5-2020

Performance Analysis of NIST Round 2 Post-Quantum Cryptography Public-key Encryption and Key-establishment Algorithms on ARMv8 IoT Devices using SUPERCOP

Sean Zakrajsek
St. Cloud State University, stzakrajsek@stcloudstate.edu

Performance Analysis of NIST Round 2 Post-Quantum Cryptography Public-key Encryption and Key-establishment Algorithms on ARMv8 IoT Devices using SUPERCOP

By

Sean Zakrajsek

A Thesis

Submitted to the Graduate Faculty of

St. Cloud State University

in Partial Fulfillment of the Requirements

for the Degree of

Master of Science

in Information Assurance

May, 2020

Thesis Committee:
Dennis Guster, Chairperson
Erich Rice
Mark Schmidt

Abstract

With tens of billions of new IoT devices being utilized, and the advent of quantum computing, our future and our security needs are rapidly changing. While IoT devices have great potential to transform the way we live, they also have a number of serious problems centering on their security capabilities. Quantum computers capable of breaking today's encryption are just around the corner, and we will need to securely communicate over the internet using the encryption of the future. Post-quantum public-key encryption and key-establishment algorithms may be the answer to address those concerns. This paper used the benchmarking toolkit SUPERCOP to analyze the performance of post-quantum public-key encryption and key-establishment algorithms on IoT devices that are using ARMv8 CPUs. The performance of the NIST round 2 algorithms were found to not be significantly different between the two ARMv8 devices.

Acknowledgement

A sincere thank you to my thesis committee, and all those at NIST who provided support and guidance. I am especially grateful to those at NIST who took their time to meet with me, and generously shared their PQC expertise.

Table of Contents

Chapter Page

List of Tables

Table Page

List of Figures

Chapter I: Introduction

Introduction

Today's network environment is seeing an exponential increase in the use of devices collectively known as the Internet of Things (IoT). These devices, such as sensors, medical devices, TVs, webcams, home thermostats, remote power outlets, lights, door locks, home alarms, etc., are all everyday objects that have been embedded with computing devices that allow them to send and receive data via the Internet, and also to be interconnected with each other. These IoT devices frequently operate on a Low power and Lossy Network (LLN), and their applications are often quite constrained. As production costs decrease, IoT devices are becoming more and more ubiquitous in all aspects of our lives.

Many of the IoT devices use Advanced RISC Machine(ARM) processors. ARM is an instruction set for processors that uses the reduced instruction set computing (RISC) architecture. ARM cores are used in products ranging from smartphones, digital cameras, handheld game consoles, and single-board computers such as the Raspberry Pi. ARM processors have improved cost, power consumption, and heat dissipation compared to processors used in desktop computers (Aroca & Gonçalves, 2012). According to ARM Holdings CEO Simon Segars, in 2017 there has been 100 billion ARM-based chips shipped since 1991. ARM estimates that it will only take 5 years for them to ship the next 100 billion ARM-based chips, leading to their prediction that IoT devices will skyrocket into the hundreds of billions (Hughes, 2017).

While IoT has great potential, it also has a number of serious problems centering on the security capabilities of the connected devices. Post-quantum public-key encryption and key-establishment algorithms may be the answer to address those concerns. Currently, there are algorithms for post-quantum public-key encryption and key-establishment that are

undergoing the second round of National Institute of Standards and Technology (NIST) evaluation. Of the 26 algorithms that made it to the second round, 17 of them are public-key encryption and key-establishment algorithms. “In 2020, NIST plans to either select finalists for a final round or select a small number of candidates for standardization” (Alagic et al., 2019, p.18).

Problem Statement

With the tens of billions of new IoT devices being utilized in the near future, they will need to be able to securely communicate over the internet using the encryption of the future. Tomorrow’s encryption algorithms must be able to function on today’s IoT devices.

Nature and Significance of the Problem

Security for today's IoT devices is actually a multifaceted issue. One of the most pressing issues is its lack of public-key encryption and key-establishment schemes, which leaves all the devices vulnerable to attack from hackers. “If one thing can prevent the Internet of things from transforming the way we live and work, it will be a breakdown in security” (Oxford dictionary, example sentence). This prophetic quote not only provides an eloquent warning, it perfectly sums up the significance of having adequate security for IoT devices. While adding current public-key encryption will be an acceptable short term security solution, it will fail to be proactive in addressing future security needs. In the not too distant future, the advent of quantum computers capable of breaking today’s encryption will bring new security challenges, and left unaddressed, will render IoT devices vulnerable once again.

It is imperative that those responsible for providing security for IoT Devices be forward thinking, and take into account how technologies will likely evolve. In order for IoT security to possess long term viability, algorithms must either be initially designed with post quantum security, or the applications must allow for algorithms to be easily replaced with ones that do

have post-quantum security (McKay et al., 2016). Public key cryptography can still be considered for inclusion if it can meet the necessary conditions, which include being robust against quantum attacks, and using a combination of general public key cryptographic schemes along with lightweight primitives, such as lightweight hash function. Unfortunately, “because the majority of modern cryptographic algorithms were designed for desktop/server environments, many of these algorithms cannot be implemented in the constrained devices used by these applications” (McKay et al., 2016, p. iii). Either new algorithms must be created, or existing algorithms must be modified to fit in the constrained devices as well as meet the new standards for post-quantum public-key encryption and key-establishment. The performance of these algorithms need to be tested in other environments as well as the desktop/server environment.

Objective of the Study

The objective for this study is to use the SUPERCOP toolkit to collect performance data on post-quantum public-key encryption and key-establishment algorithms that are running on IoT devices using ARMv8 CPU architecture. The performance metrics collected for the key-encapsulation mechanisms include the number of cycles for generating a key pair, encapsulation of keys, and decapsulation of keys. The performance metrics collected for public-key cryptosystems include cycles to generate a key pair, to encrypt a short message, and to decrypt a short message.

Study Questions/Hypothesis

There will be no significant difference between the Raspberry Pi 3B+ and the Raspberry Pi 4 ARMv8 devices when measuring the benchmarked performance of public-key encryption and key-establishment algorithms using SUPERCOP.

Limitations of the Study

This study is limited by the compatibility of the different post-quantum algorithm implementations with the SUPERCOP benchmarking toolkit. Not all of the post-quantum algorithm implementations in the second round of the NIST competition have public-key cryptosystems available for benchmarking in the SUPERCOP toolkit. Another limitation of this study is the C compiler compatibility to compile the post-quantum algorithm implementations. More on the compiler errors can be viewed in Appendix A.

Definition of Terms

Internet of Things (IoT) - There is no universally-accepted definition that exists for IoT. In the NIST special publication titled “Network of ‘Things’”, the author describes the underlying foundations for IoT without defining the IoT. He claims “That is useful since there is no singular IoT, and it is meaningless to speak of comparing one IoT to another” (Voas, 2016, p. 1). He describes Primitives as building blocks to describe the IoT, which will allow for comparisons between the different IoT’s.

> Primitives offer a unifying vocabulary that allows for composition and information exchange among differently purposed networks. They offer clarity regarding concerns that are subtle, including interoperability, composability, and continuously-binding assets that come and go on-the-fly…. This model does not specify a definition for what is or is not a ‘thing'. Instead, we consider that each primitive injects a behavior representing that ‘thing’ into a NoT’s(IoT) workflow and dataflow. ‘Things’ can occur in physical space or virtual space. (Voas, 2016, p. 1)

The primitives that are defined for the IoT are sensors, aggregators, communication channels, external utilities, and decision triggers. These make up the core components for the Internet of Things.

Low power and Lossy Network (LLN) - A network of embedded devices that have limited power, memory, and processing capability.

ARM - Advanced RISC Machine is a family of reduced instruction set computing (RISC) architectures for computer processors.

RISC - Reduced Instruction Set Computer is a computer that has a small set of simple and general instructions.

Post-Quantum Cryptography - Cryptographic algorithms that are resistant to attacks from quantum computers.

Public-key Encryption - Cryptographic system that utilizes a pair of keys, one public and one private. To ensure security, only the private key is required to be kept secret. The public key can be openly distributed without compromising security.

SUPERCOP - System for Unified Performance Evaluation Related to Cryptographic Operations and Primitives. SUPERCOP is a toolkit developed for measuring the performance of cryptographic software.

Summary

Today's IoT devices are very vulnerable to attack from hackers. While current public-key encryption will provide immediate protection, and should be standard on all network connected devices, it will not address future security needs that will arise with the advent of the quantum computer. The number of new IoT devices is going to be in the billions. They will need to be able to securely communicate over the internet using post-quantum encryption. Tomorrow's encryption algorithms must be able to function on today's IoT devices. The primary objective for

this study is to analyze the performance of post-quantum public-key encryption on IoT devices that are using ARM CPUs.

Chapter II: Background and Review of Literature

Introduction

This chapter will focus on the background information and literature reviewed in this study, as well as literature that is related to the methodology used. The articles published to date discuss the importance of having encryption on IoT devices, but are lacking in detailed information regarding the topic of this paper. Topics that this chapter covers include IoT guidelines, why devices should be made powerful enough to use encryption, introduction to the NIST algorithms, and literature related to the methodology used.

