Development and Examination of in-browser GPU Accelerated Cryptography

Dajne Win

4th February 2017

School of Engineering, Computer and Mathematical Sciences

A thesis submitted to Auckland University of Technology in partial fulfillment of the requirements for the degree of Master of Information Security and Digital Forensics.

Abstract

Many of us use encryption frequently whether realising it or not; it is the active yet often invisible element keeping our information and data safe. Despite this, many of us underestimate the value of encryption in our daily lives. Schneier (2016) explains encryption is instrumental in protecting identities, governments, lawmakers, law enforcement, military, critical infrastructure, communications networks, power grids, transportation, and everything else we rely on in society. "As we move to the Internet of Things ... encryption will become even more critical to our personal and national security" (Schneier, 2016). Understanding the ever-changing threat landscape, predicting potential trends, and current security issues are the core roles of the security researcher. The process of establishing frameworks helps mitigate risks of the critical reliance on encryption.

One of the challenges encryption faces is it is inherently computationally intensive and therefore slow. Due to mobile devices' focus on performance over security, it is vital to find methods to accelerate modern encryption algorithms to preserve information security in the future. Previous research has successfully investigated the use of hardware to accelerate encryption algorithms. Algorithm accelerators have used Graphics Processing Units (GPU) for many years and have proven these to be effective for parallel workloads. An advantage is that GPUs are already part of most computer systems, making them a fertile area for research into hardware performance. However, previous research has been limited to system specific compiled code.

This research explores the ability to perform acceleration on any modern browser through a scripted programming language. The selection of NTRUEncrypt for this experiment was due to its suitability towards acceleration, protection against quantum computers and as an alternative to RSA or Elliptic Curve Cryptography (ECC). A pure JavaScript and GPU accelerated version of NTRUEncrypt were developed. The Three.js library was selected to utilise the latest version of WebGL in modern browsers and reduce development time. OpenGL ES 1.0 compatible shaders then replaced the addition and convolution operations of NTRUEncrypt, utilising the system GPU for processing.

Performance comparison of encryption and decryption between NTRUEncrypt.js and NTRUEncrypt-GPU.js was then performed. Polynomial convolution at the highest security settings was 1.6

times faster on the GPU compared to the Central Processing Unit (CPU). However, results from this experiment show NTRUEncrypt-GPU.js failed to accelerate the NTRUEncrypt cryptographic algorithm. Furthermore, comparisons within this research showed JavaScript was up to 80 times slower than C, C++, and Java.

Future research into accelerated cryptography would provide further knowledge, understanding and open new opportunities for improvement to information security. While NTRUEncrypt-GPU.js failed to accelerated NTRUEncrypt using currently available standards, preliminary testing using Compute Shaders proved successful and warrents further investigation.

Contents

List of Figures

List of Tables

List of Abbreviations

3DES	Triple Data Encryption Standard
AES	Advanced Encryption Standard
CPU	Central Processing Unit
CSPRNG	Cryptographic Pseudo-Random Number Generator
CUDA	Compute Unified Device Architecture
DES	Data Encryption Standard
DH	Diffie-Hellman
Dual EC DRBG	Dual Elliptic Curve Deterministic Random Bit Generator
ECC	Elliptic Curve Cryptography
ECDH	Elliptic Curve Diffie-Hellman
FIPS	Federal Information Processing Standard
GCD	Greatest Common Divisor
GPGPU	General Purpose Graphics Processing Unit
GPU	Graphics Processing Unit
NIST	National Institute of Standards and Technology
OpenCL	Open Computing Language
OpenGL	Open Graphics Library
PFS	Perfect Forward Secrecy
PRNG	Pseudo-Random Number Generator
SIMD	Single Instruction, Multiple Data
SISD	Single Instruction, Single Data
SM	Streaming Multiprocessors
SVP	Shortest Vector Problem
TRNG	True Random Number Generator
XOR	Exclusive-OR

Attestation of Authorship

I hereby declare that this submission is my own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another person (except where explicitly defined in the acknowledgements), nor material which to a substantial extent has been submitted for the award of any other degree or diploma of a university or other institution of higher learning.

Acknowledgements

Firstly, I would like to thank my primary thesis supervisor Dr Alastair Nisbet. Providing direction, motivation and constructive feedback throughout the entire process. I would highly recommend him for anyone seeking a supervisor who truly cares about the student, and pushes you to do as well academically as he knows you are capable of.

Secondly, my secondary thesis supervisor Dr Seth Hall, who without I probably would not have made it as far with development. This was a technical thesis, without Seth's expertise in the area of GPU computing, architecture and development it would not have been possible to undertake this subject.

My fiancé Chloe Ellis, she provided the support throughout my entire academic study and without her none of this would have been possible!

Numerous other staff at AUT who provided support for my crazy hardware requirements, sourcing or algorithm questions. It feels both amazing and crazy when you have questions that are able to stump those who have considerably more experience and knowledge than yourself.

Lastly but certainly not least my friends, peers, and family who provided support and motivation throughout, it takes a community to put this together.

1 Introduction

Rivest (1990) describes cryptography as secure communication between parties in the presence of adversaries. It is the process of transforming data in such a way that it is incomprehensible to anyone other than the intended recipients. Cryptography is essential for information security as it provides confidentiality, integrity, authentication, and non-repudiation. Maintaining information security requires continuous academic research, especially due to the public nature of networks such as the Internet. Cryptography algorithms consist of three major operations - key generation, encryption, and decryption. A key is required to perform the encryption, which transforms plaintext into ciphertext. A key is then required to perform decryption operations to obtain the plaintext. All cryptographic algorithms can be constructed using either of two different techniques: substitution or transposition. This initial examination of the foundation of cryptography in section 2.1.1 provides a basic understanding of these two techniques. The following sections then examine in chronological order the development to modern day cryptographic algorithms. Cryptographic history is of particular importance to explore as many older algorithms are still widely used today. Comparatively, asymmetric cryptography, such as Diffie-Hellman, has only been available since 1978. Further study is required to improve cryptographic knowledge and information security.

Moore's Law (1965) is a simple observation that the number of components per silicon chip would double every two years. A doubling in the number of components available for processing leads to increased efficiency and higher processing capabilities. Moore's Law (1965) and the availability of quantum computers will make the currently used algorithms obsolete within a few years. Thus, an investigation into alternative algorithms is required to maintain information security and improve efficiency in the future.

Recently, there has been a movement toward mobile devices that favour power efficiency over performance. These devices have physical limitations of size, heat, and power consumption, meaning they are not as powerful as current desktop or server hardware. Investigations into the more efficient use of processing power are essential to sustain mobile device functionality. A large proportion of processing power on mobile devices is communication with other devices. Implementation of secure communication using industry best practices requires frequent use of cryptography. An issue plagued by previous research is that

most algorithm acceleration requires either specialised hardware or software compiled specifically for that particular hardware system. Implementing an algorithm in a device agnostic language, such as JavaScript, means eliminating the software compilation problem and alleviating the hardware requirement problem. Any device with a modern browser can then utilise the algorithm to communicate securely. However, this does not completely solve the problem associated with Moore's Law. As previously stated, one problem with algorithm acceleration is the reliance on specialised hardware. Many devices that can run a modern browser will additionally have a modern GPU. Many researchers have investigated accelerating various algorithms, including cryptography, and most demonstrate some success. Previous research affirms it is possible to accelerate cryptography using a GPU. The current software and hardware technology ready for implementation leads to a confirmation of the problem solution. Therefore, this research seeks to investigate, confirm, and develop the in-browser acceleration of a cryptographic algorithm using a device's GPU.

2 Literature Review

2.1 Cryptography

In the following sections this research investigates and provides the foundation for cryptography. In section 2.1.1 reasoning for why cryptography exists, where it has come from, and the building blocks for modern cryptography are briefly discussed. Section 2.1.2 introduces cryptography's fit within information security. A brief discussion in section 2.1.3 introduces secure random number generation and its importance to cryptography. Next, in section 2.1.4 this paper moves into a discussion of modern symmetric cryptographic algorithms, why they were selected, replaced and how they operate. Finally in section 2.1.5 is an in depth investigation into available asymmetric cryptographic algorithms.

2.1.1 Foundation

Damico (2009) suggests cryptography has been utilised for communication for thousands of years, as early as 1900 BC. Secure communication is vital to survival and is considered to have been born out of the military and used for communicating troop movements and intelligence. Attempts were made to intercept and manipulate the information the enemy knows while maintaining the security of your own communications. One example that highlights the issues of poor security standards is discussed by Weadon (2000). Weadon mentions that cryptanalysis helped the United States in the lead-up to the World War II Battle of Midway. United States cryptanalysts had partially broken the Japanese JN-25 Naval code. Many communication interceptions allowed the cryptanalysts to infer that an operation or objective "AF" would soon be underway by the Japanese navy. The US cryptanalysts concluded the code "AF" probably referred to Midway. The US confirmed this by sending out a false unencrypted broadcast that the Water purification system on Midway had broken, they then intercepted Japanese communications that mentioned "AF" had a water problem. As a result, the intelligence gathered from this communication flaw allowed the US time to set up a pre-emptive strike against the Japanese navy, resulting in a victory that is considered the turning point in the Pacific War.

To discuss modern cryptography it is important to understand the historical basis that it derives from. All cryptographic systems are

based on either transposition, changing the order of characters; or substitution, replacing plaintext characters with other characters or symbols. These two techniques provide the basis for all currently known cryptography, classical to modern.

A straightforward example of substitution is the Caesar cipher. To perform a Caesar cipher encryption, shift the alphabet a number of places left or right and substitute the plaintext letters for the now ciphertext letters, then reverse this process to obtain the plaintext message as illustrated in figure 2.1.

Plaintext Alphabet	ABCDEFGHIJKLMNOPQRSTUVWXYZ
Ciphertext Alphabet	XYZABCDEFGHIJKLMNOPQRSTUVW
Key	Shift right 3

Plaintext Message	HELLO WORLD
Encrypted Message	EBIIL TLOIA

Figure 2.1: Example of Caesar Cipher

An example that clearly illustrates transposition is the Route cipher. As illustrated in figure 2.2 simply place all plaintext characters in a grid, then read off in a given pattern.

Plaintext Message	UNDER ATTACK
Route	UEAA NRTC D TK
Ciphertext	UEAANRTCD TK

Figure 2.2: Example of Route Cipher

The total number of possible keys for a given encryption algorithm is known as the key space. The complication with many early examples of cryptographic ciphers is that they do not have a large enough key space. For example, the Caesar cipher has a key space of only 25 and while this may take time for a human to go through every single combination, a modern desktop computer would take a matter of milliseconds to brute force this encryption algorithm. Classical ciphers, therefore, offer poor security for modern communications systems. However, they have provided a fundamental basis without which modern cryptographic algorithms could not have developed.

2.1.2 Information Security Aspects

Authentication, availability, confidentiality, integrity, and non-repudiation create the five foundational aspects of information security. Cryptography provides four of these five aspects. The keys or certificates used in asymmetric encryption provide authentication, proving the bearer is who they say they are. Cryptography additionally provides confidentiality by allowing parties to encrypt data, therefore limiting access to only entities in possession of the decryption key. A single change to the ciphertext would result in failure to decrypt the message, making any tampering plainly evident and assuring integrity. Message signing can provide an additional layer of integrity. In adhering to the principles of authentication, confidentiality and integrity, cryptography also achieves non-repudiation. Non-repudiation is achieved as only the holder of the key is able to participate in successful encryption or decryption. Cryptography, when implemented correctly, an essential part of an effective information security strategy.

2.1.3 Random Number Generation Weaknesses

Before examining cryptographic algorithms directly, an important, although often overlooked, part of the implementation of a cryptographic algorithm is secure random number generation. It is often quicker to exploit a weakness in the random number generator of an implementation of a cryptographic algorithm, due to a large key space or algorithm complexity. There are three types of random number generators: Pseudo-Random Number Generator (PRNG), Cryptographic Pseudo-Random Number Generator (CSPRNG) and True Random Number Generator (TRNG). PRNG's are pseudo-random meaning they eventually have some pattern of predictability, and this makes them unsuitable for cryptographic algorithms. CSPRNGs are PRNGs that meet certain requirements and require a larger amount of entropy, therefore requiring a larger amount of processing power. This increased processing power means that a developer's implementation has to decide between speed and security, providing an opportunity for an attacker to exploit. One-way functions, such as hashing, create another level of entropy and are comparable to one-time pads, which are considered unbreakable. TRNG in computing require generating entropy outside of the computer system. A hardware device affected by physical phenomena, such as nuclear decay, thermal noise or atmospheric noise that then converts this into a random series

of bits is used to create true entropy. Specialised hardware and the speed of entropy generation are two issues with this process. Many hardware TRNGs are implemented by taking only a limited amount of entropy, then passing this entropy through a hashing algorithm to provide quick and secure CSPRNG.

2.1.4 Symmetric Encryption

Symmetric encryption algorithms use the same key for both encryption and decryption. Therefore, during communication both parties must have the same key or shared secret.

Data Encryption Standard (DES) Published in 1977 as a Federal Information Processing Standard (FIPS), the Data Encryption Standard (DES) was the first symmetric encryption algorithm standardised for public use. DES provided security for unclassified electronic government data until 1999. Only legacy systems continue to use DES afterwards, while Triple DES was designed to replace it. DES is a block cipher and works with 64-bit blocks of plaintext at a time. The key must be 64-bits, 56-bits are used for the key and 8-bits are used for parity. DES performs 16 rounds on the plaintext input and an initial and final permutation. DES utilises a Feistel function for each round, this splits the input into two 32-bit halves, left and right. The left half is exclusive-OR (XOR) with the output of the Feistel function of the right half, before the next round where the halves are swapped. The Feistel function takes an input of 32-bits, performs expansion to 48-bits, and then an XOR of the 48-bits with a round subkey. The S-boxes perform substitution which reduces the output back to 32-bits; finally, the output of the S-boxes is permutated. Creating the input key requires a permutation, which is a simple bit rotation incremented per round, this calculates the subkey for each round. Due to its operations, DES requires little memory, and can, therefore, be cheaply implemented in hardware. Because DES has existed for a comparatively long time there are numerous libraries available for any architecture or programming language. Compared to modern cryptography alternatives it is considered slow. Since 2008 DES research indicates it is vulnerable to brute force attacks as the key space is only 2^{56}. While an implementation of 3DES (DES performed three times) is used to alleviate the security concerns of DES, this is extremely slow and thus an alternative was sought.

Advanced Encryption Standard (AES) The Advanced Encryption Standard (AES), originally known as Rijndael, was first published in 1998 by Daemen & Rijmen. The National Institute of Standards and Technology (NIST) held a competition in 2001 and Rijndael was selected as the replacement for DES. AES has an increased keyspace compared to DES, which alleviates the problem of brute force attacks. The design of AES requires fewer rounds, only table lookups, and XOR operations, which makes AES less computationally intensive than DES. The number of rounds AES performs is dependent on the key size. AES supports three key sizes: 128-bit, 192-bit, and 256-bit which use 10, 12, and 14 rounds respectively. Initialisation derives the round keys from the input cipher key; this is done using the Rijndael key schedule.

The schedule begins with a simple 8-bit rotation on a 32-bit word. Next is an rcon operation in the exponentiated Galois field of 2, followed by the S-Box operation. The rcon operation performs exponentiation of an input value within Rijndael's finite field, similar to a substitution. The output of the S-Box operation then produces an expanded key for each round of AES. The first round only performs an AddRoundKey operation, which is simply a bitwise XOR with the input state and round key. Each round then performs SubBytes, ShiftRows, MixColumns, and AddRoundKey operations until the final round which excludes the MixColumns operation.

SubBytes performs a substitution of the state bytes with a lookup table. ShiftRows performs transposition of the state where the last three rows are shifted a set number of steps. Finally, MixColumns is another transposition operation that shifts each column a fixed number of steps.

AES has a considerably larger key space than DES, making brute force attacks infeasible. Some hardware implementations, such as Intel AES-NI as initially defined by Gueron (n.d.), allow for fast and efficient use of AES. Harrison & Waldron (2007) examined the acceleration of AES on a Graphics Processing Unit (GPU). Their findings demonstrated a GPU was only useful for bulk AES encryption and decryption; they achieved a maximum of 870.8Mbits/s (108.85 MB/s). Furthermore Yang & Goodman (2007) investigated accelerating both AES and DES symmetric key encryption algorithms on two different GPUs. Their experiment demonstrated a significantly improved acceleration, 3.5 Gbits/s (437.5 MB/s) on an HD 2900 XT GPU, compared to 1.6 Gbits/s (200 MB/s) on a high-performance CPU. However

Chesebrough & Conlon (2012) demonstrated an implementation that took advantage of Intel AES-NI and achieved 579.9MB/s which is more than five times faster than Harrison et al.'s implementation and almost 1.5 times faster than Yang & Goodman's. One issue that both Harrison & Waldron and Yang & Goodman encountered was the limiting factor of memory bandwidth. Yang & Goodman (2007) discuss how comparing AES on two different GPUs, one with more processing units than the other, did not scale as expected. Instead, both cards had an equal number of memory fetch units, and they found this to be the limiting factor.

Even with algorithms that are considered secure, such as AES, communicating the key for use in symmetric encryption between two parties is difficult, given the only line of communication may be public. If the symmetric key is compromised, then communication from all parties is compromised. As the number of authorised parties increases this adds additional risk. Asymmetric encryption is used to overcome these symmetric encryption problems and communicate securely between parties.

2.1.5 Asymmetric Encryption

Asymmetric cryptography, also known as public-key cryptography, is a newer concept than symmetric cryptographic systems. Many asymmetric cryptography systems exist and are based on the various strengths of underlying mathematical properties. The three most widely used asymmetric cryptographic systems are Diffie-Hellman (Merkle, 1978), RSA (Rivest, Shamir & Adleman, 1978) and Elliptic Curve Cryptography (ECC) (Miller, 1986; Koblitz, 1987).

Trapdoor functions provide the basis for most asymmetric cryptography. Trapdoor functions are mathematical functions that are easy to compute in one direction yet difficult to compute in the opposite direction without knowledge of the private key. The important distinction between a trapdoor and one-way function is that the trapdoor can be reversed, given the private data. For example, given the number 3397 which is the product of two prime numbers, it is challenging to obtain the original two prime numbers (43 and 79) without calculating all possible combinations of prime numbers. It is, however, comparatively trivial to multiply 43 and 79. The product of two primes is an example of the prime factorization problem, which is utilised by RSA. The discrete logarithm problem, used by ECC,

and the shortest vector problem, used by NTRUEncrypt, are both considered trapdoor functions.

Along with trapdoor functions many cryptographic algorithms, both symmetric and asymmetric, rely on prime numbers. A prime number is a natural number greater than one which has no positive divisors other than 1 and itself. This property makes prime numbers useful for many cryptographic algorithms. Testing if a number is prime is computationally intensive, as the only sure way of accurately checking is to perform a full

$$(n/2)\ mod\, 2 == 0$$

test (where n is the number to test). This results in $O(n)$ computations which, for large prime numbers, becomes infeasible. For some cryptographic algorithms, coprime integers are necessary; this is two integers that have their greatest common divisor equal to 1.

Another important mathematical basis is Fermat's little theorem, which is used by all three asymmetric cryptographic systems: Diffie-Hellman, ECC and RSA. Liskov (2011) defines Fermat's little theorem as stating that if p is a prime number and a is any number not divisible by p, then

$$a^{p-1} \equiv 1\ mod\, p$$

This theorem is important because it allows us to perform fewer calculations to determine if p is probably prime. While considerably faster than performing a full test, this theorem fails for Carmichael numbers, which pass Fermat's little theorem for primality; however, they are not truly prime. In most cases, Fermat's little theorem can quickly establish if a number is prime and then a more accurate method can be used to determine if p is truly prime if that level of accuracy is required.

Diffie-Hellman Having been first published in 1978 by Diffie, Hellman, and Merkle (1978), Diffie-Hellman is the oldest asymmetric cryptographic algorithm. Given that two parties want to communicate securely over an insecure network, Diffie-Hellman can be used to generate a shared secret. For example, both Alice and Bob select two values g and p, which can be publicly known, where p is a prime number, and g is a primitive root modulo of p. Alice and Bob then each select a secret integer a and b respectively. Alice calculates and sends Bob A, where $A = g^a\ mod\, p$. Bob performs a calculation and sends Alice B, where $B = g^b\ mod\, p$. Alice then computes s, where $s = B^a\ mod\, p$ while

Bob computes s, where $s = A^b \, mod \, p$. Further secret communication can occur as now both Alice and Bob have the same shared secret s that was not publicly transmitted. An illustration of a Diffie-Hellman key exchange is demonstrated in figure 2.3.

