CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Lightweight Key Generation in IoT

Benchmark and implementation on embedded devices

Master's thesis in Computer Science and Engineering

Tobias J. Andersson, Axel Windisch

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2025

Master's thesis 2025

Lightweight Key Generation in IoT

Benchmark and implementation on embedded devices

Tobias J. Andersson, Axel Windisch

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2025

Lightweight Key Generation in IoT
Benchmark and implementation on embedded devices
Tobias J. Andersson Axel Windisch

Supervisor: Victor Morel, Department of Computer Science and Engineering
Co-supervisor: Francisco Blas Izquierdo Riera, Department of Computer Science and Engineering
Advisor: Daniel Hemberg, Scionova AB
Examiner: Ahmed Ali-Eldin Hassan, Department of Computer Science and Engineering

Master's Thesis 2025
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2025

Lightweight Key Generation in IoT
Benchmark and implementation on embedded devices
Tobias J. Andersson
Axel Windisch
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

The steady growth of active IoT devices has made the question of secure communication by non-traditional digital devices more relevant than ever. One aspect of such security that currently lacks standards is lightweight key generation. The emergence of post-quantum computing, while still novel, presents another dimension on the issue of key generation. In this paper, the key generation of RSA, FourQ, Curve25519, and Streamlined NTRU Prime are benchmarked on three low-power IoT devices. RSA and Curve25519 are selected because they are standard choices in many communication protocols used today. FourQ is selected because it is the fastest Elliptic Curve known right now, with a similar security-level as Curve25519. Streamlined NTRU Prime is selected as it is a post-quantum secure public-key algorithm with relatively small key size. This paper shows that FourQ, Curve25519, and Streamlined NTRU Prime is viable on low-power IoT devices while also presenting a lightweight implementation of Streamlined NTRU Prime which is better suited for use on low-power resource-constrained IoT devices.

Keywords: Lightweight, Key Generation, IoT, Lattice-based, Elliptic curve, Benchmark, Atmega, Xtensa, ARM Cortex, Streamlined NTRU Prime.

Acknowledgements

We would like to thank our supervisor Victor Morel for his patience and guidance and our co-supervisor Francisco Blas Izquierdo Riera for his technical expertise. We would also like to thank our supervisors at Scionova AB, Daniel Hemberg and Erik Dahlgren for their input and for providing the hardware. Finally, we would like to thank our examiner Ahmed Ali-Eldin Hassan and our opponent Carl Blomqvist for their feedback.

Tobias J. Andersson, Axel Windisch, Gothenburg, 2025-08-26

Contents

List of Figures

List of Tables

List of Algorithms

1

Introduction

The amount of everyday devices that are connected to the Web has increased steadily for years [1]. *Internet of Things* (IoT) is a common term used to describe these devices which are not usually thought of as digital machines, such as digital assistants, smart refrigerators, or sensors connected to the Internet. These devices communicate with some server by either sending data, receiving instructions, or both [2]. As with all connected devices, security and privacy are important aspects. Indeed, there have been several major attacks specifically targeting IoT devices.

Some of these attacks have aimed to create botnets, such as the Mirai botnet [3] and Bashlite [4]. The Mirai botnet managed to infect nearly 65 000 IoT devices in less than 24 hours and soon reached steady populations of between 200 000 and 300 000 infected devices. This allowed the creators to perform massive distributed denial-of-service attacks by causing infected devices to simultaneously send requests to a chosen target [3].

There's also been attacks against infrastructure such as water systems in the U.S.A. [5]. In a final example, hackers were able to steal 10 GB of high-roller data from a casino database by gaining access to the internal network through a fish tank thermostat [6]. Attacks like these highlight the need for improved cybersecurity for IoT devices.

Conventional cryptography schemes were developed to protect ordinary computers, phones, and tablets. However, these schemes are often computationally heavy. Therefore, the limited computational power and often limited power reserve of some IoT devices have introduced new challenges for cryptographic security [7]. This report focuses on low-power IoT devices such as sensors. When a fleet of IoT devices are deployed for a longer period of time, keys must be changeable during the deployment period. Traditional schemes and algorithms are often too computationally heavy to be effective on low-power IoT devices, as they require too much memory space or consume too much energy. Instead, lightweight cryptography approaches are necessary [8].

Another issue related to cybersecurity is the advent of post-quantum computing. Although devices capable of post-quantum computing are still a rarity today[1], preparing for a future where they are more common is essential to keep IoT secure in the long run. Implementing post-quantum secure encryption protects data from being intercepted and stored now for later decryption when quantum computing

[1] A few hundred according to an estimation by SpinQ Technology Inc.

become available. Therefore, this report puts a specific emphasis on Streamlined NTRU Prime, a post-quantum secure key generation algorithm in relation to low power IoT devices.

The National Institute of Standards and Technology (NIST) is an American government agency which, among other tasks, aims to provide security standards for companies and government agencies. It is one of the world premier organizations for standardization, alongside the International Organization for Standardization (ISO) [2]. In February 2023 NIST presented the winner of a five-year-long competition aimed at deciding on an encryption standard for IoT devices. The winner of the competition was the Ascon group of cryptographic schemes, subsequently published as NIST's suggested lightweight standard in June 2023 [9]. However, the Ascon standard only covers private-key encryption and hashing, and there is, as of yet, no single standard for public-key key generation schemes. This report examines and benchmarks some alternatives for lightweight public-key key generation.

1.1 Problem

The main problem investigated in this thesis is lightweight key generation on IoT devices. **Key generation** is the process of generating the keys required for encryption and decryption in a cryptographic scheme. This is a computationally heavy part of any encryption scheme, and finding schemes that are more efficient or computationally cheaper than what conventional cryptography offers would be useful to increase the performance and lower the energy cost of IoT devices.

The main challenges are limitations on memory space, energy, and runtime. **Memory space** of IoT devices are often very limited and must be used in an optimal way. **Energy consumption** must be kept low to obtain a reasonable battery lifetime. **Runtime** constraints are also important, as users desire an adequate level of responsiveness.

More specifically, the research questions this thesis aims to answer are:

RQ1 Does lattice-based or elliptic curve key generation schemes perform the best?

RQ2 What are the memory requirements of the algorithms, and how can they be reduced to fit on small devices?

RQ3 How do the algorithms' performance vary depending on the processor architecture?

To address these questions, this project implements and benchmarks some popular and/or emerging lightweight cryptographic algorithms for key generation. The implementation is performed on a few different microcontrollers that are representative of common industry standards. The benchmarks will measure **memory space utilization**, **CPU utilization**, **power consumption**, and **runtime**.

The chosen schemes are based on elliptic curves or lattices, which are two different

[2] `https://www.iso.org`

classes of key generation algorithms. In particular, **FourQ**, which is based on elliptic curves, and the lattice based **Streamlined NTRU Prime** (SNTRUP) [10] will be tested. FourQ is the fastest known Elliptic Curve cryptographic scheme on traditional computers [11]. NTRU Prime is chosen to represent Lattice-based cryptography since it is one of the fastest alternatives according to Fernandez-Carames [12]. The popular key generation schemes **RSA** and **Curve25519** will be included for comparison, but are expected to perform worse than FourQ and NTRU Prime according to the result of previous surveys [9], [13].

The chosen devices represent three types of architecture which are commonly found in IoT devices. **Arduino Nano ESP32** is based on the ESP32 architecture and is the least constrained of the three devices, with 512 KB RAM and 240 MHz clock speed [14]. **Raspberry Pi Pico W** is based on the ARM architecture and has 264 KB RAM and 133 MHz clock speed [15]. **Arduino Mega 2560 Rev3** is based on the AVR[3]-type Advanced RISC architecture (ARA) and has 8 KB RAM and 16 MHz clock speed [16]. Out of the architecture types investigated in this report, AVR is the most constrained. In particular, the Arduino Mega 2560 Rev3 used in this report has only 8 KB of SRAM and a 16 MHz clock speed. As such, examining the viability of public key cryptography on this device is important.

Our contributions are:

- A comparison of the performance of the key-generation of four different public key cryptographic schemes on three common IoT architectures.
- The first fully functioning implementation of the Streamlined NTRU Prime key generation algorithm able to run on 8 KB RAM.

1.2 Report Outline

Chapter 2 introduces the background, with the basics of cryptography covered in Section 2.1. Public-key cryptography is explained in Section 2.2, which also cover details of the cryptographic schemes examined in the report. The architecture types examined are described in 2.3.

The related work is presented in Chapter 3. Section 3.1 presents benchmarks on lightweight public-key cryptography and Section 3.2 presents benchmarks on other lightweight cryptographic systems. Section 3.3 presents previous implementations of similar cryptographic schemes on AVR architecture.

The methods used are described in Chapter 4, where the process of algorithm selection is described in Section 4.1. The benchmarked implementations (in particular of SNTRUP on the Arduino MEGA) are described in Section 4.3. The gathering of benchmark data is described in Section 4.4, and the analysis of the collected data is described in Section 4.5.

[3]AVR is not officially an abbreviation of anything in particular. It is used as the full name, capitalized, by the manufacturer Atmel.

Chapter 5 contains the results of the benchmark and some short comparisons between the results of the benchmarks. Section 5.2 presents the average power consumption for each benchmark. In Section 5.1, the runtime and approximate clock cycle count of each benchmark is presented. Lastly, the memory consumption of each benchmark is presented in Section 5.3.

Chapter 6 contains both a discussion of the result in Section 6.1 and a presentation of the limitations of the report in Section 6.2. Additionally, risks and ethics are discussed in Section 6.3. Lastly Chapter 7 which contains the conclusion drawn from the results and discussion.

2

Background

While a standard for symmetric key cryptography has been introduced by NIST [9], no standard for public key lightweight cryptography exists. Public key cryptography allows a device to change keys during deployment, and some can also be modified to provide verification. A standard suite for post-quantum cryptography has been presented by NIST [17], but is not focused on resource-constrained devices.

The rest of this chapter is structured as follows: Section 2.1 introduces basic terms and concepts used in cryptography. Public key cryptography in general and the particular algorithms benchmarked are discussed in the introduction of Section 2.2. Lastly, Section 2.3 describes the microprocessor architectures used for the benchmark.

2.1 The Basics of Cryptography

Cryptography can be divided into **private-key cryptography** and **public-key cryptography** [18]. Private-key, also known as symmetric-key, cryptography uses a single secret key that is shared by all parties to encrypt data and is used to encrypt bulk data. Encryption by private-key cryptography is quick and requires relatively small key sizes. A major problem with private-key cryptography is how to securely distribute this key.

In public-key cryptography, two sets of keys are used. Each party has one unique set of keys, and each set of keys consists of a public and a private key. The sender encrypts the data with the receiver's public key, and this data can only be decrypted with the receiver's private key. This allows secure transmission of data over insecure communication channels. However, it is also significantly more computationally heavy than private-key cryptography. Therefore, it is common to select the secret private key for communication between two parties by public-key cryptography, and use private-key encryption for the rest of the communication.

Hash functions are another important class of functions used in cryptography. A hash function takes as input a string of arbitrary length and outputs a digest which is a string of fixed length. This is useful in cryptography because it can be used to generate a bit string from a message before encryption that is attached to the ciphertext after encryption. The receiver may then use the same hash function on the decrypted message. If the digest attached to the ciphertext and the digest generated by the receiver match, there is reasonable proof that the message has not

been tampered with.

The development of quantum computers is expected to break several cryptographic schemes such as RSA [19] and ECDH [20], two of the schemes described in Section 2.2. **Post-Quantum Security** is a field of cryptographic research which aim to ensure cryptographic security in a post-quantum world. While post-quantum security is not yet as critical as classical security, exploring post-quantum secure options is important.

