Kristian Leirimaa

Supporting PQC migration and cryptographic agility with automated CBOM generation

Master's Thesis in Information Technology

October 30, 2024

University of Jyväskylä

Faculty of Information Technology

Author: Kristian Leirimaa

Contact information: `kristian.m.p.leirimaa@student.jyu.fi`

Supervisor: Timo Hämäläinen

Title: Supporting PQC migration and cryptographic agility with automated CBOM generation

Työn nimi: PQC migraation ja kryptografisen ketteryyden tukeminen automatisoidulla CBOM työkalulla

Project: Master's Thesis

Study line: Tietotekniikka

Page count: 104+1

Abstract:

The technological development of quantum computers has advanced dramatically in recent years as organizations and governments seek to take advantage of the increasing computing power of quantum computers. Although quantum computers have the potential to benefit people and economies in many areas, they also threaten the security of modern cryptography, especially the widely used public key cryptography such as RSA, DH, ECC, and DSA. Since these algorithms will be completely broken in the future, quantum-safe alternatives are being developed and researched to mitigate the threat. History has shown that replacing cryptographic algorithms is a long and difficult process, and given the complexity of modern information systems, automated tools are needed to support post-quantum cryptography migration and cryptographic agility. In this research, an automated Cryptography Bill of Materials (CBOM) generator was built as a solution to this need and it's feasibility was analyzed. Design science principles were used to guide the research process, as well as the building and evaluation of the created artifact. The main result of the research is an artifact capable of generating CBOMs by scanning cryptographic algorithms from JavaScript source files that implement the Node.js Crypto module. In addition, the research proved that an automated CBOM tool based on regular expression searches is a feasible and accurate solution

for capturing cryptographic components.

Keywords: PQC, post-quantum cryptography, quantum computing, public-key cryptography, private-key cryptography, CBOM, Cryptography Bill of Materials, PQC migration, cryptographic agility

Suomenkielinen tiivistelmä: Kvanttitietokoneiden tekninen kehitys on edennyt hurjaa vauhtia viime vuosina, organisaatioiden ja valtiollisten toimijoiden toivoessa pääsevänsä hyödyntämään kvanttikoneiden jatkuvasti kasvavaa laskentatehoa. Vaikka ihmiset ja talous voivat hyötyä kvanttitietokoneista monilla aloilla, ne myös uhkaavat nykyaikaisia salausmenetelmiä, erityisesti laajasti käytössä olevia julkisen avaimen salausmenetelmiä, kuten RSA, DH, ECC ja DSA. Tulevaisuudessa nämä salausalgoritmit tulevat olemaan täysin murrettuja, minkä vuoksi vaihtoehtoisia kvanttiturvallisia salausmenetelmiä luodaan ja tutkitaan parhaillaan, jotta uhka saadaan torjuttua. Historiasta tiedetään, että haavoittuneiden salausmenetelmien korvaaminen on pitkä ja vaikea prosessi, ja varsinkin nykyajan monimutkaisten tietojärjestelmien kanssa automatisoituja työkaluja tarvitaan tukemaan migraatiota kvanttiturvallisiin salausmenetelmiin sekä lisäämään järjestelmien kryptografista ketteryyttä. Tässä tutkimuksessa automatisoitu Cryptography Bill of Materials (CBOM) työkalu luotiin vastaamaan edellä mainittuihin tarpeisiin ja työkalun soveltuvuutta analysoitiin. Suunnittelutieteen periaatteita käytettiin ohjaamaan tutkimusprosessia sekä työkalun rakentamista ja evaluointia. Tutkimuksen tuloksena valmistui työkalu, joka kykenee luomaan CBOM tiedostoja etsimällä kryptografisia algoritmeja JavaScript tiedostoista joissa on hyödynnetty Node.js Crypto moduulia. Lisäksi, tutkimus osoitti, että säännöllisiin lausekkeisiin perustuva automatisoitu CBOM työkalu on käyttökelpoinen ja tarkka ratkaisu kryptografisten komponenttien poimimiseen.

Avainsanat: PQC, kvanttikoneturvalliset salausmenetelmät, kvanttilaskenta, julkisen avaimen salaus, yksityisen avaimen salaus, CBOM, salausmenetelmien materiaaliluettelo, PQC migraatio, kryptografinen ketteryys

Glossary

Term	Definition
Asymmetric-key cryptography	Method for securely sharing secrets over public channels. Consists of public- and private key pairs, where the public key is used for encryption and private key is used for decryption. The following terms are often used as synonyms: public-key cryptography, public-key encryption, public-key algorithm, asymmetric-key encryption and asymmetric-key algorithm.
CBOM	Cryptography Bill of materials is a nested list of cryptographic components.
DH	Diffie-Hellman is a key exchange method based on public-key cryptography.
ECC	Elliptic curve cryptography is a public-key encryption method based on the use of elliptic curves.
IND-CCA2	Key-indistinguishability under chosen ciphertext attacks.
KEM	Key encapsulation mechanism is a method for sharing a secret (like a secret key) over a public channel in a secure way.
NIST	National Institute of Standards and Technology.
PQC	Post-quantum cryptography. Cryptography that is safe against attacks done with quantum computers and traditional computers.
Quantum cryptography	Cryptography that uses quantum physics as a part of it's security solution.
RSA	Rivest-Shamir-Adleman is a public-key encryption algorithm.
SBOM	Software-bill-of-materials is a nested list of software components.
Symmetric-key cryptography	A shared secret key between communicating participants is used to encrypt and decrypt messages. The following terms are often used as synonyms: private-key cryptography, secret-key cryptography, symmetric-key encryption and symmetric-key algorithm.

List of Figures

Figure 1. A table made by NIST about the "Impact of quantum computing on common cryptographic algorithms" (Chen et al. April 2016). 5
Figure 2. Estimates for the quantum resilience of modern cryptosystems. Derived from National Academies of Sciences and Meidicine March 2019, p.98. 13
Figure 3. NIST security strength classifications for symmetric-key algorithms. Table derived from R. A. Grimes 2019 14
Figure 4. Message authentication using symmetric encryption (Fall and Stevens 2011). ... 16
Figure 5. Digital signature using public-key encryption (Fall and Stevens 2011). 16
Figure 6. The use of hashed passwords (Stallings 2018, p. 315). 17
Figure 7. Estimated cryptographic algorithm security levels based on key lengths. Table derived from (Hankerson and Menezes 2011) 24
Figure 8. Comparison of the security levels of KYBER and AES. 29
Figure 9. Comparison of KEM's in the NIST security category 1. Derived from R. e. al. April 2024. 31
Figure 10. Comparison of KEM's in the NIST security category 3. Derived from R. e. al. April 2024. 31
Figure 11. A replica of Hevner's three cycle view framework for design science research (Hevner 2007). 42
Figure 12. Representation of the design science process for this research. Together with the starting and ending points, as well as the number of iterations planned for each process. 45
Figure 13. BOM specification for the artifact. Derived from CycloneDX 2024. 51
Figure 14. Example BOM file that displays the hierarchy of the components. 52
Figure 15. The command line interface of the created artifact. 55
Figure 16. Code snippet of `pqcbom.js`. A JavaScript BOM object is created and `scanDirectory` function is called. 55
Figure 17. Function call for `findNodeCryptoComponents` 56
Figure 18. The `algorithm` and `relatedCryptoMaterial` properties of the `NodeCrypto` class are used to store and categorize Node.js Crypto method calls. 57
Figure 19. The regular expression for searching related-crypto-material method calls related to the Node.js Crypto module. 57
Figure 20. The regular expression for searching cryptographic algorithm method calls related to the Node.js Crypto module. 58
Figure 21. Function call for `addComponent`, which extracts relevant data from found regular expression matches and uses it to create cryptographic components. . 58
Figure 22. Call for `extractFirstParameter` function. 59
Figure 23. As a temporary solution, only the cryptographic algorithm data is extracted from related-crypto-material components, and thus they are treated similarly to cryptographic algorithm components. 59
Figure 24. The creation of the cryptographic components in the `addComponent` function. 60

Figure 25. A code snippet of the `NistQuantumSecLevel` class used for retrieving the NIST quantum security levels of different cryptographic algorithms. 61
Figure 26. Demonstration of the pqcbom-tool: A CBOM file named *jsTestFileBOM* was created by scanning `jsTestFile.js`. Only 38 components were generated instead of the expected 39. 63
Figure 27. The updated `jsTestFileBOM.json` after correcting the coding error. All 39 components are now correctly identified from `jsTestFile.js`............ 65
Figure 28. With the addition of the Node.js Crypto module reference to the file, the second scan of `jsInvalidTestValues.js` identified a set of 9 cryptographic components. Thus, the artifact functioned as intended................................ 66
Figure 29. A `Set` of 14 related-crypto-material method calls was found in the scan of `relatedCryptoMatTestFile.js`, when 15 were expected.................... 68
Figure 30. After a quick fix to the regular expression, all 15 components where found and created to `relatedCrypoMatTestFileBOM2.json`....................... 68
Figure 31. Scanning `cryptoExamples_by_ChatGPT.js` resulted a `Set` of eight cryptographic algorithms and eight related-crypto-materials. 71

Contents

1 INTRODUCTION ... 1

2 THE QUANTUM THREATS ... 4
2.1 Quantum computing and quantum computers ... 5
2.1.1 History ... 5
2.1.2 Quantum computer introduction ... 7
2.2 Shor's algorithm ... 9
2.3 Grover's algorithm ... 10

3 QUANTUM THREATENED CRYPTOGRAPHY ... 12
3.1 Symmetric-key ... 13
3.1.1 AES ... 15
3.2 Hashing ... 16
3.3 Asymmetric-key ... 19
3.3.1 Diffie-Hellman ... 20
3.3.2 RSA (Rivest-Shamir-Adleman) ... 21
3.3.3 Digital Signature Algorithm (DSA) ... 22
3.3.4 Elliptic curve cryptography-based algorithms ... 23

4 QUANTUM RESISTANT SOLUTIONS ... 25
4.1 Families of PQC algorithms ... 25
4.1.1 Code-based ... 25
4.1.2 Hash-based ... 26
4.1.3 Lattice-based ... 27
4.1.4 Multivariate-system based ... 27
4.2 NIST solutions ... 28
4.2.1 Crystals - Kyber ... 28
4.2.2 Crystals - Dilithium ... 30
4.2.3 Falcon ... 30
4.2.4 SPHINCS+ ... 30
4.3 Possible solutions ... 31
4.3.1 Classic McEliece ... 32
4.3.2 BIKE ... 32
4.3.3 HQC ... 33

5 BILL OF MATERIALS ... 34
5.1 Software Bill of Materials (SBOM) ... 34
5.2 Cryptography Bill of Materials (CBOM) ... 36

6 RESEARCH QUESTIONS AND STRATEGY ... 39
6.1 Design science ... 39
6.2 Implementation in this research ... 43

7 ARTIFACT PLANNING ... 47

7.1 Verifying the need 47
7.2 Selecting the scheme 48
7.3 Programming language 48
7.4 Target and scope 49
7.5 Scanning method 49
7.6 Cryptographic libraries 50
7.7 Data fields 50
7.8 Data extraction 52
7.9 Minimum requirements 53

8 ARTIFACT 54
8.1 Setup and interface 54
8.2 Example data flow 55

9 EVALUATIONS 62
9.1 First evaluation 62
9.1.1 First test file 62
9.1.2 Second test file 66
9.1.3 Third test file 67
9.1.4 Summary 69
9.2 Second evaluation 70
9.2.1 Summary 72

10 CONCLUSIONS AND DISCUSSION 75
10.1 Discussion 75
10.2 Conclusions 77

11 SUMMARY 79

BIBLIOGRAPHY 81

APPENDICES 97
A Artifact source code and test files 97

1 Introduction

The development of quantum computers has progressed rapidly since the first quantum computers were built in the late 1990's. With the exponential increase in computational power compared to classical computers, quantum computers exhibit remarkable possibilities in fields such as weather forecasting (R. A. Grimes 2019), drug development (Cao, Romero, and Aspuru-Guzik 2018), chemical development (Cao et al. 2019), cybersecurity (Wallden and Kashefi March 2019), artificial intelligence, battery development (R. A. Grimes 2019), machine learning (Maria Schuld and Petruccione 2015), optimization problems, finance (Orús, Mugel, and Lizaso 2019) and manufacturing (Bova, Goldfarb, and Melko December 2021). It is estimated that the quantum technology market size will be 106 billion USD by 2040, with quantum computing being the most lucrative of the quantum technology fields, possibly reaching even a 93 billion USD market size by 2040 ("Quantum Technology Monitor" 2023). With all these positive scientific and monetary predictions, it is easy to see why quantum computing has gained so much interest.

Advances in quantum computing have additionally created security concerns, which also contribute to the increased interest in quantum computing. With sufficiently powerful quantum computers it is possible to break many of our modern day cryptographic algorithms, which are widely used to secure our communications over the internet. Particularly, public-key cryptography, like RSA, DH, ECC, and DSA will be vulnerable. Although the threat is not imminent, replacing widely used algorithms has proven to be a long and slow process, as was the case with SHA1, which was found vulnerable in 2004 but was still used in 2018 by some old browsers and servers that did not support SHA256 (National Academies of Sciences and Meidicine March 2019, p. 109). Because replacing widely used cryptographic algorithms is such a long process, the U.S. government's National Institute of Standards and Technology (NIST) began a competition-like standardization process in 2016 to find secure post-quantum cryptography (PQC) algorithms (Standards and Technology January 2017; NIST December 2016).

In addition to the slowness of adapting new protocols, a major driver of PQC development is the fact that malicious actors could harvest the quantum-vulnerable data today and decrypt

it later, when a sufficiently powerful quantum computer is built. This could expose critical governmental, political, or industrial secrets that could be relevant years or even decades later.

As of the writing of this paper in 2023-2024, NIST has already selected and standardized three PQC algorithms (Technology August 2024b, August 2024a, August 2024c), and other algorithm candidates are still being assessed and may later be added to the NIST PQC standards. Ultimately, the standardized algorithms will need to be used to replace specific quantum-vulnerable cryptographic algorithms. However, modern IT/OT systems can be extremely large and complex, making the management of cryptographic algorithms a tedious and demanding task. Especially, since many organizations do not even have an inventory of where cryptographic algorithms are used (Barker, Polk, and Souppaya April 2021). For the security of existing IT and OT systems, it is imperative to increase cryptographic agility and thereby accelerate the replacement of vulnerable components. As stated by NIST, "tools are urgently needed to facilitate the discovery of where and how public-key cryptography is being used in existing technology infrastructures" (Barker, Polk, and Souppaya April 2021). The goal of this research is to address this need by providing a Cryptography Bill of Materials (CBOM) tool capable of automatically scanning systems and extracting cryptographic data from specific files to create a list of cryptographic components. In addition, the cryptographic components will receive a NIST-categorized quantum security level to indicate the algorithms security level against powerful quantum computers. This goal aims to produce an innovative artifact that provides practical value to the artifact's application domain. Another goal is to produce new knowledge about automated CBOM generators by answering the research questions based on the knowledge gained from the artifact creation process.

The structure of the paper is as follows: chapter 2 provides general information about the reasons behind the emerging quantum threat, including quantum computers and Shor's and Grover's algorithms. Chapter 3 discusses the most common cryptographic algorithms that are vulnerable to the quantum threat. Chapter 4 provides insight into the solutions that are currently being researched or standardized as quantum-safe alternatives to some of the quantum-vulnerable algorithms mentioned in chapter 3. Chapter 5 provides general information about Bill of Materials (BOMs), specifically Software BOMs (SBOMs) and Cryptogra-

phy BOMs (CBOMs). Chapter 6 explains the research methods, questions, and strategies. Chapter 7 presents the steps involved in planning the artifact and motivates choices made. Chapter 8 presents the artifact and chapter 9 evaluates the artifact with two different data sets. Chapter 10 discusses and presents the results of the research and finally, chapter 11 provides a summary of the thesis.

2 The quantum threats

Our modern digital information and communication systems rely heavily on the security provided by symmetric-, asymmetric- and hashing algorithms. Some of the most common use cases for these cryptographic systems include: encryption, authentication, digital signing, secure web browsing, cryptocurrencies, smart cards, network encryption, virtual private networks, wireless security, and email encryption (R. A. Grimes 2019). With so many important applications relying on these cryptographic methods, it is extremely important that they are secure and trustworthy. However, the development of quantum computers is gradually bringing us closer to a reality where these cryptosystems will be weakened or completely broken by specific quantum algorithms that were invented back in the 1990s (will be introduced later in sections 2.2 and 2.3). It is uncertain when such quantum computers, capable of running the quantum algorithms that pose a threat to our modern cryptography, will be built, but according to some estimates it is theoretically possible by 2030-2035 (National Academies of Sciences and Meidicine March 2019; Moody March 2024; Presman April 2024). Figure 1 presents the impact that large-scale quantum computers can have on specific cryptographic algorithms.

This chapter is dedicated for providing general information about the main inventions that pose a threat to our modern cryptography, namely: quantum computers, Shor's algorithm, and Grover's algorithm. First, a brief introduction to quantum computers will be given, followed by a very general descriptions of Shor's and Grover's algorithms.

Cryptographic Algorithm	Type	Purpose	Impact from large-scale quantum computer
AES	Symmetric key	Encryption	Larger key sizes needed
SHA-2, SHA-3	---------------	Hash functions	Larger output needed
RSA	Public key	Signatures, key establishment	No longer secure
ECDSA, ECDH (Elliptic Curve Cryptography)	Public key	Signatures, key exchange	No longer secure
DSA (Finite Field Cryptography)	Public key	Signatures, key exchange	No longer secure

Figure 1. A table made by NIST about the "Impact of quantum computing on common cryptographic algorithms" (Chen et al. April 2016).

2.1 Quantum computing and quantum computers

This section provides readers with essential background information on the origins of quantum computing, helping them understand the rapid advancements in quantum technology. It also includes a general introduction to quantum computers, explaining why they possess such immense computing power.

2.1.1 History

The modern theory of quantum mechanics took its shape in the 1920s, as a result of the research of several physicists who independently contributed to its development over many decades. Initially, the interest in quantum mechanics was purely theoretical, as practical applications were lacking and technological development was still in its early phases. This started to gradually change in 1980 when mathematician Yuri Manin first proposed the idea of using quantum mechanical computers to simulate quantum systems (Manin 1980). Physicist Richard Feynman independently developed the concept further in 1982 (Feynman 1982; Nielsen and Chuang 2010).

In 1993, Bernstein and Vazirani proved that quantum Turing machines (formulated by Deutcsh in 1985 (Deutsch and Penrose 1985)) are capable of solving certain problems in polynomial time, whereas a classical probabilistic Turing machine requires exponential time to solve the same problem (Bernstein and Vazirani 1993). This was the first time that a computational model was able to violate a foundational principle of computer science - the extended Church-Turing thesis, which states that a probabilistic Turing machine can efficiently (meaning that the difference in speed can only be polynomial) simulate any computational devices (National Academies of Sciences and Meidicine March 2019; Nielsen and Chuang 2010).

Roughly a year later, in 1994, quantum computing research gained even more interest, when mathematician Peter Shor showed that with a sufficiently powerful quantum computer it would be possible to solve the discrete logarithm problem as well as the integer factorization problem in polynomial time (Shor 1994). As the difficulty of these mathematical problems is the basis for many cryptographic algorithms, Shor's findings and the development of quantum computers became a big theoretical threat to information security (more on Shor's algorithm in section 2.2).

In 1996, Lov Grover showed that quantum computers could be used to provide a quadratic speed-up for searches in unstructured search spaces (such as finding a specific number from an unsorted list of numbers) (Grover 1996). Together with powerful quantum computers, Grover's algorithm could pose a threat to cryptographic methods such as symmetric cryptographic keys and cryptographic hash functions. However, this threat can be overcome by doubling the size of the hashes and symmetric keys (R. A. Grimes 2019), and therefore is not as harmful as Shor's algorithm. More details on Grover's algorithm will be presented in section 2.3.

Shor's and Grover's algorithms theoretically proved that quantum computers could exponentially outperform classical computers in certain computational tasks, making quantum computing an even more interesting topic for researchers and investors around the world. This led to an increase in investments in quantum computing research and development. Already in 1998, the first 2-qubit NMR (nuclear magnetic resonance) quantum computer was built (Jones, Mosca, and Hansen May 1998), soon to be followed by another one (Chuang, Gershenfeld, and Kubinec April 1998). Now, over two decades later, as one of the quantum

computing industry leaders, IBM released their 1121-qubit quantum processor, Condor, at the end of 2023 (Gambetta 2023).

2.1.2 Quantum computer introduction

Conventional computers use bits (binary digits) as their basic unit of information, which can only have a value of 1 or 0. Since a single bit can represent a single value out of two choices (0 or 1), with two bits, there are four possible outcomes (01, 10, 11, 00) and with three bits, there are eight possible outcomes (000, 001, 010, 100, 110, 101, 011, 111). As seen from this pattern, adding a single additional binary digit increases the number of end results exponentially ($2^1, 2^2, 2^3$...). (R. A. Grimes 2019).

For quantum computers, the basic unit of information is called a qubit. A single qubits state can be 1 and 0 at the same time, at least before the qubits quantum state is measured. What makes this possible, is a quantum property known as quantum superposition. Quantum superposition is the main reason why quantum computers are capable of performing certain operations so much faster than conventional computers, as it enables each qubits bit state to be exponential to itself and to additionally added qubits. Meaning that a single qubit can be two states (0 and 1) of out two choices (0,1) at the same time, and two qubits can be four states out of four choices (00 and 01 and 10 and 11), and so on. Therefore, a similar amount of bits and qubits will have the same number of possible states, but a bit can only be one state at a time, while a qubit can be all of them simultaneously. (R. A. Grimes 2019).

However, when the value of a qubit is measured, the quantum superposition state collapses, causing it to behave similarly to a bit, providing a single binary outcome (1 or 0). Bits, on the other hand, have the same state whether they are measured or not. Because qubits are extremely sensitive to disturbance (noise), they must to be isolated from the outside world in order to maintain their quantum state, which is one of the many challenges in building quantum computers.

The qubits sensitivity to noise causes quantum computers to have high error rates, which can cause quantum algorithms to produce wrong results or even prevent the algorithm from running properly. To reduce error rates to acceptable levels, quantum computers need to run

quantum error correction algorithms and create so-called "logical" qubits, which are stable and have very low error rates. Logical qubits are essential for the successful execution of quantum algorithms such as Shor's algorithm. The issue with quantun error correction algorithms is that they are extremely resource-intensive to produce, requiring many physical qubits to create a single logical qubit. Until recently, researchers believed that at least 1000 physical qubits were needed to create a single logical qubit (Castelvecchi December 2023; Fowler et al. 2012), or a maximum of 10 logical qubits (Bravyi et al. August 2023). However, interesting research was published in December 2023, where researchers were able to use a logical quantum processor that was based on reconfigurable atom arrays to achieve an astonishing number of 48 logical qubits with only 280 physical qubits (Bluvstein et al. December 2023).

