Road to Cryptographic Agility

Guillaume Michel
Cybersecurity Master Student

School of Computer and Communication Sciences

Diploma Project

March 2021

Responsible
Prof. Serge Vaudenay
EPFL / LASEC

Supervisor
Michael Osborne
IBM Research

Contents

1 Introduction **4**

2 Background **7**
2.1 About quantum computers 7
2.1.1 Quantum computers 7
2.1.2 Implications on today's cryptography 8
2.2 Migration to post-quantum cryptographic algorithms 9
2.3 Cryptographic Policy 10

3 Cryptographic Agility **12**
3.1 Definition 12
3.2 Need for Cryptographic Agility 12
3.3 Existing agile cryptographic libraries 17
3.3.1 Java Cryptographic Architecture (`JCA`) 17
3.3.2 Gnu Privacy Guard Made Easy (`gpgme`) 18
3.3.3 Qt SSL library (`QSsl`) 19
3.3.4 OpenSSL 19
3.4 Agile Cryptographic Framework Requirements 20
3.4.1 Design 20
3.4.2 Interface of cryptographic implementations 21
3.4.3 Interface of the source code 23
3.4.4 Configuration interface 23
3.4.5 Data structures 26
3.4.6 Migration of encrypted data 26
3.4.7 Nature of the agile cryptographic framework 27
3.5 Proof of concept: Signature microservice 31
3.5.1 Design 32
3.5.2 Results 33
3.6 Cryptographic Agility Strategy 33
3.6.1 Migration to an agile framework 33
3.6.2 Data migration 34
3.6.3 Cryptographic Agility in CI/CD 36

4 Cryptographic Inventory 37
4.1 Definition . . . 37
4.2 Need for Cryptographic Inventory . . . 37
4.2.1 Before Agile Cryptography . . . 37
4.2.2 After Agile Cryptography . . . 39
4.3 Inventory building techniques . . . 39
4.4 Existing Cryptographic Inventories . . . 40
4.4.1 Cryptosense Analyzer . . . 40
4.4.2 InfoSec Global AgileScan . . . 40
4.5 Inventory interface . . . 40
4.5.1 Dashboard . . . 41
4.5.2 Output format . . . 44
4.5.3 Diagrams . . . 44
4.6 Cryptographic Inventory in CI/CD . . . 46

5 Implementation 48
5.1 Goals . . . 48
5.2 Design . . . 49
5.3 Challenges . . . 54
5.4 Limitations . . . 55
5.5 Installation and running guide . . . 56
5.6 Future work . . . 65

6 Future work 72

7 Conclusion 74

Chapter 1

Introduction

In the recent years, quantum computers have been evolving rapidly. They are bringing to life many quantum algorithms running exclusively on quantum computers that will be able to solve some problems more efficiently. One of these, Shor's algorithm solves the integer factorization problem in polynomial time. Yet, the security of many cryptographic algorithms securing web browsing, private messages and online payments relies on the hardness of the integer factorization problem, and an algorithm able to solve this problem in polynomial time seriously threatens the security of these applications. So far, quantum computers are not powerful enough to factorize the large integers currently in use, but once they will be, most of the public-key cryptography in use will be vulnerable to attacks from quantum computers. As we rely on these algorithms in our daily life when using a computer or smartphone, the impact would be a cataclysm for the society.

Fortunately, there is a way to avoid this disaster. There exists some public-key cryptographic algorithms for which no effective attack using a quantum computer has been found. Replacing the current algorithms with these *post-quantum* ones would prevent the breakdown of our security protocols. The National Institute of Standards and Technology (NIST) is hosting a competition to define the new post-quantum cryptographic standards, and is expected to announce the winners in the years to come.

However, we are not out of the woods yet. Once the selected algorithms will be announced, the real challenge will be to smoothly transition from the algorithms we are currently using to the new standards. This migration is expected to cause much trouble to developers as migration of cryptographic algorithms is known to be chaotic and such a massive migration has never occurred in the past. Taking the example of MD5, the cryptographic hash function has been found vulnerable to collision attacks in 2005 [19] and cryptographers already recommended a switch to SHA1 from 1996. In 2021, MD5 is still widely used, and its successor, SHA1, has since been deprecated following vulnerabilities discovery [18]. Therefore, after 16 years, the retirement of MD5 is still not settled, which illustrates quite well the messiness of migrations.

Cryptography is currently consumed in a static way, which is not propitious to migrations. Cryptographic algorithms are implemented in numerous libraries in multiple programming languages, and developers consume cryptography by statically calling functions exported by these implementations. The software infrastructure depends among other things on the cryptographic algorithms in use, on the selected security level and digest size. A single update in these parameters can generate significant changes to the overall software infrastructure. For this reason, replacing the use of a cryptographic algorithm by another in a complex project is a tedious work, requiring not only to replace every function call to the deprecated algorithm, but also requiring to adapt all the software infrastructure.

As the nature of this challenge is very practical, only few academic papers discuss the topic. In 2019, many cryptographers participated to a workshop whose goal was to identify research challenges in the area of post-quantum cryptography migration and cryptographic agility [14]. No proposal solving the challenges of cryptographic agility and migration has been published yet, but many companies are starting to plan the migration to post-quantum cryptography ahead. Hence, the industry is currently pushing to find a solution to the big migration problem.

This diploma project is an exploratory work focused on the big cryptographic migration from the pre- to the post-quantum world. The goals of the project are the following: (1) identify the needs of cryptographic migration, (2) discover which challenges cryptographic agility could solve (3) formalize requirements for a successful cryptographic agile framework (4) propose design guidelines for the implementation of such framework, (5) discuss the usefulness of cryptographic inventories and (6) envision the roadmap from legacy to agile cryptography. This work describes a proof of concept and a cryptographic inventory builder implementation, but the source codes are not provided as they are property of IBM Research. This report will not contain any sensitive information from IBM Research.

In Chapter 2, we introduce quantum computers and how they are redefining the game in cryptography. We give a brief introduction to how they operate, and their recent development. We introduce Shor's algorithm and its implications on current cryptography. We explain more in details why a migration to post-quantum cryptography is required and how it could be carried out. We also define what a cryptographic policy is, and present some of its use cases.

Chapter 3 begins with the definition of *Cryptographic Agility*. We explain why there is a need for cryptographic agility, and which challenges could be addressed by an agile use of cryptography. Then, we review existing cryptographic libraries with some agile features and evaluate the level of agility. We try to define requirements for a cryptographic agile framework. This contribution contains design guidelines and suggestions on the nature, the architecture and the interfaces of such framework. After this, we describe the agile Signature microservice we built as a simple proof of concept, illustrating some of the mentioned requirements. We conclude this chapter by proposing a strategy to migrate from the use of legacy to agile cryptography.

The usefulness of cryptographic inventories is discussed in Chapter 4. We start by introducing the notion of cryptographic inventory. We then detail the need for cryptographic inventories and the problems it tackles, before and after the switch to agile cryptography. We discuss the different techniques to build the inventory of a software, with a special focus on automated methods. We introduce the Software Bill of Material and explain how it can be used with a focus on cryptography dependencies. We review two existing cryptographic inventories services from *Cryptosense* and *Infosec Global*. We explore the different possible interfaces for cryptographic inventory tools, and examine the potential features of a dashboard graphical interface. We consider the nature of the cryptographic inventory output, and present appropriate diagrams for the data visualization of dependency graphs. We conclude the chapter with a discussion on the role of cryptographic inventory in Continuous Integration / Continuous Deployment (CI/CD) pipelines.

In Chapter 5, we describe the implementation of the purpose built cryptography inventory tool for Go implementations. We start by stating the goals of this tool. We then detail the architectural design, and explain the strategical choices undertaken during implementation. We describe the main challenges we had to overcome in the implementation as well as the limitations for this project. We provide an installation and running guide, demonstrating an example of how to build a cryptographic inventory using the tool. We finish the section with a description of potential features which could be added as a future work.

The future work of this project is presented in Chapter 6. It describes the next steps following the creation of a cryptographic inventory builder on the road to cryptographic agility. It provides milestones for the development of cryptographic agility, and for the migration from legacy to agile cryptography.

We finally conclude this report in Chapter 7, where we restate the main results of this diploma project. We highlight the need for cryptographic agility and its goals. We give suggestions on how it could be reached from what we have learned in this project. We underline the importance of building and using cryptographic inventories, in the migration process to either post-quantum or agile cryptography.

Chapter 2

Background

In this chapter, we briefly introduce the different concepts and technologies related to cryptographic inventory and cryptographic agility. We first shortly discuss about quantum computers and their implications in today's cryptographic world. We discuss the need for a migration of cryptographic algorithms, and the challenges caused by such a migration. We finish this chapter by giving a description of cryptographic policies, and their use cases.

2.1 About quantum computers

2.1.1 Quantum computers

We will not expand much on this topic, as quantum physics and quantum computing are not the subject of this work. This section offers a very high level summary on what quantum computers are, and what they are capable of doing.

A quantum computer is a type of computer making use of quantum physics to perform certain kinds of operations more efficiently compared with a classical computer. The difference between a classical and a quantum computer is the following: the classical computer handles information as *bits*, which can be represented as `0` and `1`, whereas quantum computer handles information as *qubits*, which is a quantum state that can be represented either by `0`, `1` or by a superposition of `0` and `1`. As long as the value of a qubit is not measured, its state can be `0`, `1` or a superposition of these two states. When the value of the qubit is measured, the qubit is revealed to be either `0` or `1` according to a certain probability [13]. These property from quantum physics allows computer to perform some computations more efficiently than regular computers, but the results of these computation might be inaccurate in some cases. To solve hard problems, a quantum computer needs to run quantum algorithms, designed to solve a specific problem. Such algorithms don't always exists, thus there are some problems that can not be solved more efficiently by a quantum computer than by a regular computer. People have been working on quantum computing since 1968 [20], but the first experimental quantum computer able to process only 2 qubits was built in 1998.

The *computational power* of a quantum computer is determined by the number of qubits available for use. A quantum computer with a large number of qubits will be able to solve problems more complicated than a quantum computer with a low number of qubits. Large companies are investing a lot in research and development of quantum computers. In 2019, Google demonstrated that they were able to solve complex problems using their 54-qubits quantum computer [3]. IBM recently announced their road map to reach a quantum computer with the ability to process more than a thousand qubits by 2023 [9]. Quantum computer require to be able to process a larger number of qubits to be efficient in solving useful problems, and this time will probably arrive soon, given the exponential growth in the number of qubits a quantum computer is able to process.

2.1.2 Implications on today's cryptography

Shor's algorithm [17], published in 1994, is a polynomial-time quantum algorithm for integer factorization. It can basically retrieve all prime factors for any given integer `N` in a time polynomial to $\log N$. The current largest number factored using Shor's algorithm is 35, and was factored by IBM in 2019 [2]. This shows that quantum computers are not ready to factor large integers yet.

As Shor's algorithm is made for integer factorization, and can run in polynomial time, quantum computers are a threat to today's cryptography relying on the hardness of large integer factorization, discrete logarithm problem and elliptic curve discrete logarithm problem. Most public-key cryptographic schemes currently used, including RSA, and elliptic curves schemes, depend on the hardness of these problems, and will thus be broken by quantum computer, running Shor's algorithm. HTTPS, TLS, SSH, VPN, cryptocurrencies etc. will all become vulnerable to quantum computers, which means the security protecting us when we browse the web, exchange messages, make online payments will break as soon as a quantum computer is able to factorize large numbers. Moreover, when these computers become available, they will be able to break any encrypted message recorded in the past, meaning that encrypted top secret messages exchanged today between governments can be decrypted in the future.

Fortunately, cryptographers are developing new algorithms that are qualified of *Post-quantum* or *Quantum-resistant*. They aim to replace currently used public public-key encryption, key-establishment and digital Signature schemes, vulnerable to Shor's algorithm. These algorithms, running on regular computer, rely on the hardness of problems that cannot be solved more efficiently by quantum computers, yet. The National Institute of Standards and Technology (NIST) initiated a competition to define the post-quantum cryptographic algorithms standards. All submissions have been reviewed and evaluated during the first two rounds, and the finalists have been announced in July 2020 [1]. In this contests, there are two categories of algorithms, the first one is public-key encryption and key-establishment algorithms and the second one is for digital signature Algorithms. Among the finalists, one or multiple candidates will be selected to be the new standard(s), for each category. There might be multiple algorithms selected in the same category, as they are different metrics for evaluation, depending on the context, the optimal algorithm will differ. These submissions are using approaches such as lattice-based cryptography, code-based cryptography, supersingular elliptic curve isogeny cryptography, but I will not get into their details as it is out of the scope of this project.

The situation is less alarming concerning symmetric-key cryptography. Grover's algorithm [4], would allow quantum computers to find the unique input to a black box function producing a particular output value, with high probability, in $\mathcal{O}(\sqrt{N})$ operations, N being the size of the function's domain. Hence, it could potentially provide a quadratic speedup for exhaustive key search in block ciphers. It is not proved that this algorithm will ever be practically relevant for this purpose, but even if it is, doubling the key size for symmetric-key algorithms is sufficient to preserve the same security level [5]. Thus, making sure symmetric-key algorithms stay secure in the quantum world, would not require much change.

2.2 Migration to post-quantum cryptographic algorithms

As discussed in 2.1, once they will have a sufficient amount of qubits, quantum computers will be able to break most security protocols used today, with a retroactive effect. To make sure that all computer security will not collapse once the quantum computers are out, all protocols, infrastructures and applications making use of vulnerable cryptographic primitives need to migrate to quantum-resistant alternatives. Even though, quantum computers are not yet available and powerful enough to break security standard, sensitive encrypted data can be recorded, and decrypted once quantum computers are ready. Hence it is best if this migration is realized as soon as possible. It would be disastrous for any entity to keep using cryptographic algorithms vulnerable to quantum computers, after this turning point.

NIST plans to announce the new post-quantum standard algorithms in the years to come. By then, the implementation of these algorithms will become available, and cryptographic programming libraries will include these new implementations. The developers will then have to migrate all vulnerable algorithms they are using to quantum-safe ones, and make sure that this migration does not break anything in the existing code. Multiple challenges will be faced during the migration. (1) Most cryptographic libraries are not agile, which means that when developers use cryptography, they have to call one of the library's function, corresponding to a specific algorithm. Hence, a migration would require the developers to change every single function call to a vulnerable algorithm, by a function call to a safer one, which is a tedious work. (2) Some software have to be redesigned to make use of these new algorithms. Sometimes, software depends on a statically defined key length, or protocol that has to be changed. (3) The developers have to understand the implementation of the new quantum-safe algorithms in order to use them correctly. However, not all developers have a good knowledge of cryptography and security principles, which could lead to a misuse of quantum-safe algorithms, making them vulnerable. (4) A lot of applications have dependencies on other applications and libraries, and it is sometimes hard and tedious to verify which cryptographic algorithms are used by the dependencies. If one of the dependencies has not migrated to quantum-safe cryptographic algorithms, or is not maintained anymore, then the developer has to replace it to make sure their application stays secure.

As of early 2021, it is too early to start a migration to post-quantum algorithms because the standards have not been released yet. However, some companies start to plan this migration ahead. Some of them are looking for new *agile* libraries or frameworks, in order to facilitate the post-quantum migration, and the next migrations that will follow. The next migrations can be caused by the discovery of new attacks against the post-quantum standards, or by new more efficient technologies threatening the cryptography in use. As stated earlier, migrating all uses of a given cryptographic algorithm is a tedious task, so automatizing this process would spare a lot of effort and money. Nevertheless, at least one manual migration is anyway required to migrate to a potentially *agile* cryptographic framework, being able to handle next migrations automatically. Cryptographic inventory tools are very useful for manual migration, as they are able to highlight which pre-quantum algorithms have not been migrated yet.

2.3 Cryptographic Policy

A cryptographic policy is a set of rules describing how cryptography should be used. A cryptographic policy can be defined at multiple levels. It can be defined at a company-wide level, implying it should be followed for all implementations and activities related to the company, at a department level, or simply on a specific project. Cryptographic policing is also useful for compliance to new privacy laws, such as the European General Data Protection Regulation (GDPR). More and more companies are adopting a cryptographic policy, and many of them are publicly releasing it. As there are not information security experts in every business, multiple companies ask for external consulting firms to build a cryptographic policy for them.

When looking at concrete cryptographic policies, we observe that some of them are very vague, while some other are really precise. Some example of weak policing include requiring the use of encrypted email services and Virtual Private Network (VPN), sensitive data transfer over HTTPS. Some companies have a white list of authorized protocols and modes for remote data access, such as FTP or SCP. Policies can also include advice for key management. Well designed security policies even describe which algorithm should be used for different actions, such as encryption, key exchange, signature etc., describing precisely the key length and the parameters of the cryptographic algorithms. Weak and broken algorithms such as MD5 are often forbidden for use.

Cryptographic policies are currently enforced manually. Developers have to make sure that their code is compliant with the policy, and that they do not use an external dependency using inappropriate cryptographic algorithms. A cryptographic inventory tool can help developers to keep track, of the cryptographic algorithms in use, and they can compare the inventory with the list of cryptographic primitives allowed by the policy. However, there is for now no inventory tool giving an automated feedback for a specific cryptographic policy, which would be quite useful.

Chapter 3

Cryptographic Agility

3.1 Definition

Cryptographic agility is defined as a software design practice encouraging the quick adoption of new cryptographic algorithms and primitives. It allows developers to update the use of cryptographic algorithms in software implementations without making significant changes to the software infrastructure. A software or system can be considered as crypto-agile if its security parameters and cryptographic algorithms can be easily changed in an automated way. Cryptographic agility inherits from most Agile Software Development principles, such as allowing implementation changes at a late stage of a project. Cryptography agility helps to better understand how cryptography is used, and has the capability to build cryptographic inventories, that can be use to certify compliance with security regulations.

In this section we will define the goals of an agile cryptographic *framework*, which is a structure providing cryptographic agility for software. We use the word *framework*, as the nature of the structure can take multiple forms, that we will describe in this section.

3.2 Need for Cryptographic Agility

As described in Section 2.1, the arrival of powerful quantum computers will cause a huge migration from all systems, software and protocols to quantum safe cryptographic algorithms. Most of the current code bases do not use agile cryptographic frameworks, hence the migration will be a tedious manual work. As not all software engineers have a background in cryptography, they may not migrate to the new quantum algorithms correctly, introducing security vulnerabilities in their applications. For these reasons, a manual migration of cryptographic algorithms is to be avoided if possible.

As computer hardware and software evolve at a fast pace, no cryptographic algorithm is time proof. There will always be a need for cryptographic migration, as research advances in this domain. Large scale migrations such as the migration to quantum safe cryptography will not happen at frequent time intervals, but migrations are unavoidable in order to keep security guarantees. Consequently, a solution to avoid periodic manual migrations is to automate these migrations. This would allow Chief Information Security Officer (CISO), or any high level security policy maker to perform automated migrations by themselves, in a fast, secure and inexpensive way.

Moreover, cryptographic agile frameworks could also provide solutions to the following challenges:

1. **Algorithms retirement**: As cryptographic algorithms are not time proof, they need to be retired, once they are considered as insecure, or after new algorithms are able to replace them. The consequence following the retirement of an algorithm consists in replacing the deprecated function by the new standard. This mini-migration has a significantly lower impact on the security of software, compared with the future retirement of the set of all non-quantum cryptographic algorithms, however it should not be neglected.

   MD5 is a good example of algorithm retirement. MD5 is a cryptographic hash function introduced in 1991 by Ronald Rivest [15], and was widely used and considered as a standard. However, researchers discovered vulnerabilities in 1996, and a collision vulnerability was found in 2005 [19], making this algorithm insecure for use, when relying on the collision resistance property. Hence cryptographers recommended the retirement of MD5, replacing it with SHA1. In 2021, after cryptographers showed that SHA1 is vulnerable to collision attacks [18], MD5 is still widely used by many companies. This means that the transition is not finished yet, although the successor of MD5 is already deprecated. This example illustrates the need for a quick and automated methodology to retire weak algorithms.

2. **Flexible security level**: The security level of a cryptographic algorithm, usually depends on the key length and is often expressed as *n-bits* security. Currently, to change the security level of a cryptographic function in an implementation, the developer has to go through all of the function calls, and update bit security parameter for each use. Having a cryptography agile framework would allow to easily update the bit security level without having to manually edit the source code.

   For example, symmetric encryption algorithms such as AES, will keep the same security level in the post-quantum world, if the key size is doubled. Thus, there is a need to have a flexible security level. Moreover, as computer are always more powerful, if the algorithms we are currently using remain secure over time, we will need to increase the key size at a point to keep the same security guarantees.

3. **Platform compatibility**: Ideally, the use of cryptographic libraries and implementation should be platform agnostic. It is true that different platforms might have different architectures, and thus different software optimizations, but a cryptographic library should be used similarly on all platforms. An agile cryptographic framework should be usable for all platforms (Linux, Windows, IBM Z, Android etc.) in order to allow software developers to build agile and platform agnostic applications. Ideally, the same library should be usable by all programming languages. In practice, it may not be optimized, but having bridges from all programming languages would allow to have a single trusted framework grouping all algorithms, making it convenient for use.

4. **Implementation agility**: Software developers should be able to use an agile cryptographic interface to make abstraction of cryptographic algorithms in use. This means that the source code will not make direct function calls to specific algorithms, nor store the keys in an array which size is defined as a constant. This would allow developers to update the cryptographic algorithms used by their software in an agile way, without modifying the source code. This would make applications cryptography agnostic, not depending directly on a specific cryptographic implementation. Implementation agility would also allow software developers to simply and safely use cryptography, with a minimal risk of misusing. Hence, developers don't need to have a background in cryptography to make a correct use of it.

   For example, instead of calling the method `aes128_keygen()` to generate a key, followed by `aes128_encrypt(key, data)` to encrypt some data, a developer would call the method `enc_keygen()` to generate an encryption key for an abstract encryption algorithm, and then call the method `encrypt(key, data)` to get a ciphertext. There are two possibilities for the agile cryptographic library: (1) the library determines by itself which algorithm should be used or (2) the library is configured by a cryptographic policy, describing which algorithm should be used.

5. **Cryptography policing**: An agile cryptographic framework would give the opportunity to regulate which cryptographic algorithms and protocols are used, at a company-wide level. A company, department team, or project team can define a cryptographic policy, describing which and how algorithms and protocols should be used. The policy is often use to comply with local data regulations. The agile framework could provide a list of used algorithms, useful to verify compliance with the policy.

Furthermore, the framework would define a clear format for cryptographic policing, for example using YAML data serialization language. A policy in such a format could be given as input to the cryptographic framework, describing how it should behave. It would for example define which algorithms should be used for encryption, key exchange, signature etc. The policy could be stored at the root of the project, and migrating to a new algorithm should be as simple as editing the corresponding line in the policy text file.