2.1.1 Lightweight Cryptography

IoT devices have very limited CPU-power and space. This makes conventional cryptography too demanding if current standards are to be upheld, and too insecure if lesser standards are used [8]. Instead, NIST recommends lightweight cryptography where one or more parts, such as block size or key schedules of an encryption scheme, are scaled back or simplified.

Ascon was published as NIST's chosen suite for lightweight encryption standardization in June 2023 [9]. It is a lightweight cipher suite that provides authenticated encryption with associated data and hashing functionality [21] using 128-bit keys. Ascon covers both private-key encryption and hash functions, but does not standardize public-key cryptography. This project focuses specifically on lightweight key generation and generally on public-key cryptography.

2.1.2 Definitions

A common way to define the strength of a specific implementation of a cryptographic scheme is by its security level. This is often presented as "n-bit" security which means that an attacker would statistically have to perform 2^n attacks to break the encryption. Security recommendations will often suggest a minimum security level.

Two common types of security definition is security against Chosen-plaintext attack (CPA) and against Chosen-ciphertext attack (CCA). CPA means that an attacker can get the encrypted result (ciphertext) of any plaintext given to the cryptosystem. CCA means that an attacker can get the decrypted result (plaintext) of any ciphertext given to the cryptosystem.

2.2 Public-key Cryptographic Systems

There are several popular key generation schemes with their own benefits and drawbacks [22]. **RSA** is secure and can perform additional tasks such as signing and encryption/decryption [23]. Elliptic-Curve Diffie-Hellman **ECDH** is an improvement of the **Diffie-Hellman** exchange, which employs elliptic curve cryptography to gain advantages in terms of key length and overall performance [24]. While ECDH can only perform key generation and exchange, it performs these tasks more efficiently than RSA [25]. Another potentially useful family of schemes is **NTRU** [26], a class of post-quantum secure lattice-based schemes. A major difference between

ECDH/Lattice-based cryptography and RSA is that the former two relies heavily on field theory for security, while the latter relies on the difficulty of factorization for security.

All these cryptographic systems rely on the assumption that the mathematical problem of finding the input of a given algorithm's output is **computationally hard**. A problem is computationally hard if it cannot be solved by any polynomial time algorithm. Proving computational hardness is difficult [27]. Computational hardness is a key aspect when defining the security of a cryptographic system.

RSA is presented in Subsection 2.2.1. The Diffie-Hellman Key exchange is presented in Subsection 2.2.2, and elliptic curves are described in Subsection 2.2.3. Lastly, lattice-based cryptography is presented in Subsection 2.2.4. In the following subsections, we will call the initiator of a cryptographic scheme Alice and the receiver Bob. This is standard practice for such descriptions.

2.2.1 RSA

RSA was one of the first public-key cryptosystems introduced and is widely adopted. The key generation is initialized by calculating $n = p \times q$, where (p, q) are two large random primes. Next, another large random integer d is chosen which is relatively prime to $(p-1) \times (q-1)$. Finally, the integer e which is the multiplicative inverse of $d \bmod (p-1) \times (q-1)$ is calculated. The relation between the numbers are $e \times d \equiv 1 \bmod ((p-1) \times (q-1))$. Given a message m, encryption to a ciphertext c is calculated as $c = m^e \bmod n$ and decryption from c is calculated as $m = c^d \bmod n = c^{e \times d} \bmod n$. It is based on the security assumption that factorization of the large primes (p, q) is computationally hard [23].

2.2.2 Diffie-Hellman Key Exchange

The Diffie-Hellman key exchange is a popular public-key protocol where the two parties Alice and Bob can securely exchange keys and communicate through an insecure channel. A Diffie-Hellman key exchange between Alice and Bob is described in Algorithm 4. The secret key s is the same for both sides because

$$s = pk_B^{sk_A} = g^{sk_b \times sk_a} = g^{sk_a \times sk_b} = pk_A^{sk_B},$$

where all expressions are mod p [28]. Finding sk_A given only g and pk_A is called the Discrete Logarithm Problem and is assumed to be computationally hard.

2.2.3 Elliptic Curves

Another important mathematical object in cryptography is elliptic curves. Given a finite field F_q, an elliptic curve E is a curve that consists of all points in F that satisfies the equation $y^2 = x^3 + ax + b$ and a point representing infinity. Let P be a point on E of prime order n. Then $\langle P \rangle = \{\mathcal{O}, P, 2 \times P, 3 \times P, ..., (n-1) \times P\}$ is the cyclic subgroup of E generated by P. Finding k given a point $Q = k * P \in \langle P \rangle$ and P is called the Discrete Logarithm Problem over Elliptic Curves (ECDLP). ECDLP

is assumed to be computationally hard [29]. In Elliptic Curve Cryptography, k is the private key and $Q = k \times P$ is the public key. Elliptic curves are often used as key spaces for Diffie-Hellman key exchanges. The resulting schemes are called **Elliptic Curve Diffie-Hellman** (ECDH).

2.2.3.1 Curve25519

Curve25519 [30] is a specific elliptic curve and associated function library developed by D.J. Bernstein. The curve was chosen to provide high speed, short keys, and short code. It also provides protection against timing attacks as there is no time variability, and does not require key validation because every 32 byte string is accepted as a Curve25519 public key. In particular, the curve was chosen because it allows faster x-coordinate point addition and x-coordinate scalar multiplication by the formulas suggested by Montgomery in [31].

Curve25519 is an F_p restricted x-coordinate scalar multiplication on $E(F_{p^2})$, where $p = 2^{255}-19$ and $E = y^2 = x^3+48662x^2+x$. In order to optimize the speed, Bernstein made several design choices aimed at reducing the complexity of calculations and unwarranted variance. This included using x as a public key instead of (x,y), using a prime field instead of an extension field, and using a known secure curve with a known secure twist instead of wasting time on prohibiting keys on the twist.

2.2.3.2 FourQ

Four$\mathbb{Q}$ is a very fast elliptic curve aimed at a 128-bit security level [11]. The curve is the twisted Edwards curve

$$\mathcal{E}/\mathbb{F}_{p^2} : -x^2 + y^2 = 1 + dx^2y^2$$

defined in the quadratic extension field

$$\mathbb{F}_{p^2} := F_p(i), \quad \text{where} \quad p = 2^{127} - 1 \quad \text{and} \quad i^2 = -1.$$

The Four$\mathbb{Q}$ curve and library is designed to be fast, simple, versatile, and available to the public. In particular, it is designed to be the fastest elliptic curve currently available. The experiments performed by Costello et al. [11] shows that Four$\mathbb{Q}$ is significantly faster than the competing schemes. One such design choice is to exploit endomorphisms present in the curve. With clever implementation, multiscalar multiplication can be sped up significantly through scalar decomposition supported by the endomorphisms. Even without accelerating operation with formulas based on endomorphism, Four$\mathbb{Q}$ is about 1.2–1.6 times faster than Curve25519 on traditional (desktop) computers.

2.2.4 Lattice-based Cryptography

Lattice-based cryptography is an emerging class of public key cryptography that has gained some popularity among cryptographers due to its small key size, fast operations, and post-quantum security. It is based on vector problems in lattices.

Given two points $(A, B) \in \mathcal{R}^n$ and an origin O, a lattice is the infinite field created by combinations of adding or subtracting the vectors AO and BO. Given a third point C, the (search) Shortest Vector Problem (SVP) is defined as finding the minimum non-zero Euclidean distance from C while traversing the points of the lattice. SVP is assumed to be computationally hard.

This project only examines **NTRU Prime** which is a more secure improvement of NTRU Classic, as we discuss next. Streamlined NTRU Prime is used in combination with X25519 as the default key exchange method in OpenSSH as of version 9.0 [32]. An alternative lattice-based technique is **learning with errors** (LWE) introduced by Oded Regev [33].

The cryptographic security of LWE relies on the difficulty of the Shortest Independent Vector (SIVP) and GapSVP problems, a system of equations for public and private keys, and on a vector of "errors" that is inserted during encryption. Decryption is done through iterative loops between Gaussian elimination and a quantum algorithm using a Closest Vector Problem-oracle and the quantum Fourier transform. The LWE cryptosystem as described by Regev requires key material to be distributed by some other method. This weakens one of the major advantages of a public key cryptosystem over a private key alternative. It is however often included in discussions of, and benchmarks on, post-quantum cryptography. Further reading can be found in Appendix B.

2.2.4.1 NTRU Classic

NTRU Classic [26] is a public key cryptosystem which uses a polynomial-based mixing system and reduction modulo p,q for encryption, and an unmixing system whose validity depends on elementary probability theory. A polynomial-based mixing system deterministically maps a set onto itself in another order according to some pre-specified polynomial. Unmixing performs the same operation in reverse. The security of the system relies on the interaction between mixing and reduction, and the difficulty of finding extremely short vectors.

An instance of the NTRU cryptosystem depends on three integer parameters (N,p,q) and four polynomial sets $\mathcal{L}_f$, $\mathcal{L}_g$, $\mathcal{L}_\phi$, and $\mathcal{L}_m$ with degree N-1 and integer coefficients. p does not have to be prime, but $gcd(p, q) = 1$ and q should be considerably larger than p and a power of 2. All values can be found in the ring $R = \mathbb{Z}[X]/(X^N - 1)$.

When creating an NTRU key, Alice first choose two random polynomials $f_A, g \in \mathcal{L}_g$. The polynomial f must have inverses modulo q and modulo p. These inverses are denoted by F_q and F_p such that

$$F_q \circledast f_A \equiv 1 \ (mod\ q) \ and \ F_p \circledast f_A \equiv 1 \ (mod\ p). \qquad (1)$$

Next, the quantity

$$h_A \equiv F_q \circledast g \ (mod\ q) \qquad (2)$$

is computed. The polynomial h_A is Alice's public key and the polynomial f_A is her private key.

Alice can then encrypt a message m with Bob's public key h_B as

$$e \equiv p\phi \circledast h_B + m \ (mod\ q),$$

where $\phi \in \mathcal{L}_\phi$ is randomly selected. Bob decrypts e by first computing

$$a \equiv f_B \circledast e \ (mod\ q),$$

where f_B is his private key and the coefficient a is chosen from the interval $(-q/2, q/2)$. Treating a as a polynomial with integer coefficients, Bob recovers the message by computing

$$F_p \circledast a \ (mod\ p).$$

2.2.4.2 Streamlined NTRU Prime

Streamlined NTRU Prime (SNTRUP) [10] uses a large Galois group of prime degree in order to mitigate certain vulnerabilities of NTRU Classic such as Odlyzko's meet-in-the-middle attack [34] and information leaks. While NTRU Classic uses rings of the form $(\mathbb{Z}/q)[x]/(x^p-1)$ where p is a prime and q is a power of 2, SNTRUP uses fields of the form $(\mathbb{Z}/q)[x]/(x^p - x - 1)$, where p, q are prime.

SNTRUP is parameterized by positive integers (p, q, t) where p, q are prime numbers. Furthermore, $t \geq 1$, $p \geq 3t$, $q \geq 32t+1$, and $x^p - x - 1$ is irreducible in the polynomial ring $(\mathbb{Z}/q)[x]$. The ring $\mathbb{Z}[x]/(x^p - x - 1)$, the ring $(\mathbb{Z}/3)[x]/(x^p - x - 1)$, and the field $(\mathbb{Z}/q)[x]/(x^p - x - 1)$ are abbreviated as $\mathcal{R}, \mathcal{R}/3,$ and $\mathcal{R}/q$ respectively. An element of $\mathcal{R}$ is **small** if all of its coeffficients are in $-1, 0, 1$. An element is $t - small$ if exactly $2t$ of its coefficients are nonzero.