Another challenge with quantum computers is that large amounts of classical data cannot yet be efficiently converted to a quantum computer readable quantum state. Therefore, it is challenging to utilize quantum computers to analyze previously collected classical data, because the time required to convert the data into quantum state would typically be so long that the benefits of using a quantum computer would be diminished. (National Academies of Sciences and Meidicine March 2019).

Because quantum computers and qubits are so sensitive and complex, it is not feasible to simply compare the number of qubits in a quantum computer to determine which computer is more powerful, as there are many factors besides the number of qubits that affect the efficiency of quantum computers. To address the need for comparing the power and speed of quantum computers, IBM developed a metric called "quantum volume" (Cross et al. September 2019) that describes the amount of quantum work a quantum computer can do in a given period of time (R. A. Grimes 2019). Measuring the quantum volume of a quantum computer requires determining the number of qubits a quantum computer has, as well as gate and measurement error rates, device cross-talk, coherence time, connectivity (between qubits and other components), and circuit software compiler efficiency (R. A. Grimes 2019). IBM's quantum volume is not the only method used for quantum benchmarking, as cross-entropy benchmarking (Arute et al. October 2019) and IonQ's Algorithmic Qubits (Staff January 2024) have also been used. So far, there is no general consensus on which quantum bench-

marking method should be used.

2.2 Shor's algorithm

As briefly mentioned in subsection 2.1.1, Peter Shor published a paper in 1994 in which he mathematically proved that factoring large prime integers and computing discrete logarithm problems could be done in polynomial time with sufficiently powerful quantum computers (Shor 1994). Shor's algorithm can be used to refer to multiple different algorithms that were created by Peter Shor, but it is typically used to refer to the method of factoring large prime integers, as it has had the biggest impact on the security of our cryptosystems. In this paper, Shor's algorithm will be used to refer to the quantum algorithms that Shor published in his 1994 paper (Shor 1994) and if needed, the algorithm (integer factoring or discrete log) will be specified.

R. A. Grimes 2019 does a good job of explaining how Shor's algorithm works without going into too much mathematical detail: "Shor's algorithm allows quantum computers to factor prime numbers faster by using an equation that takes a purely random guess at one of the prime numbers and turns it into a much closer guess, which then quickly finds the actual prime numbers. Shor's algorithm uses the mathematical relationship of the two involved prime numbers in a way that dramatically cuts the number of guesses needed as compared to a classical brute-force method. A very large number of guesses is still needed, but when those guesses are done using the quantum property of superposition, they can be generated nearly instantaneously on a quantum gate computer. Within all those guesses are the right two prime numbers.".

Before Shor's discovery, factoring large prime integers and computing discrete logarithm problems were thought to be such difficult computational problems that they were trusted to be secure and therefore implemented into specific cryptosystems like RSA, Diffie-Hellman, and elliptic curve cryptography. Even though Shor published his paper three decades ago, these cryptosystems are still widely used because in the early 1990s quantum computers - and therefore the threat too - was only theoretical. Additionally, Shor's algorithm requires a lot of resources from a quantum computer, so despite the advancements in quantum mechanics, it

will still take some time until such quantum computers are built that could break the currently used asymmetric cryptography.

According to some estimates, breaking RSA (with key sizes ranging from 1024 to 4096 bits) with Shor's algorithm would require a quantum computer to have approximately 2050 to 8194 logical qubits. For elliptic curve cryptography (with key sizes ranging from 256 to 521 bits), it would require around 2330 to 4719 logical qubits (National Academies of Sciences and Meidicine March 2019). At the time of writing this paper, the highest number of logical qubits that has been achieved is 48 logical qubits (Bluvstein et al. December 2023), which highlights how much progress is still required in quantum computing for the threat to actualize.

Although there still seems to be some time before there is a sufficiently powerful and stable quantum computer that is capable of running Shor's algorithm, replacing these vulnerable algorithms needs to be done urgently, as new scientific breakthroughs could dramatically shorten the time until such quantum computers exist. Additionally, there is no guarantee that someone hasn't already secretly built such a quantum computer, or might do so in the future. Although very unlikely, as it would require vast investments without the possibility to reap the publicity and monetary benefits.

2.3 Grover's algorithm

Grover's algorithm is a quantum search algorithm. This means that it uses quantum mechanical properties to achieve faster search results, and in the case of Grover's algorithm, this is done within unstructured search spaces. For classical binary computers, the best way to solve this unstructured search space problem is to simply go through every value in the search space and compare it to the searched value; if the values match, the search is stopped; otherwise, the search continues until the correct value is found. This method requires at most going through all N (where N is the total number of values within the search space) values within the search space, whereas, using a quantum computer and Grover's algorithm, the same problem can be solved within a square root of N, thus providing a quadratic speedup over classical computers. (Grover 1996).

As briefly mentioned in subsection 2.1.1, Grover's algorithm was invented by Lov Grover in 1996 and it slightly threatens the safety of our modern symmetric-key cryptography and hashing algorithms. Unlike Shor's algorithm, which provides an exponential speed-up for breaking asymmetric algorithms, Grover's algorithm provides a polynomial speed-up for breaking symmetric algorithms. Therefore, the current security level of symmetric algorithms can be maintained by doubling the key and hash lengths.

Similar to Shor's algorithm, Grover's algorithm is currently only a theoretical threat, as not enough powerful quantum computers have been built to successfully run Grover's algorithm. Research suggests that about 3000 to 7000 logical qubits are needed to break AES with key lengths of 128, 192 or 256 (Grassl et al. 2016).

3 Quantum threatened cryptography

This chapter introduces the most common modern cryptographic algorithms that are affected by the development of quantum computers and the aforementioned quantum algorithms. Additionally, the concepts of symmetric and asymmetric cryptography are explained and some of the most relevant practical implementations of these cryptographic methods are presented. The mathematical side of the cryptographic functions will not be discussed in detail, as it is outside the scope of this research and has already been explained in the original reference materials.

As previously discussed, symmetric-key cryptography is believed to be safe against powerful quantum computers, provided that their key lengths are doubled. However, the currently used asymmetric-key cryptography will be unsafe against quantum computers and will therefore be replaced with quantum computer safe alternatives. The National Academies of Sciences, Engineering, and Medicine have created an interesting table of literature-reported estimates of the quantum resilience of current cryptosystems (National Academies of Sciences and Meidicine March 2019, p.98). Figure 2 is a modified and simplified version of the same table and it presents the estimates about the quantum resilience of some popular currently used cryptosystems.

Cryptosystem	Category	Key Size	Security Parameter	Quantum Algorithm Expected to Defeat Cryptosystem	Logical Qubits Required
AES-GCM	Symmetric encryption	128 192 256	128 192 256	Grover's algorithm	2,953 4,449 6,681
RSA	Asymmetric encryption	1024 2048 4096	80 112 128	Shor's algorithm	2,050 4,098 8,194
ECC, Discrete-log problem	Asymmetric encryption	256 384 521	128 192 256	Shor's algorithm	2,330 3,484 4,719
SHA256	Bitcoin mining	N/A	72	Grover's Algorithm	2,403
PBKDF2 with 10,000 iterations	Password hashing	N/A	66	Grover's algorithm	2,403

Figure 2. Estimates for the quantum resilience of modern cryptosystems. Derived from National Academies of Sciences and Meidicine March 2019, p.98.

3.1 Symmetric-key

Symmetric-key cryptography (also known as private-key cryptography) is based on a shared secret key between two (or more) communicating parties. The shared secret key is used to encrypt and decrypt the data sent between the communicating parties, so that only those who have obtained the shared key can participate in the communication. The issue with private-key cryptography is that the secret key needs to be shared remotely among the communicating parties while minimizing the risk of eavesdroppers also obtaining the secret key. The solution to this problem is asymmetric-key cryptography (also called public-key cryptography), which will be introduced in section 3.3.

Symmetric-key cryptography and hashing are unlikely to be broken by quantum computers, only weakened. Grover's algorithm will make it four times faster to break these cryptographic methods, which will require the use of longer keys to maintain the current levels of security. Currently, AES is most commonly used with 128-bit keys and SHA with 256-bit keys. Figure 3 shows NIST's classification of AES-128 at security level 1 and SHA-256 at security level 2, where level 1 represents the lowest security and level 5 the highest. This means that they are not very resilient against attacks made with quantum computers, and therefore their use in governmental and critical infrastructure systems will likely be prohib-

ited in the future. Fortunately, when the bit size of both is doubled (AES to 256 bits and SHA to 512 bits), they belong to the most secure category of said classifications and can be trusted to be quantum resistant.

NIST security level	Algorithm	Quantum resistance
5	AES-256	Strongest
4	SHA-384/SHA3-384	Very Strong
3	AES-192	Stronger
2	SHA-256/SHA3-256	Strong
1	AES-128	Weak

Figure 3. NIST security strength classifications for symmetric-key algorithms. Table derived from R. A. Grimes 2019

3.1.1 AES

In 1997, NIST set out to develop an encryption algorithm that would be secure enough to protect sensitive government data. NIST issued a call for algorithms in an effort to find promising candidates. The winner of this competition would get their algorithm standardized by NIST as the Advanced Encryption Standard (AES). The call had specific requirements for the algorithms, such as: "... the AES would specify an unclassified, publicly disclosed encryption algorithm(s), available royalty-free, worldwide. In addition, the algorithm(s) must implement symmetric-key cryptography as a block cipher and (at a minimum) support block sizes of 128-bits and key sizes of 128-, 192-, and 256-bits." (Computer Security Division December 2016). A total of 15 algorithm candidates were initially selected, of which five were chosen as finalists, and they were: MARS, RC6, Rijndael, Serpent, and Twofish. NIST couldn't differentiate the finalists based on security alone, since they seemed to be equally secure. Therefore, Rijndael was chosen based on its strong performance on most platforms and its ease of implementation to hardware (Burr March 2003). In November 2001, NIST selected Rijndael as the proposed algorithm for AES and published the Advanced Encryption Standard as Federal Information Processing Standard (FIPS) 197. (Computer Security Division December 2016). AES is a variant of the Rijndael algorithm, as some modifications were made after its selection as the encryption standard (Daemen April 2004).

Since NIST published AES in 2001, it has been the most popular symmetric cipher in the public and private sectors (R. A. Grimes 2019). In addition to its security, AES is popular due to its fast encryption and decryption speeds and ease of implementation. While quantum computers and Grover's algorithm will not completely break AES, they will require updates to hardware and software, as many systems still default to AES-128 and are not equipped to use AES-256 or higher (R. A. Grimes 2019). This is particularly true for many IoT devices, which, due to their small size, have limited resources and are therefore prone to latency issues and memory deficiencies. For a more detailed description of AES, see the FIPS Publication 197 (Dworkin May 2023).

3.2 Hashing

Hash functions are one-way functions that use the binary information of the data as part of the hash creation process. Hashing is mainly used for verifying the integrity and ensuring the authenticity of data (see figures 4 and 5), but it is also used for secure password management and in the bitcoin mining process (Sampaio de Alencar 2022).

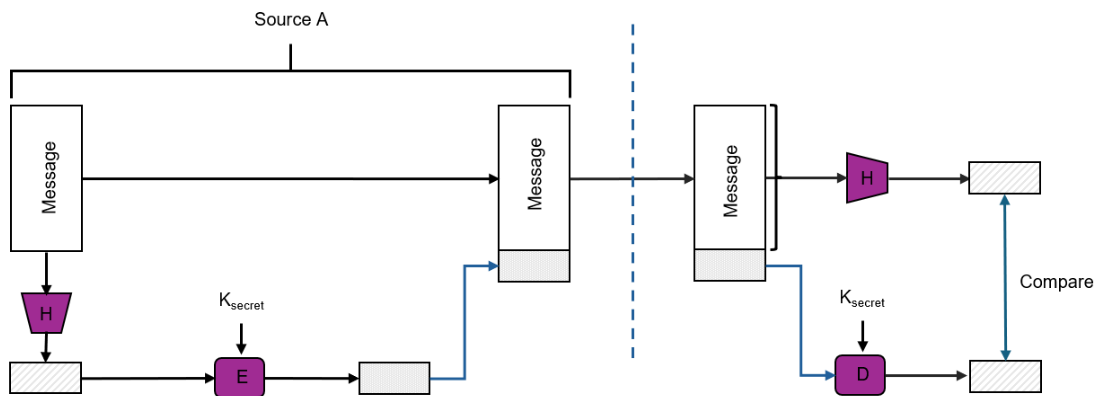

Figure 4. Message authentication using symmetric encryption (Fall and Stevens 2011).

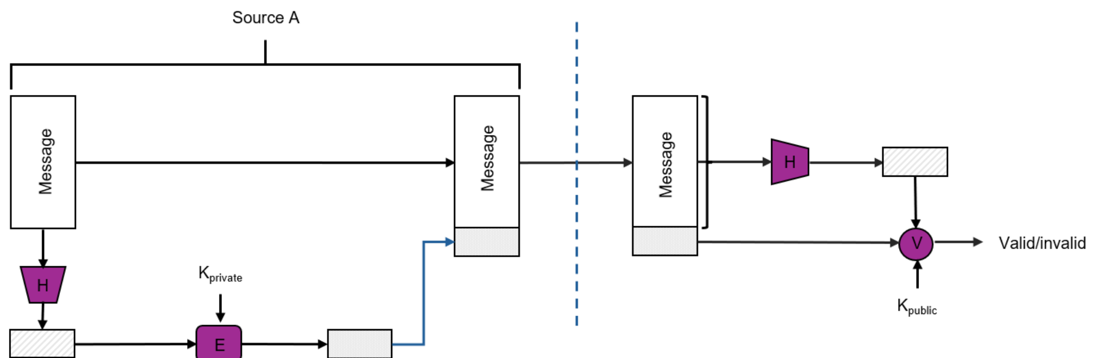

Figure 5. Digital signature using public-key encryption (Fall and Stevens 2011).

From a security perspective, it is unwise to store passwords as plaintext, which is why most UNIX-based operating systems instead store the hash values of passwords. To add a layer of security, the system combines the password with a salt value, which is typically a pseudorandom number, and passes this value as an input to a slow hash function that outputs a

hash code (see figure 6). A slow hash function is used to make attacks against the system more laborious. The hashed password and a plaintext copy of the salt are then stored in a password file that corresponds to the user ID. When a user provides the system with a user ID and a password, the system uses the user ID to find the correct password file from which it retrieves the hashed password and the plaintext salt. The salt and the user-given password are provided as input into the hashing algorithm, and the resulting hash value is compared to the stored password hash value. If the results match, the password is accepted. (Stallings 2018, p. 315).

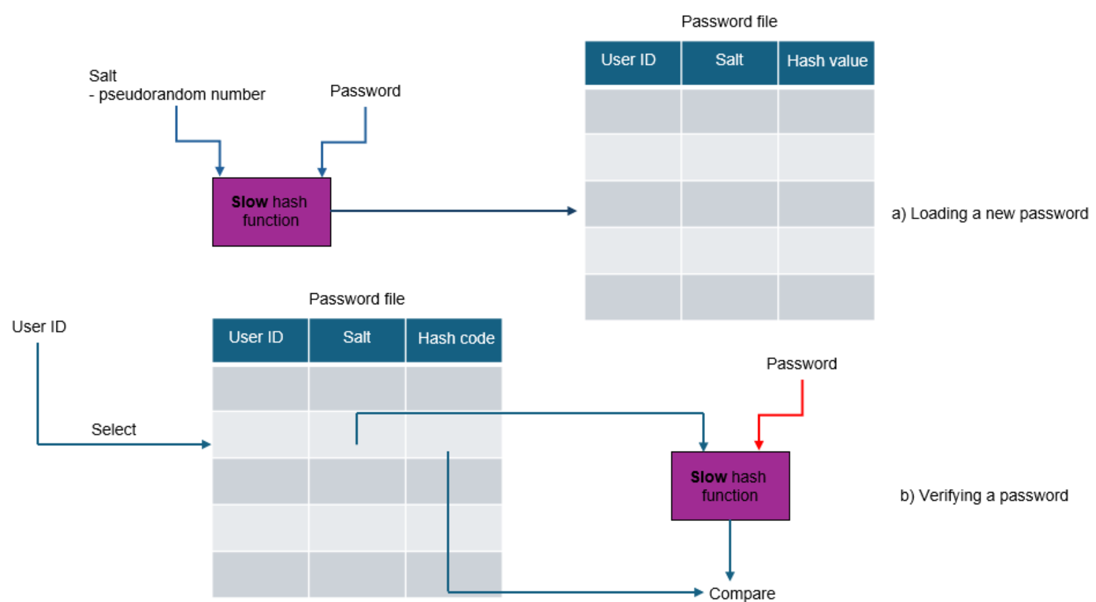

Figure 6. The use of hashed passwords (Stallings 2018, p. 315).

Hashing algorithms typically produce hash values of fixed length, and the length varies depending on the hashing algorithm. If even a small portion of the original data is altered (for example, a single character in the message is changed) and then hashed, the new hash value would be completely different from the original hash value. Because hashing functions typically repeat the same hash operation, and only the input data affects the output hash value, a single unaltered data file will always produce the same hash value when the same hashing algorithm is used.

Key aspects of hashing algorithms include hashing speed and minimizing duplicate hash values, known as hash collisions. This occurs when two or more distinct data sets produce

the same hash value. Hash collisions are a security concern and can be exploited for different types of malicious activities, such as the hash collision attack (Liang and Lai January 2007). Another important feature of a good hash algorithm is irreversibility, which means that it is impractical to revert the hash back to the original message.

Good hash algorithms rarely have collisions, and it should be difficult to intentionally create them. One way to increase the rarity of hash collisions is to have an algorithm that randomly selects the hash algorithm used from a set of hash algorithms, making it more difficult for the malicious actor to at least use the system's own hash function to try to figure out the original message.

Currently, Secure Hash Algorithm-2 (SHA-2) is the most commonly used hashing algorithm and it can be used with different output sizes like 224, 384, 256 and 512 bits (R. A. Grimes 2019). Particularly, SHA-256 is widely used, even though NIST published an upgraded version, SHA-3, in 2015 (Technology August 2015). The main reason for SHA-256's popularity is its balance between speed and security; while SHA-3 is more secure, it can be up to two times slower than SHA-512 (R. Grimes February 2018). When SHA-3 was standardized, most hardware and software did not support the fast computation of the SHA-3 underlying Keccak function, which caused it to be slower and less popular than the still-secure SHA-2. However, with the correct hardware and software, SHA-3 is faster and more secure than SHA-2, which is why it is likely to become more common in the future as SHA-2 eventually weakens and the majority of hardware supports the fast computation of the Keccak function.

The security of hashing typically relies on three important properties (Fall and Stevens 2011, p. 817):

- **Preimage resistance**: It should be difficult to determine the original message from the hash function's output value.
- **Second preimage resistance**: For a specified input, it is computationally infeasible to find another input which produces the same output.
- **Collision resistance**: It should be difficult to find two distinct messages that produce the same hash value.

Note: In a system that is under attack, the difference between Second Preimage Re-

sistance (SPR) and Collision Resistance (CR) is as follows:

- SPR: The attacker has obtained a message and its hash value and is trying to find another message that produces the same hash value.
- CR: The attacker tries to find two messages that produce the same hash value; the goal could be to weaken a digital signature, for example.

These security properties are useful against modern computers, but the exponential increase in computing power provided by quantum computers, along with the speedup offered by Grover's algorithm, poses a risk to the currently used hashing algorithms. Hence, it is likely that in the near future SHA-256 will no longer be secure, and SHA-512 will need to be used to maintain the same level of security in hashing that we have today.

3.3 Asymmetric-key

Asymmetric-key cryptography is mostly used for digital signatures and key establishment. In other words, to authenticate and to establish a secure communication channel between two or more parties over the internet. This is achieved by having the communicating parties create their own public and private key pairs, sharing the public keys with each other, and keeping the private keys confidential. So if Alice wants to send a message to Bob, she can use Bob's public key to encrypt the message, which can then only be decrypted and read by using Bob's corresponding private key. Public keys are often distributed using digital certificates signed by a trusted entity known as the Certificate Authority (CA). This helps ensure the authenticity of the public key.

Although asymmetric-key cryptography can encrypt messages, it is much slower and less efficient than symmetric-key cryptography. Therefore, symmetric-key cryptography is mainly used to encrypt data, while asymmetric-key cryptography is used to securely share the secret key (or session key) used in symmetric encryption. This hybrid cryptosystem is the most common way to use asymmetric and symmetric-key cryptography because they complement each other perfectly. R. A. Grimes 2019 even called the discovery of public-key cryptography as the "holy grail" of encryption to emphasize its significance: "The holy grail in encryption was to find a method that allowed two or more parties to exchange symmet-

ric keys across an untrusted (even knowingly malicious) communications channel without having to first establish ahead of time a private communication method to exchange the symmetric-keys for each participant".

Public-key encryption methods typically rely on three main computationally difficult mathematical problems: integer factorization, discrete logarithms, and elliptic curves (Buchanan 2017). However, quantum computers are expected to efficiently solve all of these problems, rendering them insecure for use in future cryptographic algorithms.

3.3.1 Diffie-Hellman

The Diffie-Hellman key-exchange is a cryptographic algorithm for establishing a secure communication channel in insecure public networks. It was invented by Whitfield Diffie and Martin Hellman in 1976 (Diffie and Hellman 1976) and was one of the first public key cryptosystems. Ralph Merkle also contributed significantly to the invention of the algorithm, which is why it's also called the Diffie-Hellman-Merkle key exchange, as suggested by Hellman in 2002 (Hellman May 2002). Unfortunately, the extended name didn't quite catch on, so Merkle's contribution isn't as well known.

With the Diffie-Hellman key exchange, two entities that wish to establish a secure and private communication channel first agree on a shared public key over an insecure public channel. Then, using specific mathematical operations, they combine their personal private keys with the shared public key and exchange the resulting new public keys. Finally, they use their own private keys and mathematically combine them with the "new public key", to create a shared secret key that is ultimately the same for both participants. Now, through mathematical calculations, both entities have the same shared symmetric key, without ever exposing the entire secret to the insecure public channel. The shared symmetric key can then be used to efficiently and securely encrypt and decrypt messages that they send to each other. (Diffie and Hellman 1976; Ahmed et al. November 2012).

Although some information is transmitted over an insecure public channel, the security of the secret key is not compromised because the Diffie-Hellman key exchange is based on the computationally difficult discrete logarithm problem (Diffie and Hellman 1976). Attempting

to solve the secret key from the bits of shared information would require too many computational resources and too much time to be a practical option for malicious actors. However, in his paper, Peter Shor proved that powerful quantum computers could efficiently compute discrete log problems (Shor 1994), making Diffie-Hellman vulnerable.

Diffie-Hellman is typically used only in key-exchange implementations, and its typical key sizes range from 2048 to 4096 (R. A. Grimes 2019). Practical implementations of Diffie-Hellman include applications such as virtual private networks (VPNs), secure shell (SSH), and secure file transfer protocol (SFTP) (Buchanan 2017). Additionally, Diffie-Hellman is used in SSL (Freier, Karlton, and Kocher August 2011), S/MIME (Schaad, Ramsdell, and Turner April 2019), and OpenPGP (Finney et al. November 2007).