6. **Encrypted Files Migration**: Another component of cryptographic agility is the ability to migrate encrypted data. This migration to new cryptographic algorithms is easy as long as no encrypted data is stored, but encrypted data stored on disk or in the cloud poses new challenges. For example, in an instant peer-to-peer messaging app, the migration would be complete as long as the messages are exchanged using the new cryptographic algorithms, supposing messages are stored in clear text on the end devices. However, if messages are stored encrypted on the devices, the use of new algorithms would mix messages encrypted with different algorithms. This is undesired, because messages encrypted using the old algorithm would potentially be insecure, in the case the migration happened to replace a vulnerable algorithm. And we don't want to deal with multiple algorithms when decrypting messages for the sake of simplicity. Thus, encrypted messages already stored at the moment of the migration have to migrate too. This challenge being quite complex, we will expand more in Section 3.6.2

7. **Contextual Agility**: The agile framework should be able to define an appropriate cryptographic policy from contextual details, helping compliance with local governmental regulations. By scanning simple systems parameters, such as geographical region, or system type (personal laptop, physical server, virtual machine etc.) the framework could decide which cryptographic algorithms to use automatically. For instance, a server located in Europe storing customers personal information will select cryptographic algorithms compliant with the GDPR.

   Given the context in which cryptographic functions are called, the framework should be able to determine which algorithm would be the most appropriate. For example, highly confidential data exchanged over the internet will not use the same encryption, as a computationally limited IoT device sending encrypted logs over a private network.

8. **Quick vulnerability patch**: When a vulnerability or exploit is discovered in a cryptographic algorithm or implementation, it would be easy for developers to know quickly if their application is vulnerable to the weakness, as the agile framework provides a list of all cryptographic primitives in use. If they find their software vulnerable, they

would be able to temporally switch to another algorithm, while the implementation is being fixed. Once a patch is available, it can simply be applied on the agile framework, so no source code has to be modified.

We illustrate this argument with the example of Heartbleed [7]. Heartbleed is a security bug in the implementation of the OpenSSL library, widely used, including in the TLS protocol implementation. The bug was discovered in 2014, and a patch was immediately issued. Even though most devices quickly received the patch, 91,063 devices were still vulnerable to the attack 5 years after the patch was published [16]. Having the ability to patch automatically and quickly security vulnerabilities would solve these kind of issues.

9. **Ease of use**: An agile cryptographic framework would allow developers without further knowledge in cryptography and security protocols to use cryptographic algorithms correctly. Many developers end up working with cryptography, although they don't have a proper background in information security. Thus, it is common that vulnerabilities are found in software because cryptographic algorithms have not been used correctly by the developers. For example, some IV are not randomly generated, which makes them predictable, nonces are reused. Evidently, this can introduce severe security vulnerabilities in the developed software and systems.

   A study analyzed 95 applications from the Apple App Store, and discovered that 64 of them contains security flaws of various severeness, due to cryptographic misuse [12]. Having a framework giving a high level of abstraction on cryptographic algorithms to developers could avoid the introduction of cryptrographic misuse. The framework could handle key management, IV and nonces generation, security parameters etc.

10. **Algorithm negotiation**: Having a cryptographic agile framework would allow client-server or peer-to-peer (p2p) applications to dynamically negotiate the cipher suites they will use. The list of accepted cipher suites, with order of preferences could be given as input for each system, and would facilitate the negotiation. This would allow systems to use their favorite cryptographic algorithms, instead of using the ones defined as default. Furthermore, this would allow parties to use algorithms that are compliant with the local regulations (e.g GDPR). Thus, more freedom of choice can be given to experienced developers, willing to use alternate algorithms as they may be more optimized for a specific task. Moreover, as the list of accepted ciphers should contain only cryptographic algorithms and protocols considered as safe by the entity, this serves as a mitigation against downgrade attacks. In

the case of decentralized p2p systems, it is beneficial to support agile negotiation in a set of multiple cryptographic algorithms, otherwise, if the only communication protocol gets updated for one of the peers, it cannot communicate anymore with the rest of the network.

11. **Composability agility**: Cryptographers and security software developers should be able to use existing cryptographic primitives in order to build new protocols, or security tools in an agile way. Thus the cryptographic framework should provide a large interface for security developers willing to compose existing primitive to create new schemes easily. This can be seen as an *advanced* mode.

12. **Future proof**: As stated before, the ability to quickly and automatically perform cryptographic algorithms migrations, when some algorithms are not considered as safe anymore. However, it is likely that new cryptographic primitive will appear. They will not necessarily replace older schemes because they are safer, but may be more efficient or providing different properties. For example, Fully Homomorphic Encryption (FHE) is a form of encryption permitting to perform operations on encrypted data. FHE offers new capabilities, even though it does not replace any existing technology. There might be new cryptographic offering new capabilities in the future, such as quantum cryptographic algorithms and thus an agile framework should be ready to include them, and facilitate migration from legacy algorithms.

13. **Framework agility**: Last but not least, the cryptographic agile framework should be agile itself, in addition to providing agile cryptography. This framework should be build using agile development techniques, and easy to update, and add new primitives. It should support cryptographic algorithms implementations from multiple providers.

3.3 Existing agile cryptographic libraries

In this section, we review existing cryptographic libraries that have some agile components. We will also show why the level of agility is not enough to fulfill the needs described in Section 3.2. The list of described libraries is non exhaustive, and includes the libraries with interesting agile properties.

3.3.1 Java Cryptographic Architecture (`JCA`)

Java Cryptographic Architecture (JCA) is Java default cryptographic library. Its structure is interesting as it offers string defined modular cryptographic functions. Listing 3.1 illustrates how to sign data using `JCA`. The algorithms used for the signature are specified as strings given to the

`KeyPairGenerator.getInstance()` and `Signature.getInstance()` functions. Hence, the function calls are not static function calls to a specific algorithm implementation, which gives an opportunity for agility. Migrating from one cryptographic algorithm to another would only take to modify the strings inside of the function calls, no data structure needs to be adapted. To gain even more agility, it would be possible to define a Java source code file containing a cryptographic policy, in the form of constant strings defined for each capability. In this setting, a migration would only require to modify this file defining which algorithms are used. However, it is not possible to simply migrate the algorithm used to encrypt data at rest on disk, and there is not support for a migration to post-quantum cryptography.

```java
//Creating KeyPair generator object
KeyPairGenerator keyPairGen = KeyPairGenerator.getInstance("DSA
    ");
//Initializing the key pair generator
keyPairGen.initialize(2048);
//Generate the pair of keys
KeyPair pair = keyPairGen.generateKeyPair();
//Getting the private key from the key pair
PrivateKey privKey = pair.getPrivate();
//Creating a Signature object
Signature sign = Signature.getInstance("SHA256withDSA");
//Initialize the signature
sign.initSign(privKey);
//Adding data to the signature
sign.update("Hello world".getBytes());
```

Listing 3.1: Example of signature in Java

3.3.2 Gnu Privacy Guard Made Easy (gpgme)

Gnu Privacy Guard Made Easy (`gpgme`) is a C, C++ and Python library designed to make access to Gnu Privacy Guard (GnuPG) easy to application developers. GnuPG[1] is an open source implementation of the OpenPGP standard, allowing to sign and encrypt data and communications. `gpgme` is the interface to GnuPG, and the communication between the interface and the server is made through the `libassuan` library, which is a purpose built IPC medium [11]. The overall architecture is quite complicated, but `gpgme` exposes general purpose functions in which a algorithm must be specified. `gpgme` defines enums of public key algorithms and hash algorithms for this purpose. Data is exchanged between `gpgme` and GnuPG, and is stored in generic data buffers, which are consequently generic. Thus, it would be possible as for `JCA` to define all the algorithms that are used through the program as constants in a specific file serving as cryptographic policy. Migrating from one algorithm to another only takes to modify this file. However, as for `JCA`

[1]https://gnupg.org

there is no way to migrate encrypted data, and no possibility to upgrade to post-quantum cryptographic algorithms yet. This library was designed to be agile in the sense that some implementations details should be changed easily before the deployment, but not to support large migrations.

3.3.3 Qt SSL library (`QSsl`)

Qt SSL[2] (`QSsl`) is a C++ library implementing a variety of algorithms and protocols where the caller can specify which algorithm they want to use. It is possible to configure the `QSslConfiguration` object to define which cryptographic algorithms should be used. It is also possible to select which version of TLS should be used, which demonstrate versioning agility. This library is agile, as it supports custom algorithm choice and configuration, but not in the sense of supporting migrations.

3.3.4 OpenSSL

OpenSSL[3] is a software library containing open-source implementations of the SSL and TLS protocols. The main library is written in C, and contains many basic cryptographic functions implementations. It contains wrappers allowing the functions to be used from other programming languages. In OpenSSL current version, 1.1.1, some cryptographic algorithms can agilely be selected as demonstrated in Listing 3.2. The algorithm selection is made at line 5, and only OpenSSL implementations can be picked. This offers an agility level similar to the other discussed libraries.

```
1  EVP_CIPHER_CTX *ctx;
2  EVP_CIPHER *ciph;
3
4  ctx = EVP_CIPHER_CTX_new();
5  ciph = EVP_aes_128_cbc();
6  EVP_EncryptInit_ex(ctx, ciph, NULL, key, iv);
7  EVP_EncryptUpdate(ctx, ciphertext, &clen, plaintext, plen);
8  EVP_EncryptFinal_ex(ctx, ciphertext + clen, &clentmp);
9  clen += clentmp;
10
11 EVP_CIPHER_CTX_free(ctx);
```

Listing 3.2: Encryption in OpenSSL 1.1.1

OpenSSL is planning a transition to version 3.0, which will contain major architectural changes. From an agility perspective, they plan on adding support for cryptographic algorithm implementations from multiple providers. We show in Listing 3.3 how encryption would change between OpenSSL 1.1.1 and OpenSSL 3.0. The algorithm provider is specified on line 5 when the encryption algorithm is selected. The algorithm implementation is identified

[2] `https://doc.qt.io/qt-5/ssl.html`
[3] `https://openssl.org`

by its provider and string identifier. OpenSSL 3.0 will offer more agility, as it will support multiple cryptographic implementation providers, but still offer no solution for migrations.

```
EVP_CIPHER_CTX *ctx;
EVP_CIPHER *ciph;

ctx = EVP_CIPHER_CTX_new();
ciph = EVP_CIPHER_fetch(osslctx, "aes-128-cbc", NULL);/* <== */
EVP_EncryptInit_ex(ctx, ciph, NULL, key, iv);
EVP_EncryptUpdate(ctx, ciphertext, &clen, plaintext, plen);
EVP_EncryptFinal_ex(ctx, ciphertext + clen, &clentmp);
clen += clentmp;

EVP_CIPHER_CTX_free(ctx);
EVP_CIPHER_free(ciph);                                 /* <== */
```

Listing 3.3: Encryption in OpenSSL 3.0

3.4 Agile Cryptographic Framework Requirements

We did not build an agile cryptographic framework, because it would require a full team of experts, and take more time than the duration of this project. However, we gathered the requirements and goals for such a service, and describe a possible high level design for an agile cryptographic framework. We built a simple Agile Signature Microservice serving as proof of concept, that we describe in Section 3.5.

3.4.1 Design

All problems in computer science can be solved by another level of indirection

David Wheeler

One possible way to make cryptography agile is to add a level of indirection in the way we consume cryptography. Currently, programmers have to make direct function calls to specific cryptographic algorithms implementations in order to use cryptographic algorithms, which is why a cryptography migration requires a tedious work. A migration from a cryptographic algorithm to another would require to change all occurrences of an algorithm's function calls to another algorithm, as well as adapting the structure of the software. Figure 3.1 shows the current situation. The source code makes direct calls to cryptographic algorithms implementations, for instance in the OpenSSL library, and in this example, changing the encryption algorithm from `aes128` to `salsa20` would require to modify the source code to call the functions `salsa20_keygen()` and `salsa20_encrypt(data, k)` instead of the functions making use of `aes128`. The change may appear to be simple in this pseudocode, but in practice, there are usually a numerous number of occurrences, and the structure containing the key or the cipher may differ according to algorithms, thus the data structures need to be adapted. This adaptation can require more changes, causing a hard time when migrating cryptographic algorithms.

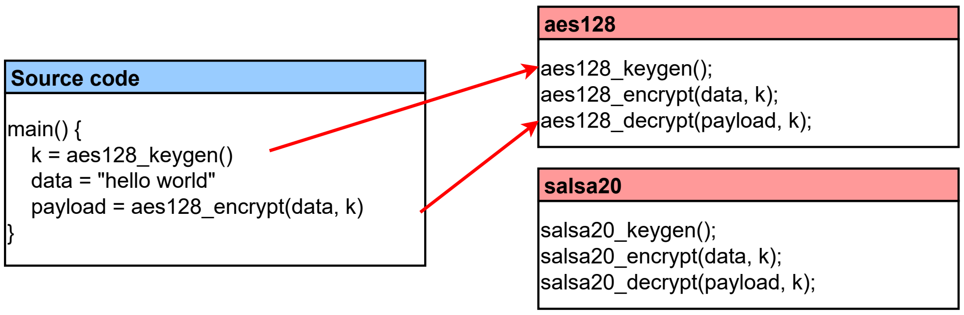
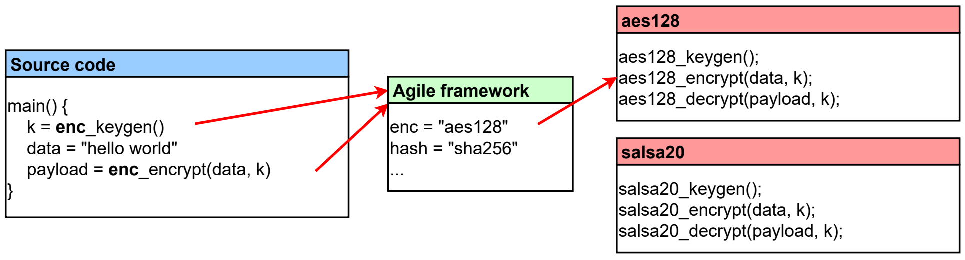


Figure 3.1: Current use of cryptography

Figure 3.2 shows how the agile use of cryptography could look like if we add a layer of indirection. The source code would not directly call the implementation of specific cryptographic algorithms, but instead call generic cryptographic functions exposed by an agile framework, which would be a middleware. This middleware would be configured to redirect the function call to an appropriate function, defined by the middleware configuration. Hence, taking the example, the source code will be updated to call the generic `enc_keygen()` function, returning a key for the selected symmetric encryption algorithm, in a generic encryption key format. The software developer does not have to care about which algorithm to use, and has a less risk to introduce cryptography misuse. An algorithm migration in this setting would be easy to manage. The configuration of the agile framework should be changed, for instance replace the encryption algorithm set to `aes128` to `salsa20`, which is a single line to be modified. However, if data is already stored encrypted, this solution does not solve the problem of encrypted data migration, that we will discuss later.

3.4.2 Interface of cryptographic implementations

We now describe the interface between the cryptographic middleware and the cryptographic algorithms implementation, which correspond to the link *iii.* from Figure 3.3. The cryptographic algorithms implementations should have an interface as simple as possible, without compromising any security. The agile cryptographic framework would certainly need to have a specific way to handle each specific algorithm while sorting the algorithms



```
main() {
    k = enc_keygen()
    data = "hello world"
    payload = enc_encrypt(data, k)
}
```

```
enc = "aes128"
hash = "sha256"
...
```

```
aes128_keygen();
aes128_encrypt(data, k);
aes128_decrypt(payload, k);
```

```
salsa20_keygen();
salsa20_encrypt(data, k);
salsa20_decrypt(payload, k);
```

Figure 3.2: Agile use of cryptography, adding a layer of indirection

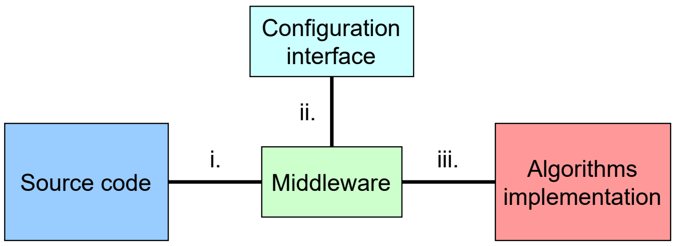


Figure 3.3: Interactions of the cryptographic middleware

by categories, and only exposing these categories to its interface. Thus, each algorithm implementation must belong to one or multiple algorithm categories, for instance using tags, and must respect the constraints of this category. For instance, a symmetric encryption algorithm should have an interface containing at least an encrypt and a decrypt functions, and optionally containing a key generation function. For some algorithms, the key is required to be a random sequence of bytes, and thus can be generated directly by the middleware. For each implementation, the parameters given to the mandatory functions can differ, but the functionality has to be the same, i.e encrypt an array of byte with the given key. Additional parameters, such as Initialization Vectors and Nonces can be handled by the middleware, to have a flat interface for all algorithms in the same category.

The interface itself should be agile, it should be easy to add new cryptographic algorithms to the framework. Ideally, the implementations should also handle versioning, and all versions should be accessible to the middleware. In some cases, a cryptography policy will require to use a specific version of an algorithm that has been in use for many years and has proven to be safe, and in some other cases, the policy will require to use the last version with the last features, but it might have undiscovered flaws. As implementations may be subject to vulnerability patches, performance improvement or other light changes, the middleware should be able to select which version to use according to the context. The agile framework should be able to smoothly update from one version of an algorithm implementation to another, for example after a vulnerability is discovered and a patch is issued. Thus, the interface is supposed to stay the same between different versions, if possible.

3.4.3 Interface of the source code

We describe in this subsection the interface that the cryptographic middleware exposes to developers building source code, which correspond to link *i.* on Figure 3.3. The interface of the framework should be as simple as possible to use. This would make cryptography more accessible to developers, and would avoid implementations flaws in applications as not all developers have a background in security. The interface should expose all types of functions that could be of use for a programmer, i.e key exchange, symmetric and asymmetric encryption, hash, signature, message authentication codes etc. And the programmer should be able to specify some preferences through the interface, for instance the security level or digest size. These settings should of course be handled in an agile way on the program side, they can for example be defined in a file with all constants or in a dedicated cryptography policy file.

Limiting direct access to cryptographic algorithms could cause issues to some developers building new security schemes, or willing to use directly a specific library. These developers may tend not to use an agile framework if the framework limits them in the cryptographic primitives they can use. Moreover, as composability agility is a goal of cryptographic agility, security experts should be able to access all the libraries used by the middleware. As stated before, the interface should be kept simple to use, and exposing all primitives would have a negative impact. Keeping a simple interface for most users and having an *expert* mode for experienced security developers seems to be an optimized trade-off.

Ideally, the middleware would also serve as a key manager, so that the programmer does not have to handle keys and store them. This functionality would add key management challenges to the middleware, which depend on the nature of the middleware itself. Hence, we will use the abstraction that the agile framework acts as a key manager and discuss these challenges later.

3.4.4 Configuration interface

The configuration interface is the interface from which we can set up the agile framework, specifying which and how cryptography must be used. This interface is the link *ii.* from Figure 3.3. This interface has to be as complete and agile as possible, to provide a high level of agility to the users. It is through this interface that a cryptographic policy should be given as input to the cryptographic framework, defining which cryptographic algorithms should be used. It is also through this interface that an *expert* mode could be enabled. There are multiple designs possible for the configuration interface.

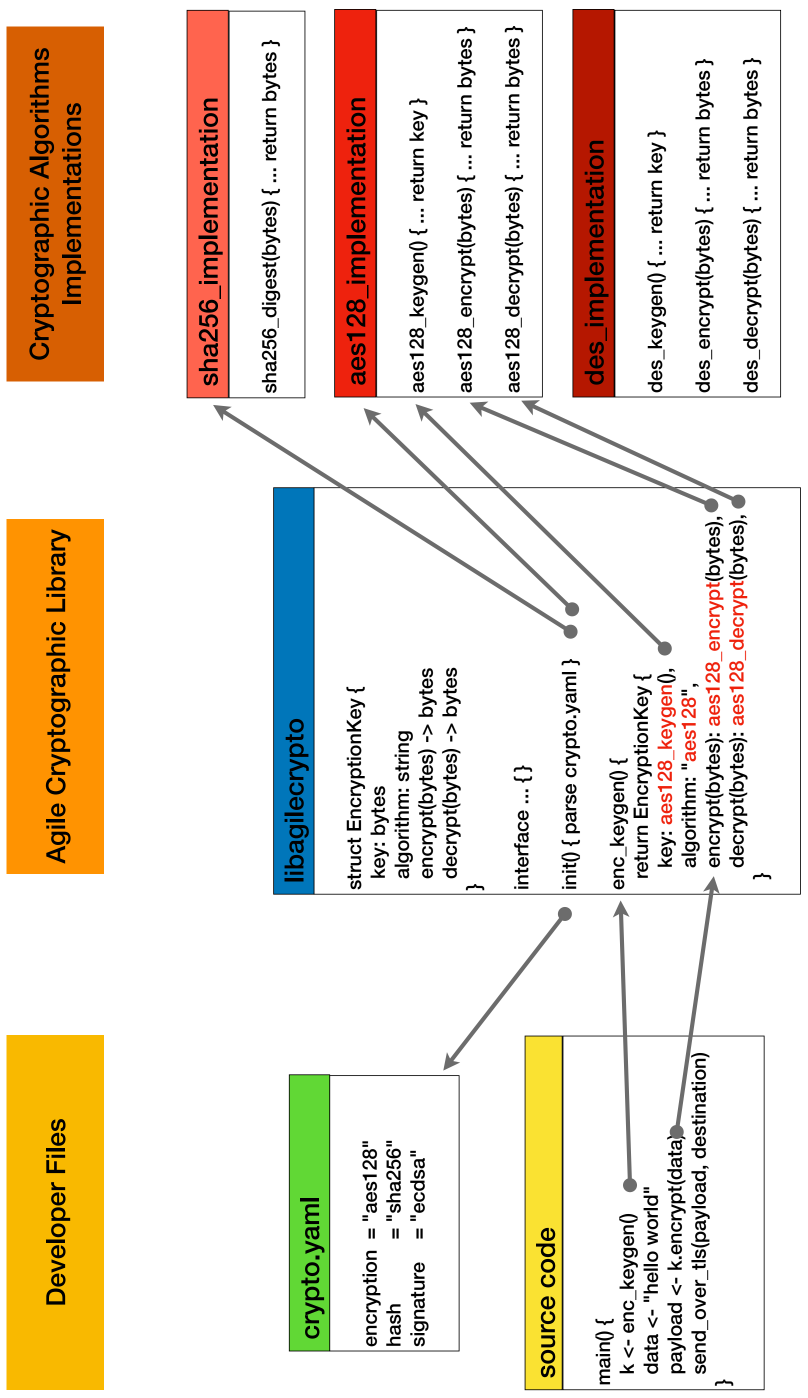
A very simple configuration interface would be to maintain a text, JSON or YAML file at the root of the project, containing a simple cryptographic policy. This file needs to have a standard format, and a specific name, so that the middleware is able to find the configuration file, and parse it to select the appropriate algorithms. Retiring a cryptographic algorithm and replacing it by another one would be as simple as changing the appropriate line in the cryptographic policy that is parsed by the middleware. This solution is simple to implement, and successfully provide cryptographic agility.