If Alice wants to send a message m to Bob, Bob must have an NTRU public key. This key is generated by Algorithm 5. Alice can then use Bob's public key to **encapsulate** m according to Algorithm 6 to generate a ciphertext $C||c$, where c is the encrypted message and C is the key confirmation used for authentication. Lastly Bob can then **decapsulate** $C||c$ to obtain the message m from c and authenticate its correctness from C through Algorithm 7.

2.3 Architecture Types

Traditional desktop computers and IoT devices differ significantly in their architectures due to the different requirements for their different use cases. The prominent instruction set used in almost all desktop computers is the x86-64 architecture, which is a 64-bit instruction set architecture (ISA). Processors that are based on the x86-64 ISA are called complex instruction set computer (CISC) processors, due to the instruction set being complex and usually performs several low-level operations per instruction. While the x86-64 based processors are good for performance and portability, they are usually not very energy efficient, nor memory space-efficient. This has led IoT device manufacturers to look elsewhere for more efficient architecture types. This in turn has led the IoT development scene to be much more heterogeneous than the traditional desktop computer scene when it comes to architecture types used.

One common trait of the architectures used in IoT devices is that they are almost always based on some ISA that makes them reduced instruction set computer (RISC) processors. A RISC processor usually executes only one or a few low-level operations per instruction. For example handling memory access and arithmetic are usually separate operations in RISC machines, whereas they can be executed using one instruction in CISC machines. We explain the different architectures used in this report in more details in Subsections 2.3.3 – 2.3.1.

2.3.1 Arduino MEGA 2560 Rev3

Arduino MEGA 2560 is based on the ATmega2560 microcontroller, which is an 8-bit AVR RISC-based microcontroller developed by Atmel Corporation. The microcontroller includes 4kB of EEPROM, 8kB SRAM, 256kB of flash memory and has support for 54 digital I/O lines, 16 of which also support analog I/O. It supports common communication protocols such as Universal Synchronous and Asynchronous Receiver-Transmitter (USART) and Serial Peripheral Interface (SPI). The Arduino MEGA 2560 does not have support for any wireless communications.

2.3.2 Arduino Nano ESP32

Arduino Nano ESP32 is based on the ESP32-S3 microcontroller developed by Espressif. The ESP32-S3 features a dual-core Xtensa LX7 processor, an 32-bit RISC architecture developed by Cadence Design Systems. The processor is described to offer high performance and to be highly configurable. The architecture is designed so that it can be customized to include only the required features needed for a specific application, reducing both the silicon area and the power consumption. LX7 is also customizable in the sense of extendability, as the architecture supports hardware accelerators, which allows for efficient implementations of specific functions such as cryptographic functions and hash functions.

The Arduino Nano ESP32 features 512kB of RAM as well as 384kB of ROM, and the processor is running at a speed of 240MHz. The Arduino Nano ESP32 is expected to be the fastest device due to its high clock-speed and high memory. In terms of wireless connectivity, the Arduino Nano ESP32 is capable of Wi-Fi and Bluetooth Low-Energy.

2.3.3 Raspberry Pi Pico W

Raspberry Pi Pico W is a based on the RP2040 microcontroller. The RP2040 is a dual-core microcontroller which is based on the ARM Cortex-M0+ architecture. ARM Cortex-M0+ is a RISC processor core developed by ARM Limited. The processor is built on the Armv6-M architecture and has support for Thumb/Thumb-2 subset instruction set architecture. The main difference between the Thumb and Thumb2 instruction sets are that Thumb has a 16-bit instruction width, which reduces code size and improves fetch efficiency on narrow memory buses. Thumb2 on the other hand works in a hybrid mode between 16-bit and 32-bit instructions. With the support of 32-bit instructions, a larger subset of the full ARM instruction

set architecture can be supported, but not all. An interesting aspect of the ARM Cortex-M0+ architecture is that it lacks hardware to perform division, this can however be achieved using software.

Raspberry Pi Pico W comes with 264 kB of SRAM and 2 MB of ROM. According to the RP2040 datasheet, the SRAM is physically partitioned into six banks: four 64kB banks and two 4kB banks. This is done to improve memory bandwidth when using both cores. However, the datasheet is a bit unclear about how these two works as it states "Software *may* choose to use these for per-core purposes, e.g. stack and frequently-executed code, guaranteeing that the processors never stall on these accesses.", so if the system is running using both cores, there might be a limit of 4kB for the stack. Lastly, the Raspberry Pi Pico W has support for Wi-Fi, as well as Bluetooth 5.2.

3

Related Work

Lightweight key generation as a topic has not been as thoroughly examined as many other security or lightweight cryptography questions. Most surveys of lightweight cryptography that exists mainly focuses on private-key cryptography, with some research going into hash algorithms. There are, however, a few surveys and benchmark that either surveys lightweight public key generation or provides some baseline for comparison between lightweight public-key and private-key cryptographic schemes.

Another topic covered in this paper is the implementation and optimization of cryptographic schemes for these platforms. Given the constraints of the platforms involved, even minor improvements might provide tangible benefits. In particular, we have looked at work on optimizing cryptosystems and associated algorithms for AVR architecture, since the Arduino MEGA is the least powerfull of the three microprocessors covered in this paper.

This chapter starts by covering previous lightweight public-key cryptographic benchmarks in Section 3.1. Other lightweight key generation benchmarks are covered in Section 3.2. Lastly, some previous implementations of relevant cryptosystems for AVR architecture are covered in Section 3.3

3.1 Lightweight Public-key Cryptographic Benchmarks

Lara-Nino et al. [35] (2018) provided a survey on elliptic curve cryptography in a lightweight context. The paper compared around 20 implementations on various processors, comparing runtime, cycles, and in some cases power, energy, and memory usage. Since the comparison is based on reports that have different methodology, report different parameters, and use different hardware, it is somewhat difficult to draw definite conclusions from the results. However, the paper provided a comprehensive summary of the protocols, implementations, and design ideas of that time.

Buchanan et al. [7] (2017) benchmarked the memory size and execution time of several lightweight block/stream ciphers and hash functions. It also briefly introduces an asymmetric authentication method for RFID devices.

Chaudhary et al. [36] (2019) compared a few different lattice-based variants, including

NTRU Classic, Learning with Errors (LWE) and Ring-LWE. The two most significant points from this report are that a) Ring-LWE is significantly faster than general LWE, and b) NTRU Classic was implemented on IoT hardware while the other two were implemented on a conventional processor, which makes comparison between NTRU Classic and the alternatives complicated.

While Fernandez-Carames [12] (2020) focused on post-quantum resistant cryptosystems, the results are still relevant for this project because it surveys cryptosystems for Internet of Things. Out of all the surveys examined, this is the most comprehensive on public-key lightweight cryptography, hence the results were used as a basis for the choice of cryptosystems to examine further. In the conclusion of this paper, the authors offer a prediction of future trends and suggests suitable foci for further research.

3.2 Benchmarks on Other Lightweight Cryptographic Systems

As part of the report on the NIST Lightweight Competition, Turan et al. [9] (2023) performed benchmarks on several private-key lightweight algorithms, but did not include any public-key key generation algorithms. McKay et al. [8] (2017) also studied lightweight cryptography for NIST, but this report only discusses the RSA or Advanced Encryption Standard (AES)-based lightweight key exchange scheme ALIKE, and does not provide benchmarking at all. Another comprehensive survey of different private-key lightweight cryptosystems was presented by Thakor et al. [37] (2021). Since the surveyed cryptosystems are private key, the results are only tangentially relevant to this project.

Alvarez et al. [13] (2017) provided execution times of key generation and key exchange of RSA, the elliptic curve P256, Curve25519, and FourQ on an Intel Core i7-5930k CPU with 32 GB of RAM. RSA, Curve25519, and FourQ are also benchmarked in our report. However, there are major differences between the benchmark procedure. The microprocessors used in our report are significantly slower and have significantly less memory space. The benchmarks presented in our report also covers power consumption, memory usage, and cycle counts.

3.3 Implementations on AVR Architecture

Arduino MEGA Rev3 is the weakest of our three platforms and represents AVR architecture, requiring optimizations to run reasonably well. Both hashing and specific algorithms have been improved and in part optimized for AVR architectures in some papers.

Osvik [38] improved the execution speed of SHA256 for AVR architecture specifically. Cheng et al. [39] built upon these results to improve Sigma operations of the SHA-512 algorithm by merging bytewise right-rotation with XOR. They also optimize memory accesses by using Indirect Addressing Mode with Displacement so that word-wise

rotation can be done without unrolling. This saves clock cycles but slightly increases RAM usage.

Hutter et al. [40] ported the NaCl library to AVR microcontrollers as a proof of concept and also provided porting descriptions and benchmarks. The benchmarks were carried out on an ATmega2560, which is significantly more powerful than the platform we are using. The authors provided two different implementations, one focused on performance and one which seeks to limit space requirements.

Cheng et al. [41] optimized NTRU Classic for 8-bit AVR microcontrollers, but not Streamlined NTRU Prime. The hash algorithm is the same as in Cheng et al. [39]. Another major focus of this paper is the optimization of memory usage without making timing attacks viable.

Streamlined NTRU Prime for AVR architecture was later implemented by Cheng et al. [42] on an ATmega1281. This device is similar to the ATMega2560 that is used in this report, but notably has twice as much RAM memory. Through a combination of Toom-Cook and Karatsuba multiplication techniques, they were able to achieve reasonable execution times for Streamlined NTRU Prime.

Boorgany et al. [43] implemented R-LWE and NTRU Classic on smart cards and improved Fast Fourier Transform (FFT) and discrete Gaussian sampling. The authors test two different methods to make sampling more efficient: using a cumulative distribution table or a Bernoulli sampler. The implementation provides CPA-security but not CCA-security since the padding necessary is removed. The results show that half of the clock cycles during execution is used by FFT, while another third is used for sparse multiplication.

Monteverde [44] implemented an NTRU Classic variant on ATmega128. While this implementation is well optimized, it is not an NTRU Prime variant and is thus potentially vulnerable to the same lattice-based attacks as NTRU Classic. There are still several interesting optimization techniques and tricks that may be viable for Streamlined NTRU Prime implementations as well.

4

Methods

In order to examine the viability of both classical and post-quantum secure public key generation, four different public key algorithms will be benchmarked on three different low power IoT devices. The benchmarks are performed by running the FourQ, Curve25519, RSA, and NTRU algorithms on an Arduino Nano ESP32, a Raspberry Pi Pico W, and an Arduino Mega 2560 microcontroller. During the execution of these algorithms, current and runtime is measured. The process of **algorithm selection** is described in Section 4.1. In Section 4.2, the process of finding existing implementations of the chosen algorithms is described. In the **implementation** phase described in Section 4.3, hardware-specific implementations of the algorithms are written or modified as needed for the platforms. During the **benchmarking** phase described in Section 4.4, tests measuring power consumption, memory consumption, and speed are run on the microprocessors. Finally, the **data analysis** of the benchmarks and the software used to perform it are described in Section 4.5.

4.1 Algorithm Selection

The algorithms selected for comparison are RSA, Curve25519, FourQ and Streamlined NTRU Prime. RSA was chosen to provide a clear baseline when conducting the benchmarks. It is a historically significant public-key cryptosystem which dates back to 1977, making it by far the oldest algorithm in the selection. RSA has been a standard for several decades in protocols such as SSL, TLS, PGP [45] and SSH. This makes RSA a well-tested and widely adopted protocol which makes it a suitable candidate to use as a baseline. Since RSA is an old algorithm, it is expected to be slower than the other contenders. RSA is based on prime number factoring, which leads to the keys being large.