3.3.2 RSA (Rivest-Shamir-Adleman)

RSA was invented by Ron Rivest, Adi Shamir, and Leonard Adleman in 1977 (Gardner August 1977), and the formal paper was published in 1978 (Rivest, Shamir, and Adleman 1978). The name RSA is derived from the initials of the inventors. It is by far the most commonly used asymmetric-key encryption method, possibly accounting for up to 95 percent of all asymmetric-key encryption use cases (National Academies of Sciences and Meidicine March 2019, p. 68).

RSA is commonly used for digital signatures and to exchange shared keys used in symmetric encryption schemes. While symmetric encryption schemes, such as AES, use small 128- to 256-bit keys for encryption, RSA's key lengths range from 1024 to 4096 bits, which makes it slow and computationally expensive as a primary encryption scheme. Especially since 1024-bit RSA keys are now considered weak, the recommendation is to use at least 2048-bit keys (Barker and Dang January 2015). As a result, RSA is often used in combination with symmetric-key cryptography in hybrid cryptosystems, where RSA provides a secure method for exchanging private keys, and symmetric-key algorithms handle the actual data encryption. RSA is widely used in various security protocols and applications, including SSL/TLS (Krawczyk, Paterson, and Wee 2013), SSH ("How to Use ssh-keygen to Generate a New SSH Key?", no date), GPG and PGP (Finney et al. November 2007), S/MIME

(Schaad, Ramsdell, and Turner April 2019), digital certificates, key management protocols, and DNSSEC (Wouters and Surý June 2019).

The security of RSA is based on the difficulty of factoring large prime numbers. The larger the prime number, the longer and more secure is the resulting key. This means that the length of the RSA keys is directly related to their robustness. However, even though longer keys generally provide greater security, they also require more computational resources. RSA is already considered slow and computationally expensive, so simply making keys infinitely longer is not a viable option to save RSA from the threats posed by Shor's algorithm and quantum computers.

Other important security practices for RSA include key rotation and secure storage practices, as well as the use of padding schemes to prevent chosen ciphertext attacks. One such scheme is Optimal Asymmetric Encryption Padding (OAEP), which ensures that the input to the RSA algorithm is unpredictable, thus reducing the risk of successful chosen ciphertext attacks (Jonsson and Kaliski February 2003).

3.3.3 Digital Signature Algorithm (DSA)

As its name suggests, the Digital Signature Algorithm is used to generate and authenticate digital signatures. It is derived from the ElGamal signature scheme (Elgamal July 1985) and its security relies on the discrete logarithm problem (Kravitz July 1993). DSA was invented by a former NSA employee, David W. Kravitz, and the patent was granted to "The United States of America, as represented by the Secretary of Commerce, Washington, D.C." (Kravitz July 1993). In 1994, DSA was specified in the U.S. Government's Federal Information Processing Standard (FIPS) 186 and was titled the Digital Signature Standard (DSS) (Standards and Technology May 1994). DSA is no longer approved for digital signature generation in the latest FIPS publication (Chen et al. February 2023). More secure algorithms like RSA, EdDSA or ECDSA should be used instead.

OpenSSH already disabled DSA back in 2015 from all the OpenSSH versions starting with 7.0. They claimed it was too weak and therefore advised against using it (”OpenSSH Legacy Options”, no date). Unfortunately, despite DSA's omission from FIPS 186-5 and OpenSSH,

no inherent weaknesses or explanations for DSA's omission were found during this research. One possible explanation could be that DSA couldn't compete with better alternatives and was therefore deemed impractical. Since DSA uses large keys, ranging from 1024 bits to 3072 bits, it cannot compete with RSA's versatility (RSA can be used for multiple tasks, while DSA only handles signatures) or with the much smaller key sizes and faster computations of more modern digital signature algorithms like ECDSA. Additionally, 1024-bit keys are considered weak for DSA, so at least 2048-bit keys are recommended. However, such long keys are too resource-intensive for digital signatures, and much better alternatives are available.

3.3.4 Elliptic curve cryptography-based algorithms

Elliptic curve cryptography (ECC) was developed independently by Neil Koblitz in 1985 (Koblitz, Menezes, and Vanstone March 2000) and Victor Miller in 1986 (Miller 1986). It is based on the mathematical properties of elliptic curves and its security relies on the difficulty of the elliptic curve discrete logarithm problem. Elliptic curve methods are typically implemented in other discrete-logarithm problem-based cryptographic algorithms (such as DH and DSA), which increases their complexity and thus enables faster computation and smaller keys and certificates while maintaining the same level of security (Johnson, Menezes, and Vanstone August 2001). For the reasons mentioned above, the relatively new ECC's have gained much popularity over older cryptographic algorithms like RSA, DSA, and DH, which need to use very large keys to provide sufficient security, as can be seen from figure 7. Additionally, small key and signature sizes are major advantages, especially when dealing with IoT devices, which tend to have limited computational resources (Johnson, Menezes, and Vanstone August 2001).

Some of the most relevant ECC algorithms are Elliptic Curve Diffie-Hellman (ECDH) (Housley August 2018), Elliptic Curve Digital Signature Algorithm (ECDSA) (Standards and Technology February 2023a), and Edwards-Curve Digital Signature Algorithm (EdDSA) (Bernstein et al. 2011). These ECC algorithms can be viewed as enhanced versions of their traditional cryptographic counterparts, as they offer improved security with smaller key sizes and faster computations. The elliptic curve "variants" serve the same purposes as the algo-

rithms from which they are derived, meaning that ECDH is used for key exchange implementations, while ECDSA and EdDSA are utilized for digital signatures. ECC's are used in applications and protocols such as TLS/SSL, PGP, IKE for IPsec, SSH, Bitcoin, Wi-Fi, Bluetooth security, and smart cards.

Bits of security	Symmetric key	Hash functions	RSA/DH	ECC
112	3-key Triple-DES	SHA-224	2048	224
128	AES-128	SHA-256	3072	256
192	AES-192	SHA-384	7068	384
256	AES-256	SHA-512	15360	512

Figure 7. Estimated cryptographic algorithm security levels based on key lengths. Table derived from (Hankerson and Menezes 2011)

4 Quantum resistant solutions

Post-quantum cryptography (PQC) refers to cryptographic algorithms that are designed to be secure against attacks made on both classical and quantum computers, and should not be confused with quantum algorithms, which are algorithms that are designed to use quantum mechanical properties (like Shor's and Grover's algorithms). This chapter introduces the most common types of PQC algorithms and the most promising PQC algorithm solutions that aim to replace the vulnerable public-key cryptography.

4.1 Families of PQC algorithms

There are four main types of PQC algorithm categories, of which lattice-based algorithms are the most prominent among the NIST PQC algorithm candidates. However, NIST acknowledges that vulnerabilities may be found in lattice-based (or any other) PQC algorithms in the future, and therefore seeks to standardize several different types of PQC algorithms as a precaution. In the event that a single type of PQC algorithm family was deemed to be insecure, there would still be other types of secure and already standardized PQC algorithms that could be quickly be used as replacements for the vulnerable PQC algorithm type.

This section introduces the four most common types of PQC algorithm categories. Other types of PQC algorithm categories, such as isogeny-based cryptography, which are less prominent in the NIST PQC standardization project,are left out of the scope of this study. For example, SIKE was the only isogeny-based candidate, and it was disqualified in the third round for being insecure (Moody 2022).

4.1.1 Code-based

The history of code-based cryptography dates back to 1978, when Robert McEliece first proposed a code-based encryption scheme, the McEliece cryptosystem (McEliece. April 1978). Such schemes are based on the use of error correcting codes (ECC), and their security is based on the difficulty of correcting errors for random linear codes (Cybersecurity May 2021). Essentially, specific algorithms are used to create "errors" in plaintext so that

its original message is obscured (encrypted), and corresponding "error-correcting" codes are used to restore the plaintext to its original form (decrypted) (R. A. Grimes 2019).

When the McEliece cryptosystem was invented, there was no knowledge of quantum computers and the threats that they would pose to cryptography, but coincidentally, the code-based cryptosystems appear to be quantum-safe when presented with sufficiently large keys (Mavroeidis et al. 2018). Due to their long history, code-based encryption schemes are among the most studied post-quantum schemes (Cybersecurity May 2021) and have so far withstood decades of security testing. Nevertheless, the McEliece cryptosystems and other code-based schemes haven't gained much popularity because they suffer from very large key sizes. While the encryption and decryption processes are fast, overly large keys were unappealing when computational resources were more limited than they are today. Even for some modern technologies, such as IoT devices, the large key sizes can be a problem. In a 2009 study, the security levels and key sizes of different algorithms were compared, and the results were as follows: to provide an 80-bit security level, RSA had a public key size of 1kb and DSA and DH had 2kb, while the McEliece encryption key was 500kb and the signature key was 4000kb (Perlner and Cooper 2009). However, regardless of the large key size, NIST is currently evaluating a version of the original McEliece cryptosystem, called "classic McEliece", for PQC standardization, as it appears to provide sufficient security against quantum computers. Classic McEliece is presented in more detail in the subsection 4.3.1.

4.1.2 Hash-based

As the name suggests, hash-based cryptography is based on hash functions, which was already covered in section 3.2. Hash functions have been used and studied for a long time, so their strengths and weaknesses are well understood. SPHINCS+ is the only hash-based signature scheme selected for PQC standardization, and will be discussed in more detail in subsection 4.2.4.

4.1.3 Lattice-based

Lattice-based cryptography is based on hard mathematical problems, such as the Short Vector Problem (SVP), Learning With Rounding (LRW), Learning With Errors (LWE), and their variants. The use of lattices in cryptography was first proposed by Ajtai in 1996 (Ajtai 1996) and it was based on the short vector problem. Another major milestone in lattice-based cryptography was achieved in 2005 when Regev introduced the use of LWE in lattice-based cryptography (Regev 2005). Since Ajtai's discovery, lattice-based cryptography has been studied and developed continuously, but prior to the emerging quantum threat, it wasn't widely used because of its relatively large key sizes. Additionally, there is still a healthy suspicion in the cryptography community about the security of lattice-based schemes because they are based on difficult mathematical problems that are not yet well understood. For now, lattice-based cryptography is believed to be safe against classical and quantum computer attacks, and therefore NIST has already selected a couple of lattice-based algorithms, Crystals Kyber and Crystals Dilithium, to be standardized. Of all the algorithms submitted to NIST's PQC evaluation, lattice-based cryptography has been the most popular (R. A. Grimes 2019). Of these, most have been based on the problems of Module Learning With Errors (MLWE) and Module Learning With Rounding (MLWR) (Cybersecurity May 2021).

R. A. Grimes 2019 provides a great simplified explanation of the use of lattices in cryptography: "With lattice-based cryptography, a complex lattice function is created as the private key. The public key is generated as a modified version of the original lattice. Content is encrypted using the modified version (the public key), and only the holder of the original lattice version (the private key) can easily recover the encrypted message back to its original plaintext state".

4.1.4 Multivariate-system based

Multivariate cryptography is based on the difficulty of solving systems of multivariate polynomials over finite fields (Chen et al. April 2016). They have been thoroughly studied and researched since the late 1980's, when the first multivariate cryptosystem was proposed by Matsumoto and Imai 1988, and are believed to be secure against quantum computers. How-

ever, many such schemes have since then been broken or found to be vulnerable to classical computer attacks, so their usability and security is still uncertain. Of the nineteen signature schemes that were submitted to the NIST PQC project, seven were multivariate based, and only two of them, Rainbow and GeMMS, proceeded to the third round of the evaluation process. In the end, they didn't make it past the third round because they required too large performance trade-offs to achieve adequate levels of security against known attack scenarios (Cybersecurity May 2021; Alagic et al. September 2022).

Multivariate algorithms are not well suited for encryption/decryption because they tend to have very large public keys and slow decryption. Instead, they are better suited for signature schemes, since despite of their large public keys (160 KB or more), they can provide very short signatures (as small as 33 bytes). (Cybersecurity May 2021).

4.2 NIST solutions

In the third round of the NIST PQC evaluation competition, one Key Encapsulation Mechanism (KEM) algorithm (Kyber) and three digital signature algorithms (Dilithium, Falcon, and SPHINCS+) were selected for standardization. Although these algorithms have undergone extensive security and performance testing, they are still relatively new and may have undetected vulnerabilities. For this reason, it is currently advised to use hybrid cryptosystems rather than moving directly to using only post-quantum cryptography. Hybrid cryptosystems use both conventional cryptographic algorithms and PQC algorithms simultaneously. The conventional cryptographic algorithms work as a safety precaution in case the PQC algorithms prove to be insecure. Next, brief introductions are provided for each NIST PQC competition (third round) finalist.

4.2.1 Crystals - Kyber

Kyber is the only KEM that was selected for NIST PQC standardization in the third round of the PQC project. As such, NIST is assigning it as the primary PQC algorithm to be used for quantum-safe key exchanges. It is based on structured lattices and, according to NIST: "has good all-around performance and security" (Moody March 2024). Its security

is based on the Module Learning With Errors (MLWE) problem, and it uses a square matrix (instead of the usual rectangular matrix) as its public key. Additionally, Kyber's MLWE problem has been modified to use polynomial rings instead of integers. It is IND-CCA2 (Key-indistinguishability under chosen ciphertext attacks) -secure against chosen ciphertext attacks and can be used in three different sizes, all of which achieve different levels of security. See figure 8 for comparisons of AES and KYBER security levels. Kyber's public keys range from 800 to 1568 bytes, secret keys from 1632 to 3168 bytes, and ciphertexts from 768 to 1568 bytes. It is mainly used with the following three parameter sets: Kyber-512, Kyber-768 and Kyber-1024. (Schwabe December 2020).

Bits of security	NIST Security level classification	AES	KYBER
128	1	AES-128	KYBER-512
192	3	AES-192	KYBER-768
256	5	AES-256	KYBER-1024

Figure 8. Comparison of the security levels of KYBER and AES.

4.2.2 Crystals - Dilithium

Dilithium is one of the three digital signature algorithms selected for standardization in the third round of NIST's PQC standardization project. NIST proposes it to be used as the primary PQC digital signature scheme because of its efficiency, security, and relatively simple implementation (Moody 2022). Like Kyber, Dilithium is also based on structured lattices. Dilithium's security is based on the MLWE problem and it is EUF-CMA (Existential Unforgeability under Chosen Message Attacks) secure. It has three different versions: Dilithium2, -3 and -5 and their public key sizes range from 1312 to 4595 bytes and signatures from 2420 to 4595 bytes (Schwabe February 2021).

4.2.3 Falcon

Falcon was selected for PQC digital signature standardization as an alternative solution. Its name is derived from "Fast-Fourier Lattice-based Compact Signatures over NTRU", and as the name suggests, it is based on structured lattices. Some of the main reasons for its selection were fast verification speed, small bandwidth, and good security (Moody 2022). However, its implementation can be complicated for some applications, which is one of the reasons why it "lost" to Dilithium. The underlying hard problem for Falcon is the Short Integer Solution (SIS) over NTRU lattices. Falcon has two versions: Falcon-512 and Falcon-1024. According its creators, Falcon-512 is roughly equivalent to RSA-2048, with a public key size of 897 bytes and a signature size of 666 bytes. Using the same test equipment, Falcon-1024 achieved a public key size of 1793 bytes and a signature size of 1280 bytes. (”Fast-Fourier Lattice-based Compact Signatures over NTRU”, no date).

4.2.4 SPHINCS+

SPHINCS+ is a stateless hash-based signature scheme. It includes parameter sets based on three different hash functions (specified in the name): SPHINCS+-SHAKE256, SPHINCS+-SHA-256, and SPHINCS+-Haraka (Schwabe August 2023). NIST accepted only the SHAKE- and SHA-versions for standardization and proposes using them with parameter sets: SHA2-128, SHA2-192, SHA2-256, SHAKE128, SHAKE192 and SHAKE256 (Standards and Tech-

nology August 2023b). Security is one of the main reasons why SPHINCS+ was selected, but also because, unlike the other third-round finalists, it is not based on lattices and therefore remains a secure PQC alternative, even if lattice-based cryptography is deemed insecure. Of the three digital signature algorithms selected, it has the worst performance (slow and large signatures). SPHINCS+ has very small public keys, ranging from 32 to 64 bytes, but the signatures range from 7856 to 49856 bytes, which is much larger compared to the other third round finalists (A. e. al. November 2022; Standards and Technology August 2023b).

4.3 Possible solutions

This section presents the three KEMs that are currently being evaluated for standardization in the fourth round of the NIST PQC project. While Kyber is lattice-based, all of the KEMs in the fourth round are code-based. NIST estimates that the fourth round will conclude in the fall of 2024, resulting in the selection of additional KEMs for PQC standardization. Figures 9 and 10 compare the performance differences (in NIST security category 1 and 3) of the PQC third round finalist KEM (Kyber) and the fourth round contestants.

Algorithm	Private key (bytes)	Public key (bytes)	Ciphertext size (bytes)
Kyber-512	32	800	768
BIKE	2801	1540	1572
HQC	56	2249	4497
mceliece348865f	6492	261120	96

Figure 9. Comparison of KEM's in the NIST security category 1. Derived from R. e. al. April 2024.

Algorithm	Private key (bytes)	Public key (bytes)	Ciphertext size (bytes)
Kyber-768	32	1184	1088
BIKE	418	3082	3114
HQC	64	4522	9042
mceliece460896f	13608	524160	156

Figure 10. Comparison of KEM's in the NIST security category 3. Derived from R. e. al. April 2024.

4.3.1 Classic McEliece

Classic McEliece is based on the McEliece cryptosystem, which was already mentioned in section 4.1.1. As Classic McEliece inherits the McEliece cryptosystems stable security history of over 40 years (B. e. al. April 2024), it is the most researched and trusted PQC algorithm in the PQC standardization project. Classic McEliece is based on binary Goppa codes and it is designed to be an IND-CCA2 secure KEM even against quantum computers ("Classic McEliece: Intro" March 2023). It has fast encapsulation/decapsulation algorithms and the smallest ciphertexts in the project, but also the largest public keys and slow key generation (B. e. al. April 2024) (Moody March 2024). There are four main variants: mceliece348864, mceliece460896, mceliece6688128, mceliece6960119, and mceliece8192128. Their ciphertexts range from 96 to 208 bytes and their public keys range from 261120 to 1357824 bytes ("Classic McEliece: Implementation" October 2022). Although Classic McEliece has extremely large public keys, its creators argue that with CCA security, key pairs can be reused, and therefore the drawbacks of having large public keys may not be as serious in practice (B. e. al. April 2024).

4.3.2 BIKE

BIKE (derived from Bit Flipping Key Encapsulation) is a code-based KEM scheme that is instantiated with Quasi-Cyclic Moderate Density Parity-Check (QC-MDPC) codes (Suite, no date). Of the fourth round candidates, it has the best performance with public key sizes ranging from 1541 to 5122 bytes, and ciphertext sizes ranging from 1573 to 5154 bytes ("BIKE - Open Quantun Safe" October 2022). BIKE has some similarities to Classic McEliece, as it is based on the Niederreiter cryptosystem (proposed by Harald Niederreiter in 1986), which is a variant of the McEliece cryptosystem (Niederreiter 1986). Classic McEliece and BIKE both use error-correcting codes in their public keys, but with BIKE, the public key can be compressed due to its quasi-cyclic structure (Cybersecurity May 2021). For decoding, BIKE uses the Black-Gray-Flip (BGF) decoder, which repeatedly flips the input bits that seem most likely to be errors (Cybersecurity May 2021; Aragon October 2022). BIKE has parameter sets for NIST security levels 1, 3, and 5, with each set being referred to accordingly: BIKE-L1, BIKE-L3, and BIKE-L5 ("BIKE - Open Quantun Safe" October 2022; Labs October

2023).

4.3.3 HQC

HQC (Hamming Quasi-Cyclic) is an IND-CCA2 secure code-based KEM that is based on the problem of decoding random quasi-cyclic codes. One of the advantages of HQC is that the decoding of cyclic codes has a long history (Prange September 1962), which gives credibility to its security. On the other hand, the decoding process is subject to failures, which is a minor inconvenience. The failure rate is extremely low and would only require an additional round of decryption (R. A. Grimes 2019), but it is a fault nonetheless. HQC has three different instances: HQC-128, HQC-192, and HQC-256, with public key sizes ranging from 2249 to 7245 bytes and ciphertext sizes ranging from 4433 to 14421 bytes (”HQC - Open Quantum Safe” April 2023).

5 Bill of Materials

A Bill of Materials (BOM) is a structured list of items that can have a variety of use cases depending on the type of BOM that is used. Typically, BOM's are used to describe all of the items needed to create a finished product and can thought of a set of instructions for product engineering, design, manufacturing, and repairing. For many organizations, BOMs are an essential part of supply chain management. Although BOMs were originally designed for product manufacturing and engineering, over the years they have been adopted in several different areas and use cases, such as: finance, marketing, human resources, accounting, and information technology (Formlabs September 2020). In the information technology field, BOMs are primarily used for software (Software Bill of Materials, SBOM), hardware (Hardware Bill of Materials, HBOM), and the newest addition is cryptography (Cryptography Bill of Materials, CBOM). For the purposes of this study, only SBOMs and CBOMs are relevant and will be discussed in more detail below.

5.1 Software Bill of Materials (SBOM)

A software bill of materials is a nested inventory of the software components and dependencies that make up a software product. It is typically used to manage supply chains and improve software security, which is why it is included in many guides related to Cybersecurity Supply Chain Risk Management (C-SCRM) (Agency January 2024; Boyens et al. October 2021). SBOMs can be depicted in various file formats, but XML and JSON are the most common because they are supported by most standards. Analyzing SBOMs provides information about software vulnerabilities, licenses, and versions, making it easier to manage risks and outdated software components. Using SBOMs also promotes software transparency, which enables software consumers to make more informed decisions about the products they purchase. The importance of software transparency was highlighted in 2020-2021 by the infamous SolarWinds ("SolarWinds Security Advisory" April 2021) and Apache Log4j (Cybersecurity and Agency April 2022) incidents, which revealed how compromised software updates and widespread dependencies on a single open source component can lead to significant security risks, emphasizing the need for SBOMs and effective software com-

ponent management.

There are three main SBOM standards: the Open Worldwide Application Security Project (OWASP) foundation's CycloneDX, the Software Package Data eXchange (SPDX) by The Linux Foundation, which is defined in the ISO/IEC 5962:2021 standard, and finally, the NIST-created Software Identification (SWID) tagging, which is defined in the ISO/IEC 19770-2:2015 standard (Telecommunications and Administration 2021). Although these standards have many similarities, they have their own main focus areas and supported file types that guide the decision for choosing one over the other. SWID tag's main focus is on software lifecycle management, and it is a "standardized XML format for a set of data elements that identify and describe a software product" (Waltermire et al. April 2016). SPDX's strength and focus is on licensing, and it supports JSON, YAML, tag/value, and RDF/XML formats. Since OWASP is a foundation focused on cybersecurity, CycloneDX is naturally focused on software security and it supports JSON and XML file types. In a study by Stalnaker et al. 2024, out of 50 surveyed participants, 16 reported using SPDX, 8 CycloneDX, 12 used both, and SWID was only used by 5 respondents. While this is by no means conclusive, it does gives insight into the popularity of each of the three standards.

