Team Reflective Statement

Yiru Pan (Vivian)'s reflective statement:

In this research project, I am mainly in charge of Lamport digital signature scheme. The reason why I chose this scheme is this scheme can resist quantum computing, which sounds very intriguing. At the beginning, I need to search related academic papers, and uploaded them in google drive for other teammates to read and learn. Then I attended every group meeting, and shared my ideas with others proactively. In the experiment stage, I implemented code of Lamport scheme, and uploaded it to Spartan to test, finished all designed scenarios. During the implementation, I knew the reason why Lamport can resist quantum computing, while other schemes cannot. Lamport uses large number of hash function calculations instead of relying on algorithm complexity. Its security depends on the performance of hash function. The state of art implementation of Lamport is to use SHA256, which has been proved safe in most scenarios. My code can accomplish three steps, including key generation, signing message, signature verification. During the test, with the input scale increasing, the measurement data can demonstrate that the Lamport scheme is dominant by key generation, because it contains a large number of hash calculations in key generation stage. What is more, I finished design of presentation slides, and generated all data form in a uniform format. During the video making stage, I helped other teammates to truncate speech sheet and control time limit, I cooperated well with other teammates, and learned very useful skills from others, such as using Latex to edit report and using python to generate plot. These skills are very helpful for my future academic life.

From this research project, I also learned other digital signature schemes, which broaden my knowledge scope significantly. I learned how to write professional academic paper in Latex, and the professional evaluation metric for different schemes. This project is a good opportunity for me to sharpen my research skills as well as teamwork skills. We did a good job in the presentation, and improved in the final report per the comments from teacher and tutor. I indeed learned a lot from this subject, and this subject is one of my favorite subjects during my master life. Thanks for lecturer, tutors and my teammates, it has been a pleasure to learn this subject.

Tenglun Tan (Tan Tan)'s reflective statement:

Initially I was very excited to do this research project, since last semester I have been working with a professor in the cryptography area, meeting with several "Blockchain Maniacs" and doing literature reviews on some recent research papers every week. In this project our team choose the Experiment track, which gives me a chance to apply the theory in practice. Our team works in a very cooperative and efficient way. As an active contributor to our team, I was in charge of the ECDSA part, learning ECDSA algorithms, running experiments, and analysing results that can be used to draw the conclusion in the overall comparison part. I wrote ECDSA part in the report and present it in our presentation. In the end I was surprised to find out that concepts such as point multiplication on elliptic curve that was very confusing to me several months ago become clearer now. When reflecting back, I found out it was the research papers I have read in this area and the experiments I have done that helps me understand the mechanism of ECDSA better. I analyzed twelve papers talking about ECDSA with high citations. Several of them illustrates the key generation, message signing and signature verification thoroughly so that I will not miss any point of the algorithms. I also learned how previous researchers did their performance and security experiments, so I can stand on the shoulders of giant to research further. My teammates gave a lot of help as well. We held several group meetings to share ideas and discuss the progress we have made for the past period. From them, I was exposed to the new algorithms such as EdDSA and Lamport, whose performance results are very exciting. At the beginning I used pure python package to implement ECDSA and found out it was pretty slow to generate key pairs, even slower than RSA. My teammate Victor helped me find out another implementation of ECDSA using C which is nearly 10 times faster than the pure python package. For the report and presentation, I used python matplotlib to generate appropriate figures for all the plots and Vivian used Mac Numbers to generate appropriate tables. I also learned the truth that implementation method does matter for the same algorithms from my team working experience. In the overall comparison part, I contributed my insights and got a big picture of the object of this project. I really enjoyed this team learning and working experience.

Zhuohan Xie (Theo)'s reflective statement:

In this project, we distributed our experiments into four digital algorithms, RSA, ECDSA, EdDSA and lamport and assigned each team member to explore one of these algorithms and conducted the cross-comparison after that. My part is EdDSA, which is a very new algorithm that is believed to achieve higher security with higher speed. Since it is not covered in the lecture materials, I have to read two papers which I cited in the EdDSA part to get an idea about what EdDSA is. After that, I tried to implement this algorithm on OpenSSL, but it seems that it is not well supported by OpenSSL, I could generate key pairs

by OpenSSL but could not sign the messages or verify the signatures with it. Therefore, I gave up OpenSSL and tried to find python package called ed25519. After understanding the principles of this package, I imported this package in my python code to generate key pairs, sign the messages and verify the signatures and measure the time of each step. It works well on my local laptop and the time each step takes is small. However, An interesting thing is that I found this package did not work well when I uploaded it to the cloud server which is provided by my teammate. It took much longer time on the cloud server than on my local laptop. After that, I experimented two other packages, PyNacl and libNacl with assistance of my teammate. These two algorithms work well on the cloud server, and I compared the performance of these three and analysed the experiment results in the presentation slides and report. In the presentation, I made EdDSA part of slides and contributed my insights in the cross comparison and helped to build comparison metric in the last slide and explained EdDSA part in the recording video. My slide is mainly focused on the experiment I did in EdDSA since it is the main focus of our research topic.

As for report, I wrote the EdDSA section, it is not just about the experiment part like I mentioned above in the slide, it is the whole thing about EdDSA, a general overview of this algorithm, parameters of this algorithm, mathematical equations behind message signing and signature verification processes. I read several papers so as to get the gist and write this part. Also, I illustrated my experiment results in detail in performance part, which is also covered in the slide, but with more analysis. Finally, the other main part of EdDSA section of the report is the security part, in this part, I not only analysed what security level EdDSA could provide, but also found a valid attack against it and wrote the detail in the report to illustrate that even though EdDSA might defend side channel attacks, but it could still be valuable to fault attacks and came to a conclusion that protections or countermeasures against fault attacks should be considered if the algorithm is applied in areas where they are possible. Overall, I enjoyed our team work and learned significant knowledge from this project.

San Kho Lin (Victor)'s reflective statement:

For the algorithm, I choose RSA since I wanted to study deeper and go-beyond the topic taught in the class. The motivation is related to my real life working experience where I need to develop and deploy distributed systems and oversee many Cloud Computing Clusters and instances. On daily basis, tools such as SSH, OpenSSL, PyCA and BouncyCastle are my tool-of-trade to develop and deploy software in client-server environment. Generating SSL certificates, revoking and maintaining certificates, signing software releases, as such daily routines involve dealing with Public Key Cryptography and, encounter with RSA is inevitable. I am glad I have enrolled this subject and undertake rigorous assessments to strengthen my knowledge further on cryptography and underlying mathematics.

In term of contribution to team work, my teammates are kind enough to nominate me as a team leader. As a team leader, I initiated to carry out research methods, gather related quality literature, brainstorm benchmarking methodology, formulate evaluation strategy, propose tools and quality implementations, write Python code, setup Cloud VM, perform cross comparison runtime and analysis, draw conclusion, take care of report format and overall editorial on Latex. Furthermore, I also try my best to understand and extend my contribution on the rest of the digital signature algorithms presenting in this research paper. Closely collaboration with my teammate make me reach to the deeper level of understanding on the presenting topics; of which doing it all by myself alone might not be possible within short amount of time. As a team, it is our mutual goal to get highest mark on the research work we present. Therefore, all team members work hard to present the quality research and excellent experiment at our best effort. All in all, I believe, we are the high performance team!

For Latex authoring, we use Overleaf (https://www.overleaf.com). For report format, we use ACM conference paper and citation format in 10pt, two-columns template.

- https://www.acm.org/publications/proceedings-template
- https://github.com/borisveytsman/acmart/

Team 73
yirup, tenglunt, zhuohanx, sanl1
Research Project: Experiment Track
Digital Signature Schemes Evaluation
COMP90043 Cryptography and Security, 2018 SM2

Digital Signature Schemes Evaluation

San Kho Lin
829463
University of Melbourne
sanl1@student.unimelb.edu.au

Yiru Pan
889832
University of Melbourne
yirup@student.unimelb.edu.au

Tenglun Tan
876792
University of Melbourne
tenglunt@student.unimelb.edu.au

Zhuohan Xie
871089
University of Melbourne
zhuohanx@student.unimelb.edu.au

1 INTRODUCTION

Digital signature is a mathematical scheme that used to validate the authenticity of a message. By signing a digital document, one can ensure that the signer is the original author of the digital document and, other parties cannot modify it. It is widely used technique to add authentication, integrity and non-repudiation properties to a digital document. The idea of digital signature is one of the applied public key cryptography originally discussed in Diffie-Hellman paper [1]. Particularly, we are interested in 3 digital signature algorithms – RSA, ECC and Lamport signature – to evaluate and compare them.

1.1 Background

RSA [2] is the well-known implementation of public key cryptography and digital signature scheme. It is based on the hard factorization problem of two large prime numbers p and q such that it is not computationally feasible for anyone to factor n i.e. $n = p \cdot q$. We use RSA as our baseline for formulating evaluation metrics.

Elliptic Curve Cryptography (ECC) is another efficient public key cryptography technique. Particularly, we focus on ECDSA [3] which is based on the difficulty of solving discrete logarithm problem in Elliptic Curve group. We also investigate EdDSA [4] that has used twisted Edwards curves modifications and parameter tuning for top performance. ECC is well-known for providing greater security with smaller key sizes. However, ECC discrete logarithms computation is known to be broken by Shor's algorithm [5] on a hypothetical quantum computer.

Lamport one-time digital signature scheme [6] is an efficient method for constructing a digital signature. It has been increased interest due to characteristics such as fast verification, one-way property and quantum-proof. RSA, ECC rely on the computational hard problems which are prone to be broken when a quantum computer is practically implemented.

1.2 Motivation

In this project, we explore 3 digital signature schemes: RSA, ECC, and Lamport signature scheme. The main idea of choosing these 3 schemes is their uniqueness in technique and mathematical approaches. We study their mathematical foundations and associated process for key generation, signing and verification of digital signatures. We perform experiments to measure their performance such as timing private/public keys generation, signing and verifying the data messages. We also discuss their performance-security tradeoffs, complexity and implementations. Our objective is to observe characteristic and performance of each scheme and, produce qualitative evaluation metrics and discuss suitable application domain for each of them.

1.3 Experiment Setup and Tools

We begin our research on publicly accessible quality implementation of these algorithms. After experimenting many initial trials, we have empirically finalised to choose the following cryptographic tools and libraries for experiment.

- OpenSSL (https://www.openssl.org)
- NaCl (https://nacl.cr.yp.to)
- LibSodium (https://libsodium.org)

Our choice of programming language is Python and, therefore we have investigated tools and wrapper around these chosen libraries. We have also written our best effort implementation, sourced and utilised third party implementation for each digital signature scheme when we deem appropriate to strengthen our discussion and experiments. Such activity, sources and details are further cited and discussed in respective section.