Figure 3.4 describes what the architecture of this design would look like. There is a `crypto.yaml` file in the *Developer Files*, which can be the root of the project. This file describe which cryptographic algorithms should be used for `encryption`, `hashing` and `signature`. The `libagilecrypto` have generic types that are defined, for example the `EncryptionKey` that is described. The configuration file will be parsed by the middleware, during its initialization, and the content of the file will define how the functions will behave, see the `enc_keygen()` function as example. This function will redirect to the algorithm that is read in the `crypto.yaml` file. This function will return a generic encryption key object, with defined encryption and decryption methods. These methods are the ones exposed by the selected cryptographic algorithm implementation.

Another possibility for configuration interface would be to run a graphical interface, for instance in the form of a dashboard. This graphical interface could guide the user to design an appropriate cryptography according the the user's needs. Moreover, the dashboard could list all keys and certificates that are used, encrypted files location, and more details that could be useful for developers. A cryptographic inventory could easily be extracted from the cryptographic policy, and a risk assessment can be made. For the risk assessment, it would be possible for the middleware to detect if a cryptographic key is used multiple times, and flag it to the user. Moreover, having an interactive interface could give more granularity to cryptography. For example, if there are two uses of cryptographic hash functions in the application, with different needs, it would be possible to manually select which algorithm is used in each case through the dashboard.

If the middleware has the capability to scan the source code of the application, it could analyze the function calls from the source code to the functions it exposes, and report any inappropriate use. The scan could also discover static uses of cryptography that are to be avoided. This would reduce the number of implementation errors.

A security developer using the *expert* mode would be able to define new groups of algorithms and modify the interface between the framework and the source code, to make it as agile as possible. Through the configuration dashboard, security developers or CISOs could also add bindings to alternate cryptographic algorithms implementations that are not included by default. It would allow them to add proprietary implementations to the



```
encryption = "aes128"
hash       = "sha256"
signature  = "ecdsa"
```

```
main() {
  k <- enc_keygen()
  data <- "hello world"
  payload <- k.encrypt(data)
  send_over_tls(payload, destination)
}
```

```
struct EncryptionKey {
  key: bytes
  algorithm: string
  encrypt(bytes) -> bytes
  decrypt(bytes) -> bytes
}

interface ... {}

init() { parse crypto.yaml }

enc_keygen() {
  return EncryptionKey {
    key: aes128_keygen(),
    algorithm: "aes128",
    encrypt(bytes): aes128_encrypt(bytes),
    decrypt(bytes): aes128_decrypt(bytes),
  }
```

```
sha256_digest(bytes) { ... return bytes }
```

```
aes128_keygen() { ... return key }
aes128_encrypt(bytes) { ... return bytes }
aes128_decrypt(bytes) { ... return bytes }
```

```
des_keygen() { ... return key }
des_encrypt(bytes) { ... return bytes }
des_decrypt(bytes) { ... return bytes }
```

Figure 3.4: Example of an agile cryptographic library architecture.

same framework and the centralize all cryptography at the same place. It would then be possible to export a cryptographic inventory or cryptography configuration, including potential additional third party algorithms implementations, and share this configuration at a team or company level. This would allow a company to have an unified cryptographic policy, that could be adapted for each project.

3.4.5 Data structures

The agile cryptographic framework should define agile data structures to contain data linked to cryptography, i.e keys and cipher payloads. For cipher payloads, a structure could be as described in Listing 3.4. The structure contains which algorithm and mode have been used to produce the cipher, the encrypted data, and optionally any Initialization Vector (IV), the location of the encryption key etc. This kind of data structures would allow the agile framework to handle all ciphers object similarly with generic methods. Of course a support for all algorithms is needed, but having a generic object structure adds a level of indirection and allows the algorithm, and payload to be changed while keeping the same object. These data structures are returned through the interface with the source code, to that the developers do not need to update the data structures when a migration is performed.

```
Cipher structure {
    Algorithm       string
    Mode            string
    Payload         byte[]
    IV              byte[]
    KeyLocation     string
    ...
}
```

Listing 3.4: Possible Cipher object structure

3.4.6 Migration of encrypted data

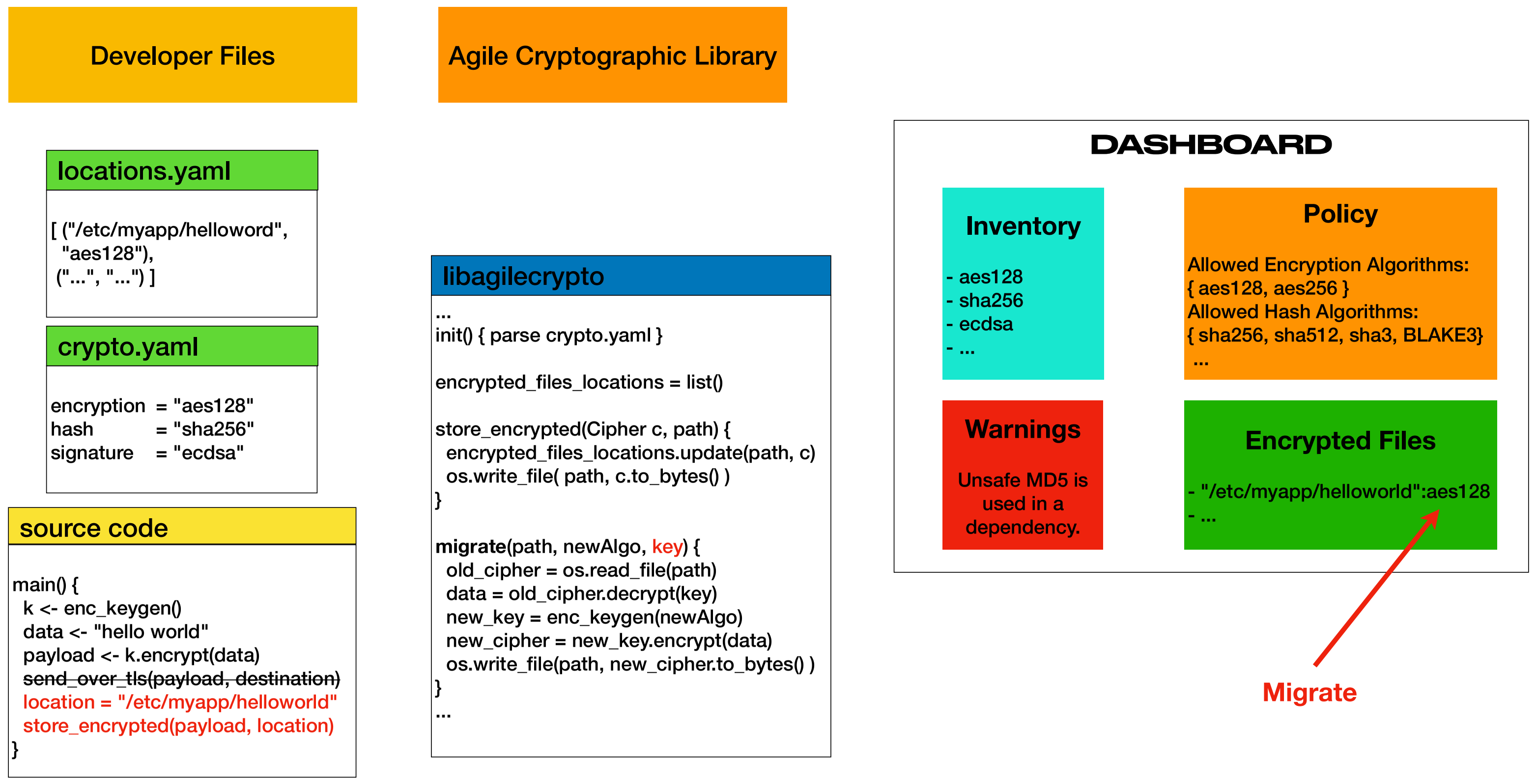
When updating an encryption algorithm which encrypted data is then stored on disk or in the cloud, it is best to migrate the encrypted data as well. For instance, in an instant messaging application storing messages encrypted using DES on the host, a migration from DES to AES would cause new encrypted messages to be stored in a different format that the old encrypted messages that have been stored in the past. This is undesirable because usually a migration occurs when an algorithm is considered as insecure, it should not be used anymore, even for data that was using it before, otherwise it becomes vulnerable. Keeping data encrypted using multiple algorithms after a migration cause the application to handle the decryption of two types of encrypted data, without knowing which message should be decrypted using which algorithm.

The middleware can be useful in this migration task. If the agile cryptographic framework also serves as key manager, and knows where the encrypted data is stored, it can then load the encrypted data, decrypt it using the decryption key, generate new keys for the replacement algorithm, encrypt the plaintext data using the new encryption key, and overwrite it at the location it was stored. In the case in which the middleware does not act as a key manager, an alternative would be to keep an inventory of cryptographic primitives, such as keys, Initialization Vectors etc. and when a migration is needed, the inventory is passed to the middleware which is able to perform the migration as described previously. It is important to note that it is impossible to migrate cryptographically hashed data from one algorithm to another, as cryptographic hash functions are one-way and thus cannot be reverted. We discuss how to address this challenge more in details in Subsection 3.6.2.

Figure 3.5 pictures a possible design of an agile framework supporting encrypted data migration. On the right hand side, there is a dashboard, which represent the interface for the Agile Cryptographic Library. In the *Developer Files*, for instance at the root of the project, there is the same `crypto.yaml` file as described before, and there is in addition, the location of the encrypted files that are stored on the disk. The source code of the sample application has been modified and is not sending encrypted data over the network, but storing it on disk. The `libagilecrypto` service exposes the `store_encrypted` function, which given a ciphertext structure, containing metadata such as the algorithm that was used for encryption, writes the sequence of bytes at the given location, and updates the file containing the location of encrypted files. This allows to keep track of all encrypted files that are stored on disk, with the algorithm that has been used for encryption. `libagilecrypto` also contains a `migrate` function, that will migrate the encrypted file at the given path to a new algorithm that is passed to the function. It is of course necessary to pass the key used for encryption to the function. We can imagine that the `libagilecrypto` also serves as a key manager and thus can provide the key. Using this function, a user can simply migrate encrypted files from the dashboard the graphical interface. This way, migration can be perform only by interacting with the middleware graphical interface, and without any modification to the source code. When the main program will interact with the data again, it will do so through the middleware, and will not notice the difference, even though the algorithm was changed.

3.4.7 Nature of the agile cryptographic framework

The cryptographic agile framework could be implemented in different ways while offering the interfaces described above. We will discuss the forms that the middleware could take, and describe the related trade-offs.



```
[ ("/etc/myapp/helloword",
  "aes128"),
 ("...", "...") ]
```

```
encryption  = "aes128"
hash        = "sha256"
signature   = "ecdsa"
```

```
main() {
  k <- enc_keygen()
  data <- "hello world"
  payload <- k.encrypt(data)
  ~~send_over_tls(payload, destination)~~
  location = "/etc/myapp/helloworld"
  store_encrypted(payload, location)
}
```

```
...
init() { parse crypto.yaml }

encrypted_files_locations = list()

store_encrypted(Cipher c, path) {
  encrypted_files_locations.update(path, c)
  os.write_file( path, c.to_bytes() )
}

migrate(path, newAlgo, key) {
  old_cipher = os.read_file(path)
  data = old_cipher.decrypt(key)
  new_key = enc_keygen(newAlgo)
  new_cipher = new_key.encrypt(data)
  os.write_file(path, new_cipher.to_bytes() )
}
...
```

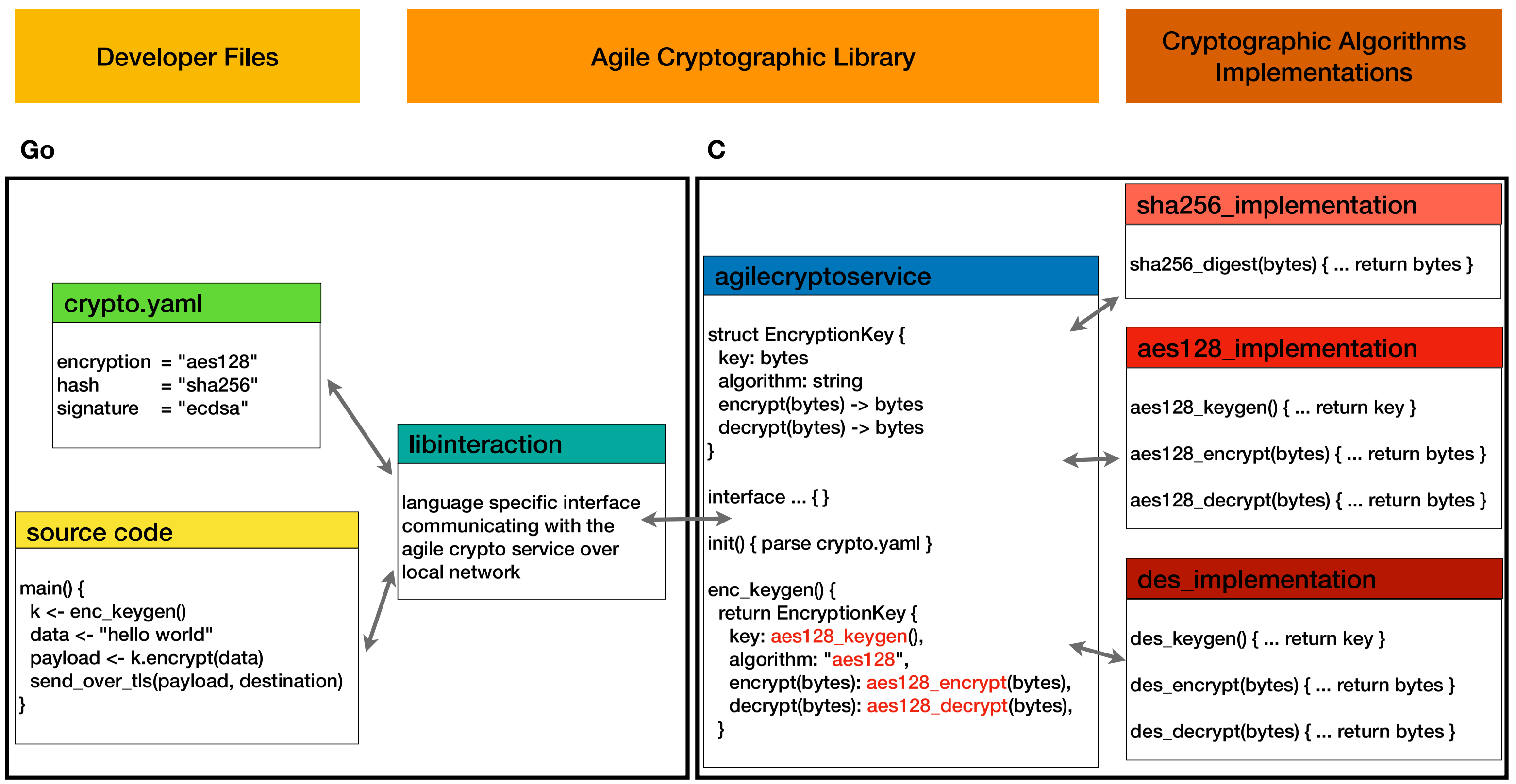
Figure 3.5: Example of an agile cryptographic library handling encrypted data migration.

The most intuitive to implement such an agile cryptographic framework, is to build it as a library. This library would act as a new layer of indirection, the source code would call the cryptographic functions exposed by the library instead of calling directly the implementations. The library will redirect the function calls to cryptographic algorithms from the source code to the actual algorithm implementation in an agile way, and return agile data structures.

An agile cryptographic library can be built as a static library or as an dynamic library. The difference between static and dynamic libraries, is that a project using a static dependency has to be compiled again if the library is to be updated, which is not to case for dynamic libraries. After a dynamic library gets updated, it only needs to be recompiled, and all binaries relying on it can then use the updated version. Moreover, when using a static library, the static library code get duplicated resulting in larger project binaries, whereas for dynamic libraries, the dependency is only linked so there is no code duplication. Hence, using dynamic libraries to build a cryptographic middleware would provide a more agile library, as it could be updated without having to recompile all code depending on the middleware.

The cryptographic framework could also be built as a service. This solution provide more agility than a library but also poses a number of challenges. A possibility for an agile cryptographic framework is to use Cryptography-as-a-Service (CaaS). This solution would mean to send requests over the network to a remote service provider to use cryptographic functions. However, CaaS is not optimized for a generalized use of cryptography, as some operations should be performed offline, for instance disk encryption operations. Thus CaaS would not be an appropriate candidate for an agile cryptographic service.

However, an agile cryptographic framework could be designed as a Microservice. An agile cryptographic microservice would run locally on a host, and communicate with process through the local network stack. A microservice could be updated without the programs using it noticing the update, as long as the interface stays the same. A microservice can have a single implementation, and one interface library for each programming language, thus centralizing the critical cryptographic operations in a single trusted implementation. Moreover, the same cryptographic algorithms implementations will be used independently of the language of the source code. An example of such an architecture is shown on Figure 3.6. In this example, the service is implemented in the C programming language, and it makes use of cryptographic algorithms implementations made in C. A Go language client making the bridge between the Go language libraries and the service is described as `libinteraction`. This library will be imported in all Go source code willing to make use of the cryptographic service, and the role of `libinteraction` is to expose the same interface as the service itself, but for Go applications, and to translate these requests to send them over the local


Developer Files
Agile Cryptographic Library
Cryptographic Algorithms Implementations
Go
C
crypto.yaml
source code
libinteraction
language specific interface communicating with the agile crypto service over local network
agilecryptoservice
sha256_implementation
aes128_implementation
des_implementation

```
encryption  = "aes128"
hash        = "sha256"
signature   = "ecdsa"
```

```
main() {
  k <- enc_keygen()
  data <- "hello world"
  payload <- k.encrypt(data)
  send_over_tls(payload, destination)
}
```

```
struct EncryptionKey {
  key: bytes
  algorithm: string
  encrypt(bytes) -> bytes
  decrypt(bytes) -> bytes
}

interface ... { }

init() { parse crypto.yaml }

enc_keygen() {
  return EncryptionKey {
    key: aes128_keygen(),
    algorithm: "aes128",
    encrypt(bytes): aes128_encrypt(bytes),
    decrypt(bytes): aes128_decrypt(bytes),
  }
```

```
sha256_digest(bytes) { ... return bytes }
```

```
aes128_keygen() { ... return key }

aes128_encrypt(bytes) { ... return bytes }

aes128_decrypt(bytes) { ... return bytes }
```

```
des_keygen() { ... return key }

des_encrypt(bytes) { ... return bytes }

des_decrypt(bytes) { ... return bytes }
```

Figure 3.6: Example of an agile cryptographic service.

network to the service. Porting the service to a new programming language would only require to build a client in this language to make the link between the language and the service. The service would be compatible and identical for all platforms and all languages, which makes it simple to use. The service could easily run a graphical interface for configuration allowing to configure the service in an interactive way. A service also offer the possibility to support add-ons, for instance to support cryptographic operations on hardware. However, a cryptographic service must make sure that operations are isolated, as all process would share the same microservice. To provide agility, each application using the microservice should have its own cryptographic policy, and thus the apps must authenticate at each request, to allow the service to handle the service correctly. Furthermore, if the service also acts as a key manager, it is critical that the keys can be accessed only by the owner process, and not by other processes. The authentication would add complexity to the microservice.

We will now compare a dynamic agile cryptographic library with an agile cryptographic microservice. The dynamic library would be faster to perform operations, as there is no data to serialize and send over the local network stack. The dynamic library does not need the applications to authenticate, which makes it even more optimized compared with a microservice. A microservice would use more resources as a dynamic library, as it must always be running on the host. However, the dynamic library needs to be built in multiple programming languages for a better accessibility, and each library has to use cryptographic algorithms implementations written in the same language, or to make bridges to other languages. A microservice architecture would allow to have a single implementation, making use of a single set of cryptographic algorithms implementations and making them accessible to multiple programming languages. It can also be customized with a graphical interface and add-ons, providing more agility compared with the dynamic library. Hence, a dynamic library is to be preferred for performance optimization, and a microservice is to be preferred for a better level of agility.

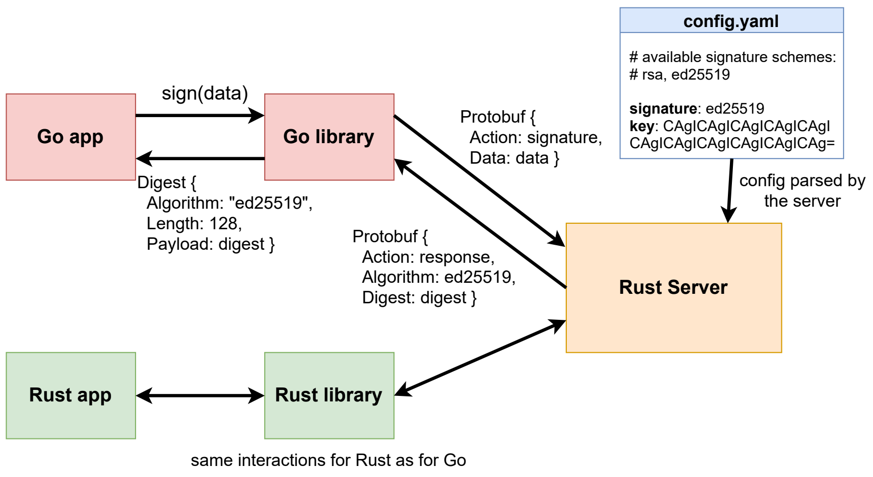
3.5 Proof of concept: Signature microservice

For the purpose of this project, we built a signature microservice serving as a simple proof of concept for larger cryptographic services. The goal of this service is to demonstrate that it is possible to build a simple microservice having bridges to multiple programming languages, convenient for use. In this case, we decided to use Rust and Go. We consider that the system as trusted: communications between client and server are not encrypted, and we consider that the server does not leak any information.

3.5.1 Design

This microservice is a very simple Rust application communicating with client over TCP. The microservice has a YAML configuration file, where the signature algorithm to be used is defined. There are only two signature schemes available for this simple proof of concept: rsa and ed25519. When starting, it will parse the YAML configuration file, and open a TCP socket. In the very simple client, an user can input a message to be signed by the service. The message will then be transmitted over TCP to the signing service. The signing service will get the signature request, compute the signature of the given string using the algorithm described in the configuration file and a key derived from a constant seed. It then sends the signature over TCP back to the client. The client can also request the service to verify a signature for a given string, for the same constant key. The service will return `true` if the signature verifies successfully and `false` otherwise.