The popularity of SBOMs has been growing steadily, driven by the increase in cybersecurity threats and awareness, as well as regulatory requirements or recommendations from governments and international organizations (Rangari March 2023). For example, the U.S. government strongly advocates and requires SBOMs from software suppliers that sell products to the U.S. federal agencies (House May 2021; Anchore February 2024). Additionally, the European Union (EU) Cyber Resilience Act (CRA) suggests that the use of SBOMs is likely increase in the EU, as the objectives of the CRA and the benefits of SBOMs go hand in hand (Commission September 2022). According to The Linux Foundation's report in January 2022, of the 412 surveyed organizations, 47% were already using SBOMs and 76% had some level of SBOM readiness (Stephen Hendrick January 2022). In addition, the report predicts that 78% will be using SBOMs by the end of 2022 and 88% by 2023. However, according to research by Xia et al. May 2023, of 17 interviewed and 65 surveyed SBOM practitioners, 83.1% agreed that "most existing third-party software or components, either open source or proprietary, are not equipped with SBOMs", which contradicts the predic-

tions of The Linux Foundations report.

Altough SBOMs can increase software security and transparency when correctly implemented, there are still some unresolved challenges. The lack of an industry SBOM standard and the pressure to meet government deadlines can drive immature SBOM production and consumption. In addition, too much focus on SBOMs can distract attention from other security issues, such as network vulnerabilities. The immaturity of SBOM generation tools is also an issue, as many tools only support specific environments and generate different types of SBOMs, requiring SBOM consumers to spend a significant amount of time and resources selecting several different SBOM generation tools, customizing them to their needs, and testing to ensure that they work properly. This problem is highlighted by the findings of Zahan et al. March 2023, which indicates that practitioners want tools that can automatically generate SBOMs for different use cases, thereby reducing the burden of SBOM implementation. There are also concerns that the public distribution of SBOMs could make it easier to find exploits within an application, since all of the software used and its versions can be found in the SBOM document (”Why we need to put the brakes on public software bills of material” June 2021). However, this concern promotes the ideology of "security through obscurity", which is a criticized security practice that should not be relied upon. It is even listed in the Common Weakness Enumeration project as one of the top 25 most dangerous software weaknesses for 2023 (Corporation October 2023). (Martins December 2023).

5.2 Cryptography Bill of Materials (CBOM)

A Cryptography Bill of Materials is similar to an SBOM, but focuses on describing the cryptographic components and dependencies of a piece of software. CBOMs are used to discover, manage, and report cryptographic assets, which promotes cryptographic agility and makes it easier to identify weak cryptographic algorithms (Foundation April 2024b). Although CBOM documents could be embedded in SBOMs (if they follow the same standard), it is important to manage them separately for the sake of cryptographic agility. Additionally, recreating an entire SBOM document because of a minor change in cryptographic algorithms would be a waste of resources.

CBOMs are still a very new concept and there are only a few organizations involved in creating CBOM-related standards or tools. CBOM was originally developed by IBM as a part of their commercial quantum-safe Explorer tool, which is marketed to aid in the transition process to quantum-safe systems and applications (IBM September 2023). IBM's CBOM scheme is an extension of the CycloneDX SBOM scheme, and altough the tool for creating CBOMs is only commercially available, IBM's CBOM scheme is publicly available on GitHub (IBM April 2024). In addition to IBM's CBOM scheme and tool, there is the recently released (April 2024) CycloneDX CBOM scheme (Foundation April 2024a), and the OWASP SBOM tool's (cdxgen) added functionality of creating CBOMs for Java projects ("CycloneDX Generator" May 2024). Santander Security Research also has their cryptobom-forge tool, which can create CBOMs out of CodeQL outputs ("CodeQL" May 2024; Research January 2024). Apart from these, no other CBOM tools or standards were found during this research, highlighting the immaturity of the field. However, as preparations for PQC migration begin and new cryptographic policies and advisories emerge (Agency September 2022; House May 2022; "OMB M-23-02" November 2022), the need for CBOMs and CBOM tools increases. It is likely that many SBOM generator tools will soon be modified to include a CBOM generation functionality, just like cdxgen, since the main concept behind generating SBOMs and CBOMs is very similar. On GitHub, the topic tag of "sbom-generator" returns 75 public repositories, which gives some idea of how many SBOM generators might be available if all the other SBOM generators (such as commercial tools) are included in this list. If even a small fraction of these eventually include CBOM generation, in addition to the newly emerging separate CBOM tools, the number of CBOM generators is likely to increase dramatically in the future.

The threat that quantum computers pose to modern public-key cryptography will require a significant effort to overcome, as never before have so many cryptographic algorithms been threatened simultaneously. Automated CBOM tools are optimal solutions for reducing the burden of PQC migration and achieving cryptographic agility. Due to the immaturity of the field, there is a very limited amount of information about CBOMs and no research data is available. However, as more CBOM generators emerge and the use of CBOMs increases, so does the list of possible research topics and the need for such data. Comparing existing CBOM generators will certainly be an important research topic in the future, as will deter-

mining the perceived usefulness of CBOMs and CBOM generators. This research aims to be at the forefront of creating CBOM research data and will hopefully prove useful for future research.

6 Research questions and strategy

This chapter introduces the selected research strategy and its implementation in this study. Additionally, the research questions will be presented.

6.1 Design science

Design science is a practical research paradigm that focuses on creating purposeful artifacts to solve identified problems for the benefit of people and organizations (Hevner et al. 2004). It bridges the gap between theory and practice by providing a set of guidelines for the artifact creation and evaluation processes. According to Iivari January 2007, there are two factors that differentiate IS design science from the practice of building IT artifacts, and they are the rigor of constructing IT artifacts and the scientific evaluation of the artifacts.

Walls, Widmeyer, and El Sawy March 1992 argue that since "design" is a noun and a verb, a product and a process, design theory in information systems (IS) must also consist of something that is produced and the plan for achieving the desired end result. March and Smith December 1995 extended the design science definitions by identifying four design artifacts and two design processes that can be used or created in IS design science research. The produced artifacts can be broadly categorized as: "constructs (vocabulary and symbols), models (abstractions and representations), methods (algorithms and practices), and instantiations (implemented and prototype systems)" (Hevner et al. 2004; March and Smith December 1995). The processes consist of evaluating and building artifacts. The artifact is built for a specific task and evaluated to determine if progress has been made, so the basic questions are: "does it work?" and "how well does it work?" (March and Smith December 1995). Additionally, it is integral to design science research that the artifacts have a formal specification and that their utility has been evaluated, for example, by comparing them to other similar artifacts or by testing them in the intended organizational environment (Hevner et al. 2004).

According to Hevner et al. 2004, IS design science research aims to find solutions to so-called "wicked problems". Rittel and Webber 1973 defined the traits of wicked problems, which Hevner et al. 2004 summarizes as follows:

- "unstable requirements and constraints based upon ill-defined environmental contexts
- complex interactions among subcomponents of the problem and its solution
- inherent flexibility to change design processes as well as design artifacts (i.e., malleable processes and artifacts)
- a critical dependence upon human cognitive abilities (e.g., creativity) to produce effective solutions
- a critical dependence upon human social abilities (e.g., teamwork) to produce effective solutions"

The evaluation process is an integral part of the design science research paradigm. It provides feedback on the artifact design and increases knowledge about the research problem, thus enabling more informed decisions about how to improve the product and the design process (Hevner et al. 2004). Markus, Majchrzak, and Gasser 2002 noticed that this iterative process had to be repeated several times before the final design artifact was created in their study. In the end, it took 70 functional prototypes to arrive at a solution that satisfied the requirements of the artifact. IT artifacts can be evaluated using various different quality attributes, such as functionality, accuracy, performance, reliability, completeness, consistency, or usability (Hevner et al. 2004). Therefore, design science researchers must select appropriate quality metrics for analyzing the effectiveness of the artifacts. Additionally, as Rittel and Webber 1973 defined that "wicked problems have no stopping rule" and "solutions to wicked problems are not true-or-false, but good-or-bad", the end goal of the evaluation process is to produce an artifact that is a satisfactory solution to the identified problem, in accordance with the research resources.

Hevner et al. 2004 summarized the methods typically used in the artifact evaluation process into five main categories: observational, analytical, experimental, testing, and descriptive. The observational evaluation method can be either a case study, where the artifact is studied in a business environment, or a field study, where it is used and monitored in multiple projects. The analytical evaluation method consists of four different subgroups, which are, static analysis, architecture analysis, optimization and dynamic analysis. The experimental methods include simulations and controlled experiments, and the testing can be done as functional (black box) testing or structural (white box) testing. Finally, the descriptive evalu-

ation methods can be either informed arguments or detailed constructed scenarios. For more detailed definitions of the subgroups, see Hevner et al. 2004, p.86.

Hevner et al. 2004 created seven guidelines to guide design science research. First guideline: the artifacts created in design science research must be innovative and purposeful. Second guideline: the artifacts must be created for a specific problem domain. To ensure that the artifact is purposeful within the problem domain, the third guideline emphasizes the importance of a thorough evaluation of the artifact. Because the artifact must be innovative, the fourth guideline specifies that it must either improve the way known problems are solved or provide a solution to an unsolved problem. This distinguishes design science research from routine design, which uses existing knowledge to create artifacts. The fifth guideline is as follows: "The artifact itself must be rigorously defined, formally represented, coherent, and internally consistent". The sixth guideline explains that the creation process and the artifact itself typically include a search process that presents the problem space and the method for finding an effective solution. The final, seventh guideline promotes the importance of communicating the research results effectively. Hevner et al. 2004 advised against the mandatory use of the guidelines and stated that the guidelines must be adjusted to the needs of the research project.

In his essay, Iivari January 2007 discusses the ontology, epistemology, methodology, and ethics of design science. He summarized his ideas to twelve theses, several of which were used by Hevner 2007 for the creation of a three cycle view of design science research. Hevner combined the three cycles into the IS research framework from Hevner et al. 2004, resulting in the following figure 11. In this modified framework, the relevance cycle bridges the application domain to the design science research activities and includes the iterative processes of field testing the artifact and updating its requirements based on the test results. Since the relevance cycle determines the requirements of the artifact, it also defines the final acceptance criteria. The core of design science research is in the design cycle, which is influenced and supported by the relevance and rigor cycles. The design cycle iterates between building the artifact and its design processes and the evaluating of the artifact. The evaluation process continues to modify the artifact design and the design process until all of the requirements set in the relevance cycle are met. Thus, the design cycle evaluation process defines when the final acceptance criteria are met and the artifact building process can be stopped. Finally, the

rigor cycle bridges the design science research activities with the existing knowledge base, providing the grounding theories and methods for research and adding new knowledge to the knowledge base. According to Hevner 2007, good design science research is defined by the synergy and contributions of the relevance and rigor cycles. This is logical, as contributions to the knowledge base (rigor cycle) are the key selling point to the academic audience, while the contributions to the application environment (relevance cycle) are of interest to the practitioner audience. (Hevner 2007).

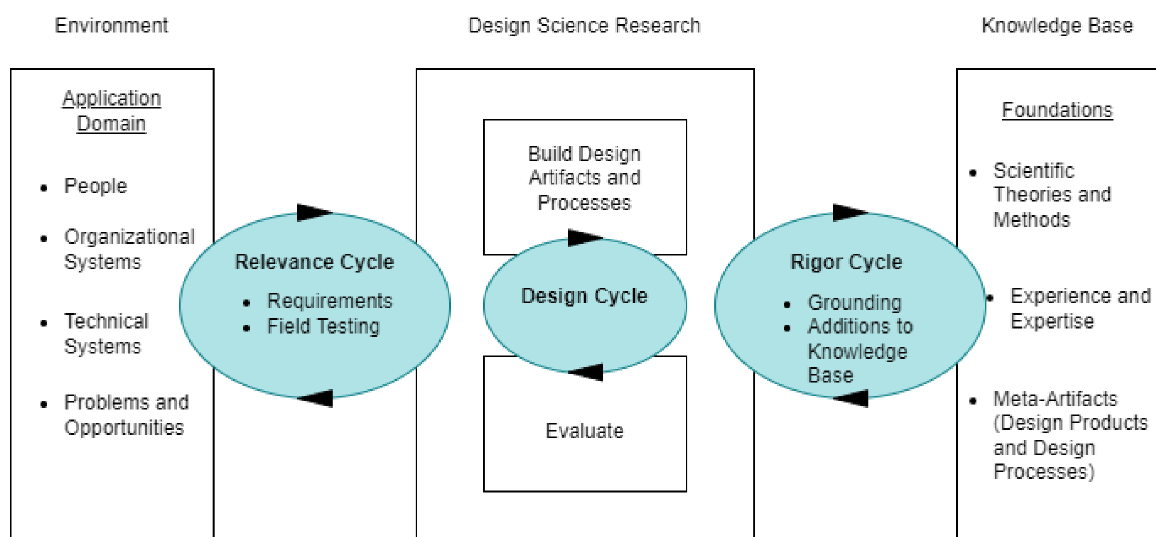

Figure 11. A replica of Hevner's three cycle view framework for design science research (Hevner 2007).

As with any research methodology, design science has some issues that need to be considered. For example, rapid advances in technology can render the results of design science research obsolete before the artifact has even been effectively implemented in an organizational setting or before adequate compensation from the artifact creation has been received. March and Smith December 1995 pointed out that artifacts are perishable, and therefore the results of design science research are perishable as well. Another potential problem with design science research is an excessive focus on the solvable problem and the artifact, while neglecting to build a strong theoretical foundation. This can result in an artifact that doesn't fit the intended organizational setting or meet the needs of the people who will use it. (Hevner et al. 2004).

Ultimately, the goal of all research is to provide new and interesting contributions. For design science research, the contributions can be based on the generality, novelty, and significance of the designed artifact. Typically in design science research, the contribution comes from the designed artifact, but extending the foundations of the design science knowledge base or improving design science methodologies are also important contributions. In conclusion, it is integral to design science research to clearly present and demonstrate the utility of the designed artifact, along with sufficient evidence. (Hevner et al. 2004).

6.2 Implementation in this research

As mentioned in chapter 1, the goal of this research is to create a CBOM tool to facilitate the discovery of cryptographic components in existing technologies, thereby increasing cryptographic agility. Additionally, the CBOM generator will produce NIST quantum security levels for cryptographic components, which will support PQC migration by increasing the knowledge of cryptographic components that are quantum vulnerable.

With these goals in mind, design science was chosen as the research method because it enables building a practically useful tool while providing guidelines for conducting the design process in a scientifically rigorous and justified manner. While there are other practical research methods that have been widely used in IS research, they tend to have slightly different goals and emphases compared to design science. For example, action research, which has been debated as being similar to design science research (Järvinen February 2007), focuses on addressing the problems of a specific client. In contrast, design science research aims to create innovative artifacts that pursue general, unsituated goals, typically targeting potential clients rather than existing ones (Iivari and Venable January 2009). Additionally, Iivari and Venable January 2009 argues that design science has a focus on creating "new reality" and that much of action research is focused on understanding "existing reality", such as organizational problems and human behaviour. These definitions are consistent with the selection of design science as the research method for this study because a completely new IT artifact is being created, the problem it addresses is not limited to a specific client, and it has a general goal of supporting PQC migration and cryptographic agility.

Hevner's three cycle view (see figure 11) provides the main theoretical framework for the artifact creation process in this research. The previously mentioned goals of this research are focused on providing value to the application domain and thus relate to the relevance cycle. Since automated CBOM tools are a very new concept and few such tools exist, there is a lack of research data or information regarding their accuracy, development, and feasibility. Therefore, an additional goal of this research is to provide additions to the knowledge base about CBOM tools by answering the following research questions:

- What are the challenges of creating an automated CBOM tool?
- Can a automated CBOM tool accurately capture cryptographic components from source code?
- Is it feasible to build a CBOM generator by using regular expression searches?

This additional goal is related to the rigor cycle and aims to provide value to the research community. To achieve these goals, the three cycles will be iterated until sufficient results are obtained or time and resource constraints prevent continuing further.

As mentioned in section 6.1, Hevner et al. 2004 categorized the artifact evaluation methods into five main categories, which include additional subgroups. Of these categories, the following will be used in this research:

- Method: Experimental
  - Subgroup: Simulation
  - Description: Execute artifact with artificial data

Figure 12 provides an illustration of the design process for this research. It contains six separate steps that revolve around the three design cycles (relevance, design, and rigor cycles), and the design process begins at the green dot (*1. Grounding*) and ends at the red dot (*6. Addition to Knowledge base (KB)*). Each step is situated above the cycle to it belongs, and the number or letter included within brackets at the end of each step explains how many times the cycle is iterated during the step. The +1 means that the cycle is iterated once, and the +n means that the cycle is iterated multiple times. Thus, the relevance and rigor cycles are iterated twice, and the design cycle is iterated 1+n times (where n >= 1). Going forward,

the design process and its steps are explained in more detail.

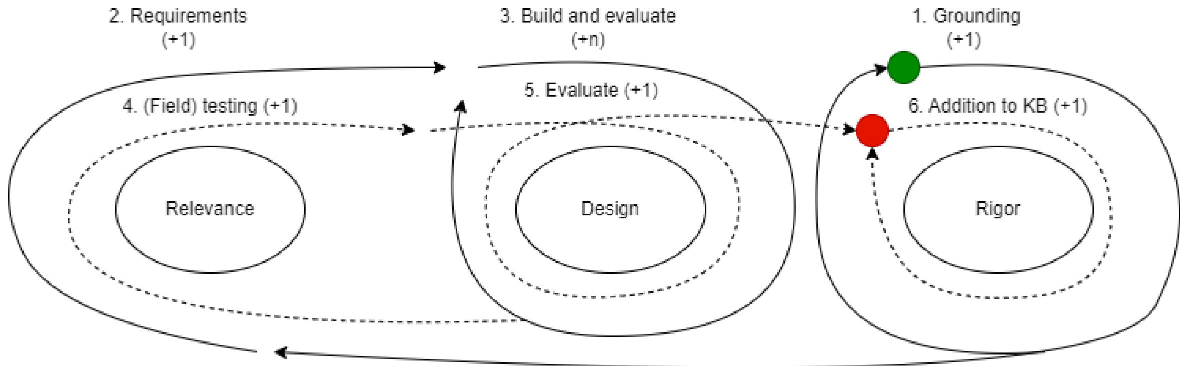

Figure 12. Representation of the design science process for this research. Together with the starting and ending points, as well as the number of iterations planned for each process.

The research begins from the rigor cycle with the introduction of the key topics motivating this research, which are: quantum algorithms, the progress of quantum computers, and the standardization process of PQC algorithms (discussed in chapters 1, 2, 3, and 4). This first iteration of the rigor cycle establishes the motivation for the research and presents the theoretical background (*1. Grounding (+1)* in figure 12). It is followed by an iteration of the relevance cycle (*2. Gather requirements (+1)* in figure 12), where already existing technical systems are researched, and requirements are gathered to guide the artifact design process (discussed in chapters 5, 6, and 7).

After the artifact requirements are gathered, the design and build process begins. The design cycle will be iterated a number of times as the artifact is repeatedly evaluated during the design process to ensure that the built components work as intended (*3. Build and evaluate (+n)* in figure 12 and chapter 8 of thesis). During this phase, the artifact is evaluated with artificial JavaScript data created solely for the purpose of this study. This build and evaluate cycle will be iterated until the artifact succeeds in processing the artificial data in a reasonable manner. Only the final iteration of this phase will be reported because the preceding iterations are done with an incomplete artifact. This final iteration will be discussed in section 9.1 (named "First evaluation" because it is the first evaluation reported in this thesis).

Finally, the design process moves to the second (and final) iteration of the relevance cycle, where the artifact will go through final testing to ensure that the minimum requirements are met (section 9.2 of the thesis). Originally, this phase (*4. (Field) testing (+1)* in figure 12) was intended to be done by scanning open source Node Crypto projects gathered from GitHub, but unfortunately no suitable data was found for this method. Therefore, the "field" testing is conducted with data provided by ChatGPT (OpenAI 2024). The data was created by ChatGPT when it was asked to do the following: "Create a JavaScript file that contains many examples of the following NodeJS method calls:"-followed by the list of method calls in the `NodeCrypto` class presented in chapter 8. After testing the artifact with this data, found deficiencies are fixed and the artifact is evaluated one last time, completing the final iteration of the design cycle (*5. Evaluate (+1)* in figure 12 and section 9.2 of the thesis). The final iteration of the rigor cycle follows as the research results are presented in chapters 10 and 11 of the thesis, thus providing additions to the existing knowledge base (*6. Addition to KB (+1)* in figure 12).

7 Artifact planning

This chapter presents the planning phase of the artifact, along with the reasoning behind the decisions made. The planning began by verifying the need for the artifact and confirming the absence of similar tools, followed by selecting the CBOM scheme, programming language, and defining the target and scope of the tool. It then continued with the selection of the scanning method, supported cryptographic library, supported CBOM data fields, and, finally, the data extraction method. Additionally, a brief description of the minimum artifact requirements is provided.

7.1 Verifying the need

The artifact planning began with researching already existing CBOM and SBOM tools to find out what kind of tools exists and what would still be needed. As a result, it became clear that there is a lack of open-source CBOM tools, and with the increase in laws and regulations related to enhancing cybersecurity and mandating the use of SBOMs (as was discussed in chapter 5), the need for such tools exists now and probably even more so in the future as applications become more complex and difficult to manage. Additionally, the regulations might later be extended to include the usage of CBOMs also.

As mentioned in the introduction chapter 1, NIST said that there is an urgent need for tools that can locate where and how public-key cryptography is used in existing technology infrastructures (Barker, Polk, and Souppaya April 2021), which is something that CBOM generators can accomplish. In addition, Ott, Peikert, and al. September 2019 identified the key challenges of PQC migration, which included the "problem of obsoleting deprecated algorithms", which they emphasized to be "suprisingly hard within a complicated world of deeply entrenched deployments and frameworks with little or no support for phased retirement". Additionally, Ott, Peikert, and al. September 2019 mentioned that the extensiveness of modern cryptographic infrastructure calls for automated tools to make PQC migration more achievable. In conclusion, there is a clear need for automated CBOM generators, and creating such tools can support PQC migration.

7.2 Selecting the scheme

After confirming the need and lack of such tools, the next step was to determine if an existing CBOM scheme could be used or if one should be created during this study. Of the most popular SBOM schemes (CycloneDX, SPDX, and SWID), CycloneDX is the most security-oriented and was used by IBM to create their CBOM scheme. These two reasons alone made CycloneDX a compelling choice. In addition, there are many open-source tools that support CycloneDX, which can be useful for SBOM (and possibly CBOM) presentation, analysis, modification, and vulnerability scanning. IBM's CBOM scheme was excluded from the scheme selection, because it is a part of their consumer product and the artifact created in this study is intended to be an open-source application.