Curve25519 was chosen because it is a modern, state-of-the-art elliptic curve that is widely adopted today. It can be used for key-exchange in X25519, and signing using Ed25519. Many modern applications are using the Curve25519 elliptic curve, such as WhatsApp, Signal and Wireguard [46]. All of these applications make promises on being secure, which gives credibility to the Curve25519 as a cryptographic primitive.

FourQ is an elliptic curve-based cryptosystem that has been developed by Microsoft Research in 2015. With promises of being faster than Curve25519 in regard to variable-base scalar multiplications, which is one of the core operations in the key

generation process of Curve25519, it is interesting to test if the promised speeds of FourQ can be translated to viable execution time on very resource constrained devices.

Streamlined NTRU Prime was selected as a modern, quantum-resistant public-key cryptosystem. Its use as a key-exchange algorithm in OpenSSH made it a good choice for a quantum-resistant algorithm that have seen use in a real-world application since 2022. Since OpenSSH is used in all major operating systems, including Microsoft Windows, Apple macOS and Linux, SNTRUP is widely adopted and was therefore chosen.

4.2 Survey of Existing Implementations

Both the Arduino Nano ESP32 and Raspberry Pi Pico W have support for Mbed TLS, which is an open source library that provides cryptographic APIs. Mbed TLS supports both RSA and elliptic curves, however since the API does not allow a program to only generate a Curve25519 key without performing a whole ECDH, Mbed TLS could not be used to measure the Curve25519 key generation. The RSA key generation implementation provided by Mbed TLS could be used on both Arduino Nano ESP32 and Raspberry Pi Pico W.

An implementation of Curve25519 optimized specifically for embedded devices, developed by Daniel Beer and Nikolas Rösener was selected. Their implementation is small, easy to understand and is developed entirely in C, making it a good candidate for our benchmarks.

There are not many FourQ implementations available, so the only feasible implementation is the official reference implementation developed by Microsoft Research. They provide six different implementations that are optimized for certain targets and platforms. We used the “64-bit and portable” implementation for all three platforms, and more specific optimizations will be discussed in Section 4.3.1.

For NTRU Prime, the C reference implementation written by Daniel J. Bernstein et al. was used on both Arduino Mega ESP32 and Raspberry Pi Pico. However, the reference implementation used too much memory for the program to be able to run on the Arduino Mega 2560 with only 8KB of memory. Unable to find any other implementation of the Streamlined NTRU Prime that was optimized for embedded or resource-constrained systems, a new implementation was developed which is described in Section 4.3.4.

4.3 Implementations

The implementation of the chosen lightweight cryptographic key generation scheme for the microcontrollers are based on currently available or proposed algorithms, including existing implementations and libraries. The algorithms are benchmarked on the hardware platforms with regard to execution speed, memory space requirements, and power demand.

The test programs performs the key generation process of the algorithms and are written in C. Compared to the Arduino Nano ESP32 and Raspberry Pi Pico W, the Arduino Mega 2560 has significantly less memory. Consequently, extra care must be taken to ensure a space-efficient implementation. Additionally, the Arduino Mega operates at a lower clock frequency, so development must balance memory-efficiency with execution speed to avoid unreasonable performance degradation.

The implementation choices are presented in the following sections ordered by algorithm. First, the choices for **FourQ** are presented in Subsection 4.3.1, followed by the choices for the related **Curve25519** in Subsection 4.3.2. The implementation choices for **RSA** are presented in Subsection 4.3.3. Lastly, the choices for **NTRU Prime** are presented in Subsection 4.3.4.

4.3.1 FourQ

FourQ has around 128-bit security level [11] regardless of implementation, as this is defined by the specifications of the FourQ algorithm itself. When using the reference implementation of FourQ, there are options in **FourQ.h** that can be set to enable some optimizations. The parameters used in this report can be seen here:

```
#define RADIX           32
typedef uint32_t        digit_t;      // Unsigned 32-bit digit
typedef int32_t         sdigit_t;     // Signed 32-bit digit
#define NWORDS_FIELD    4
#define NWORDS_ORDER    8
```

The reference implementation of FourQ uses precomputed tables to increase the execution speed of both variable-base, fixed-based and double scalar multiplications. These tables are by default too large to fit the memory of the Arduino Mega. This was fixed by reducing the window width W and table parameter V in order to calculate smaller tables to save space. Additionally, these tables were placed in flash memory to further reduce the memory requirement.

4.3.2 Curve25519

The implementation of Curve25519 developed by Daniel Beer and Nikolas Rösener was used for all three platforms. Their implementation is written entirely in C and is optimized specifically for low-memory systems. On their GitHub page, they have conducted a series of tests on an ATmega128, which is a similar type of microprocessor as the Arduino Mega 2560 which also use an ATmega processing unit. They also claim that their implementation of the scalar multiplication use under half a kilobyte of stack memory. The implementation has a 128-bit security level.

4.3.3 RSA

The RSA implementation was implemented using a fork of a subset of the GNU Multiple Precision Arithmetic Library (GMP), called gmp-ino. This library contained

the mini-gmp modules and was used to represent large integer numbers. GMP represent large numbers by allocating a data type called `mpz_t`, this data structure uses an array of limbs that contains enough bits for the desired number. A limb is an integer of a set size. These limbs were originally set to 16-bit integers in gmp-ino, which was changed to 8-bit integers as the Arduino Mega is an 8-bit microcontroller.

The gmp-ino library provided functions to perform some necessary arithmetic used in RSA. However, some of these implementations were too expensive when considering memory usage, consequently, some memory-optimized implementations were implemented. One important notice on this implementation is that the generation of random primes were disabled during the benchmarking. Since finding primes with a length of 1024-bits is a non-deterministic problem, it could take hours to find primes on the 8-bit microcontroller. Instead, a precalculated prime of sufficient size was used in order to be able to examine the run time of the rest of the key generation algorithm. Finding primes efficiently is a whole research subject in itself, and it was decided that the focus should be targeted towards the other, more modern cryptosystems. Both this implementation and the implementations provided by Mbed TLS uses key sizes of 2048 bits, which translates to a 112-bit security level.

4.3.4 Streamlined NTRU Prime

All the SNTRUP implementations have the following parameter set, $p = 761, q = 4591$, and $t = 143$. This results in a security level of 128-bits.

In order to implement SNTRUP to work on 8 KB of RAM, we had to use some properties of the key generation process to minimize memory usage and reuse as much memory as possible. The main part of the key generation revolves around finding polynomial inverses for polynomials with the length p and with a coefficient modulo q.

A polynomial inverse exists if the greatest common divisor (GCD) between two functions is 1. The GCD of two polynomials can be found with the polynomial extended GCD algorithm as shown in Algorithm 1. Here a is assumed to be the ring polynomial and b the polynomial that is being tested for invertibility.

Because the polynomial space in SNTRUP are rings, a polynomial $\mathbf{P}$ is only invertible if the GCD of $\mathbf{P}$ and the ring polynomial is 1. This makes it possible to divide the process of the GCD algorithm into two, the first part where all the quotients of the divisions are calculated, and a second part where the polynomial inverse is reconstructed using these quotients. Since we are only testing if a specific polynomial has an inverse, and the ring polynomial is given, we are only interested in the t-polynomial from algorithm 1.

At every step of the polynomial Euclidean division, the newly computed coefficient of the quotient can replace the largest zeroed coefficient of the dividend. As a result, the combined length of the resulting quotient and remainder of the division will always be less or equal to the length of the old dividend. This property is very important, as it allows us to make the whole division in-place by overwriting the old dividend with the new remainder followed by the quotient. This way, the two

Algorithm 1 Extended GCD Algorithm

Require: a, b univariate polynomials
  $(r_0, r_1) = (a, b)$
  $(s_0, s_1) = (1, 0)$
  $(t_0, t_1) = (0, 1)$
  for $i := 1; r_i \neq 0; i := i + 1$ **do**
    q := quo(r_{i-1}, r_i)
    $r_{i+1} := r_{i-1} - qr_i$
    $s_{i+1} := s_{i-1} - qs_i$
    $t_{i+1} := t_{i-1} - qt_i$
  end for
  $g = r_{i-1}$
  $u = s_{i-1}$
  $v = t_{i-1}$
  $au + bv = g$

arrays representing the polynomials will be getting replaced by all the quotients and the latest remainder. All extra information we will need to store to separate them is one bit to mark where polynomials end. Since we know that all coefficients in the polynomials are reduced modulo 4591, we only need 2^{13} bits to represent the actual coefficient, so with one extra bit to store where the polynomials end, we can represent each coefficient using 14 bits.

Algorithm 2 Algorithm to generate quotient list

Require: *dividend*, *divisor* pointers to univariate polynomials
  while dividend $\neq 0$ **do**
    $dividend/divisor = q + r$
    $dividend = [r, [].append(q)]$
    SWAP(dividend, divisor)
  end while

When the two polynomials have been reduced to two arrays containing subpolynomials that are the quotients from the divisions, the final inverse can be calculated. The first operation is to check whenever the last remainder is a constant, as this is required for the statement $GCD(a, b) = 1$ to hold. The constant is then extracted from the polynomial in order to free up space, and its modular inverse is calculated and saved for later. In the other array, the first subpolynomial can also be removed as it represents a remainder in which we are not interested. Since the construction of the inverse relies on subtraction and multiplications of polynomials, we know that the intermediate resultants will be growing for every step. To make sure that no information is lost, we first reverse all subpolynomials and then shift them to LSB. This ensures that a coefficient can only be overwritten during calculations after that coefficient has already been used in the calculation, thus making it safe to overwrite it. In order to calculate the value of the inverse, we first calculate $-q_0$ as the first step in Algorithm 1 is to calculate $s_2 = s_0 - qs_1$, where $s_0 = 0$ and $s_1 = 1$. The algorithm used to construct the inverse can be seen in Algorithm 3.

Algorithm 3 Algorithm to construct inverse

Require: t, old_t pointers to polynomials
  while number of polynomials > 2 **do**
    $old\_t = old\_t - quotient(old\_t) \times t$
    SWAP(t, old_t)
  end while

One thing to note is that this implementation is not constant-time. While constant-time would make the key generation more resistant to side-channel attacks, it is not necessarily required that the key generation is constant-time since it is an operation that is done infrequently, and side channel attacks often requires physical access to the device.

4.4 Benchmarking

Benchmarks are performed to measure performance and energy demand with these key generation algorithms on microcontrollers. The hardware platforms that are used are based on three different CPU architectures: ARM M0+, Xtensa LX7, and Atmega2560. The platforms also have varying amounts of RAM and ROM to reflect different kinds of IoT devices. The specifications for the selected hardware platforms can be seen in Table 4.1.

Platform	CPU	ROM	RAM	Clock speed
Arduino MEGA Rev3	ATmega2560	256 kB	8 kB	16 MHz
Arduino Nano ESP32	Xtensa LX7	384kB	512kB	240 MHz
Raspberry Pi Pico W	ARM Cortex M0+	2 MB	264kB	133 MHz

Table 4.1: Hardware specifications for the chosen IoT-devices.

4.4.1 Amperage and Runtime

The setup for the test performs key generation with one of the schemes on the device. After a 100 ms delay the relevant memory space is reset, and the scheme is run again. The function used for Nano was `delay()`, the Pico test used `sleep_ms()`, and MEGA used `delay()`. This is repeated 100 times to give statistically sound average values.

Before each run a pin on the board is set to high and after each run the same pin is set to low. This is done to facilitate the separation of the data from the cryptosystem run from other functions, and to facilitate the identification of separate runs.