Background and Literature Related to the problem

According to Hewlett Packard's 2015 report "Internet of Things research study," 70 percent of IoT devices did not encrypt communications to the Internet and local network, and 60 percent did not use encryption when downloading software updates. These numbers were pulled from testing 10 devices that they believed to be a good indicator for the IoT market at that time. It's a good bet that IoT security has changed very little since then.

By the year 2020, Cisco predicts there will be over 50 billion IoT devices connected to the internet, while Gartner, a global research and advisory firm, more modestly predicts the internet of things will be closer to 20 billion units. While there is no clear consensus on the exact number of IoT devices that will be in use by 2020 and beyond, the range for the predictions of total number devices is consistently in the tens of billions. That makes for a lot of vulnerable devices that are connected to the internet.

> As IoT data travels through multiple hops in a network, a proper encryption mechanism is required to ensure the confidentiality of data. Due to a diverse integration of services, devices and network, the data stored on a device is vulnerable to privacy violation by compromising nodes existing in an IoT network.

> The IoT devices susceptible to attacks may cause an attacker to impact the data integrity by modifying the stored data for malicious purposes. (Khan & Salah, 2018, p. 397)

If you don't use encryption in your IoT environment, then you may get hacked.

In 2019, NIST came out with a list of guidelines called the "Core Cybersecurity Feature Baseline for Securable IoT Devices: A Starting Point for IoT Device Manufacturers". These guidelines were developed to help promote the best available practices for mitigating risks to IoT devices. Topics covered by the guidelines include the common risk mitigation area of Data Protection, which includes protecting data in transmission from unauthorized access and modification. These are just recommendations from NIST, and are not regulations that have to be followed.

There are several potential barriers to using encryption on IoT devices. "Encryption can protect sensor data transmission integrity and confidentiality including cloud-to cloud communication, but it might render the IoT sensors unusable due to excessive energy requirements" (Voas, 2016, p. 22). In order to make sure that a device is capable of using encryption, it needs to be designed with a powerful enough CPU and have enough energy to power it.

> Many sensor networks depend on the timely transmission of sensor data to aggregators or other controllers. Any delay of sensor data — especially in time-critical applications such as CO alarms — due to latency can have serious consequences or can render the sensor data useless. Security solutions (e.g., device authentication, encryption) applied to sensor networks may introduce latency. (Cichonski et al., 2019, p. 29)

Devices need to be designed with powerful enough CPUs because encryption is a processing-intensive operation. “Be forward-looking and size hardware resources for potential future use. As an example, if a device has a 10-year lifespan, it may be necessary to update the encryption algorithm or key length the device uses, and the new algorithm or key length may make encryption more processing-intensive” (Fagan et al.,2019, p. 14).

As we look towards the post-quantum future, encryption processing needs will increase. With this increase, the devices that we have today may not be powerful enough to perform cryptography. That is why when we design an IoT device today, we need to keep the resources of the hardware in mind.

> Select or build a device with sufficient hardware resources (e.g., processing, memory, storage, network technology, power), as well as firmware and software resources, to support the desired features. For example, encryption is processing-intensive, and a device with limited processing might not be able to support encryption that customers need. Some devices cannot support the use of an operating system or Internet Protocol (IP) networks. (Fagan et al., 2019, p. 14)

Making a device powerful enough to use encryption can be difficult when the device is battery-powered. When a device runs on a battery, the most important thing is keeping the power draw low, and in particular low standby power, as devices can be asleep for minutes or hours before waking up briefly. The power needed for the device to use encryption may be unattainable, or may unacceptably shorten the battery life.

Currently NIST is on round 2 for the selection of the standards for post-quantum cryptography. Round 3 will begin sometime in 2020 or 2021. NIST considers the cost and performance of the algorithms to be the second most important characteristic for selecting the next standards. Memory requirements and computational efficiency are both considerations in

the cost of the algorithm. “NIST has completed preliminary efficiency analysis of the post-quantum public-key encryption algorithms on the reference platform, an Intel x64 running Windows or Linux and supporting the GNU Compiler Collection (GCC) compiler” (Alagic et al., 2019, p. 5). NIST has only been considering the performance of these algorithms using an Intel x64 CPU and not any ARM processors. This is one area that is lacking an efficiency analysis for these algorithms.

Out of the starting 82 candidate public-key encryption and digital signature algorithms, only 26 remain in the second round of the NIST Post-Quantum Cryptography Standardization Process. The 17 Second-Round Candidate public-key encryption and key-establishment algorithms include; BIKE, Classic McEliece, CRYSTALS-KYBER, FrodoKEM, HQC, LAC, LEDAcrypt, NewHope, NTRU, NTRU Prime, NTS-KEM, ROLLO, Round5, RQC, SABER, SIKE, and Three Bears. The 9 digital signature candidate algorithms in the second round include; CRYSTALS-DILITHIUM, FALCON, GeMSS, LUOV, MQDSS, Picnic, qTESLA, Rainbow, and SPHINCS+.

Literature Related to the Methodology

To compare other partially related methodologies, the research paper on “A Comprehensive Evaluation of Cryptographic Algorithms: DES, 3DES, AES, RSA and Blowfish,” was examined. In it, the authors go over the different evaluation parameters that they used in their experiment. In this project, the parameters that will undergo analysis include encryption, decryption time, and memory used. Encryption time is the time it takes for the encryption algorithm to convert plaintext into ciphertext. Decryption time is the opposite, the ciphertext is converted back into plaintext. The encryption and decryption time will be measured in milliseconds. This time will affect the performance of the system. The different algorithms will use different key sizes, number of operations done by the algorithm, initialization vectors used

and type of operations. This leads to requiring different memory sizes for implementation of the different algorithms (Patil et al., 2016).

Summary

This chapter reviewed IoT guidelines, discussing the common risk mitigation area of Data Protection. It also examined the roadblocks to using encryption on IoT devices, with lack of enough power being the biggest obstacle. The 17 NIST post-quantum public-key encryption algorithms were introduced, and although somewhat limited, the available literature related to the methodology was reviewed.

Although the articles published to date do show a robust discussion about the importance of having encryption on IoT devices, there is a lack of detailed information regarding the topic of this paper, which is focused on analyzing the performance of post-quantum public-key encryption algorithms on IoT devices that are using ARM CPUs.

Chapter III: Methodology

Introduction

This chapter will provide an explanation of the methodology used and the procedures applied in order to achieve the objectives of this research. The first section covers aspects of how the study was designed, including a discussion of the data collected and the hardware used. The next section focuses on the algorithm implementations that were used in this study, and it includes an explanation of the selection criteria used to choose the algorithms tested. The last section of this chapter focuses on the benchmarking toolkit SUPERCOP, and its use in this research.

Design of the Study

This study used a quantitative approach to determine how well each of the post-quantum public-key encryption and key-establishment algorithms, found in the second round of NIST standardization, performed on chosen hardware. Due to the large number of different implementations that each of the algorithms have, this paper only focused on the performance of one of the implementations per post-quantum algorithm. If none of the implementations for one of the algorithms were able to compile on the chosen device and operating system, then that algorithm was not included in the study. The operating system and processor architecture were chosen to maximize the number of working implementations for analysis.

The implementation chosen for each of the post-quantum schemes were the optimized versions if they were available. If an optimized version was not available, then the reference implementation was used instead. If there were multiple versions with different key lengths, then the implementation with the shortest key was chosen. The implementations of the post-quantum schemes were integrated into the benchmarking toolkit SUPERCOP.

Data Collection

Data collection was conducted using the benchmarking toolkit SUPERCOP version 20191221. This toolkit includes key-encapsulation mechanisms for all 17 of the post-quantum public-key encryption and key-establishment algorithms. It also includes several of the public-key cryptosystems. All of the Key-Establishment Mechanisms (KEM) data was collected using the SUPERCOP toolkit ECRYPT Benchmarking of Asymmetric Systems (eBATS). Performance measurements for KEM that SUPERCOP collects are listed on the bench.cr.yp.to website. (Bernstein, n.d.)

- Time in cycles to generate a key pair - the secret key and the corresponding public key.
- Time for encapsulation - time to compute a ciphertext and corresponding session key.
- Time for decapsulation - the time to compute the session key from the ciphertext.
- Space in bytes for a public key.
- Space in bytes for a ciphertext.
- Space in bytes for a session key.

Performance measurements for the public-key cryptosystems that SUPERCOP collects are listed on the bench.cr.yp.to website. (Bernstein, n.d.)

- Time in cycles to generate a key pair - the secret key and the corresponding public key.
- Time to encrypt a short 59 byte message.
- Time to decrypt a short 59 byte message
- Space in bytes for a secret key.
- Space in bytes for a public key.
- Ciphertext length for a 0-byte message.
- Ciphertext overhead for a 23-byte message.
- Ciphertext overhead for a long message.