For the reasons above, CycloneDX was initially chosen as the foundation for the CBOM scheme developed in this study. However, during the artifact's development, CycloneDX released their CBOM scheme and related documentation, which was an optimal solution for this study. Therefore, a pivot was made and the CycloneDX CBOM scheme was selected to guide the artifact development. As it happens, the CycloneDX CBOM scheme was the result of collaboration with IBM and the OWASP Foundation, making it very similar to the IBM's CBOM scheme. Nevertheless, they are separate from each other and also operate under different licenses, as IBM's CBOM scheme is under the Apache License Version 2.0 and CycloneDX CBOM scheme is mostly under the Creative Commons Attribution 4.0 International license and only partially under the Apache License Version 2.0.

7.3 Programming language

After researching existing SBOM tools and considering the technical skills available for this research, JavaScript was selected as the most suitable programming language for the artifact creation. One of the reasons for choosing JavaScript was that an already existing BOM tool, cdxgen ("CycloneDX Generator" May 2024), proved to be compherensive, impressive, and it was created with JavaScript. Cdxgen was proof that JavaScript could work well for the intended artifact creation, and it was also a skill set that was available for this research. Additionally, CycloneDX supports JSON and XML formats, and JSON in particular is highly

compatible with JavaScript, which further supported the selection of JavaScript as the programming language for this project.

7.4 Target and scope

Much time was spent considering what would be the intended scan target of the artifact: would it be an operating system, a file directory, or a Docker container? At the time of the artifact planning, there did not exist any open source CBOM generators. Thus, there was no need to consider overlap with other similar tools, and there were no such restrictions on the target and scope of the artifact.

The scope was decided to be file directories and the scan targets would be specific source files related to cryptographic component management. Ideally, the scope would have been wider, allowing for the scanning of Docker containers and GitHub repositories, but time and resource limitations required for restricting the scope. The popularity of JavaScript (Overflow July 2024) guided the decision to prioritize the scanning of JavaScript source files, and to extend the functionality to other programming languages later if possible. To conclude, the artifact would be used to recursively scan file directories for JavaScript source files, find cryptographic components from them, and create a single CBOM file containing the results.

7.5 Scanning method

The next decision to be made was how the tool should function to achieve its goals of cryptographic component scanning. Using artificial intelligence for the cryptographic component analysis and CBOM creation process would be a great solution, but due to the lack of available time and resources, this was not a viable option for this research. Therefore, two different methods were considered, which were: using regular expressions to find matching patterns and to extract relevant information, or creating a database containing cryptographic algorithms and their properties and using this data to find cryptographic components from file systems. The former method could be more accurate and provide more detailed cryptographic components, but it could also be tedious to create, and its scope would have to be

quite limited to fit within the allocated resources of a master's thesis. On the other hand, the latter method would be less accurate, but its scope would be much larger, meaning that it could be able to scan many different types of files and systems much earlier than the regular expression-method.

Eventually, the decision came down to what was more important for this research: capturing cryptographic data accurately, or being able to scan large systems, but possibly with more false positives and with less accurate data. Since the whole idea of CBOMs is to improve cryptography management and increase cybersecurity resilience, having a large number of false positives or inadequate data seemed counterproductive, which is why the regular expression-method was selected as the scanning method.

7.6 Cryptographic libraries

The next step was to be determine what kind of data should be extracted from the JavaScript source files in order to build sufficiently comprehensive cryptographic CBOM components. This required researching the relevant cryptographic libraries and their methods for creating cryptographic components. For JavaScript, the most commonly used cryptographic libraries are the Node.js Crypto module and the Web Crypto API. The Node.js Crypto module is primarily used for backend (server-side) applications, while the Web Crypto API is typically used in client-side web applications, such as browsers. Since it was necessary to start with one library, Node.js Crypto was selected due to its popularity and broader range of use cases. If time permitted, the Web Crypto API would be included later.

7.7 Data fields

At this stage, it was necessary to determine which data fields from the CycloneDX BOM specification would be included in the artifacts cryptographic component data extraction. Three criteria guided the selection process: the fields relevance to the research topic, the resources required to implement a working solution capable of extracting the field values, and adherence to the CycloneDX BOM specification by including the 'required' fields (CycloneDX 2024). The selected fields are presented in the BOM specification shown in figure

13, along with their parent fields, requirement information, type, and description. Figure 14 presents an example BOM file where the hierarchy of each field can be seen more clearly.

Field name	Parent	Required	Type	Description
bomFormat		x	enum (of string)	Specifies the BOM format. Must be "CycloneDX".
specVersion		x	string	The CycloneDX specification version used.
metadata			object	Provides additional information about the BOM
timestamp	metadata		string	The date and time when the BOM was created.
component	metadata		object	The component that the BOM describes.
type	component	x	enum (of string)	Specifies the type of the component. Receives the default value "application".
name	component	x	string	Name of the component. Will be "PQCBOM tool" until further developed.
version	component		string	The component version.
components			array	A list of software components
name	components	x	string	Name of the cryptographic component.
type	components	x	enum (of string)	The type of the component. Because this is a CBOM tool, this will receive the value of "cryptographic-asset".
properties	components		array	Contains additional information about an object.
name	properties	x	string	Name of the property. Will be "SrcFile", because the tool scans sourcefiles.
value	properties		string	Specifies the location of the sourcefile.
cryptoProperties	components		object	Contains information about the properties of cryptographic assets.
assetType	cryptoProperties	x	enum (of string)	Can be "algorithm", "certificate", "protocol" or "related-crypto-material".
algorithmProperties	cryptoProperties		object	Properties of cryptographic algorithms.
parameterSetIdentifier	algorithmProperties		string	An identifier for the parameter set of the cryptographic algorithm.
mode	algorithmProperties		enum (of string)	The mode of operation in which the cryptographic algorithm (block cipher) is used. Must be one of: "cbc", "ecb", "ccm", "gcm", "cfb", "ofb, "ctr", "other" or "unkown".
classicalSecurityLevel	algorithmProperties		integer	The classical security level that a cryptographic algorithm provides (in bits).
nistQuantumSecurityLevel	algorithmProperties		integer	The NIST security strength category as defined in https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/evaluation-criteria/security-(evaluation-criteria). A value of 0 indicates that none of the categories are met. Value must be 0, 1, 2, 3, 4, 5 or 6.

Figure 13. BOM specification for the artifact. Derived from CycloneDX 2024.

Most of the fields do not require additional explanation beyond what is written in the specification in figure 13, but a few selected ones are presented in more detail. The `type` field, under the parent field *component*, is always assigned the value *application*. Similarly, the `name` field, also under *component*, consistently receives the value *PQCBOM tool*. Ideally, these values should change depending on the scanned component, but doing so would have required significant additional resources for minimal added value. As a result, these features were deferred for future development. The planned `assetType` values are *algorithm*, *certificate*, *protocol*, and *related-crypto-material*, as shown in the specification in figure 13. However, the primary focus of this research will be on capturing cryptographic algorithms, and capturing the other asset types will be included if possible within the time and resource constraints.

```
{
  "bomFormat": "CycloneDX",
  "specVersion": "1.6",
  "metadata": {
    "timestamp": "2024-05-20T09:13:51.743Z",
    "component": {
      "type": "application",
      "name": "PQCBOM tool",
      "version": "0.0.1"
    }
  },
  "components": [
    [
      {
        "name": "aes-128-cbc-hmac-sha256",
        "type": "cryptographic-asset",
        "properties": {
          "name": "SrcFile",
          "value": "/home/kurisu/PQC/cyclonedx_pqc/testing/testdirs/javascript/jsTestFile.js"
        },
        "cryptoProperties": {
          "assetType": "algorithm",
          "algorithmProperties": {
            "parameterSetIdentifier": "128",
            "mode": "cbc",
            "classicalSecurityLevel": 128,
            "nistQuantumSecurityLevel": 1
          }
        }
      },
```


Figure 14. Example BOM file that displays the hierarchy of the components.

7.8 Data extraction

The next key step involved studying the Node.js Crypto module to understand how cryptographic components are created and how data can be extracted. After reviewing the library documentation (Node.js Developers 2024) and examining use cases found online, it became apparent that the most important method calls to search for typically included the words *create* (such as *createCipher* or *createPrivateKey*) or *generate* (such as *generateKeyPair*). There were a few exceptions, such as *hash* or *getDiffieHellman*. The method call parameters typically include the specifications of the cryptographic components, which need to be extracted for the cryptographic component creation process. Therefore, the data extraction plan involves searching for JavaScript source files that implement the Node.js Crypto module, and, if found, identifying specific method calls used to create cryptographic components and extracting data from their parameter values.

7.9 Minimum requirements

The following list outlines the minimum requirements that the artifact aims to meet. The requirements are limited to the artifact's scope of functionality.

- Accurately capture cryptographic components.
- Follows the CycloneDX standard v1.6.
- Captures the "required"-fields defined by CycloneDX standard.
- Capture and present the following cryptographic component BOM fields:
  - Name of the component (`name`)
  - Type of the component (`type`)
  - The directory location of the scanned file (`value`)
  - The asset type of the component. At least capture cryptographic algorithms (`assetType`)
  - The identifier for the parameter set of the cryptographic algorithm (`parameterSetIdentifier`)
  - Bit representation of the classical security level of the cryptographic algorithm (`classicalSecurityLevel`)
  - The NIST-categorized quantum security level for the cryptographic algorithm (`nistQuantumSecurityLevel`)

8 Artifact

This chapter provides an overview of how the artifact was constructed and how it functions. To see the code behind the functionalities described in this chapter, refer to Appendix A.

8.1 Setup and interface

The setup process requires downloading the source directory from the link in Appendix A and installing Node.js. The artifact is a JavaScript (JS) file named `pqcbom.js`, which features a simple command-line interface that utilizes the Yargs Node.js library (Yargs Developers 2024). To run the script, use the following command:

```
node /location/of/file/pqcbom.js
```

Ensure that the path is updated to match the actual location of the script. Alternatively, if the script is added as an executable to the `$PATH`, it can be executed by typing the following command (with the `[argument] updated to some argument presented in the next paragraph`) in the command line of your operating system:

```
pqcbom -[argument]
```

From this point forward, the explanation of the artifact usage is done with the assumption that it has been made executable. Figure 15 presents the different arguments that can be given to the command-line interface. This manual page is displayed when the artifact is executed with the `-h` or `-help` argument. The `-i` and `-input` arguments must be followed by the directory path, which will then be recursively scanned for JavaScript files. The `-o` and `-output` arguments can be specified after the input arguments and directory path to define the name of the output BOM file, as shown below:

```
pqcbom -i /home/files/application1 -o application1
```

This will result in a file named `application1.json`. If no output file name is given, then the output file will be `bom.json` by default. If no input directory path is given, the

tool will scan the current directory from which the pqcbom command is executed.

```
root@DESKTOP-F67KJIL:/home/kurisu/PQC/cyclonedx_pqc# pqcbom -h
Options:
  -i, --input     Input directory path
  -o, --output    Output file name
  -h, --help      Show help                                        [boolean]
  -v, --version   Show version number                              [boolean]
root@DESKTOP-F67KJIL:/home/kurisu/PQC/cyclonedx_pqc#
```

Figure 15. The command line interface of the created artifact.

8.2 Example data flow

After receiving the command to scan a file directory, the pqcbom.js starts by calling the createBomFile function. This function creates a JavaScript bom object containing the header and metadata information for the BOM fileand initializes the components array, to which all discovered cryptographic components will be added. At this stage, the scanDirectory function is called (as shown in figure 16), with the directory path passed as a parameter.

```
// Create JS bom-object.
const bomObj = {
    bomFormat: "CycloneDX",
    specVersion: "1.6",
    metadata: {
        timestamp: getTime(),
        component: {
            type: "application",
            name: "PQCBOM tool",
            version: pqcbomVersion
        }
    },

    // scanDirectory returns an array that contains javascript objects (components)
    components: new Array(scanDirectory(dirPath))
}
```

Figure 16. Code snippet of pqcbom.js. A JavaScript BOM object is created and scanDirectory function is called.

The scanDirectory function recursively searches all directories within the given directory path for JavaScript files. When a JavaScript file is found, the getComponents

function is called to check whether the file uses the Node.js Crypto module. If the module is not used, the function resumes scanning the directories. If the module is found, it calls the function `findNodeCryptoComponents`, which uses regular expressions to search for cryptographic components within the file.

The `findNodeCryptoComponents` function receives a string array as a parameter, containing the names of relevant Node.js Crypto method calls that are used to search for cryptographic components. The function returns a set of the discovered cryptographic components.

A class called `NodeCrypto` is created to store and categorize method calls into their respective cryptographic component categories. Additionally, the `NodeCrypto` class is used to store the bit lengths for MODP 1,2,5, and 14-18 algorithms, as this information is not included in the method names.

Figure 17 presents the calls for the `findNodeCryptoComponents` function and the `nodeCryptoObj` object, which is an instance of the `NodeCrypto` class. The Node.js Crypto method calls related to cryptographic algorithms are stored in the `algorithm` property of the `nodeCryptoObj`, following the same pattern for `relatedCryptoMaterials`, as shown in figure 18.

```
// get all the node crypto library's method calls that are relevant for creating cryptographic components.
// currently no duplicates are added.
setOfAlgRegexpMatches = findNodeCryptoComponents(nodeCryptoObj.algorithm, fileContent, 'algorithm');
setOfCryptMatRegexpMatches = findNodeCryptoComponents(nodeCryptoObj.relatedCryptoMaterial, fileContent, 'relatedCryptoMaterial');
```

Figure 17. Function call for `findNodeCryptoComponents`.

```
export class NodeCrypto {
    constructor() {

        this.algorithm = [                  //methods
            'createCipher',                 //crypto.createCipher('algorithm', ..)     these can be used without `crypto.
            'createCipheriv',               //crypto.createCipheriv('algorithm', ..)
            'createDecipheriv',             //createDecipheriv('algorithm', ...)
            'createDecipher',               //createDecipher('algorithm', ...)
            'createHash',                   //crypto.createHash(algorithm[, options])
            'createHmac',                   //crypto.createHmac(algorithm, key[, options])
            'createSign',                   //crypto.createSign(algorithm[, options])
            'createVerify',                 //crypto.createVerify(algorithm[, options])
            'hash',                         //crypto.hash(algorithm, data[, outputEncoding])
        ]

        this.relatedCryptoMaterial = [      // keys
            'createPrivateKey',             //crypto.createPrivateKey(key)
            'createPublicKey',              //crypto.createPublicKey(key)
            'createSecretKey',              //crypto.createSecretKey(key[, encoding])
            'createSecretKeyBuffer',        //crypto.createSecretKeyBuffer('sha224WithRSAEncryption');
            'createDiffieHellman',          // createDiffieHellman(2048)
            'createDiffieHellmanGroup',     //crypto.createDiffieHellmanGroup(name)
            'getDiffieHellman',             // crypto.getDiffieHellman(groupName)
            'createECDH',                   //crypto.createECDH(curveName)
            'generateKey',                  //crypto.generateKey(type, options, callback)
            'generateKeyPair',              //crypto.generateKeyPair(type, options, callback)
            'generateKeyPairSync',          //crypto.generateKeyPairSync(type, options)
            'generateKeySync',              //crypto.generateKeySync(type, options)
        ]
```

Figure 18. The `algorithm` and `relatedCryptoMaterial` properties of the `NodeCrypto` class are used to store and categorize Node.js Crypto method calls.

Within the `findNodeCryptoComponents` function, different regular expressions have been formed for searching related crypto materials (see figure 19) and algorithms (see figure 20). These regular expressions include the method calls stored in the `NodeCrypto` class and are used to search for matching patterns within the scanned JavaScript file. If a match is found, it is added to an array of found components, which is later converted to a JavaScript `Set` object to eliminate duplicates, and then returned to the `getComponents` function. If no components are found, an empty `Set` is returned.

```
case 'relatedCryptoMaterial':
    searchElementsArray.forEach(element => {
        const relCryptMatRegexp = new RegExp(`(^(crypto|diffieHellman|ecdh)\\.)|\\b${element}\\(((\\d+)|\\{|(['"](\\w+)(-(\\w*))*["'])\\s*[^;]*\\s*((\\}\\))|(\\);))`, 'g');
```

Figure 19. The regular expression for searching related-crypto-material method calls related to the Node.js Crypto module.

```
case 'algorithm':
    searchElementsArray.forEach(element => {
        const algorithmRegexp = new RegExp(`(^(crypto|diffieHellman|ecdh)\\.)|\\b${element}\\((((['"](\\w+)(-(\\w*))*['"]))(,|\\))`, 'g');
```

Figure 20. The regular expression for searching cryptographic algorithm method calls related to the Node.js Crypto module.

After the `findNodeCryptoComponents` function returns the `Set` containing the raw data of cryptographic component method calls to the `getComponents` function, each regular expression match is individually sent to the `addComponent` function for data extraction and cryptographic component creation (see figure 21). First, the `addComponent` function calls the `extractFirstParameter` function (see figure 22), which uses regular expressions to retrieve the name value of the cryptographic component from the regular expression match string. The function is named `extractFirstParameter` because the first parameter of the method call typically contains information about the used cryptographic algorithm (in the Node.js Crypto module).

```
// go through all found method calls, create a crypto component from the method calls first parameter values and
// add component to components-list.
if(setOfAlgRegexpMatches.size > 0){
    setOfAlgRegexpMatches.forEach(regexpMatch => {
        components.push(addComponent(filePath, fileExtension, 'algorithm', regexpMatch));
    });

}
if(setOfCryptMatRegexpMatches.size > 0){
    setOfCryptMatRegexpMatches.forEach(regexpMatch => {
        components.push(addComponent(filePath, fileExtension, 'related-crypto-material', regexpMatch));
    });

}
```

Figure 21. Function call for `addComponent`, which extracts relevant data from found regular expression matches and uses it to create cryptographic components.

```
/**
 * Creates and returns a cycloneDX cryptographic component.
 * @param {Path to file} filePath
 * @param {Specifies the type of cryptographic asset} cryptoAssetType
 * @returns a cycloneDX cryptographic component
 */
function addComponent(filePath, fileExtension, cryptoAssetType, regexpMatchString){

    let nodeCryptoObject = new NodeCrypto();
    let NistQTSecLevelClassInstance = new NistQuantumSecLevel();
    const digitRegexp = new RegExp(/\d{3,}/, "g");

    let firstParam = extractFirstParameter(regexpMatchString); // retreives the parameter name from the method call (regexMatchString)
```

Figure 22. Call for `extractFirstParameter` function.

After `extractFirstParameter` function returns the name of the cryptographic component, the `addComponent` function extracts the remaining cryptographic data from the regular expression match string. Different sets of rules are used for data extraction depending on whether the crypto asset type is `algorithm` or `related-crypto-material`. However, due to time and resource constraints, the creation of related-crypto-material components was left for future versions of the artifact. As a temporary solution, related-crypto-material components are handled as if they were cryptographic algorithm components (see figure 23).

```
if(cryptoAssetType === 'related-crypto-material'){
    //NOTE: currently this extracts only algorithm data as it is most relevant to thesis.
    // This will need to be edited back to create related crypto material components.
    cryptoAssetType = 'algorithm'; //Changing this to algorithm as a temporary fix. Modify later.
```

Figure 23. As a temporary solution, only the cryptographic algorithm data is extracted from related-crypto-material components, and thus they are treated similarly to cryptographic algorithm components.

Figure 24 shows the part of the code where cryptographic components are assigned their values and created. These fields, which are detailed in the minimum requirements section 7.9 of the thesis, are also presented with example values in figure 14. While the values for `parameterSetIdentifier`, `mode`, and `classicalSecurityLevel` are extracted from the found method calls, the `nistQuantumSecurityLevel` is generated by the artifact. To assign quantum security levels to the identified cryptographic algorithms, a class called `NistQuantumSecLevel` was created. It is used for storing and categorizing different cryptographic algorithms to their specified quantum security levels, which range from level 0 to level 6. An example of this categorization is shown in figure 25.

When all the values of a cryptographic algorithm component have been extracted or created, the `addComponent` function returns the component to the `getComponents` function, which adds it to a `Set` of components. This set is then returned to the `scanDirectory` function, which in turn passes it to the `createBom` function. The `createBom` function now has a JavaScript object containing all of the search results, which it converts into JSON format, and generates a JSON BOM file that contains this data. The final output is a CycloneDX CBOM file listing all cryptographic algorithms found within the JavaScript files that used the Node.js Crypto module in the specified directory path.

```
const component = {
    name: firstParam,
    type: 'cryptographic-asset',
    bomref: undefined,
    properties: {
        name: 'SrcFile',
        value: filePath
    },
    cryptoProperties: {
        assetType: cryptoAssetType,
    }
}

switch (component.cryptoProperties.assetType){
    case 'algorithm':
        component.cryptoProperties.algorithmProperties = {
            primitive: undefined,
            parameterSetIdentifier: paramSetID,
            mode: algorithmMode,
            executionEnvironment: undefined,
            implemenetationPlatform: undefined,
            certificationLevel: undefined,
            cryptoFunctions: undefined,
            classicalSecurityLevel: classicalSecLvl,
            nistQuantumSecurityLevel: nistQTsecLvl
        }
        break;
```

Figure 24. The creation of the cryptographic components in the `addComponent` function.

```
export class NistQuantumSecLevel{
    constructor(){
        this.levelZero = {
            level : 0,
            algorithms : [ // very weak
            // < AES 128, DES, SHA-224, RSA/DH-2048, ECC 224,
            'DES',
            'SHA224',
            'SHA-224',
            'SHA3-224',
            'RSA-2048',
            'DH-2048',
            'ECC-224'
            ]
        }
        this.levelOne = {
            level : 1,
            algorithms : [ // weak
                'AES128',
                'AES-128',
                'KYBER-512',
                'RSA-3072',
                'DH-3072',
                'ECC-256'
            ]
        }
```

Figure 25. A code snippet of the `NistQuantumSecLevel` class used for retrieving the NIST quantum security levels of different cryptographic algorithms.

9 Evaluations

In this chapter, the artifact is evaluated twice, and a summary of both evaluations is provided. The first evaluation, discussed in section 9.1, corresponds to step three of the used design process (see figure 12), where the evaluation is conducted multiple times during the artifact's construction phase. The second evaluation corresponds to steps four and five of the design process, where the artifact is tested and evaluated for the final time. As mentioned in section 6.2, the evaluation method used in this research falls under the *experimental* category and *simulation* subgroup of the design evaluation methods outlined by Hevner et al. 2004, where the artifact is executed using artificial data.

The most important evaluation criteria for this artifact is accuracy, though other quality attributes may also be briefly analyzed. The CBOM fields `parameterSetIdentifier`, `classicalSecurityLevel`, and `nistQuantumSecurityLevel` are the most critical and error-prone, which is why their success rate will be evaluated in the summaries of both evaluations.