ALICE	Publicly Exchanged	BOB
$g = 10$	$g = 10 \quad p = 541$	$g = 10$
$p = 541$		$p = 541$
$a = 7$		$b = 13$
$A = g^a \, mod \, p$		$B = g^b \, mod \, p$
$A = 10^7 \, mod \, 541$		$B = 10^{13} \, mod \, 541$
$A = 156$	$A = 156 \quad B = 486$	$B = 486$
$s = B^a \, mod \, p$		$s = A^b \, mod \, p$
$s = 486^7 \, mod \, 541$		$s = 156^{13} \, mod \, 541$
$s = 333$		$s = 333$

Figure 2.3: Example of Diffie-Hellman

Diffie-Hellman on its own only provides a key exchange between parties. Therefore, the parties must then decide on another encryption algorithm to utilise the shared secret key. Due to this limitation, Diffie-Hellman does not provide authentication and is thus susceptible to a man-in-the-middle attack. However, Diffie-Hellman can be implemented to provide Perfect Forward Secrecy (PFS) by generating ephemeral keys for one-time use or single session use. Thus, if an attacker compromises one communication session, other sessions performed between the same parties would not be compromised.

RSA The RSA public-key cryptosystem published by Rivest, Shamir and Adleman in 1978 is similar to Diffie-Hellman. RSA relies upon the difficulty of factoring large prime numbers for security. RSA has an advantage over Diffie-Hellman, that is the ability to provide authentication, through message signing. Key generation for RSA is computationally intensive compared to its encryption and decryption operations. RSA consists of four essential functions: key generation, encryption, decryption and signing. Key generation is the most computationally intensive function of RSA as the selection of two large prime numbers can be $O(n)$ for each in the worst case. RSA utilises modular exponentiation for encryption and decryption, specifically the functions $C = P^e mod \; n$ and $P = C^d mod \; n$. Modular exponentiation requires the base number be taken to the power of either e or d (dependent on encryption or decryption) within $mod \, n$. While this operation is trivial for a computer to perform on small numbers e.g.,

$3^3 = 27$. This operation becomes increasingly computationally intensive with large numbers. This shortcoming of RSA, led to the development of algorithms more efficient at performing modular exponentiation. As illustrated in figure 2.4 an example of repeated squaring allows quicker calculation as only the calculation of the square exponents less than the input exponent is required. A continuation of repeated squaring, the modulus is converted to base 2 and calculating each square of the base number in this modulus the rule $x^a + b = x^a \cdot x^b$ then applies. For example $3^{200}\ mod\, 50 = 3^{128} \cdot 3^{64} \cdot 3^8\ mod\, 50$ as 200 base 2 = 11001000 (or 128 + 64 + 8, which is where all the 1's are in base 2).

3^{200} mod 50	
3^1 mod 50 = 3 mod 50	3^{16} mod 50 = 21 mod 50
3^2 mod 50 = 9 mod 50	3^{32} mod 50 = 41 mod 50
3^4 mod 50 = 31 mod 50	3^{64} mod 50 = 31 mod 50
3^8 mod 50 = 11 mod 50	3^{128} mod 50 = 11 mod 50

$200_2 = 11001000$
$200 = 128 + 64 + 8$
3^{200}mod 50 = $3^{128\ +\ 64\ +\ 8}$
$3^{200} mod\ 50 = 3^{128} \cdot 3^{64} \cdot 3^8 mod\ 50$
$3^{200} mod\ 50 = 11 \cdot 31 \cdot 11\, mod\ 50$
3^{200}mod 50 = 3751 mod 50
3^{200}mod 50 = 1 mod 50

Figure 2.4: Repeated Squaring Example

In the case of RSA, the selection of prime numbers must be random and large. The Euclidian Algorithm is an efficient method for computing the greatest common divisor (GCD) of two numbers which is the largest number that divides both without leaving a remainder, example as illustrated in figure 2.5.

$a = 200\ and\ b = 45$
$GCD(a, b)$
$a\ mod\, b = r$
$While\, r \neq 0:$
$200\ mod\, 45 = 20$
$a = 45 \quad b = 20$
$45\ mod\, 20 = 5$
$a = 20 \quad b = 5$
$20\ mod\, 5 = 0$
$\therefore\ the\ GCD(200, 45) = 5$

Figure 2.5: Example of Euclidian Algorithm

Key generation for RSA commences with selecting two large prime numbers p and q. The product n of these two numbers is computed

where $n = p \cdot q$. $\phi(n)$ is then calculated where

$$\phi(n) = (p-1) \cdot (q-1)$$

A small odd number e is selected (normally 65537) where e is relatively prime to $\phi(n)$ alternatively, where the greatest common divisor of $\phi(n)$ and e is 1. The private integer d is computed where

$$d \cdot e \; mod \, (\phi(n)) = 1$$

The public key consists of e and n and can then be given out to any party that wishes to communicate. The private key, made up of d and n, must be kept secret. Illustrated in figure 2.6 is an example of RSA key generation.

ALICE
Key Bits = 16
p = 79
q = 29
$n = p \cdot q$
$n = 79 \cdot 29$
n = 2291
$\Phi(n) = (p-1) \cdot (q-1)$
$\Phi(n) = (79-1) \cdot (29-1)$
$\Phi(n) = 2184$
e = 5
$d : (d \cdot e) mod \Phi(n) = 1$
Compute d = (? . 5) mod 2184 = 1
d = 437
Public Key (e, n) = (5, 2291)
Private Key (d, n) = (437, 2291)

Figure 2.6: Example of RSA Key Generation

RSA encryption is simple. Firstly, Bob must obtain Alice's public key. Bob then selects an integer P to encrypt. He then computes the ciphertext C using Alice's public key where

$$C = P^e mod \; n$$

Bob can then send the encrypted integer C to Alice, and only Alice will be able to decrypt with her private key. An example illustrated in figure 2.7 shows this process.

Alice	Publicly Exchanged	Bob
Public Key (e, n) = (5, 2291)	→	Alice's Public Key (e, n) = (5, 2291)
		$P = 90$
		$C = P^{\mathrm{e}} \bmod n$
		$C = 90^5 \bmod 2291$
$C = 997$	$C = 997$	$C = 997$

Figure 2.7: RSA Encryption Example

RSA decryption is performed similarly to encryption. As illustrated in figure 2.8 once Alice has received the encrypted integer C from Bob, she computes plaintext P where $P = C^d mod\ n$, d and n are from Alice's Private Key.

Alice	Publicly Exchanged	Bob
$C = 997$	$C = 997$	$C = 997$
Private Key (d, n) = (437, 2291)		$P = 90$
$P = C^d \bmod n$		
$P = 997^{437} \bmod 2291$		
$P = 90$		

Figure 2.8: RSA Decryption Example

The advantage of RSA over Diffie-Hellman is that the sender's private key could additionally sign an encrypted message. In this case, Bob could sign the message and in this way Alice knows that only Bob (or the computer in possession of Bob's private key) could have sent the message. In this way RSA provides authentication and non-repudiation, and a correct implementation makes man-in-the-middle attacks impossible. While both RSA and Diffie-Hellman are considered reliable and secure, they are both computationally intensive. Using anything more than 4096-bit RSA on a mobile device is infeasible, and thus, alternatives are sought.

Elliptic Curve Cryptography (ECC) Published separately by both Miller (1986) and Koblitz (1987), Elliptic Curve Cryptography (ECC) did not enter into wide use until included in OpenSSL version 0.9.8 in 2005. The reasoning behind the slow adoption of new encryption algorithms is to allow academic researchers and cryptanalysts time to attempt to find problems with the algorithm. At the time of ECC's publication, RSA had already become a widely implemented standard, and no immediate threat to security required replacement. Comparing ECC to RSA, ECC can provide an equivalent level of

security at a significantly smaller key size. ECC is reliant on the discrete logarithm problem for its security. Based on special curves that satisfy the equation $y^2 = x^3 + ax + b$. Scalar point multiplication is utilised extensively in ECC; this is the method for multiplying an integer value with a point on a curve. The simplest method which Hankerson, Menezes & Vanstone (2006) describes is double-and-add. Given the random number $r = 2$, and the point $G = \left(\frac{9}{5}\right)$ performing double-and-add (in $mod\,23$) would give us the results $G2 = \left(\frac{18}{10}\right)$ as $G2 = \left(\frac{2*9\,(mod\,23)}{2*5\,(mod\,23)}\right)$.

Selection of an Elliptic Curve determines parameters used for key exchange and signing. p is a prime number that the curve operates in as all point calculations performed modulo p, usually noted as F_p. a and b are integers that define the curve. G is the generator or base point, given in point form this is a point on the curve that provides the basis for further calculations. n the number of different distinct points on the curve. h the ratio of curve points, calculated using $\frac{p}{n}$. Alice selects a curve then selects a random number d_A, where $0 < d_a < n$. Alice then computes Q_A by performing point scalar multiplication where $Q_A = d_A \cdot G$. Alice's public key is Q_A which is a distinct point on the curve F_{23}. Alice's private key is d_A which is an integer. An example in figure 2.9 of ECC key generation illustrates this process.

Curve F_{23} Parameters
$y^2 = x^3 + x$
$a = 1$
$b = 0$
$p = 23$
$G = (9, 5)$
$n = 23$
Alice
$Curve = F_{23}$
$d_A = 6$
$Q_A = d_A \cdot G$
$Q_A = 6 \cdot (9, 5)$
$G : (9, 5)$
$G2 : (8, 7)$
$G3 : (2, 19)$
$G4 : (12, 22)$
$G5 : (3, 17)$
$G6 : (18, 10)$
$10^2 \ mod \, 23 = 18^3 + 18 \ mod \, 23$
$Q_A = (18, 10)$
Public Key $Q_A = (18, 10)$
Private Key $d_A = 6$

Figure 2.9: Example of ECC Key Generation

Similar to Diffie-Hellman, ECC on its own is unable to provide encryption and decryption. It can, however, perform a key exchange, enabling communicating parties to generate a secret key for secure symmetric encryption. Elliptic Curve Diffie-Hellman (ECDH) begins with obtaining the public key of the recipient. If Bob is communicating with Alice, he must first select a random integer r where $0 < r < n$. Bob then computes R by performing point scalar multiplication where $R = r \cdot G$. Bob sends Alice R, and the shared secret key s is computed where $s = r \cdot Q_A$. Alice receives the computed point R from Bob. Alice can compute the shared secret key s where $s = d_A \cdot R$. Finally, an agreed symmetric algorithm can use the shared secret s that Alice and Bob now have as illustrated in figure 2.10 of an example of Elliptic Curve Diffie-Hellman (ECDH) key exchange.

Alice	Publicly Exchanged	Bob
Public Key $Q_A = (18, 10)$	Alice's Public Key $Q_A = (18, 10)$	Alice's Public Key $Q_A = (18, 10)$
		$r = 7$
		$R = r \cdot G$
		$R = 7 \cdot (9, 5)$
$R = (13, 20)$	$R = (13, 20)$	$R = (13, 20)$
$s = d_A \cdot R$		$s = r \cdot Q_A$
$s = 6 \cdot (13, 20)$		$s = 7 \cdot (18, 10)$
$s = (3, 17)$		$s = (3, 17)$

Figure 2.10: Example of ECDH

Although ECC is considerably faster than RSA and Diffie-Hellman, both parties must agree on using the same elliptic curve. Patents on curves have further prevented many from using ECC, and this has significantly impacted the confidence in systems that use ECC. There are many other confidence issues surrounding ECC. Some security experts have been sceptical about suggested curves from privately funded cryptographic institutions and governments. In 2007 it was proposed by Shumow & Ferguson that the NSA had deliberately placed a backdoor in the Dual Elliptic Curve Deterministic Random Bit Generator (Dual EC DRBG). The NSA initially denied they had intentionally created a backdoor. However, documents later leaked by Edward Snowden demonstrated the NSA had pushed to become the sole editor of the Dual EC DRBG standard. These issues have slowed the uptake and confidence in ECC.

Many have sought replacements for Diffie-Hellman, RSA, and ECC. Both Diffie-Hellman and RSA are computationally intensive and as processors become more efficient, a linear increase in bit size to increase security in turn results in a larger increase in computational intensity. ECC has many political and confidence problems and, while less computationally intensive than RSA and Diffie-Hellman, does not provide the ability to encrypt or decrypt on its own. While these three options are secure for now, the underlying problem with all three algorithms is that they are not secure against quantum computers. Jaeger (2007) describes cryptography as an arms race between cryptographers and cryptanalysts. This war forces constant research as cryptographers search for secure algorithms, while cryptanalysts attempt to break these algorithms. Perlner & Cooper (2009) investigated several quantum resistant public key cryptography algorithms due to the current reliance on factorisation and discrete

logarithm problem, both of which will become trivial to break once quantum computers are available.

2.2 Quantum Computing

2.2.1 Introduction

Significant advances have recently been made in the development of quantum computing, with Veldhorst et al. (2014) producing a 2-qubit quantum computer on silicon. More recently Monz et al. (2015) have scaled the Shor algorithm on a 7-qubit system. Shor's algorithm (1999) allows a quantum computer to reduce the prime factorisation and discrete logarithm problems to polynomial time. The prime factorization and discrete logarithm problems are what Diffie-Hellman, RSA, and ECC rely on for security. Thus, a sufficiently large quantum computer running Shor's algorithm would render these algorithms ineffective. Therefore security research needs to seek emerging technology and provide solutions to prevent security breaches from future hardware developments. Bernstein (2009) briefly describes some alternative asymmetric algorithms such as hash-based Merkle signature scheme (Merkle, Charles et al., 1979), code-based McEliece scheme (McEliece, 1978), and lattice-based NTRU public-key-encryption (Hoffstein, Pipher & Silverman, 1998). NTRU consists of both a public-key-encryption scheme known as NTRUEncrypt and a signing scheme NTRUSign. The research and scrutiny that NTRUEncrypt has undergone, its publication as a standard and it's resistance to known quantum computing attacks makes it the most likely replacement in a post-quantum cryptography world.

2.2.2 NTRUEncrypt

NTRUEncrypt was first published in 1998 by Hoffstein et al. and described by Perlner & Cooper (2009) as the most practical lattice-based cryptography. The IEEE P1363 standard on public-key cryptography added NTRUEncrypt in 2008 and in 2011 NTRUEncrypt was added to the X9.98 standard, both provide further confidence and assurance in NTRUEncrypt. NTRUEncrypt's security relies on the shortest vector problem (SVP). Given the basis for a lattice, the shortest vector to a random non-lattice vector must be found. Where, RSA performs a selection of two large prime numbers, comparatively NTRUEncrypt has an input of three publicly known parameters

$(N,\ p,\ q)$ that affect the selection of private polynomial f and its computed inverse f_p. All operations are performed using small polynomial rings with the number of coefficients being parameter N, a power of two modulus q, and a small prime number p.

Key generation for NTRUEncrypt commences by selecting two random small polynomials (coefficients between $\{-1, 0, 1\}$) f and g. Polynomial selection allows computing the inverses f_p and f_q where $f \cdot f_p = 1(mod p)$ and $f \cdot f_q = 1\ (mod\ q)$ is performed using a modified Euclidean algorithm, f and f_p comprise the private key pair. Finally the public key polynomial h is computed where $h = p \cdot f_q \cdot g\ (mod\, q)$. An example of NTRUEncrypt key generation is illustrated in 2.11.

$N = 11,\ q = 32,\ p = 3$
f = -1 + x + x^2 - x^4 + x^6 + x^9 - x^{10}
g = -1 + x^2 + x^3 + x^5 - x^8 - x^{10}
f_p: *where* $f \cdot f_p = 1\ (mod\, p)$
f_p = 1 + 2x + $2x^3$ + $2x^4$ + x^5 + $2x^7$ + x^8 + $2x^9$
f_q: *where* $f \cdot f_q = 1\ (mod\, q)$
f_q = 5 + 9x + $6x^2$ + $16x^3$ + $4x^4$ + $15x^5$ + $16x^6$ + $22x^7$+ $20x^8$ + $18x^9$ + $30x^{10}$
$h = p \cdot f_q \cdot g\ (mod\, q)$
h = 8 + 25x + $22x^2$ + $20x^3$ + $12x^4$ + $24x^5$ + $15x^6$ + $19x^7$+ $12x^8$ + $19x^9$ + $16x^{10}$

Figure 2.11: Example of NTRUEncrypt Key Generation

As illustrated in figure 2.12 encryption is performed where: $e = h \cdot r + m\ (mod\, q)$. First, polynomial convolution with public key h and blinding value r is performed, then the message to be encrypted in binary is added to this output. Finally, each coefficient is calculated modulo q. Only the holder of the private key can now decrypt the encrypted polynomial e.

Parameters	$N = 11,\ q = 32,\ p = 3$
h	$31 + 15x + 2x^2 + 7x^3 + 5x^5 + 12x^6$ $+26x^7 + 6x^8 + 8x^9 + 16x^{10}$
r	$-1x + 1x^2$
$r \cdot h$	$-8 - 15x + 16x^2 + 13x^3 - 5x^4 + 7x^5$ $-5x^6 - 7x^7 - 14x^8 + 20x^9 - 2x^{10}$
m	$1 + 1x^2 + 1x^5 + 1x^6$
$r \cdot h + m$	$-7 - 15x + 17x^2 + 13x^3 - 5x^4 + 8x^5$ $-4x^6 - 7x^7 - 14x^8 + 20x^9 - 2x^{10}$
$e = h \cdot r + m\ (mod\, q)$	$25 + 17x + 17x^2 + 13x^3 + 27x^4 + 8x^5$ $+28x^6 + 25x^7 + 18x^8 + 20x^9 + 30x^{10}$

Figure 2.12: Example of NTRUEncrypt Encryption

Obtaining an encrypted polynomial e is the first operation required for decryption. Next, the polynomial convolution of private polynomial f and e within modulus q produces polynomial a. Polynomial a is then reduced within modulus p, producing polynomial b. Finally, the plaintext message c is calculated $c = f_p \cdot b(mod p)$ where f_p is the private polynomial f inverse modulo p. An illustrated example in figure 2.13 describes this process of decryption.

Parameters	$N = 11,\ q = 32,\ p = 3$
e	$25 + 17x + 17x^2 + 13x^3 + 27x^4 + 8x^5$ $+28x^6 + 25x^7 + 18x^8 + 20x^9 + 30x^{10}$
f	$1x^4 - 1x^5 - 1x^6 + 1x^7 + 1x^8 - 1x^9 + 1x^{10}$
f_p	$2 + 2x + 2x^2 + 2x^3 + 2x^4 + 2x^5 + 2x^7 + 1x^8 + 1x^9$
$a = f \cdot e\ (mod\, q)$	$-3 + 4x - 1x^2 - 2x^4 + 3x^5 - 3x^8 + 7x^9 - 1x^{10}$
$b = a\ (mod\, p)$	$0, 1, -1, 0, 1, 0, 0, 0, 0, 1, -1$
$c = f_p \cdot b\ (mod\, p)$	$1 + 1x^2 + 1x^5 + 1x^6$

Figure 2.13: Example of NTRUEncrypt Decryption

The NTRUOpenSourceProject provides an open source implementation of NTRUEncrypt, discussed in section 3.6.1. A key feature of NTRUOpenSourceProject is the definition of usable key parameters. These key parameters use a naming convention: ees401ep1, where 401 is the N parameter. Both the parameter q and p are hard-coded for every key parameter where *p=3* and *q=2048*.

The SVP, which forms the security basis of NTRUEncrypt, is considered a non-deterministic polynomial-time hard (NP-hard) problem. Even with approaches published by Lenstra, Lenstra & Lovász (1982), a quantum computer would not reduce the security of the algorithm. Thus NTRUEncrypt is one algorithm of several that researchers are investigating for use after quantum computers become widespread.

Part of the reason for the slower uptake of any cryptographic algorithm in information security is that "new" does not always mean "better". Standards require many years of academic research before they will be considered fit for use. Approval of expert research is a defensive mechanism to prevent non-experts from utilising a "new" and "improved" security algorithm, only to have an expert quickly defeat it months later. Due to this approach, NTRUEncrypt is still considered relatively new and only continued research by information security researchers will prove its soundness.

Another criticism of NTRUEncrypt is the fact that the company Security Innovation Inc hold a patent for the NTRUEncrypt cryptosystem. While they do provide an open source a version written in both Java and C, previous experience of patents around cryptographic algorithms creates difficulties for information security experts, and thus they are less likely to research and implement these algorithms. An example of this is the patenting of certain ECC curves which caused some issues for those wishing to implement the cryptographic algorithm. However, it is important to understand that even with a patent, the underlying security comes from a mathematical problem and shouldn't therefore limit research.