Hardware and Operating System Environment

This study used both the Raspberry Pi 3 B+ and Raspberry Pi 4 to conduct the performance tests. The operating system used was the current version of Ubuntu 19.10. This allowed both the Raspberry Pi 3B+ and Raspberry Pi 4 to run the ARMv8 instruction set architecture. The ARMv8 architecture introduced the 64 bit execution state AArch64, making it compatible with the benchmarking toolkit SUPERCOP (Arm Ltd, n.d.).

The first ARMv8 device to be tested was the Raspberry Pi 3B+. It has the Broadcom BCM2837B0 quad-core A53 processor running at 1.4 GHz as well as 1 GB of RAM. The Raspberry Pi 3B+ is shown below in Figure 1.

Figure 1. Raspberry Pi 3 B+

The other system on a chip (SoC) that was tested was the Raspberry Pi 4. It has a Broadcom BCM2711, Quad coreCortex-A72 processor running at 1.5GHz. It also has twice as much memory as the Raspberry Pi 3B+ at 2 GB of RAM. The Raspberry Pi 4 is shown in Figure 2 below.

Figure 2. Raspberry Pi 4

Summary

This chapter discussed how this study was designed, which data measurements were germane to collect, and important details about the hardware used. For the scope of this study, it is important to define why each of the implementations were chosen. The benchmarking toolkit SUPERCOP was very useful for testing out many different measurements related to

cryptographic performance. The toolkit came prepackaged with all of the latest post-quantum cryptographic algorithms needed for this study. Although the Raspberry Pi 3B+ and the Raspberry Pi 4 look practically identical on the outside, under the hood, the Raspberry Pi 4 possesses superior processing speed and RAM. The tests listed in this chapter put the hardware to the test to see if the algorithms would perform significantly different depending on which device was being used.

Chapter IV: Data Presentation and Analysis

Introduction

This chapter presents and analyzes the data collected by the methods mentioned in the previous chapter. First, the data that was collected will be presented in a series of tables and graphs. Next, the performance of each of the different key-encapsulation methods will be analyzed for both the Raspberry Pi 3B+ and Raspberry Pi 4. The last section of this chapter will focus on analyzing the performance of the public-key cryptosystems.

Data Presentation - Key-Encapsulation Mechanisms

The following tables include data collected from 14 of the 17 post-quantum key-encapsulation mechanisms using version 20191221 of SUPERCOP. There were 3 algorithms that had failed to compile while running the tests. More information on those 3 algorithms and why they failed can be viewed in Appendix A. For information on the compile time, primitive, implementation and the compiler used for the algorithm implementations that were able to successfully compile, see Appendix B. The tables for space size, in bytes, that each algorithm used for a secret key, public key, ciphertext, and session key, can be viewed in Appendix C. Each of the algorithm implementations in the tables are either the optimized versions, the version with the smallest key length, or the version that was able to compile. The following tables show the first quartile, median, and third quartile cycles it took to either generate a key pair, encapsulation, or decapsulation of keys for both of the devices tested. The values are the average of many speed measurements ("eBACS: ECRYPT Benchmarking of Cryptographic Systems", n.d.).

Table 1 and Table 2 show the cycles to generate a key pair for the Raspberry Pi 3B+ and Raspberry Pi 4 respectively. For both devices, the fastest key-encapsulation mechanism to generate key pairs was lightsaber and the slowest key-encapsulation mechanism to generate

key pairs was mceliece348864f. There was a massive difference between the fastest and slowest key-encapsulation mechanism tested. The fastest key-encapsulation mechanism, lightsaber, was over 4500 times faster than the slowest mceliece348864f for key generation. For both the RAspberry Pi 3B+ and the Raspberry Pi 4 there were stability issues with many of the Classic McEliece algorithm implementations generating keys. The key-encapsulation mechanism mceliece348864f was one of the only Classic McEliece algorithm implementations that did not have stability issues. The algorithm implementations that did have stability issues tended to produce a large variance between the first, median, and third quartile.

Table 1

Cycles to generate a key pair for KEM on Raspberry Pi 3b+

quartile	median	quartile	system
161220	161702	162019	lightsaber
194547	196753	205281	r5nd0kem2iot
266971	265331	268275	kyber512
283474	282306	286272	newhope512cca
331105	331725	332057	threebears624r2cpax
1505188	1517069	1536146	hqc1281
3944721	3995211	4012819	ntrukem443
21096658	21111815	21172881	sntrup653
21968557	21975555	21985111	frodokem640
53282599	101947929	151081615	bike2l1nc
112207782	112577462	112777600	ledakem13
148714473	161685593	190310821	ntskem1264
170332098	170273427	171121725	sikep503
735216332	741023508	745088028	mceliece348864f

Table 2

Cycles to generate a key pair for KEM on Raspberry Pi 4

quartile	median	quartile	system
100288	99373	99529	lightsaber
107115	108324	112041	r5nd0kem2iot
134446	133962	134134	kyber512
134772	134505	135136	newhope512cca
208662	209888	210125	threebears624r2cpax
1036922	1068988	1079539	ntrukem443
1175044	1183380	1206649	hqc1281
12367120	12385844	12442216	sntrup653
15911486	15944148	15981147	frodokem640
25234359	47522680	69974155	bike2l1nc
69931488	70185868	70424832	ledakem13
94252120	141133288	312818998	ntskem1264
144212647	144185373	144533570	sikep503
437367115	449889390	454446677	mceliece348864f

The next two tables show the number of cycles it took for encapsulation. This is the time it takes to compute a ciphertext and corresponding session key, given a user's public key. The fastest and slowest key-encapsulation mechanisms for key generation were the two fastest encapsulation schemes for both Raspberry Pi 3B+ and Raspberry Pi 4.

Table 3

Cycles for encapsulation on Raspberry Pi 3B+

quartile	median	quartile	system
228944	227909	228451	lightsaber
283309	295304	463256	mceliece348864f
288099	289925	298868	r5nd0kem2iot
330755	331240	332722	ledakem13
377220	378673	381677	kyber512
414093	414212	418681	threebears624r2cpax
440119	441561	453019	newhope512cca
450100	459282	478675	ntrukem443
508543	544320	607653	ntskem1264
727667	729533	735709	sntrup653
2211011	2229902	2249088	bike2l1nc
3024409	3046613	3069753	hqc1281
24180080	24185866	24209172	frodokem640
280446444	280476545	281379281	sikep503

Table 4

Cycles for encapsulation on Raspberry Pi 4

quartile	median	quartile	system
142244	149368	154765	mceliece348864f
143773	143444	143490	lightsaber
160110	161077	166302	r5nd0kem2iot
179223	180000	180322	kyber512
210296	208937	209215	ledakem13
210882	210550	211079	newhope512cca
214501	217742	249918	ntrukem443
258894	261057	261494	threebears624r2cpax
424221	459114	523020	ntskem1264
482848	487049	488390	sntrup653
1343338	1359618	1442061	bike2l1nc
2342002	2368109	2415056	hqc1281
18255132	18356963	18388720	frodokem640
237423418	237557045	238075832	sikep503

The next two tables show the number of cycles it took for decapsulation. This is the time it takes to compute the session key from the ciphertext, given the user's secret key. Once again the slowest algorithm implementation was sikep503 for both the Raspberry Pi 3B+ and the Raspberry Pi 4. Lightsaber was beaten by both threebears624r2cpax and r5nd0kem2iot for the fastest implementation for decapsulation.

Table 5

Cycles for decapsulation on Raspberry Pi 3B+

quartile	median	quartile	system
61266	61233	61334	threebears624r2cpax
146201	156040	156925	r5nd0kem2iot
276788	277128	277370	lightsaber
491568	488199	490089	kyber512
538448	540960	557435	newhope512cca
738774	741586	753260	ntrukem443
1227650	1239358	1335353	mceliece348864f
1343348	1357655	1383831	ntskem1264
1666956	1666587	1671357	sntrup653
2131109	2134735	2161925	ledakem13
5162268	5172510	5237488	hqc1281
24401849	24436275	24498193	frodokem640
64049890	64152064	76001578	bike2l1nc
298374048	298407255	299534009	sikep503

Table 6

Cycles for decapsulation on Raspberry Pi 4

quartile	median	quartile	system
38522	38339	38391	threebears624r2cpax
82311	83016	84223	r5nd0kem2iot
167236	167685	168379	lightsaber
218907	219436	219466	kyber512
243330	243308	244938	newhope512cca
291892	292805	298973	ntrukem443
669115	676821	679115	mceliece348864f
863900	899553	927884	ntskem1264
1083039	1083628	1087204	sntrup653
1383899	1390815	1414088	ledakem13
3613350	3744759	3796920	hqc1281
18208288	18235332	18249562	frodokem640
26192249	26254360	27222465	bike2l1nc
252600237	253272348	253456422	sikep503

Data Presentation - Public-Key Cryptosystems

The following tables include data collected from the three cryptosystems that were available in version 20191221 of SUPERCOP. These are the cryptosystems that were submitted to the second round of the NIST competition for public-key encryption. SUPERCOP included other cryptosystems of algorithms other than the ones submitted to the NIST competition. However, they were not relevant to this paper and therefore were not included in

the tables below. The tables for the secret key, public key, encrypting 0 bytes, encrypting 23 bytes, and encrypting many bytes are included in Appendix D. Each of the algorithm implementations in the tables are the optimized versions. The following tables show the first quartile, median, and third quartile cycles it took to either generate a key pair, encrypt 59 bytes, or decrypt 59 bytes for both of the devices tested. The three public-key cryptosystem algorithms tested in this paper are NTRU Prime, LEDAcrypt, and Round 5.