Each simple client implementation is using a simple library interface which exposes the functions available for the microservice, and handles communication with the server, as depicted in Figure 3.7. Each interface library exposes data structures, such as generic signature digest here, and functions, such as a signature function, that will be usable by all projects in the same programming language. Hence, if we want to make the microservice compatible with more languages, we must build a library in this language serving as interface with the micro-service.



```
# available signature schemes:
# rsa, ed25519

signature: ed25519
key: CAgICAgICAgICAgICAgI
CAgICAgICAgICAgICAgICAg=
```

Figure 3.7: Signature microservice architecture

We built two different clients, one in Rust and one in Go. The clients communicate with the service over TCP using Google's Protocol Buffers format[4]. This format allows data structures to be exchanged between multiple

[4]`https://developers.google.com/protocol-buffers/`

programming languages, making the crypto-service agile. We decided to use this format, as it has a low data overhead and is available for a large number of programming languages.

3.5.2 Results

We were happy to see that this format of microservice was working as expected. The interactions between the client and server in multiple programming languages was successful and showed the agility that can be brought by this kind of service. We have seen that both Rust and Go clients could successfully manage different signature formats, according the type of signature performed by the server. Thus, it is possible to create a generic `digest` structure that is able to contain a signature digest of variable length, assessing the agility of this microservice.

However, it made us realize the complexity of a more complete agile cryptographic framework exposing more primitives. Key management is another challenge that needs to be addressed. To simplify the question, we let the server manage the keys, but in a real world application we either need a different server process for each running application, or we need to authenticate each application using the service, to make sure keys can be accessed only by the right application. Otherwise, the library interface could generate the key pairs, and send them over a channel with confidentiality and integrity to the service. Then the key pairs could be handled by the application or by the library interface instance.

We decided not to go further in this direction, as it would take a lot of efforts to produce a tool that would be usable in practice.

3.6 Cryptographic Agility Strategy

3.6.1 Migration to an agile framework

The switch from our current software implementations to new cryptographic agile frameworks requires a consequent migration. Indeed, to avoid the huge manual migration to post-quantum cryptography, a huge manual migration to a cryptographic agile framework is necessary. This means that all legacy software must be updated. Once a cryptographic agile framework is available, all function calls to cryptographic algorithms should be replaced with function calls to the corresponding functions from the agile framework. This migration would be hard to automate, as the replacement functions depend on the context in which the old function were used. This also implies that there is a risk that the migration is not done correctly, and this may introduce vulnerabilities in the source code. On the other hand, the migration might correct some misuses of cryptographic functions.

However, this migration can be facilitated. As updating manually each function call is a tedious task, some software could highlights the parts of the code that have to be updated. A cryptographic inventory tool could perform this work. It would scan all the dependencies of a given project, and return all of the function calls to cryptographic algorithms. A cryptographic inventory would be very helpful, as it is able to scan the dependencies' dependencies, and this recursively. This allows developer to filter their dependencies using not agile cryptography, and to replace them by agile libraries. As a software relies on its dependencies, it can be fully agile, only if all of its dependencies are fully agile too. We can imagine that an automated tool can suggest a replacement for cryptographic function calls, but the change has to be validated by a developer.

The first step to get to cryptographic agility, is to choose a cryptographic policy that will be applied on the targeted software. One cryptographic agile framework should be selected, before starting the migration process. Once this is done, the next task is to make an inventory of the all cryptographic primitives used by a software, using for instance a cryptographic inventory tool. Once identified, they should be replaced by function calls to the agile framework, selected beforehand. All libraries using non agile cryptographic implementations should be replaced too. Then, the cryptographic policy should be given as input to the cryptographic framework, and the source code migration is complete. The software will be able to benefit from the cryptographic agility offered by the framework.

3.6.2 Data migration

We discuss in this section the consequences of switching to a cryptographic agile framework from a legacy code, and changing the cryptographic algorithms in use. Hence, this change of algorithm will require a migration of data. However, it is also possible to migrate to an agile framework, while keeping the same algorithms used before the migration.

Encrypted data migration

In parallel of the source code migration to a cryptographic agile framework, it is necessary to migrate the stored data encrypted with the old scheme. This migration is necessary, as the new cryptographic framework will not be able to decrypt and make use of the already encrypted data. For this purpose, we can imagine that an inventory tool will be able to list all encrypted data, linked with a given application, and list the associated algorithms and key locations. Then, a tool taking as input (1) the data location, (2) the cryptographic algorithm and parameters and (3) the secret key would be able to decrypt the data, and re-encrypt it using the agile framework. Note that this tool would ideally be integrated in the framework.

Hashed data migration

The migration of hashed data is unfortunately not this simple. For example, most password based authentication systems hash passwords to avoid storing them in clear text, and store only the hash digest and the associated salt. However, hashing cannot be reversed by design, because there are resistant against preimage attacks. Consequently, it is impossible to reverse the hash operation to recover a password, and to hash it again using a different algorithm. There are two possibilities for the migration of hashed data.

1. **Password onion**: This solution, reported to by used by Facebook [8] consists in having multiple *layers* of hash functions. For example, Facebook's password onion is represented in Algorithm 1. The layers consist in a sequential combination of hash functions. We can see that Facebook used to hash its passwords using MD5, then they migrated to SHA-1, and they needed to generate a salt. Thus the digest was the result of the HMAC using SHA-1, with as input the old MD5 digest, and a random salt. We can see that since then, they added a PRF, and use scrypt to hash the existing digest, and truncate it with HMAC to have a decently sized digest.

Algorithm 1: Facebook Password Onion

```
1 $h_1 \leftarrow \mathrm{MD5}(pw)$
2 $sa \leftarrow^{\$} \{0,1\}^{160}$
3 $h_2 \leftarrow \mathrm{HMAC[SHA\text{-}1]}(h_1, sa)$
4 $h_3 \leftarrow \mathrm{PRF\text{-}Cl}(h_2) = \mathrm{HMAC[SHA\text{-}256]}(h_2, msk)$
5 $h_4 \leftarrow \mathrm{scrypt}(h_3, sa)$
6 $h_5 \leftarrow \mathrm{HMAC[SHA\text{-}256]}(h_4)$
7 Return $(sa, h_5)$
```

The advantage of such a scheme, is that the migration can be done instantly, using the old digest, and it is somehow flexible, as it will always be possible to add new layers to the onion. However, when an user authenticates, the back-end has to go through multiple hash function, to verify if the password is correct, which introduce complexity, compared with a single hash function.

2. **Require client login**: This alternative is simpler, but it requires an action from the users. Once the user authenticate using their password, the service hashes the given password using the old hashing algorithm and the associated salt, and compares the digest with the stored one. If they match, then the service will be able to hash the correct password using another hashing algorithm and a freshly generated salt. The result will replace the user's entry in the password database.

The simplicity of this method makes it elegant, but it requires all users to enter their password to migrate successfully, which induce the following drawbacks. (1) All users have to login, this implies that inactive accounts will not be migrated. The accounts can be removed after a given period to keep the password database consistent, but this might be an undesired side effect. (2) The migration takes time. The migration time depends on the time taken by all users to perform the login operation, which could take months. (3) The password database need to support multiple formats. As the updates in the password database is a long process, both old and new entries will coexist for a while, and the system should be able to handle old and new authentications differently, which adds complexity.

Shared secrets migration

When a cryptographic algorithm used to communicate over the network with a remote host is replaced by another one, shared secrets that were used to communicate must be migrated too. One solution can be to get a fresh shared secret, by using a key exchange protocol to derive an appropriate shared secret. Another solution can be to give the old secret as input to a Pseudo Random Function (PRF) and then to a Key Derivation Function to derive a new shared secret for the new algorithm.

3.6.3 Cryptographic Agility in CI/CD

When the source code and data have been migrated, it would be good to add cryptographic agile tools in Continuous Integration/Continuous Deployment (CI/CD) pipelines. These tools could perform static source code analysis to assess that the code is only using agile cryptography. This would prevent deploying any code that is not cryptographically agile. The CI/CD tool could be integrated to the cryptographic framework, and provide policy compliance certifications and risk assessment before the code is deployed.

Chapter 4

Cryptographic Inventory

4.1 Definition

A cryptographic inventory consists in a listing of all cryptographic algorithms and primitives used by a software, a system or a company. Its representations can go from a simple text list of cryptographic algorithms to a complete dashboard giving advice on how to improve the security of a system. Ideally, a cryptographic inventory of a software would include the inventory of the software's dependencies, in a recursive way. There are multiple ways to build a cryptographic inventory. Some companies decide to manually build their cryptographic inventory, by reviewing their source code, and listing every cryptographic primitive in use. There exists automated scanning tools analyzing source code and binaries and producing a complete cryptographic inventory.

4.2 Need for Cryptographic Inventory

We discuss in this section the problems that are addressed by cryptographic inventories. We first focus on current issues, before the migration to post-quantum cryptography and cryptographic agility. We will then focus on why a cryptographic inventory is useful, even after we gain cryptographic agility.

4.2.1 Before Agile Cryptography

A cryptographic inventory can currently help addressing the following challenges:

1. **Compliance**: As discussed before, multiple countries introduced diverse regulation on data security and privacy, and all company must comply to these regulations. These regulations may specify which

algorithms to use for specific actions, and may forbid the use of certain algorithms, considered as unsafe. Moreover, multiple companies introduced an internal cryptographic policy, as discussed in Section 2.3. These policies are often more restrictive than the regulations, and developers must make sure that their code is compliant with these policies. A cryptographic inventory is helpful to prove compliance with a policy or regulation. It can be shown to internal and external auditors for reviews. Ideally, the inventory would be made with a certified tool, missing no use of cryptography and signing the produced inventory. This kind of inventory would irrefutably certify the compliance of a piece of software.

2. **Risk assessment**: Cryptographic vulnerabilities or weaknesses can be revealed by the inventory as it displays all uses of cryptography. A Chief Information Security Officer (CISO) can use it to make sure that no deprecated libraries or algorithms are used throughout a project, without having to review all of the code manually. They can also verify that cryptographic algorithms are used correctly, as the inventory can check whether a nonce have been used twice, or whether a secret key was really randomly generated.

   Moreover, when an algorithm or implementation is found to be vulnerable, a CISO can immediately verify if and where it is used in some of the company software. They can even check how it has been used, and accordingly plan a way to apply a security patch or use a secure alternative.

3. **Ease of migration**: When a cryptographic algorithm is deprecated or when a better technology should be used, a migration is necessary. A cryptographic inventory can help engineers in a migration by showing them which cryptographic algorithms need to be updated, where and how they are used. The inventory can show to a developer each occurrence of function calls that need to be changed, so that the developer do not miss any.

   The cryptographic inventory is a way to facilitate the migration to an agile cryptographic framework, which in turn will allow a seamless migration to post-quantum cryptographic schemes. The ultimate goal of cryptographic inventory is to provide cryptographic agility, meaning that one can change the cryptographic algorithms used by a system from an inventory dashboard.

4. **Primitives listing**: A cryptographic inventory also offer the capability to list all keys, certificates, passwords, salts etc. that are used by a system. It is able to retrieve the location where these components are stored, for instance on disk or in a cloud. This is useful for

system administrators to manage certificates expiration and renewals. Using such a tool, it would also be possible to verify if some passwords, whose hash digest is stored, are exposed in a known password cracking dictionary. This can show if a key is used multiple times, and can help developers to determine if the multiple use of a single key is appropriate.

4.2.2 After Agile Cryptography

After an agile cryptographic framework is in use, the cryptographic inventory tool could be merged with the framework itself. As a cryptographic policy will be given to the cryptographic framework as input, describing how cryptography should be used, the inventory will mostly be included in the policy. However, the inventory can ensure that cryptography is used correctly, using the agile framework and that a source code contains no static function call to a specific cryptographic algorithm implementation. It could also be used to prove that a software complies with local data security regulations, or list all the primitives that are used by the framework. Hence, adding the inventory tool to the cryptographic agile framework definitely makes sense.

4.3 Inventory building techniques

There are multiple techniques to build a cryptographic inventory. (1) Manual inventory: developers have to go through the source code, and list all cryptography in use. This task is time consuming, tedious and inaccurate, as developers could miss a cryptographic function. (2) Static source code analysis: an automated tool will scan the source code to list all cryptography in use by a software. This method is automated and fast to operate, and exhaustive. However, it will detect cryptographic functions that may never be used in practice, as it scans all possible execution paths. For instance, many cryptographic libraries include insecure algorithms such as MD5 or DES for legacy reasons, but the static analyzer will be unable to determine if these functions are actually used, and they will be flagged. Moreover, it is a challenging task to check if cryptographic algorithms were used correctly, using a static code analyzer (3) Run-time analysis: this method consists in tracing run-time execution to detect which functions are called. It detects all cryptography that is actually in use, but requires the software to run to be able to perform a scan. However, as different executions may differ, different cryptography could be used depending on the context, and performing symbolic execution to test all execution paths is not always possible, and would essentially give the same result as a static analysis. However, run-time analysis can be combined with fuzzing to get a good cryptography coverage.

4.4 Existing Cryptographic Inventories

4.4.1 Cryptosense Analyzer

Cryptosense has a cryptographic inventory solution that they sell as a service, *Cryptosense Analyzer*. This inventory solution is consequently closed source, and the only information we have is from their whitepaper [6]. This whitepaper is mostly business oriented, explaining what is a cryptographic inventory, and why it would be useful. It states that their inventory service combines both static and run-time analysis approaches. The Cryptosense Analyzer has coverage for Java JCE, .NET cryptography, PKCS#11 and OpenSSL and provides an inventory of algorithms, key-lengths, libraries, modes of operations, passwords, parameters, vulnerability analysis and cryptographic policy enforcement. It can be integrated to Continuous Integration (CI) pipelines thanks to its APIs and plugins for Maven, Gradle and Jenkins.

4.4.2 InfoSec Global AgileScan

InfoSec Global also developed an automated cryptographic inventory solution, *AgileScan*, which is sold as a service. Thus, it is closed source, and we were not able to analyze how the inventory is actually being built. They published a whitepaper [10], explaining mostly why a cryptographic inventory is necessary for companies, and describing very briefly the technologies they are using. AgileScan has a list of known cryptographic libraries and will detect their use, and determine if the version of the library is up-to-date, or if it is outdated and contains security flaws. But it is not mentioned how they detect the use of a library. They also inspect the executed binaries to determine which cryptographic algorithm is used. AgileScan also includes a common configuration fault detection mechanism, but no precise description is given.

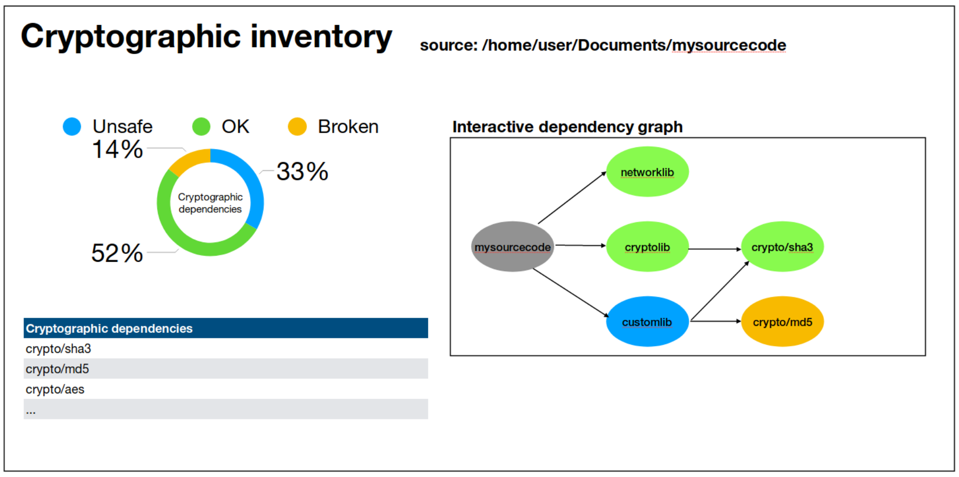
4.5 Inventory interface

In this section, the terms *package*, *library* and *functions* are used. A *package* is the Go term for a *library*, which is an implementation that can be imported and used by other programs. Thus both terms are equivalent in this section. A *library* contains *functions*, that are individual programming functions implementations that can be called through the *library* interface. Each *function* belongs to a specific *library*. Dependencies can be seen as *libraries* or *functions*. For example, we can say that $library_A$ depends on $library_B$ and $library_C$, or that $function_1$ depends on $function_2$ and $function_3$. But, in the same setting, we can also say that $function_1$ depends on $library_B$

and $library_C$ if $function_2$ belongs to $library_B$ and $function_3$ belongs to $library_C$.

4.5.1 Dashboard

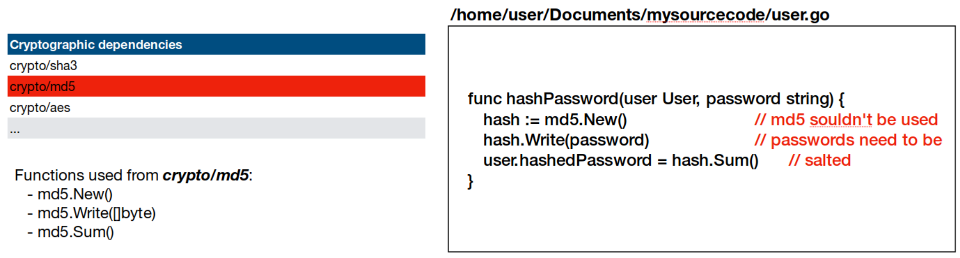
Once the inventory is built, either by static code analysis or by run-time analysis, it has to be displayed in a human readable way to be useful. Ideally, a cryptographic inventory would have an interactive dashboard interface, allowing developers to better assess risk, visualize dependency tree, certify compliance etc. The dashboard should show insights for each library, for instance which functions are publicly exposed, which functions are used by another program, what are the dependencies for each function. The dashboard would also contain a text editor, allowing the developer to modify the source code from within the inventory dashboard. The dependency trees should be displayed in a human readable way, as they often appear to be very complex. A solution for this is to make the diagrams interactive, to display only a part of the dependency tree, and allow the user to navigate through the tree. We describe the formats of the tree diagrams below. Potential features of the dashboard will be displayed, as it should be scrollable, features will be shown in different figures.



Cryptographic dependencies
crypto/sha3
crypto/md5
crypto/aes
...

Figure 4.1: Potential cryptographic inventory dashboard look

Figure 4.1 represent how the inventory could look like, in a simplified way. As the dashboard would contain a lot of features, not all of them can fit on screen at the same time. Figure 4.1 includes a simple risk assessment chart, listing how many function calls to cryptographic function are considered as `Broken`, `Unsafe` or `OK` according to a categorization given as input to the inventory. It also include a list of all cryptographic libraries on which the scanned software relies directly or indirectly. The list contains search and filter options, for instance to show only unsafe cryptographic algorithms, or post-quantum algorithms in use. On the right, the interactive dependency graph would show the partial package or function dependency tree. As it is interactive, the user can navigate through the graph by clicking on the nodes they are interested in, to explore their dependencies. Furthermore, the search or filter function from the cryptographic dependencies list can be use to filter the nodes appearing on the interactive dependency graph, for instance to display only the packages that rely on a specific cryptographic library.



Cryptographic dependencies
crypto/sha3
crypto/md5
crypto/aes
...

Functions used from ***crypto/md5***:
- md5.New()
- md5.Write([]byte)
- md5.Sum()

/home/user/Documents/mysourcecode/user.go

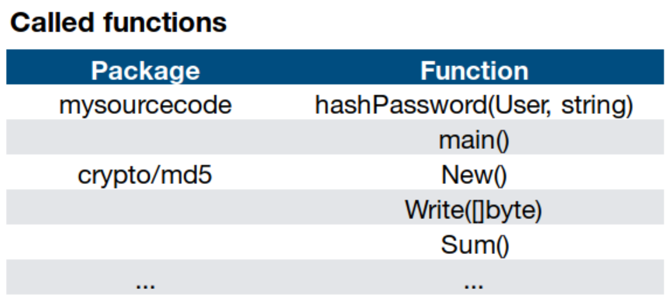
```
func hashPassword(user User, password string) {
   hash := md5.New()                    // md5 souldn't be used
   hash.Write(password)                 // passwords need to be
   user.hashedPassword = hash.Sum()     // salted
}
```

Figure 4.2: Cryptographic dependencies list along with text editor providing insights on the use of cryptographic algorithms

Figure 4.2 includes the same cryptographic dependencies list as shown in Figure 4.1. One entry is selected in the list, and insights are shown for this package. Below this list, are printed all the functions of the package that are used by the project or one of its dependencies. The right hand side contains a code snippet where one of the functions from the package is used. The cryptographic inventory is able to scan the source code, and add insightful comments on the usage of cryptographic algorithms. For instance, in this screenshot, the program advises against using the MD5 algorithm to hash a password, and recommend to use a salt for password hashing. The developer can then directly edit the source code in the text window, on the right.

Figure 4.3 represents the full package and function inventory of the scanned project. The left column contains the list of all packages on which the project is relying, and the right column contains all the functions that can possibly be called by the software, ordered by package. Of course this list also supports search and filter capabilities, and can interact with the other components of the dashboard, such as the interactive dependency graph from Figure 4.1, or the text editing interface from Figure 4.2.

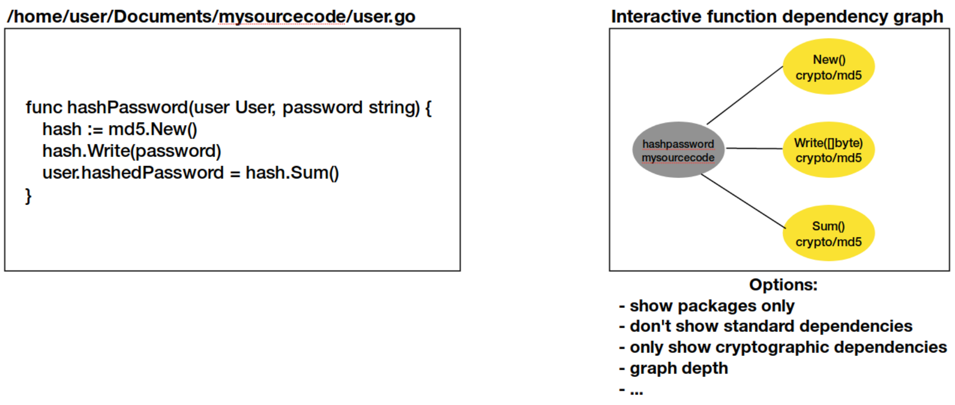
Figure 4.4 has a text editor on the left hand side, and has the associated function's dependency graph on the right hand side. The dependency graph is interactive, and has filter options, to print only the selected nodes, to keep it understandable. It is for example possible to choose the depth of the graph, or hide the standard package dependencies.



Called functions

Package	Function
mysourcecode	hashPassword(User, string)
	main()
crypto/md5	New()
	Write([]byte)
	Sum()
...	...

