arithmetic of elliptic curves*, Vol. 106, Springer Science & Business Media.