Table 7 and 8 below show the cycles to generate a key pair for the Raspberry Pi 3B+ and Raspberry Pi 4 respectively. There was a huge discrepancy between the fastest and slowest algorithm. The algorithm implementation r5nd1pke5d was over six thousand times faster than ledapkc10 for both devices. For each of the algorithms, the Raspberry Pi 4 outperformed the Raspberry Pi 3B+.

Table 7

Cycles to generate a key pair for public-key encryption on Raspberry Pi 3B+

quartile	median	quartile	system
173636	174079	175740	r5nd1pke5d
1712548	1722841	1799144	ntruees401ep2
1068451694	1070934568	1072781948	ledapkc10

Table 8

Cycles to generate a key pair for public-key encryption on Raspberry Pi 4

quartile	median	quartile	system
97220	99188	99784	r5nd1pke5d
621187	637728	641796	ntruees401ep2
655194875	683602187	685294895	ledapkc10

The tables 9 and 10 below show the number of cycles it took to encrypt 59 bytes of data on the Raspberry Pi 3B+ and Raspberry Pi4 respectively. The implementation ntruees401ep2 had a faster time for encryption than r5nd1pke5d, even though the latter was much faster for key generation.

Table 9

Cycles to encrypt 59 bytes on Raspberry Pi 3B+

quartile	median	quartile	system
102309	103019	109573	ntruees401ep2
290008	290793	292096	r5nd1pke5d
20689497	20700112	20710384	ledapkc10

Table 10

Cycles to encrypt 59 bytes on Raspberry Pi 4

quartile	median	quartile	system
61229	62513	62837	ntruees401ep2
163139	166090	166737	r5nd1pke5d
13714784	13644864	13658967	ledapkc10

Table 11 and Table 12 show the number of cycles for decrypting a 59 byte message. Once again, ledapkc10 was the slowest algorithm out of the three by far on both the Raspberry Pi 3B+ and Raspberry Pi 4.

Table 11

Cycles to decrypt 59 bytes on Raspberry Pi 3B+

quartile	median	quartile	system
149209	150341	158108	ntruees401ep2
420210	421204	422267	r5nd1pke5d
5694057	5698432	5702822	ledapkc10

Table 12

Cycles to decrypt 59 bytes on Raspberry Pi 4

quartile	median	quartile	system
75879	75248	76495	ntruees401ep2
230025	230488	231765	r5nd1pke5d
3218949	3267138	4427467	ledapkc10

Data Analysis

The data collected by SUPERCOP was first examined by creating several graphs that show how closely the grouping was for the algorithms on both of the devices. Then calculations were done for a two-tailed paired t-test. The null hypothesis for each of the different key-encapsulation mechanisms were tested using this t-test.

The performance of the KEM key generation time comparing the Raspberry Pi 3B+ and the Raspberry Pi 4 can be seen in Figure 3 below. The graph is on a log scale for both the x and y axis. The x axis shows the time in CPU cycles and the y axis shows the public key size in bytes. The data points to the farthest left are the algorithms that were the fastest. The data points are on the same level vertically, because the algorithms generate a public key that is the

same length on both of the devices.

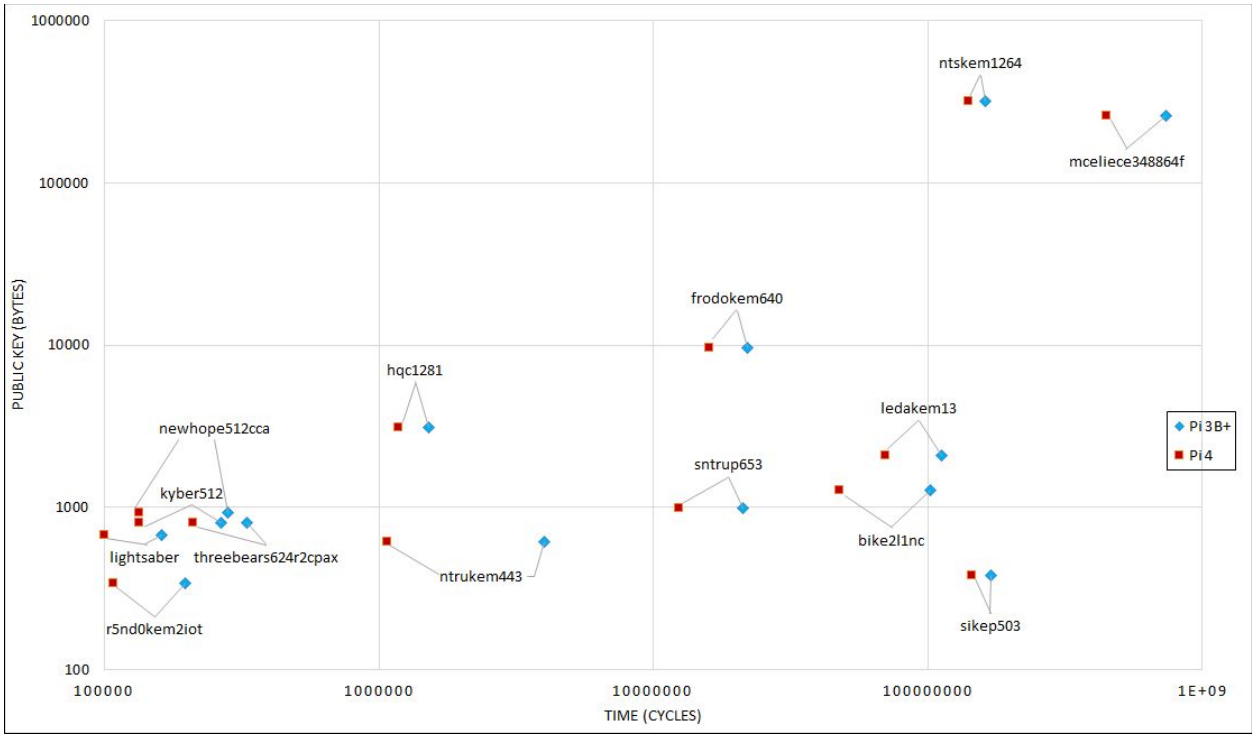

Figure 3. KEM Key Generation Time. Key-encapsulation mechanism cycles to generate key pairs vs. size of public key.

The next graph shows how the data for KEM encapsulation time is grouped for both the Raspberry Pi 3B+ and the Raspberry Pi 4. This graph has a log scale on both the x and y axis. The x axis shows the time in CPU cycles and the y axis shows the ciphertext length in bytes. The data points are on the same level vertically, because the algorithms generate a public key that is the same length on both of the devices.

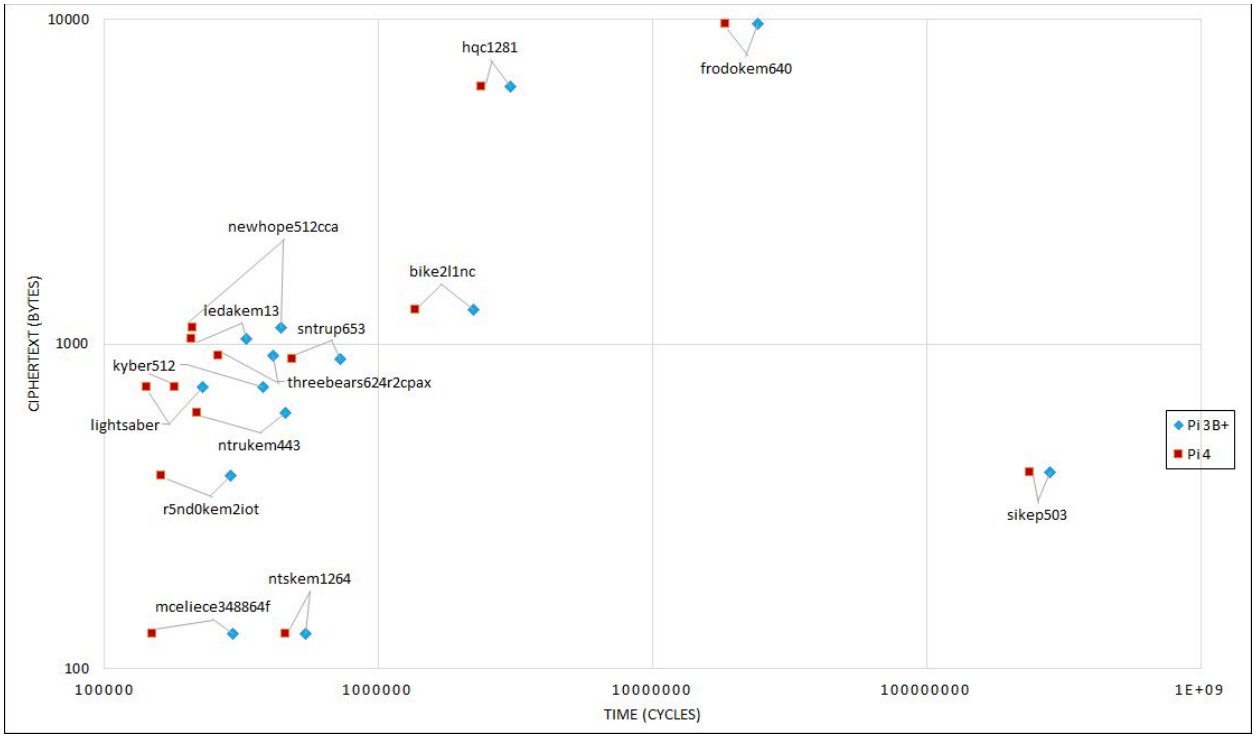

Figure 4. KEM Encapsulation Time. Key-encapsulation mechanism cycles to encapsulate keys vs size of the ciphertext in bytes.