Figure 4.3: List of all functions on which the target program is depending on, ordered by package



/home/user/Documents/mysourcecode/user.go

```
func hashPassword(user User, password string) {
   hash := md5.New()
   hash.Write(password)
   user.hashedPassword = hash.Sum()
}
```

Interactive function dependency graph
hashpassword mysourcecode
New() crypto/md5
Write([]byte) crypto/md5
Sum() crypto/md5

Options:
- show packages only
- don't show standard dependencies
- only show cryptographic dependencies
- graph depth
- ...

Figure 4.4: Interactive function dependency graph for a code snippet

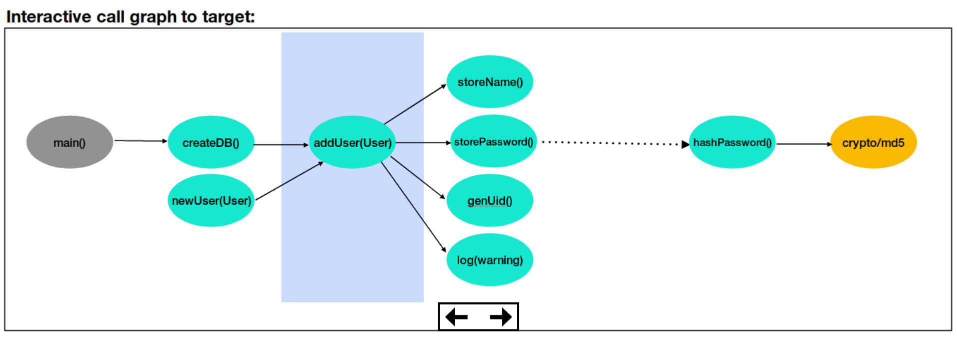


Figure 4.5: Potential cryptographic inventory dashboard look

Figure 4.5 represents an interactive call graph browser. It is a tool to navigate a large dependency graph, choosing the origin and destination packages or functions. For example, in Figure 4.5 the origin function is the `main()` function from the scanned source code and the destination package is `crypto/md5`. The paths from the origin to the destination are printed in the window. It is possible to move the focus zone, on blue on the Figure to display all functions or packages that depend on the selected node, or these on which the selected node depends on. This feature is useful to understand which part of the source code is making an indirect use of specific functions or packages.

4.5.2 Output format

Ideally, the inventory should be exportable in a text format, JSON for example, so that it could be used by other software analyzing the content of the inventory. The list of all dependencies of a program can be called a Software Bill of Material (SBoM), describing all the software components. A SBoM is a common way to list a software dependency, although there are multiple ways to list the dependencies. In cryptography, the list of all cryptographic functions that are used by a software can also be called a Cryptographic Bill of Material (CBoM). The Software Package Data Exchange (SPDX)[1] is a special file format introduced by the Linux Foundation, and offers the possibility to represent SBoMs. SPDX was initially built to document information on software licenses and copyrights under which a software is distributed, but now also includes a possibility to describe dependencies. It is an attempt to standardize source code metadata. The upside of using SPDX to export a cryptographic inventory, is that this format aims to be a standard, and is likely to be compatible with many applications in the future. However the downside is that this very complex format was not initially built to list software dependencies, and it adds complexity to a very simple data structure to be exported. Thus, an export in the JSON format would be simpler.

4.5.3 Diagrams

The inventory can also contain a graphical visualization, which is more human friendly than a large text file. In order to represent dependencies it is possible to use a directed acyclic graph picturing the relationships of the different libraries as depicted in Figure 4.6. This diagram was drawn using Graphviz. However, when the graph is getting more complex, the result is less human readable. Figure 4.7 is still quite readable, but the full dependency graph of complete projects making use of many libraries make PDF readers crash when trying to open the file. So, plotting dependency

[1] `https://spdx.dev/`

graphs this way is good for simple packages or for a partial dependency graph of a limited depth, but cannot be used for large projects. It is also possible to build interactive graphs, containing all the required dependencies, but showing only subgraphs that are easily understandable, the graphs are interactive as the visible dependencies change when a click on a node is performed. These graphs are not displayed, as this report is unfortunately not interactive.

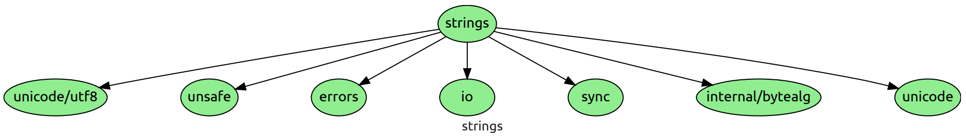


Figure 4.6: Direct dependencies of the Go `strings` package

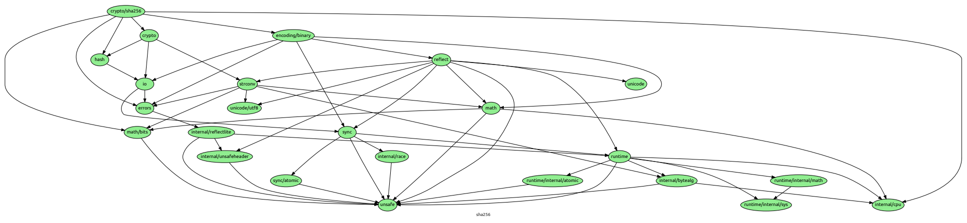


Figure 4.7: All dependencies of the Go `crypto/sha256` package

Another way to graphically represent dependency relationships, is to use chord diagrams, as depicted in Figures 4.8 and 4.9, representing the dependencies of the Go package `strings`. In the chord diagrams we use, each package is depicted as a portion of a circle and is given a specific color. The links of this color represent the package's dependencies. For instance, in Figure 4.9, the package the package `io` is a dependency of the package `strings`, and `sync` and `errors` are dependencies of the package `io`. These graphs are interactive HTML graphs, hovering the mouse over a specific link or package will hide unrelated links to highlight the selected content. Figure 4.9 is the focused version of Figure 4.8, with focus on the `io` package. This allows to have a good data visualization of the dependencies even though it is less hierarchical than Figures 4.6 and 4.7. For simple packages with a small number of dependencies, such as the Go package `strings`, a hierarchical diagram such as Figure 4.6 gives a better overview compared with the chord diagram represented in Figure 4.8.

However, for packages with many dependencies, the chord diagram is a better data representation. PDF readers are not able to load the dependency graph of the Go package `github.com/hyperledger/fabric/bccsp/sw`, so no visualization is possible. Figure 5.7 and 5.8 show the chord dependency diagram of the Go package `github.com/hyperledger/fabric/bccsp/sw`.

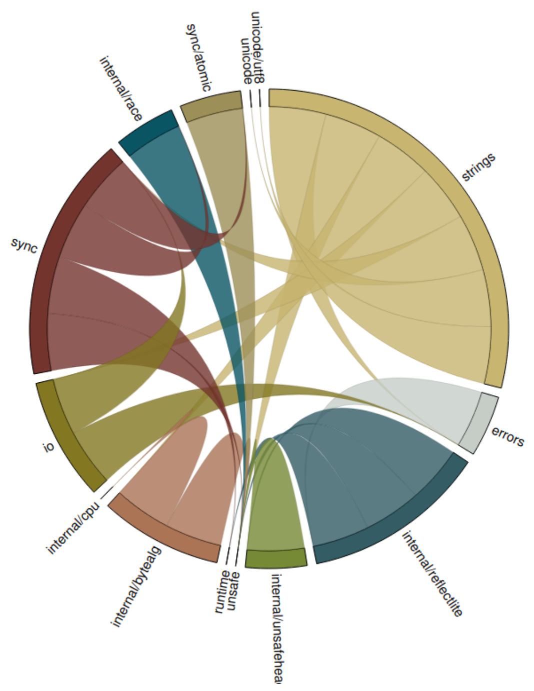


Figure 4.8: Chord diagram of the Go `strings` package

It is true that Figure 5.7 is not very insightful, as there are too many links between the libraries, however, as the library is interactive, it is possible to see each package's direct dependencies. In the focused version of the diagram, Figure 5.8, we can see all the dependencies from the Go package `github.com/hyperledger/fabric/bccsp/sw`.

There is a good Javascript library proposing a lot of diagram templates, `https://d3js.org`. We found that the chord diagram was the most insightful for dependency representations, so we did not plot dependencies using other types of diagrams.

4.6 Cryptographic Inventory in CI/CD

A cryptographic inventory tool would be very useful in CI/CD pipelines, in order to ensure compliance with a cryptographic policy, check the presence of any weak cryptographic algorithm potentially in use, verify if any secret is hard coded and pushed to a remote platform. All these checks would occur before the deployment of the software, thus mitigating the introduction of undesirable cryptography. As the cryptographic inventory provides a list of all the dependencies that can be used by a program, a company could define a white list of authorized libraries, that are compliant with the security standards of the company. They would also be able to define a black list

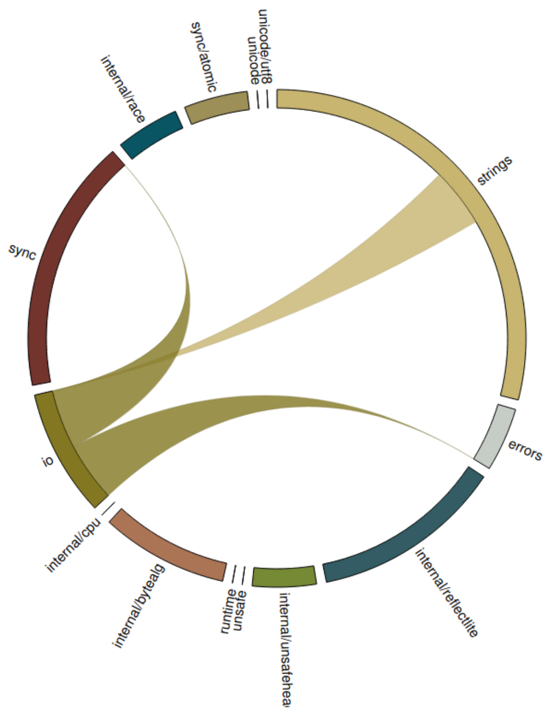


Figure 4.9: Chord diagram of the Go `strings` package, focusing on the `io` package, showing its dependencies and the package depending on it.

of cryptographic algorithms that must never be used, for instance because they are deprecated. Once that an agile cryptographic framework is in use, the inventory tool will be able to detect any non-agile use of cryptography, and flag it to keep only agile cryptography in the code base.

Chapter 5

Implementation

As building a full cryptographic agile framework is out of the scope of this project, I decided to implement a cryptographic inventory tool, as it will be required for the migration to post-quantum cryptography and for the migration to a cryptographic agile framework as explained in Chapter 3. I chose to implement an inventory tool for the Go programming language, with the possibility to focus cryptographic libraries. I picked Go, as this language has a well designed package manager, convenient to resolve and analyze dependencies. Moreover, it is one of the programming languages I know best. The tool is using static source code analysis to lists all of the dependencies in an exhaustive way. Hence, it might find cryptographic libraries that will never be called in practice, but if a library is never called, then it is not supposed to be part of the source code. A run-time analysis tool would be more accurate on the libraries that are actually in use, but may miss libraries used only in a specific context. Furthermore, except for backward compatibility issues, there is no reason to keep functions that are never called in a source code.

5.1 Goals

The goal of this inventory tool, is to provide a Software Bill of Material (SBoM), listing all the dependencies of a project recursively. It should then provide the dependency tree in a human readable way, so that it can be used by developers in practice. This tool should help developers to visualize all the packages on which their code is relying. This could prevent an unconscious use of a weak cryptographic algorithm in one of the dependencies. One of the goals is to provide a precise inventory of all cryptographic functions which could be called by the program. It can also be used for risk evaluation. When taking as input a list of known cryptographic libraries with a security grade, the inventory can highlight the use of cryptographic algorithms, along with the security level they offer to the project.

IBM is interested in making an inventory tool, as having a cryptographic inventory is the first step to cryptographic agility. This tool is interesting as a proof of concept, or internal Go code dependency analysis. It could be used to develop a better cryptographic inventory tool supporting more programming languages, or to include an inventory capability in an agile cryptographic framework. IBM currently has customers willing to build their own cryptographic inventory, mostly for risk assessment and regulation compliance. However, some other customers already anticipate the post-quantum cryptography migration and would like to take the necessary steps to be ready for the big migration.

The goal of the project is not to build a production ready tool with a nice interface, as I don't have enough knowledge in front end programming to make a nice interface. I prefer to focus on the functional side. As an ideal cryptographic would include a full interactive graphical dashboard, this project contains the back end mechanisms required to build such tool. This project only contains a minimal interface, including all described features, but is not very convenient to use. Moreover, the tool cannot be included in a CI/CD pipeline, as it requires extra work to adapt it for this usage. Thus the goal of the tool is not to be production ready, but to serve as a proof of concept, base or inspiration for future full inventories tools.

5.2 Design

The Go inventory tool was built making use of the Go package management mechanisms. Go packages easily can be imported as dependencies to other Go project. Packages are uniquely identified by their *path*. There are three types of package imports: standard packages, remote packages and local packages. Standard packages are the included when Go is set up on a host, they are stored on disk and resolved automatically by the package manager. A standard package *path* is the package's name, sometimes it is a path as it is a subpackage i.e `strings`, `net/http` etc. Remote packages are Go source code, published online on platforms such as Github or Gitlab repositories. The *path* of a remote package is the url where the package is accessible, without the `https://`, for instance `github.com/guillaumemichel/passtor`. These package resolution requires the Go package manager to query the remote repository to fetch the package source code. Local packages can be imported to projects, if they are stored at the correct location on disk, in the `$GOPATH`. The *path* of a local package can be anything, but if the package is aimed to be published, it is good to pick a valid path for remote packages. This allows developers to use proprietary libraries, that are not publicly available.

As remote repositories can automatically be fetched by the package manager, it is easy to get the source code of remote dependencies to analyze them. This would not be possible for other languages such as C/C++ as it is harder to retrieve remote dependencies, and some of them are already compiled so a static source code analysis would not be possible. Based on these capabilities, static code analysis tools have been developed for Go and provide interesting features. For example the package `golang.org/x/tools/go/loader` allows to load a package's basic information, such as its functions, imports, variables etc., and gives access the source code.

Using the `golang.org/x/tools/go/callgraph` package we are able to retrieve the call graph for target function or package. A call graph is the graph of the dependencies of a function, and their own dependencies recursively, as depicted in Figure 5.1. This figure contains the call graph from the function `Sum256` from the `crypto/sha256` Go package. An arrow represent a function call from the source of the arrow to its destination, and thus a dependency. For instance, `Sum256` from package `crypto/sha256` makes function calls to `Reset`, `Write` and `checkSum` from the package `crypto/sha256`. It indirectly depends on the `block` function from package `crypto/sha256`, as it is a dependency of the `Write` function, as well as on `PutUint64` and `PutUint32` functions from the `encoding/binary` package. Thus the function `Sum256` from package `crypto/sha256` depends on 6 functions from two packages, its own package and `encoding/binary`.

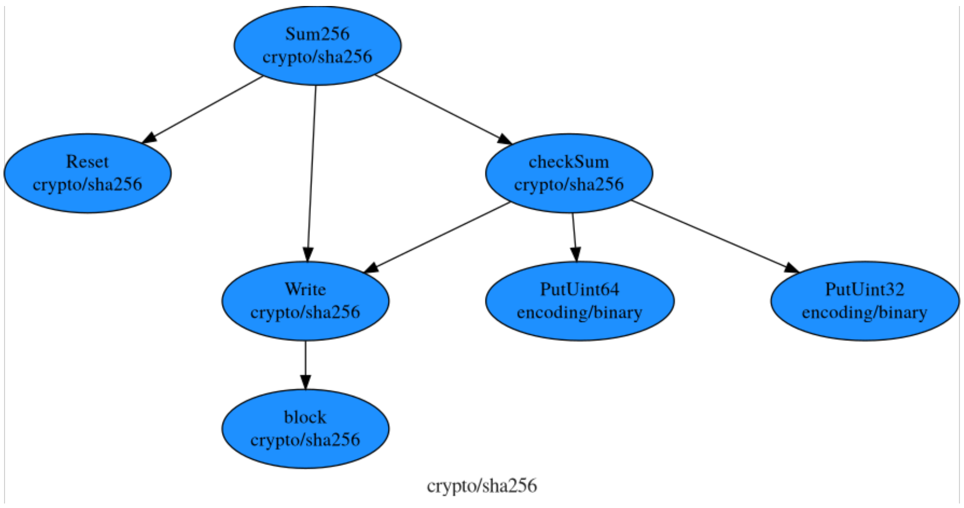


Figure 5.1: Call graph of the `Sum256` function from the Go package `crypto/sha256`

The call graph only includes the dependencies of the target function or package, and not the complete dependency set of the target's indirect dependencies. We will illustrate this by an example. We define the root of the call tree, as the $main()$ function from $package_A$. $package_A$ only contains a single function: $main()$. $package_B$ contains two functions: $function_1()$ and $function_2()$. $package_C$ and $package_D$ contain a single function, respectively $function_3()$ and $function_4()$. $main()$ from $package_A$, depends on $function_1()$ from $package_B$. $function_1()$ depends on $function_3()$ and $function_2()$ depends on $function_4()$. Thus, the dependency graph of $package_A$ only contains $package_B$ and $package_C$ but not on $package_D$ even though it is a dependency of $package_B$, itself a dependency the root package. However, if the root of the dependency tree was $package_B$, both $package_C$ and $package_D$ would be included in the dependency graph.

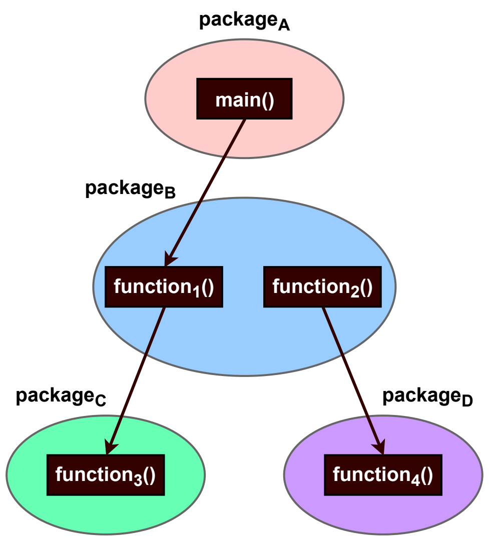


Figure 5.2: Dependency graph example

Taking the information from the obtained call graph, we build our own data structures for the function and package graphs. Elements in the package dependency graph are defined as shown in Listing 5.1. Each package is uniquely defined by its `Path`, and has a `Name`, which is the last part of the `Path`. Each package has links to all of its functions, here defined of a map, mapping the function name to the Function structure, described in Listing 5.2. The `DependingOn` variable is the set of all the direct dependencies of the described element. These dependencies are the imported packages. The `DependenceOf` variable is the set of all packages that directly import the described element. This means that if $package_A$ depends on $package_B$, $package_A$ will be included in $package_B$'s `DependenceOf`, and $package_B$ will be included in $package_A$'s `DependingOn`. The overall structure is a double linked acyclic graph, which is similar to a directed acyclic graph (DAG) except that the pointers are two-ways. This feature allows to build the dependency tree of a specific package, but also allows to build the reverse dependency tree of all packages depending on a chosen dependency. This can be used in particular to verify the dependency chain on deprecated cryptographic functions, in order to understand where and how they may be used.

```go
// Package structure, element of the dependency tree. A
// package has pointers to all of its function that are
// used in the calltree, and to the packages it depends
// on, and is a dependence of. A package is uniquely
// identified by its path
type Package struct {
  Name          string
  Path          string
  Functions     map[string]*Function
  DependingOn   map[string]*Package
  DependenceOf  map[string]*Package
}
```

Listing 5.1: Function element structure

An element in the function graph is defined as shown in Listing 5.2. All functions are uniquely identified by the combination of its name, and by the package it belongs to. Each element contains a set `DependingOn` of all the functions it directly depends on, and a set `DependenceOf` of all functions directly depending on itself. These sets are similar to the equivalent on the package structure. The Function elements also have an `ID()` method, returning the unique function identifier defined with the concatenation of the unique package path, a ":" and the function name. For instance, the unique identifier for `function1()` from `packageA` is `packageA:function1`.

```go
// Function structure, element of the callgraph tree.
// A function has pointers to the package it belongs
// to, to the packages it depend on, and to the
// packages of which it is a dependence. A function
// is uniquely identified by its id() function, which
// is a string composition of its Name, and of the
// path of the package it belongs to.
type Function struct {
  Name          string
  Package       *Package
  DependingOn   map[string]*Function
  DependenceOf  map[string]*Function
}

// ID return a function's unique identifier, package path +
// function name
func (f *Function) ID() string {
  return f.Package.Path + ":" + f.Name
}
```

Listing 5.2: Function element structure

The most convenient form of an inventory tool would be a dashboard graphical interface, as described in Section 4.5.1. However, as I unfortunately don't have experience with web programming and graphical interfaces, I decided to opt for a Terminal User Interface (TUI). I could have learn how to build a dashboard, but I found more interesting to invest my time on agile cryptography rather than on web programming, which is not directly related to the topic of this project and to my major. The interactivity of the interface was necessary to be able to navigate through the dependency tree, and this is why I chose to build a TUI rather than a program that would only output static graphs. I decided to use the Go TUI interface from the package Go `github.com/rivo/tview`, as it provides all the basic features needed for this tool.

The TUI displays the dependency tree visualizations for packages and functions dependency trees, as displayed in the right column of Figures 5.4 and 5.5. The dependency tree of the selected element, package or function is always displayed in the right column of the interface. This structure is a dependency tree, which means that it may print duplicates of the same element, if it is a dependency for multiple elements. Due to this redundancy, the graph contains a lot of elements as some full branches of the dependency tree are duplicated. This dependency tree has a configurable depth, that eases its navigation. The dependency tree needed to be interactive, hence selecting an element in the dependency tree, displays this element's dependency tree instead, and the full dependency tree can easily be navigated this way. We also added a reverse dependency tree feature, which reverse the direction of the dependency tree. This means that the tree displays the elements depending on the selected element, instead of showing the selected element's dependencies. Thus, if we take the package dependency tree of the target package, in the reverse dependency tree, all the leaves will be duplicates of the target package, as it is the root of the normal dependency tree. The reverse dependency tree of the target package will be empty, as nothing depends on it. This feature is useful to understand the dependency chain of a specific library. For instance, if a weak algorithm implementation is present in the packages list, displaying its reverse dependency tree help to visualize why it is indirectly a dependency of the a project, and which package is importing it directly.

As the TUI is not able to display complex graphs, we added an export feature to export the dependency graphs in a visual form. Once a dependency tree is built it is possible to export its graph built using GraphViz[1] as an image. This results in a graph similar to Figure 5.1. There is a possibility to export both package dependency graphs and function dependency graphs, but the function dependency graph is far more complex compared with the package dependency graph. The inventory tool also has a feature to export the package chord dependency diagram. This diagram type is illustrated in Figures 5.7 and 5.8. As the library to produce these graphs

[1]`https://graphviz.org`

is a Javascript library, the graphs are made available from a browser at the address `http://localhost:8080`. The inventory tool also offers the possibility to export the dependency graph in JSON format, this is useful in the case the dependency graph has to be given as input in another software.