9.1 First evaluation

The first evaluation of the artifact is conducted using artificial data created specifically to test the artifact's functionality and address any identified errors. The used test files are listed in Appedix A, under the file names `jsTestFile.js`, `jsInvavlidTestValues.js`, and `relatedCryptoMatTestFile.js`. During this evaluation, the artifact will scan each of the three test files separately, and the resulting BOM files will be analyzed.

9.1.1 First test file

Figure 26 shows the artifact's output when a CBOM file is generated by scanning `jsTestFile.js`. The console displays two different `Set` objects: the first contains 32 entries representing method calls and parameters of found cryptographic algorithms, while the second contains 6 entries for related-crypto-materials.

```
root@DESKTOP-F67KJIL:/home/kurisu/PQC/cyclonedx_pqc/testing/testdirs/javascript# ls
bom.json  getCurvesList  getHashesList  jsInvalidTestValues.js  jsTestFile.js  jsTestFile2.js  nodeCryptoTest.js  regexptest.js  relatedCryptoMatTestFile.js
root@DESKTOP-F67KJIL:/home/kurisu/PQC/cyclonedx_pqc/testing/testdirs/javascript# pqcbom -h
Options:
  -i, --input    Input directory path
  -o, --output   Output file name
  -h, --help     Show help                                          [boolean]
  -v, --version  Show version number                                [boolean]
root@DESKTOP-F67KJIL:/home/kurisu/PQC/cyclonedx_pqc/testing/testdirs/javascript# pqcbom -i jsTestFile.js -o jsTestFileBOM
/home/kurisu/PQC/cyclonedx_pqc/testing/testdirs/javascript/jsTestFile.js
Supported file type: .js
Found import: import * as crypto from 'crypto'
Found import: import crypto from 'crypto'
Found import: import { createHash } from 'crypto'
Found import: import crypto from 'crypto'
Found import: import * as crypto from 'crypto'
Found import: import { createHash } from 'crypto'
Found import: import { createHash as ch } from 'crypto'
Found import: import crypto, { createHash } from 'crypto'
Found import: import { DiffieHellman, ECDH, createHmac, createPrivateKey, createPublicKey, diffieHellman } from 'node:crypto'
Node Crypto library found!
findNodeCryptoComponents functions tmpSet:  Set(32) {
  "createCipher('algorithm')",
  "createCipher('aes-128-cbc-hmac-sha256')",
  "createCipher('camellia-128-cbc')",
  "createCipher('des3')",
  "createCipher('id-aes256-GCM')",
  "createCipher('RSA-SHA224')",
  "createCipheriv('algorithm')",
  "createCipheriv('aes128')",
  "createCipheriv('blowfish')",
  "createCipheriv('id-aes192-wrap')",
  "createCipheriv('rc2-64-cbc')",
  "createCipheriv('RSA-RIPEMD160')",
  "createDecipheriv('algorithm')",
  "createDecipheriv('aes-256-cfb1')",
  "createDecipheriv('des-cbc')",
  "createDecipheriv('id-smime-alg-CMS3DESwrap')",
  "createDecipheriv('seed-cbc')",
  "createDecipheriv('RSA-SHA1-2')",
  "createDecipher('algorithm')",
  "createDecipher('bf-ecb')",
  "createDecipher('des-ede3-cfb1')",
  "createDecipher('rc4-hmac-md5')",
  "createDecipher('camellia256')",
  "createDecipher('RSA-SHA3-256')",
  "createHash('algorithm')",
  "createHmac('algorithm')",
  "createHmac('RSA-SHA384')",
  "createSign('algorithm')",
  "createSign('RSA-SHA3-384')",
  "createVerify('algorithm')",
  "createVerify('RSA-SHA3-512')",
  "hash('algorithm')"
}
findNodeCryptoComponents functions tmpSet:  Set(6) {
  "createPrivateKey('sha384WithRSAEncryption');",
  "createPublicKey('sha3-224');",
  "createSecretKeyBuffer('sha224WithRSAEncryption');",
  "createDiffieHellman('blake2s256');",
  'createDiffieHellman(1024);',
  "createECDH('id-rsassa-pkcs1-v1_5-with-sha3-256');"
}

file jsTestFileBOM.json created!

root@DESKTOP-F67KJIL:/home/kurisu/PQC/cyclonedx_pqc/testing/testdirs/javascript# ls
bom.json  getCurvesList  getHashesList  jsInvalidTestValues.js  jsTestFile.js  jsTestFile2.js  jsTestFileBOM.json  nodeCryptoTest.js  regexptest.js  relatedCryptoMatTestFile.js
root@DESKTOP-F67KJIL:/home/kurisu/PQC/cyclonedx_pqc/testing/testdirs/javascript#
```

Figure 26. Demonstration of the pqcbom-tool: A CBOM file named *jsTestFileBOM* was created by scanning `jsTestFile.js`. Only 38 components were generated instead of the expected 39.

Overall, the `jsTestFile.js` was expected to produce 39 cryptographic components, but only 38 components were created in the CBOM file. The missing component was associated with the following method call:

```
crypto.createHash('RSA-SHA512/224');
```

The problem occurred because the parameter included the character `/`, which was not accounted for in the regular expressions used. After incorporating the missing character into the regular expressions in functions `findNodeCryptoComponents` and `extractFirstParameter`, all 39 components were successfully created. The final CBOM file produced during this evaluation phase is available in Appendix A under the file name `jsTestFileBOM.json`. The output and all 39 found components can be seen in figure 27.

```
kurisu@DESKTOP-F67KJIL:~/PQC/cyclonedx_pqc/testing/testdirs/javascript$ node /home/kurisu/PQC/cyclonedx_pqc/tool/pqcbom.js -i jsTestFile.js -o jsTestFileBOM
Debugger listening on ws://127.0.0.1:53549/eca02271-152f-4210-aad3-e358766bf311
For help, see: https://nodejs.org/en/docs/inspector
Debugger attached.
/home/kurisu/PQC/cyclonedx_pqc/testing/testdirs/javascript/jsTestFile.js
Supported file type: .js
Found import: import * as crypto from 'crypto'
Found import: import crypto from 'crypto'
Found import: import { createHash } from 'crypto'
Found import: import crypto from 'crypto'
Found import: import * as crypto from 'crypto'
Found import: import { createHash } from 'crypto'
Found import: import { createHash as ch } from 'crypto'
Found import: import crypto, { createHash } from 'crypto'
Found import: import { DiffieHellman, ECDH, createHmac, createPrivateKey, createPublicKey, diffieHellman } from 'node:crypto'
Node Crypto library found!
findNodeCryptoComponents functions tmpSet:  Set(33) {
  "createCipher('algorithm')",
  "createCipher('aes-128-cbc-hmac-sha256')",
  "createCipher('camellia-128-cbc')",
  "createCipher('des3')",
  "createCipher('id-aes256-GCM')",
  "createCipher('RSA-SHA224')",
  "createCipheriv('algorithm')",
  "createCipheriv('aes128')",
  "createCipheriv('blowfish')",
  "createCipheriv('id-aes192-wrap')",
  "createCipheriv('rc2-64-cbc')",
  "createCipheriv('RSA-RIPEMD160')",
  "createDecipheriv('algorithm')",
  "createDecipheriv('aes-256-cfb1')",
  "createDecipheriv('des-cbc')",
  "createDecipheriv('id-smime-alg-CMS3DESwrap')",
  "createDecipheriv('seed-cbc')",
  "createDecipheriv('RSA-SHA1-2')",
  "createDecipher('algorithm')",
  "createDecipher('bf-ecb')",
  "createDecipher('des-ede3-cfb1')",
  "createDecipher('rc4-hmac-md5')",
  "createDecipher('camellia256')",
  "createDecipher('RSA-SHA3-256')",
  "createHash('algorithm')",
  "createHash('RSA-SHA512/224')",
  "createHmac('algorithm')",
  "createHmac('RSA-SHA384')",
  "createSign('algorithm')",
  "createSign('RSA-SHA3-384')",
  "createVerify('algorithm')",
  "createVerify('RSA-SHA3-512')",
  "hash('algorithm')"
}
findNodeCryptoComponents functions tmpSet:  Set(6) {
  "createPrivateKey('sha384WithRSAEncryption');",
  "createPublicKey('sha3-224');",
  "createSecretKeyBuffer('sha224WithRSAEncryption');",
  "createDiffieHellman('blake2s256');",
  'createDiffieHellman(1024);',
  "createECDH('id-rsassa-pkcs1-v1_5-with-sha3-256');"
}

file jsTestFileBOM.json created!
```

Figure 27. The updated `jsTestFileBOM.json` after correcting the coding error. All 39 components are now correctly identified from `jsTestFile.js`.

9.1.2 Second test file

The file `jsInvalidTestValues.js` is used primarily to verify that the tool does not generate components from invalid values, that is, those that do not match the intended regular expression search criteria. As shown in figure 28, the first scan produced an empty CBOM file, which is the expected outcome.

No components were created during this first scan because `jsInvalidTestValues.js` did not reference the Node.js Crypto module, which is one of the first criteria for the tool to continue with searching for cryptographic components. In the second run, after adding a reference to the Node.js Crypto module in `jsInvalidTestValues.js`, the tool created nine cryptographic components, which all contained a parameter called `invalidValue`. This was also an expected outcome because the artifact does not yet validate that method call parameters contain actual algorithms, as this would be difficult to achieve. As a result, test values like `algorithm` and `invalidValue` result in the creation of cryptographic components.

```
root@DESKTOP-F67KJIL:/home/kurisu/PQC/cyclonedx_pqc/testing/testdirs/javascript# pqcbom -i jsInvalidTestValues.js -o jsInvalidTestValuesBOM
/home/kurisu/PQC/cyclonedx_pqc/testing/testdirs/javascript/jsInvalidTestValues.js
Supported file type: .js

file jsInvalidTestValuesBOM.json created!

root@DESKTOP-F67KJIL:/home/kurisu/PQC/cyclonedx_pqc/testing/testdirs/javascript# pqcbom -i jsInvalidTestValues.js -o jsInvalidTestValuesBOM2
/home/kurisu/PQC/cyclonedx_pqc/testing/testdirs/javascript/jsInvalidTestValues.js
Supported file type: .js
Found import: import * as crypto from 'crypto'
Found import: import * as crypto from 'crypto'
Node Crypto library found!
findNodeCryptoComponents functions tmpSet:  Set(9) {
  "createCipher('invalidValue')",
  "createCipheriv('invalidValue')",
  "createDecipheriv('invalidValue')",
  "createDecipher('invalidValue')",
  "createHash('invalidValue')",
  "createHmac('invalidValue')",
  "createSign('invalidValue')",
  "createVerify('invalidValue')",
  "hash('invalidValue')"
}

file jsInvalidTestValuesBOM2.json created!
```

Figure 28. With the addition of the Node.js Crypto module reference to the file, the second scan of `jsInvalidTestValues.js` identified a set of 9 cryptographic components. Thus, the artifact functioned as intended.

9.1.3 Third test file

The previous two test files were primarily used to test the extraction of cryptographic data from method calls with differently named parameters. However, `relatedCryptoMatTestFile.js` contains more complex use cases, some of which are based on examples from the Node.js Crypto documentation (Node.js Developers 2024).

As the name suggests, `relatedCryptoMatTestFile.js` was intended to test the identification and creation of related-crypto-material components. However, during this stage of the build-and-evaluate cycle, it became evident that enabling the tool to generate related-crypto-material components (or certificate components) would be outside the resources allocated for this research. These cryptographic components contain so much important data that creating regular expression rules for their extraction would have required more time than was available. Therefore, as previously mentioned, at this stage the tool only extracts cryptographic algorithms from related-crypto-material components and uses that data to create cryptographic algorithm components.

Figure 29 presents the tool's output when `relatedCryptoMatTestFile.js` was scanned. The output CBOM file can be found from Appendix A under the name `relatedCryptoMatTestFileBOM.json`. A `Set` of 14 cryptographic components was found, while 15 was expected. Upon inspection, the missing method call was identified to be:

```
generateKey('hmac'...
```

The absence was due to deficiencies in the regular expression that extracts related-crypto-material method calls. After fixing the regular expression, all 15 components were found, as seen in figure 30.

```
root@DESKTOP-F67KJIL:/home/kurisu/PQC/cyclonedx_pqc/testing/testdirs/javascript# pqcbom -i relatedCryptoMatTestFile.js -o relatedCryptoMatTestFileBOM
/home/kurisu/PQC/cyclonedx_pqc/testing/testdirs/javascript/relatedCryptoMatTestFile.js
Supported file type: .js
Found require('node'): require('node:crypto')
Node Crypto library found!
findNodeCryptoComponents functions tmpSet:  Set(14) {
  "getDiffieHellman('modp14');",
  "getDiffieHellman('modp16');",
  "hash: 'SHA-256',",
  'generateKey({\n' +
    "    name: 'HMAC',\n" +
    "    hash: 'SHA-256',\n" +
    '    length: 256,\n' +
    "  }, true, ['sign', 'verify']);",
  "cipher: 'id-aes256-GCM',",
  "generateKeyPair('rsa-pss', {\n" +
    '    modulusLength: 4096,\n' +
    '    publicKeyEncoding: {\n' +
    "      type: 'spki',\n" +
    "      format: 'pem',\n" +
    '    },\n' +
    '    privateKeyEncoding: {\n' +
    "      type: 'pkcs8',\n" +
    "      format: 'pem',\n" +
    "      cipher: 'id-aes256-GCM',\n" +
    "      passphrase: 'top secret',\n" +
    '    },\n' +
    '  }, (err, publicKey, privateKey) => {\n' +
    '    // Handle errors and use the generated key pair.\n' +
    '  });',
  "generateKeyPairSync('ec', {\n    namedCurve: 'sect239k1',\n  });",
  "generateKeyPairSync('rsa', {\n    modulusLength: 2048,\n  });",
  "cipher: 'aes-256-cbc',",
  "generateKeyPairSync('rsa', {\n" +
    '    modulusLength: 4096,\n' +
    '    publicKeyEncoding: {\n' +
    "      type: 'spki',\n" +
    "      format: 'pem',\n" +
    '    },\n' +
    '    privateKeyEncoding: {\n' +
    "      type: 'pkcs8',\n" +
    "      format: 'pem',\n" +
    "      cipher: 'aes-256-cbc',\n" +
    "      passphrase: 'top secret',\n" +
    '    },\n' +
    '  });',
```

Figure 29. A `Set` of 14 related-crypto-material method calls was found in the scan of `relatedCryptoMatTestFile.js`, when 15 were expected.

```
root@DESKTOP-F67KJIL:/home/kurisu/PQC/cyclonedx_pqc/testing/testdirs/javascript# pqcbom -i relatedCryptoMatTestFile.js -o relatedCryptoMatTestFileBOM2
/home/kurisu/PQC/cyclonedx_pqc/testing/testdirs/javascript/relatedCryptoMatTestFile.js
Supported file type: .js
Found require('node'): require('node:crypto')
Node Crypto library found!
findNodeCryptoComponents functions tmpSet:  Set(15) {
  "getDiffieHellman('modp14');",
  "getDiffieHellman('modp16');",
  "hash: 'SHA-256',",
  'generateKey({\n' +
    "    name: 'HMAC',\n" +
    "    hash: 'SHA-256',\n" +
    '    length: 256,\n' +
    "  }, true, ['sign', 'verify']);",
  "generateKey('hmac', { length: 512 }, (err, key) => {\n" +
    '    if (err) throw err;\n' +
    "    console.log(key.export().toString('hex'));",
  "cipher: 'id-aes256-GCM',",
  "generateKeyPair('rsa-pss', {\n" +
    '    modulusLength: 4096,\n' +
    '    publicKeyEncoding: {\n' +
    "      type: 'spki',\n" +
    "      format: 'pem',\n" +
    '    },\n' +
    '    privateKeyEncoding: {\n' +
    "      type: 'pkcs8',\n" +
    "      format: 'pem',\n" +
    "      cipher: 'id-aes256-GCM',\n" +
    "      passphrase: 'top secret',\n" +
    '    },\n' +
    '  }, (err, publicKey, privateKey) => {\n' +
    '    // Handle errors and use the generated key pair.\n' +
    '  })',
  "generateKeyPairSync('ec', {\n    namedCurve: 'sect239k1',\n  })",
  "generateKeyPairSync('rsa', {\n    modulusLength: 2048,\n  })",
  "cipher: 'aes-256-cbc',",
  "generateKeyPairSync('rsa', {\n" +
    '    modulusLength: 4096,\n' +
    '    publicKeyEncoding: {\n' +
    "      type: 'spki',\n" +
    "      format: 'pem',\n" +
    '    },\n' +
    '    privateKeyEncoding: {\n' +
    "      type: 'pkcs8',\n" +
    "      format: 'pem',\n" +
    "      cipher: 'aes-256-cbc',\n" +
    "      passphrase: 'top secret',\n" +
    '    },\n' +
    '  })',
```

Figure 30. After a quick fix to the regular expression, all 15 components where found and created to `relatedCrypoMatTestFileBOM2.json`.

9.1.4 Summary

When the test method calls with incorrect parameter values (such as `algorithm` or `invalidValue`) where removed, the three test files mentioned above contained a total of 45 actual cryptographic components. Of the 30 actual cryptographic components in `jsTestFileBOM.json`, 19 components received `parameterSetIdentifier` and `classicalSecurityLevel` values, meaning that the artifact was unable to extract these values from 11 components. The same pattern followed with the `nistQuantumSecurityLevel`, where 19 components received the value, while 11 did not. Since `jsInvalidTestValues.js` contained only faulty values, it will not be further analyzed here. For `relatedCryptoMatTestFileBOM.json`, all 15 components received the `parameterSetIdentifier` and `classicalSecurityLevel` values, while 7 out of 15 received the `nistQuantumSecurityLevel` value. Overall, 34 out of 45 components correctly received the `parameterSetIdentifier` and `classicalSecurityLevel` values, and 26 out of 45 components received the `nistQuantumSecurityLevel` as intended.

The reasons for the missing values were easily identified when the artifact and the produced CBOMs were analyzed. The `nistQuantumSecurityLevel` values were missing due to a lack of data regarding NIST quantum security levels for cryptographic algorithms and their variants. In this research, a small dataset of cryptographic algorithms and their corresponding NIST quantum security levels was created in the *nistQuantumSecLevels.js* file. However, it includes only some of the main cryptographic algorithms that are frequently highlighted in discussions about quantum computers and post-quantum cryptography. Creating a more comprehensive dataset that contains a large number of cryptographic algorithms and their NIST quantum security levels would be extremely useful for generating CBOMs and could additionally provide valuable information about the security status of the used cryptographic algorithms. While this was outside the scope of this study, it could be an excellent topic for future research.

As for the missing `parameterSetIdentifier` and `classicalSecurityLevel` values, the explanation also lies mostly within missing data. The artifact extracts the needed values from the names of the cryptographic algorithms (such as `192` from `AES-192`), but this method does not work with algorithms that do not include the classical security level

in their name, such as `des3`, `blowfish`, or `id-smime-alg-CMS3DESwrap`. Consequently, these cryptographic algorithms do not receive the required values. Similarly to the `nistQuantumSecurityLevel`, addressing this deficiency would require generating a comprehensive list of cryptographic algorithms and their classical security levels. This would allow the artifact to locate the algorithm and then search a database for additional information. Although creating such a database was beyond the resources allocated for this research, its development would greatly enhance the accuracy of CBOMs and is a great topic for future research.

9.2 Second evaluation

The second evaluation follows a similar approach to the first, with the key difference being the use of test data created by ChatGPT (OpenAI 2024). Additionally, the artifact will be analyzed to determine if the minimum requirements outlined in section 7.9 have been met. The file created by ChatGPT (OpenAI 2024), along with the resulting CBOM file, can be found in Appendix A, under the names of `cryptoExamples_by_ChatGPT.js` and `cryptoExamplesBOM.json`.

The `cryptoExamples_by_ChatGPT.js` contained 21 method calls for creating cryptographic components, four of which contained parameters that the artifact was unable to interpret. Figure 31 shows that 16 method calls were found during the scan, leaving one component missing. Upon inspection, the missing component was identified as an intentionally removed duplicate value (`generateKeyPairSync('rsa'...)`, confirming that the artifact worked as expected.

```
root@DESKTOP-F67KJIL:/home/kurisu/PQC/cyclonedx_pqc/testing/testdirs/javascript# pqcbom -i cryptoExamples_by_ChatGPT.js -o cryptoExamplesBOM
/home/kurisu/PQC/cyclonedx_pqc/testing/testdirs/javascript/cryptoExamples_by_ChatGPT.js
Supported file type: .js
Found require('node'): require('crypto')
Node Crypto library found!
findNodeCryptoComponents functions tmpSet:  Set(8) {
  "createCipher('aes192',",
  "createCipheriv('aes-256-cbc',",
  "createDecipheriv('aes-256-cbc',",
  "createDecipher('aes192',",
  "createHash('sha256')",
  "createHmac('sha256',",
  "createSign('SHA256')",
  "createVerify('SHA256')"
}
findNodeCryptoComponents functions tmpSet:  Set(8) {
  'createDiffieHellman(2048);',
  "createDiffieHellmanGroup('modp14');",
  "getDiffieHellman('modp14');",
  "createECDH('secp256k1');",
  "generateKey('hmac', { length: 256 }, (err, key) => {\n" +
    '    if (err) throw err;\n' +
    "    console.log('Generated Key (generateKey):', key.export().toString('hex'));",
  "generateKeyPair('rsa', {\n" +
    '    modulusLength: 2048,\n' +
    '}, (err, publicKey, privateKey) => {\n' +
    '    if (err) throw err;\n' +
    "    console.log('Generated Key Pair (generateKeyPair) Public Key:', publicKey.export({ type: 'pkcs1', format: 'pem' })",
  "generateKeyPairSync('rsa', {\n    modulusLength: 2048,\n})",
  "generateKeySync('hmac', { length: 256 })"
}

file cryptoExamplesBOM.json created!
```

Figure 31. Scanning `cryptoExamples_by_ChatGPT.js` resulted a `Set` of eight cryptographic algorithms and eight related-crypto-materials.

Of the four method calls that contained unreadable parameters, three used variables as parameters, and one had a byte buffer parameter (`crypto.createSecretKey(crypto.randomBytes(32));`). The issue with extracting data from the `createSecretKey` method call is that the Node.js Crypto documentation does not specify which cryptographic algorithms can be used. The documentation simply states that the method "creates and returns a new key object containing a secret key for symmetric encryption or Hmac". This challenge needs to be addressed in the future development of the artifact by further analyzing the `createSecretKey` method call. The issue regarding the use of variables as parameters was already identified in the early stages of the artifact's development, but creating a working solution was not possible within the time restrictions of this research.