Investigating research into accelerating NTRUEncrypt Hermans, Vercauteren & Preneel (2010) compared NTRUEncrypt, RSA, and ECC and accelerated each algorithm using CUDA. Their research showed NTRUEncrypt was up to 1300 times faster than 2048-bit RSA. Hermans et al. research experiment additionally found that NTRUEncrypt, with parameters (1171, 2048, 3), was up to 117 times faster at encryption operations than ECC using the NIST-224 curve. This result is a considerable improvement as, during Hermans et al. (2010) experiment, they operated NTRUEncrypt at a considerably higher security level than RSA 2048-bit and ECC NIST-224.

2.3 Hardware

2.3.1 Central Processing Unit (CPU)

Oancea, Andrei & Dragoescu (2014) discuss the development of parallel computing, and the history of the movement towards General Purpose GPU (GPGPU) programming. The number of transistors and the clock frequency of processors has increased over time, (Oancea et al., 2014) compare the Intel 8086 with 29,000 transistors to the Intel Core i7-920 with 731,000,000 transistors. This process has continued to accelerate since 2014, and a modern Intel Broadwell microarchitecture now has approximately 1,900,000,000 transistors (Kowaliski, 2015), more than 2.5 times more transistors in the same area. This acceleration comes at the cost of increased heat generation and therefore thermal design becomes a limiting factor. To somewhat alleviate this issue, processor designers can introduce multiple cores. By 2015 it was common to have a modern computer with a dual core processor, with some being quad core with high-end servers and workstations being up to 12 cores per processor. While these significant increases are important,

computing power requirements however, have increased at a faster rate and software now demands significantly higher performance.

2.3.2 Graphics Processing Unit (GPU)

Nvidia GPUs take advantage of the power of a Single Instruction, Multiple Data (SIMD) stream architecture. Classified by Flynn in 1972, SIMD architecture and many others are used throughout computing. The advantage of SIMD over Single Instruction, Single Data (SISD) is the natural parallelism that it provides, allowing larger amounts of data to be processed simultaneously. While SIMD is the general architecture, Nvidia has slight differences in design and implement what they call Streaming Multiprocessors (SM). The difference between Nvidia's SMs, compared to other SIMDs, is their design. Nvidia has attempted to reduce complexity for developers, allowing them to take advantage of hardware to accelerate different algorithms, thus removing some of the control over low-level SIMD controls. At the same time, Nvidia's system is highly optimised to maximise the use of each SM, improving efficiency. Nvidia concentrates on allowing code to be deployed from one system to another without the need to rewrite accelerated code. Yang & Goodman (2007) explain that the increased use of GPGPU programming is due in large part to advances in the performance of GPUs over CPUs, as well as industry leaders, such as Nvidia, creating easier development platforms such as CUDA.

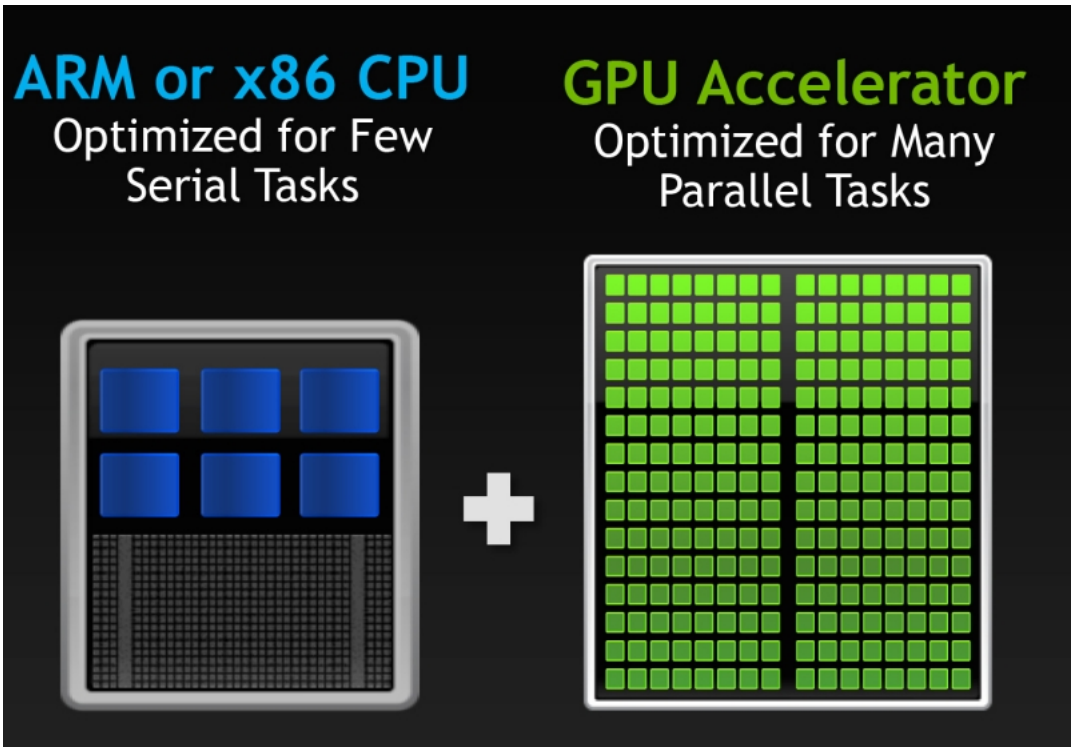

Figure 2.14: CPU vs GPU(Nvidia, 2008)

As illustrated in figure 2.14, the difference between a CPU and a GPU is at the hardware design level. A CPU consists of many cores, called arithmetic units that perform a series of calculations. While a single CPU can contain many arithmetic units, they have a control bus for communication, and thus do not operate directly in parallel. The instruction set of a CPU is significantly larger than a GPU, as it must be able to perform a larger set of variable calculations. A GPU is made up of many floating-point units. Dedicated units perform addition, subtraction and multiplication with floating point numbers. Due to the extremely specific application, these circuits can be highly optimised and when multiple units are combined, produce SIMD architecture. This dedicated use case is extremely useful for processing large datasets, or algorithms that perform well when parallelised.

Harrison & Waldron (2009) utilised an Nvidia 8800 GTX for their research. This model was the latest that allowed the use of DirectX 10, CUDA and 32-bit integer processing. Harrison & Waldron (2009) noted the need to take into account different memory architectures to improve performance. Each SM has a small amount of fast local storage, however, the bulk of memory is off-chip, making transfers comparably slower.

2.4 Software

2.4.1 Accelerated Programming

There is a constant battle between program efficiency and hardware performance. Traditionally it was inexpensive to develop more efficient algorithms, than to assign more processing power to a particular problem. This efficiency cost has alleviated for most basic tasks; however the requirements of computers to perform actions quickly has increased. This requirement is demonstrated by the differences between Nielsen (1994) and Ritter, Kempter & Werner (2015)'s studies. Nielsen (1994) suggests that 0.1 seconds is required for an application to feel instantaneous, while 1.0 second is required not to interrupt a user's flow. In contrast, Ritter, Kempter & Werner (2015) study, showed the highest level of acceptable latency was around 170 milliseconds for dragging tasks, and 300 milliseconds for low attention tasks such as tapping a button on a touchscreen. The problem is that data sets for more advanced calculations have increased at a rate greater than that of hardware or software efficiency. There are three routes that developers rely on: improve the user experience, improve the algorithms or functions used and improved processing resources.

A significant amount of research is undertaken every year, by both academia and industry, to improve the efficiency of applications. Since the inception of the CUDA platform by Nvidia, as well as OpenCL and OpenGL by the Khronos Working Group, it has become easier for software developers to implement, and a considerable number of programs now take advantage of this increase in computing capability. For example, software such as the latest Adobe Photoshop CC 2015, takes advantage of this technology to improve image manipulation. This same technology can be applied to research software, such as MATLAB, to allow for faster and more in-depth study by researchers. CUDA however, is only supported by Nvidia hardware and this limits its usefulness as many devices do not possess their hardware, especially mobile devices. CUDA is discussed further in section 2.4.2.

Creating new algorithms or improving the efficiency of existing algorithms can considerably improve the processing capabilities. An example is a dedicated hardware chip for video decoding built into modern systems. The implementation of this dedicated chip is more efficient and therefore requires less power than a CPU. Unfortunately, this chip will only work for a particular video codec, and due to its

speciality would not accelerate any other systems or processes on a device.

This can create complexities, for both developers and hardware designers to include low-level processing capabilities that will be utilised and as mentioned, many accelerated techniques are narrowly focused as the algorithm is tailor made to perform one function in the most efficient way possible.

Amdahl (1967) presents what is known as Amdahl's Law, which is used to find the maximum expected improvement to a system when only part of it is improved. This law is useful for parallel computing to identify the maximum increase in speed an algorithm may be able to achieve with multiple processors. Applying this law to cryptography algorithms would allow the identification of the minimum possible amount of time for execution, and therefore the algorithms efficiency.

2.4.2 Compute Unified Device Architecture (CUDA)

Compute Unified Device Architecture (CUDA) is described by Nvidia (2008) as a parallel programming model and software environment, with ease of use and low learning curve in mind. Harrison & Waldron (2009) were the first to be able to take advantage of this environment, as it was not available prior to 2008. It allows software developers to use a set of C extensions to abstract otherwise complicated hardware calls, thus allowing developers to take advantage of the increased processing power of an Nvidia GPU. A major criticism of CUDA is it is only able to run on Nvidia hardware. While Nvidia is considered an industry leader, this condition means many devices cannot take advantage of CUDA accelerated software.

2.4.3 Open Computing Language (OpenCL)

To overcome Nvidia's limitation on CUDA, the Open Computing Language (OpenCL) framework was developed and published (2008) through the non-profit company Khronos Group. This openness allows many technology industry leaders to design and implement this framework on their devices. OpenCL considers a system as many computing units, thus devices other than GPUs can also process algorithms. OpenCL allows for greater portability than CUDA, as most of Nvidia's GPUs can also run OpenCL; however portability comes at the cost of performance.

2.4.4 Open Graphics Library (OpenGL) and the OpenGL Rendering Pipeline

Introduced in 1992, OpenGL is designed to make it easier for developers to create 2D and 3D graphics applications on multiple device platforms. Woo, Neider, Davis & Shreiner (1999) describe the essential operations of OpenGL. Geometric data is required to construct shapes; various shaders are then executed to generate colour, position and other attributes. Next shaders convert into fragments, through a process called rasterization. Finally, each fragment can run a fragment shader to determine final attributes. An example illustration, in figure 2.15, of the OpenGL pipeline process as described by Woo, Neider, Davis & Shreiner (1999).

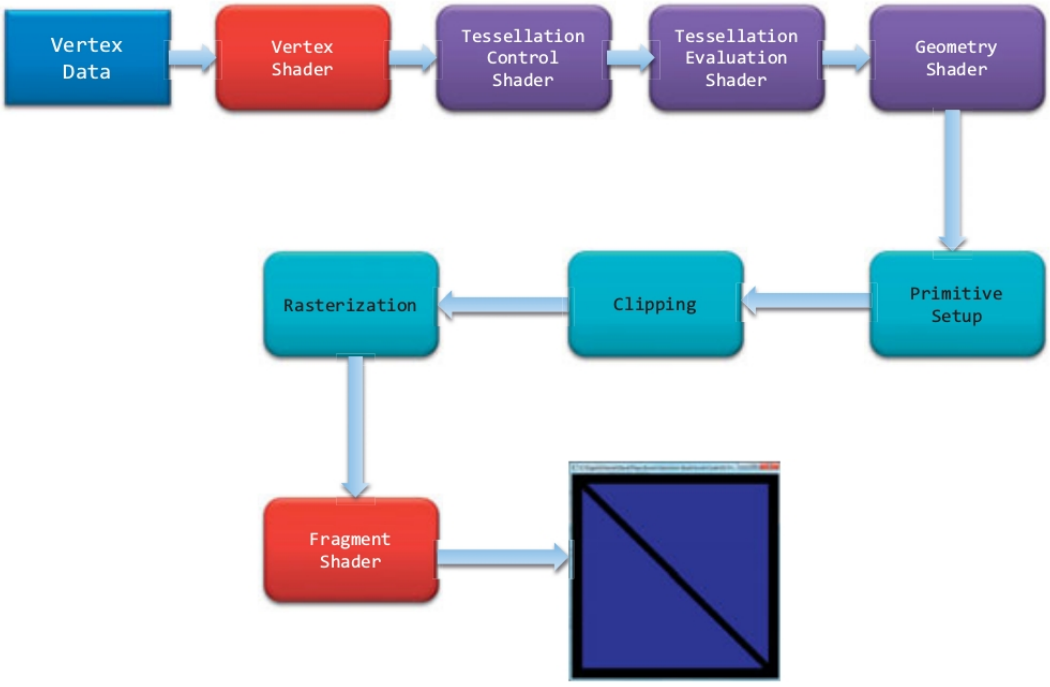

Figure 2.15: OpenGL Pipeline (Woo et al., 1999)

2.4.5 C

Considered the foundation of modern computer programming, developed in the early 1970's by Ritchie. C is currently in the top 10 most used programming languages according to Borges, Valente, Hora & Coelho (2015). Historically tied to the development and release of the Unix operating system, the operating system development moved from assembly to C. Popularity for the C programming language grew due to its efficiency and cross-platform functionality. Development of

the C language still continues to this day, and has influenced many other programming languages.

C is extremely efficient. "Comparing Java vs. C/C++ Efficiency Differences to Interpersonal Differences" (n.d.) investigate the performance difference between Java and C/C++ on 40 different implementations of the same program. "Comparing Java vs. C/C++ Efficiency Differences to Interpersonal Differences" found that Java required 2-3 times more memory and three times more runtime than the C or C++ implementations. Considering the hardware available in the early 1970's, it was especially important for programming languages to be more efficient with resources. C is a compiled language, and thus optimisation occurs during program compilation. C's compilers are mature and efficient, and many powerful compilers exist for various systems and hardware configurations. An advantage of C is its ability to utilise memory pointers. Operating on variables in place, rather than being copied, which allows for more efficient memory control. Memory pointers have produced efficient data structures, such as arrays. Compared to other programming languages available around 1970, C was also the first to provide a platform for developing graphical interfaces. The source code can be transferred to other systems and compiled specifically for each system using their compiler. While possible, for C program compilation for multiple different architectures, it cannot achieve the same level of portability as Java.

Although not as complex linguistically or syntactically as the assembly programming language, C is considered a difficult language to learn, when compared to modern languages such as Java or Python. This complexity can cause developers to mishandle pointers and memory which can produce problems, such as crashes or memory leaks. Some compilers can attempt to fix or warn the developer about these issues, while other compilers might mask certain problems or cause them to become worse. Perhaps the reason for movement away from C is due to C not being an object oriented programming language and therefore does not benefit from the object oriented design. Due to the age of C, there are multiple versions: ANSI, C99, C11. Some compilers and architectures have additional custom versions. CUDA specifically is an extension of C++ and thus will not compile and run on a machine that does not have the CUDA environment. While this was a better alternative for cross-platform development in the early 1970's many alternative modern programming languages now handle this cross-platform development in a simpler way.

2.4.6 JavaScript

According to Borges, Valente, Hora & Coelho (2015), JavaScript is the language with the highest number of popular systems. Borges, Valente, Hora & Coelho's research found JavaScript is responsible for more than one-third of the popular applications on GitHub. Herhut et al. (2013) describe JavaScript as the most popular language on the web, with the ability to run on many platforms. Flanagan (2011) points out that all modern browsers include JavaScript interpreters, this, in his opinion, makes JavaScript the most ubiquitous programming language in history. One of the primary drivers towards the use of JavaScript is the emergence of HTML5. According to Flanagan, this has changed JavaScript from a simple scripting language into an efficient general purpose programming language.

As mentioned by Flanagan, all modern browsers have JavaScript interpreters. Recent research by Gartner (2015) suggests almost a doubling of the number of Internet-connected devices every two years. This research would indicate an increase of expected interactivity between different systems. Developers, therefore, search for platforms that allow them to code the least, while accessing the greatest number of users for their software. Java and other similar programming languages follow a 'write once, run everywhere' approach. However, the reliance on a single company or organisation to provide an interpreter for every system makes this difficult on closed source systems. Most modern computer systems provide a browser, and all modern browsers support JavaScript. The end user does not have to install anything to utilise JavaScript based applications through the browser.

Nicholls (2012) discussed an early attempt at implementing JavaScript on a GPU. Nicholls's creation of LateralJS was an attempt to allow JavaScript to take advantage of hardware accelerated operations and data parallelization on GPUs. Nicholls chose OpenCL over CUDA, as it would run on many different hardware configurations. The benchmark that Nicholls (2012) performed was a small loop of FLOPs on an ATI Radeon 6770m GPU, Intel Core i7 CPU CL and within the V8 JavaScript engine purely on the Intel Core i7 CPU. Unfortunately, the experiment was unsuccessful in demonstrating acceleration using a GPU as illustrated in figure 2.16.

GPU CL ATI Radeon 6770m	CPU CL Intel Core i7 4x2.4Ghz	V8 Intel Core i7 4x2.4Ghz
116.571533ms	0.226007ms	0.090664ms

Figure 2.16: Execution Time (Nicholls, 2012)

JavaScript executes on the client side when implemented through a browser. Client side execution means the web server can transfer some processing requirements to the end user's computer or device. JavaScript allows for manipulating the graphical interface for the end user, of which the client has more information about their system. JavaScript does not require installation and JavaScript code is minimal or can be minimised. Compared to other programming languages JavaScript is considered an easy language to learn because it is dynamic, weakly typed and is executed by any modern browser.

JavaScript is an interpreted language, and therefore developers do not have access to lower level system controls, and must rely on an interface. JavaScript receives some criticism for being dynamic and weakly typed, resulting in a considerable number of error checking and parsing, especially of user input data. As JavaScript's execution is client side, there is no guarantee that the client is either genuine or malicious. Malicious JavaScript can also be passed from the server to the client to execute. Similar to this security issue, the reliance on the end user and their system to interpret JavaScript can cause problems. Different systems can handle execution differently, especially bleeding edge applications. Constant sanitisation of end user input is required to ensure it does not contain errors or is a malicious attack. As of 2015, the majority of multi-core systems cannot take advantage of multiple cores as JavaScript is still single-threaded. The remedy for this is with web workers and compiled web code, however neither have been standardised, and web workers are not compatible with all browsers.

2.4.7 WebGL

Until recently, it has been difficult to create modern cross-platform programs that utilise hardware acceleration, given the complexity of accelerated programming. The ubiquity of JavaScript meant a solution was sought to enable hardware acceleration in a modern browser. In March 2013, WebGL became a standard and since then it has been integrated into most modern desktop and mobile browsers. The advantage of WebGL is that it provides a Javascript API for 3D and 2D graphics processing without the requirement to download extra software. The obvious advantage is access to the GPU through JavaScript, as this was not previously available to developers.

2.4.8 Three.js

While WebGL gives access to the GPU via JavaScript; it does not, however, make it easier for developers to write applications which take advantage of the hardware access. Three.JS is a library that makes WebGL easy to use. Three.JS wraps many key OpenGL, and WebGL features in an easy to use library, while still allowing access to lower levels such as fragment shaders, vertex shaders, and multipass rendering.

2.5 GPU Cryptography

Cook & Keromytis (2006) first investigated the use of GPUs for cryptography by implementing the AES encryption standard through OpenGL. While their research provided promising results, it ran into many problems such as the lack of modular arithmetic, unsigned integers, branching and large integers. These limitations meant that Cook & Keromytis could only implement simple symmetric cryptographic algorithms on a GPU.

Yang & Goodman (2007) furthered Cook & Keromytis's research by taking two more modern GPUs, the AMD HD 2900 XT and AMD X1950 XTX. Due to the advancement in technology in only a year, Yang & Goodman were able to produce a positive result, managing up to 16 times faster execution of AES. While Yang & Goodman overcame some of the issues faced by Cook & Keromytis's research, many still remained. The GPU hardware available in 2007 provided integers and branching although it still did not provide support for large integers,

hence the continued concentration on symmetric encryption such as DES and AES.

At a similar time, Harrison & Waldron (2007) investigation produced similar results as Yang & Goodman They, however, utilised a Nvidia GeForce 7900GT GPU and their conclusion was that the available GPU hardware was still too limited, and only bulk encryption and decryption was useful. One suggestion made was looking to future work, utilising the new Nvidia G80 architecture with CUDA.

Harrison & Waldron (2009) were able to take advantage of the new development environment provided by the Nvidia CUDA API. As noted in their previous research published in 2007, this environment was not available prior to 2008. Harrison & Waldron provide some of the first research investigating asymmetric cryptography executed using a GPU. Their research demonstrates 1024-bit RSA decryption on a GPU operating four times faster than a similar implementation on a CPU. Not only did this demonstrate an impressive acceleration of RSA, they additionally demonstrated higher throughput and decreased latency. Limitations on large numbers still existed, however Harrison & Waldron demonstrated, through previous research, that this restriction would most likely cease to be a problem in the next generation of GPU hardware.