Dependency graphs can be filtered in a custom way. When the filter mode is enabled, the dependency graph is only partially displayed. Packages to be filtered can be selected, and these packages will be the only leaves of the dependency tree. This graph will contain all the paths from the root of the graph, the target package, to the filtered packages. For example, Figure 5.14 displays the dependency graph of the package `github.com/hyperledger/fabric/bccsp/sw` with the package `crypto/md5` as only filter. It only displays the different paths from the target package until the filtered package, and not the rest of the dependency graph. In the implementation, it is possible to add lists of filters, that can be combined in order to filter many packages, or only a specific subgroup in an agile way. For instance, if all standard cryptographic packages are defined as filters, the dependency graph will only contain packages that depends on standard cryptographic packages, directly or indirectly. Thus, the output graph will display all occurrences of cryptographic function calls in the target package and its dependencies, thus making its cryptographic inventory. For convenience, it is possible to import a list of filters from a simple JSON file, to avoid entering all filters manually for each dependency scan. There is also a functionality to export filter lists to JSON files, for convenience in future uses. Once that the filters are applied on the dependency graph, it can be exported in JSON, Graphviz and Chord formats, as described above.

5.3 Challenges

Multiple implementation challenges were encounters during this project. Building the call graph of a Go package has been quite a headache in the beginning. I did not know about the existence of the package recovering the call tree of any package `golang.org/x/tools/go/callgraph`, as this package is not widely used. I started to implement my own static code analyzer for Go, scanning the source code and resolving all imports and function calls. However, it was very challenging to retrieve the right dependencies that were used, only by analyzing the text of the source code of packages. Thus, I was truly relieved when I discovered the appropriate package solving the problem I was trying to address.

Dependency tree visualization has also been a difficult challenge to address. I discovered the Javascript `d3` library with the capability to build custom interactive diagram, and I immediately wanted to make use of it to represent the dependency trees. However, I have no experience with Javascript, it was thus very challenging for me to understand how to plot any data. Learning the basics of Javascript could have helped me to use this specific library, but it would have taken me a lot of precious time, and learning a new programming language was not in the scope of my project. I found an interesting use of this library on Github. The example was written in Go, and displayed Chord diagrams using Javascript code and a simple http server. I spent some time to study the structure of the Javascript code, and what I should adapt for my specific application, and was able to adapt the code for my project. I was able to integrate this sub project in the main implementation, and this is why the chord graph has to be accessed from a browser.

As stated before, I have no experience in general web programming (HTML, CSS, Javascript), and the initial interface of the inventory tool was supposed to be a graphical dashboard interface. I followed tutorials to learn how to build a dashboard using web technologies. But it turns out to be more complicated than I initially thought. I tried to make use of the Elastic Search and Kibana stack, but I quickly figured out that the features were not the ones I needed. I really needed to have an interactive interface, and not simply resolving dependency graphs and saving them as images, as real time navigation in the dependency tree gives more insights about the data than a static image. Thus, I decided to make an interactive interface in the command line environment. I had already built a TUI for a project in the past, thus I was able to use the same library, with which I was already familiar, to quickly build an interactive interface. The main interface is a TUI, but images of dependency graphs can be exported, and an interactive Chord graph is available from the browser. Of course, it is not as convenient and aesthetics as a dashboard, but it was a good trade-off between the time spent on the interface development, and the final result.

5.4 Limitations

PDF and image readers are quite fragile, and most of them cannot open complex files. In our case, large graphviz exported dependency graphs cannot be opened by PDF and image readers as they contain too many elements. Thus the use of these graphs is limited to relatively small graphs containing at most around 100 elements. Even if it was possible to export and open larger graphs, their use would be limited, as the graph would be too complex to be understood. Working with sub parts of the graph using the reverse dependency tree and filter features should offer a best experience than navigating a giant and complex graph.

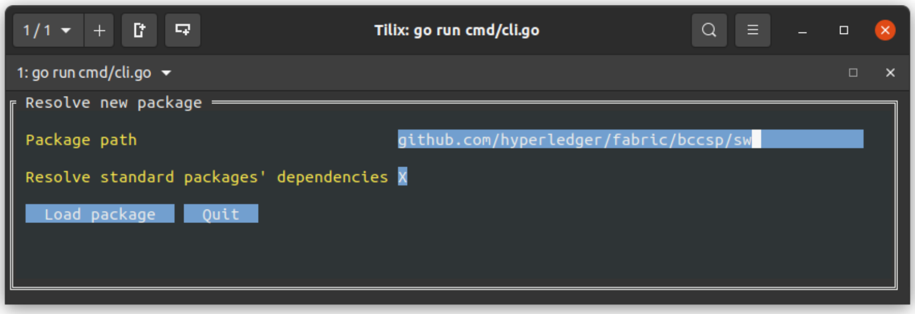
The TUI offers only a limited experience compared with a full dashboard graphical interface. The visual output is quite limited to the command line, and all interaction has to go through keyboard shortcut, which is not very convenient. As discussed above, a dashboard interface would allow to have more interactions with the data, build filters by clicking directly on packages, and navigate the dependency graph interactively by clicking on the nodes to explore or scrolling the 2D graph representation.

5.5 Installation and running guide

The source code of the project is available at `https://github.ibm.com/Cryptographic-Agility/go-sbom`, and is restricted to IBM employees, as the source code is confidential. The repository has to be cloned, or downloaded as an archive. The software can be run using the following command at the root of the project's repository:

```
> go run .
```

This will launch the program and give the interface as described in Figure 5.3. This screen asks the user to enter the unique identifier of the package to load. The package can be a standard Go package, or a package remotely accessible on a public repository. The checkbox below indicates whether the dependencies of standard dependencies should also be resolved. Standard dependencies are part of the Go language, and thus can be trusted, furthermore, it takes less time to load packages if the dependencies of standard packages are not resolved. Once these boxes are filled, pressing the `Load` button will load the selected package and its dependencies. This operation might take several minutes for packages with many dependencies.



```
Resolve new package

Package path                              github.com/hyperledger/fabric/bccsp/sw

Resolve standard packages' dependencies X

 Load package   Quit
```

Figure 5.3: Start screen

Before being able to scan a repository, it must first be fetched using the following command:

```
> go get -u [package path]
```

For private projects, the source code has to be stored in the appropriate location in the `$GOPATH/src` directory. To fetch it from a private repository, such as `github.ibm.com`, it is necessary to generate an access token, and use the following commands:

```
> git config --global url."https://<username>:<token>@ \\
    github.ibm.com".insteadOf "https://github.ibm.com"
> go env -w GOPRIVATE=github.ibm.com
> go get -u github.ibm.com/<username>/<project>
```

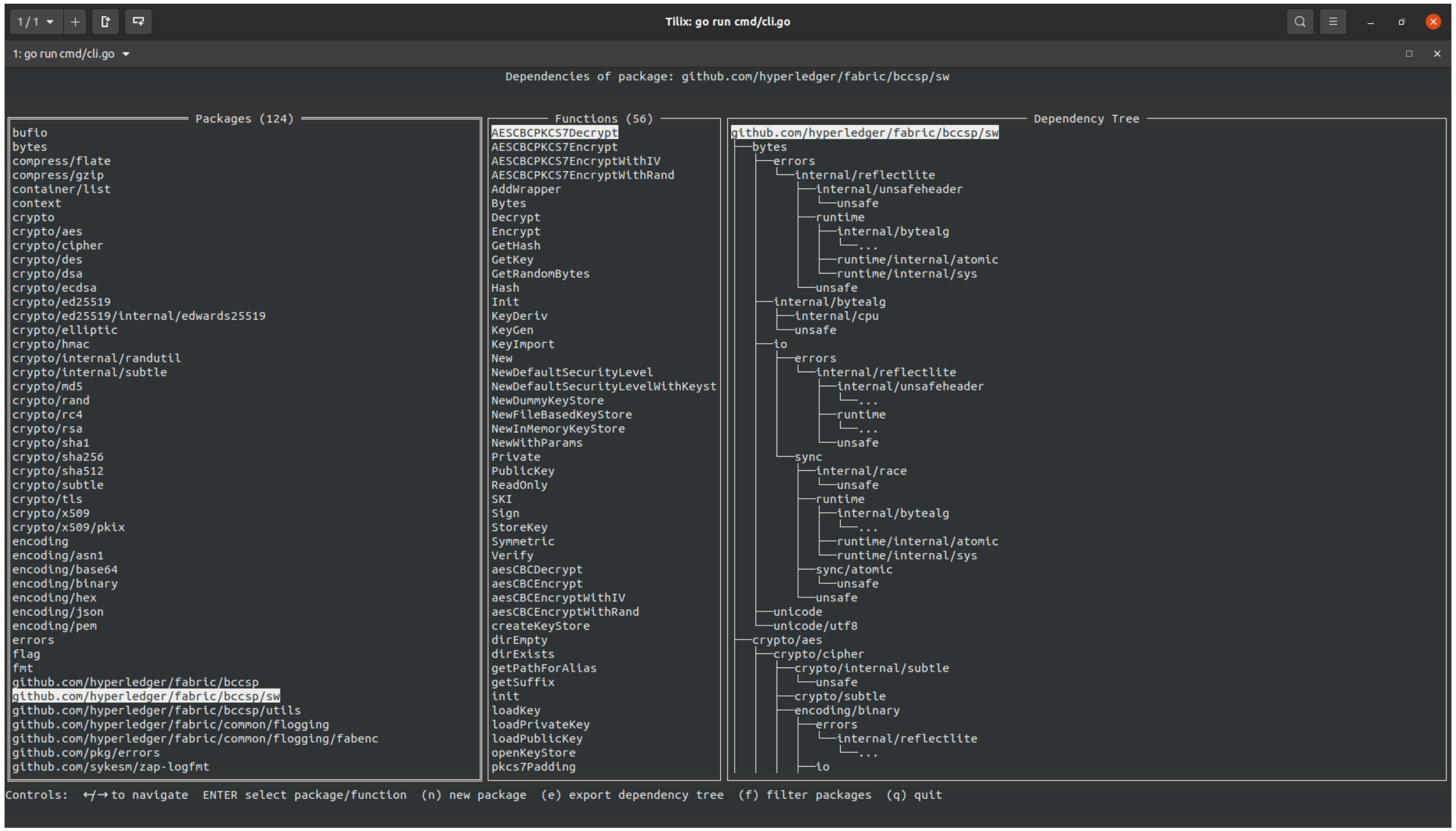
Once all dependencies are loaded, the display looks like Figure 5.4. The left column contains all package dependencies of the target package, in alphabetical order. The list can be scrolled using arrows to select any package. The middle columns contains the functions from the selected package from the right column that are part of the dependency tree of the target package, in alphabetical order. The right column contains the package dependency tree of the selected package, in Figure 5.4 it is the dependency tree of the target package. In this dependency tree, some libraries can appear multiple times, such as the `errors` package and its dependencies on Figure 5.4. The reason is that `errors` is a dependency for the packages `bytes`, and `io`. Compared with a graph without duplicates, the dependency tree is more hierarchical and readable, but redundant. A maximal depth can be set to the dependency tree, to keep a reasonable size. It is possible to navigate through the dependency tree, and selecting an element with `ENTER`, will display the sub-dependency tree, with the selected package as root. Therefore, this tree is quite easy to navigate, to explore the graph, part by part. Note that cyclic package dependency is not possible by design in Go.

Selecting a function from the middle column will display the dependency tree of this function in the right column, instead of the package dependency tree, as displayed in Figure 5.5. As the functions can rely on functions from other packages, the functions are identified with their `ID`, as defined in Section 5.2, with their package path, followed by their local function name. This dependency tree has the same features as the package dependency tree, except that it displays functions dependencies. Thus, this dependency tree is deeper than the package dependency tree. When selecting a specific function from the function dependency tree, the associated package and functions will be selected in the left and middle columns respectively, and this function will become the new root of the function dependency tree in the right column. Note that cyclic function dependencies are not displayed after the first cycle, as in the example shown in Listing 5.3.

```
a → b → c → a → ...

a
⊢ b
  ⊢ c
    ⊢ a
```

Listing 5.3: Cyclic function dependencies tree



```
Dependencies of package: github.com/hyperledger/fabric/bccsp/sw
```

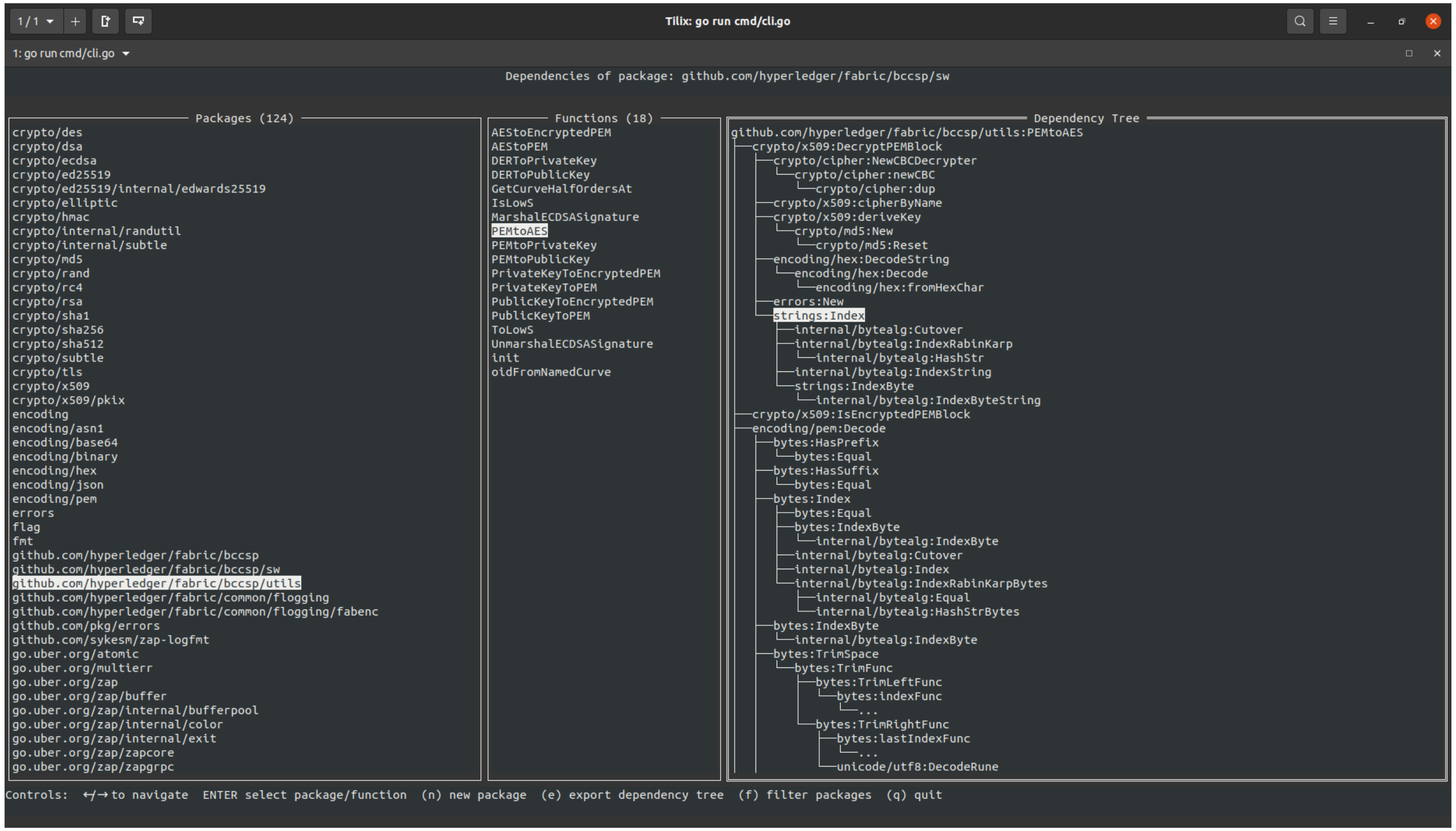
```
Packages (124)
bufio
bytes
compress/flate
compress/gzip
container/list
context
crypto
crypto/aes
crypto/cipher
crypto/des
crypto/dsa
crypto/ecdsa
crypto/ed25519
crypto/ed25519/internal/edwards25519
crypto/elliptic
crypto/hmac
crypto/internal/randutil
crypto/internal/subtle
crypto/md5
crypto/rand
crypto/rc4
crypto/rsa
crypto/sha1
crypto/sha256
crypto/sha512
crypto/subtle
crypto/tls
crypto/x509
crypto/x509/pkix
encoding
encoding/asn1
encoding/base64
encoding/binary
encoding/hex
encoding/json
encoding/pem
errors
flag
fmt
github.com/hyperledger/fabric/bccsp
github.com/hyperledger/fabric/bccsp/sw
github.com/hyperledger/fabric/bccsp/utils
github.com/hyperledger/fabric/common/flogging
github.com/hyperledger/fabric/common/flogging/fabenc
github.com/pkg/errors
github.com/sykesm/zap-logfmt
```

```
Functions (56)
AESCBCPKCS7Decrypt
AESCBCPKCS7Encrypt
AESCBCPKCS7EncryptWithIV
AESCBCPKCS7EncryptWithRand
AddWrapper
Bytes
Decrypt
Encrypt
GetHash
GetKey
GetRandomBytes
Hash
Init
KeyDeriv
KeyGen
KeyImport
New
NewDefaultSecurityLevel
NewDefaultSecurityLevelWithKeyst
NewDummyKeyStore
NewFileBasedKeyStore
NewInMemoryKeyStore
NewWithParams
Private
PublicKey
ReadOnly
SKI
Sign
StoreKey
Symmetric
Verify
aesCBCDecrypt
aesCBCEncrypt
aesCBCEncryptWithIV
aesCBCEncryptWithRand
createKeyStore
dirEmpty
dirExists
getPathForAlias
getSuffix
init
loadKey
loadPrivateKey
loadPublicKey
openKeyStore
pkcs7Padding
```

```
Dependency Tree
github.com/hyperledger/fabric/bccsp/sw
├───bytes
│   ├───errors
│   │   └───internal/reflectlite
│   │       ├───internal/unsafeheader
│   │       │   └───unsafe
│   │       ├───runtime
│   │       │   ├───internal/bytealg
│   │       │   │   └───...
│   │       │   ├───runtime/internal/atomic
│   │       │   └───runtime/internal/sys
│   │       └───unsafe
│   ├───internal/bytealg
│   │   ├───internal/cpu
│   │   └───unsafe
│   ├───io
│   │   ├───errors
│   │   │   └───internal/reflectlite
│   │   │       ├───internal/unsafeheader
│   │   │       │   └───...
│   │   │       ├───runtime
│   │   │       │   └───...
│   │   │       └───unsafe
│   │   └───sync
│   │       ├───internal/race
│   │       │   └───unsafe
│   │       ├───runtime
│   │       │   ├───internal/bytealg
│   │       │   │   └───...
│   │       │   ├───runtime/internal/atomic
│   │       │   └───runtime/internal/sys
│   │       ├───sync/atomic
│   │       │   └───unsafe
│   │       └───unsafe
│   ├───unicode
│   └───unicode/utf8
├───crypto/aes
│   ├───crypto/cipher
│   │   ├───crypto/internal/subtle
│   │   │   └───unsafe
│   │   ├───crypto/subtle
│   │   ├───encoding/binary
│   │   │   ├───errors
│   │   │   │   └───internal/reflectlite
│   │   │   │       └───...
│   │   │   ├───io
```

```
Controls:  ←/→ to navigate  ENTER select package/function  (n) new package  (e) export dependency tree  (f) filter packages  (q) quit
```

Figure 5.4: Main menu displaying package dependency tree



```
Tilix: go run cmd/cli.go
1: go run cmd/cli.go

Dependencies of package: github.com/hyperledger/fabric/bccsp/sw

Packages (124)
crypto/des
crypto/dsa
crypto/ecdsa
crypto/ed25519
crypto/ed25519/internal/edwards25519
crypto/elliptic
crypto/hmac
crypto/internal/randutil
crypto/internal/subtle
crypto/md5
crypto/rand
crypto/rc4
crypto/rsa
crypto/sha1
crypto/sha256
crypto/sha512
crypto/subtle
crypto/tls
crypto/x509
crypto/x509/pkix
encoding
encoding/asn1
encoding/base64
encoding/binary
encoding/hex
encoding/json
encoding/pem
errors
flag
fmt
github.com/hyperledger/fabric/bccsp
github.com/hyperledger/fabric/bccsp/sw
github.com/hyperledger/fabric/bccsp/utils
github.com/hyperledger/fabric/common/flogging
github.com/hyperledger/fabric/common/flogging/fabenc
github.com/pkg/errors
github.com/sykesm/zap-logfmt
go.uber.org/atomic
go.uber.org/multierr
go.uber.org/zap
go.uber.org/zap/buffer
go.uber.org/zap/internal/bufferpool
go.uber.org/zap/internal/color
go.uber.org/zap/internal/exit
go.uber.org/zap/zapcore
go.uber.org/zap/zapgrpc

Functions (18)
AEStoEncryptedPEM
AEStoPEM
DERToPrivateKey
DERToPublicKey
GetCurveHalfOrdersAt
IsLowS
MarshalECDSASignature
PEMtoAES
PEMtoPrivateKey
PEMtoPublicKey
PrivateKeyToEncryptedPEM
PrivateKeyToPEM
PublicKeyToEncryptedPEM
PublicKeyToPEM
ToLowS
UnmarshalECDSASignature
init
oidFromNamedCurve

Dependency Tree
github.com/hyperledger/fabric/bccsp/utils:PEMtoAES
├──crypto/x509:DecryptPEMBlock
│  ├──crypto/cipher:NewCBCDecrypter
│  │  └──crypto/cipher:newCBC
│  │     └──crypto/cipher:dup
│  ├──crypto/x509:cipherByName
│  ├──crypto/x509:deriveKey
│  │  └──crypto/md5:New
│  │     └──crypto/md5:Reset
│  ├──encoding/hex:DecodeString
│  │  └──encoding/hex:Decode
│  │     └──encoding/hex:fromHexChar
│  ├──errors:New
│  └──strings:Index
│     ├──internal/bytealg:Cutover
│     ├──internal/bytealg:IndexRabinKarp
│     │  └──internal/bytealg:HashStr
│     ├──internal/bytealg:IndexString
│     └──strings:IndexByte
│        └──internal/bytealg:IndexByteString
├──crypto/x509:IsEncryptedPEMBlock
├──encoding/pem:Decode
│  ├──bytes:HasPrefix
│  │  └──bytes:Equal
│  ├──bytes:HasSuffix
│  │  └──bytes:Equal
│  ├──bytes:Index
│  │  ├──bytes:Equal
│  │  ├──bytes:IndexByte
│  │  │  └──internal/bytealg:IndexByte
│  │  ├──internal/bytealg:Cutover
│  │  ├──internal/bytealg:Index
│  │  └──internal/bytealg:IndexRabinKarpBytes
│  │     ├──internal/bytealg:Equal
│  │     └──internal/bytealg:HashStrBytes
│  ├──bytes:IndexByte
│  │  └──internal/bytealg:IndexByte
│  ├──bytes:TrimSpace
│  │  └──bytes:TrimFunc
│  │     ├──bytes:TrimLeftFunc
│  │     │  └──bytes:indexFunc
│  │     │     └──...
│  │     └──bytes:TrimRightFunc
│  │        ├──bytes:lastIndexFunc
│  │        │  └──...
│  │        └──unicode/utf8:DecodeRune
│  │

Controls:  ←/→ to navigate  ENTER select package/function  (n) new package  (e) export dependency tree  (f) filter packages  (q) quit
```