Amperage is measured with the Nordic Semiconductors Power Profiler Kit II (PPK2) as it has been specifically developed to measure the power consumption of low-energy systems such as IoT devices [47]. The PPK2 is connected to the indicator pin, ground

pins, and VCC pins of the microprocessor. The setups are shown in Figure 4.1-Figure 4.3. The PPK2 also provides time stamps at each sample which can be used to calculate runtime.

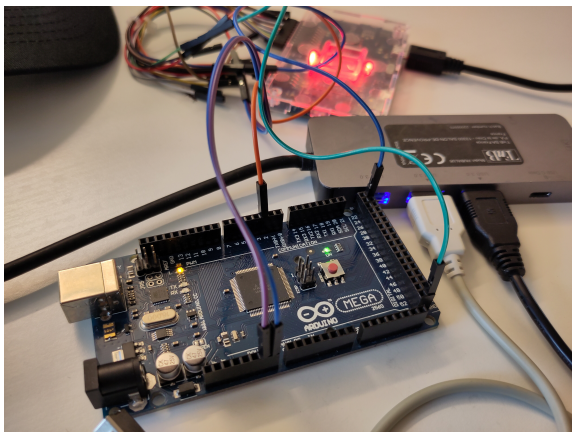

Figure 4.1: Arduino MEGA connected to PPK2.

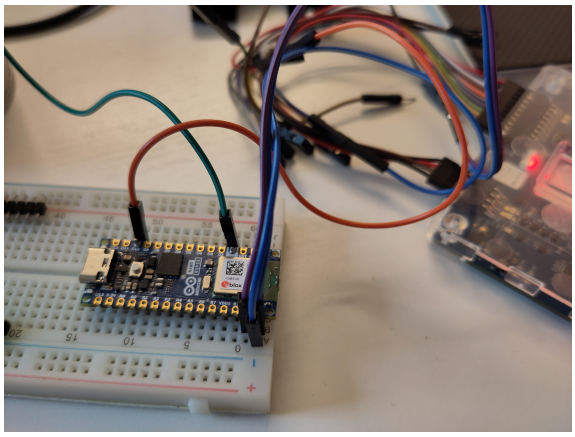

Figure 4.2: Arduino Nano ESP32 connected to PPK2.

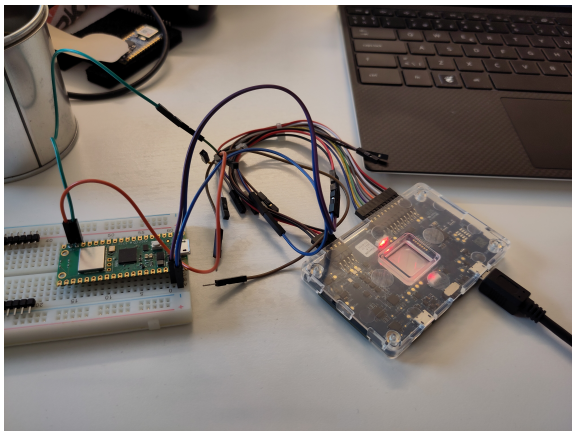

Figure 4.3: Raspberry Pi Pico W connected to PPK2.

4.4.2 Memory Consumption

To measure the amount of used memory for each platform, there are some core differences that needs to be taken into account. The Arduino Nano ESP32 is the only device that runs on the Espressif IoT Development Framework (ESP-IDF) which in turn also runs on the FreeRTOS kernel. This means that the Arduino Nano ESP32 have access to a more advanced API that can record the stack and heap size of running processes. Another important difference is that in FreeRTOS, every task has its own stack, so using the built-in `xTaskCreate()` to create a new task with a pre-defined stack size, together with the FreeRTOS function `uxTaskGetStackHighWaterMark()` to get where the stack was at its highest, this method should give a quite good estimate on how much memory is being used.

Raspberry Pi Pico W and Arduino Mega 2560 on the other hand, lacks these API calls, but are on the other hand not powered by FreeRTOS. This means that while we do not get any functions to easily track the stack of our functions, the system has no underlying software that defines different stack areas or such, we only get a global stack. To track the total stack usage we decided to use a technique where we first fill the stack with a known value or pattern, then run our program and then check at what address the stack was at its largest. This way we can see where the stack was at its highest point.

4.5 Data Analysis

Analysis of the collected data is performed with a python script which run through the csv files generated by the PPK row by row. At each row the script checks the state of a pin which indicates if the cryptosystem is running. At all rows where the pin is high, the amperage is added to the variable `current` and the variable `counter` is increased by 1. If the pin is low nothing happens. If the pin is low and `active` is true, `active` is set to false and the calculation stage is performed. The runtime, approximate clock cycle count, average amperage, average power, and total energy is then saved to their respective arrays.

4.5.1 Calculating Runtime

If the pin is high and the bool `active` is false, `active` is set to true and the time stamp is saved to `int time_start`. The current time-stamp is subtracted by `time_start` to give the **total runtime** of the cryptosystem iteration. The total runtime is then multiplied by the clock frequency of the device to provide an approximate amount of **clock cycles** used by the cryptosystem iteration

4.5.2 Calculating Energy Consumption

The base amperage of the devices was decided by looking at the amperage during the wait state of the devices between the iterations of the cryptosystems. Mega required some extra work because in contrast to the Pico and Nano devices the standard delay function caused a busy waiting loop which had a higher amperage than the actual

operations. Because of this, `avr/sleep_cpu()` from the `avr-libc` library was used to identify base amperage for MEGA.

The `current` is divided by `counter` to give the average total amperage. The average total amperage is then subtracted by the base amperage of the device in wait mode. The result is multiplied by 1×10^6 to get the **average amperage** in ampere consumed by executing the cryptosystem on the device:
$I_{avg} = (\frac{I_{total}}{sample\_count} - I_{base}) \times 10^6$. The average amperage is then multiplied by 5 V which is the voltage provided by the PPK to get **average power** in Watt, and the average power is multiplied by the total runtime to get the **total energy** consumed by the execution in Joule:
$Energy = I_{avg} \times 5V \times t_{run}$.

5

Results

We present in this chapter the results of the benchmarks and approximated clock cycle count. The average runtime and approximate clock cycles are presented in Section 5.1. The average energy consumption results of each benchmark are presented in Section 5.2. Lastly, the memory consumption of each benchmark is presented in Section 5.3.

5.1 Performance

In our first set of experiments, we compare the performance of RSA, Curve25519, FourQ, and SNTRUP on the Arduino Mega, Arduino Nano ESP32, and Raspberry Pi Pico W. Table 5.1 shows the execution times, and Table 5.2 show the approximate cycle count.

The average time required to perform an RSA key generation on the Arduino MEGA is slightly more than 4 minutes, which is too long to be viable in most circumstances. The Arduino Nano ESP32 outperforms the other devices on all cryptosystems, although the difference between the Arduino Nano ESP32 and the Raspberry Pi Pico W is relatively small for the Curve25519 key generation. The difference between the Arduino Nano ESP32 and the Raspberry Pi Pico W is slightly less when considering clock cycles, although the Arduino Nano ESP32 still outperforms the Raspberry Pi Pico W when running most cryptosystems. A noticeable exception is Curve25519, where the Raspberry Pi Pico W requires less clock cycles.

Our custom implementation of SNTRUP takes less than a minute per iteration to execute on the Arduino MEGA. This is less than a fourth of the execution time of RSA on the same device. The custom implementation of SNTRUP is also significantly faster than the reference implementation on both the Arduino Nano ESP32 and Raspberry Pi Pico W.

The performance of the cryptosystems can also be measured by how many clock cycles it takes to perform all its operations. The values are approximated as $t * f$ where t is the runtime of the key generation and f is the clock frequency of the device. The Arduino MEGA requires significantly more clock cycles than the other two devices, while the Arduino Nano ESP32 is the most efficient for all cryptosystems except Curve25519.

	Platform		
Algorithm	MEGA	Nano ESP32	Pico W
RSA (Mbed TLS)	NA	1396.064	45 726.231
RSA (this work)	248 667.526	NA	NA
Curve25519	7810.367	146.809	167.383
FourQ	5233.187	2.356	6.4415
SNTRUP (reference)	NA	1252.626	5669.054
SNTRUP (this work)	54 303.78	233.513	581.548

Table 5.1: Average execution times in milliseconds of the key-generation of the cryptographic systems measured across the different platforms.

	Platform		
Algorithm	MEGA	Nano ESP32	Pico W
RSA (Mbed TLS)	NA	335.055	6081.589
RSA (this work)	3978.68	NA	NA
Curve25519	124.966	35.234	22.262
FourQ	83.731	0.566	0.857
SNTRUP (reference)	NA	300.630	753.984
SNTRUP (this work)	868.860	56.043	77.346

Table 5.2: Approximated average clock cycle count in Megacycles used by the cryptosystems on the different platforms.

5.2 Energy Consumption

As shown in Table 5.3 FourQ consumes by far the least energy of all the tested cryptosystems, with Curve25519 consuming more energy but still significantly less than RSA and the reference implementation of SNTRUP. Arduino Nano ESP32 is the most energy efficient device, consuming roughly half the energy for RSA and FourQ, and about two thirds the energy for the reference implementation of SNTRUP compared to Raspberry Pi Pico W. Only Curve25519 has similar energy consumption on the two devices, see Figure 5.1. On the other hand, the Arduino MEGA device requires magnitudes more energy than both the Arduino Nano Esp32 and the Raspberry Pi Pico.

	Platform		
Algorithm	MEGA	Nano ESP32	Pico W
RSA (Mbed TLS)	NA	152.718	3311.93
RSA (this work)	13 259.95	NA	NA
Curve25519	361.055	15.946	13.920
FourQ	262.187	0.281	0.553
SNTRUP (reference)	NA	138.234	181.345
SNTRUP (this work)	2292.499	22.613	49.528

Table 5.3: Average energy consumption in milliJoules of the cryptosystems on the different platforms.

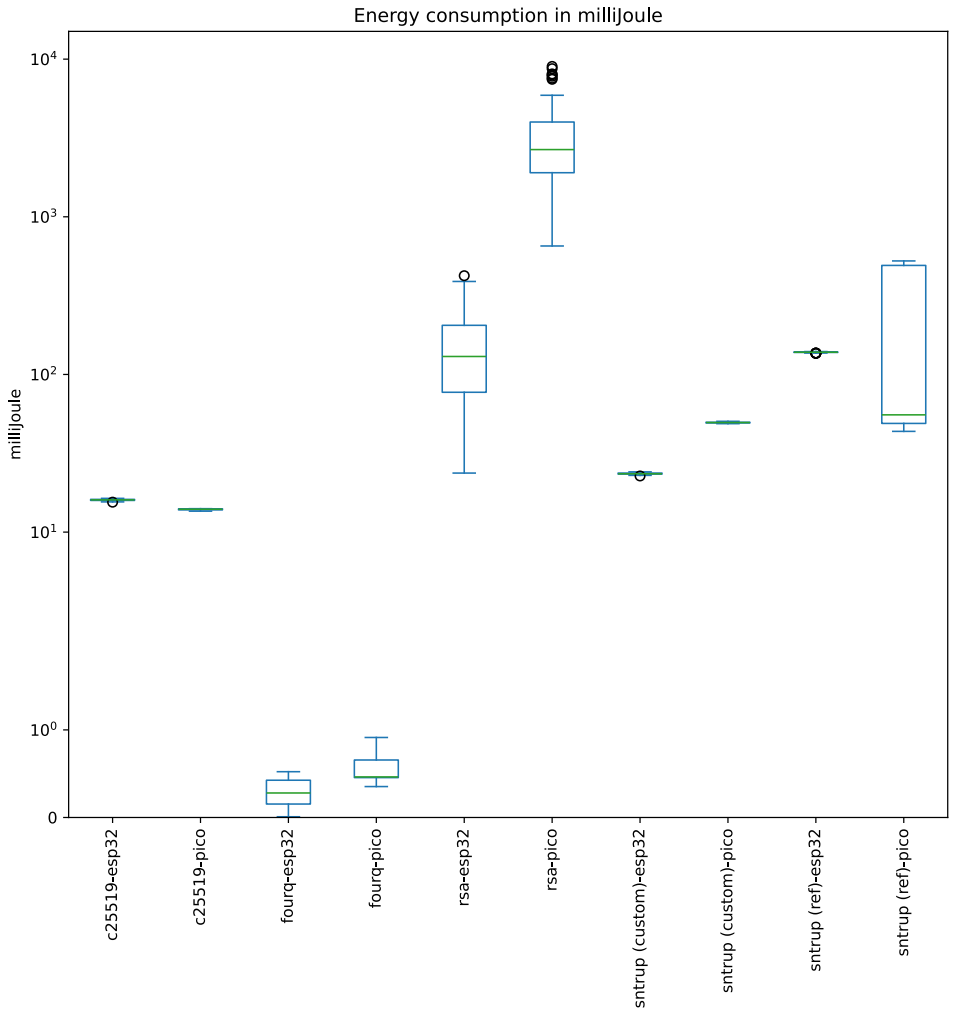

Figure 5.1: Energy consumed in milliJoule by the algorithm implementations on Raspberry Pi Pico W and Arduino Nano ESP32.