Additionally, in order to get the uniform experiment result, we use NeCTAR (https://nectar.org.au) Research Cloud virtual machine (VM) instance for all our runtime benchmarking. This NeCTAR Cloud VM is the *m2.medium* instance flavour which comprise of **2 vCPU** and **6GB RAM** memory. The choice of base operating system is *Ubuntu 18.04.1 LTS*. We name this Cloud VM to **Krypton** for easy reference.

For the test datasets, we use the Comma-separated Values (CSV) data that is publicly available at:

- http://eforexcel.com/wp/category/downloads/

The average message size is 160 bytes per-line in plain text format. Initially, we also aim to experiment on different data formats (jpg, doc). Since our focus is observing algorithm characteristic, a small plain text message is sufficient enough for our experiment. Besides, we justify that the message will get hash function applied before signing process.

The remainder of the paper is organised as follows. Sections 2, 3, 4 and 5 covers RSA, ECDSA, EdDSA and Lamport signature scheme investigation details. Section 6 provides cross comparison performance analysis and quantitative research on generating runtime measurement. Finally, section 7 gives qualitative research outcome, concluding remarks and identify future works.

2 RSA

RSA [2] is well-studied and popular Public Key Cryptography System (PKCS) appear in many standards such as NIST [3], RFC8017 [7]. It is based on two computationally hard problems known as Integer Factorisation problem (as one-way function) and RSA problem (as trap-door function). The factoring large integers is well-known "hard" problem in computational mathematics and Number theory. Especially in RSA, we consider two large prime numbers such that $N = p \cdot q$; by only knowing N, it is infeasible to find prime factors p and q of N. Thus, integer factoring problem serves as one-way function in RSA.

RSA propose working on Integer Modulo multiplicative Group $\mathbb{Z}_N^*$ such that let e, d be two integers satisfying $e \cdot d = 1 \mod \varphi(N)$ where $\varphi(N) = (p-1) \cdot (q-1)$. Then, RSA defines e be public exponent, d be private exponent and N be public modulus. Therefore, $\langle N, e \rangle$ pair forms public key; similarly $\langle N, d \rangle$ pair forms private key. For the integer representation of message $M \in \mathbb{Z}_N^*$, encryption defines as $C = M^e \mod N$ and, decryption defines as $M = C^d \mod N$ which indeed satisfy $M = (M^e)^d = C^d \pmod N$. Note that for given publicly known N and e for $C = M^e \mod N$, RSA make strong assumption that it is infeasible to determine M. It is the second computational "hard" problem that manifest in RSA. RSA states that finding eth root of modulo integer N of unknown factorisation is hard and, conclude as RSA problem. Only if we know the private component d then we can decrypt the message and, therefore, RSA problem serves as trap-door function.

For computing RSA digital signature, it is the inverted operation of encryption and decryption process. That is, signing operation is $S = M^d \mod N$ and, signature verification on M is $S^e = M \mod N$.

2.1 Signature Security

We observe that RSA problem is stronger assumption than Integer Factoring problem. It means that when it might be possible to find an efficient algorithm for the RSA problem but finding efficient algorithm for the factoring problem remains infeasible. According to [8], we learn that the General Number Field Sieve (GNFS) algorithm is the best known efficient algorithm for integer factoring which has run time complexity of $exp\big((c + o(1))n^{1/3}log^{2/3}n\big)$ on n-bit integers for some $c < 2$. Generally speaking, it is sub-exponential time complexity.

Since introduction of RSA and more than 30-years, RSA "hard" problem assumptions withstand mathematical attacks, when RSA is implemented and used properly. However, studies [9] [10] [11] show that many attacks are still possible for improper use of RSA algorithm. In the context of research presenting in this paper, we are interested in attacks related to RSA digital signature scheme and, its countermeasures. In previous section, we explain mathematical foundation on how basis RSA work. This simple version of RSA is also referred to as **Textbook RSA** in cryptography literature.

The textbook RSA version has some implication on multiplicative property of RSA signature which lead to **Existential Forgery attack** such that for some messages $M1$ and $M2$, it is $(M1 * M2)^d = M1^d * M2^d$; if $S1$ is signature of $M1$ and $S2$ is signature of $M2$; then, $(S1 * S2)$ is the signature of $(M1 * M2)$, therefore signature forgery is possible. Also, since signature is the inverse of encryption operation, what if the same key pair is used for signing and encryption. Furthermore, what if we try to sign very long message i.e. $M >$ modulo N.

Another attack to consider for RSA signature is **Blinding attack**. Suppose Bob wishes Alice to sign a message M but Alice normally refuse to do. Alice public key is $\langle N, e \rangle$ then, Bob choose random $X \in \mathbb{Z}_N^*$ and, create a blinded message $M_b = X^e \cdot M \mod N$ and ask Alice to sign M_b. Since M is hidden due to multiplicative process, Alice may sign $S_b = (M_b)^d \mod N$. Now Bob can compute signature for M as $S = S_b \cdot X^{-1} \mod N$. We can now show that:

$$S^e = (S_b)^e / X^e = (M_b)^{ed} / X^e \equiv M_b / X^e = M \pmod N$$

Hence, S is the signature for M.

Another important aspect of RSA signature security is the Low Public Exponent e. According to RSA standard RFC8017 [7], it defines e can be any random integers between $3 \leq e \leq (N-1)$, as long as e is relatively prime to public modulus N i.e. $\gcd(e, N) = 1$. It may be desirable to choose lower e value for performance purpose during exponentiation by repeated squaring and multiplying algorithm computation step of RSA process. However, [9] survey many attacks related to implication on choosing low public exponent value

(e.g. $e = 3$) such as Broadcast attack using Chinese Remainder theorem and, efficient Coppersmith's [10] theorem on finding small roots of polynomials modulo a composite N. By observation, we find that Fermat primes[1] ($F_n = 2^{2^n} + 1$), especially $F4 = 2^{16} + 1 = 65537$ is recommended to defeat certain attacks. [9] explains that when 65537 is used, signature verification just need 17 multiplications compare to approximately 1000 multiplications when a random $e \leq \varphi(N)$. This is due the fact that the binary representation of 65537 number[2] contains most 0s binary state, which effectively fast in low-level machine code computation.

Furthermore, we also investigate the public exponent e value usage of well-known implementations. In OpenSSL, it generates e with $F4$ option[3] by default and, also offers $e = 3$. In OpenSSH (v5.3)[4], it uses $e = 35$ and, PuTTY[5] uses $e = 37$.

Therefore, we can observe that RSA public exponent e is empirical choice over some interesting *Prime Number* that depends on standard practice and, a good compromise between performance and security trade-off.

2.2 Padding and Hashing

The real world RSA implementation has to consider technique to countermeasure the issue arise from the basic textbook RSA. [12] further explains that without proper prepossessing scheme, the textbook RSA signing and encryption is fundamentally insecure. RSA standard PKCS#1 introduces hash-then-sign padding scheme such that it calculates the cryptographic hash of the input message, together with some fixed padding, and this is then used as an input for the RSA function. However, such padding scheme is simple and strictly deterministic i.e. for the same input and key, the output will always be the same. In [13], it explains how Chosen Ciphertext Attacks can be mounted against observing the returns query of decryption-box on only one bit telling whether the ciphertext correlates to some unknown block (i.e. padding block) of data encrypted using PKCS#1, and the attacker could eventually gain information on C^d.

Therefore, later RSA standard improvement includes a more complex format schemes such as Optimal Asymmetric Encryption Padding (OAEP) [14] for encryption process and, Probabilistic Signature Scheme (PSS) [15] for signature process. [16] survey and review the PSS scheme and how it works. It also explains the key concept such as Hash-then-Sign, Mask Generating Function (MGF) and Full Domain Hashing (FDH). In a nutshell, PSS adds randomness to the signature process, and a technique for reverting the randomness during verification. Randomisation is crucial to prevent dictionary attacks. This is important as attacks may be possible if the source of the random numbers is weak.

[1] https://en.wikipedia.org/wiki/Fermat_number
[2] https://en.wikipedia.org/wiki/65,537
[3] https://www.openssl.org/docs/manmaster/man1/genrsa.html
[4] https://github.com/openssh/openssh-portable/blob/V_5_3/key.c#L688
[5] https://git.tartarus.org/?p=simon/putty.git;a=blob;f=sshrsag.c;hb=HEAD#l9

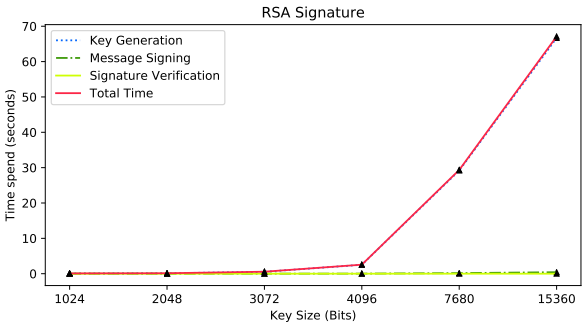

Figure 1: RSA Varying Keysize Runtime

2.3 Performance

We use PyCA [17] with OpenSSL backend for RSA experiment. To get RSA performance in reasonable secured setting, we setup parameters as follows:

```
public_exponent = 65537
param_hash = hashes.SHA256()
openssl_backend = default_backend()
rsa_padding = padding.PSS( \
    mgf=padding.MGF1(param_hash), \
    salt_length = padding.PSS.MAX_LENGTH)
```

We, then vary the different key sizes ranging from bits value of 1024, 2048, 3072, 4096, 7680 and 15360 with a fixed load of signing only **one** message. The choice of key sizes are from NIST's Recommendation for Key Management [18] standard; except 4096-bit is chosen empirically due to popular usage over Internet for digital certificate purpose. We fix the load to one message because we want to observe the highest 15360-bit runtime performance.

As depict in Figure 1, RSA key generation is predominantly influence the overall runtime. Message signing and signature verification runtime become insignificant and taken less than a second in all key sizes. Furthermore, as we can see in the graph, RSA key generation runtime is exponential growth rate as we increase key size from 4096-bit to 15360-bit. According to NIST [18], RSA key length 1024-bit and less should not use anymore. 2048-bit key size is acceptable usage until 2030. Key length of more than 3072-bit should use after 2030.

Even though it will not be linear, by hypothesis, we can approximate N number of messages runtime for key size 15360-bit by just multiply 1 message runtime reported in graph. For example, 70s x 1000 messages = 70,000s time required for key size 15360-bit. We emphasize the notion of signing 1 message with highest key size 15360-bit as the baseline evaluation metric for comparing in later sections.