The research Harrison & Waldron (2010) investigated accelerated cryptography at the operating system level. Operating system level cryptography provides many advantages over previously researched cryptography, as it means a developer no longer has to include special accelerated cryptography libraries to use this feature. However, this approach, and others before it, are limited to extremely specific use cases and hardware architectures. While Harrison & Waldron (2009) and Harrison & Waldron (2010) saw great improvements to acceleration from the utilisation of the CUDA API, the requirement that a system must have a Nvidia GPU limits its overall use. However, all previous research from 2006 to 2010 was written in C or C++, and this requires performing the compilation of each program or library for each system architecture.

Nicholls (2012) research attempted to implement an OpenCL interface via JavaScript. This early attempt at implementing JavaScript on a GPU was unsuccessful. However since 2012 many previously discussed libraries, technologies, and standards have advanced and with the current software and hardware convergence on this topic should finally provide a strong foundation for device agnostic accelerated

cryptography.

In sections 2.1 to 2.2, an examination of what cryptography is, the development of different cryptographic algorithms, and the current state of modern cryptography. In sections 2.3 to 2.4, a discussion was made of the currently available hardware and software. All discussed research should establish an understanding of the current state of cryptography in information security. Cryptography represents a computation problem, increased security requires increased processing capability. Thus algorithms such as Diffie-Hellman, RSA, and ECC are considerably slow and inefficient compared to NTRUEncrypt. Given the successful research into the area of NTRUEncrypt acceleration using a GPU, this speedup may be able to be increased further.

Moore's law and the pending threat of quantum computers, require further research into current alternatives, and while currently used cryptographic algorithms are sufficient at the moment they are inefficient on mobile platforms. Previous researchers in the area of accelerated cryptography were constrained by existing hardware and software, of which these problems no longer exist.

3 Research Design

3.1 Aims

This research aims to:

- Improve overall cryptographic research and knowledge.
- Increase the body of knowledge in information security.
- Provide a research platform for other accelerated cryptographic algorithms.
- Create more efficient security options for mobile and low power systems.
- Investigate quantum secure algorithms.
- Investigate the growing use of JavaScript and provide some insight into its validity for cryptography.

3.2 Methodology

The experimental research design allows for control over the variables defined in the experiment, allowing more accurate results to determine relationships between sets of data. This approach, however, requires knowledge of all of the possible variables in a system for experimentation. For most experiments, this is infeasible. Some variables are unknown at the time of experiment design. Known variables are either controlled, variable, or insignificant. The experiment designer must be cautious not to dismiss variables as insignificant when they may appear to be. The advantage of this approach is that effective experiment design should make the experiment repeatable and results comparable to others. The data is first-hand, direct observation allows for a reduction in miscommunicated or misinterpreted data. March & Smith (1995) discuss two dimensions for research in information technology, a broad research design, and broad type of outputs. The first dimension concentrates on the design and research activities: build, evaluate, theorise, and justify. While the first dimension takes its strengths from natural science research, the second is derived specifically for information technology, concentrating on outputs: representational constructs, models, methods, and instantiations. The research experiment presented in this paper covers both of these dimensions in order to provide validation for the proposed hypothesis and increase human knowledge in the area of information security.

3.3 Questions

The questions that will be answered from this research are as follows:

1. Can in-browser cryptography be accelerated using a GPU?
2. Does increasing the bit size of a cryptographic algorithm have a positive linear correlation to processing time and security?
3. Does the programming language that a cryptographic algorithm is implemented in have an effect on processing time?
4. Does measuring the processing time of cryptographic algorithms provide an accurate means of comparison?
5. Is a GPU faster at performing public key encryption than a CPU implementation of the same algorithm?
6. Is a GPU faster at performing private key decryption than a CPU implementation of the same algorithm?
7. What is the impact of latency on a cryptographic algorithm?
8. Does a GPU implementation of a cryptographic algorithm increase throughput compared to a CPU implementation?

3.4 Hypothesis

Question 1

Can in-browser cryptography be accelerated using a GPU? Previous research, discussed in section 2.5 of this research, indicates that accelerating a cryptographic algorithm is possible. Based on previous research utilising OpenGL and cryptographic acceleration this research will achieve a similarly successful result. The advancement of the JavaScript language and the libraries available provide evidence that this examination and experiment will produce a positive result.

Question 2

Does increasing the bit size of a cryptographic algorithm have a positive linear correlation to processing time and security? Many cryptographic algorithms have a positive exponential correlation to the bit size. Algorithms such as RSA have been studied and demonstrate cubic growth for time complexity. Increasing the bit size of a cryptographic algorithm should enhance security, although not proportionally and this proportion differs between various algorithms.

Question 3

Does the programming language of an implemented cryptographic algorithm have an effect on processing time? Compiled languages can adapt for the intended system or architecture. During execution languages such as C do not require the overhead of interpretation, unlike a language such as JavaScript. However given recent advancements in increasing JavaScript performance, the results of this experiment should show that JavaScript is not significantly slower than C, C++, or Java.

Question 4

Does measuring the processing time of cryptographic algorithms provide an accurate means of comparison?

Measuring the time required to solve a problem allows an understanding of the computational complexity of an algorithm. In this research algorithms that provide the highest security in exchange for the lowest complexity are considered the most efficient. Measuring the processing time should provide an accurate real world benchmark for comparison between other cryptographic algorithms.

Question 5

Is a GPU faster at performing public key encryption than a CPU implementation of the same algorithm? Based on previous research, as discussed in section 2.5, the results from desktop systems should only see acceleration at the highest bit size and data throughput. However, lower bit size values will be similar or slightly slower in speed than a CPU implementation. Tested mobile devices should receive the greatest acceleration of encryption as the mobile systems specified in section 3.6.2 have more powerful GPUs compared to their CPU.

Question 6

Is a GPU faster at performing private key decryption than a CPU implementation of the same algorithm? Some acceleration should be observed however only at the higher end of bit size and data throughput. Similar to the encryption hypothesis, however, for researched cryptographic algorithms, the decryption

operation appears to be more computationally intensive compared to encryption. Therefore a greater level of acceleration in decryption is expected.

Question 7

What is the impact of latency on a cryptographic algorithm? Due to the architecture of all systems listed in section 3.6.2, the latency between the CPU and GPU should be greater between the CPU and RAM. Jang, Han, Han, Moon & Park (2011) produced results showing up to 15 times increased latency for GPU operations compared to CPU. For GPU operations this latency will be the largest portion of time taken by encryption or decryption. Latency should not be an issue for CPU bound operations.

Question 8

Does a GPU implementation of a cryptographic algorithm increase throughput compared to a CPU implementation? Based on the research discussed in section 2.5, the output of this research should show an increase in throughput from a GPU implementation compared to a CPU implementation.

3.5 Phases

The aim of this research is to answer the research questions from section 3.3. The research consists of four major phases, with a total of 13 minor phases. The major phases are research, development, experiment, and analysis and discussion.

3.5.1 Research

Initial Research This research is required to find and test various libraries and options for development and examination. The first stage is to examine research that already exists and see what and how previous studies may have implemented their solutions. Next, a selection of cryptographic algorithms is compared against previous investigations, future potential and previous success of acceleration. Finally, a selection of implementation strategies and libraries is briefly tested to ensure successful performance of the research.

Pilot Test The pilot test demonstrates if algorithms could be accurately measured across different systems and the comparisons between the two matched previous research and knowledge about how the algorithms operate.

3.5.2 Development

At the time of this research, there is no benchmark software for academic research for NTRUEncrypt. Development of many other benchmarks was required to answer the research questions. A number of benchmarks are needed to establish a baseline between compiled code executed on each device, interpreted code and accelerated code. This approach allowed the comparison of benchmarks previously used by cryptographic researchers on to the JavaScript, and accelerated JavaScript code implemented specifically for this research.

Android Benchmark The development of an Android application allows collection of results from the mobile devices listed in section 3.6.2. The application utilises the built-in encryption libraries available to Java and Android as well as the open source version of NTRUEncrypt.

Java Benchmark Development of a simple command-line version running the same classes as the Android application to provide experimental testing benchmark results for all Desktop, Server and Laptop hardware listed in section 3.6.2.

JavaScript Benchmark Development of a JavaScript benchmark to provide a clear comparison between compiled implementations of cryptographic algorithms and interpreted implementations.

NTRUEncrypt.js Development of a JavaScript version of NTRUEncrypt. The open source Java version provided a basis for which to implement the algorithm.

NTRUEncrypt-GPU.js Extension of NTRUEncrypt.js except using the Three.js library to place some of the computational load onto the GPU of the system executing the code.

3.5.3 Experiment

Crypto++ is used to establish a comparison to other researchers' efforts at accelerating cryptography.

Java provides a comparison of a compiled language. A desktop and Android version would be similar to minimise variables in implementation differences. NTRUEncrypt currently has an open source version written in Java.

The JavaScript experiment is executed on hardware systems that support a modern browser. The experiment includes RSA and ECC from a well known cryptographic JavaScript library as well as NTRUEncrypt entirely developed for this research. Finally, the testing of an accelerated JavaScript NTRUEncrypt provides the data required to answer the key questions of this research.

3.5.4 Analysis and Discussion

Data Gathering Gathering data created by the experiments and organising it in such a way as to be useful for later analysis.

Data Analysis Analysis of data, creating comparisons, looking for trends. Graphing data related to answering research questions as well as interesting or unexplained results.

Discussion The discussion surrounding findings from the experiments. Allows the discussion of obtained results, answers the research questions and either supports or disproves the hypothesis.

Conclusions Conclusions are drawn from results directly related to the research investigation, and this provides a statement of facts obtained through research and experimentation.

3.6 Experimental Setup

3.6.1 Software

Crypto++ (Version 5.6.1)

Crypto++ is an open source library that implements a large number of cryptographic algorithms. Initially released in 1995 it has provided

the basis for academic research, this makes it especially useful for comparison of past and future research in cryptography. The version of Crypto++ that was installed was only able to provide relevant benchmarks for AES.

OpenSSL (Version 1.0.1f 6 Jan 2014)

OpenSSL, initially founded in 1998, is a library of functions to allows communication using SSL or TLS protocols. It is widely used on Linux operating systems and provides some basic cryptographic functions for the operating system.

Android Studio (Version 1.3.2)

Android Studio is the official IDE for Android application development. Utilised to develop a mobile application to provide performance data into the difference between native cryptography and JavaScript. Base Java security libraries provide RSA, ECC, AES, and Hashing algorithms to compare to both JavaScript and GPU accelerated cryptographic algorithms.

AsmCrypto (Version 11/12/2015)

Minified JavaScript was downloaded from GitHub. A performance JavaScript implementation of cryptographic algorithms. It was developed in response to a growing demand for cryptographic algorithms in JavaScript and a requirement for a focus on performance. Optimisations made for this library demonstrate increases up to 40% faster than other cryptographic libraries. For the focus of this research, it only supports AES and RSA, with plans to implement ECC support in the future.

Web Cryptography API

The Web Cryptography API, also known as WebCrypto, is an interface for JavaScript. It allows operating system agnostic access to implementations of cryptographic algorithms, without exposing the data directly to JavaScript. Encryption algorithms available are dependent on the browser and operating system used. WebCrypto was able to provide AES, RSA, and ECC for all systems tested.

Firefox (Desktop Version 45.0.1) (Mobile Version 45.0.1)

Firefox is a modern web browser built by the Mozilla Foundation. It provides emphasis on open standards, security, extensions and internet communities.

Chrome (Desktop Version 49.0.2623.110) (Mobile Version 39.0.2171.93)

Chrome is another modern web browser developed by Google. According to StatCounter (2016), it is the most used browser with 47.16% market share.

NTRUOpenSourceProject ntru-crypto (Version 01/02/2016)

Available from GitHub, NTRUOpenSourceProject provides an open source NTRUEncrypt cryptographic algorithm and reference code for C and Java programming languages.

Three.js (Version T67)

Described as a lightweight 3D library, the goal of Three.js is to wrap functions of WebGL and make it easier for developers. Available from GitHub, this library is constantly receiving updates to increase the number of features and stability.

3.6.2 Hardware

Table 3.1: List of Mobile Hardware used for Experiments

	Samsung Galaxy S2	**ASUS Zenfone 2**	**Samsung Tab 10.1**	**Nvidia TK1**
CPU	Dual-Core Cortex-A9	Quad-Core Atom Z3580	Dual-Core Cortex-A9	Quad-Core Cortex-A15
GPU	Mali-400 GPU	PowerVR G6430	Nvidia Tegra 2 T20 ULP Geforce	Nvidia Kepler
RAM	1 GB	4 GB	1 GB	2 GB
OS	Android 4.1.2	Android 5.0	Android 3.1	Ubuntu 14.04 LTS

Table 3.2: List of Desktop and Laptop Hardware used for Experiments

	Custom Desktop	**HP Desktop**	**Toshiba Thinkpad**
CPU	Quad-Core Intel i5 3570k	Quad-Core Intel i5 4570	Dual-Core Intel Core 2 Duo T9400
GPU	Nvidia 660 GTX	AMD Radeon R7 360	Intel GMA 4500MHD
RAM	16 GB	16 GB	2 GB
OS	Windows 10	Ubuntu 14.04 LTS	Lubuntu 14.04 LTS

Table 3.3: List of Server Hardware used for Experiments

	Pohutukawa Server	**Tesla Server**
CPU	Hex-Core with HT Intel Xeon X5660	Hex-Core with HT Intel Xeon E5-2620
GPU	Nvidia Quadro 6000 GPU	Nvidia Tesla K40m GPU
GPU2	Nvidia Tesla C2070 GPU	Nvidia Tesla K40m GPU
RAM	24 GB	32 GB
OS	Ubuntu Server 16.04 LTS	Ubuntu Server 14.04 LTS

3.6.3 Variables

Table 3.4: Crypto++ Benchmark Variables

Independent Variables	**Dependent Variables**
Operating System	Algorithm Megabytes per Second
CPU Architecture	Algorithm Operations per Millisecond
Number of CPU Cores	
CPU Clock Rate	
Algorithm	

Table 3.5: Android and Java Benchmark Variables

Independent Variables	**Dependent Variables**
Operating System	Algorithm Megabytes per Second
CPU Architecture	Algorithm Operations per Millisecond
Number of CPU Cores	
CPU Clock Rate	
Algorithm	
Java Version	

Table 3.6: JavaScript Benchmark Variables

Independent Variables	Dependent Variables
Operating System	Algorithm Megabytes per Second
CPU Architecture	Algorithm Operations per Millisecond
Number of CPU Cores	
CPU Clock Rate	
Algorithm	
Browser	

Table 3.7: NTRUEncrypt.js Benchmark Variables

Independent Variables	Dependent Variables
Operating System	Algorithm Megabytes per Second
CPU Architecture	NTRUEncrypt Encryption Polynomial Convolution
Number of CPU Cores	NTRUEncrypt Encryption Polynomial Addition
CPU Clock Rate	NTRUEncrypt Decryption Polynomial Convolution 1
Algorithm	NTRUEncrypt Decryption Polynomial Convolution 2
Browser	Algorithm Operations per Millisecond

Table 3.8: NTRUEncrypt-GPU.js Benchmark Variables

Independent Variables	Dependent Variables
Operating System	Algorithm Megabytes per Second
CPU Architecture	NTRUEncrypt Encryption Polynomial Convolution
Number of CPU Cores	NTRUEncrypt Encryption Polynomial Addition
CPU Clock Rate	NTRUEncrypt Decryption Polynomial Convolution 1
GPU Architecture	NTRUEncrypt Decryption Polynomial Convolution 2
Bus between CPU and GPU	Algorithm Operations per Millisecond
Algorithm	
Browser	

3.6.4 NTRUEncrypt

3.6.5 Pilot Test

To test the hypothesis of this research an investigation into the trend of the difference in speed between AES, RSA, and ECC was required. Two tests were executed on each the HP Elite Desktop and the Tesla Server. The first test was to establish whether Crypto++ would operate correctly, this was simply installed on both systems and then run using the command: *cryptest -b*

Examining the results in table 3.9 it is evident that AES, on the HP Desktop, is significantly faster than on the Tesla Server. The reason for this becomes clear when the command: *cat /proc/cpuinfo | grep aes*

is executed displaying that the HP Elite Desktop has AES-NI enabled while the Tesla Server does not.

Table 3.9: Crypto++ Pilot Test Results (in MB/s)

Algorithm	**HP Desktop**	**Tesla Server**
AES/GCM	2115	248
AES/CCM	584	146
AES/CBC (128-bit key)	693	257
AES/CBC (192-bit key)	586	220
AES/CBC (256-bit key)	508	192

Executing: *openssl speed rsa ecdh* on each machine produced the results shown in table 3.10. These results show that while the HP Elite Desktop is slightly faster than the Tesla Server, it does not have the same hardware acceleration available for RSA and ECDH. As expected for both RSA and ECDH as the bit size is increased, the number of operations per second decreases, due to the added computational requirements. Taking into consideration equivalent security bit sizes, as illustrated in table 3.11, comparing ECDH nistp192 and RSA-1024, the HP Elite Desktop ECDH nistp192 is 1656.5 operations per second slower than RSA -1024 Signing. However as the bit size of ECDH is increased, using curve nistk571 which is equivalent to 15360 bit RSA (which provides 3.75x more security than 4096 bit RSA), it can be observed that ECDH nistk571 is 4.85 times faster than 4096 bit RSA Signing.

Table 3.10: OpenSSL Pilot Test Results (in Operations per Second)

Algorithm	**HP Desktop**	**Tesla Server**
RSA-512 Sign	22209.1	21236.4
RSA-512 Verify	278142.7	252911.1
RSA-1024 Sign	7154.8	6531.2
RSA-1024 Verify	102538.4	93238.0
RSA-2048 Sign	920.6	859.9
RSA-2048 Verify	31434.7	28045.3
RSA-4096 Sign	124.4	121.9
RSA-4096 Verify	8513.3	7622.0
ECDH secp160r1	6733.0	6034.3
ECDH nistp192	5498.3	5000.9
ECDH nistp224	9685.7	8763.8
ECDH nistp256	5438.7	4923.4
ECDH nistp384	1652.5	1460.0
ECDH nistp521	1473.1	1319.6
ECDH nistk571	602.9	541.1

Table 3.11: Encryption Equivalent Bit Sizes

Bits of Security	**AES**	**ECC**	**RSA**	**NTRUEncrypt**
80	-	160	1024	449
112	-	224	2048	677
128	128	256	3072	761
192	192	384	7680	1087
256	256	521	15360	1499

These results demonstrate that the initial set up of benchmarking software is correct, as well as providing a primary basis for comparison of cryptographic algorithms and positive evidence for hypothesis posed for this research.

4 Research Implementation

4.1 Crypto++ Benchmark

Crypto++ was selected to provide a baseline examination between the speed of different cryptographic algorithms. Both Yang & Goodman (2007) and Harrison & Waldron (2009) have used Crypto++ in previous cryptographic research, and the results gathered from this research experiment allows comparisons between them. Since Crypto++ is built using the C language, its compiler requirements make it extremely complicated to compile and execute on mobile platforms. Therefore, for these experiments it was installed and executed on all desktop, laptop or server hardware as defined in section 3.6.2.

First, the latest sourcecode for Crypto++ was downloaded and compiled. Next, executing the command *./cryptest.exe* produces an HTML document with timings of each algorithm. For accuracy, this benchmark was run several times to gather minimums, maximums, and variance of each algorithm on every system. The results can be found in section 5.1.

4.2 Android and Java Encryption Benchmark

Mobile platforms are not the designed target for Crypto++; thus a Java cryptographic benchmark was developed to produce relevant comparisons. The Java benchmark implementation used the standard cryptographic libraries for Java, as well as the latest version of the open source NTRUEncrypt available on GitHub. The Java benchmark was then directly imported into both an Android and Java project. The benchmark provides timing information for RSA, ECC, and NTRUEncrypt.

The Android Application Package (APK) was copied to the Android devices listed in section 3.6.2 and installed and executed. The Android application simply measures the time taken for the execution of each algorithm, and outputs this information to the screen via a multi-line textbox. This data is then collected into tables for later analysis. The Java version involved copying the compiled Java Archive (JAR) file to the target system and executing it via the command *java -jar CryptoBenchmark.jar*. The timing of each algorithm outputs to the command line and is then entered into tables for later analysis.