5.3 Memory Consumption

In our final experiment, we examine the memory footprint of each of the algorithms. Table 5.4 shows that the memory consumption of the Arduino MEGA and Raspberry Pi Pico W tend to be similar, while the Arduino Nano ESP32 typically uses more memory. Curve25519 for example uses more than twice as much memory on the Arduino Nano ESP32 than the Arduino MEGA and Raspberry Pi Pico W.

	Platform		
Algorithm	MEGA	Nano ESP32	Pico W
RSA (Mbed TLS)	NA	4220	3392
RSA (this work)	7883	NA	NA
Curve25519	510	1364	640
FourQ	1322	2708	2500
SNTRUP (reference)	NA	20272	16440
SNTRUP (this work)	5567	7380	5416

Table 5.4: Memory usage (in bytes) during key-generation on the different platforms.

6

Discussion

In this chapter, both the results of the benchmarks and the cryptosystems are discussed and compared. Furthermore, the risks, ethical considerations, and limitations are presented and discussed. The results are discussed in Section 6.1. Risks and ethical considerations are covered in Section 6.3. Lastly, the limitations are covered in Section 6.2.

6.1 Results

The results show that the difference between the devices are pretty consistent, with Arduino Nano ESP32 outperforming the other devices across all cryptosystems, except for Curve25519 when comparing memory consumption. This is to be expected since the Arduino Nano ESP32 is a more powerful device than the Raspberry Pi Pico W and even more so compared to the Arduino MEGA. The Arduino MEGA is by far the least powerful, taking orders of magnitude more time and power to execute any given cryptosystem. In contrast, Raspberry Pi Pico W and Arduino Nano ESP32 are relatively close for most cryptosystems, with Arduino Nano ESP32 slightly outperforming Raspberry Pi Pico W. The two major exceptions are RSA where Raspberry Pi Pico W performs much worse, and Curve25519 where Raspberry Pi Pico W slightly outperforms the Arduino Nano ESP32.

Comparing the key-generation of the different cryptosystems gives a pretty straightforward ordering from fastest and least power demanding. FourQ is by far the fastest, followed in order by Curve25519, NTRU Prime, and RSA. Our implementation of SNTRUP is significantly faster and less demanding than RSA, while the reference implementation is only slightly faster on the Arduino Nano ESP32. On the Raspberry Pi Pico W, the NTRU Prime reference implementation vastly outperforms RSA, while the relative difference is much less on the Arduino Nano ESP32.

The difference in memory consumption between Arduino Nano ESP32 compared with Arduino MEGA and Raspberry Pi Pico W may depend on several reasons. Firstly, the Arduino Nano ESP32 uses the framework ESP-IDF. This provides the environment with much more features, but at a cost of higher memory consumption. In comparison, the Raspberry Pi Pico C/C++ SDK provides a much more bare-bones API, with less background processes running by default. The Arduino Nano ESP32 consistently uses the most memory across all cryptosystems, although it is hard to find a consistent overhead added to that particular device. This might be due to the

difference in the technique used to measure memory, but it would be hard to make a fair assessment with the stack-filling technique used on Arduino Mega and Raspberry Pi Pico W since the Arduino Nano ESP32 is running on a real-time kernel.

Our Streamlined NTRU Prime implementation is significantly faster than the reference implementation. This is probably largely due to it not being constant time. However, since the key-generation is being run much more infrequently than the actual encryption and decryption, the need for it to be constant-time is not as important. It is hard to say if our implementation would be performing on par with the reference implementation since it is a lot less memory intensive, and memory operations take a lot of time. However, the amount of memory accesses has not been analyzed, and therefore no conclusions can be made. The results show that Streamlined NTRU Prime can fit in the flash memory and execute in less than a minute, even on an ATmega 2560. This shows that post-quantum secure cryptographic algorithms can be provided even on very weak IoT devices.

6.2 Limitations

The project does not cover: code-based cryptography, cryptanalysis, or private-key cryptography. Code-based cryptography schemes are very slow [12] and thus are ill-fitted for resource constrained IoT devices. The project relies on research done by other cryptanalysts to ensure that the algorithms are secure. No further cryptanalysis will be done in the project. NIST recently presented ASCON as the lightweight standard for private-key encryption and hashing. This makes further investigation into private-key cryptography a low priority at the moment.

When the project started, NIST had not yet finished its post-quantum competition. Streamlined NTRU Prime was a candidate, but eventually it lost out to the KYBER cryptographic suite. We picked SNTRUP as our algorithm because of its smaller key size than KYBER and because SNTRUP had already been in common use for some time.

The report also does not cover other aspects of IoT, such as architectural developments or changes in IoT network models. Architectural developments may change the performance of some algorithms noticeably, but the architectures benchmarked in the project cover three different architecture families that are described in Section 4.4. Changes to IoT network models can affect the relative isolation of end devices and their connections. However, there will still be cases where a communication channel may be accessible by an attacker and thus needs to be protected, or where another device on the same network can be infected and used to access the network.

Lastly, network security at large is not covered by the report. While the results may be of use when constructing future protocols, in particular for lightweight secure connections, they do not explore differences in anything else than the key generation algorithms. Other aspects of lightweight networking and lightweight cryptography are not within the scope of this report.

This report do not examine pseudo-random generators (PRGs) as many devices have

associated PRGs, there is no industry standard, and the development of PRGs are complicated enough for its own report.

6.3 Risk Analysis and Ethical Considerations

All the chosen algorithms have been deployed for some time and should be secure for now. Compromising key sizes or calculations in order to accommodate microprocessor constraints would weaken the security of the schemes. In the experimental settings of the project, there are no actual risks of security breaches because the devices will not be connected to the internet.

From an ethical perspective, the main issue is that the algorithms investigated in this report are dual-use technologies as defined by the Wassenaar Arrangement [48]. A dual-use technology can be used for both civilian and military purposes. If the technology ends up in the wrong hands, it might give malicious elements advantages. Enhanced security increases privacy and security for everyone, including those who use the technology for malicious purposes, such as (war) crimes. On the other hand, ensuring the integrity and confidentiality of home networks enhances both the security and safety of civilians. Another important benefit of IoT cryptography is that medical or industrial sensors can be kept secure, limiting vulnerability to attacks against hospitals, industry, or infrastructure.

7

Conclusion

In this chapter we present our conclusions to the research questions posed in Section 1.1 in Section 7.1. These conclusions are based on the results from Section 5. Key take aways from the results that do not directly answer a research question is presented in Section 7.2 This is followed by Section 7.3 where we discuss potential directions for future work based on the outcome of this study.

7.1 Addressing the Research Questions

Regarding **RQ1**: *"Does lattice-based or elliptic curve key generation schemes perform the best?"*, our benchmarks confirm that he elliptic curves are faster and uses less memory when generating keys than SNTRUP. However, the improvements that were implemented in this thesis proved to both reduce memory consumption and increase the speed of the algorithm when compared to the reference implementation. With a memory consumption of just 5416 bytes on the Raspberry Pi Pico W, compared to the 16440 bytes used by the reference implementation, **we managed to reduce memory usage by around 67%**. When we compare the execution times of the reference implementation and ours, we see a 5.36 speed-up on the Arduino Nano ESP32, and a 9.75 speed-up on the Raspberry Pi Pico W. So while elliptic curves might be the fastest and consume the least amount of memory, **it is certainly possible to generate post-quantum secure key pairs on resource-constrained IoT-devices**.

RQ2: *"What are the memory requirements of the algorithms, and how can they be reduced to fit on small devices?"* Curve25519 and FourQ both have small memory requirements and do not need any extra work to fit even on the Arduino MEGA. The RSA cryptosystem barely fit on the ATmega device, but much of its memory footprints comes from the key sizes. **The reference implementation of SNTRUP required a substantial amount of memory**, which had to be reduced significantly to run on the ATmega device. This was achieved by making inversion, EGCD computations, and other mathematical operations significantly more memory efficient. In the EGCD implementation in particular, division in place had a great impact on the memory consumption.

RQ3: *"How do the algorithms' performance vary depending on the processor architecture?"* is answered as such: Examining the execution of the same code on different devices shows a variance in relative execution time and energy consumption between the different schemes, which suggests that some operations may be heavier

on some architectures. One specific difference that can be seen is that **the relative difference between FourQ, Curve25519 and SNTRUP is much smaller on Arduino MEGA than on Arduino Nano ESP32 and Raspberry Pi Pico W**. Another major difference is how slow RSA is on Raspberry Pi Pico W. The MbedTLS library used is developed by ARM, which is the designer of the architecture used in the Raspberry Pi Pico W.

7.2 Summary of Results

The experiments showed that the elliptic curves did outperform SNTRUP in all aspects, with faster key generation, smaller memory footprint and lower energy consumption. In terms of speed, **FourQ is the fastest algorithm on all three devices**, with an average execution speed of just 0.281 ms on the Arduino Nano ESP32, 0.553 ms on the Raspberry Pi Pico W and 5233 ms on the Arduino Mega 2560. The runner-up was Curve25519 with execution times of 146.8 ms on the Arduino Nano ESP32, 167.4 ms on the Raspberry Pi Pico W, and 7810.4 ms on the Arduino Mega. Given that Curve25519 is the de-facto standard in many of the modern communication protocols used today, it is impressive to see that a fully functional key pair can be generated on average in under 8 seconds on an 8-bit microcontroller while only using 510 bytes of RAM. However, **RSA is not recommended**, as it is very slow, especially on the Arduino Mega and the Raspberry Pi Pico W.

The greatest variance between the different key generation algorithms can be seen in execution time, and consequently in the amount of clock cycles. When comparing FourQ to the implementation of SNTRUP proposed in this thesis, the increase is almost exactly 100 times slower on the Arduino Nano ESP32 and 90 times slower on the Raspberry Pi Pico W. However, the memory consumption did only increase by 2.7 times, and 2.17 times respectively. On the Arduino Mega 2560 however, the execution time is only 10 times slower, but the memory consumption rose with a factor of 4.21. This might have to do with the optimizations that were implemented in order to get FourQ to run on the 8-bit architecture, mentioned in Section 4.3.1.