3 ECDSA

The Elliptic Curve Digital Signature Algorithm (ECDSA) realizes Digital Signature Algorithm (DSA) using the Elliptic Curve Cryptography (ECC). In 1999, ECDSA was accepted as an ANSI standard. In 2000, it was accepted as NIST and IEEE standards [19]. Breaking traditional digital signature scheme such as RSA and DSA needs to solve integer factorization problem or discrete logarithm problem, where both of them only take sub-exponential time. Comparing to RSA and DSA, the running time to break ECDSA is fully exponential. Therefore, ECDSA can achieve same security with smaller key sizes and faster running time.

3.1 ECDSA Algorithm

Same as all the other digital signature schemes, ECDSA contains three algorithms representing three stages: Key Generation, Message Signing and Signature Verification. Computing Elliptic Curve Discrete Logarithm Problem (ECDLP) in the group of points on an elliptic curve defined over a finite field is the main difficulty that ensures the security of a public key system [20]. Typically, we use SHA-256 as the hash function in Message Signing and Signature Verification stages.

Domain Parameters: There are five domain parameters in ECDSA algorithms that are common to all entities participating in the network: (q, E, G, n, h). These include a chosen elliptic curve E defined over a finite field F_q and a base point $GE \in E(F_q)$. The Parameters table below further illustrates the meaning of each parameter.

Parameters of ECDSA:

- q: The prime number that define the finite zone, which is also referred as order.
- E: An elliptic curve $y^3 = x^3 + ax + b$ which is defined over the prive field F_q.
- G: An elliptic curve base point in $E(F_q)$.
- n: The integer order of G. Must be a prime number.
- h: The cofactor $\frac{E(F_q)}{n}$.

Algorithms: Algorithm 1 shows Key Generation, Algorithm 2 shows Message Signing and Algorithm 3 shows Signature Verification stages according to [21].

Algorithm 1 ECDSA Key Generation

INPUT: Domain Parameters
1. *Private key* $\leftarrow d \in R[1, n-1]$
2. *Public key* $\leftarrow Q = dG \in E(F_q)$
OUTPUT: Private key d, public key Q.

Algorithm 2 ECDSA Message Signing

INPUT: Domain Parameters, message M, signer private key d
1. *Session key* $k \leftarrow A$ randomly chosen number in $[1, n-1]$
2. $R \leftarrow kG$
3. $r \leftarrow x(R) \mod n$
4. $s \leftarrow k^{-1}(H(M) + r \times d) \mod n$
OUTPUT: ECDSA signature (r, s).

Algorithm 3 ECDSA Signature Verification

INPUT: Domain Parameters, message M, signer public key Q, and signature (r, s)
1. $w \leftarrow s^{-1} \mod n$
2. $u \leftarrow H(M)w \mod n$, $v \leftarrow rw \mod n$
3. Calculate $R \leftarrow uP + vQ \in E(F_q)$
4. Accept if and only if $x(R) = r \mod n$.
OUTPUT: Accept/Reject

3.2 Performance Experiments

Our experiments for ECDSA first compares RSA and ECDSA with the same security level. Then we see the speed performance of ECDSA with different elliptic curves and key sizes, and comparing with RSA using the same level of security. Finally we state three different implementation approaches.

Public Key Security Level

As mentioned before, ECDSA uses Elliptic Curve Cryptography (ECC) to realize public key cryptosystems and public key infrastructures. Its strength-per-key-bit is significantly greater than RSA for this reason. Figure 2 table shows the comparable strengths for Symmetric, RSA and ECC ciphers by their key sizes [18]. We can see from Figure 3, key size used for generating a secure RSA are much larger than ECC even for the low security levels.

For example, the lowest level of security people can accept today is the 1024 bits RSA, and it reaches 80 security level. However if the cipher uses ECC, it only takes 160-223 bits to reach the same level of security. In [22], the authors give us a more comprehensive explanation about how the key sizes are obtained for different security levels using each type of cipher. The smaller key size not only refers to the faster computations that can be executed, but also helps reduce the bandwidth, storage space, processing power, and power consumption.

Security Strength	Symmetric Key Algorithms	DSA Key Size	RSA Key Size	ECC Key Size
≤ 80	2TDEA	L = 1024, N = 160	1024	160-223
112	2TDEA	L = 2048, N = 224	2048	224-255
128	AES-128	L = 3072, N = 256	3072	256-383
192	AES-192	L = 7680, N = 384	7680	384-551
256	AES-256	L = 15360, N = 512	15360	512+

Figure 2: NIST Comparable Key Strengths

Speed Performance

Our objective is to observe how ECDSA performs with different key sizes in Key Generation, Message Signing and Signature Verification process. Several choices have to be made before implementing an ECDSA system, including selection of elliptic curve domain parameters, algorithms for elliptic curve arithmetic, protocol arithmetic and field arithmetic.

In our experiment, we use FIPS 186-4 [3] revised by NIST to include ECDSA as specified in ANSI X9.62. We use five prime fields

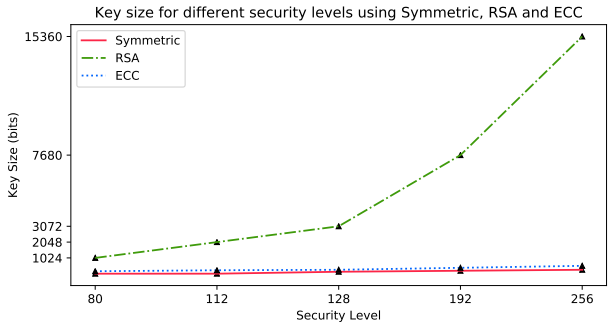

Figure 3: NIST key size requirement for different security strength using Symmetric, RSA and ECC

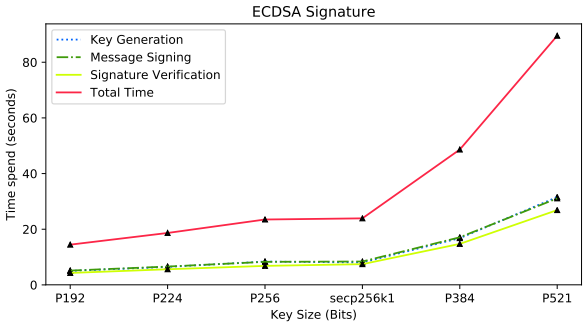

Figure 4: ECDSA performance under different Elliptic Curves with fixed message size 6000

and one randomly selected elliptic curve for each fields recommended by NIST FIPS 186-4. The curves are denoted as $p192$, $p224$, $p256$, $p384$, $p521$.

$$p192 = 219^2 - 2^{64} - 1$$
$$p224 = 2^{224} - 2^{96} - 1$$
$$p256 = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$$
$$p384 = 2^{384} - 2^{128} + 2^{96} + 2^{32} - 1$$
$$p521 = 2^{521} - 1$$

Since those curves use different key sizes, we can do the experiment with them to observe the speed performance of varying key sizes. Additionally, for the 256 bit size we add another curve $secp256k1$ defined in Standards for Cryptography (SEC), since it is used in Bitcoin public-key cryptography. Unlike other NIST Curves, $secp256k1$ was constructed in a predictable and non-random way, which makes the computation more efficient. The dataset we used is the file that contains 6000 lines of messages. And the implementation we used is FastECDSA, which will be discussed further in Implementation session. Figure 4 and Figure 5 table show the experiment result of ECDSA Performance under Different Elliptic Curves (Key sizes).

From the table and graph, we observed that for each individual curve, the time for performing all three stages in ECDSA are fairly the same. For smaller key sizes from 192 to 256, the time for generating, signing and verifying differ only a little. After size 256, time for the three stages grows significantly.

RSA vs ECDSA

Tables in Figure 6, 7, 8 compare the time using for Key Generation, Message Signing, Signature Verification with comparable key sizes for ECDSA and RSA. We use message size of 1 (i.e. signing and verifying 1 line of message). Both implementations of the two digital signature schemes are using OpenSSL and the same one message to ensure the comparability. We draw those data in Figure 9 to see the trends. From the result, ECDSA outperforms RSA in Key Generation and Message Signing stages, but underperforms RSA in Signature Verification stage. To reach the same security level in Key Generation stage, the 15360 bits RSA has to take 66.67 seconds while 512 bits ECDSA only takes 0.14 seconds, significantly faster. In Message Signing stage, at the beginning RSA is faster than ECDSA. As the key sizes increasing, however, ECDSA speeds up and surpasses RSA and RSA slows down. Finally, when the key size is very large, ECDSA still outperforms RSA a lot. In the Signature Verification stage, ECDSA takes considerably much longer time than RSA. And for the small key size, the time used by RSA for the verification are barely changed as the key sizes increasing.

Key Length		Key Generation Time	
RSA	ECDSA	RSA	ECDSA
1024	192	0.078125954	0.00603008270264
2048	224	0.117223978	0.0019359588623
3072	256	0.545434952	0.00864791870117
7680	384	29.17641091	0.0156280994415
15360	521	66.67729783	0.1413789650403

Figure 6: Key Generation Time for RSA vs ECDSA

Key Length		Message Signing Time	
RSA	ECDSA	RSA	ECDSA
1024	192	0.004239798	0.00493788719177
2048	224	0.008449793	0.00101184844971
3072	256	0.014075041	0.00778484344482
7680	384	0.164775848	0.0150728225708
15360	521	0.420565128	0.1403407546702

Figure 7: Message Signing Time for RSA vs ECDSA

Key Length		Signature Verification Time	
RSA	ECDSA	RSA	ECDSA
1024	192	0.000551939	0.00353193283081
2048	224	0.00069499	0.000929117202759
3072	256	0.00050211	0.00549483299255
7680	384	0.001395941	0.00981307029724
15360	521	0.002123117	0.32059066723145

Figure 8: Signature Verification for RSA vs ECDSA

From the above analysis we can conclude that ECDSA with the smaller key size can save a lot of resources such as time and space

		Key Size					
		P192	P224	P256	Secp256k1	P384	P512
Time (Second)	Key Generation	5.029524803	5.029524803	8.379398108	8.379398108	16.7968998	31.55780387
	Signing Message	5.136360168	5.136360168	8.284299135	8.397820234	17.10064411	31.08830404
	Verify Signature	4.291093349	5.615871191	6.830735445	7.451044321	14.7343595	26.87315464
	Total Time	14.45697832	18.65773344	23.49443269	23.90831614	48.63190341	89.51926255

Figure 5: ECDSA performance under different Elliptic Curves with fixed message size 6000

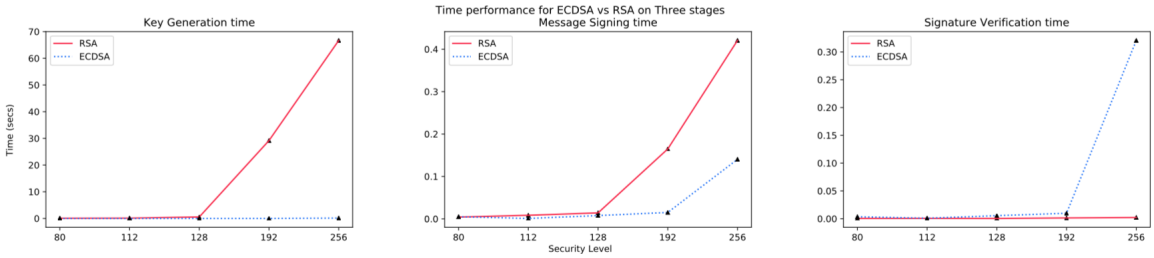

Figure 9: Time Performance for ECDSA vs RSA on Three Stages

storage, which is better than RSA. RSA can be used in the case when the receiver needs to verify more messages than the number that are produced.