The next graph shows how the data for KEM decapsulation time is grouped for both the Raspberry Pi 3B+ and the Raspberry Pi 4. This graph has a log scale on both the x and y axis. The x axis shows the time in CPU cycles and the y axis shows the ciphertext length in bytes. The data points are on the same level vertically, because the algorithms generate a public key that is the same length on both of the devices.

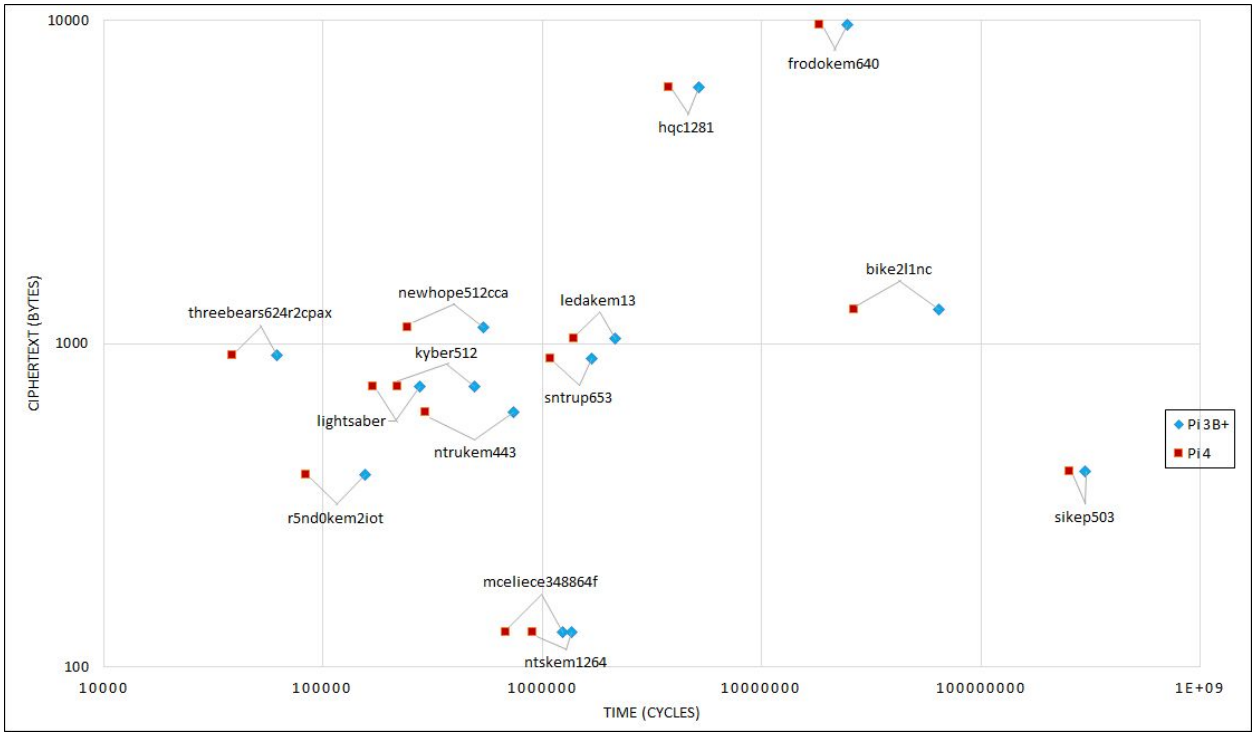

Figure 5. KEM Decapsulation Time. Key-encapsulation mechanism cycles to decapsulate keys vs size of the ciphertext in bytes.

While the graphs above give a good visualization of several different performance metrics, including the time to generate a key pair, the time for encapsulation, and the time for decapsulation, the two-tailed paired t-test was used to determine if the null hypothesis should be rejected. The probabilities were created by comparing the difference in the mean time for each of the algorithms on both the Raspberry Pi 3B+ and the Raspberry Pi 4. These probabilities were compared against a significance level of $\alpha = 0.05$. This process was repeated for all of the different performance metrics collected. Due to the limited scope of this paper, described in the methodology chapter of this paper, the statistical analysis has a limited sample size that only includes the best algorithm implementations.

Summary

This chapter presented the data collected using the SUPERCOP toolkit. During testing, only 14 of the 17 KEM algorithms were able to compile to run the tests. The graphs give a good visualization of how all of the performance metrics for each of the algorithms compare on the Raspberry Pi 3B+ and Raspberry Pi 4. The three cryptosystem algorithms tested showed that one of the algorithms was way slower than the other two. Finally, this chapter also discussed how the two-tailed paired t-test was used to determine if the null hypothesis should be rejected.

Chapter V: Results, Conclusions, and Future Work

Introduction

This chapter will review the results of the data that was analyzed in the previous chapter. First, the null hypothesis will be broken down into its different measurable performance metrics. Next, the conclusions of this study will be presented. Finally, future work related to this study will be discussed.

Results

The null hypothesis of this study proposes that there is no difference between the two ARMv8 devices regarding the benchmarked performance of public-key encryption and key-establishment algorithms using SUPERCOP. The null hypothesis was tested by doing a statistical analysis for each of the measurable performance metrics to see if the two devices performed significantly different. This study explores several different performance metrics, including the time to generate a key pair, the time for encapsulation, and the time for decapsulation.

The first null hypothesis proposes that there will be no difference between the two ARMv8 devices regarding the performance of the time to generate a key pair for the KEMs. The time to generate a key pair for the key-encapsulation mechanisms is the number of cycles it takes to generate a secret key and a corresponding public key. The results from the Raspberry Pi 3B+ (M = 95524670, SD = 196217289) and the Raspberry Pi 4 (M = 63156072, SD = 122431006) indicate that there was no significant difference between the two devices, $t(13) = 1.58$, $p = .137$. With the probability being higher than the $\alpha = 0.05$ in the two-tailed paired t-test, the null hypothesis is accepted and the alternative hypothesis is rejected.

The second null hypothesis proposes that there will be no difference between the two ARMv8 devices regarding the performance of the time for encapsulation. The time for

encapsulation is the number of cycles it takes to compute a ciphertext and corresponding session key, given a user's public key. The results from the Raspberry Pi 3B+ (M = 22432206, SD = 74536171) and the Raspberry Pi 4 (M = 18722862, SD = 63166994) indicate that there was no significant difference between the two devices, $t(13) = 1.22$, $p = .244$. With the probability being higher than the $\alpha = 0.05$ in the two-tailed paired t-test, the null hypothesis is accepted and the alternative hypothesis is rejected.

The third null hypothesis proposes that there will be no difference between the two ARMv8 devices regarding the performance of the time for decapsulation. The time for decapsulation is the number of cycles it takes to compute the session key from the ciphertext, given the user's secret key. The results from the Raspberry Pi 3B+ (M = 28630827, SD = 79576776) and the Raspberry Pi 4 (M = 21900157, SD = 67066381) indicate that there was no significant difference between the two devices, $t(13) = 1.69$, $p = .115$. With the probability being higher than the $\alpha = 0.05$ in the two-tailed paired t-test, the null hypothesis is accepted and the alternative hypothesis is rejected.

Conclusion

The results of this study show that the speed for the public-key encryption and key-establishment algorithms tested on the Raspberry Pi 3B+ and Raspberry Pi 4 were not significantly different. Taking a look at the number and selection of the samples may help explain how this result was not totally unexpected. This study was narrowly focused on testing only the fastest implementation (optimized or had the shortest key length) for each of the tested post-quantum algorithms. With only 14 post-quantum algorithms tested, this selection process generated a small sample size. Having such a small sample size can affect the quality of the results for the statistical analysis.

This study does not take into account the different security levels for each of the algorithm implementations. If the focus of the algorithm was to be as robust as possible to attacks, such as perfect forward secrecy, resistance to side-channel and multi-key attacks, as well as resistance to misuse, then that algorithm would need to sacrifice speed for increased security. Depending on the security needs, it may be more beneficial to use the slower algorithm implementation for encryption. The implementations that are more optimized were able to perform better on all hardware without causing a significant bottleneck, while poorly optimized implementations may lead to bottlenecks and exacerbated differences in algorithm speed.