4.3 JavaScript Crypto Benchmark

Similar to the Android implementation, there are few benchmarked numbers available in cryptographic research. To provide a basis for comparison, two libraries were selected: AsmCrypto and WebCrypto. These libraries were chosen to provide timings for hashing, AES, RSA and ECC. Executing JavaScript cryptography initially encountered a problem; the browser tab would close or crash. Crashing occurred due to a feature within the browser that assumed the JavaScript had failed if it took longer than 10 seconds. Several workarounds for this issue were tested. The first option was to place the browser in debug mode, thus disabling the timeout issue. However, the desktop version of Chrome was the only browser that allowed, or correctly followed, this particular feature. The second option was to run the tests on a separate thread, however at the time of writing this research JavaScript does not provide a way of running separate threads. There is, however, a limitation of threading within JavaScript using Promises. Promises execute and attempt to return and provide the expected result of the called function. The timing tests follow using this second option of Promises. However, some of the libraries and algorithms had a considerable variance, and this may be the result of the Promises and is discussed in further detail in section 5.3.

4.4 NTRUEncrypt.js

A current implementation of NTRUEncrypt in JavaScript did not exist. The first portion of the experiment is to break apart the NTRUEncrypt algorithm, understanding each part, and then complete an implementation in JavaScript. Fortunately, many technical documents exist as well as open source versions written in C and Java, and these provided a strong foundation. This benchmark was implemented over the course of several weeks.

The creation of NTRUEncrypt.js required no extra libraries. The possibility of optimisations made in the future was maintained by not implementing additional mathematical libraries.

JavaScript has no native way to represent polynomials. This was an initial problem. However, as illustrated in figure 4.1, a solution was to utilise arrays and the value's position within the array. The position in the array is viewed mathematically as the coefficient x power component. Thus the first value is x^0 and then next x^1 and so on, for the

length of the polynomial. The array representation makes polynomial addition trivial, adding like terms consists of adding the two values at the same point from two arrays.

Array Position	[0, 1, 2, 3, 4, 5]
JavaScript Array	[3, 4, 8, 3, 6, 3]
Polynomial	$3 + 4x + 8x^2 + 3x^3 + 6x^4 + 3x^5$

Figure 4.1: Example of Polynomial Representation in JavaScript

Multiplication of polynomials, also known as convolution, is different for NTRUEncrypt as the polynomial is within a ring. Convolution involves multiplying every value by every other value, performing multiplication of coefficients with an addition of variables. The data structure provides the difference where the position in the array is the variable power. The ring causes variable powers to wrap around. For example, using the values in figure 4.1 [3, 4, 8, 3, 6, 3] and performing a convolution with another polynomial: [8, 7, 1, 3, 2, 4] taking the value 6 (position 4) and multiplying it by 1 (position 2) results in $6x^6$. However these polynomials are of length 6, therefore the result would wrap around, and the output would be $6x^0$ as $6 \, mod \, 6 = 0$. Next, each result is added to the value already stored in the output array at that position. An illustrated example in figure 4.2 shows polynomial convolution.

Polynomial Array 1	[3, 4, 8, 3, 6, 3]
Polynomial Array 2	[8, 7, 1, 3, 2, 4]
Convolution Values	$\begin{bmatrix} 24 & 21 & 3 & 9 & 6 & 12 \\ 16 & 32 & 28 & 4 & 12 & 8 \\ 16 & 32 & 64 & 56 & 8 & 24 \\ 9 & 6 & 12 & 24 & 21 & 3 \\ 6 & 18 & 12 & 24 & 48 & 42 \\ 21 & 3 & 9 & 6 & 12 & 24 \end{bmatrix}$
Result	[92, 112, 128, 123, 107, 113]

Figure 4.2: Example of Polynomial Convolution

Some operations require the polynomial coefficients to operate within a positive modulus. The operation takes the modulus of each coefficient, and while that result is negative adds the modulus until the value is positive. For example $-19 \, mod \, 5 = -4$ this coefficient would then become $1 \, mod \, 5$ as $-4 + 5 = 1$.

Two other operations required were dividing or multiplying a polynomial by x. Effectively dividing the polynomial by x within a ring

shifts all coefficients left and multiplying shifts all coefficients right, both with wrap around. The example in figure 4.3 illustrates both the multiplication and division operations.

Polynomial Array A	[3, 4, 8, 3, 6, 3]
Divide A by x	[4, 8, 3, 6, 3, 3]
Multiply A by x	[3, 3, 4, 8, 3, 6]

Figure 4.3: Example of Polynomial Multiplication and Division by x

Key generation took up a large portion of development time, specifically finding the inverse of a polynomial. The polynomial inverse operation calculates f_q and f_p, which first requires the selection of the private polynomial f. The algorithm is similar to the Euclidean algorithm, although for polynomials, performing a large number of shifts and swapping. Not all polynomials generated have inverses and thus calculating the polynomial inverse will fail in this case and return null; therefore another polynomial f must be selected.

The process of calculating the two inverse polynomials f_{p} and f_{q} is performed by two separate algorithms. The first approach, as illustrated in code in figure 4.4, calculates the inverse polynomial f in modulo prime p. The second takes the output f_p and reduces the computation time by taking advantage of the fact that q is a power of two.

```
function calculatePolynomialInverse(polyF, prime, invModArray) {
  var k = 0; var b = []; var c = []; tmpf = []; tmpg = [];
  for(i = 0; i < N + 1; i++) {
    b[i] = 0; c[i] = 0; tmpf[i] = 0; tmpg[i] = 0;
  }
  b[0] = 1;
  for(i = 0; i < N; i++) {
    tmpf[i] = modPrime(f[i], prime);
  }
  tmpg[N] = 1; tmpg[0] = (prime   1);
  var degreeF = getDegree(polyF); var degreeG = N;
  while(true) {
    while((tmpf[0] == 0) && (degreeF > 0))
    {
      degreeF  ; tmpf = divideByX(tmpf);
      c = multiplyByX(c); k++;
    }
    if(degreeF == 0) {
      var f0Inv = invModArray[tmpf[0]];
      if(f0Inv == 0) {
        return null;
      }
      var shifty = N   k; shifty = shifty % N;
      if(shifty < N) {
        shifty = shifty + N;
      }
      var inversePoly = [];
      for(i = 0; i < N; i++) {
        inversePoly[(i+shifty) % N] = modPrime(f0Inv * b[i],
            prime);
      }
      return inversePoly;
    }
    if(degreeF < degreeG) {
      var tmpSwap1; tmpSwap1 = tmpf;
      tmpf = tmpg; tmpg = tmpSwap1;
      var tmpSwap2; tmpSwap2 = b;
      b = c; c = tmpSwap2;
      var tmpSwap3; tmpSwap3 = degreeF;
      degreeF = degreeG; degreeG = tmpSwap3;
    }
    var u = modPrime(tmpf[0] * invModArray[tmpg[0]], prime);
    for(i = 0; i < tmpf.length; i++) {
      tmpf[i] = modPrime((tmpf[i]   (u*tmpg[i])), prime);
    }
    for(i = 0; i < b.length; i++) {
      b[i] = modPrime(b[i]   (u*c[i]), prime);
    }
  }
}
```

Figure 4.4: NTRUEncrypt.js Polynomial Inverse Calculation Code Sample

Dissecting this first process took a significant amount of time. First, the value N, which is the size of the polynomial to be inverted, must

be known. Initialisation creates a counter value $k = 0$ and several polynomials: b, c, f, g. The initialisation of polynomial b begins with the first coefficient equal to 1 and the rest equal to 0. The polynomial f, not to be confused with the private polynomial f, is equal to the input polynomial with the coefficients modulo prime p. Finally, the N^{th} value of the polynomial g is set equal to 1 and the first value equal to the prime $p - 1$.

The degree of f is required, df is equal to the power of the largest coefficient value in polynomial f. Degree dg, which is the degree of polynomial g is always initialized to the size N. Now that initialization is complete, the computation of the inverse can begin; the algorithm can be broken down into four main components.

The first is a loop that performs many divisions of polynomial f by x, a simple shift left operation; and a multiply polynomial c by x operation, a simple shift right operation. For every iteration of this loop, k is incremented, and df is decremented. Once the condition of the first loop is false, a check if df is equal to 0 is completed, then the operation has finished. It will return the inverted polynomial if there is one.

The inverted polynomial check is performed first by taking the inverse modulo p of the first coefficient in polynomial f; if this is equal to 0, then the polynomial provided does not have an inverse. If the inverted polynomial check produces a value other than 0, a *shift* value is calculated where

$$shift = N - k$$

and then readjusted within the polynomial ring N. Finally, the inverse polynomial is calculated, using the *shift* value and setting the location coefficient value equal to the inverse modulo p of *f[0]* multiplied by the value in polynomial b at position i within the loop. The resulting returned polynomial is the calculated inverse of the input polynomial. If the value df is not yet equal to 0, then a check to see if df is less than dg is performed. If it is, then many swaps are performed. Polynomials f and g swap, polynomials b and c swap, and finally the values df and dg swap.

The final portion of the loop calculates a value u where

$$u = (f[0] * invModPrime[g[0]]) \, mod \, p$$

This value is then used for all values of the polynomials f and b to modify the coefficients and maintain them within the polynomial ring

p. The loop continues until the condition of $df = 0$ is met. The difficulty with this calculation is that it is intensive, and given the default approach a polynomial, may not have an inverse. Therefore private polynomial selection has to be repeated until a suitable polynomial is obtained.

```
function calculatePowerPolynomialInverse(polyA, invModArray)
{
  consoleMessage("Computing Polynomial Inverse fq...");
  var tmpq = p 1;
  var b = calculatePolynomialInverse(polyA, 2, invModArray);
  while(tmpq < q) {
    tmpq *= tmpq;
    var c = modConvolution(f, b, tmpq);
    c[0] = 2   c[0];
    if(c[0] < 0)
    {
      c[0] += tmpq;
    }
    for(i = 1; i < b.length; i++) {
      c[i] = (tmpq   c[i]);
    }
    b = modConvolution(b, c, tmpq);
  }
  for(i = 0; i < b.length; i++)
        {
            b[i] = b[i] % q;
        }
  return b;
}
```

Figure 4.5: NTRUEncrypt.js Polynomial Inverse Power of Two Calculation Code Sample

The process of calculating the inverse polynomial power of two is quicker, as the only requirement is to perform the first inverse calculation for f_p, that is private polynomial inverse modulo p. Having already calculated f_p, processing is halved by performing another series of polynomial convolutions and shifts. The output of this function, code illustrated in figure 4.5, provides f_q.

A technical limitation outside the scope of this research is JavaScript's inability to provide a way to access a CSPRNG. Although APIs have become available, there are many limitations as well as the considerable debate as to whether these are PRNG or CSPRNG. This research utilised the JavaScript API window.crypto.getRandomValues for number generation; production use should replace this.

During encryption and decryption the message polynomial m is represented in binary (0, 1), while polynomials used throughout are in ternary encoding when $p = 3$ (-1, 0, 1). The encryption result requires particular care as the result must have positive coefficients while decryption reveals the message in binary format. Polynomial convolution in this implementation is performed in ternary encoding, thus producing coefficients that are negative. The negative coefficient outputs then require conversion. An example, in figure 4.6, further illustrates the process of binary to ternary conversion during encryption convolution and addition operations.

q	32
Polynomial h	[7, 26, 1, 24, 28, 23, 19]
Polynomial r	[1, −1, 0, 0, 0, 0, 0]
Polynomial m	[0, 1, 1, 0, 0, 0, 0]
Polynomial $r \cdot h$	[−12, 19, −25, 23, 4, −5, −4]
Polynomial $e = (r \cdot h) + m$	[−12, 20, −24, 23, 4, −5, −4]
Binary Conversion $e = e\, positiveModulo\, q$	[20, 20, 8, 23, 4, 27, 28]

Figure 4.6: Example of Ternary to Binary Polynomial Conversion

Implementation of suggested enhancements into NTRUEncrypt.js was not performed. Due to the complexity of the algorithm and what could be accelerated using a GPU many of these improvements were omitted. Including them would improve the efficiency further, however require extensive changes to the source code.

4.5 NTRUEncrypt-GPU.js

The development of NTRUEncrypt-GPU.js provided many complications. Cryptography suits a computational approach such as OpenCL or CUDA. However, only OpenGL ES through WebGL was available. As there was no JavaScript version of NTRUEncrypt or GPU implementation, the development was built entirely for this thesis.

The HTML5 Canvas object is used to draw graphic elements on a web page through the use of JavaScript. The library Three.js allows access to WebGL through a Canvas object that can utilise all the functionality of WebGL 1.0 as well as providing OpenGL ES 2.0 support. On some browsers, experimental support is available for WebGL 2.0, and this provides support for OpenGL ES 3.0+ support. However, this was not available at the time of research.

A Vertex shader is required by OpenGL; a simple two-dimensional plane created to process the data required. While using a three-dimensional object would initially appear to provide a greater surface in which to process data, OpenGL only processes viewable vertices and thus hiding some data when rendered.

NTRUEncrypt-GPU.js requires two fragment shaders for encryption: polynomial addition and polynomial convolution. These shaders operate at a per pixel level, and each pixel is made up of four components: Red, Green, Blue, and Alpha. This is due to fragment shaders design for computer graphics rather than computation. However taking advantage of these components, each message to be encrypted will operate within its channel. Therefore a key of size 167 will operate on 167 pixels and be able to encrypt four messages of size 167 at a time. The fragment shaders operate in a two shader pass, with the first polynomial multiplication fragment shader outputting its texture as the input to the polynomial addition fragment shader.

Setting up Three.js, illustrated in figure 4.7, follows most of the same flow as Woo et al. (1999) illustrated in figure 2.15. Since the design intention of the library is for graphics output, a scene and camera object is required. A scene is what defines where and what will be rendered, and can contain objects, lights and cameras. The camera object provides the viewport through which a scene can be viewed, and therefore what will render. A renderer object is created to facilitate the input of objects to render in a scene. With the Three.js library there are two types: CanvasRenderer and WebGLRenderer. The WebGLRenderer is the object required for this research as it utilises WebGL to render the scene, thus using the GPU of the underlying system. However the CanvasRenderer is used as a fallback by the Three.js library. This renderer does not support the operations required by this research and would not provide the level of acceleration required.

Uniforms are global variables defined within each shader that allow the passing of data from the instantiating programming language to the shader itself. These uniforms are limited to integers, floats, vectors, colours, matrices, textures, and arrays of each of these types. For accuracy to be maintained, as well as to prevent value clamping, the float type uniforms were used extensively throughout development. A JSON variable defines uniforms, where the name and type of the uniform match the uniform name and type inside the shader.

Finally, a material object is created using JSON which holds the uniforms and the appropriate vertex and fragment shader. This

material is then combined with a geometry object into a mesh object and finally added to the scene for later rendering. BufferGeometry is used to create custom attributes for geometry utilised by the vertex shader. The BufferGeometry object provides the vertices of the object that will later render within the scene.

```
myUniforms1 = {
  publicKeyH: { type: "fv1", value: publicKeyHFA },
  randomPolynomialR: { type: "fv1", value: randomPolynomialRFA }
};

var material_FS = new THREE.ShaderMaterial({
  uniforms: myUniforms1,
  vertexShader: document.getElementById('vertexShader').
     textContent,
  fragmentShader: document.getElementById('
     fragmentShaderEncryptionConvolution'+N).textContent
});

var mesh = new THREE.Mesh(geometry(), material_FS);
scene.add(mesh);

var convolutionTexture = new THREE.WebGLRenderTarget( width,
   height, renderTargetParams() );

var startTiming = performance.now();
renderer.render(scene, camera, convolutionTexture, true);
var currentTiming = performance.now();
var output = "> #### GPU Polynomial Convolution Encryption took
   " + (currentTiming   startTiming) + "ms";
outputDiv.innerHTML = outputDiv.innerHTML + '<p>' + output + '</
   p>';
```

Figure 4.7: Sample Code of Three.js Setup

Rendering the scene requires a target. By default, this is the canvas created on the current web page. However this does not allow for multi-shader rendering, and the output of the canvas in the browser returns different data in different browsers. An output texture becomes the WebGLRenderTarget, and this allows rendering to a texture not displayed on screen. This WebGLRenderTarget texture can then input into the next shader that requires its data.

The encryption process consisted of two fragment shaders for the two major operations required by NTRUEncrypt encryption. These fragment shaders are required to perform a convolution between the public key and a blinding value, then the result of this polynomial convolution is added to the message polynomial.

```
<script id="fragmentShaderEncryptionConvolution11" type="x
    shader/x fragment">
     uniform float publicKeyH[11];
     uniform float randomPolynomialR[11];
     void main() {
       vec2 stCoord = vec2(gl_FragCoord.x,gl_FragCoord.y);
       int coordInt = int(floor(stCoord.x));
       for(int i = 0; i < 11; i++) {
         for(int j = 0; j < 11; j++) {
           if(int(floor(mod(float(i+j),float(11)))) == coordInt)
               {
             gl_FragColor.r += publicKeyH[i] * randomPolynomialR[
                 j];
           }
         }
       }
     }
</script>
```

Figure 4.8: NTRUEncrypt Encryption Polynomial Convolution Fragment Shader

The polynomial convolution fragment shader, illustrated in figure 4.8, takes two float arrays as input: public key h and the blinding value r. As shown in figure 4.2, the output is position dependent on the two polynomial inputs. Therefore, it initially obtains the x position of the required shader fragment pixel. Finally, a polynomial convolution is performed by looping through both float arrays and multiplying values together, to end in its output position.

```
<script id="fragmentShaderEncryptionAddition11" type="x shader/x
    fragment">
    uniform sampler2D sTexture;
    uniform float message[11];
    float modulus(float value, float modulus) {
      float outputValue = mod(floor(value), floor(modulus));
      if(outputValue < 0.0) {
        outputValue += modulus;
      }
      return floor(outputValue);
    }
    void main() {
      vec2 stCoord = vec2(gl_FragCoord.x,gl_FragCoord.y);
      int coordInt = int(floor(stCoord.x));
      vec4 inputTextureAtThisPixel = texture2D(sTexture,vec2(
          gl_FragCoord.x/11.0,0.5));
      gl_FragColor.r = inputTextureAtThisPixel.r;
      for(int i = 0; i < 11; i++) {
        if(coordInt == i) {
          gl_FragColor.g += float(int(floor(modulus(float(
              inputTextureAtThisPixel.r + message[i]),float(32)))
              ));
        }
      }
    }
</script>
```

Figure 4.9: NTRUEncrypt Encryption Polynomial Addition Fragment Shader

The polynomial addition fragment shader, illustrated in figure 4.9, takes two inputs; the first being the texture output of the polynomial multiplication fragment shader, and the second the float array of the message to be encrypted. Each fragment shader only outputs to the position corresponding to the current fragment shader x position. This output limitation requires a single for loop and an if check, however, it can output different encrypted messages to each component.

```
<script id="fragmentShaderDecryptionConvolution11" type="x
    shader/x fragment">
     uniform float cipherText[11];
     uniform float privateF[11];
     void main() {
       vec2 stCoord = vec2(gl_FragCoord.x,gl_FragCoord.y);
       int coordInt = int(floor(stCoord.x));
       for(int i = 0; i < 11; i++) {
         for(int j = 0; j < 11; j++) {
           if(int(floor(mod(float(i+j),float(11)))) == coordInt)
              {
             gl_FragColor.r += cipherText[i] * privateF[j];
           }
         }
       }
     }
</script>
```

Figure 4.10: NTRUEncrypt Decryption Polynomial Convolution Fragment Shader 1

The decryption process has slightly increased complexity, compared to the encryption process. The decryption process has three major operations; however only two of these were implemented using the GPU and fragment shaders. The first polynomial convolution, illustrated in figure 4.10, is performed with the recipient's private polynomial f and the ciphertext e within modulo q. The conversion of the output of the first convolution, from ternary to binary representation, is performed within modulus p. This conversion is performed faster in JavaScript on the CPU compared to the GPU. Now that the ciphertext is in the correct representation modulo p the final polynomial convolution, illustrated in figure 4.11, is handled by a second fragment shader.

```
<script id="fragmentShaderDecryptionFinalConvolution11" type="x
    shader/x fragment">
     uniform float fp[11];
     uniform float b[11];
     float modulus(float value, float modulus) {
       float outputValue = mod(floor(value), floor(modulus));
       if(outputValue < 0.0) {
         outputValue += modulus;
       }
       return floor(outputValue);
     }
     void main() {
       vec2 stCoord = vec2(gl_FragCoord.x,gl_FragCoord.y);
       int coordInt = int(floor(stCoord.x));
       for(int i = 0; i < 11; i++) {
         for(int j = 0; j < 11; j++) {
           if(int(floor(mod(float(i+j),float(11)))) == coordInt)
              {
             gl_FragColor.r += fp[i] * b[j];
             gl_FragColor.g += fp[i];
             gl_FragColor.b += b[j];
           }
         }
       }
     }
</script>
```

Figure 4.11: NTRUEncrypt Decryption Polynomial Convolution Fragment Shader 2

Fragment shader inputs are called uniforms; these are objects defined by the system available OpenGL standard. Due to the usage of OpenGL for graphics, uniforms are significantly limited to the type of objects they can be defined as. Not only is this limitation on objects taken further with mobile optimised OpenGL ES, some implementations on hardware platforms will have further limitations while stating support for an OpenGL ES version. The fragment shaders developed for this research demonstrate this limitation, as the shaders operate using as a float type. While the OpenGL ES 1.0 standard supports float type shaders, many implementations will either not compile at run time or compile although not run.