Implementation Method

We use three different packages found online to implement ECDSA process. The first one is implemented purely in Python and released under the MIT license [23]. When trying to use this package to implement ECDSA, we find out it takes significantly longer time to all of the three stages. Then, we explore online and decide to adopt the second approach, FastECDSA, which is implemented in C [24]. For third method, we use the python cryptography library that has OpenSSL backend [17]. As shown in Figure 10, the pure python package takes substantially much time than the other two approaches, especially in Key Generation and Signature Verification stages. The package that uses OpenSSL has the best performance in all stages.

Figure 10: ECDSA Performance under Different Implementation

3.3 Security Considerations

The most important security considerations when using ECDSA is to against the chosen-message attack. Attacker obtains the sender's signature on a collection of messages (excluding m) and his goal is to obtain a valid signature on a single message m. We classify the possible attacks into three categories as follows.

Per-message Secrets

One attack in Signature Generation stage is related to the per-message secrets k. If the same secret k is using many times, attackers can easily break ECDSA and obtain the entities' secret keys. The mechanism is shown in Breaking ECDSA Algorithm 4 below.

Algorithm 4 Breaking ECDSA

Breaking ECDSA with static per-message secret k

$s1 \leftarrow k^{-1}(z1 + r \times dA) \mod n$

$s2 \leftarrow k^{-1}(z2 + r \times dA) \mod n$

$s1 - s2 = k^{-1}(z1 - z2) \mod n$

$k = (s1 - s2)^{-1}(z1 - z2) \mod n$

Sony Corporation used a static k as the secret to generate all the signatures in August 2013. Initially only the signed games can run on PS3 game console. After the attacker break ECDSA and recover the secret key, everyone can play every games. This issue can be avoided by destroying after each message is signed.

ECDLP Attacks

To ensure ECDSA is secure, the elliptic cryptography discrete logarithm problem (ECDLP) cannot be easily solved. It is defined in Key Generation stage as solving for d in $Q = dG$. Within many known attacks against ECDLP, there are four very famous ones. Exhaustive search, Pohlig-Hellman, Pollard-rho and Multiple logarithms, which are described details in [19].

Other Attacks

The hash function that is used to encrypt the original message is better to be collision resistant. Traditional hash function such as SHA-1 is not collision resistant nor preimage resistant, thus are weak to the attackers. Implementation attacks such as timing attacks and differential fault analysis could happen as well.

4 EDDSA

Edwards-curve Digital Signature Algorithm (EdDSA) is a digital signature scheme based on (possibly twisted) Edwards curves, which is a variant of Schnorr's signature system. It has been proven to provide high performance over a variety of platforms and be securer because it is more resilient to side-channel attacks (including timing attack etc.) and provides collision resistance as well (only holds for PureEdDSA). Also, it is space efficient since it only requires small public/private key pair (32 or 57 bytes) and produces short signatures (64 or 114 bytes) and can achieve around 128-bit and 224-bit security level separately.

4.1 Parameters

General EdDSA signature scheme has 11 parameters.

(1) p, an odd prime and EdDSA exploits an elliptic curve on the finite field $GF(p)$.
(2) b, EdDSA uses b bits for public/private keys and produces $2 * b$ bits for signature and $2^{(b-1)} > p$.
(3) $(b-1)$-bit encoding of elements of $GF(p)$.
(4) H, a hash function that produces $2 * b$-bit output.
(5) c, co-factor of base-2 logarithm.
(6) n, secure EdDSA schemes have $n+1$ bits with $c \leq n < b$.
(7) d, non-square element of $GF(p)$, which is always nearest to zero.
(8) a, non-square element of $GF(p)$, which would be 1 if $p \mod 4 = 1$ and; -1 if $p \mod 4 = 3$.
(9) B, an element from $set\{(x, y)\}$, a member of $GF(p) * GF(p)$ where $a * x^2 + y^2 = 1 + d * x^2 * y^2$.
(10) L, an odd prime which $[L]B = 0$ and $2^c * L = \#E$, where $\#E$ is the number of points on the curve).
(11) PH, pre-hash function which generates a fixed length of message before getting into EdDSA scheme, where $PH(M) = M$ means no pre-hash function.

Ed25519 has been chosen for our experiment since it is most widely used instance of EdDSA and supposed to provide a 128-bit security level, which is perfectly safe and can defeat many side channel attacks as a PureEdDSA. It takes SHA-512 as hash function and no pre-hash function, and exploits Curve25519 where:

$$-x^2 + y^2 = 1 - 121665/121666(x^2y^2)$$

4.2 Algorithm

EdDSA uses a b-bit long private key and a hash function H, producing a $2b$-bits output. One of the most common hash function used here is SHA-512 (b = 256), which is used by Ed25519. $H(k)$ can be represented as $(h_0, h_1, \ldots, h_{2b-1})$. Therefore, the signing process of message M can be expressed as shown in Algorithm 5 where A is the public key and B is the base point of order l and $A = a \cdot B$

Algorithm 5 EdDSA Algorithm

Input: M,$(h_0, h_1, \ldots, h_{2b-1})$, B and A
1. $a \leftarrow 2^{b-2} + \sum_{3 \leq i \leq b-3} 2^i h_i$
2. $h \leftarrow H(h_b, \ldots, h_{2b-1}, M)$
3. $r \leftarrow h \mod l$
4. $R \leftarrow r \cdot B$
5. $h \leftarrow H(R, A, M)$
6. $S \leftarrow (r + ah) \mod l$

return (R, S)

For verifying, a signature should be valid if $R \in E, S \in \{0, 1, \ldots, l-1\}$ and meet the equation:

$$8S \cdot B = 8 \cdot R + 8H(R, A, M) \cdot A$$

And it can be shown that it takes much harder mathematics computation to verify a signature.

4.3 Performance

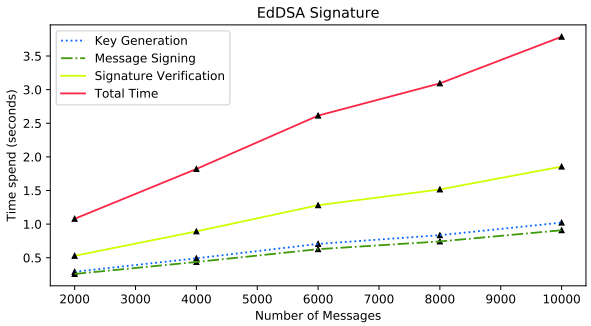

Figure 11: EdDSA Performance Runtime

Unlike RSA and ECDSA which can alter parameters to evaluate the performance, the EdDSA algorithm has fixed parameters once we decided to apply Ed25519 in the experiments. Therefore, our focus of EdDSA's performance is on the time it takes in different stages (key generation, sign, verify). As depict in Figure 11, it takes only at most 4 seconds for 10000 messages. The runtime benchmarking shows in Figure 11 uses the PyNaCL [25] implementation that run over varying messages of 2000, 4000, 8000 and 10000 (i.e. varying load). Since it is very fast algorithm, we run with varying messages to observe sensible performance measure.

Furthermore, we found three implementations for EdDSA and intended to figure out how different implementation might affect the performance. The first one we found is ed25519 python package, provided by Brain Warner [26], it is a simple and straightforward implementation, however, it consumes much time on our testing environment. Then, we found two other python packages PyNaCL [25] and LibNaCL [27], which also provide ed25519 signature algorithm, but much faster implementation. We compared these three implementations with varying message length and record total

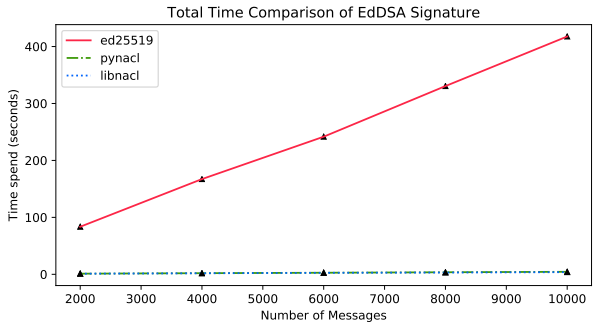

Figure 12: Total Time of EdDSA Implementations

		Message Size				
		2000	4000	6000	8000	10000
Time (Second)	Ed25519	83.52224064	167.219785	241.6784384	330.5522776	417.6510758
	PyNaCL	1.12104845	1.965936422	2.764304876	3.59608507156	4.428426504
	LibNaCL	1.079045057	1.819925785	2.613591671	3.092978954	3.78670311

Figure 13: Total Time of EdDSA Implementations Table

time of key generation, signing all messages and verifying all signatures and experiment details are shown in Figure 12 and table in Figure 13.

It is clear that all three implementations' total execution time grows almost linearly with input size increasing, which means input size has no influence with the individual execution time for each message no matter what implementation is applied.

Besides, ed25519 package takes much more time than the other two packages, therefore, different implementations do have big influence in terms of total time even though we can be sure that we all implement same algorithm (provide same level security), because they all just offer a python interface to a C implementation of Ed25519 public key signature system (http://ed25519.cr.yp.to) and refer to original paper [4], provided by creators of this algorithm, Bernstein et al.

So as to understand how much each step contributes to the total time for three implementations, we used the fixed message size of 6000 lines (i.e. fixed load), and evaluated execution times for each implementation as shown in Figure 14. It is obvious that key generation and signing processes almost take the same amount of time, however, verifying process takes much more time than these two, which is two to three times longer. As we analysed before, the verifying process needs much more complicated mathematics computation, thus, it is reasonable that it takes much longer time.

4.4 Security

EdDSA is proven to provide better security thanks to nice properties of twisted Edwards curves it works over. Particularly, Ed25519 exploits the Montgomery curve "Curve25519", which does not apply branch operations or array indexing steps which depend on secret data to implement its scalar multiplication, in order to avoid many side channel attacks [28].