Future Work

When NIST moves onto round 3 of the post-quantum cryptography public-key encryption and key-establishment algorithms standardization, even more performance testing should be done. For example, this study does not include memory testing. Future studies should include testing on how much memory is used for generating keys, encapsulation, decapsulation, etc.

Other hardware to consider for future testing are ARM Cortex-M series processors. These processors are optimized for cost and power-efficient microcontrollers. They use less energy and are less powerful than the ARM Cortex-A. Neither of the Raspberry Pi 3B+ nor Raspberry Pi 4 are considered to be constrained devices. The ARM Cortex-M series processors are used on constrained devices where memory, CPU processing power, and CPU power draw can be a problem.

The SUPERCOP version 20191221 had problems compiling on the ARMv6 Raspberry Pi Zero W. The Raspberry Pi Zero W only runs AArch32, and had problems trying to compile the different post-quantum algorithms. Of the 17 post-quantum algorithms, there were three algorithms (ROLLO, LAC, and RQC) that had problems compiling on the Raspberry Pi devices

used in this study. Future testing of the post-quantum algorithms should take this into account and make sure that there are implementations available that are able to compile correctly.

References

Alagic, G., Alperin-Sheriff, J., Apon, D., Cooper, D., Dang, Q., ... & Perlner, R. (2019). Status report on the first round of the NIST post-quantum cryptography standardization process. US Department of Commerce, National Institute of Standards and Technology.

Arm Ltd. (n.d.). ARM Cortex-A Series Programmer's Guide for ARMv8-A: ARMv8-A. Retrieved April 1, 2020, from https://developer.arm.com/docs/den0024/latest/armv8-a-architecture-and-processors/armv8-a

Aroca, R. V., & Gonçalves, L. M. G. (2012). Towards green data centers: A comparison of x86 and ARM architectures power efficiency. Journal of Parallel and Distributed Computing, 72(12), 1770-1780.

Bernstein, D.J., & Lange, T. (n.d.). eBACS: ECRYPT Benchmarking of Cryptographic Systems. https://bench.cr.yp.to, accessed 7 March 2020.

Cichonski, J., Marron, J., Hastings, N., Ajmo, J., & Rufus, R. (2019). Security for IoT Sensor Networks: Building Management Case Study (Draft) (pp. 29-29). National Institute of Standards and Technology.

eBACS: ECRYPT Benchmarking of Cryptographic Systems. (n.d.). Retrieved April 1, 2020, from https://bench.cr.yp.to/results-kem.html

Fagan, M., Megas, K., Scarfone, K., & Smith, M. (2019). Core Cybersecurity Feature Baseline for Securable IoT Devices: A Starting Point for IoT Device Manufacturers (No. NIST Internal or Interagency Report (NISTIR) 8259 (Draft)). National Institute of Standards and Technology.

HP Study Finds Alarming Vulnerabilities with Internet of Things (IoT) Home Security Systems. (2015, February 10). Retrieved from https://www8.hp.com/us/en/hp-news/press-release.html?id=1909050

Hughes, P. (2017, February 27). Inside the numbers: 100 billion ARM-based chips. Retrieved from https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/inside-the-numbers-100-billion-arm-based-chips-1345571105

McKay, K., Bassham, L., Sönmez Turan, M., & Mouha, N. (2016). Report on lightweight cryptography (No. NIST Internal or Interagency Report (NISTIR) 8114 (Draft)). National Institute of Standards and Technology.

Khan, M. A., & Salah, K. (2018). IoT security: Review, blockchain solutions, and open challenges. Future Generation Computer Systems, 82, 395-411.

Patil, P., Narayankar, P., Narayan, D. G., & Meena, S. M. (2016). A comprehensive evaluation of cryptographic algorithms: DES, 3DES, AES, RSA and Blowfish. Procedia Computer Science, 78, 617-624.

Voas, J. (2016). Networks of ‘things’. NIST Special Publication, 800(183), 800-183.

Appendix A

The three post-quantum key-encapsulation mechanisms that failed to compile include ROLLO, LAC, and RQC. When the algorithms failed to compile, SUPERCOP produced an error output. The error output is listed here in Appendix A.

ROLLO is a compilation of the three post-quantum schemes Rank-Ouroboros, LOCKER, and LAKE. Both LAKE and LOCKER are included in SUPERCOP version 20191221. For ROLLO, neither LAKE nor LOCKER would compile correctly. Due to the compiler errors it is not included in the main analysis of this study. The Compiler output is pasted below.

Compiler output for locker1:

Implementation: crypto_kem/locker1/ref

Compiler: g++ -march=native -mtune=native -O2 -fomit-frame-pointer -fwrapv -fPIC -fPIE

try.cpp: /usr/bin/ld: /home/djb/benchmarking/supercop-20191221/supercop-data/pi3bplus/aarch64/lib/libntl.a(GF2E.o): undefined reference to symbol 'pthread_setspecific@@GLIBC_2.17'

try.cpp: /usr/bin/ld: //lib/aarch64-linux-gnu/libpthread.so.0: error adding symbols: DSO missing from command line

try.cpp: collect2: error: ld returned 1 exit status

Compiler output for lake1:

Implementation: crypto_kem/lake1/ref

Compiler: g++ -march=native -mtune=native -O2 -fomit-frame-pointer -fwrapv -fPIC -fPIE

try.cpp: /usr/bin/ld: /home/djb/benchmarking/supercop-20191221/supercop-data/pi3bplus/aarch64/lib/libntl.a(GF2E.o): undefined reference to symbol 'pthread_setspecific@@GLIBC_2.17'

try.cpp: /usr/bin/ld: //lib/aarch64-linux-gnu/libpthread.so.0: error adding symbols: DSO missing from command line

try.cpp: collect2: error: ld returned 1 exit status

The LAC implementation lac128 had both a checksum failure and test failure outputs after trying to compile. It is not included in the main analysis of this study. The compiler output is pasted below.

Checksum failure for lac128:

Implementation: crypto_kem/lac128/opt

Compiler: gcc -march=native -mtune=native -O2 -fomit-frame-pointer -fwrapv -fPIC -fPIE

Test failure for lac128:

Implementation: crypto_kem/lac128/opt

Compiler: clang -mcpu=native -O3 -fomit-frame-pointer -fwrapv -Qunused-arguments -fPIC -fPIE

error 111

crypto_kem_dec does not match k

Compiler output for lac128:

Implementation: crypto_kem/lac128/avx2

Compiler: clang -mcpu=native -O3 -fomit-frame-pointer -fwrapv -Qunused-arguments -fPIC -fPIE

bin-lwe.c: In file included from bin-lwe.c:1:

bin-lwe.c: In file included from /usr/lib/llvm-9/lib/clang/9.0.0/include/immintrin.h:14:

bin-lwe.c: /usr/lib/llvm-9/lib/clang/9.0.0/include/mmintrin.h:50:12: error: invalid conversion between vector type '__m64' (vector of 1 'long long' value) and integer type 'int' of different size

bin-lwe.c: return (__m64)__builtin_ia32_vec_init_v2si(__i, 0);

bin-lwe.c: ^~~~~~~

bin-lwe.c: /usr/lib/llvm-9/lib/clang/9.0.0/include/mmintrin.h:129:12: error: invalid conversion between vector type '__m64' (vector of 1 'long long' value) and integer type 'int' of different size

bin-lwe.c: return (__m64)__builtin_ia32_packsswb((__v4hi)__m1, (__v4hi)__m2);

bin-lwe.c: ^~~~~~~

bin-lwe.c: /usr/lib/llvm-9/lib/clang/9.0.0/include/mmintrin.h:159:12: error: invalid conversion between vector type '__m64' (vector of 1 'long long' value) and integer type 'int' of different size

bin-lwe.c: return (__m64)__builtin_ia32_packssdw((__v2si)__m1, (__v2si)__m2);

bin-lwe.c: ^~~~~~~

bin-lwe.c: /usr/lib/llvm-9/lib/clang/9.0.0/include/mmintrin.h:189:12: error: invalid conversion between vector type '__m64' (vector of 1 'long long' value) and integer type 'int' of different size

bin-lwe.c: return (__m64)__builtin_ia32_packuswb((__v4hi)__m1, (__v4hi)__m2);

bin-lwe.c: ^~~~~~~

bin-lwe.c: /usr/lib/llvm-9/lib/clang/9.0.0/include/mmintrin.h:216:12: error: invalid conversion between vector type '__m64' (vector of 1 'long long' value) and integer type 'int' of different size

bin-lwe.c: return (__m64)__builtin_ia32_punpckhbw((__v8qi)__m1, (__v8qi)__m2);

bin-lwe.c: ^~~~~~~

bin-lwe.c: /usr/lib/llvm-9/lib/clang/9.0.0/include/mmintrin.h:239:12: error: invalid conversion between vector type '__m64' (vector of 1 'long long' value) and integer type 'int' of different size

bin-lwe.c: return (__m64)__builtin_ia32_punpckhwd((__v4hi)__m1, (__v4hi)__m2);

bin-lwe.c: ^~~~~~~

bin-lwe.c: /usr/lib/llvm-9/lib/clang/9.0.0/include/mmintrin.h:260:12: error: invalid conversion between vector type '__m64' (vector of 1 'long long' value) and integer type 'int' of different size

bin-lwe.c: return (__m64)__builtin_ia32_punpckhdq((__v2si)__m1, (__v2si)__m2);

bin-lwe.c: ^~~~~~~

bin-lwe.c: /usr/lib/llvm-9/lib/clang/9.0.0/include/mmintrin.h:287:12: error: invalid conversion between vector type '__m64' (vector of 1 'long long' value) and integer type 'int' of different size

bin-lwe.c: return (__m64)__builtin_ia32_punpcklbw((__v8qi)__m1, (__v8qi)__m2);

bin-lwe.c: ...