OpenGL ES 1.0 comes with many other limitations. The first, as outlined by the fragment shaders, is that each fragment shader is only able to access the output of its current pixel position. This output limitation means that the fragment shader at pixel position 5, is only able to edit the output at pixel position 5. Even when hard-coding access to a pixel position it is unable to edit or add to the

output of another fragment shader. Operationally this is evident, as the advantage of a GPU is its ability to process in parallel, and each pixel has a fragment shader that is operating in parallel. Therefore it would not make sense to allow modification of the output of another fragment shader without a form of synchronisation between each. Another limitation, discovered late in development, is that there is a maximum number of uniforms defined by the graphics hardware and drivers per system. This restriction, is due to hardware limitations placed by GPU manufacturers, and the way they process and parallelise the fragment shaders passed to the GPU. This constraint places hard limits on the size of the polynomials that can be used by NTRUEncrypt-GPU.js. This limitation was a significant problem for mobile devices, with the largest uniform maximum being 224 on the ASUS Zenfone 2. Furthermore, accuracy was a problem, as with graphics if a blue or red hue is slightly out, the eye may not even be able to perceive this; however if the byte output of encryption is even one bit out it will never decrypt. Mathematically the polynomial coefficients operate within whole numbers, however inside the fragment shaders, they operate in floating point. This operational difference means that a large amount of floating point to integer and integer to floating point conversion is required. This is illustrated in figure 4.8, the line

$$if(int(floor(mod(float(i+j), float(11)))) == coordInt)$$

where the variable *coordInt* integer is compared against the modulus of two variables i and j being looped through within the shader. The mod function requires two floating point inputs and produces a single floating point output. This output can have slight inaccuracy with trailing decimal numbers and thus the output needs to be floored and then converted for integer comparison. These issues are difficult to debug, leading to more floor, int, float conversions than is necessary.

5 Research Findings

5.1 Crypto++

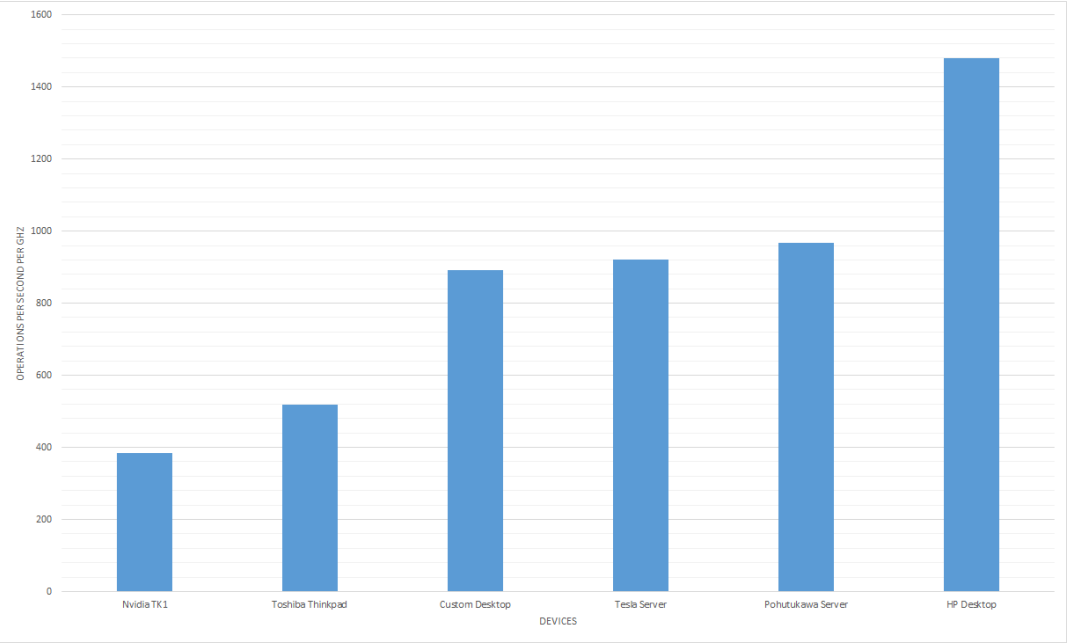

Figure 5.1: Crypto++ Total Operations per Second per CPU GHz

Table 5.1: Crypto++ Benchmark Hardware CPU Clock Speeds

Nvidia TK1	**Toshiba Thinkpad**	**Tesla Server**	**Pohutukawa Server**	**HP Desktop**	**Custom Desktop**
2.3 GHz	2.53 GHz	2.5 GHz	3.2 GHz	3.6 GHz	3.8 GHz

The Crypto++ benchmark initially appears to suit a single threaded CPU workload, and therefore would make a suggestion of an inverse linear correlation between clock speed and time taken for each algorithm. Examination of the hardware listed in section 3.6.2 referenced against figure 5.1 and the clock rates in table 5.1, results appear to support this relationship on every piece of hardware, other than the Custom Desktop.

Table 5.2: Crypto++ RSA Encryption and Decryption Results (Milliseconds per Operation)

Algorithm	HP Desktop	Tesla Server	Nvidia TK1	Toshiba Thinkpad	Pohutukawa Server	Custom Desktop
Encryption						
RSA 1024	0.01	0.02	0.1	0.03	0.02	0.02
RSA 2048	0.03	0.036	0.23	0.062	0.05	0.04
Decryption						
RSA 1024	0.29	0.32	2.584	0.58	0.41	0.324
RSA 2048	1.342	1.536	14.42	2.726	1.98	1.492

Based on the results of the Crypto++ RSA benchmarks in table 5.2, the HP Desktop takes the least amount of time per operation for both 1024-bit and 2048-bit encryption and decryption. Alternatively, the Nvidia TK1 was the slowest, averaging between 7.67 and 10.75 times slower than the HP Desktop depending on the operation. Overall, RSA 2048-bit encryption took 2.24 times longer per operation than 1024-bit RSA across all devices. While with decryption an even larger gap emerged with RSA 2048-bit decryption taking 5.212 times longer. This evidence reinforces the idea that doubling the bit security of RSA, increases the computation time with a closer exponential relationship.

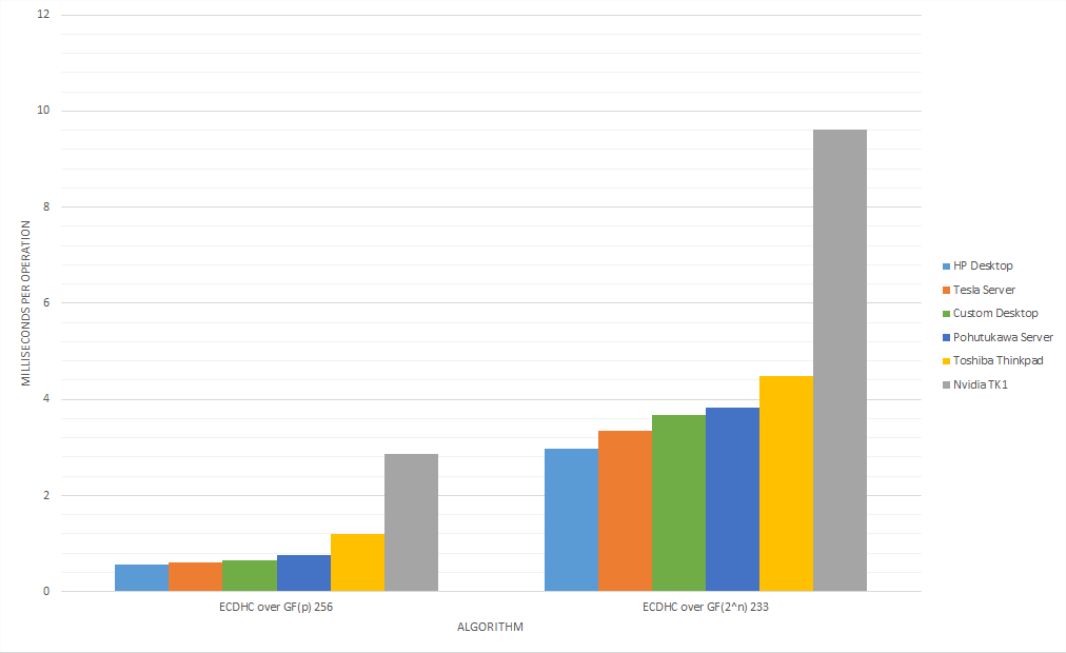

Figure 5.2: Crypto++ ECDHC Key-Pair Generation

Similar to the results for RSA, the ECDHC key-pair generation operated the quickest on the HP Desktop and slowest on the Nvidia TK1 as illustrated in figure 5.2.

Table 5.3: Crypto++ AES Results (MB/s)

Algorithm	HP Desktop	Tesla Server	Nvidia TK1	Toshiba Thinkpad	Pohutukawa Server	Custom Desktop
AES/GCM	2236.4	299	59.2	153.8	739.2	944.4
AES/CCM (128-bit key)	586.2	146	41	87.8	490.6	466.6
AES/EAX (128-bit key)	587.2	145	41	87	511	478.2

AES-NI made a significant difference when enabled on a system. As illustrated in table 5.3, where AES/GCM on the HP Desktop had both CPU support for and enabled AES-NI, making it almost 38 times faster than the Nvidia TK1. Once again, the Nvidia TK1 demonstrates the speed difference between running these algorithms on a mobile processor compared to a desktop or server variant. It is important to highlight AES-NI as it confirms the advantage of having a dedicated piece of hardware that can perform cryptographic operations. If mobile devices had an implemented AES-NI or cryptographic co-processor, some efficiency and power savings could be made. The drawback of dedicated hardware is that it is expensive, inflexible, and requires levels of expertise that may not be available within a company producing these devices. However, it is unclear from Crypto++ documentation what the MB/s represents, as AES performs both encryption and decryption yet only one result is given for each algorithm.

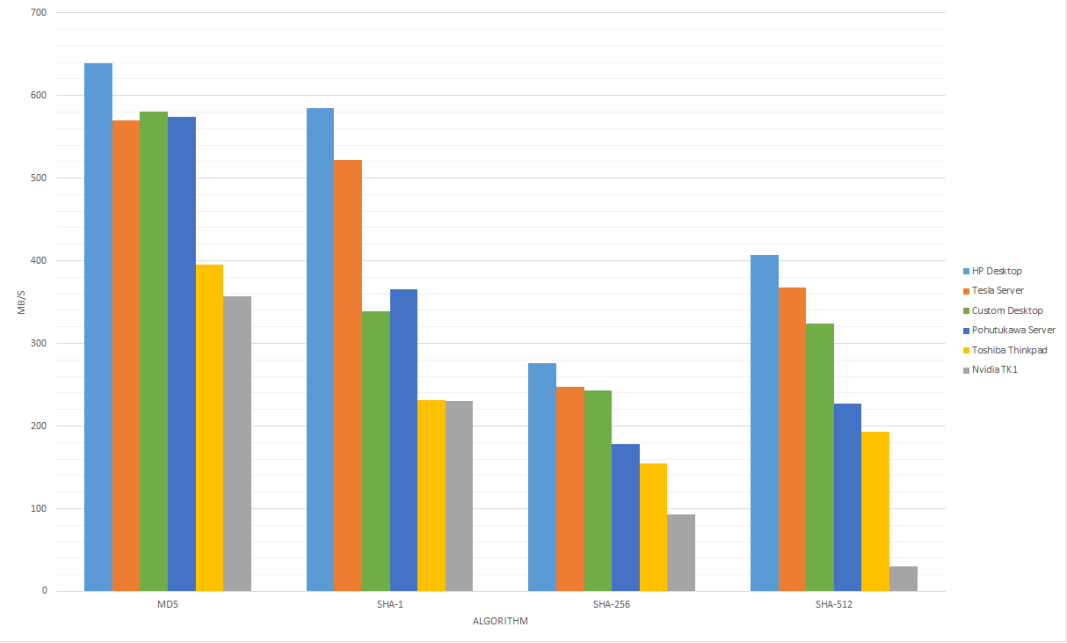

Figure 5.3: Crypto++ Hashing Algorithm Results

As illustrated in figure 5.3 including hashing algorithms in the experiment provides another comparison between algorithms on different systems. Comparing SHA-256 and SHA-512 benchmarks, the latter was quicker across all systems. SHA-512's quicker operation may be due to implementation within Crypto++, and it could be a higher level of operation or the processors tested had better efficiency with SHA-512 over SHA-256, all of which is outside the scope of this research.

5.2 Android and Java

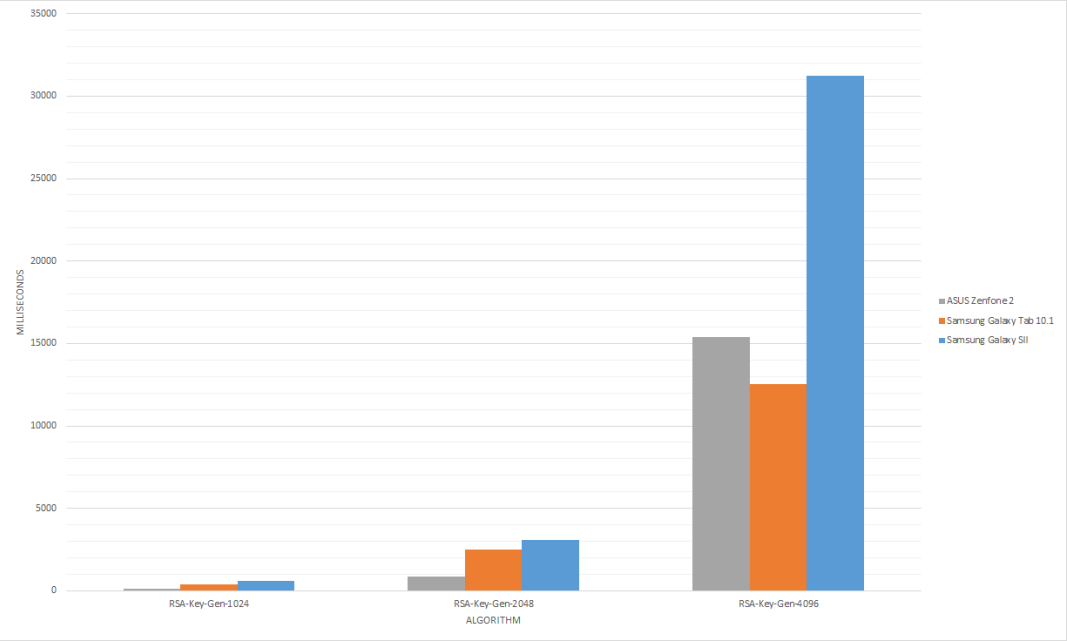

Figure 5.4: Android RSA Key Generation Results

The results from the Android mobile hardware, illustrated in figure 5.4, provide insight on how the chosen algorithms perform on the selected systems. What is prominent from this graph is the exponential nature of increasing RSA's bit size. As the bit size doubles the computation power required increases exponentially. This is demonstrated when comparing RSA key generation of 2048-bit keys to 4096-bit keys, as it takes more than ten times the amount of time across all tested systems.

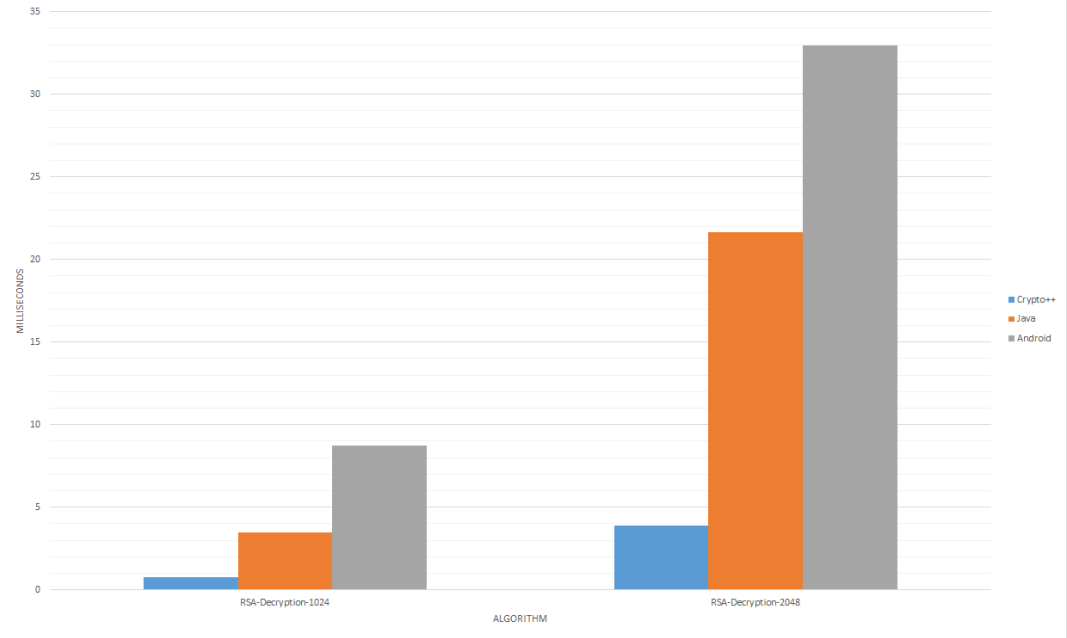

Figure 5.5: Crypto++, Java, and Android Comparison of RSA Decryption

Figure 5.5 illustrates some interesting points, comparing the average Crypto++ results with that of Java and Android. To begin with, this continues to illustrate the trend of exponential computation growth, doubling the RSA bit size, consequently more than doubles the processing power required. Additionally, the Java benchmark run on the desktop and server systems are not significantly faster than the Android version.

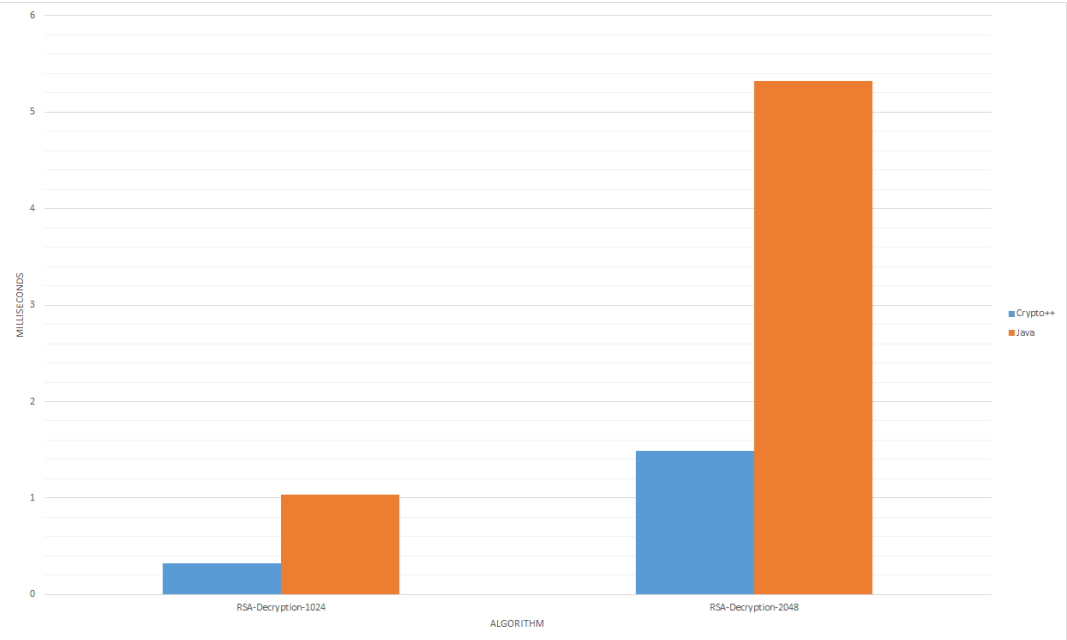

Figure 5.6: Comparison of RSA Decryption on Custom Desktop of Crypto++ and Java

However the Crypto++ benchmarks, written in C and C++, are significantly faster than the same Java implementations as illustrated in 5.6. Comparing the results from the Custom Desktop hardware, Crypto++ RSA Decryption was between 3.2 and 3.6 times faster than its equivalent in Java.

Java, both on the tested Android and Desktop platforms, was significantly slower than the C and C++ benchmarks in Crypto++ as discussed in section 5.1. However, Crypto++ had to be compiled individually on each system and could, therefore, receive optimisation for each specific hardware platform. This compilation for each platform makes it significantly more difficult for the average user to utilise on their system. The application would need to be downloaded, installed and set up before a user would be able to make use of this optimised functionality.