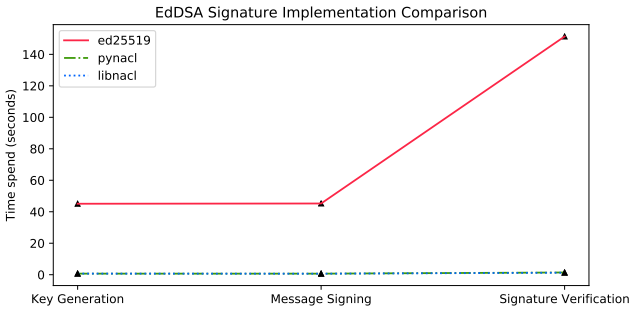

Figure 14: Steps comparison of EdDSA implementations over fixed message size 6000

However, it might still be vulnerable to fault attacks. In [28], they perform a valid attack on the signing algorithm above and proposed a fault algorithm as shown in Algorithm 6.

Algorithm 6 Fault Algorithm to attack on EdDSA

Input: M, A, (R, S) and (R, S')
$h \leftarrow H(R, A, M)$
$i \leftarrow 0$
for i < 32 **do**
  $e \leftarrow 1$
  for e < 256 **do**
    $h' \leftarrow 2^{8i}e \oplus h$
    $a \leftarrow (S - S')(h - h')^{-1} \mod l$
    if $a \cdot B == A$ **then**
      return a
    end if
    $e \leftarrow e + 1$
  end for
  $i \leftarrow i + 1$
end for
return ERROR

Then, they found that by selecting r as a random number, they can compute a new (R, S) as a signature for the message M where the equation would be:

$$8S \cdot B = 8(r + H(R, A, M)a) \cdot B = 8 \cdot R + 8H(R, A, M)a \cdot B$$
$$= 8 \cdot R + 8H(R, A, M) \cdot A$$

Therefore, the verifying equation still holds for this, which means we can forge valid signatures for messages. It illustrates that EdDSA is still vulnerable to fault attacks, and protections or countermeasures against them should be considered if the algorithm is applied in areas where fault attacks are possible.

5 LAMPORT SIGNATURE SCHEME

Lamport one-time signature scheme is an efficient method for constructing digital signature, which can resist quantum computing. The currently used digital signature schemes rely on the algorithmic

complexity which is fragile when confronting quantum computer. Lamport signatures can be built from any cryptographically secure one-way hash function. Strength of Lamport signature, however, depends on the strength of hash function it utilized in implementation.

5.1 Implementation Details

The Lamport signature scheme performs its working in three steps.

Keys Generation: Use random number generator to produce 256 key pairs of random numbers as private key, each number is 256 bits, total 16 KiB. The public key is the corresponding hash value (SHA256) of private key, thus implementing 512 hashes, each 256 bits in size, public key has same size with private key.

Message Signing: Create a 256-bit hashsum of the message. For each n-th bit in the hashsum, pick a corresponding number from the private key. For example, if the n-th bit is 0, pick an n-th number in the first row, if the n+1th bit is 1, pick the n+1th number from the second row. This way produces 256 numbers each of 256 bits, the signature size is total 8 KiB.

Signature Verification: Create a 256-bit hashsum of the message. Pick the corresponding public key for each bit, then calculate hash value of each number in signature. If hash values are same as the values picked from public key, it indicates the message received is original, otherwise, the message or signature has been tampered during the transmission or does not belong to sender.

The Figure 15 illustrates every step.

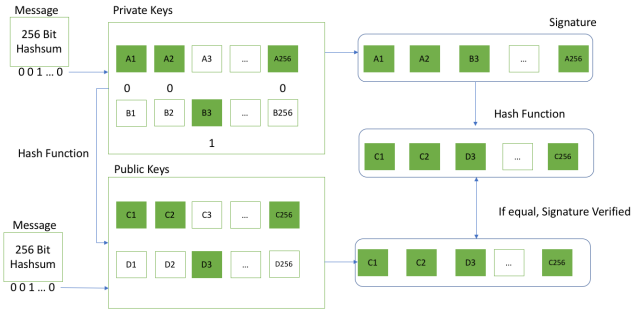

Figure 15: Implementation of Lamport Signature Scheme

5.2 Mathematical Notation

Keys: Let k be a positive integer and let $m = \{0,1\}^k$ be the hashsum of the message. Let $f : Priv \rightarrow Pub$ be a one way hash function. For $1 \leq i \leq k$ and $j \in \{0,1\}$, the sender computes hash value of each random number $Priv_{i,j}$, and get $Pub_{i,j} = f(Priv_{i,j})$. The size of private key is 2k, the size of public key is 2k as well.

Signing a message: Let $m = m_1 \ldots m_k \in \{0,1\}^k$. The signature of the message is:

$$sig(m_1 \ldots m_k) = (Priv_{1,m1}, \ldots, Priv_{k,mk}) = (s_1, \ldots, s_k)$$

Verifying a signature: Receiver validates a signature by checking $f(s_i) = Pub_{i,mi}$, for all $1 \leq i \leq k$.

5.3 Characteristics

Strength: An important factor in the strength of Lamport signature scheme is flexible. If a hash function that is employed with the scheme is proven to be weak, it can easily be replaced with another strong hash function. The common hash used in Lamport signature scheme is SHA256, which is proved as secure in most scenarios.

Lamport signature scheme is quantum resistant due to a large number of hash functions it employees in its implementation. Other digital signature schemes such as RSA, ECC, whose credibility depends on their algorithm complexity. They are much prone to be broken by quantum computer [29]. The increasing development towards a practical computer is a threat to such signature schemes.

Lamport signature scheme costs less time than traditional signature scheme based on public key cryptography. It is suitable to apply on the resource-constrained devices like IoT devices.

Weakness: Large computation and storage requirements are the main limitations of Lamport signature. For example, sender randomly generates 256 pairs of 256-bit numbers as private key, the total size of these pairs equals $2 * 256 * 256 = 16$ KiB. Sender needs to calculate 512 times hash of private key then get corresponding public key, which has same size 16KiB. The signature's size is the half of private key or public key, that is 8KiB. Compared to other signature scheme like RSA, the signature size is relatively large. Similarly, receiver needs to calculate 256 times hash of signature in verification process. Besides, this is one time scheme, key pairs cannot be used next time. It indeed requires a large memory to store these key pairs in signing process. Lamport signature scheme puts more loads on devices' storage and computation compared with traditional signature schemes [30].

The key pair can be only used to sign one message each time. If there are multiple messages to be signed, many key pairs have to be generated for a safety purpose.

5.4 Experiment

The objective of this experiment is to measure the execution time of Lamport Signature algorithm and find which step costs most of time. The program was written in Python, and no package was used. The test data set includes three different size csv files, which are 1000 entries, 5000 entries and 10000 entries. The default hash function is SHA256. The execution time of the program is as shown in Figure 16 table.

		Message Size		
		1000	5000	10000
Time (Second)	Key Generation	6.931	28.146	56.981
	Signing message	0.453	1.845	3.84
	Verify Signature	0.0004	0.0018	0.004
	Total Time	7.391	30.017	60.887

Figure 16: Lamport Runtime Table

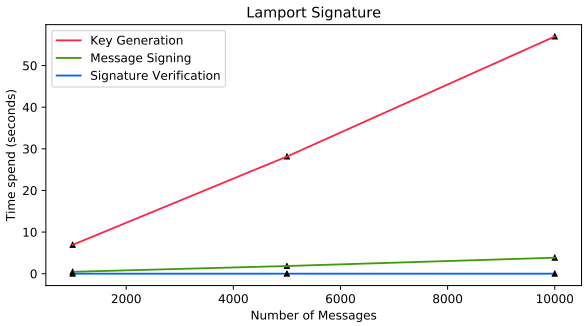

Figure 17: Lamport Runtime Graph

From the chart in Figure 17, we can observe that key generation accounts for large part of time, nearly 90% of total time, the performance of Lamport signature scheme is dominated by key generation time. Because large amount of hash calculations are in key generation step, for this case, there are 512 random numbers in public key, so there are total 512 times SHA256 calculations. While in sign message step, the program only calculates the hashsum of message. In verifying step, the program only calculates 256 times SHA256. Therefore, it is obvious that key generation is the most time-consuming step in Lamport signature.

5.5 Security Analysis

The security performance of Lamport signature is dominated by its hash function. For a hash function that generates an n-bit message digest, according to Grover's algorithm [31], finding a pre-image collision on a single invocation of an ideal hash function is upper bound on $(2^{n/2})$ operations under a quantum computing model. For each private key and its corresponding public key pair in Lamport signature, the private key length must be selected so performing a pre-image attack on the length of the input is not faster than performing a pre-image attack on the length of the output. For example, if each private key element was only 16 bits in length, it is trivial to exhaustively search all 2^{16} possible private combinations in 2^{16} operations to find a match with the output. Based on Grover's algorithm, a quantum secure system, the length of the public key elements and private key elements and the signature elements must be no less than 2 times larger than the security rating of the system [32]. An 80-bit secure system uses element lengths of no less than 160 bits, a 128-bit secure system uses element lengths of no less than 256 bits. SHA256 is commonly used in Lamport signature, SHA512 is much safer but at the cost of longer key generating time and larger signature size.

5.6 Improvement

Large computation and storage requirement in Lamport's key generation process are the main disadvantages of this scheme. Other one-time signature schemes like Merkle, HORS, Merkle-Winternitz improve Lamport efficiently, even if they all have their pros and cons. For example, each Lamport public key can only be used to sign one single message, which means many keys have to be published if many messages are to be signed, Merkle tree structure can be used for these public keys, only publishing the top of the hash tree instead, a single hash can be used to verify any given number of future signatures, this mechanism indeed decrease key storage requirement.

6 CROSS COMPARISON ANALYSIS

To evaluate performance between RSA, ECDSA, EdDSA and Lamport digital signature schemes, we perform the cross comparison runtime analysis of these algorithms with their fixed "ideal" setting. Here, the "ideal" setting, we mean that the best performance to security ratio and trade-off. However, we also note that it is still challenging to justify the ideal setting because some of these algorithms work quite differently in their mathematical foundation, pre-processing and algorithmic steps. Table in Figure 18 summarise the setting and parameter of each algorithm and, we label them $A1 \ldots A8$ accordingly. We highlight that even though we do not review particularly to the Digital Signature Algorithm (DSA), but we include DSA in our runtime performance analysis to observe better for the purpose of supplementing stronger argument for ECDSA runtime. The justification of the "ideal" setting is based on our best effort evaluating each algorithm by a combination of our collaborative investigation and, by reviewing a couple of related literature as cited in references.