Figure 5.5: Main menu displaying function dependency tree

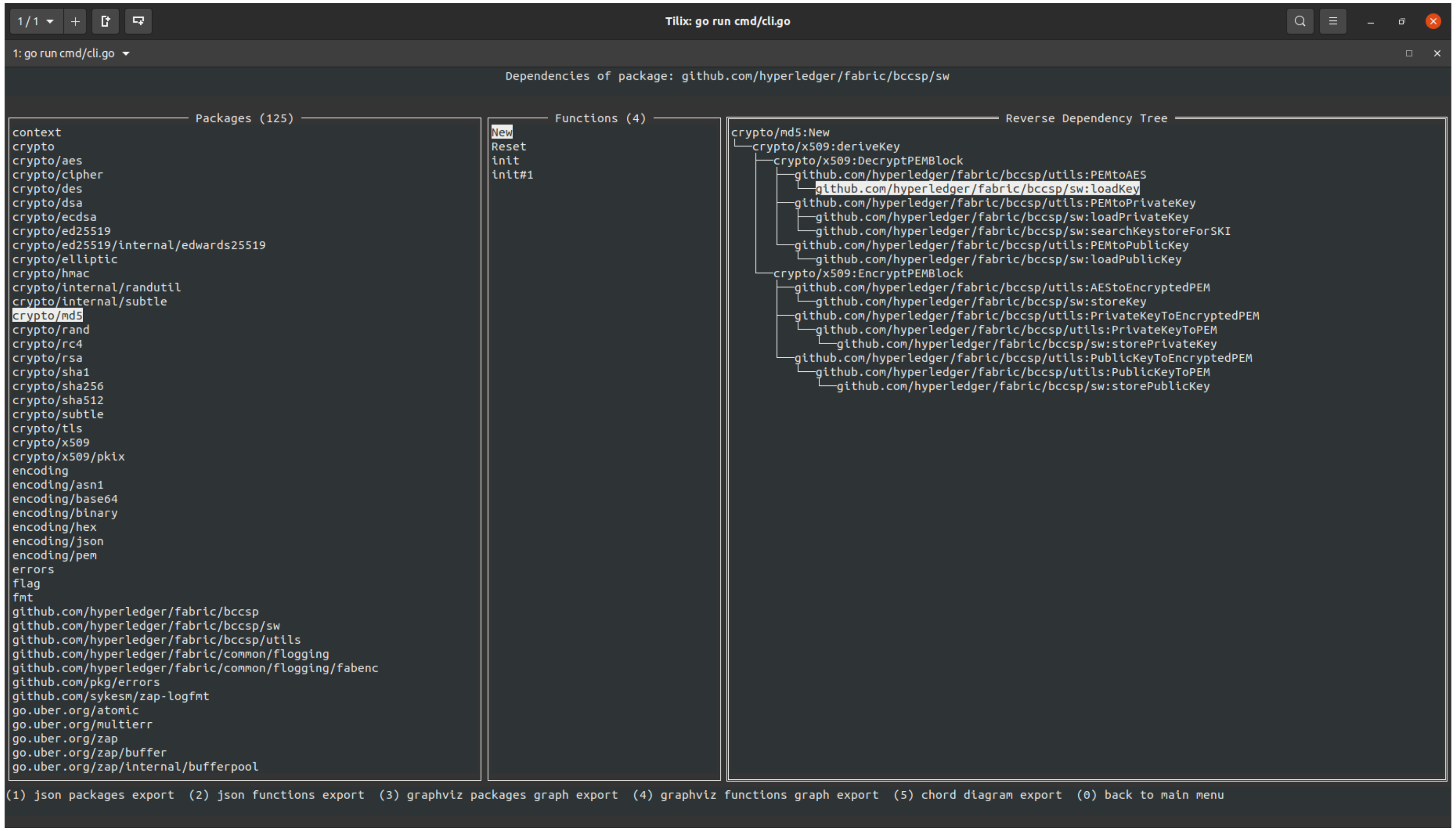
A new target package can be loaded with its dependencies by pressing `n`, which will give a form similar to Figure 5.3. Note that dependencies that have already been resolve don't need to be loaded again, the packages are stored in a cache.Thus, loading a new target package with similar dependencies as the current target package will take less time.

Pressing `r` will reverse the dependency tree, as shown in Figure 5.6. The reverse package dependency tree will display the packages that depends on the selected one, instead of showing the packages on which the selected package depends on. For instance, the dependency tree of the target package will contain all the packages from the package list in the left column, whereas its reverse dependency tree will be empty, as no package, in its dependency tree, depends on the target package. The leaves of the reverse dependency tree will all be duplicates of the target package. This feature is equivalent for the function dependency tree. The reverse dependency tree is useful to understand how a specific package or function is indirectly used by a software.

Figure 5.6 shows the example of the reverse function dependency tree of the function `New` from the `crypto/md5` package. It shows that the package `crypto/md5` is directly used only by the packages `crypto/tls` and `crypto/x509`. It is possible to look more precisely which functions make use of the package `crypto/md5`. In this specific case, we see that the `md5` algorithm is used by functions from the `crypto/x509` package, that is used by multiple functions in from the `bccsp` project. The source code of `DecryptPEMBlock` and `EncryptPEMBlock` from the `crypto/x509` package is printed in Listing 5.4. The comments state that both functions are deprecated, as Legacy PEM encryption is insecure by design. Thus this function should not have been used, and the scanner allowed us to detect the use of a deprecated function by checking the use of `md5` through the dependencies.

```go
// EncryptPEMBlock returns a PEM block of the specified type
// holding the given DER encoded data encrypted with the
// specified algorithm and password according to RFC 1423.
//
// Deprecated: Legacy PEM encryption as specified in RFC 1423
// is insecure by design. Since it does not authenticate the
// ciphertext, it is vulnerable to padding oracle attacks that
// can let an attacker recover the plaintext.
func EncryptPEMBlock(rand io.Reader, blockType string, data,
    password []byte, alg PEMCipher) (*pem.Block, error) {
     ...
}

// DecryptPEMBlock takes a PEM block encrypted according to RFC
// 1423 and the password used to encrypt it and returns a slice
// of decrypted DER encoded bytes. It inspects the DEK-Info
// header to determine the algorithm used for decryption. If no
// DEK-Info header is present, an error is returned. If an
// incorrect password is detected an IncorrectPasswordError is
```



```
Dependencies of package: github.com/hyperledger/fabric/bccsp/sw

Packages (125)
context
crypto
crypto/aes
crypto/cipher
crypto/des
crypto/dsa
crypto/ecdsa
crypto/ed25519
crypto/ed25519/internal/edwards25519
crypto/elliptic
crypto/hmac
crypto/internal/randutil
crypto/internal/subtle
crypto/md5
crypto/rand
crypto/rc4
crypto/rsa
crypto/sha1
crypto/sha256
crypto/sha512
crypto/subtle
crypto/tls
crypto/x509
crypto/x509/pkix
encoding
encoding/asn1
encoding/base64
encoding/binary
encoding/hex
encoding/json
encoding/pem
errors
flag
fmt
github.com/hyperledger/fabric/bccsp
github.com/hyperledger/fabric/bccsp/sw
github.com/hyperledger/fabric/bccsp/utils
github.com/hyperledger/fabric/common/flogging
github.com/hyperledger/fabric/common/flogging/fabenc
github.com/pkg/errors
github.com/sykesm/zap-logfmt
go.uber.org/atomic
go.uber.org/multierr
go.uber.org/zap
go.uber.org/zap/buffer
go.uber.org/zap/internal/bufferpool

Functions (4)
New
Reset
init
init#1

Reverse Dependency Tree
crypto/md5:New
└─crypto/x509:deriveKey
  ├─crypto/x509:DecryptPEMBlock
  │ ├─github.com/hyperledger/fabric/bccsp/utils:PEMtoAES
  │ │ └─github.com/hyperledger/fabric/bccsp/sw:loadKey
  │ ├─github.com/hyperledger/fabric/bccsp/utils:PEMtoPrivateKey
  │ │ ├─github.com/hyperledger/fabric/bccsp/sw:loadPrivateKey
  │ │ └─github.com/hyperledger/fabric/bccsp/sw:searchKeystoreForSKI
  │ └─github.com/hyperledger/fabric/bccsp/utils:PEMtoPublicKey
  │   └─github.com/hyperledger/fabric/bccsp/sw:loadPublicKey
  └─crypto/x509:EncryptPEMBlock
    ├─github.com/hyperledger/fabric/bccsp/utils:AEStoEncryptedPEM
    │ └─github.com/hyperledger/fabric/bccsp/sw:storeKey
    ├─github.com/hyperledger/fabric/bccsp/utils:PrivateKeyToEncryptedPEM
    │ └─github.com/hyperledger/fabric/bccsp/utils:PrivateKeyToPEM
    │   └─github.com/hyperledger/fabric/bccsp/sw:storePrivateKey
    └─github.com/hyperledger/fabric/bccsp/utils:PublicKeyToEncryptedPEM
      └─github.com/hyperledger/fabric/bccsp/utils:PublicKeyToPEM
        └─github.com/hyperledger/fabric/bccsp/sw:storePublicKey

(1) json packages export  (2) json functions export  (3) graphviz packages graph export  (4) graphviz functions graph export  (5) chord diagram export  (0) back to main menu
```

Figure 5.6: Main menu displaying the reverse package dependency tree for the package `crypto/md5`

```go
// returned. Because of deficiencies in the format, it's not
// always possible to detect an incorrect password. In these
// cases no error will be returned but the decrypted DER bytes
// will be random noise.
//
// Deprecated: Legacy PEM encryption as specified in RFC 1423
// is insecure by design. Since it does not authenticate the
// ciphertext, it is vulnerable to padding oracle attacks that
// can let an attacker recover the plaintext.
func DecryptPEMBlock(b *pem.Block, password []byte) ([]byte,
    error) {
    ...
}
```

Listing 5.4: Source code of `crypto/x509/pem_decrypt.go`

By pressing the `e` key from the main window, it is possible to export the dependency tree in different formats, as shown at the bottom of Figure 5.6. It is possible to export the dependency tree in the form of a package dependency tree or in the form of a function dependency tree in JSON format. The format of the JSON structure is very simple, it is an array of objects, with a `"package"` or `"function"` string field, which identify respectively the package or the function, and an `"imports"` array field containing all the dependencies as string identifiers of the associated package or function.

It is also possible to export the dependency trees in a graphical way, using GraphViz. This utility builds graphs as shown in 5.1. It is thus possible to export package and function dependency tree in this format, and save the images as `png` or `svg`. However, most function dependency graphs exported using Graphviz are too large to be read by PDF readers. In our example, even the package dependency graph is too complex to be opened by a PDF reader, and even if this graph could have been rendered, it would have been to complicated to interpret because of its size.

In order to have an overview of complex package dependency graphs, we added the possibility to generate the associated chord graph as shown in Figure 5.7 and 5.8. These graphs are built in Javascript and running on a Go http server, the graph is available from a browser at `http://localhost:8080`. These graphs are interactive, and by selecting a package, it will filter all the dependencies of this package in a specific color, and all the packages that depend on the selected one in different colors as demonstrated in 5.8. Thus this representation is helpful to visualize dependencies in very large graphs that could not be read by PDF readers, even though the output is not very human friendly when no package is selected, as pictured in Figure 5.7.

The inventory tool support package filtering. By pressing the key `f` from the main menu, it will open the *filters interface*, shown in Figure 5.9. It is possible to have multiple filter lists, each filter list is numbered and can be enabled by pressing the number associated with the list. It is possible

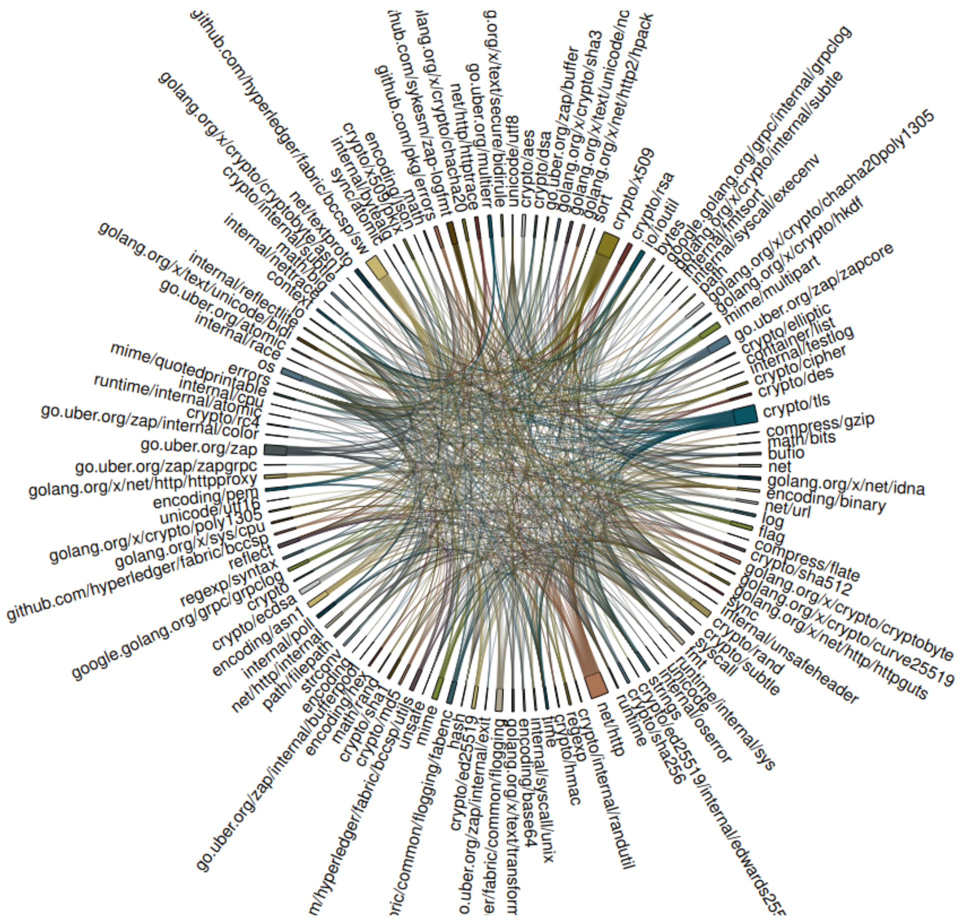


Figure 5.7: Chord diagram of `github.com/hyperledger/fabric/bccsp/sw`

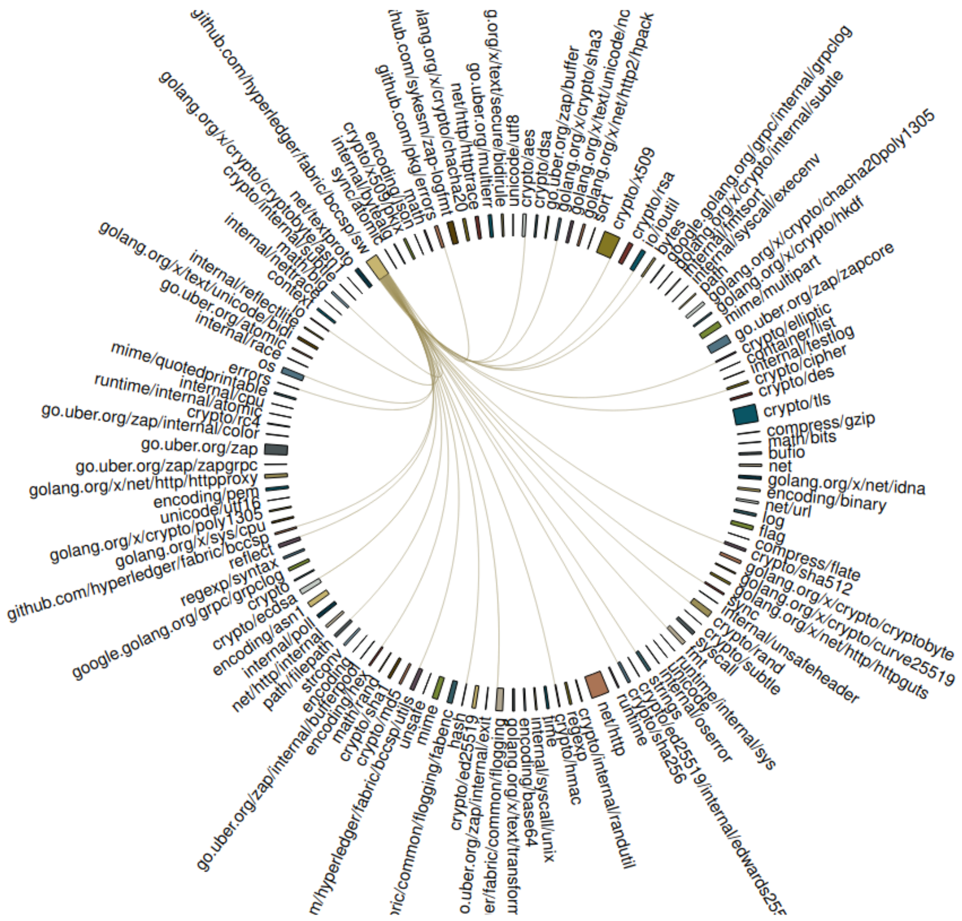


Figure 5.8: Chord diagram with dependencies highlighted for package `github.com/hyperledger/fabric/bccsp/sw`

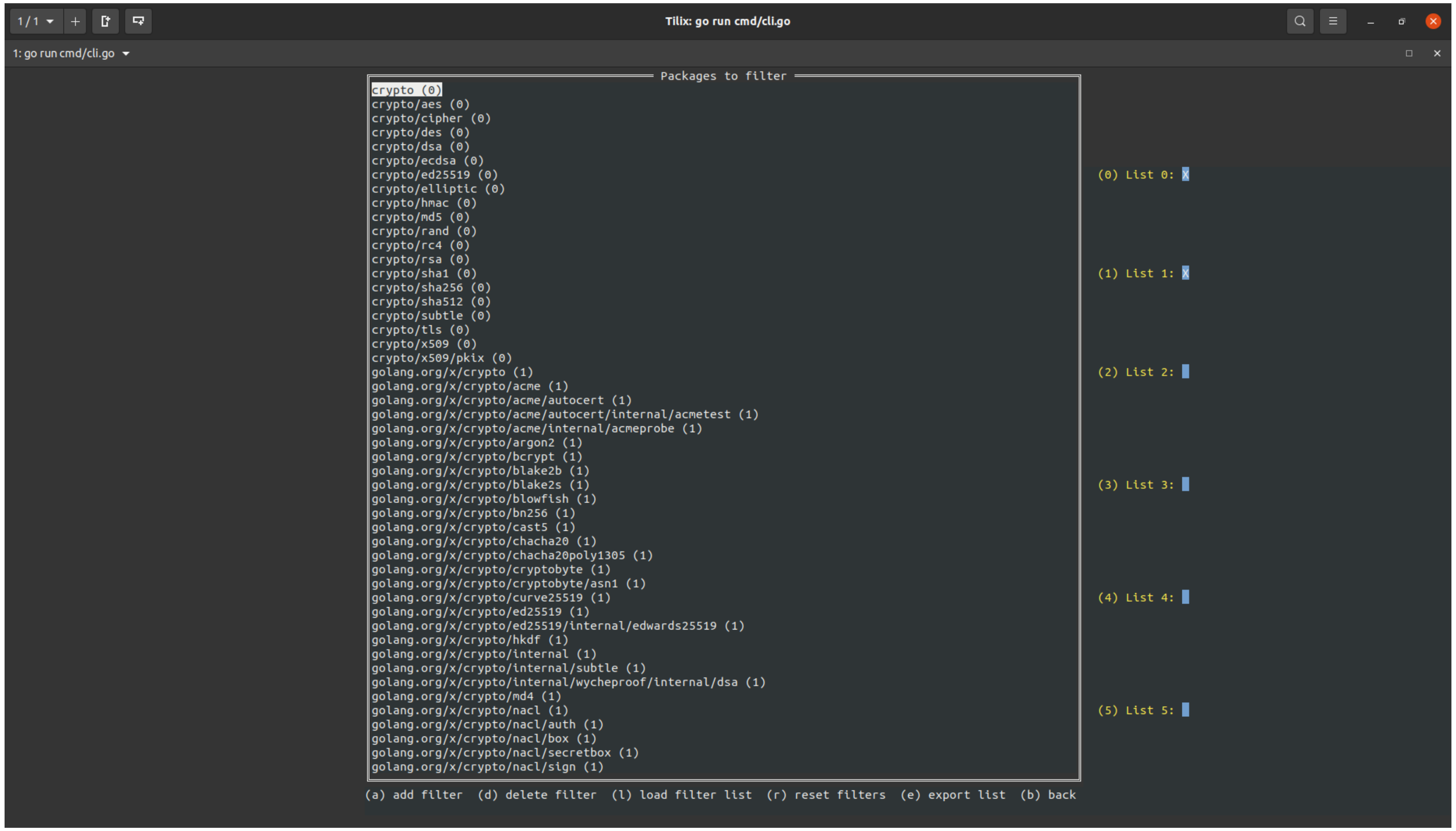
to manually add package filters, and choose to which list it belongs. There is also a feature to import filter lists from JSON files, and to export them to JSON files after they are created of modified. This allows to load large list of filters, and reusing them for different scans. The filter lists are useful to filter only a selection of packages, and if multiple lists are selected the new filter list is the union of all the selected lists. Pressing `d` deletes the selected filter, and pressing `r` deletes all filters. In Figure 5.9, we loaded two filter lists containing all Go standard cryptographic packages, and selected these lists. It is possible to make use of this filter feature to characterize cryptographic algorithms into categories, for instance *Broken*, *Non quantum safe* and *Post-quantum* algorithms.

When coming back to the main menu, after applying the packages filters, the result is as shown in Figure 5.10. The only parts left of the dependency tree are the paths from the root to the packages selected in the filter lists, here the standard cryptographic packages. We can see that the number of packages on the left column went from 124 in Figure 5.4 down to only 38 in Figure 5.10. The only packages left are the ones making direct or indirect use of the selected packages from the filter lists. It is now easier to navigate through the cryptographic dependencies and to export a cryptographic inventory, because we got rid of all other dependencies. When displaying the chord diagram of the filtered packages we get the result display in Figures 5.11 and 5.12. The output is much clearer and easier to interpret compared with Figures 5.7 and 5.8. Figure 5.8, displays the direct cryptographic dependencies of the package `net/http`.