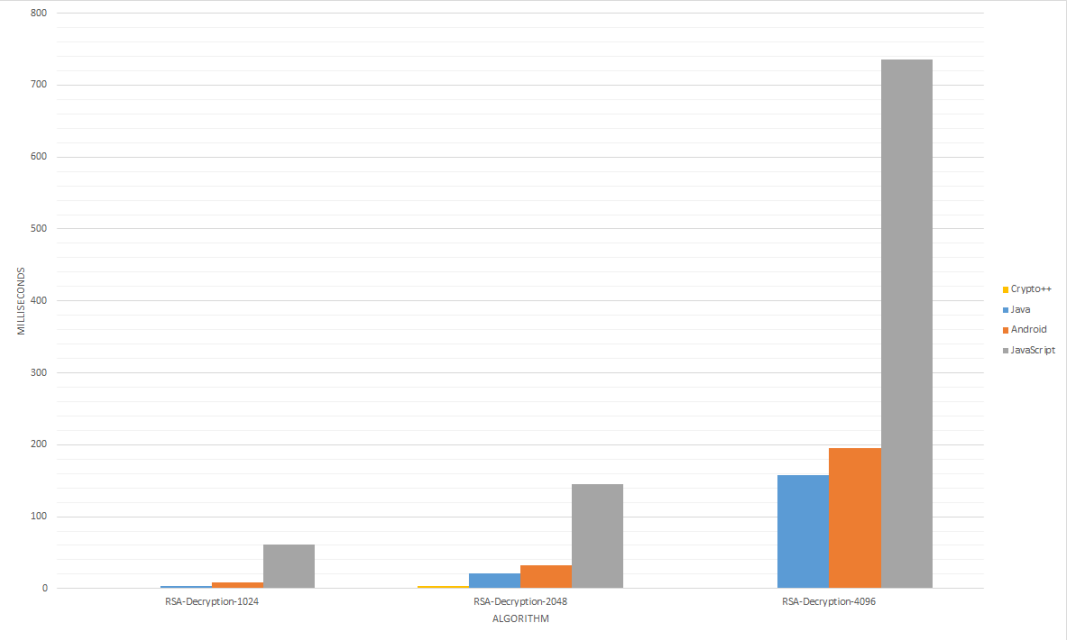

Figure 5.7: Difference between Programming Languages and RSA Decryption

Results from the experiments illustrate a strong reason for the adaptation of NTRUEncrypt. While the open source versions of NTRUEncrypt are slower overall compared to similar security of ECC, the computational difference is more linear. Referring to figure 5.7 about RSA decryption, there is an extremely positive exponential growth in computational requirement, as the number of bits increase. This exponential growth means that RSA has a greater efficiency trade-off for security. The worst case for RSA during this experiment, was a 20 time increase in milliseconds for every bit increase.

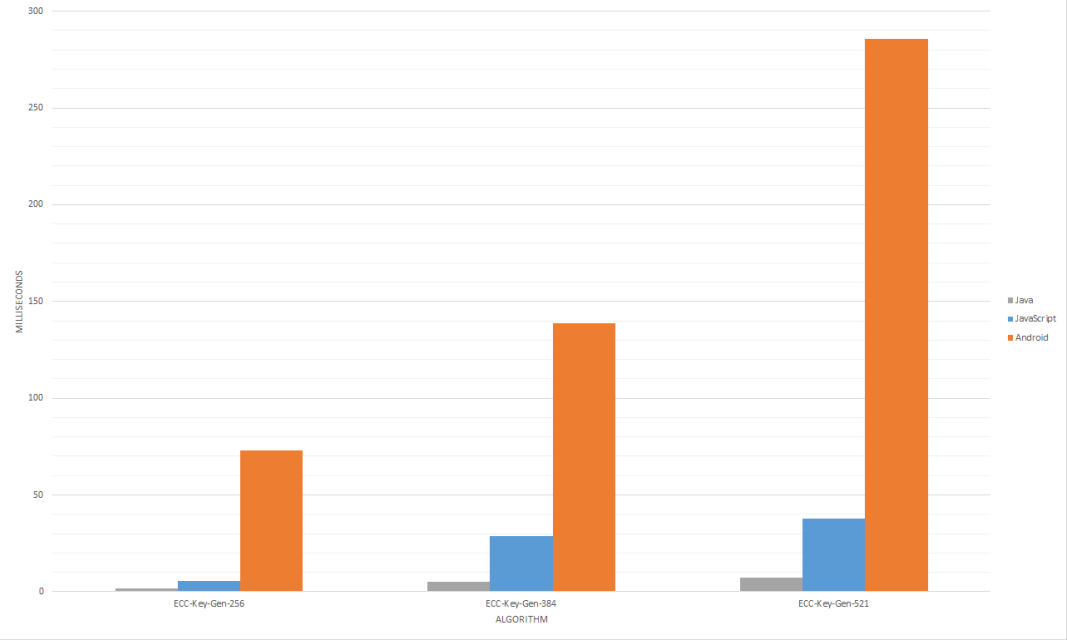

Figure 5.8: Comparison of Programming Language for Elliptical Curve Cryptography Key Generation

With ECC, as illustrated in figure 5.8, this trade-off is reduced greatly compared to RSA. However in these experiments, ECC still managed a four time increase in milliseconds, per bit increase. This security to processing ratio for ECC is an improvement over RSA, and all of the time results were well within ranges that would be considered suitable for mobile use.

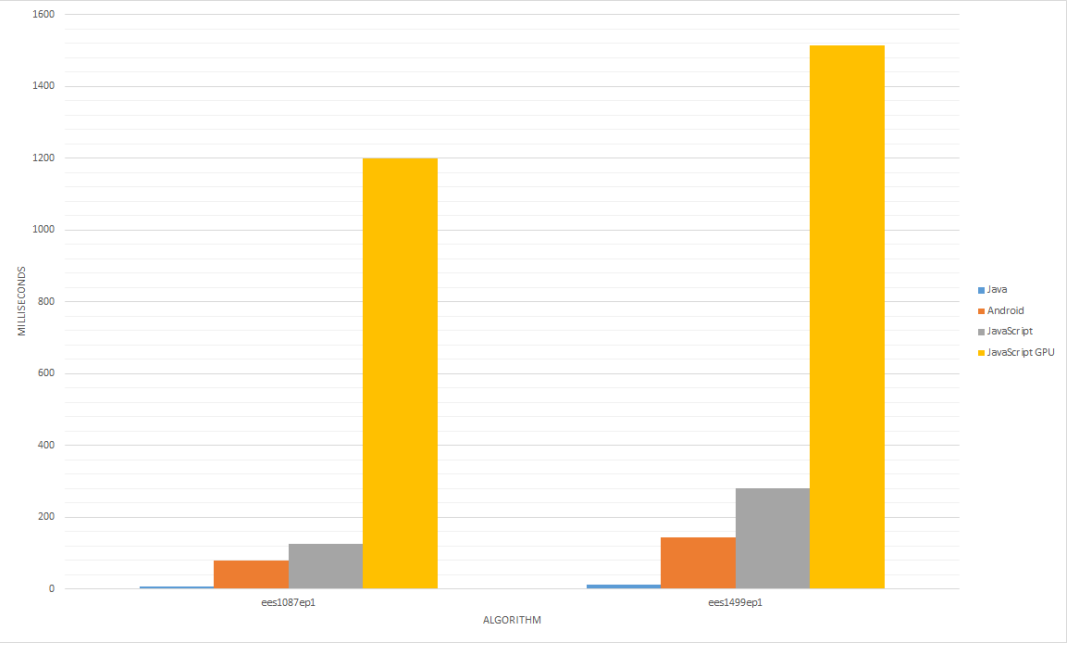

Figure 5.9: Comparison of NTRUEncrypt Encryption in Java, JavaScript, and JavaScript + GPU

However, as shown in figure 5.9 the increase in bit size for NTRUEncrypt has a more linear effect than ECC. NTRUEncrypt was calculated to have just under a two time increase in milliseconds, per bit increase. This time increase was relatively stable across all security values tested for NTRUEncrypt.

These results attempt to highlight the efficiency of NTRUEncrypt before algorithm acceleration. It is worth noting that both RSA and ECC did have lower computational differences at lower security levels. However, it is the exponential increase that makes them difficult to use at larger bit sizes. Due to this, algorithms that have more linear increases in computational requirement will better suit security applications. Moore's law requires an increase in the number of bits periodically to maintain security. When algorithm computation requirements are linear, this trade-off reduces.

5.3 JavaScript

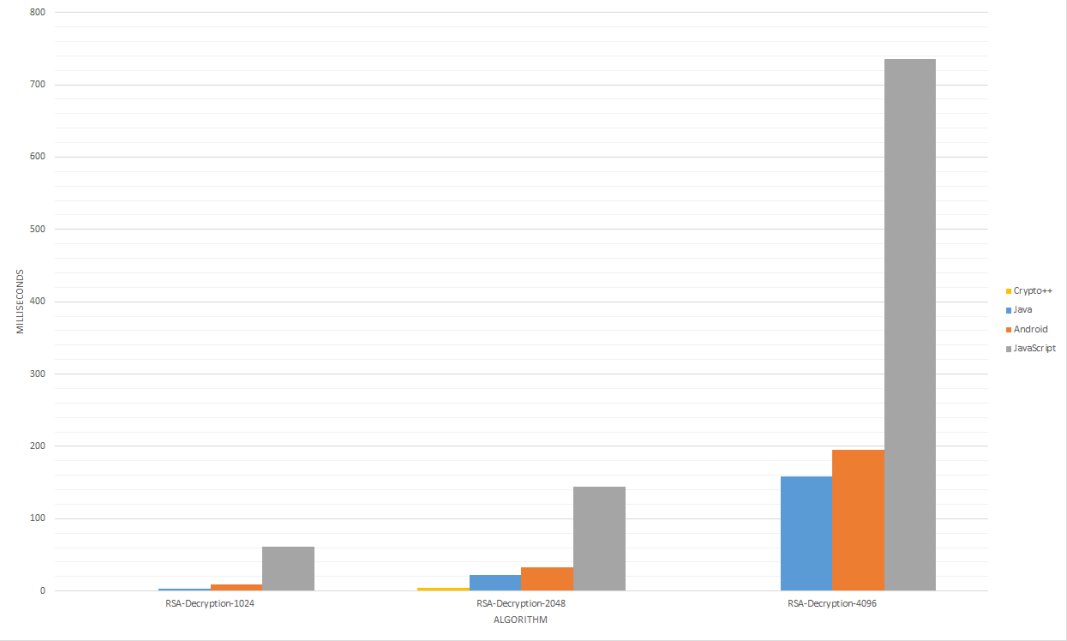

Figure 5.10: C/C++ vs Java vs Android vs JavaScript for RSA Decryption

As illustrated in figure 5.10 JavaScript was overall significantly slower than both Java and Crypto++.

The selected library ASMCrypto, while slower than WebCrypto, produced more reliable results. The average variance of ASMCrypto across RSA experiments was 9,256 milliseconds, while the same measurement for WebCrypto produced a variance of 33,641

milliseconds. Although considering the 33-34 second variance is significant, WebCrypto was up to 18 times faster with RSA key generation, encryption and decryption.

WebCrypto, specifically RSA-1024-bit, was extremely slow compared to the same machine running the WebCrypto RSA-2048-bit benchmark. The result irregularity may be due to benchmark implementation or an unidentified issue during the experiment. The results from WebCrypto RSA were removed from the final results for analysis, as these outlying results skewed data significantly.

While WebCrypto ECC provided the ability to generate ECC keys, they were significantly slower than operating through Java. Many systems did not support performing operations, other than key generation, which make WebCrypto ECC ineffective.

The major advantage of ASMCrypto over WebCrypto, was that once the library was loaded, every system had access to all algorithms that ASMCrypto could provide. Each different browser, listed in section 3.6.1, provided different compatibilities when using WebCrypto. Varying browser compatibility for WebCrypto meant there was no guarantee the end user would be able to operate a particular cryptographic algorithm, or even if they could provide the same expected output. Once the WebCrypto API becomes more mature and browsers provide compatibility for all, or most algorithms, then the performance gain over ASMCrypto will be significant enough that it will easily be replaced.

Another difficulty that WebCrypto does not address are browser timeouts. ASMCrypto requires operation within JavaScript Promises; this means that attempting to generate an RSA key effectively operates on another thread, and does not prevent other JavaScript execution. The default behaviour for browsers appeared to be if a script held the main thread up for more than 10 seconds, it was terminated by the browser. While an appropriate safety mechanism for users browsing the web, this proved to be a problem for testing cryptography in a browser. Many operations, especially on lower-powered devices, would take longer than the time-out period. WebCrypto did not have built-in JavaScript Promises, and therefore these had to be added to make sure time-outs did not occur.

5.4 NTRUEncrypt.js

The JavaScript implementation of NTRUEncrypt was between 12 and 25 times slower than the Java implementation. It was, however, 10 milliseconds faster than the Android implementation with ees1087ep1 decryption, and only between 1.6 to 1.9 times slower on other operations.

NTRUEncrypt.js used only the base algorithm and did not include any of the enhancements suggested by the technical documents provided by Security Innovations Inc. Given these optimisations, it could have been as fast as the Java implementations.

Due to its lack of enhancements, NTRUEncrypt.js suffered from a higher failure rate. During key generation a randomly selected polynomial f meant there may not be an inverse; thus selection and computation of a possible inverse would repeat.

Converting all messages from binary and ternary for encryption and back for decryption is required. To avoid message conversion, the value of p can also be a polynomial. There is no requirement for p to be an integer as long as it remains relatively prime to q. Explained further in the Security Innovation NTRU Enhancements 1, when $p = 3$ the message produced is ternary with coefficients [1, 0, -1]. However if $p = 2 + x$ then message m will remain binary as coefficients will only be [1, 0]. This enhancement does make decryption more complex and, due to time constraints, was not implemented in the version used for testing.

Convolution made up the majority of the encryption process time; between 43% and 93% of total processing time. The reason for the convolution taking up the majority of the processing time is due to it performing $N \cdot N$ multiplication and $N \cdot N$ addition operations. Performing the convolution operation is discussed in section 4.4, and illustrated in figure 4.2.

The addition only makes up a small portion of processing time in this JavaScript implementation, between 0.8% and 20%, as the array data structure outlined in figure 4.2 allows for a simple single iteration addition.

The development of NTRUEncrypt.js is intended for execution in a browser. Some of the systems listed in the hardware section 3.6.2, are server operating systems, meaning only console access was available. There are some implementations of command line JavaScript execution engines. However, these were not explored in this research.

Key generation of NTRUEncrypt.js ees1499ep1 took on average 4,739 milliseconds to generate compared to WebCrypto ECC P-521, which took on average 38 milliseconds. Comparing WebCrypto ECC to NTRUEncrypt at equivalent security for key generation, ECC is between 86 and 125 times faster. However, as noted in the JavaScript section 5.3, many systems were unable to perform any other operations with the ECC key generated after this point.

Table 5.4: JavaScript Comparison of RSA 1024-bit and NTRUEncrypt ees401ep1

Algorithm	**Processing Time (Milliseconds)**
RSA-Key-Gen-1024	1487.43
ees449ep1-Key-Gen	146.77
RSA-Encryption-1024	15.29
ees449ep1-Encryption	13.11
RSA-Decryption-1024	60.81
ees449ep1-Decryption	25.60

Comparing comparable security between RSA 1024-bit and NTRUEncrypt.js, as illustrated in table 5.4, NTRUEncrypt.js is just over ten times faster for key generation. However, the NTRUEncrypt.js encryption operation is not considerably faster than RSA. The decryption operation of NTRUEncrypt.js is more than two times faster than RSA. This difference increases as equivalent security increases, proving that NTRUEncrypt is more efficient for asymmetric encryption than RSA.

5.5 NTRUEncrypt-GPU.js

Figure 5.9, highlights the speed difference between Java, Java on Android, JavaScript, and JavaScript GPU. Results from the experiment showed NTRUEncrypt.js was slower than both Java and Android NTRUEncrypt benchmarks. The GPU accelerated NTRUEncrypt-GPU.js was between 4 and 38 times slower than NTRUEncrypt.js. NTRUEncrypt-GPU.js reduces this speed difference as the security increases. If it was possible to experiment with larger values of N, then NTRUEncrypt-GPU.js may prove to be as fast or faster than NTRUEncrypt.js.

Many of the systems were unable to run the GPU accelerated NTRUEncrypt.js, and those that could have further limitations on them, such as those outlined in the implementation section 4.5. These restrictions meant from all the hardware systems, listed in section

3.6.2; only the HP Elite Desktop system was able to provide useful benchmarks across all tests.

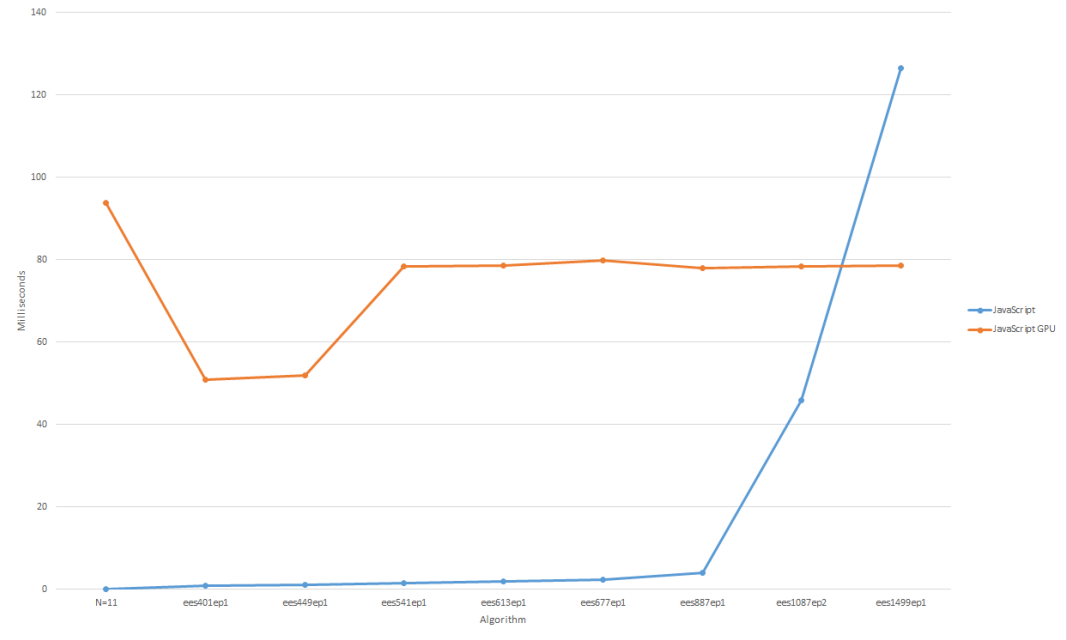

Figure 5.11: Comparison of NTRUEncrypt.js and NTRUEncrypt-GPU.js Convolution Operation on the HP Elite Desktop

However the HP Elite Desktop system, which was able to run the entire experiment, displayed promising results in the NTRUEncrypt encryption convolution operation. The convolution operation suits a GPU as they are effective at vector and multiplication mathematics. As illustrated in figure 5.11 the GPU reaches its maximum processing time at around 80 milliseconds. While the CPU implementation is significantly faster up until the ess1499ep1 convolution, where the GPU convolution is almost twice as fast. The ess1499ep1 convolution results provide positive evidence to suggest, even at higher security levels, or increased message throughput, GPU convolution would be considerably faster and therefore more efficient.

However, this single positive result was still affected by the limitation that the setup time for Three.js, and the data required for encryption, took significantly more time than the encryption operation performed on the CPU. As illustrated in figure 5.11 the total encryption time for the GPU was more than 9.5 times slower.

Comparing the results to Harrison & Waldron, 's (2009) research utilising CUDA, the RSA decryption is similar to that of the convolution results seen in figure 5.11. As described in section 2.1.5, RSA decryption involves large exponent mathematics within a

modulus, which can be simplified and accelerated using binary modular exponentiation; all of which is mathematically similar to what happens in a polynomial ring convolution.

The results from this research had problems similar to those faced by Nicholls (2012). Although many libraries are available that streamline the development process of WebGL, the actual usefulness of such libraries is still extremely limited. The advantage gained by utilising the GPU for processing is negated by the added latency and requirement to conform within the graphics processing pipeline. Similarly, the issues highlighted by Yang & Goodman (2007) and the floating point uniform limits remain the single largest limitation of running on mobile systems to this approach.

6 Discussion

6.1 Research Questions

Referring to section 3.3 on page 44 which this research aims to answer.