Our quantitative performance measurement methodology works as follows:

(1) We prepare test dataset in 10 CSV files: each file contain {1000, 2000, …, 10000} lines of messages.
(2) We read the file, iterate through line by line.
(3) For each line, we generate a fresh key pair, sign the message and perform signature verification subsequently.
(4) At each step, we record the time delta before and after; accumulate the respective counters.
(5) After the last line, we print the accumulated total time, key generation time, signing and verification time.

Krypton Cloud VM

Figure 20 and tables in Figure 21 show the runtime benckmarking of algorithms over varying work loads on Krypton NeCTAR Cloud VM instance. For the key generation, DSA (A8) and RSA (A7) takes the longer time which influence their total runtime. For message signing, ECC based algorithms (A3, A4, A5) takes longer time as well as RSA (A7). For the verification time, ECC based algorithms (A3, A4, A5) also takes longer time, but RSA (A7) achieve comparatively fast verifying time. Over the measurement on Cloud VM, particularly, the A3 (`eddsa_25519.py`) has peculiar runtime on message singing and verification steps. If we compare among different EdDSA implementations, the backend that use `NaCL/LibSodium` does not reflect such slow runtime on Cloud vCPU. We conclude that this anomaly is due to the fact that python-ed25519 [26] implementation makes use of SUPERCOP (http://bench.cr.yp.to/supercop.html) backend, of which, we assume the C implementation could be tightly coupled to *physical* CPU architecture for benckmarking purpose.

MacBook Pro

We further verify this hypothesis by running the same setup and benchmarking on the laptop which is MacBook Pro with CPU

Intel Core i5 3.1GHz and 16GB of RAM, running macOS Mojave. We make sure that the MacBook Pro has no compute intensive task running background and optimal, while performing the runtime benchmarking. Figure 22 and tables in Figure 23 show the result of running with the *physical* CPU. The result shows highly correlation with the Krypton Cloud vCPU result in all algorithm implementation, except A3 (`eddsa_25519.py`). The A3 runtime on *physical* CPU result shows that it run much faster than its vCPU counterpart.

We observe that the nature of cryptographic algorithms are compute intensive and affinity to the type of CPU used. In this case, there will be significant differences in *virtual* CPU vs *physical* CPU. However, we empirically justify that Cloud Computing is found everywhere [33] and, the usage of these cryptographic algorithms on such Cloud platform become norm and ubiquitous deployment. We also further proof that the quality aspect of these algorithms are CPU agnostic in such that we can generally say ECC algorithms usually take more time on signing and verification process; similarly, RSA and DSA algorithms usually take more time on key generation step regardless of CPU type or what platform they run on. This outcome correlate with the mathematical characteristic of each algorithms as we discussed in previous sections.

7 CONCLUSION

Table in Figure 19 summarise the qualitative measures of our evaluation outcome among RSA, ECDSA, EdDSA and Lamport digital signature schemes.

Computational Overhead: In term of compute overhead, RSA has to consider padding and randomness scheme such as PSS. ECDSA inherits complex computation in its signing and verification process. EdDSA has to consider the built-in hash computation and its scalar multiplication. Lamport signature needs many hash calculations.

Security Strength: In term of security strength, all algorithm can withstand NIST rating of security strength 128-bit [18], which is similar difficulty to breaking strong 128-bit block ciphers. With 2^{128} security target, RSA with key size 3072-bit, ECDSA with P-256, EdDSA with Ed25519 and Lamport signature with SHA-256 can achieve fairly strong security strength without compromising too much performance.

Key Size: For the key size requirement, we rate RSA is medium, ECDSA key size is small, EdDSA key is the smallest and Lamport signature requires large key size.

Signature Size: For the signature size in bytes comparison, RSA is medium, both ECDSA and EdDSA have comparatively smaller signature size than RSA, and Lamport signature is the largest.

Key Generation Time: In term of key generation time, RSA is the slowest, ECDSA is fast, EdDSA is the fastest and Lamport signature is slow. However, in real world RSA deployment, key generation is one-time effort as RSA key pair are re-used as long as private key is not compromised.

Signing Time: For signing time, RSA and ECDSA are slow, Lamport is fast and EdDSA is the fastest.

Verification Time: For signature verification, RSA and EdDSA are fast, ECDSA is slow and Lamport signature has the fastest verification time due to its solely use of hash values comparison.

Quantum Computing Resistant: In term of future proof, only Lamport signature has the quantum-safe computing property due to its solely use of one-way hash property. Computational complexity based RSA and ECC algorithms are prone to Shor's algorithm [5] attack running on powerful quantum computing.

Application Domain: In term of deployment, RSA is still dominant algorithm used in many Client-Server distributed systems such as certificate signing, secure-shell protocol, IPSecurity, so on. ECDSA is popular in blockchain technology and, especially employ in BitCoin (https://bitcoin.org) cryptocurrency. Lamport is popular in embedded and IoT device as well as when quantum proof classified secrecy is desired. EdDSA is popularly used in signing code and software releases and it has growing deployment as reported in [34].

Industrial Year: RSA, ECDSA and Lamport signature schemes have more than 30-years of industrial presence across many different type of deployment in applications and domains. Based on our experiment, EdDSA is the smallest and fastest digital signature scheme with comparable security strength to its counterpart. However, EdDSA is fairly new algorithm and, therefore, it has yet to be tested thoroughly to claim its superiority among digital signature schemes.

Based on the experiment presented, we conclude that our research contribute quantitative measurement and qualitative evaluation of the chosen digital signature schemes, observation over their characteristic and mathematical foundation and, critical analysis on three way implementation-performance-security trade-off and their applicable domains.

7.1 Future Work

For the future work, we would like to research more on definitive security evaluation of digital signature schemes. Power consumption observation is also another interesting evaluation metric to undertake experiment for deploying digital signature to ubiquitous mobile computing and IoT technology.

7.2 Source Listing

- https://github.com/victorskl/crypto-dsse
- https://github.com/victorskl/crypto-research-proposal
- https://github.com/victorskl/crypto-research-report
- https://youtu.be/jiIY31YrB-A

Alias	A1	A2	A3	A4	A5	A6	A7	A8
Program Name	eddsa_libnacl.py	eddsa_pynacl.py	eddsa_25519.py	ecdsa_pyca.py	ecdsa_fast.py	Lamport.py	rsa_pyca.py	dsa_pyca.py
Hash	SHA512	SHA512	SHA512	SHA256	SHA256	SHA256	SHA256	SHA256
ECC Curve	Ed25519	Ed25519	Ed25519	SECP256K1	SECP256K1	-	-	-
Key Size	-	-	-	-	-	-	2048	1024
Public Exponent (e)	-	-	-	-	-	-	65537	-
Padding	-	-	-	-	-	-	PSS	-
Backend	NaCL/libsodium	NaCL/libsodium	SUPERCOP	OpenSSL	C implementation	Python hashlib	OpenSSL	OpenSSL

Figure 18: Cross Comparison Algorithms Setting and Parameter

	RSA	ECDSA	EdDSA	Lamport
Computational Overhead	Padding	Signing and Verification Process	Hash Function, Scalar Multiplication	Many Hash Calculations
NIST Security Strength (128-bit)	Yes (k = 3072-bit)	Yes (NIST P-256)	Yes (Ed25519)	Yes (SHA-256)
Key Size	Medium	Small	Very Small	Large
Signature Size (bytes)	Medium	Small	Small	Large
Key Generation Time	Very Slow	Fast	Very Fast	Slow
Signing Time	Slow	Slow	Very Fast	Fast
Verification Time	Fast	Slow	Fast	Vey Fast
Application Domain	Client-Server Deployment	Blockchain, Bitcoin	Software Release	IoT, Wearable Devices
Quantum Computing Resistant	No	No	No	Yes
Industrial Years	>30	>30	~6	>30