The results for RSA are really inconsistent and shows no real correlating slowness on all devices. However, this is highly likely due to the design of the key generation algorithm itself. In the key generation process, two 1024-bit primes have to be found. Finding primes this large can be quite difficult, especially on devices that are resource-constrained. Since the problem of finding a (preferably very large) prime between 2 and 2^{1024} is non-deterministic and often is done in a probabilistic manner, it can take long time and many iterations might have to be done if unlucky randomness is encountered.

7.3 Future Work

This section provides some ideas on what can be done further with the results that were concluded in this report. The main focus will be the newly implemented version

of the key generation of Streamlined NTRU Prime that performed better than the reference implementation.

The implementation could be extended to include the entirety of the NTRU Prime crytosystem, not only the key generation. This would include both **encapsulation** and **decapsulation** in order to provide the whole key exchange mechanism. This would be an interesting topic to explore if there are other optimizations that can be done to that part of the cryptosystem as well. This topic can be even further expanded to include hybrid key exchange methods, such as the one that is the standard in OpenSSH, where a hybrid approach of NTRU Prime + ECDH with Cruve25519 is being used. With NTRU Prime keys now possible to generate on very memory-constrained devices, and the fact that Curve25519 have a very small memory footprint and fast execution time, it could be possible to bring this industry standard of key exchange to resource-constrained IoT-devices.

Another topic of research would be to make the implementation of the key generation algorithm proposed in this thesis constant time to improve security. One example of a function that is used frequently and that can be implemented to run on constant-time is the function that finds the modular inverse of a number given a modulo. This might prove to be hard since the landscape of IoT-devices are running on very fragmented hardware architectures, but it might be possible to at least implement constant time functions for the most common architectures.

Bibliography

[1] S. Sinha, *State of IoT 2023: Number of connected IoT devices growing 16% to 16.7 billion globally*, en-US, May 2023. [Online]. Available: https://iot-analytics.com/number-connected-iot-devices/ (visited on 01/18/2024).

[2] J. Fruhlinger, *What is IoT? The internet of things explained*, en, 2023. [Online]. Available: https://www.networkworld.com/article/963923/what-is-iot-the-internet-of-things-explained.html (visited on 01/18/2024).

[3] M. Antonakakis, T. April, M. Bailey, *et al.*, "Understanding the Mirai Botnet," en, ser. 26th USENIX Security Symposium, USENIX Association, 2017, pp. 1093–1110, ISBN: 978-1-931971-40-9. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/antonakakis (visited on 01/18/2024).

[4] E. Kovacs, *BASHLITE Malware Uses ShellShock to Hijack Devices Running BusyBox*, en-US, Nov. 2014. [Online]. Available: https://www.securityweek.com/bashlite-malware-uses-shellshock-hijack-devices-running-busybox/ (visited on 01/18/2024).

[5] D. Goodin, *"Disabling cyberattacks" are hitting critical US water systems, White House warns*, en-us, Mar. 2024. [Online]. Available: https://arstechnica.com/security/2024/03/critical-us-water-systems-face-disabling-cyberattacks-white-house-warns/ (visited on 05/11/2024).

[6] O. Williams-Grut, *Hackers once stole a casino's high-roller database through a thermometer in the lobby fish tank*, en-US, Apr. 2018. [Online]. Available: https://www.businessinsider.com/hackers-stole-a-casinos-database-through-a-thermometer-in-the-lobby-fish-tank-2018-4 (visited on 01/30/2024).

[7] W. J. Buchanan, S. Li, and R. Asif, "Lightweight cryptography methods," en, *Journal of Cyber Security Technology*, vol. 1, no. 3-4, pp. 187–201, 2017, ISSN: 2374-2917. DOI: 0.1080/23742917.2017.1384917. [Online]. Available: https://www.tandfonline.com/doi/epdf/10.1080/23742917.2017.1384917?needAccess=true (visited on 01/18/2024).

[8] K. McKay, L. Bassham, M. S. Turan, and N. Mouha, *Report on Lightweight Cryptography*, en, Mar. 2017. DOI: https://doi.org/10.6028/NIST.IR.8114. [Online]. Available: https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=922743.

[9] M. Sonmez Turan, K. McKay, D. Chang, *et al.*, "Status report on the final round of the NIST lightweight cryptography standardization process," en, National Institute of Standards and Technology (U.S.), Gaithersburg, MD, Tech. Rep.

NIST IR 8454, Jun. 2023, NIST IR 8454. DOI: 10.6028/NIST.IR.8454. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/ir/2023/NIST.IR.8454.pdf (visited on 01/16/2024).

[10] D. J. Bernstein, C. Chuengsatiansup, T. Lange, and C. van Vredendaal, "NTRU Prime: Reducing Attack Surface at Low Cost," en, in *Selected Areas in Cryptography – SAC 2017*, C. Adams and J. Camenisch, Eds., Cham: Springer International Publishing, 2018, pp. 235–260, ISBN: 978-3-319-72565-9. DOI: 10.1007/978-3-319-72565-9_12.

[11] C. Costello and P. Longa, *FourQ: Four-dimensional decompositions on a Q-curve over the Mersenne prime*, 2015. [Online]. Available: https://eprint.iacr.org/2015/565 (visited on 03/06/2024).

[12] T. M. Fernandez-Carames, "From Pre-Quantum to Post-Quantum IoT Security: A Survey on Quantum-Resistant Cryptosystems for the Internet of Things," en, *IEEE Internet of Things Journal*, vol. 7, no. 7, pp. 6457–6480, Jul. 2020, ISSN: 2327-4662, 2372-2541. DOI: 10.1109/JIOT.2019.2958788. [Online]. Available: https://ieeexplore.ieee.org/document/8932459/ (visited on 01/16/2024).

[13] R. Alvarez, C. Caballero-Gil, J. Santonja, and A. Zamora, "Algorithms for Lightweight Key Exchange," en, *Sensors*, vol. 17, no. 7, p. 1517, Jul. 2017, ISSN: 1424-8220. DOI: 10.3390/s17071517. [Online]. Available: https://www.mdpi.com/1424-8220/17/7/1517 (visited on 01/18/2024).

[14] *Nano ESP32 | Arduino Documentation*. [Online]. Available: https://docs.arduino.cc/hardware/nano-esp32/ (visited on 05/01/2025).

[15] R. P. Ltd, *Buy a Raspberry Pi Pico*, en-GB. [Online]. Available: https://www.raspberrypi.com/products/raspberry-pi-pico/ (visited on 05/01/2025).

[16] *Arduino Mega 2560 Rev3*, en. [Online]. Available: https://store.arduino.cc/en-se/products/arduino-mega-2560-rev3 (visited on 05/01/2025).

[17] *NIST Releases First 3 Finalized Post-Quantum Encryption Standards*, en, Aug. 2024. [Online]. Available: https://www.nist.gov/news-events/news/2024/08/nist-releases-first-3-finalized-post-quantum-encryption-standards (visited on 06/04/2025).

[18] B. Barak, *An Intensive Introduction to Cryptography*, en, Visited 2024-01-30, 2021. [Online]. Available: https://files.boazbarak.org/crypto/lnotes_book.pdf (visited on 01/30/2024).

[19] P. Shor, "Algorithms for quantum computation: Discrete logarithms and factoring," in *Proceedings 35th Annual Symposium on Foundations of Computer Science*, Nov. 1994, pp. 124–134. DOI: 10.1109/SFCS.1994.365700. [Online]. Available: https://ieeexplore.ieee.org/document/365700 (visited on 02/07/2024).

[20] M. Roetteler, M. Naehrig, K. M. Svore, and K. Lauter, *Quantum resource estimates for computing elliptic curve discrete logarithms*, arXiv:1706.06752 [quant-ph], Oct. 2017. DOI: 10.48550/arXiv.1706.06752. [Online]. Available: http://arxiv.org/abs/1706.06752 (visited on 02/07/2024).

[21] C. Dobraunig, M. Eichlseder, F. Mendel, and M. Schläffer, "Ascon v1.2: Lightweight Authenticated Encryption and Hashing," en, *Journal of Cryptology*, vol. 34, no. 3, p. 33, Jun. 2021, ISSN: 1432-1378. DOI: 10.1007/s00145-

021-09398-9. [Online]. Available: https://doi.org/10.1007/s00145-021-09398-9 (visited on 01/18/2024).

[22] J. Fruhlinger, *What is cryptography? How algorithms keep information secret and safe*, en, 2022. [Online]. Available: https://www.csoonline.com/article/569921/what-is-cryptography-how-algorithms-keep-information-secret-and-safe.html (visited on 01/18/2024).

[23] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," en, *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[24] W. Stallings, *Cryptography and Network Security (Principles and Practice)*, 7th ed. Harlow: Pearson Education Limited, 2018, pp. 330-334, ISBN: 978-1-292-15858-7.

[25] P. IoT, *Best LWC Algorithm: ECDH vs RSA Comparison*, en, Visited 2025-08-25, May 2024. [Online]. Available: https://protonestiot.medium.com/selecting-the-best-lwc-algorithm-for-your-iot-system-ecdh-and-rsa-fb8cb37ce5ac (visited on 08/25/2025).

[26] J. Hoffstein, J. Pipher, and J. H. Silverman, "NTRU: A ring-based public key cryptosystem," en, in *Algorithmic Number Theory*, G. Goos, J. Hartmanis, J. Van Leeuwen, and J. P. Buhler, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 267–288, ISBN: 978-3-540-64657-0. [Online]. Available: http://link.springer.com/10.1007/BFb0054868 (visited on 01/18/2024).

[27] J. M. Kleinberg and É. Tardos, *Algorithm Design*, 1st ed. Harlow: Pearson Education Limited, 2014, pp. 451-452, ISBN: 978-1-292-02394-6.

[28] W. Diffie and M. Hellman, "New directions in cryptography," *IEEE Transactions on Information Theory*, vol. 22, no. 6, pp. 644–654, Nov. 1976, ISSN: 1557-9654. DOI: 10.1109/TIT.1976.1055638. [Online]. Available: https://ieeexplore.ieee.org/document/1055638 (visited on 01/18/2024).

[29] M. Yasuda, T. Shimoyama, J. Kogure, and T. Izu, "Computational hardness of IFP and ECDLP," en, *Applicable Algebra in Engineering, Communication and Computing*, vol. 27, no. 6, pp. 493–521, Dec. 2016, ISSN: 1432-0622. DOI: 10.1007/s00200-016-0291-x. [Online]. Available: https://doi.org/10.1007/s00200-016-0291-x (visited on 08/25/2025).

[30] D. J. Bernstein, "Curve25519: New Diffie-Hellman Speed Records," en, in *Public Key Cryptography - PKC 2006*, vol. 3958, Series Title: Lecture Notes in Computer Science, Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 207–228, ISBN: 978-3-540-33851-2. DOI: 10.1007/11745853_14. [Online]. Available: http://link.springer.com/10.1007/11745853_14 (visited on 03/04/2024).

[31] P. L. Montgomery, "Speeding the Pollard and Elliptic Curve Methods of Factorization," *Mathematics of Computation*, vol. 48, no. 177, pp. 243–264, 1987, Publisher: American Mathematical Society, ISSN: 0025-5718. DOI: 10.2307/2007888. [Online]. Available: https://www.jstor.org/stable/2007888 (visited on 03/05/2024).

[32] *Openssh.com/txt/release-9.0*, Apr. 2022. [Online]. Available: https://www.openssh.com/txt/release-9.0 (visited on 05/01/2025).

[33] O. Regev, "On lattices, learning with errors, random linear codes, and cryptography," in *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*, ser. STOC '05, New York, NY, USA: Association for Computing Machinery, May 2005, pp. 84–93, ISBN: 978-1-58113-960-0. DOI: 10.1145/1060590.1060603. [Online]. Available: https://dl.acm.org/doi/10.1145/1060590.1060603 (visited on 03/08/2024).