Implementation: crypto_kem/lac128/opt

Compiler: clang -mcpu=native -O3 -fomit-frame-pointer -fwrapv -Qunused-arguments -fPIC -fPIE

bin-lwe.c: bin-lwe.c:98:10: warning: result of comparison of constant -1 with expression of type 'const char' is always false [-Wtautological-constant-out-of-range-compare]

bin-lwe.c: if(s[i]==-1)

bin-lwe.c: ~~~~^ ~~

bin-lwe.c: bin-lwe.c:160:10: warning: result of comparison of constant -1 with expression of type 'const char' is always false [-Wtautological-constant-out-of-range-compare]

bin-lwe.c: if(s[i]==-1)

bin-lwe.c: ~~~~^ ~~

bin-lwe.c: 2 warnings generated.

Implementation: crypto_kem/lac128/avx2

Compiler: gcc -march=native -mtune=native -O2 -fomit-frame-pointer -fwrapv -fPIC -fPIE

bin-lwe.c: bin-lwe.c:1:10: fatal error: immintrin.h: No such file or directory

bin-lwe.c: 1 | #include <immintrin.h>

bin-lwe.c: | ^~~~~~~~~~~~~

bin-lwe.c: compilation terminated.

The RQC implementation rqc128/opt had a compiler error that ends with compilation terminated and is not included in the main analysis of this study. The compiler output is pasted below.

Compiler output for rqc128/opt:

Implementation: crypto_kem/rqc128/opt

Compiler: clang -mcpu=native -O3 -fomit-frame-pointer -fwrapv -Qunused-arguments -fPIC -fPIE

ffi_elt.c: In file included from ffi_elt.c:6:

ffi_elt.c: In file included from ./ffi.h:11:

ffi_elt.c: In file included from /usr/lib/llvm-7/lib/clang/7.0.1/include/x86intrin.h:29:

ffi_elt.c: In file included from /usr/lib/llvm-7/lib/clang/7.0.1/include/immintrin.h:28:

ffi_elt.c: /usr/lib/llvm-7/lib/clang/7.0.1/include/mmintrin.h:64:12: error: invalid conversion between vector type '__m64' (vector of 1 'long long' value) and integer type 'int' of different size

ffi_elt.c: return (__m64)__builtin_ia32_vec_init_v2si(__i, 0);

ffi_elt.c: ^~~~~~~

ffi_elt.c: /usr/lib/llvm-7/lib/clang/7.0.1/include/mmintrin.h:143:12: error: invalid conversion between vector type '__m64' (vector of 1 'long long' value) and integer type 'int' of different size

ffi_elt.c: return (__m64)__builtin_ia32_packsswb((__v4hi)__m1, (__v4hi)__m2);

ffi_elt.c: ^~~~~~~

ffi_elt.c: /usr/lib/llvm-7/lib/clang/7.0.1/include/mmintrin.h:173:12: error: invalid conversion between vector type '__m64' (vector of 1 'long long' value) and integer type 'int' of different size

ffi_elt.c: return (__m64)__builtin_ia32_packssdw((__v2si)__m1, (__v2si)__m2);

ffi_elt.c: ^~~~~~~

```
ffi_elt.c: /usr/lib/llvm-7/lib/clang/7.0.1/include/mmintrin.h:203:12: error: invalid conversion
between vector type '__m64' (vector of 1 'long long' value) and integer type 'int' of different size
ffi_elt.c: return (__m64)__builtin_ia32_packuswb((__v4hi)__m1, (__v4hi)__m2);
ffi_elt.c: ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
ffi_elt.c: /usr/lib/llvm-7/lib/clang/7.0.1/include/mmintrin.h:230:12: error: invalid conversion
between vector type '__m64' (vector of 1 'long long' value) and integer type 'int' of different size
ffi_elt.c: return (__m64)__builtin_ia32_punpckhbw((__v8qi)__m1, (__v8qi)__m2);
ffi_elt.c: ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
ffi_elt.c: /usr/lib/llvm-7/lib/clang/7.0.1/include/mmintrin.h:253:12: error: invalid conversion
between vector type '__m64' (vector of 1 'long long' value) and integer type 'int' of different size
ffi_elt.c: return (__m64)__builtin_ia32_punpckhwd((__v4hi)__m1, (__v4hi)__m2);
ffi_elt.c: ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
ffi_elt.c: /usr/lib/llvm-7/lib/clang/7.0.1/include/mmintrin.h:274:12: error: invalid conversion
between vector type '__m64' (vector of 1 'long long' value) and integer type 'int' of different size
ffi_elt.c: return (__m64)__builtin_ia32_punpckhdq((__v2si)__m1, (__v2si)__m2);
ffi_elt.c: ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
ffi_elt.c: ...
Implementation: crypto_kem/rqc128/opt
Compiler: gcc -march=native -mtune=native -O2 -fomit-frame-pointer -fwrapv -fPIC -fPIE
ffi_elt.c: In file included from ffi_elt.c:6:
ffi_elt.c: ffi.h:11:10: fatal error: x86intrin.h: No such file or directory
ffi_elt.c: #include <x86intrin.h>
ffi_elt.c: ^~~~~~~~~~~~
ffi_elt.c: compilation terminated.
```

Appendix B

The following tables include SUPERCOP output for crypto_kem that includes the time in processor cycles, primitive, implementation of the algorithm, and compiler used for both the Raspberry Pi 3B+ and the Raspberry Pi 4.

Table 13

aarch64, pi3bplusubuntuserver64, crypto_kem compiler output

Time	Relative time	Primitive	Implementation	Compiler
28681863	1.00	bike2l1nc	crypto_kem/bike2l1nc/ref_ossl (BIKE_v1.0_Additional_11/18/2018)	gcc -march=native -mtune=native -O3 -fomit-frame-pointer -fwrapv -fPIC -fPIE (9.2.1 20191008)
840038	1.00	mceliece348864f	crypto_kem/mceliece348864f/vec	clang -mcpu=native -O3 -fomit-frame-pointer -fwrapv -Qunused-arguments -fPIC -fPIE (Clang 9.0.0 (tags/RELEASE 900/final))
399185	1.00	kyber512	crypto_kem/kyber512/ref	clang -mcpu=native -O3 -fomit-frame-pointer -fwrapv -Qunused-arguments -fPIC -fPIE (Clang 9.0.0 (tags/RELEASE 900/final))
64181560	1.00	frodokem640	crypto_kem/frodokem640/optimized	clang -mcpu=native -O3 -fomit-frame-pointer -fwrapv -Qunused-arguments -fPIC -fPIE (Clang 9.0.0 (tags/RELEASE 900/final))
6044516	1.00	hqc1281	crypto_kem/hqc1281/opt	gcc -march=native -mtune=native -O3 -fomit-frame-pointer -fwrapv -fPIC -fPIE (9.2.1 20191008)
1592907	1.00	ledakem13	crypto_kem/ledakem13/portableopt	gcc -march=native -mtune=native -O3 -fomit-frame-pointer -fwrapv -fPIC -fPIE (9.2.1 20191008)

Table 13 continued

Time	Relative time	Primitive	Implementation	Compiler
454115	1.00	newhope512cca	crypto_kem/newhope512cca/ref	gcc -march=native -mtune=native -O3 -fomit-frame-pointer -fwrapv -fPIC -fPIE (9.2.1 20191008)
536038	1.00	ntrukem443	crypto_kem/ntrukem443/ref	gcc -march=native -mtune=native -O3 -fomit-frame-pointer -fwrapv -fPIC -fPIE (9.2.1 20191008)
1569796	1.00	sntrup653	crypto_kem/sntrup653/factored	gcc -march=native -mtune=native -O3 -fomit-frame-pointer -fwrapv -fPIC -fPIE (9.2.1 20191008)
992215	1.00	ntskem1264	crypto_kem/ntskem1264/opt	gcc -march=native -mtune=native -O3 -fomit-frame-pointer -fwrapv -fPIC -fPIE (9.2.1 20191008)
244617	1.00	r5nd0kem2iot	crypto_kem/r5nd0kem2iot/opt	gcc -march=native -mtune=native -O3 -fomit-frame-pointer -fwrapv -fPIC -fPIE (9.2.1 20191008)
312320	1.00	lightsaber	crypto_kem/lightsaber/portable	clang -mcpu=native -O3 -fomit-frame-pointer -fwrapv -Qunused-arguments -fPIC -fPIE (Clang 9.0.0 (tags/RELEASE 900/final))
489846260	1.00	sikep503	crypto_kem/sikep503/opt	clang -mcpu=native -O3 -fomit-frame-pointer -fwrapv -Qunused-arguments -fPIC -fPIE (Clang 9.0.0 (tags/RELEASE 900/final))
298911	1.00	threebears624r2cpax	crypto_kem/threebears624r2cpax/opt	gcc -march=native -mtune=native -O3 -fomit-frame-pointer -fwrapv -fPIC -fPIE