Figure 19: Qualitative Evaluation of Digital Signature Schemes

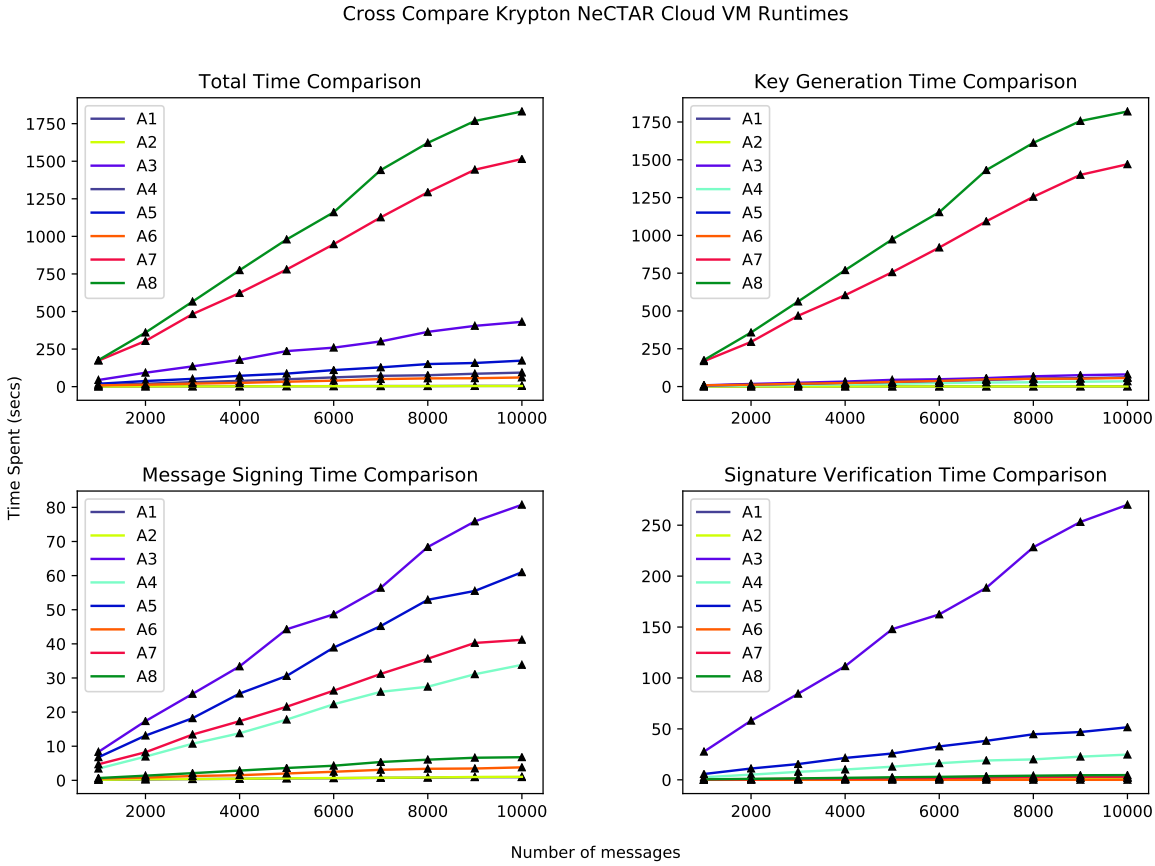

Figure 20: Cross Compare Krypton NeCTAR Cloud VM Runtimes

Krypton NeCTAR Cloud VM

	Total	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
eddsa_libnacl.py	A1	0.737154007	1.22525692	1.843674898	2.268166065	2.82523489	3.164525986	3.7466681	3.92343688	4.818909168	4.905833006
eddsa_pynacl.py	A2	0.730074167	1.292459011	1.587222099	2.027951002	2.494320869	2.852205992	3.2395432	3.66467905	4.218097925	4.339344978
eddsa_25519.py	A3	44.23239493	92.5908041	134.923605	178.0696239	236.5663102	259.4139152	301.039326	364.570596	404.3988709	431.1713789
ecdsa_pyca.py	A4	9.508374214	19.22832894	29.73819685	38.218153	48.94185901	61.59703803	71.84170604	75.81750607	85.87281489	93.41846585
ecdsa_fast.py	A5	19.16612506	37.22764683	51.79768205	72.10210681	86.82713103	110.138643	128.3171399	150.2560959	157.417125	173.1205812
Lamport.py	A6	8.625602007	12.77343917	20.06960511	24.7940731	32.76689005	39.8557899	50.27370286	54.95673108	55.98384285	60.97025394
rsa_pyca.py	A7	173.057776	303.918386	482.6531432	623.003083	779.521966	947.7326779	1126.465735	1293.344769	1443.542276	1514.745211
dsa_pyca.py	A8	176.3814411	359.8523071	565.7936041	774.8677142	979.2884769	1160.42264	1440.992059	1622.11969	1767.872176	1830.841914
	Key Gen	**1000**	**2000**	**3000**	**4000**	**5000**	**6000**	**7000**	**8000**	**9000**	**10000**
eddsa_libnacl.py	A1	0.156105757	0.260820866	0.39507246	0.482633829	0.611633539	0.676994801	0.82226491	0.841701269	1.054172993	1.044581652
eddsa_pynacl.py	A2	0.1709795	0.306952238	0.383260012	0.49313736	0.590542078	0.680860281	0.77125597	0.872943163	1.047251701	1.026942253
eddsa_25519.py	A3	8.248426199	17.19734383	25.07150364	33.08139396	44.34894276	48.30541992	56.03979969	67.77828288	75.28037786	80.32297444
ecdsa_pyca.py	A4	3.542912006	7.184781075	11.16995263	14.25402331	18.22980762	22.95973539	26.84980297	28.28008723	31.97301388	34.76207709
ecdsa_fast.py	A5	6.682162046	12.9932282	18.10763311	25.17496347	30.35726547	38.39182997	44.7647953	52.5189352	54.92479491	60.44519401
Lamport.py	A6	8.104088306	11.96241832	18.78673029	23.26437402	30.72850919	37.29274035	47.14455986	51.52258611	52.4431653	57.12043858
rsa_pyca.py	A7	167.9972985	295.0505171	468.2449756	604.3874695	756.3170781	919.4577258	1092.952193	1254.942085	1400.218388	1470.4019
dsa_pyca.py	A8	175.2587302	357.5281057	562.2470579	770.0276287	973.0873778	1153.139525	1431.949466	1611.911932	1756.666671	1819.299451
	Sign	**1000**	**2000**	**3000**	**4000**	**5000**	**6000**	**7000**	**8000**	**9000**	**10000**
eddsa_libnacl.py	A1	0.15125227	0.246485949	0.364190102	0.446854115	0.559139729	0.616137743	0.727957726	0.767063618	0.918316126	0.933630228
eddsa_pynacl.py	A2	0.167114019	0.303624868	0.376552105	0.476960897	0.59343791	0.673522949	0.761993647	0.846236706	0.983445883	1.028750181
eddsa_25519.py	A3	8.332835913	17.3562026	25.3406148	33.39835858	44.27300787	48.65641117	56.42387342	68.38452959	75.8982625	80.75785232
ecdsa_pyca.py	A4	3.434647083	6.951009989	10.72755408	13.79585552	17.79220295	22.29355311	25.93801761	27.43159056	31.09693837	33.8366406
ecdsa_fast.py	A5	6.77016449	13.11141658	18.20898891	25.42832851	30.59282279	38.9105897	45.22158718	52.91061449	55.51046705	60.99478269
Lamport.py	A6	0.514101505	0.79896903	1.265288591	1.508307457	2.009275913	2.511451006	3.083852768	3.383338928	3.465708971	3.800512552
rsa_pyca.py	A7	4.69955039	8.232041359	13.41190934	17.3273654	21.56529045	26.28354788	31.19480062	35.64369679	40.26828957	41.18370724
dsa_pyca.py	A8	0.663908005	1.365278482	2.102557898	2.855175734	3.614151478	4.281452656	5.367624283	6.055971146	6.61898303	6.777873993
	Verify	**1000**	**2000**	**3000**	**4000**	**5000**	**6000**	**7000**	**8000**	**9000**	**10000**
eddsa_libnacl.py	A1	0.278829575	0.466967583	0.706626892	0.874686003	1.077609301	1.224518538	1.432661772	1.506982565	1.893163919	1.917196751
eddsa_pynacl.py	A2	0.363619566	0.623026848	0.767989635	0.980233431	1.215940714	1.391645432	1.560788631	1.791536331	2.029885769	2.110004663
eddsa_25519.py	A3	27.64195919	58.02236676	84.4878509	111.564044	147.9148753	162.4192736	188.5399466	228.3188071	253.171926	270.0322185
ecdsa_pyca.py	A4	2.522385597	5.074881554	7.814785242	10.11325312	12.87537742	16.28888202	18.99220824	20.03667331	22.73068213	24.73947859
ecdsa_fast.py	A5	5.703709841	11.08345747	15.45728254	21.46638513	25.83404803	32.77934384	38.27114987	44.71440172	46.90944147	51.58055639
Lamport.py	A6	0.000527382	0.000808954	0.001246929	0.001625061	0.002091646	0.002445459	0.003165007	0.003414392	0.003759623	0.003972054
rsa_pyca.py	A7	0.344128847	0.611823559	0.955698013	1.234402657	1.561316252	1.891142845	2.212501049	2.641090393	2.920974731	3.000206232
dsa_pyca.py	A8	0.44617939	0.935027838	1.405872107	1.934474945	2.485359192	2.880518198	3.565968513	4.042778254	4.472308397	4.632399559