[34] N. Howgrave-Graham, J. Silverman, and W. Whyte, "A Meet-In-The-Middle Attack on an NTRU Private Key," Jul. 2003. [Online]. Available: https://www.researchgate.net/publication/2906622_A_Meet-In-The-Middle_Attack_on_an_NTRU_Private_Key.

[35] C. A. Lara-Nino, A. Diaz-Perez, and M. Morales-Sandoval, "Elliptic Curve Lightweight Cryptography: A Survey," en, *IEEE Access*, vol. 6, pp. 72 514–72 550, 2018, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2018.2881444. [Online]. Available: https://ieeexplore.ieee.org/document/8536394/ (visited on 01/16/2024).

[36] R. Chaudhary, G. S. Aujla, N. Kumar, and S. Zeadally, "Lattice-Based Public Key Cryptosystem for Internet of Things Environment: Challenges and Solutions," en, *IEEE Internet of Things Journal*, vol. 6, no. 3, pp. 4897–4909, Jun. 2019, ISSN: 2327-4662, 2372-2541. DOI: 10.1109/JIOT.2018.2878707. [Online]. Available: https://ieeexplore.ieee.org/document/8515014/ (visited on 01/23/2024).

[37] V. Thakor, M. Razzaque, and M. Khandaker, "Lightweight Cryptography Algorithms for Resource-Constrained IoT Devices: A Review, Comparison and Research Opportunities," English, *IEEE Access*, vol. 9, pp. 28 177–28 193, 2021, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2021.3052867.

[38] D. A. Osvik, *Fast Embedded Software Hashing*, 2012. [Online]. Available: https://eprint.iacr.org/2012/156 (visited on 05/14/2024).

[39] H. Cheng, D. Dinu, and J. Großschädl, "Efficient Implementation of the SHA-512 Hash Function for 8-Bit AVR Microcontrollers," en, in *Innovative Security Solutions for Information Technology and Communications*, J.-L. Lanet and C. Toma, Eds., vol. 11359, Cham: Springer International Publishing, 2019, pp. 273–287, ISBN: 978-3-030-12941-5. DOI: 10.1007/978-3-030-12942-2_21. [Online]. Available: http://link.springer.com/10.1007/978-3-030-12942-2_21 (visited on 05/14/2024).

[40] M. Hutter and P. Schwabe, *NaCl on 8-Bit AVR Microcontrollers*, 2013. [Online]. Available: https://eprint.iacr.org/2013/375 (visited on 05/21/2024).

[41] H. Cheng, J. Großschädl, P. B. Rønne, and P. Y. A. Ryan, *A Lightweight Implementation of NTRUEncrypt for 8-bit AVR Microcontrollers*, en, Dec. 2019. [Online]. Available: https://csrc.nist.gov/CSRC/media/Events/Second-PQC-Standardization-Conference/documents/accepted-papers/grobschadl-lighteight-implmentation-NTRUE.pdf.

[42] H. Cheng, D. Dinu, J. Großschädl, P. B. Rønne, and P. Y. A. Ryan, "A Lightweight Implementation of NTRU Prime for the Post-quantum Internet of Things," en, in *Information Security Theory and Practice*, M. Laurent and T. Giannetsos, Eds., ser. Lecture Notes in Computer Science, Cham: Springer International Publishing, 2020, pp. 103–119, ISBN: 978-3-030-41702-4.

DOI: 10.1007/978-3-030-41702-4_7. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-030-41702-4_7#citeas.

[43] A. Boorghany, S. B. Sarmadi, and R. Jalili, "On Constrained Implementation of Lattice-Based Cryptographic Primitives and Schemes on Smart Cards," *ACM Trans. Embed. Comput. Syst.*, vol. 14, no. 3, 42:1–42:25, Apr. 2015, ISSN: 1539-9087. DOI: 10.1145/2700078. [Online]. Available: https://doi.org/10.1145/2700078 (visited on 08/25/2025).

[44] M. Monteverde, "NTRU software implementation for constrained devices," en, M.S. thesis, KU Leuven, Leuven, 2008. [Online]. Available: https://upcommons.upc.edu/server/api/core/bitstreams/363cefa1-95e8-49bf-9d08-325b58c3f61a/content (visited on 05/21/2024).

[45] P. Wouters, D. Huigens, J. Winter, and N. Yutaka, "OpenPGP," Internet Engineering Task Force, Request for Comments RFC 9580, Jul. 2024, Num Pages: 166. DOI: 10.17487/RFC9580. [Online]. Available: https://datatracker.ietf.org/doc/rfc9580 (visited on 05/25/2025).

[46] J. A. Donenfeld, "WireGuard: Next Generation Kernel Network Tunnel," en, in *Proceedings 2017 Network and Distributed System Security Symposium*, San Diego, CA: Internet Society, 2017, ISBN: 978-1-891562-46-4. DOI: 10.14722/ndss.2017.23160. [Online]. Available: https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/wireguard-next-generation-kernel-network-tunnel/ (visited on 05/25/2025).

[47] Nordic Semiconductor ASA, *Power Profiler Kit II*, en, Aug. 2022. (visited on 02/06/2024).

[48] Wassenaar Arrangement, *List of Dual-Use Goods and Technologies and Munitions List*, en-US, 2023. [Online]. Available: https://www.wassenaar.org/control-lists/ (visited on 01/24/2024).

A

Algorithms

A.1 Diffie-Hellman

Algorithm 4 Diffie-Hellman Key Exchange

Require: $g \in \mathbb{Z}_p, \mathbb{Z}_p$ ▷ $\mathbb{Z}_p$ is a finite field of prime order p
Ensure: $s = pk_B^{sk_A} = pk_A^{sk_B}$
$sk_A \leftarrow \$Z_p$ ▷ sk is the secret key
$pk_A = g^{sk_A} \bmod p$ ▷ pk is the secret key
$A : pk_A \rightarrow B$
$sk_B \leftarrow \$Z_p$
$pk_B = g^{sk_B} \bmod p.$
$B : pk_B \rightarrow A.$
$A : s = pk_B^{sk_A} \bmod p$
$B : s = pk_A^{sk_B} \bmod p$

A.2 Streamlined NTRU Prime

Algorithm 5 SNTRUP Key Generation by Bob

Require: p, q *prime* ▷ p, q, t are positive integers
Require: $t \geq 1, p \geq 3t, q \geq 32t + 1$
while g *not invertible in* $\mathcal{R}/3$ **do**
$g \leftarrow \$\mathcal{R}$ ▷ g is a small element
end while
$f \leftarrow \$\mathcal{R}$ ▷ f is a t-small element
$h = g/(3f) \in \mathcal{R}/q$ ▷ h is the public key

Algorithm 6 SNTRUP Encapsulation by Alice

$r \leftarrow \$\mathcal{R}$ ▷ r is a t-small element
$hr = h * r \in \mathcal{R}/q$ ▷ h is Bobs public key
Round each coefficient of hr to the nearest multiple of 3, producing $c \in \mathcal{R}$
$C||K = Hash(r)$ ▷ C is used for"key confirmation", K is the session key.
$ciphertext = C||c$

Algorithm 7 SNTRUP Decapsulation by Bob

$3fc = 3 * f * c \in \mathcal{R}/q$. ▷ f taken from 5, c taken from 6
Reduce each coefficient of $3fc \in \mathcal{R}/q$ modulo 3 to obtain a polynomial $e \in \mathcal{R}/3$.
$e/g = e * 1/g \in \mathcal{R}/3$. ▷ g taken from 5
Lift e/g to a small polynomial $r' \in \mathcal{R}$.
Calculate $c', C', K',$ and r' as in 6.
if r' is t-small $\wedge c' == c \wedge C' == C$ **then** K'.
else
    False.
end if

Algorithm 8 Memory efficient greatest common divisor

Require: `mpz_t` a, b
  while $b \neq 0$:
    $a = a \bmod b$
    if $a = 0$:
      swap(a, b)
      return
    $b = b \bmod a$

A.3 RSA Implementation

Algorithm 9 Memory Efficient Extended Euclidean Algorithm

Require: `mpz_t` d, b_tmp, x1, `uint32_t` a
  `uint32_t` b
  `mpz_t` $x0 = d$
  $x1 = b\_tmp/a$
  $a = b\_tmp \bmod a$
  $b = a$
  $a = x0$
  $x1 = 0 - 1 * b//a$
  $x0 = 1$
  while $a \neq 0$:
    $x0 = x0 - x1 * a/b$
    $b = b \bmod a$
    if $b = 0$:
      swap$(x0, x1)$
      swap(a, b)
      return
    $x1 = x1 - x0 * a/b$
    $a = a \bmod b$

Algorithm 10 Memory Efficient Least Common Multiple

Require: `mpz_t` r, a, b
  `efficient_gcd`(a, b)
  swap(r,a)
  $a = \text{EEPROM}(p)$
  $r = a/r$
  $a = \text{EEPROM}(q)$
  $r = r * a$

B

LWE

The concept of **learning with errors** (LWE) is an alternative lattice-based cryptosystem introduced by Oded Regev [33]. The cryptographic security of LWE relies on the difficulty of the Shortest Independent Vector (SIVP) and GapSVP problems, a system of equations for public and private keys, and on a vector of "errors" that is inserted during encryption. Decryption is done through iterative loops between Gaussian elimination and a quantum algorithm using a Closest Vector Problem-oracle and the quantum Fourier transform.

In particular, the cryptosystem works as follows: an LWE cryptosystem with security parameter n is parameterized by two integers m, p and a probability distribution χ on $\mathbb{Z}_p$. Choose a prime number $p \geq 2$ satisfying $n^2 \leq n \leq 2n^2$. Let $m = (1+\epsilon)(n+1)\ log\ p$ for an arbitrary constant $\epsilon > 0$. Take the probability distribution χ to be $\overline{\psi}_{\alpha(n)}$ for $\alpha(n) = o(1/(\sqrt{n}\ log\ n))$, i.e., $\alpha(n)$ is such that $\lim_{n\to\infty} \alpha(n) * \sqrt{n}\ log\ n = 0$.

Before transmitting a message to Bob, Alice randomly and uniformly chooses a **private key** $s \in \mathbb{Z}_p^n$. Next, she chooses m vectors $a_1, ..., a_m \in \mathbb{Z}_p^n$ and elements $e_1, ..., e_m \in \mathbb{Z}_p^n$ independently according to χ. Then the **public key** is given by $(\mathbf{a_i}, b_i)_{i=1}^m$ where $b_i = \langle \mathbf{a_i}, \mathbf{s} \rangle + e_i$. **Encryption** is done bit by bit in two steps. First, choose a random set S uniformly from all 2^m subsets of $[m]$. If the bit is 0, it is encrypted as $(\sum_{i\in s} \mathbf{a_i}, \sum_{i\in s} b_i)$. If the bit is 1, it is encrypted as $(\sum_{i\in s} \mathbf{a_i}, \lfloor \frac{p}{2} \rfloor + \sum_{i\in s} b_i)$. Bob **decrypts** the pair $(\mathbf{a}, b)$ as 0 if $b - \langle \mathbf{a}, \mathbf{s} \rangle$ is closer to 0 than to $\lfloor \frac{p}{2} \rfloor\ mod\ p$. Otherwise, it is decrypted as 1.

Given these parameters, the public key size is $O(mn \log p) = \tilde{O}(n^2)$ and the encryption process increases the size of a message by a factor of $O(n \log p) = \tilde{O}(n)$. The LWE cryptosystem as described by Regev requires key material to be distributed by some other method. This weakens one of the major advantages of a public key cryptosystem over a private key alternative.