Further analysis of the cryptographic components in the CBOM file revealed an issue with the code. Two cryptographic components that had the name `SHA256` were created, but both were missing the fields and values of `parameterSetIdentifier`, `classicalSecurityLevel`, and `nistQuantumSecurityLevel`. The `findNodeCryptoComponent` function correctly found both components (`createSign('SHA256')` and `createVerify('SHA256')`), as shown in figure 31. Additionally, their parameters were extracted as expected, which means that the `extractFirstParameter` function also worked as in-

tended. Therefore, the issue is likely to be in the `addComponent` function.

Inspection of the `addComponent` function revealed that the issue was caused by case sensitivity. The `addComponent` function uses the Node.js Crypto method calls `getCiphers()` and `getHashes()` to retrieve arrays of the supported ciphers and hashes in the Node.js Crypto module. These arrays are used in the artifact to check if the found method calls contain any of the corresponding ciphers or hashes, and if a match is found, the data extraction begins. In this case, `SHA256` is not listed in `getHashes()`, but `sha256` is. Fortunately, this was easily fixed by adding the `i`-flag to the regular expression, allowing case to be ignored. A new scan of `cryptoExamples_by_ChatGPT.js` generated a CBOM file named `cryptoExamplesBOM2.json`. In this CBOM, both components received the previously missing values, resulting in a total of 16 cryptographic components that were successfully created from the scanned test file.

9.2.1 Summary

The second evaluation was conducted with a test file that contained 21 cryptographic components, of which the artifact successfully captured 17 and created 16 (one duplicate was removed). Thus, the artifact was unable to capture four cryptographic components. All of the 16 components received the values for `parameterSetIdentifier` and `classicalSecurityLevel`, while 13 received the `nistQuantumSecurityLevel`. As mentioned in the first evaluation, the missing NIST quantum security level values are due to a lack of data regarding cryptographic algorithms and their security categories. Further development of the artifact will require creating such a data set to expand the artifact's scope of functionality.

The minimum requirements set in section 7.9 were designed to guide the artifact development process and establish clear stopping criteria. Next, the results from both evaluations are summarized to assess how effectively the artifact's minimum requirements have been met. The minimum requirements consisted of four main requirements, one of which included a list of sub-requirements. The requirements **"Follows the CycloneDX standard v1.6"** and **"Captures the "required"-fields defined by CycloneDX standard"** are closely

related, and both have been completely achieved by following the CycloneDX-derived BOM specification (see figure 13) during the artifact development process. The requirements of **"Accurately capture cryptographic components"** and **"Capture and present the following cryptographic component BOM fields:"** are also closely related, and both have been at least moderately achieved, as proven in the following analysis where the captured cryptographic components from both evaluations are summarized.

In the first evaluation, 45 out of 45 cryptographic components were successfully identified, while 17 out of 21 were found in the second evaluation. Thus, resulting in a total of 62 out of 66 components identified, with 61 created after a single duplicate was removed. The following modified list of the minimum requirements presents the completion percentage or success rate for each minimun requirement:

- Accurately capture cryptographic components. **(62/66)**
- Follows the CycloneDX standard v1.6. **(100%)**
- Captures the "required"-fields defined by CycloneDX standard. **(100%)**
- Capture and present the following cryptographic component BOM fields **(61 non-duplicate components created)**:
  - Name of the component (`name`) **(61/61)**
  - Type of the component (`type`) **(61/61)**
  - The directory location of the scanned file (`value`) **(61/61)**
  - The asset type of the component. At least capture cryptographic algorithms. (`assetType`) **(61/61)**
  - The identifier for the parameter set of the cryptographic algorithm (`parameterSetIdentifier`) **(50/61)**
  - Bit representation of the classical security level of the cryptographic algorithm (`classicalSecurityLevel`) **(50/61)**
  - The NIST-categorized quantum security level for the cryptographic algorithm (`nistQuantumSecurityLevel`) **(39/61)**

As shown by these statistics, the artifact met the minimum requirements with good results. However, the results were achieved with a relatively small amount of artificial test data,

which may affect the reliability. If actual applications had been scanned in a field testing scenario, the results could have been different, especially since imported cryptographic components and variables as parameters are probably used more than in the test data. Nevertheless, the artifact produced promising results with the time and resources available. The concept was proven successful, but further development is needed to increase the artifact's scope of functionality and value as a tool for supporting PQC migration and cryptographic agility.

10 Conclusions and discussion

In this chapter, the research questions are answered, conclusions are presented, and the limitations of this research, as well as the weaknesses of the artifact, are discussed. Additionally, potential topics for future research are presented.

10.1 Discussion

A primary goal of this research was to gain knowledge through the process of creating and evaluating a CBOM generator. This knowledge will be used to address the following research questions.

- What are the challenges of creating an automated CBOM tool?

Throughout the research, several challenges related to the creation of automated CBOM tools were identified. For instance, including the NIST quantum security level with cryptographic components in CBOMs is important because CBOMs are promoted as tools for enhancing cryptographic agility and preparing for the transition to post-quantum cryptography. However, accurately representing these levels requires a comprehensive dataset in which most cryptographic algorithms, along with their variants, are evaluated and classified according to their respective NIST quantum security levels. Another identified challenge was the difficulty in determining the classical security level of certain cryptographic algorithms, which do not display this information in their names. To address this, a comprehensive dataset containing cryptographic algorithms, their NIST quantum security levels, classical security levels, and other relevant information is needed. Creating such a dataset would be highly valuable for CBOM generation and provide critical insights into the security classifications of various cryptographic algorithms, making it a promising topic for future research.

One of the main challenges identified in this research was the difficulty of accurately extracting cryptographic data. When using regular expressions, as was done in this study, the patterns tend to become long and complex, making them difficult to manage. Additionally, this method requires a deep understanding of the supported programming languages in order

to create accurate regular expressions based on how each language functions, is structured, and is used. Alternative solutions or a hybrid combination of both might perform better in data extraction or require less effort to function properly.

Although some challenges were identified during this study, the scope of functionality of the created artifact is fairly limited. Therefore, CBOM generators that include features like integrity verification, dependency scanning, operating system scanning, or container scanning may have additional challenges that were not encountered in this research.

- Can a automated CBOM tool accurately capture cryptographic components from source code?

As demonstrated in the evaluations, accurately capturing cryptographic components from source code is achievable with a CBOM tool. However, developing such solutions can be resource-intensive, necessitating a limitation on the tool's scope of functionality. If a more generic method for capturing cryptographic components had been used, it is likely that the tool could already scan a variety of systems and different programming language source codes. However, this would also result in less accurate CBOMs, with more false positives and missing cryptographic data. Additionally, increasing the number of captured cryptographic components (including related-crypto-materials, certificates, and their properties) would add complexity to the extracted data and likely further decrease the accuracy of the CBOM tool. In this research, accuracy was valued above the scope of functionality.

- Is it feasible to use regular expressions as a cryptographic component data extraction method?

Regular expressions proved effective for extracting the cryptographic data needed for this study and are likely to remain effective as the CBOM tool evolves and becomes more comprehensive. During the development and evaluation phases, no significant issues arose that would indicate this method would fail in later stages. However, maintaining and updating the long and complex regular expressions could present challenges, which is why alternative methods for storing, managing, and using the regular expressions could be useful as the tool evolves. For example, creating a database for storing and reusing regular expressions could

be beneficial in future development. Additionally, the lack of knowledge and documentation regarding the cryptographic libraries used can complicate the creation of accurate regular expressions, as was the case with the `createSecretKey()` method call in section 9.2.

10.2 Conclusions

This research had two distinct goals: to create a practical tool for supporting PQC migration and cryptographic management, and to generate new knowledge about the creation of automated CBOM generators. The artifact achieved the first goal by accurately capturing cryptographic components from JavaScript files and producing NIST quantum security levels for over 50 percent of the captured components. The second goal was met by answering the research questions in the previous chapter 10.1 and by presenting the conclusions in this chapter.

Although the artifact has been demonstrated as a functional prototype, its impact on supporting PQC migration and cryptographic agility remains limited due to its narrow scope of functionality. Further development of this tool and similar solutions is essential to address the challenges posed by PQC migration. CBOM generators must be capable of scanning various systems (such as Docker containers, GitHub repositories, and operating systems) and handling multiple formats to be sufficiently comprehensive for modern information systems. Additionally, because CBOMs and CBOM generators are relatively new concepts with limited prior research, further studies are necessary as these tools and frameworks evolve. The lack of prior research also limited the possibilities to reflect the study's results based on existing research, which may affect the reliability of this research.

Despite its limitations, the artifact achieved promising results, and the identified deficiencies can be fixed as development continues. The created artifact speaks for the possibilities that such tools can have in increasing the cryptographic agility of systems and thus also in supporting PQC migration. While PQC migration has elements that are independent of cryptographic agility challenges, and vice versa, it is still well established that they also have overlapping challenges (Ott, Peikert, and al. September 2019; Barker, Polk, and Souppaya April 2021; TEAM July 2020). This suggests that enhancing cryptographic agility can

address certain PQC migration challenges, and as a result, CBOM generators like the one developed in this study can aid in supporting PQC migration.

Some aspects of the research may have influenced the results in a favorable manner. For example, the test files used in the first evaluation were created by the artifact's developer, which could introduce various biases, such as confirmation bias. In this case, the test files may have been unintentionally designed to confirm the expected output of the program (Mohanani et al. December 2020). The second evaluation was conducted using data generated by artificial intelligence, as suitable open-source test data was unavailable. The data created by AI may also contain deficiencies or errors that might not have been detected. Additionally, because the artifact is unable to capture components that are defined outside the method call, predefined variables, such as imported cryptographic keys, are not captured by the artifact. Since variables are commonly used in method calls, the field testing of the artifact could result in lower cryptographic component capture rates. It is also worth noting that the test files included only a portion of the ciphers and hashes supported by the Node.js Crypto module. If all supported ciphers and hashes had been included in the test files, the cryptographic component capture rate and the NIST quantum security level creation rates might have been lower.

11 Summary

In this research, an automated CBOM generator was developed in response to the challenges that quantum computer development poses to modern cryptography. The CBOM tool was built using JavaScript as the programming language, and design science principles guided the design and evaluation processes.

The evaluations demonstrated that the created solution can accurately capture cryptographic components from JavaScript source files. However, some deficiencies were found, including the absence of data on the NIST quantum security level categorization of various cryptographic algorithms, which will need to be addressed. The literature review and practical experiments highlighted the possibilities and benefits of CBOMs and automated CBOM generators for enhancing cryptographic agility in modern, complex information systems, which is an essential step for easing the transition to post-quantum cryptography. In addition to validating the effectiveness of automated CBOM generators in supporting PQC migration and cryptographic agility, the research also provided novel insights into the process and challenges of developing CBOM generators. Furthermore, it demonstrated the feasibility of using regular expression searches as a method for extracting cryptographic component data.

Due to resource limitations, the CBOM generator's functionality was restricted to scanning JavaScript source files that use the Node.js Crypto module for creating and managing cryptographic components. Additionally, this version of the artifact only captured and generated cryptographic algorithms, although the capturing and generation of cryptographic certificates and related cryptographic materials were originally intended to be included. Ideally, the artifact would have also been able to scan cryptographic components from JavaScript files using the Web Crypto API. However, these features had to be left for future versions of the artifact due to time constraints.

The evaluation phase of the research was conducted with two separate data sets: one created by the artifact developer and the other generated by created by AI (ChatGPT). Ideally, according to design science research principles, the second evaluation would have been performed as a field test within the intended application environment. However, this was not

possible due to time constraints, and finding suitable open-source datasets for evaluation also proved challenging. These limitations in the test scenarios may have impacted the reliability of the artifact's performance results. Furthermore, the lack of prior research on CBOMs and CBOM generators made it difficult to compare the findings with existing research, potentially affecting the validity of the research.

In the future, the artifact could be developed to include the previously mentioned components for certificates and related-crypto-materials, as well as support for scanning cryptographic components related to the Web Crypto API. Then, functionality for scanning other programming languages could be added, along with the capability to scan GitHub repositories and Docker containers. As development progresses, it will be essential to optimize the creation, usage, and storage of regular expressions to avoid making management and updates overly tedious.

Bibliography

Agency, National Security. September 2022. *Announcing the Commercial National Security Algorithm Suite 2.0.* https://media.defense.gov/2022/Sep/07/2003071834/-1/-1/0/CSA_CNSA_2.0_ALGORITHMS_.PDF.

———. January 2024. *Recommendations for Software Bill of Materials (SBOM) Management.* https://media.defense.gov/2023/Dec/14/2003359097/-1/-1/0/CSI-SCRM-SBOM-MANAGEMENT.PDF.

Ahmed, Maryam, Habeeb Omotunde, Baharan Sanjabi, Amirhossein, and Difo Aldiaiz. November 2012. *Diffie helman and its Use in Secure Internet Protocols.* https://www.researchgate.net/publication/279725863_Diffie_helman_and_its_Use_in_Secure_Internet_Protocols.

Ajtai, M. 1996. ”Generating hard instances of lattice problems (extended abstract)”. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of Computing,* 99–108. STOC ’96. New York, NY, USA: Association for Computing Machinery. ISBN: 978-0-89791-785-8. https://doi.org/10.1145/237814.237838. https://dl.acm.org/doi/10.1145/237814.237838.

al., Aumasson et. November 2022. ”SPHINCS+ update”. https://csrc.nist.gov/Presentations/2022/sphincs-update.

al., Bernstein et. April 2024. ”Classic McEliece: conservative code-based cryptography”. https://csrc.nist.gov/Presentations/2024/classic-mceliece.

al., Robinson et. April 2024. ”PANEL: 4th Round - BIKE / HQC / Classic McEliece”. https://csrc.nist.gov/Presentations/2024/4th-round-pqc-panel.

Alagic, Gorjan, Daniel Apon, David Cooper, Quynh Dang, Thinh Dang, John Kelsey, Jacob Lichtinger, et al. September 2022. *Status report on the third round of the NIST Post-Quantum Cryptography Standardization process,* NIST IR 8413-upd1. National Institute of Standards and Technology (U.S.), Gaithersburg, MD. https://doi.org/10.6028/NIST.IR.8413-upd1. https://nvlpubs.nist.gov/nistpubs/ir/2022/NIST.IR.8413-upd1.pdf.

Anchore. February 2024. ”SBOM Formats, Standards, Requirements, Examples and More”. https://anchore.com/sbom/key-things-to-know-about-sboms-and-sbom-standards/.

Aragon, Nicolas. October 2022. ”BIKE: Bit Flipping Key Encapsulation”.

Arute, Frank, Kunal Arya, Ryan Babbush, Dave Bacon, Joseph C. Bardin, Rami Barends, Rupak Biswas, et al. October 2019. ”Quantum supremacy using a programmable superconducting processor”. *Nature* 574, number 77797779 (): 505–510. ISSN: 1476-4687. https://doi.org/10.1038/s41586-019-1666-5.

Barker, Elaine B., and Quynh H. Dang. January 2015. *Recommendation for Key Management Part 3: Application-Specific Key Management Guidance,* NIST SP 800-57Pt3r1. National Institute of Standards and Technology. https://doi.org/10.6028/NIST.SP.800-57Pt3r1. https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-57Pt3r1.pdf.

Barker, William, W. Polk, and Murugiah Souppaya. April 2021. *Getting Ready for Post-Quantum Cryptography: Exploring Challenges Associated with Adopting and Using Post-Quantum Cryptographic Algorithms,* NIST CSWP 15. U.S. Department of Commerce. https://doi.org/10.6028/NIST.CSWP.15. https://csrc.nist.gov/pubs/cswp/15/getting-ready-for-postquantum-cryptography/final.

Bernstein, Daniel J, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. 2011. *High-speed high-security signatures.* https://eprint.iacr.org/2011/368.

Bernstein, Ethan, and Umesh Vazirani. 1993. ”Quantum complexity theory”. In *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing,* 11–20.

Bluvstein, Dolev, Simon J. Evered, Alexandra A. Geim, Sophie H. Li, Hengyun Zhou, Tom Manovitz, Sepehr Ebadi, et al. December 2023. ”Logical quantum processor based on reconfigurable atom arrays”. *Nature* (). ISSN: 1476-4687. https://doi.org/10.1038/s41586-023-06927-3. https://doi.org/10.1038/s41586-023-06927-3.

Bova, Francesco, Avi Goldfarb, and Roger G. Melko. December 2021. ”Commercial applications of quantum computing”. *EPJ Quantum Technology* 8, number 11 (): 2. ISSN: 2662-4400, 2196-0763. https://doi.org/10.1140/epjqt/s40507-021-00091-1.

Boyens, Jon, Angela Smith, Nadya Bartol, Kris Winkler, Alex Holbrook, and Matthew Fallon. October 2021. *Cybersecurity Supply Chain Risk Management Practices for Systems and Organizations.* https://doi.org/10.6028/NIST.SP.800-161r1-draft2. https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-161r1-draft2.pdf.

Bravyi, Sergey, Andrew W. Cross, Jay M. Gambetta, Dmitri Maslov, Patrick Rall, and Theodore J. Yoder. August 2023. *High-threshold and low-overhead fault-tolerant quantum memory,* arXiv:2308.07915. https://doi.org/10.48550/arXiv.2308.07915. http://arxiv.org/abs/2308.07915.

Buchanan, William. 2017. *Cryptography.* Aalborg, DENMARK: River Publishers. ISBN: 978-1-00-079534-9. http://ebookcentral.proquest.com/lib/jyvaskyla-ebooks/detail.action?docID=5050193.

Burr, W.E. March 2003. ”Selecting the Advanced Encryption Standard”. *IEEE Security & Privacy* 1, number 2 (): 43–52. ISSN: 1558-4046. https://doi.org/10.1109/MSECP.2003.1193210.

Cao, Y., J. Romero, and A. Aspuru-Guzik. 2018. ”Potential of quantum computing for drug discovery”. *IBM Journal of Research and Development* 62 (6): 6:1–6:20. https://doi.org/10.1147/JRD.2018.2888987.

Cao, Yudong, Jonathan Romero, Jonathan P Olson, Matthias Degroote, Peter D Johnson, Mária Kieferová, Ian D Kivlichan, Tim Menke, Borja Peropadre, Nicolas PD Sawaya, et al. 2019. ”Quantum chemistry in the age of quantum computing”. *Chemical reviews* 119 (19): 10856–10915.

Castelvecchi, Davide. December 2023. ”IBM releases first-ever 1,000-qubit quantum chip”. *Nature* 624, number 7991 (): 238–238. https://doi.org/10.1038/d41586-023-03854-1.

Chen, Lily, Stephen Jordan, Yi-Kai Liu, Dustin Moody, Rene Peralta, Ray Perlner, and Daniel Smith-Tone. April 2016. *Report on Post-Quantum Cryptography,* NIST IR 8105. National Institute of Standards and Technology. https://doi.org/10.6028/NIST.IR.8105. https://nvlpubs.nist.gov/nistpubs/ir/2016/NIST.IR.8105.pdf.

Chen, Lily, Dustin Moody, Andrew Regenscheid, and Angela Robinson. February 2023. *Digital Signature Standard (DSS),* NIST FIPS 186-5. National Institute of Standards and Technology (U.S.), Gaithersburg, MD. https://doi.org/10.6028/NIST.FIPS.186-5. https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-5.pdf.

Chuang, Isaac L., Neil Gershenfeld, and Mark Kubinec. April 1998. ”Experimental Implementation of Fast Quantum Searching”. *Phys. Rev. Lett.* 80 (15): 3408–3411. https://doi.org/10.1103/PhysRevLett.80.3408. https://link.aps.org/doi/10.1103/PhysRevLett.80.3408.

”Classic McEliece: Implementation”. October 2022. https://classic.mceliece.org/impl.html.

”Classic McEliece: Intro”. March 2023. https://classic.mceliece.org/index.html.

”CodeQL”. May 2024. CodeQL. https://github.com/github/codeql.

Commission, European. September 2022. *Cyber Resilience Act.* https://digital-strategy.ec.europa.eu/en/library/cyber-resilience-act.

Computer Security Division, Information Technology Laboratory. December 2016. *AES Development - Cryptographic Standards and Guidelines | CSRC | CSRC.* https://csrc.nist.gov/projects/cryptographic-standards-and-guidelines/archived-crypto-projects/aes-development.

Corporation, MITRE. October 2023. *CWE-656: Reliance on Security Through Obscurity (4.14).* https://cwe.mitre.org/data/definitions/656.html.

Cross, Andrew W., Lev S. Bishop, Sarah Sheldon, Paul D. Nation, and Jay M. Gambetta. September 2019. ”Validating quantum computers using randomized model circuits”. *Phys. Rev. A* 100 (3): 032328. https://doi.org/10.1103/PhysRevA.100.032328. https://link.aps.org/doi/10.1103/PhysRevA.100.032328.

Cybersecurity and Infrastructure Security Agency. April 2022. *Apache Log4j Vulnerability Guidance.* https://www.cisa.gov/news-events/news/apache-log4j-vulnerability-guidance.

Cybersecurity, European Union Agency for. May 2021. *Post-quantum cryptography, Current state and quantum mitigation.* Report/Study. https://www.enisa.europa.eu/publications/post-quantum-cryptography-current-state-and-quantum-mitigation.

CycloneDX. 2024. *CycloneDX BOM Specification, version 1.6.* https://cyclonedx.org/docs/1.6/json/.

”CycloneDX Generator”. May 2024. JavaScript. https://github.com/CycloneDX/cdxgen.

Daemen, Joan. April 2004. *Note on naming.* https://csrc.nist.gov/csrc/media/projects/cryptographic-standards-and-guidelines/documents/aes-development/rijndael-ammended.pdf#page=1.

Deutsch, David, and Roger Penrose. 1985. ”Quantum theory, the Church–Turing principle and the universal quantum computer”. *Proceedings of the Royal Society of London. A. Mathematical and Physical Sciences* 400 (1818): 97–117. https://doi.org/10.1098/rspa.1985.0070. https://royalsocietypublishing.org/doi/abs/10.1098/rspa.1985.0070.

Diffie, W., and M. Hellman. 1976. ”New directions in cryptography”. *IEEE Transactions on Information Theory* 22 (6): 644–654. https://doi.org/10.1109/TIT.1976.1055638.

Dworkin, Morris J. May 2023. *Advanced Encryption Standard (AES),* NIST FIPS 197-upd1. National Institute of Standards and Technology (U.S.), Gaithersburg, MD. https://doi.org/10.6028/NIST.FIPS.197-upd1. https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197-upd1.pdf.

Elgamal, T. July 1985. ”A public key cryptosystem and a signature scheme based on discrete logarithms”. *IEEE Transactions on Information Theory* 31, number 4 (): 469–472. ISSN: 0018-9448. https://doi.org/10.1109/TIT.1985.1057074.

Fall, K.R., and W.R. Stevens. 2011. *TCP/IP Illustrated, Volume 1: The Protocols.* Addison-Wesley Professional Computing Series. Pearson Education. ISBN: 978-0-13-280818-7. https://www.r-5.org/files/books/computers/internals/net/Richard_Stevens-TCP-IP_Illustrated-EN.pdf.

”Fast-Fourier Lattice-based Compact Signatures over NTRU”. No date. Visited on May 11, 2024. https://falcon-sign.info/.