Figure 21: Cross Compare Krypton NeCTAR Cloud VM Runtimes Table

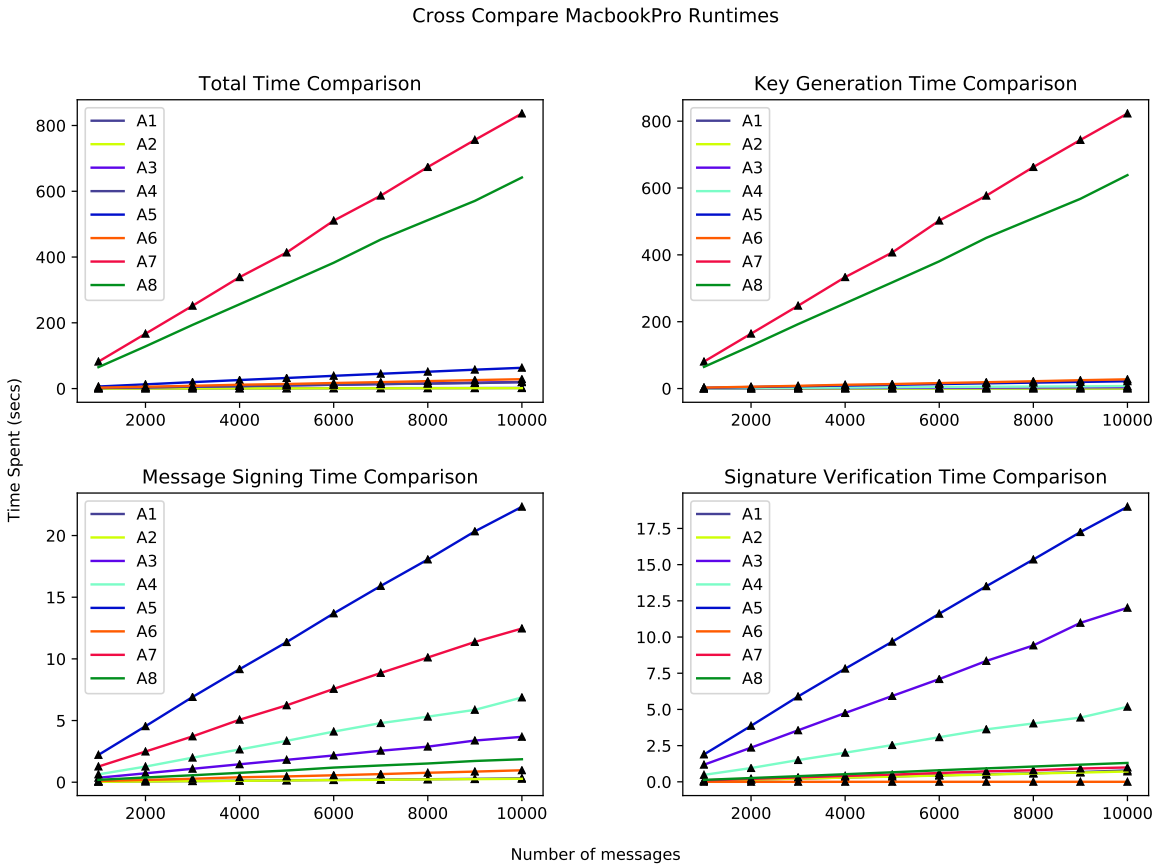

Figure 22: Cross Compare MacBookPro Runtimes

MacBook Pro

	Total	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
eddsa_libnacl.py	A1	0.206718922	0.379541159	0.528975964	0.692135096	0.86205411	1.049182892	1.222270966	1.376919985	1.560096979	1.825769901
eddsa_pynacl.py	A2	0.152851105	0.281042099	0.418936968	0.555644989	0.681886911	0.826855898	0.947084904	1.091937065	1.215368032	1.354569912
eddsa_25519.py	A3	1.944714069	3.889019012	5.864574194	7.844566822	9.778568983	11.69647408	13.77652287	15.53281307	18.15724015	19.82445908
ecdsa_pyca.py	A4	1.809518099	3.603374004	5.69143796	7.579918861	9.577972889	11.67656589	13.7069838	15.18559408	16.74096489	19.56933403
ecdsa_fast.py	A5	6.28902483	12.86617208	19.60539913	25.99380898	32.19214988	38.79615092	45.04453397	51.20495081	57.47877312	63.211092
Lamport.py	A6	2.818715096	5.608290911	8.376852989	11.71908498	13.86545801	16.91197705	19.59848309	22.64861798	25.79686999	28.5709219
rsa_pyca.py	A7	81.90022707	166.777112	251.8053041	338.7766261	413.750689	510.626514	586.6411462	673.6932731	756.1200838	836.4526689
dsa_pyca.py	A8	65.10449004	128.285274	193.6361361	256.3547549	319.2077911	382.652534	453.2070589	512.1021721	570.746681	641.9246881
	Key Gen	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
eddsa_libnacl.py	A1	0.045127153	0.083529949	0.115788937	0.149484158	0.186736107	0.228727579	0.265559673	0.299208879	0.337976217	0.393096685
eddsa_pynacl.py	A2	0.036203861	0.065115213	0.095468521	0.127623081	0.154390335	0.189043522	0.215926409	0.249575138	0.2765944	0.309524298
eddsa_25519.py	A3	0.403894663	0.807083845	1.215272665	1.634682417	2.031104803	2.424777508	2.868517876	3.221440077	3.785490751	4.113726377
ecdsa_pyca.py	A4	0.690999031	1.380112648	2.188527584	2.904364109	3.670716524	4.471916437	5.267480373	5.813014507	6.408162594	7.483846188
ecdsa_fast.py	A5	2.183817863	4.450957298	6.78805685	8.996607304	11.13446927	13.47504282	15.59460258	17.76041341	19.86233544	21.85093021
Lamport.py	A6	2.721197128	5.415254593	8.090016365	11.30872178	13.38536572	16.33769894	18.92476821	21.86547017	24.91525722	27.57961869
rsa_pyca.py	A7	80.52217102	164.0797007	247.7695403	333.2868845	406.9961543	502.4276929	577.025286	662.7485495	743.7849371	822.939276
dsa_pyca.py	A8	64.77281618	127.6238098	192.664844	255.0471773	317.5771177	380.636797	450.8726375	509.4243915	567.6894243	638.6054294
	Sign	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
eddsa_libnacl.py	A1	0.039107084	0.067910671	0.093609571	0.121930599	0.151086092	0.183926821	0.213475466	0.240676641	0.27256465	0.31815958
eddsa_pynacl.py	A2	0.031563997	0.057466745	0.086591959	0.113858223	0.140918255	0.170134544	0.194874048	0.22554183	0.250232458	0.277124882
eddsa_25519.py	A3	0.361277342	0.72112608	1.087680101	1.453286409	1.812977314	2.167640209	2.551542997	2.881452322	3.378044844	3.678430557
ecdsa_pyca.py	A4	0.636485815	1.265372038	1.989849806	2.652520418	3.352683306	4.102686644	4.791735649	5.313122511	5.866740942	6.87111783
ecdsa_fast.py	A5	2.213134289	4.539782524	6.908641338	9.164433241	11.36064005	13.6880393	15.91240621	18.05886006	20.33768797	22.33104086
Lamport.py	A6	0.095177174	0.18830061	0.279933691	0.399962902	0.469170809	0.561350822	0.657776833	0.764547348	0.86042428	0.967033863
rsa_pyca.py	A7	1.269771099	2.485003233	3.714429379	5.064259052	6.233247995	7.564002752	8.856570482	10.11113214	11.37029457	12.46019983
dsa_pyca.py	A8	0.19010663	0.384306192	0.563460112	0.761254072	0.949015856	1.185089588	1.356798887	1.517317533	1.722083807	1.861899853
	Verify	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
eddsa_libnacl.py	A1	0.086454391	0.160718679	0.226122618	0.29756403	0.371603727	0.449618816	0.525471687	0.592051268	0.672624588	0.790100813
eddsa_pynacl.py	A2	0.077557087	0.145658255	0.21912694	0.290917873	0.357791662	0.434111357	0.496501923	0.571876526	0.639072657	0.712193727
eddsa_25519.py	A3	1.177778721	2.357604742	3.557043314	4.750913382	5.927227497	7.095568657	8.346542358	9.418957949	10.97768307	12.01746202
ecdsa_pyca.py	A4	0.478735924	0.951450109	1.502687216	2.009141922	2.536696434	3.080529928	3.623173714	4.030770063	4.436508417	5.182082415
ecdsa_fast.py	A5	1.888226509	3.866354227	5.895234823	7.814577818	9.675366163	11.60614085	13.50765395	15.35402393	17.24752975	18.99491715
Lamport.py	A6	9.27e-05	0.00021863	0.00032568	0.000403643	0.000490665	0.000626802	0.000705242	0.000831604	0.000956059	0.001034021
rsa_pyca.py	A7	0.103438377	0.202701807	0.305972815	0.40743041	0.497382879	0.607035875	0.72511816	0.797843933	0.918581009	0.992576361
dsa_pyca.py	A8	0.132561684	0.261276245	0.388671637	0.525073528	0.657772064	0.797947168	0.929007769	1.055784225	1.183903456	1.304363251

Figure 23: Cross Compare MacBookPro Runtimes Table

REFERENCES

[1] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Trans. Inf. Theor.*, 22(6):644–654, September 2006.

[2] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, February 1978.

[3] Cameron F. Kerry, Acting Secretary, and Charles Romine Director. Fips pub 186-4 federal information processing standards publication digital signature standard (dss), 2013.

[4] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *Journal of Cryptographic Engineering*, 2(2):77–89, Sep 2012.

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

[6] Leslie Lamport. Constructing digital signatures from a one way function. Technical report, Microsoft, October 1979.

[7] K. Moriarty, B. Kaliski, J. Jonsson, and A. Rusch. PKCS#1: RSA Cryptography Specifications Version 2.2, 2016. Online; accessed 2018-10-15; https://tools.ietf.org/html/rfc8017.

[8] Michael Case. A beginnerâĂŹs guide to the general number field sieve. *Oregon State University, ECE575 Data Security and Cryptography Project*, 2003.

[9] Dan Boneh et al. Twenty years of attacks on the rsa cryptosystem. *Notices of the AMS*, 46(2):203–213, 1999.

[10] Don Coppersmith. Small solutions to polynomial equations, and low exponent rsa vulnerabilities. *Journal of Cryptology*, 10(4):233–260, 1997.

[11] Coron et al. New attacks on pkcs# 1 v1. 5 encryption. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 369–381. Springer, 2000.

[12] Dan Boneh, Antoine Joux, and Phong Q Nguyen. Why textbook elgamal and rsa encryption are insecure. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 30–43. Springer, 2000.

[13] Daniel Bleichenbacher. Chosen ciphertext attacks against protocols based on the rsa encryption standard pkcs# 1. In *Annual International Cryptology Conference*, pages 1–12. Springer, 1998.

[14] Mihir Bellare and Phillip Rogaway. Optimal asymmetric encryption. In *Workshop on the Theory and Application of of Cryptographic Techniques*, pages 92–111. Springer, 1994.

[15] Mihir Bellare and Phillip Rogaway. Probabilistic signature scheme, July 24 2001. US Patent 6,266,771.

[16] Johannes Böck. Rsa-pss–provable secure rsa signatures and their implementation, 2011.

[17] Python Cryptographic Authority. Pyca cryptography. Online; accessed 2018-10-15; https://github.com/pyca/cryptography.

[18] Elaine Barker. Nist sp 800-57 part 1 rev.4: Recommendation for key management, part 1: General, 2016.

[19] Don Johnson, Alfred Menezes, and Scott Vanstone. The elliptic curve digital signature algorithm (ecdsa). *International journal of information security*, 1(1):36–63, 2001.

[20] Ali Al Imem. Comparison and evaluation of digital signature schemes employed in ndn network. *arXiv preprint arXiv:1508.00184*, 2015.

[21] Sabyasachi Karati, Abhijit Das, Dipanwita Roychowdhury, Bhargav Bellur, Debojyoti Bhattacharya, and Aravind Iyer. New algorithms for batch verification of standard ecdsa signatures. *Journal of Cryptographic Engineering*, 4(4):237–258, 2014.

[22] Arjen K. Lenstra, Thorsten Kleinjung, and Emmanuel Thomé. Universal Security; From bits and mips to pools, lakes - and beyond. In Marc Fischlin and Stefan Katzenbeisser, editors, *Number Theory and Cryptography*, volume 8260 of *Lecture Notes in Computer Science*, pages 121–124, Darmstadt, Germany, November 2013. Springer. Humoristic.

[23] Brian Warner. Pure-python ecdsa. Online; accessed 2018-10-15; https://github.com/warner/python-ecdsa.

[24] AntonKueltz. Python library for fast elliptic curve crypto. Online; accessed 2018-10-15; https://github.com/AntonKueltz/fastecdsa.

[25] Python Cryptographic Authority. Pynacl: Python binding to the libsodium library. Online; accessed 2018-10-15; https://github.com/pyca/pynacl.

[26] Brian Warner. Python bindings to the ed25519 digital signature system. Online; accessed 2018-10-15; https://github.com/warner/python-ed25519.

[27] SaltStack. libnacl: Python ctypes wrapper for libsodium. Online; accessed 2018-10-15; https://github.com/saltstack/libnacl.

[28] Y. Romailler and S. Pelissier. Practical fault attack against the ed25519 and eddsa signature schemes. In *2017 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 17–24, Sept 2017.

[29] G. M. Abdullah, Q. Mehmood, and C. B. A. Khan. Adoption of lamport signature scheme to implement digital signatures in iot. In *2018 International Conference on Computing, Mathematics and Engineering Technologies (iCoMET)*, pages 1–4, March 2018.

[30] S. Alboaie, D. Cosovan, L. Chiorean, and M. F. Vaida. Lamport n-time signature scheme. In *2018 IEEE International Conference on Automation, Quality and Testing, Robotics (AQTR)*, pages 1–6, May 2018.

[31] N. Benchasattabuse, P. Chongstitvatana, and C. Apomtewan. Quantum rough counting and its application to grover's search algorithm. In *2018 3rd International Conference on Computer and Communication Systems (ICCCS)*, pages 21–24, April 2018.

[32] HyunYong Lee and JongWon Kim. The enhanced lamport signature for secure service overlay networks. In *2009 International Conference on Information Networking*, pages 1–5, Jan 2009.

[33] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, April 2010.

[34] IANIX. Things that use Ed25519, 2018. Online; accessed 2018-10-15; https://ianix.com/pub/ed25519-deployment.html.