Question 1

Can in-browser cryptography be accelerated using a GPU? As discussed in section 2.5 on page 40, a number of researchers such as: Cook & Keromytis, Yang & Goodman, and Harrison & Waldron have previously investigated the use of a GPU to successfully accelerate a cryptographic algorithm. However, the results from this research demonstrate that acceleration was not achieved. Current software and hardware support did not allow for the proper acceleration of a cryptographic algorithm. Given better access and dedicated GPGPU computing such as Compute Shaders in WebGL 2.0 there is still a possibility of this being usable. Experimentation with another algorithm may still be able to take advantage of WebGL.

Question 2

Does increasing the bit size of a cryptographic algorithm have a linear positive correlation to processing time and security? Results from this research demonstrate ECC had a slightly negative linear relationship. This is an interesting result as the difference between curve 256 and 384 is approximately 2.15 times speed, while the difference between curve 384 and 521 is 1.91 times, this gives ECC a ratio of 4.11.

For example if curve 256 takes 26 milliseconds to generate, using this ratio if we double the bit security and generate curve 521 it will take approximately $26 \cdot 4.11 = 106.86$. NTRUEncrypt displayed a somewhat linear positive correlation between bit size and processing time.

In this research NTRUEncrypt had a ratio of 5.53 for bits of security, this means that if an ees887ep1 key generation takes 526 milliseconds, then doubling the security and performing ees1499ep1 key generation would take approximately $526 \cdot 5.53 = 2,908.78$ milliseconds.

The relationship for RSA in the results obtained during the experiment, demonstrated a positive exponential cubed relationship. This is due to the difference between generating 1024 to 2048 bit, and 2048 to 4096

bit keys, which the ratio calculated to approximately 5.69 and 8.05 respectively. This clearly demonstrates an almost three times increase for each bit level of security, and this resulted in a large ratio of 45.81; which unlike NTRUEncrypt and ECC was not actually a complete doubling ratio either. This means a system that generates a 1024 bit key in 674 milliseconds would take approximately $674 \cdot 45.81 = 30,875.94$ milliseconds. Given that 30,875 millisecond key generation is the same 128 bit security level as ECC curve 256, at 26.76 milliseconds and NTRUEncrypt ees887ep1, at 525.97 milliseconds RSA is certainly not the choice for efficient security performance.

Cryptographers and mathematicians constantly discuss the various changing security levels provided by cryptographic algorithms. However, many others rely on the standard comparable strengths provided by NIST Special Publication 800-57 Barker (2016). As discussed in the hypothesis, the doubling of an algorithm's key size does not directly equate to a doubling of security for all algorithms tested in this research.

Question 3

Does the programming language that a cryptographic algorithm is implemented in have an affect on processing time? As shown in figure 5.7 Crypto++, which is implemented in C, is significantly faster than Java. Java is considerably faster than JavaScript.

Question 4

Does measuring the processing time of cryptographic algorithms provide an accurate means of comparison? Provided cryptographic algorithms are measured several times with variance recorded then processing time provides an accurate means of comparison. A number of tests in this experiment displayed significant minimum and maximum times that would not be a fair representation of the average run time.

Question 5

Is a GPU faster at performing public key encryption than a CPU implementation of the same algorithm? For NTRUEncrypt-GPU.js only, the encryption convolution using

ess1499ep1 displayed an improvement, however only the HP Elite Desktop system was able to execute it. For hardware defined as mobile only the Nvidia TK1 was only able to run the lowest test parameters of NTRUEncrypt-GPU.js. This demonstrated a significant overhead when attempting to use the GPU.

Question 6

Is a GPU faster at performing private key decryption than a CPU implementation of the same algorithm? Since the process of decryption involves two convolutions, comparisons of the convolutions would support the statement. However the total time including setup means that in the implementation of NTRUEncrypt-GPU.js it was not faster than the CPU based version. For hardware defined as mobile only the Nvidia TK1 was able to run the lowest test parameters of NTRUENcrypt-GPU.js. This demonstrated a significant overhead when attempting to use the GPU.

Question 7

What is the impact of latency on a cryptographic algorithm? The majority of the total time spent utilising the GPU is taken up preparing, moving, and retrieving the data. Data transfer latency is why GPUs are better suited to large amounts of data, to make use of the processing power they possess and minimise the transfer bottleneck. In the instance of CPU computation, since it is happening directly within system memory, the bus is fast enough to make the bottleneck the processor speed.

Question 8

Does a GPU implementation of a cryptographic algorithm increase throughput compared to a CPU implementation? As illustrated in figure 5.11 on page 81, at higher security bit sizes the answer is yes. If multiple messages were to be encrypted or decrypted, specifically the convolution portion of the processing of NTRUEncrypt-GPU.js, this would make it preferable to a simple CPU implementation. However compatibility was still an issue on the desktop systems and a fall back to CPU would be required in any production deployment.

Unfortunately this question cannot be answered for mobile devices based on the results of this experiment, see section 7.2 on page 88

for research limitations. Given the possibility of NTRUEncrypt-GPU.js working then it may have demonstrated an increase in throughput compared to a CPU implementation as the CPU on all of the mobile devices tested were comparatively slow.

7 Conclusions

In this instance, the Three.js library was unable to successfully accelerate NTRUEncrypt using the GPU. As stated in the discussion, section 5.5 on page 80, NTRUEncrypt-GPU.js was up to 38 times slower than NTRUEncrypt.js. However, the convolutions performed by NTRUEncrypt-GPU.js were 1.6 times as fast as NTRUEncrypt.js at the highest available security. Furthermore, comparisons within this research showed JavaScript was up to 80 times slower than C, C++, and Java. As discussed in section 2.4.5 on page 36, the problems faced by previous research have been due to the limitations on portability for code written in C and C++. Despite this research's attempt to address these problems the results show NTRUEncrypt-GPU.js had a similar problem as it was limited to only one of nine systems. This results were due to a hardware limit on uniform buffers, a factor that will not change with new libraries, drivers, or operating systems. Several recommendations can be made from these results. Firstly, efficiency improvements to JavaScript engines. The results of this experiment show JavaScript was up to 80 times slower than C, C++, and Java; indicating JavaScript is not a suitable programming language for high performance computing. Secondly, as an extension of more efficient JavaScript engines, developing engines that are truly multi-threaded. As discussed in section 4.3 on page 56, JavaScript must emulate threads using Promises or Web Workers. All systems listed in section 3.6.2 on page 50 had at least two cores, given JavaScript is currently unable to take advantage of multiple cores, it could see considerable performance increases if JavaScript could utilise these additional processing cores. Thirdly, waiting for the implementation of WebGL 2.1 standard in modern browsers allows the use of Compute Shaders. As discussed in the Future Research section 7.3.1 on page 89, Compute Shaders are designed to offer compute capability within OpenGL, without the restrictions of the standard graphics pipeline. From the experiment results, NTRUEncrypt should be considered a suitable replacement for RSA. While NTRUEncrypt was not as quick as ECC, the ability to perform encryption and decryption operations make this a better alternative when a single asymmetric algorithm is required. NTRUEncrypt is also currently considered quantum secure and based on the efficiency trends seen in both this research experiment and others, adds further validation to NTRUEncrypt's suitability for secure cryptographic systems.

7.1 Implications

Given this research focuses on in-browser accelerated cryptography, the concept is to create easier to use platforms, which allow for end-to-end encryption. Many online services attempt to provide security by assuring users that they data they send to others online is only available by those the user has allowed. End-to-end encryption involves performing cryptography between clients, rather than between the client and the server. Thus if the service provider is unable to decipher communication through their system, then no form of interception, be it legal or illegal, can occur without first compromising both end users. Referring to the aims of this research in section 3.1 on page 43, it has added to the overall knowledge of cryptography and information security. The research experiment has provided useful outputs for future researchers of accelerated cryptography. The development of a JavaScript version of NTRUEncrypt will also be of use to researchers, developers, and others outside of academia. Although this research and experiment did not produce an accelerated version for mobile or low power systems, looking at future research and the positive result of the encryption convolution, this would be possible to pursue for future research. Another aim of this investigation was to show that cryptography and information security does not have to be slow. Cryptography is often viewed as slowing a process down by adding complexity. With more research into the area of performance of cryptography, researchers can provide efficient, secure platforms. Ultimately the goal of this research is to advance end-to-end encryption options that allow experts to concentrate on providing secure platforms that enable everyday users to remain protected without knowledge of how cryptography or information security works. NTRUEncrypt has been proven to be effective and while JavaScript was slow, considering that it can be executed on any system with a modern browser it provides a platform to allow easy to use end-to-end encryption.

7.2 Research Limitations

7.2.1 Available Hardware

Although there are many different hardware platforms described in section 3.6.2 on page 50, having more platforms may have provided more data, specifically another system that was compatible across all tests. It appears from these results that both Linux and another AMD

Graphics Card would be required to enable more platforms to test against the HP Elite Desktop.

7.2.2 Development Time

As with all Masters theses, the time is constrained. Additionally, when developing a piece of software, development needs to be constantly maintained throughout the lifetime of the project. Due to this time constraint, many of the optimisations to both the algorithm and the software itself could not be completed. The software will be made open-source to allow others to contribute as NTRUEncrypt.js could be useful for other projects and research.

7.2.3 WebGL and Browser Standards

Many of the problems faced with NTRUEncrypt-GPU.js are due to the poor implementation of WebGL. WebGL has many advantages and disadvantages as the OpenGL ES 1.0 specifications do not require anyone to implement the whole set of OpenGL ES 1.0 standards to call themselves compliant. Meanwhile, different browsers are implementing this standard with varying requirements, meaning even if a system supports floating point uniform buffers in Firefox, it may not be the case in Chrome on the same system. After performing this research and retrospectively examining this research this has always been a disadvantage of OpenGL and OpenCL when compared against similar technology such as CUDA. Developers prefer standardisation that makes sense. If a piece of hardware claims to be CUDA 6.5 compliant, then all specifications set by that standard are available. However, this is not the case for OpenGL as a piece of hardware claiming to be compliant with OpenGL version 3.0 could only support 10% of the total standard.

7.3 Future Research

7.3.1 Compute Shaders

Modern browsers will see the implementation of WebGL 2.0 in the next specification. The 2016 design specification for WebGL 2.0 provides an API that conforms to the OpenGL ES 3.0 API, and a planned WebGL 2.1 will support OpenGL ES 3.1 operations. Compute Shaders have been integrated into OpenGL ES 3.1 and are specifically designed

to run outside of the normal OpenGL pipeline, enabling compute functions. Many of the limitations faced by this research were due to performing compute functions confined within the graphics pipeline. The aim of Compute Shaders fixes this issue and initial implementation of NTRUEncrypt.js using Compute Shaders has shown significant improvement over OpenGL shaders.

Table 7.1: Preliminary Results of NTRUEncrypt 1499 on Nvidia TK1

	Nvidia TK1
Compute Shaders	264ms
JavaScript	294ms
Java	20ms

As shown in table 7.1, preliminary tests of compute shaders were slower than Java. The compute shaders were able to perform Vec4 operations, meaning that in 264 milliseconds 4 messages were encrypted. This means the encryption time per message using compute shaders is approximately 66 milliseconds.

The convolution operation contains a double for loop that the compiler is unable to expand and optimise. By moving these loops back to the CPU and modifying the convolution computation this should improve the performance. Future testing, development, and optimisation would further improve this algorithm.

7.3.2 Power Testing

This research and experiments made the assumption that running an algorithm as quickly as possible was the best solution for mobile devices. It additionally made the assumption that running on the GPU would reduce power consumption or increase efficiency. Proving these assumptions would require testing by performing power readings of devices and adjusting the scaling to attempt to answer these questions.

7.3.3 Random Number Generation

When implementing NTRUEncrypt in JavaScript, there is currently a debate over whether JavaScript can provide CSPRNGs. For this research window.crypto.getRandomValues was used. However, some researchers consider this only to be a PRNG. While this research experiment was attempting to answer a performance question the security of the developed NTRUEncrypt.js, if used in production, would

require a suitable random number generator to maintain high levels of security.

Appendix

Accelerated CryptoJs

All of the code created specifically for this thesis is available on Github, https://github.com/cptwin/accelerated_cryptojs

References

Amdahl, G. M. (1967). Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the april 18-20, 1967, spring joint computer conference* (pp. 483 485). doi: 10.1145/1465482.1465560

Barker, E. (2016). Nist special publication 800-57 part 1 revision 4. *NIST Special Publication*, *800*(57). Retrieved from `http://nvlpubs.nist.gov/`

Bernstein, D. J. (2009). Introduction to post-quantum cryptography. In *Post-quantum cryptography* (pp. 1 14). Springer.

Borges, H., Valente, M. T., Hora, A. & Coelho, J. (2015). On the popularity of github applications: A preliminary note. *arXiv preprint arXiv:1507.00604* .

Chesebrough, R. & Conlon, C. (2012). *Implementation and performance of aes-ni in cyassl embedded ssl.* yaSSL. Retrieved from `https://www.yassl.com/`

Comparing java vs. c/c++ efficiency differences to interpersonal differences. (n.d.). *Communications of the ACM*, *42*(10), 109 - 112. Retrieved 18/11/2015, from `https://www.researchgate.net/`

Cook, D. & Keromytis, A. D. (2006). *Cryptographics: exploiting graphics cards for security* (Vol. 20). Springer Science & Business Media. doi: 10.1007/0-387-34189 -7

Daemen, J. & Rijmen, V. (1998). The block cipher rijndael. In *Smart card research and applications* (pp. 277 284). doi: 10.1007/10721064 26

Damico, T. M. (2009). A brief history of cryptography. *Student Pulse*, *1*(11). Retrieved from `http://www.inquiriesjournal.com/`

Flanagan, D. (2011). *Javascript: The definitive guide: Activate your web pages.* O'Reilly Media. Retrieved from `https://books.google.co.nz/`

Flynn, M. (1972). Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, *C-21*(9), 948-960. doi: 10.1109/TC.1972.5009071

Gartner. (2015). *Gartner says 6.4 billion connected "things" will be in use in 2016, up 30 percent from 2015.* Retrieved 02/05/2016, from `http://www.gartner .com/`

Gueron, S. (n.d.). Advanced encryption standard (aes) instructions set. Retrieved 04/03/2016, from `https://software.intel.com/`

Hankerson, D., Menezes, A. & Vanstone, S. (2006). *Guide to elliptic curve cryptography.* Springer New York.

Harrison, O. & Waldron, J. (2007). Aes encryption implementation and analysis on commodity graphics processing units. , *4727*, 209-226. doi: 10.1007/978-3-540 -74735-2 15

Harrison, O. & Waldron, J. (2009). Efficient acceleration of asymmetric cryptography on graphics hardware. , *5580*, 350-367. doi: 10.1007/978-3-642 -02384-2 22

Harrison, O. & Waldron, J. (2010). Gpu accelerated cryptography as an os service. , *6480*, 104-130. doi: 10.1007/978-3-642-17697-5 6

Herhut, S., Hudson, R. L., Shpeisman, T. & Sreeram, J. (2013). River trail: A path to parallelism in javascript. *SIGPLAN Not.*, *48*(10), 729 744. doi: 10.1145/ 2544173.2509516

Hermans, J., Vercauteren, F. & Preneel, B. (2010). Speed records for ntru. In *Topics in cryptology-ct-rsa 2010* (pp. 73 88). Springer. doi: 10.1007/978-3-642 -11925-5 6

Hoffstein, J., Pipher, J. & Silverman, J. H. (1998). Ntru: A ring-based public key cryptosystem. In *Algorithmic number theory* (pp. 267 288). Springer. doi: 10.1007/BFb0054868

Jaeger, T. (2007). *Introduction computer and network security.* Retrieved 12/01/2016, from `http://www.cse.psu.edu/`

Jang, K., Han, S., Han, S., Moon, S. B. & Park, K. (2011). Sslshader: Cheap ssl acceleration with commodity processors. In *Nsdi*. Retrieved 17/11/2015, from `https://www.usenix.org/`

Khronos OpenCL Working Group. (2008). The opencl specification. , *1*(29), 8. Retrieved 12/02/2016, from `https://www.khronos.org`

Koblitz, N. (1987). Elliptic curve cryptosystems. *Mathematics of computation*, *48*(177), 203 209. doi: 10.1090/S0025-5718-1987-0866109-5

Kowaliski, C. (2015). *Intel's broadwell-u arrives aboard 15w, 28w mobile processors.* Retrieved 10/06/2016, from `https://techreport.com`

Lenstra, A. K., Lenstra, H. W. & Lovász, L. (1982). Factoring polynomials with rational coefficients. *Mathematische Annalen*, *261*(4), 515 534. doi: 10.1007/ BF01457454

Liskov, M. (2011). Fermat's little theorem. In H. C. A. van Tilborg & S. Jajodia (Eds.), *Encyclopedia of cryptography and security* (pp. 456 456). Boston, MA: Springer US. doi: 10.1007/978-1-4419-5906-5 449

March, S. T. & Smith, G. F. (1995). Design and natural science research on information technology. *Decision support systems*, *15*(4), 251 266. doi: 10.1016/0167-9236(94)00041-2

McEliece, R. (1978). A public-key cryptosystem based on algebraic coding theory. , *4244*, 114 116. Retrieved 01/09/2015, from `http://ntrs.nasa.gov/`

Merkle, R. C. (1978, April). Secure communications over insecure channels. *Commun. ACM*, *21*(4), 294 299. doi: 10.1145/359460.359473

Merkle, R. C., Charles, R. et al. (1979). Secrecy, authentication, and public key systems.

Miller, V. (1986). Use of elliptic curves in cryptography. In *Advances in cryptology-crypto'85 proceedings* (p. 417-426). doi: 10.1007/3-540-39799-X 31

Monz, T., Nigg, D., Martinez, E. A., Brandl, M. F., Schindler, P., Rines, R., ... Blatt, R. (2015). Realization of a scalable shor algorithm.
doi: 10.1126/science.aad9480

Moore, G. E. (1965). Cramming more components onto integrated circuits. *Electronics, 38: 8*, *16*. doi: 10.1109/N-SSC.2006.4785860

National Bureau of Standards. (1977). Federal information processing standards publication 46. *National Bureau of Standards, US Department of Commerce*. Retrieved 03/12/2015, from `http://csrc.nist.gov/`

Nicholls, J. (2012). *Javascript on the gpu.* Retrieved 02/09/2015, from `http://www.slideshare.net/`

Nielsen, J. (1994). Usability engineering.

Nvidia. (2008). *Cuda programming guide.* Retrieved 04/09/2015, from `http://docs.nvidia.com`

Oancea, B., Andrei, T. & Dragoescu, R. M. (2014). GPGPU computing. *arXiv preprint arXiv:1408.6923*.

Perlner, R. A. & Cooper, D. A. (2009). Quantum resistant public key cryptography: A survey. In *Proceedings of the 8th symposium on identity and trust on the internet* (pp. 85 93). New York, NY, USA: ACM. doi: 10.1145/1527017.1527028

Ritchie, D. M. (1993, March). The development of the c language. *SIGPLAN Not.*, *28*(3), 201 208. doi: 10.1145/155360.155580

Ritter, W., Kempter, G. & Werner, T. (2015). User-acceptance of latency in touch interactions. In M. Antona & C. Stephanidis (Eds.), *Universal access in human-computer interaction. access to interaction* (Vol. 9176, p. 139-147). Springer International Publishing. doi: 10.1007/978-3-319-20681-3 13

Rivest, R. L. (1990). Cryptography. In J. V. Leeuwen (Ed.), *Handbook of theoretical computer science* (Vol. 1, pp. 717 755). Elsevier.

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

Schneier, B. (2016). The value of encryption. , *50, No. 2*. Retrieved 09/04/2016, from `http://www.riponsociety.org/`

Shor, P. W. (1999). Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM review*, *41*(2), 303 332. doi: 10.1137/S0036144598347011

Shumow, D. & Ferguson, N. (2007). On the possibility of a back door in the nist sp800-90 dual ec prng. In *Proc. crypto* (Vol. 7). Retrieved 09/04/2016, from `http://rump2007.cr.yp.to/`

StatCounter. (2016). *Statcounter top 9 browsers from jan to apr 2016.* Author. Retrieved 12/04/2016, from `http://gs.statcounter.com/`

Veldhorst, M., Yang, C., Hwang, J., Huang, W., Dehollain, J., Muhonen, J., ... Itoh, K. (2014). A two qubit logic gate in silicon. *arXiv preprint arXiv:1411.5760*.

Weadon, P. D. (2000). *The battle of midway: Af is short of water.* NSA. Retrieved 21/08/2015, from `https://www.nsa.gov/`

Woo, M., Neider, J., Davis, T. & Shreiner, D. (1999). *Opengl programming guide: The official guide to learning opengl.* Addison-Wesley Reading.

Yang, J. & Goodman, J. (2007). Symmetric key cryptography on modern graphics hardware. , *4833*, 249-264. doi: 10.1007/978-3-540-76900-2 15