Feynman, Richard P. 1982. ”Simulating physics with computers”. *International Journal of Theoretical Physics* 21 (6-7): 467–488. https://doi.org/10.1007/BF02650179.

Finney, Hal, Lutz Donnerhacke, Jon Callas, Rodney L. Thayer, and Daphne Shaw. November 2007. *OpenPGP Message Format.* Request for Comments, RFC 4880. Internet Engineering Task Force. https://doi.org/10.17487/RFC4880. https://datatracker.ietf.org/doc/rfc4880.

Formlabs. September 2020. *Bill of Materials (BOM): The Bill That Reduces All Others.* https://formlabs.com/eu/blog/bill-of-materials-bom/.

Foundation, OWASP. April 2024a. *Authoritative Guide to CBOM.* https://cyclonedx.org/guides/OWASP_CycloneDX-Authoritative-Guide-to-CBOM-en.pdf.

———. April 2024b. ”CycloneDX v1.6 Released, Advances Software Supply Chain Security with Cryptographic Bill of Materials and Attestations”. https://cyclonedx.org/news/cyclonedx-v1.6-released/.

Fowler, Austin G, Matteo Mariantoni, John M Martinis, and Andrew N Cleland. 2012. ”Surface codes: Towards practical large-scale quantum computation”. *Physical Review A* 86 (3): 032324.

Freier, Alan O., Philip Karlton, and Paul C. Kocher. August 2011. *The Secure Sockets Layer (SSL) Protocol Version 3.0.* Request for Comments, RFC 6101. Internet Engineering Task Force. https://doi.org/10.17487/RFC6101. https://datatracker.ietf.org/doc/rfc6101.

Gambetta, Jay. 2023. ”The hardware and software for the era of quantum utility is here”. Visited on December 4, 2023. https://www.ibm.com/quantum/blog/quantum-roadmap-2033.

Gardner, Martin. August 1977. *Mathematical Games, August 1977.* https://www.scientificamerican.com/article/mathematical-games-1977-08/.

Grassl, Markus, Brandon Langenberg, Martin Roetteler, and Rainer Steinwandt. 2016. ”Applying Grover’s Algorithm to AES: Quantum Resource Estimates”. In *Post-Quantum Cryptography,* edited by Tsuyoshi Takagi, 29–43. Lecture Notes in Computer Science. Cham: Springer International Publishing. ISBN: 978-3-319-29360-8. https://doi.org/10.1007/978-3-319-29360-8_3.

Grimes, Roger. February 2018. https://www.csoonline.com/article/564619/why-arent-we-using-sha3.html.

Grimes, Roger A. 2019. *Cryptography apocalypse: preparing for the day when quantum computing breaks today's crypto.* John Wiley & Sons.

Grover, Lov K. 1996. "A Fast Quantum Mechanical Algorithm for Database Search". In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing,* 212–219. STOC '96. Philadelphia, Pennsylvania, USA: Association for Computing Machinery. ISBN: 0897917855. https://doi.org/10.1145/237814.237866. https://doi.org/10.1145/237814.237866.

Hankerson, Darrel, and Alfred Menezes. 2011. "NIST Elliptic Curves". In *Encyclopedia of Cryptography and Security,* 843–844. Springer, Boston, MA. ISBN: 978-1-4419-5906-5. https://doi.org/10.1007/978-1-4419-5906-5_255. https://link.springer.com/referenceworkentry/10.1007/978-1-4419-5906-5_255.

Hellman, M.E. May 2002. "An overview of public key cryptography". *IEEE Communications Magazine* 40, number 5 (): 42–49. ISSN: 0163-6804. https://doi.org/10.1109/MCOM.2002.1006971.

Hevner, Alan R. 2007. *A Three Cycle View of Design Science Research.*

Hevner, Alan R., Salvatore T. March, Jinsoo Park, and Sudha Ram. 2004. "Design Science in Information Systems Research". *MIS Quarterly* 28 (1): 75–105. ISSN: 0276-7783. https://doi.org/10.2307/25148625.

House, The White. May 2021. *Executive Order on Improving the Nation's Cybersecurity.* https://www.whitehouse.gov/briefing-room/presidential-actions/2021/05/12/executive-order-on-improving-the-nations-cybersecurity/.

———. May 2022. "National Security Memorandum on Promoting United States Leadership in Quantum Computing While Mitigating Risks to Vulnerable Cryptographic Systems". https://www.whitehouse.gov/briefing-room/statements-releases/2022/05/04/national-security-memorandum-on-promoting-united-states-leadership-in-quantum-computing-while-mitigating-risks-to-vulnerable-cryptographic-systems/.

Housley, Russ. August 2018. *Use of the Elliptic Curve Diffie-Hellman Key Agreement Algorithm with X25519 and X448 in the Cryptographic Message Syntax (CMS).* Request for Comments, RFC 8418. Internet Engineering Task Force. https://doi.org/10.17487/RFC8418. https://datatracker.ietf.org/doc/rfc8418.

”How to Use ssh-keygen to Generate a New SSH Key?” No date. https://www.ssh.com/academy/ssh/keygen.

IBM. September 2023. ”IBM Quantum Safe Explorer”. https://www.ibm.com/downloads/cas/O5B0WXVZ.

———. April 2024. ”Cryptography Bill of Materials”. https://github.com/IBM/CBOM.

Iivari, Juhani. January 2007. ”A Paradigmatic Analysis of Information Systems As a Design Science”. *Scandinavian Journal of Information Systems* 19, number 2 (). ISSN: 1901-0990. https://aisel.aisnet.org/sjis/vol19/iss2/5.

Iivari, Juhani, and John Venable. January 2009. ”Action research and design science research - Seemingly similar but decisively dissimilar”. JournalAbbreviation: 17th European Conference on Information Systems, ECIS 2009, *17th European Conference on Information Systems, ECIS 2009* (): 1653.

Järvinen, Pertti. February 2007. ”Action Research is Similar to Design Science”. *Quality & Quantity* 41, number 1 (): 37–54. ISSN: 1573-7845. https://doi.org/10.1007/s11135-005-5427-1.

Johnson, Don, Alfred Menezes, and Scott Vanstone. August 2001. ”The Elliptic Curve Digital Signature Algorithm (ECDSA)”. *International Journal of Information Security* 1, number 1 (): 36–63. ISSN: 1615-5262. https://doi.org/10.1007/s102070100002.

Jones, Jonathan A., Michele Mosca, and Rasmus H. Hansen. May 1998. ”Implementation of a quantum search algorithm on a quantum computer”. *Nature* 393, number 66836683 (): 344–346. ISSN: 1476-4687. https://doi.org/10.1038/30687.

Jonsson, Jakob, and Burt Kaliski. February 2003. *Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1.* Request for Comments, RFC 3447. Internet Engineering Task Force. https://doi.org/10.17487/RFC3447. https://datatracker.ietf.org/doc/rfc3447.

Koblitz, Neal, Alfred Menezes, and Scott Vanstone. March 2000. ”The State of Elliptic Curve Cryptography”. *Designs, Codes and Cryptography* 19, number 2 (): 173–193. ISSN: 1573-7586. https://doi.org/10.1023/A:1008354106356.

Kravitz, David W. July 1993. *Digital signature algorithm,* US5231668A. https://patents.google.com/patent/US5231668/en.

Krawczyk, Hugo, Kenneth G. Paterson, and Hoeteck Wee. 2013. ”On the Security of the TLS Protocol: A Systematic Analysis”. In *Advances in Cryptology – CRYPTO 2013,* edited by Ran Canetti and Juan A. Garay, 429–448. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer. ISBN: 978-3-642-40041-4. https://doi.org/10.1007/978-3-642-40041-4_24.

Labs, AWS. October 2023. *BIKE-KEM: A Post-Quantum Key Encapsulation Mechanism.* https://github.com/awslabs/bike-kem.

Liang, Jie, and Xue-Jia Lai. January 2007. ”Improved Collision Attack on Hash Function MD5”. *Journal of Computer Science and Technology* 22, number 1 (): 79–87. ISSN: 1860-4749. https://doi.org/10.1007/s11390-007-9010-1.

Manin, Yuri. 1980. *Computable and Uncomputable (in Russian).* Sovetskoye Radio.

March, Salvatore T., and Gerald F. Smith. December 1995. ”Design and natural science research on information technology”. *Decision Support Systems* 15, number 4 (): 251–266. ISSN: 0167-9236. https://doi.org/10.1016/0167-9236(94)00041-2.

Maria Schuld, Ilya Sinayskiy, and Francesco Petruccione. 2015. ”An introduction to quantum machine learning”. *Contemporary Physics* 56 (2): 172–185. https://doi.org/10.1080/00107514.2014.964942. https://doi.org/10.1080/00107514.2014.964942.

Markus, M. Lynne, Ann Majchrzak, and Les Gasser. 2002. ”A Design Theory for Systems That Support Emergent Knowledge Processes”. *MIS Quarterly* 26 (3): 179–212. ISSN: 0276-7783.

Martins, Laura. December 2023. *The SBOM Evolution: Is it ready for prime time?* https://cybertechaccord.org/the-sbom-evolution-is-it-ready-for-prime-time/.

Matsumoto, Tsutomu, and Hideki Imai. 1988. *Public Quadratic Polynomial-Tuples for Efficient Signature-Verification and Message-Encryption.* Edited by D. Barstow, W. Brauer, P. Brinch Hansen, D. Gries, D. Luckham, C. Moler, A. Pnueli, et al. 419–453. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer. ISBN: 978-3-540-45961-3. https://doi.org/10.1007/3-540-45961-8_39.

Mavroeidis, Vasileios, Kamer Vishi, Mateusz D. Zych, and Audun Jøsang. 2018. ”The Impact of Quantum Computing on Present Cryptography”. ArXiv:1804.00200 [cs], *International Journal of Advanced Computer Science and Applications* 9 (3). ISSN: 21565570, 2158107X. https://doi.org/10.14569/IJACSA.2018.090354. http://arxiv.org/abs/1804.00200.

McEliece., R. J. April 1978. ”A Public-Key Cryptosystem Based On Algebraic Coding Theory.” *The Deep Space Network Progress Report* Volume 42-44 (): 114–116. https://ipnpr.jpl.nasa.gov/progress_report/42-44/44N.PDF.

Miller, Victor S. 1986. ”Use of Elliptic Curves in Cryptography”. In *Advances in Cryptology — CRYPTO ’85 Proceedings,* edited by Hugh C. Williams, 417–426. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer. ISBN: 978-3-540-39799-1. https://doi.org/10.1007/3-540-39799-X_31.

Mohanani, Rahul, Iflaah Salman, Burak Turhan, Pilar Rodríguez, and Paul Ralph. December 2020. ”Cognitive Biases in Software Engineering: A Systematic Mapping Study”. *IEEE Transactions on Software Engineering* 46, number 12 (): 1318–1339. ISSN: 1939-3520. https://doi.org/10.1109/TSE.2018.2877759.

Moody, Dustin. 2022. *NIST PQC: LOOKING INTO THE FUTURE.*

Moody, Dustin. March 2024. *Are we there yet? An Update on the NIST PQC Standardization Project.* https://csrc.nist.gov/Presentations/2024/update-on-the-nist-pqc-standardization-project.

National Academies of Sciences, Engineering, and Meidicine. March 2019. *Quantum Computing: Progress and Prospects.* Washington, D.C.: National Academies Press. ISBN: 978-0-309-47969-1. https://doi.org/10.17226/25196. https://www.nap.edu/catalog/25196.

Niederreiter, Harald. 1986. ”Knapsack-type cryptosystems and algebraic coding theory”. *Prob. Contr. Inform. Theory* 15 (2): 157–166.

Nielsen, Michael A, and Isaac L Chuang. 2010. *Quantum computation and quantum information.* Cambridge university press.

NIST, Information Technology Laboratory, Computer Security Division. December 2016. ”Public-Key Post-Quantum Cryptographic Algorithms: Nominations | CSRC”. https://csrc.nist.gov/news/2016/public-key-post-quantum-cryptographic-algorithms.

Node.js Developers. 2024. *Node.js Crypto Module Documentation.* https://nodejs.org/api/crypto.html.

”OMB M-23-02”. November 2022. https://www.whitehouse.gov/wp-content/uploads/2022/11/M-23-02-M-Memo-on-Migrating-to-Post-Quantum-Cryptography.pdf.

”BIKE - Open Quantun Safe”. October 2022. https://openquantumsafe.org/liboqs/algorithms/kem/bike.html.

”HQC - Open Quantum Safe”. April 2023. https://openquantumsafe.org/liboqs/algorithms/kem/hqc.html.

OpenAI. 2024. *ChatGPT: A Large Language Model.* https://chat.openai.com/. Accessed: 2024-09-20.

”OpenSSH Legacy Options”. No date. https://www.openssh.com/legacy.html.

Orús, Román, Samuel Mugel, and Enrique Lizaso. 2019. ”Quantum computing for finance: Overview and prospects”. *Reviews in Physics* 4:100028.

Ott, David, Christopher Peikert, and et al. September 2019. *Identifying Research Challenges in Post Quantum Cryptography Migration and Cryptographic Agility.* ArXiv:1909.07353 [cs], arXiv:1909.07353. https://doi.org/10.48550/arXiv.1909.07353. http://arxiv.org/abs/1909.07353.

Overflow, Stack. July 2024. *Most used programming languages among developers worldwide as of 2024.* In Statista. Retrieved September 16, 2024. https://www.statista.com/statistics/793628/worldwide-developer-survey-most-used-languages/.

Perlner, Ray A., and David A. Cooper. 2009. ”Quantum resistant public key cryptography: a survey”. In *Proceedings of the 8th Symposium on Identity and Trust on the Internet,* 85–93. IDtrust ’09. New York, NY, USA: Association for Computing Machinery. ISBN: 978-1-60558-474-4. https://doi.org/10.1145/1527017.1527028. https://dl.acm.org/doi/10.1145/1527017.1527028.

Prange, E. September 1962. ”The use of information sets in decoding cyclic codes”. *IRE Transactions on Information Theory* 8, number 5 (): 5–9. ISSN: 2168-2712. https://doi.org/10.1109/TIT.1962.1057777.

Presman, Dylan. April 2024. *The U.S. Government’s Transition to PQC.* https://csrc.nist.gov/Presentations/2024/u-s-government-s-transition-to-pqc.

”Quantum Technology Monitor”. 2023. Visited on November 24, 2023. https://www.mckinsey.com/~/media/mckinsey/business%20functions/mckinsey%20digital/our%20insights/quantum%20technology%20sees%20record%20investments%20progress%20on%20talent%20gap/quantum-technology-monitor-april-2023.pdf.

Rangari, Shailesh Y. March 2023. https://www.isaca.org/resources/news-and-trends/industry-news/2023/why-are-regulations-demanding-sbom-adoption.

Regev, Oded. 2005. ”On lattices, learning with errors, random linear codes, and cryptography”. In *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing,* 84–93. STOC ’05. New York, NY, USA: Association for Computing Machinery. ISBN: 978-1-58113-960-0. https://doi.org/10.1145/1060590.1060603. https://dl.acm.org/doi/10.1145/1060590.1060603.

Research, Santander Security. January 2024. ”Cryptobom Forge Tool: Generating Comprehensive CBOMs from CodeQL Outputs”. https://github.com/Santandersecurityresearch/cryptobom-forge.

Rittel, Horst W. J., and Melvin M. Webber. 1973. ”Dilemmas in a General Theory of Planning”. *Policy Sciences* 4 (2): 155–169.

Rivest, R. L., A. Shamir, and L. Adleman. 1978. ”A method for obtaining digital signatures and public-key cryptosystems”. *Communications of the ACM* 21 (2): 120–126. ISSN: 0001-0782. https://doi.org/10.1145/359340.359342.

Sampaio de Alencar, Marcelo. 2022. *Cryptography and Network Security.* Aalborg, DENMARK: River Publishers. ISBN: 978-87-7022-406-2. http://ebookcentral.proquest.com/lib/jyvaskyla-ebooks/detail.action?docID=29357120.

Schaad, Jim, Blake C. Ramsdell, and Sean Turner. April 2019. *Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 4.0 Message Specification.* Request for Comments, RFC 8551. Internet Engineering Task Force. https://doi.org/10.17487/RFC8551. https://datatracker.ietf.org/doc/rfc8551.

Schwabe, Peter. December 2020. *Kyber.* https://pq-crystals.org/kyber/.

———. February 2021. *Dilithium.* https://pq-crystals.org/dilithium/index.shtml.

———. August 2023. *SPHINCS+.* https://sphincs.org/.

Shor, P.W. 1994. ”Algorithms for quantum computation: discrete logarithms and factoring”. In *Proceedings 35th annual symposium on foundations of computer science,* 124–134. Ieee. https://doi.org/10.1109/SFCS.1994.365700. https://ieeexplore.ieee.org/document/365700/authors#authors.

”SolarWinds Security Advisory”. April 2021. https://www.solarwinds.com/sa-overview/securityadvisory.

Staff, IonQ. January 2024. ”Algorithmic Qubits: A Better Single-Number Metric”. Visited on December 4, 2023. https://ionq.com/resources/algorithmic-qubits-a-better-single-number-metric.

Stallings, W. 2018. *Effective Cybersecurity: A Guide to Using Best Practices and Standards.* Pearson Education. ISBN: 978-0-13-477295-0.

Stalnaker, Trevor, Nathan Wintersgill, Oscar Chaparro, Massimiliano Di Penta, Daniel M German, and Denys Poshyvanyk. 2024. ”BOMs Away! Inside the Minds of Stakeholders: A Comprehensive Study of Bills of Materials for Software Systems”. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering,* 1–13. ICSE ’24. New York, NY, USA: Association for Computing Machinery. ISBN: 9798400702174. https://doi.org/10.1145/3597503.3623347. https://dl.acm.org/doi/10.1145/3597503.3623347.

Standards, National Institute of, and Technology. May 1994. *Digital Signature Standard (DSS),* Federal Information Processing Standard (FIPS) 186 (Withdrawn). U.S. Department of Commerce. https://csrc.nist.gov/pubs/fips/186/upd1/final.

———. January 2017. *Post-Quantum Cryptography Standardization - Post-Quantum Cryptography.* https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization.

———. February 2023a. *Digital Signature Standard (DSS).* Technical report NIST FIPS 186-5. Gaithersburg, MD: National Institute of Standards and Technology (U.S.) https://doi.org/10.6028/NIST.FIPS.186-5. https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-5.pdf.

———. August 2023b. *Stateless Hash-Based Digital Signature Standard,* NIST FIPS 205 ipd. National Institute of Standards and Technology, Gaithersburg, MD. https://doi.org/10.6028/NIST.FIPS.205.ipd. https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.205.ipd.pdf.

Stephen Hendrick, The Linux Foundation, VP Research. January 2022. *Software Bill of Materials (SBOM) and Cybersecurity Readiness.* https://www.linuxfoundation.org/research/the-state-of-software-bill-of-materials-sbom-and-cybersecurity-readiness.

Suite, BIKE. No date. ”BIKE - Bit Flipping Key Encapsulation”. Visited on May 14, 2024. https://bikesuite.org/.

TEAM, ETSI COMS. July 2020. *CYBER; Migration strategies and recommendations to Quantum Safe schemes.* https://www.etsi.org/deliver/etsi_tr/103600_103699/103619/01.01.01_60/tr_103619v010101p.pdf.

"Why we need to put the brakes on public software bills of material". June 2021. https://techbeacon.com/security/why-we-need-put-brakes-public-software-bills-material.

Technology, National Institute of Standards and. August 2015. *SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions,* Federal Information Processing Standard (FIPS) 202. U.S. Department of Commerce. https://doi.org/10.6028/NIST.FIPS.202. https://csrc.nist.gov/pubs/fips/202/final.

———. August 2024a. *Module-Lattice-Based Digital Signature Standard,* Federal Information Processing Standard (FIPS) 204. U.S. Department of Commerce. https://doi.org/10.6028/NIST.FIPS.204. https://csrc.nist.gov/pubs/fips/204/final.

———. August 2024b. *Module-Lattice-Based Key-Encapsulation Mechanism Standard,* Federal Information Processing Standard (FIPS) 203. U.S. Department of Commerce. https://doi.org/10.6028/NIST.FIPS.203. https://csrc.nist.gov/pubs/fips/203/final.

———. August 2024c. *Stateless Hash-Based Digital Signature Standard,* Federal Information Processing Standard (FIPS) 205. U.S. Department of Commerce. https://doi.org/10.6028/NIST.FIPS.205. https://csrc.nist.gov/pubs/fips/205/final.

Telecommunications, National, and Information Administration. 2021. *Survey of Existing SBOM Formats and Standards.* https://www.ntia.gov/files/ntia/publications/sbom_formats_survey-version-2021.pdf.

Wallden, Petros, and Elham Kashefi. March 2019. "Cyber security in the quantum era". *Communications of the ACM* 62, number 4 (): 120–120. ISSN: 0001-0782, 1557-7317. https://doi.org/10.1145/3241037.

Walls, Joseph G., George R. Widmeyer, and Omar A. El Sawy. March 1992. "Building an Information System Design Theory for Vigilant EIS". *Information Systems Research* 3, number 1 (): 36–59. ISSN: 1047-7047, 1526-5536. https://doi.org/10.1287/isre.3.1.36.

Waltermire, David, Brant Cheikes, Larry Feldman, and Gregory Witte. April 2016. *Guidelines for the Creation of Interoperable Software Identification (SWID) Tags,* NIST Internal or Interagency Report (NISTIR) 8060. National Institute of Standards and Technology. https://doi.org/10.6028/NIST.IR.8060. https://csrc.nist.gov/pubs/ir/8060/final.

Wouters, Paul, and Ondřej Surý. June 2019. *Algorithm Implementation Requirements and Usage Guidance for DNSSEC.* Request for Comments, RFC 8624. Internet Engineering Task Force. https://doi.org/10.17487/RFC8624. https://datatracker.ietf.org/doc/rfc8624.

Xia, Boming, Tingting Bi, Zhenchang Xing, Qinghua Lu, and Liming Zhu. May 2023. ”An Empirical Study on Software Bill of Materials: Where We Stand and the Road Ahead”. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE),* 2630–2642. Melbourne, Australia: IEEE. ISBN: 978-1-66545-701-9. https://doi.org/10.1109/ICSE48619.2023.00219. https://ieeexplore.ieee.org/document/10172696/.

Yargs Developers. 2024. *Yargs.* https://yargs.js.org/.

Zahan, Nusrat, Elizabeth Lin, Mahzabin Tamanna, William Enck, and Laurie Williams. March 2023. ”Software Bills of Materials Are Required. Are We There Yet?” *IEEE Security & Privacy* 21, number 2 (): 82–88. ISSN: 1558-4046. https://doi.org/10.1109/MSEC.2023.3237100.

Appendices

A Artifact source code and test files

Source code: https://github.com/krmapale/CBOM_tool/tree/main/tool

Test files: https://github.com/krmapale/CBOM_tool/tree/main/testing/testdirs/javascript