Table 14

aarch64, pi4ubuntuserver64, crypto_kem compiler output

Time	Relative time	Primitive	Implementation	Compiler
28323497	1.00	bike2l1nc	crypto_kem/bike2l1nc/ref_ossl (BIKE_v1.0_Additional_11/18/2018)	gcc -march=native -mtune=native -O3 -fomit-frame-pointer -fwrapv -fPIC -fPIE (9.2.1 20191008)
836798	1.00	mceliece348864f	crypto_kem/mceliece348864f/vec	clang -mcpu=native -O3 -fomit-frame-pointer -fwrapv -Qunused-arguments -fPIC -fPIE (Clang 9.0.0 (tags/RELEASE 900/final))
402344	1.00	kyber512	crypto_kem/kyber512/ref	clang -mcpu=native -O3 -fomit-frame-pointer -fwrapv -Qunused-arguments -fPIC -fPIE (Clang 9.0.0 (tags/RELEASE 900/final))
63898904	1.00	frodokem640	crypto_kem/frodokem640/optimized	clang -mcpu=native -O3 -fomit-frame-pointer -fwrapv -Qunused-arguments -fPIC -fPIE (Clang 9.0.0 (tags/RELEASE 900/final))
6043445	1.00	hqc1281	crypto_kem/hqc1281/opt	gcc -march=native -mtune=native -O3 -fomit-frame-pointer -fwrapv -fPIC -fPIE (9.2.1 20191008)
1582114	1.00	ledakem13	crypto_kem/ledakem13/portableopt	gcc -march=native -mtune=native -O3 -fomit-frame-pointer -fwrapv -fPIC -fPIE (9.2.1 20191008)
457541	1.00	newhope512cca	crypto_kem/newhope512cca/ref	gcc -march=native -mtune=native -O3 -fomit-frame-pointer -fwrapv -fPIC -fPIE (9.2.1 20191008)

Table 14 continued

Time	Relative time	Primitive	Implementation	Compiler
534005	1.00	ntrukem443	crypto_kem/ntrukem443/ref	gcc -march=native -mtune=native -O3 -fomit-frame-pointer -fwrapv -fPIC -fPIE (9.2.1 20191008)
1570757	1.00	sntrup653	crypto_kem/sntrup653/factored	gcc -march=native -mtune=native -O3 -fomit-frame-pointer -fwrapv -fPIC -fPIE (9.2.1 20191008)
992119	1.00	ntskem1264	crypto_kem/ntskem1264/opt	gcc -march=native -mtune=native -O3 -fomit-frame-pointer -fwrapv -fPIC -fPIE (9.2.1 20191008)
245236	1.00	r5nd0kem2iot	crypto_kem/r5nd0kem2iot/opt	gcc -march=native -mtune=native -O3 -fomit-frame-pointer -fwrapv -fPIC -fPIE (9.2.1 20191008)
311889	1.00	lightsaber	crypto_kem/lightsaber/portable	clang -mcpu=native -O3 -fomit-frame-pointer -fwrapv -Qunused-arguments -fPIC -fPIE (Clang 9.0.0 (tags/RELEASE 900/final))
489766721	1.00	sikep503	crypto_kem/sikep503/opt	clang -mcpu=native -O3 -fomit-frame-pointer -fwrapv -Qunused-arguments -fPIC -fPIE (Clang 9.0.0 (tags/RELEASE 900/final))
298022	1.00	threebears624r2cpax	crypto_kem/threebears624r2cpax/opt	gcc -march=native -mtune=native -O3 -fomit-frame-pointer -fwrapv -fPIC -fPIE

Appendix C

The following tables include the SUPERCOP output for the amount of space in bytes for a secret key, public key, ciphertext, and session key for Raspberry Pi 3B+ and Raspberry Pi 4. aarch64, pi3bplusubuntuserver64, crypto_kem

Table 15

Secret key

bytes	system
16	r5nd0kem2iot
434	sikep503
540	ledakem13
701	ntrukem443
804	threebears624r2cpax
1518	sntrup653
1568	lightsaber
1632	kyber512
1888	newhope512cca
3165	hqc1281
4964	bike2l1nc
6452	mceliece348864f
9216	ntskem1264
19872	frodokem640

Table 16

Public key

bytes	system
342	r5nd0kem2iot
378	sikep503
611	ntrukem443
672	lightsaber
800	kyber512
804	threebears624r2cpax
928	newhope512cca
994	sntrup653
1271	bike2l1nc
2080	ledakem13
3125	hqc1281
9616	frodokem640
261120	mceliece348864f
319488	ntskem1264

Table 17

Ciphertext

bytes	system
128	mceliece348864f
128	ntskem1264
394	r5nd0kem2iot
402	sikep503
611	ntrukem443
736	kyber512
736	lightsaber
897	sntrup653
917	threebears624r2cpax
1040	ledakem13
1120	newhope512cca
1271	bike2l1nc
6234	hqc1281
9736	frodokem640

Table 18

Session key

bytes	system
16	frodokem640
16	r5nd0kem2iot
16	sikep503
32	bike2l1nc
32	mceliece348864f
32	kyber512
32	ledakem13
32	newhope512cca
32	ntrukem443
32	sntrup653
32	ntskem1264
32	lightsaber
32	threebears624r2cpax
64	hqc1281

aarch64, pi4ubuntuserver64, crypto_kem

Table 19

Secret key

bytes	system
16	r5nd0kem2iot
434	sikep503
540	ledakem13
701	ntrukem443
804	threebears624r2cpax
1518	sntrup653
1568	lightsaber
1632	kyber512
1888	newhope512cca
3165	hqc1281
4964	bike2l1nc
6452	mceliece348864f
9216	ntskem1264
19872	frodokem640

Table 20

Public key

bytes	system
342	r5nd0kem2iot
378	sikep503
611	ntrukem443
672	lightsaber
800	kyber512
804	threebears624r2cpax
928	newhope512cca
994	sntrup653
1271	bike2l1nc
2080	ledakem13
3125	hqc1281
9616	frodokem640
261120	mceliece348864f
319488	ntskem1264

Table 21

Ciphertext

bytes	system
128	mceliece348864f
128	ntskem1264
394	r5nd0kem2iot
402	sikep503
611	ntrukem443
736	kyber512
736	lightsaber
897	sntrup653
917	threebears624r2cpax
1040	ledakem13
1120	newhope512cca
1271	bike2l1nc
6234	hqc1281
9736	frodokem640

Table 22

Session key

bytes	system
16	frodokem640
16	r5nd0kem2iot
16	sikep503
32	bike2l1nc
32	mceliece348864f
32	kyber512
32	ledakem13
32	newhope512cca
32	ntrukem443
32	sntrup653
32	ntskem1264
32	lightsaber
32	threebears624r2cpax
64	hqc1281

Appendix D

The following tables include the SUPERCOP public-key cryptosystem output for the secret key, public key, encrypting 0 bytes, encrypting 23 bytes, and encrypting many bytes for Raspberry Pi 3B+ and Raspberry Pi 4.

aarch64, pi3bplusubuntuserver64, crypto_encrypt

Table 23

Secret Key

bytes	system
26	ledapkc10
493	r5nd1pke5d
607	ntruees401ep2

Table 24

Public Key

bytes	system
461	r5nd1pke5d
557	ntruees401ep2
4488	ledapkc10

Table 25

Encrypting 0 bytes

bytes	system
552	ntruees401ep2
636	r5nd1pke5d
8976	ledapkc10

Table 26

Encrypting 23 bytes

bytes	system
529	ntruees401ep2
636	r5nd1pke5d
8953	ledapkc10

Table 27

Encrypting many bytes

bytes	system
544	ntruees401ep2
636	r5nd1pke5d
4521	ledapkc10

aarch64, pi4ubuntuserver64, crypto_encrypt

Table 28

Secret Key

bytes	system
26	ledapkc10
493	r5nd1pke5d
607	ntruees401ep2

Table 29

Public Key

bytes	system
461	r5nd1pke5d
557	ntruees401ep2
4488	ledapkc10

Table 30

Encrypting 0 bytes

bytes	system
552	ntruees401ep2
636	r5nd1pke5d
8976	ledapkc10

Table 31

Encrypting 23 bytes

bytes	system
529	ntruees401ep2
636	r5nd1pke5d
8953	ledapkc10

Table 32

Encrypting many bytes

bytes	system
544	ntruees401ep2
636	r5nd1pke5d
4521	ledapkc10