Figure 5.13 represents a cropped version of the Graphviz package dependency graph diagram. After applying the filters, the graph could be open by a PDF reader, but is too large to be included on a A4 paper sheet, thus the graph was cropped to display a preview. Even though the number of packaged is now reduced, the graph representation stays quite complex, and filtering less packages may be useful in order to get a smaller and more readable graph. Figure 5.14 displays the result of filtering only a single package, `crypto/md5`, and the result is easily readable.

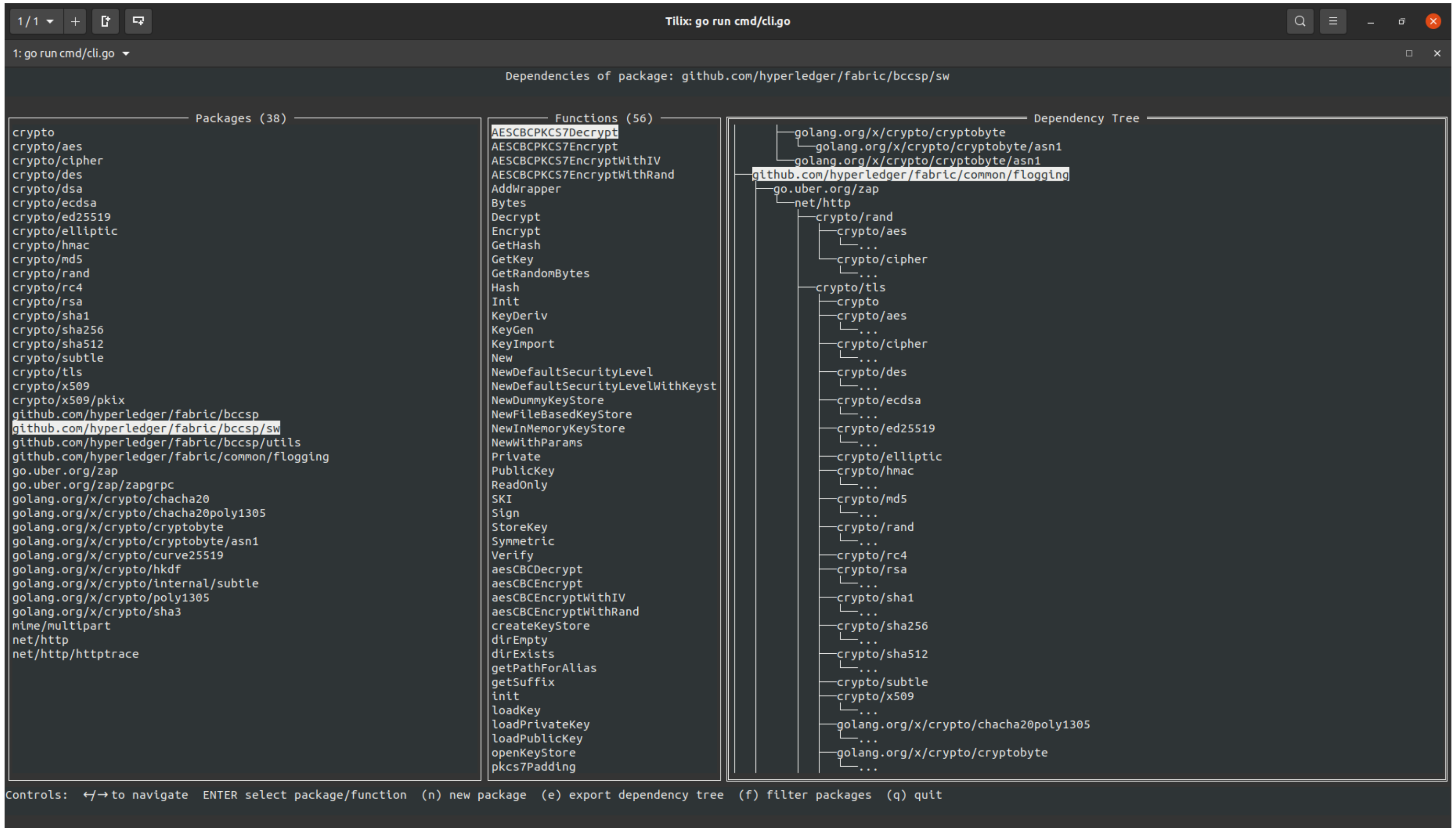
5.6 Future work

This inventory tool currently has limited capabilities, and therefore, adding more features could make it more useful in production. An interesting feature would be to export the inventory in the SPDX format, as discussed in Section 4.5.2. As discussed the SPDX format is quite complex, and its main purpose is to list licenses. However, dependencies can be included and having an output that is a recognized standard is always convenient for compatibility between software. Another useful feature to add would be the support for cryptographic policies. If the inventory can take as input a



```
                    Packages to filter
crypto (0)
crypto/aes (0)
crypto/cipher (0)
crypto/des (0)
crypto/dsa (0)
crypto/ecdsa (0)
crypto/ed25519 (0)                                          (0) List 0: X
crypto/elliptic (0)
crypto/hmac (0)
crypto/md5 (0)
crypto/rand (0)
crypto/rc4 (0)
crypto/rsa (0)
crypto/sha1 (0)                                             (1) List 1: X
crypto/sha256 (0)
crypto/sha512 (0)
crypto/subtle (0)
crypto/tls (0)
crypto/x509 (0)
crypto/x509/pkix (0)
golang.org/x/crypto (1)                                     (2) List 2:
golang.org/x/crypto/acme (1)
golang.org/x/crypto/acme/autocert (1)
golang.org/x/crypto/acme/autocert/internal/acmetest (1)
golang.org/x/crypto/acme/internal/acmeprobe (1)
golang.org/x/crypto/argon2 (1)
golang.org/x/crypto/bcrypt (1)
golang.org/x/crypto/blake2b (1)
golang.org/x/crypto/blake2s (1)                             (3) List 3:
golang.org/x/crypto/blowfish (1)
golang.org/x/crypto/bn256 (1)
golang.org/x/crypto/cast5 (1)
golang.org/x/crypto/chacha20 (1)
golang.org/x/crypto/chacha20poly1305 (1)
golang.org/x/crypto/cryptobyte (1)
golang.org/x/crypto/cryptobyte/asn1 (1)
golang.org/x/crypto/curve25519 (1)                          (4) List 4:
golang.org/x/crypto/ed25519 (1)
golang.org/x/crypto/ed25519/internal/edwards25519 (1)
golang.org/x/crypto/hkdf (1)
golang.org/x/crypto/internal (1)
golang.org/x/crypto/internal/subtle (1)
golang.org/x/crypto/internal/wycheproof/internal/dsa (1)
golang.org/x/crypto/md4 (1)
golang.org/x/crypto/nacl (1)                                (5) List 5:
golang.org/x/crypto/nacl/auth (1)
golang.org/x/crypto/nacl/box (1)
golang.org/x/crypto/nacl/secretbox (1)
golang.org/x/crypto/nacl/sign (1)
(a) add filter  (d) delete filter  (l) load filter list  (r) reset filters  (e) export list  (b) back
```

Figure 5.9: Package filters menu



```
Dependencies of package: github.com/hyperledger/fabric/bccsp/sw
```

```
Packages (38)
crypto
crypto/aes
crypto/cipher
crypto/des
crypto/dsa
crypto/ecdsa
crypto/ed25519
crypto/elliptic
crypto/hmac
crypto/md5
crypto/rand
crypto/rc4
crypto/rsa
crypto/sha1
crypto/sha256
crypto/sha512
crypto/subtle
crypto/tls
crypto/x509
crypto/x509/pkix
github.com/hyperledger/fabric/bccsp
github.com/hyperledger/fabric/bccsp/sw
github.com/hyperledger/fabric/bccsp/utils
github.com/hyperledger/fabric/common/flogging
go.uber.org/zap
go.uber.org/zap/zapgrpc
golang.org/x/crypto/chacha20
golang.org/x/crypto/chacha20poly1305
golang.org/x/crypto/cryptobyte
golang.org/x/crypto/cryptobyte/asn1
golang.org/x/crypto/curve25519
golang.org/x/crypto/hkdf
golang.org/x/crypto/internal/subtle
golang.org/x/crypto/poly1305
golang.org/x/crypto/sha3
mime/multipart
net/http
net/http/httptrace
```

```
Functions (56)
AESCBCPKCS7Decrypt
AESCBCPKCS7Encrypt
AESCBCPKCS7EncryptWithIV
AESCBCPKCS7EncryptWithRand
AddWrapper
Bytes
Decrypt
Encrypt
GetHash
GetKey
GetRandomBytes
Hash
Init
KeyDeriv
KeyGen
KeyImport
New
NewDefaultSecurityLevel
NewDefaultSecurityLevelWithKeyst
NewDummyKeyStore
NewFileBasedKeyStore
NewInMemoryKeyStore
NewWithParams
Private
PublicKey
ReadOnly
SKI
Sign
StoreKey
Symmetric
Verify
aesCBCDecrypt
aesCBCEncrypt
aesCBCEncryptWithIV
aesCBCEncryptWithRand
createKeyStore
dirEmpty
dirExists
getPathForAlias
getSuffix
init
loadKey
loadPrivateKey
loadPublicKey
openKeyStore
pkcs7Padding
```

```
Dependency Tree
│    ├─golang.org/x/crypto/cryptobyte
│    │  └─golang.org/x/crypto/cryptobyte/asn1
│    └─golang.org/x/crypto/cryptobyte/asn1
└─github.com/hyperledger/fabric/common/flogging
   ├─go.uber.org/zap
   │  └─net/http
   │     ├─crypto/rand
   │     │  ├─crypto/aes
   │     │  │  └─...
   │     │  └─crypto/cipher
   │     │     └─...
   │     ├─crypto/tls
   │     │  ├─crypto
   │     │  ├─crypto/aes
   │     │  │  └─...
   │     │  ├─crypto/cipher
   │     │  │  └─...
   │     │  ├─crypto/des
   │     │  │  └─...
   │     │  ├─crypto/ecdsa
   │     │  │  └─...
   │     │  ├─crypto/ed25519
   │     │  │  └─...
   │     │  ├─crypto/elliptic
   │     │  ├─crypto/hmac
   │     │  │  └─...
   │     │  ├─crypto/md5
   │     │  │  └─...
   │     │  ├─crypto/rand
   │     │  │  └─...
   │     │  ├─crypto/rc4
   │     │  ├─crypto/rsa
   │     │  │  └─...
   │     │  ├─crypto/sha1
   │     │  │  └─...
   │     │  ├─crypto/sha256
   │     │  │  └─...
   │     │  ├─crypto/sha512
   │     │  │  └─...
   │     │  ├─crypto/subtle
   │     │  ├─crypto/x509
   │     │  │  └─...
   │     │  ├─golang.org/x/crypto/chacha20poly1305
   │     │  │  └─...
   │     │  ├─golang.org/x/crypto/cryptobyte
   │     │  │  └─...
```

```
Controls:  ←/→ to navigate  ENTER select package/function  (n) new package  (e) export dependency tree  (f) filter packages  (q) quit
```

Figure 5.10: Main menu displaying the filtered package dependency tree

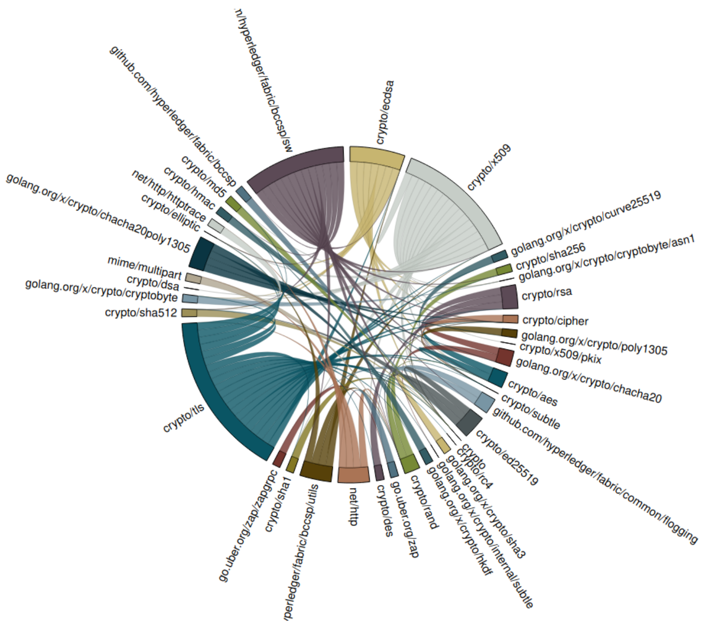

n/hyperledger/fabric/bccsp/sw
crypto/ecdsa
crypto/x509
github.com/hyperledger/fabric/bccsp
crypto/md5
crypto/hmac
net/http/httptrace
crypto/elliptic
golang.org/x/crypto/chacha20poly1305
mime/multipart
crypto/dsa
golang.org/x/crypto/cryptobyte
crypto/sha512
crypto/tls
golang.org/x/crypto/curve25519
crypto/sha256
golang.org/x/crypto/cryptobyte/asn1
crypto/rsa
crypto/cipher
golang.org/x/crypto/poly1305
crypto/x509/pkix
golang.org/x/crypto/chacha20
crypto/aes
crypto/subtle
github.com/hyperledger/fabric/common/flogging
crypto/ed25519
crypto
crypto/rc4
golang.org/x/crypto/sha3
golang.org/x/crypto/internal/subtle
golang.org/x/crypto/hkdf
crypto/rand
go.uber.org/zap
crypto/des
net/http
rperledger/fabric/bccsp/utils
crypto/sha1
go.uber.org/zap/zapgrpc

Figure 5.11: Filtered chord diagram of the Go package `github.com/hyperledger/fabric/bccsp/sw`

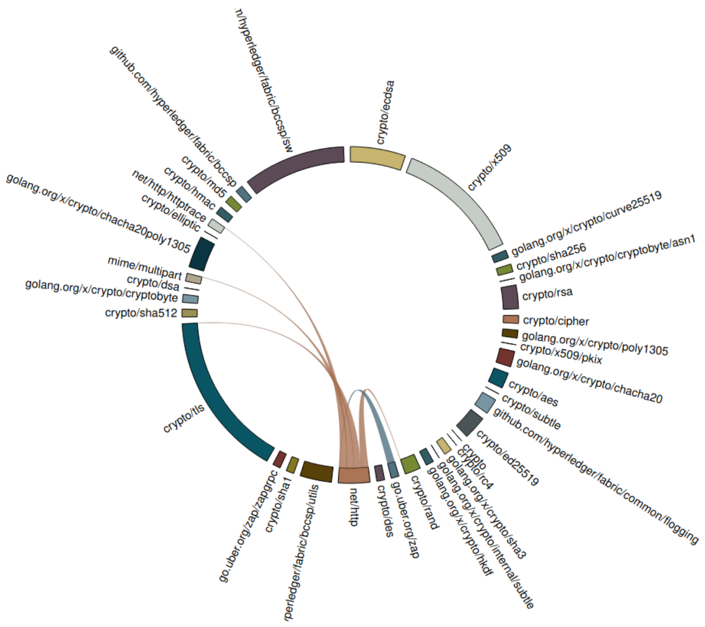

n/hyperledger/fabric/bccsp/sw
crypto/ecdsa
crypto/x509
github.com/hyperledger/fabric/bccsp
crypto/md5
crypto/hmac
net/http/httptrace
crypto/elliptic
golang.org/x/crypto/chacha20poly1305
mime/multipart
crypto/dsa
golang.org/x/crypto/cryptobyte
crypto/sha512
crypto/tls
golang.org/x/crypto/curve25519
crypto/sha256
golang.org/x/crypto/cryptobyte/asn1
crypto/rsa
crypto/cipher
golang.org/x/crypto/poly1305
crypto/x509/pkix
golang.org/x/crypto/chacha20
crypto/aes
crypto/subtle
github.com/hyperledger/fabric/common/flogging
crypto/ed25519
crypto
crypto/rc4
golang.org/x/crypto/sha3
golang.org/x/crypto/internal/subtle
golang.org/x/crypto/hkdf
crypto/rand
go.uber.org/zap
crypto/des
net/http
rperledger/fabric/bccsp/utils
crypto/sha1
go.uber.org/zap/zapgrpc

Figure 5.12: Filtered chord diagram with dependencies highlighted for package `net/http`

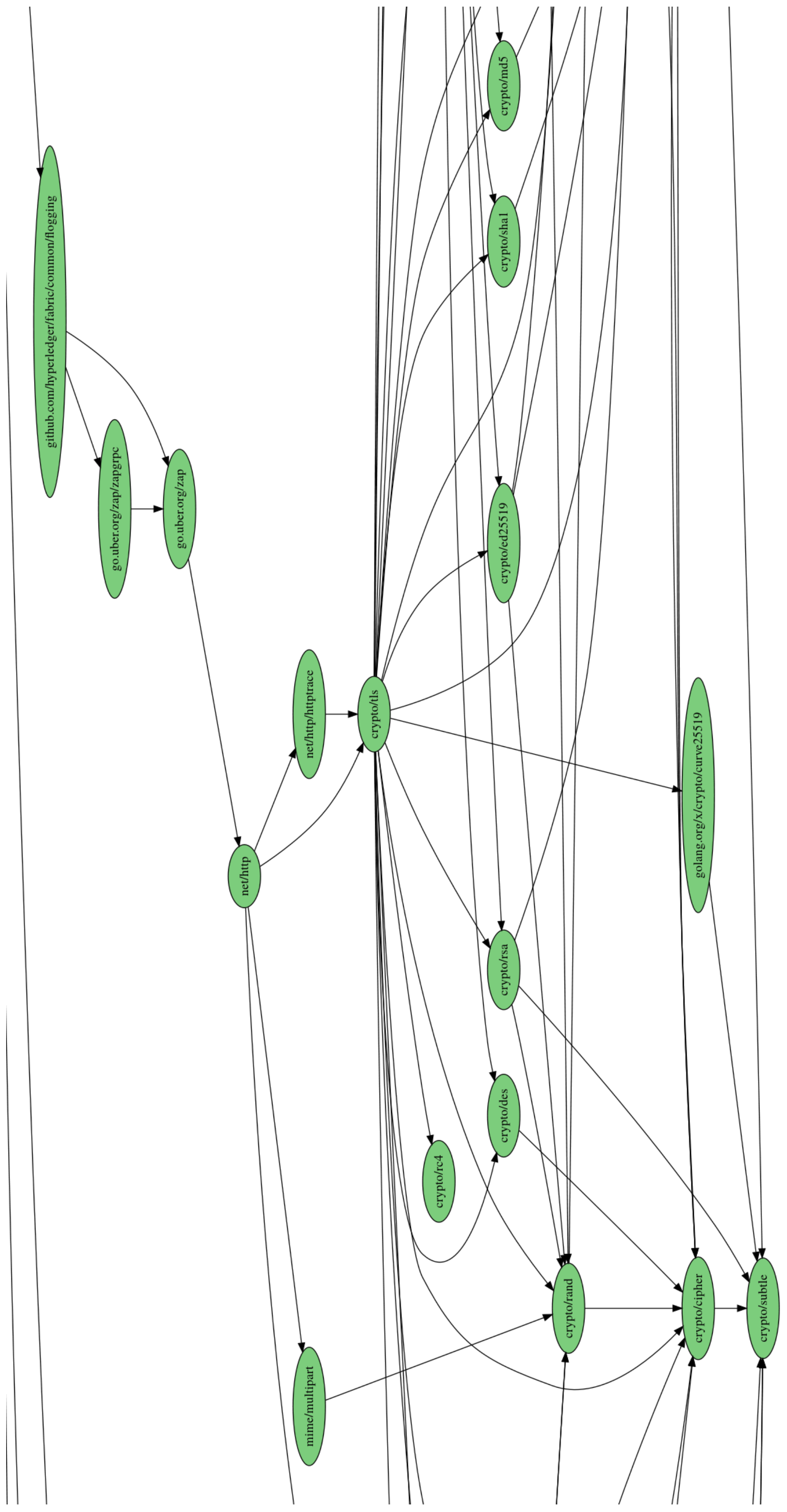


Figure 5.13: Cropped package dependency tree drawn using GraphViz

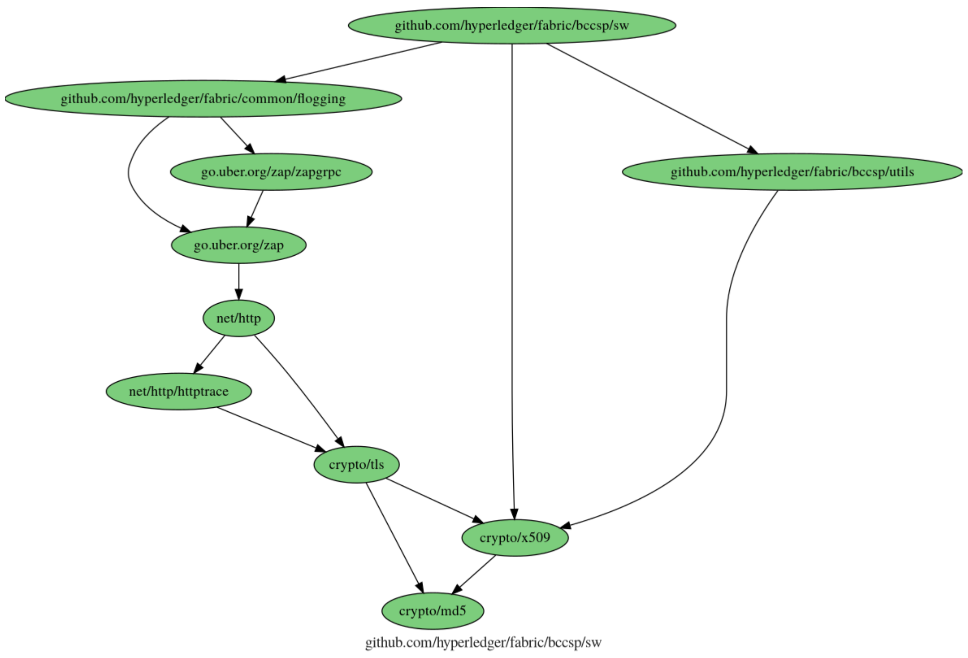

github.com/hyperledger/fabric/bccsp/sw
github.com/hyperledger/fabric/common/flogging
go.uber.org/zap/zapgrpc
github.com/hyperledger/fabric/bccsp/utils
go.uber.org/zap
net/http
net/http/httptrace
crypto/tls
crypto/x509
crypto/md5
github.com/hyperledger/fabric/bccsp/sw

Figure 5.14: Graphviz representation of the package `github.com/hyperledger/fabric/bccsp/sw` filtering only `crypto/md5` usage

cryptographic policy and verify if some source code complies with the policy, this could be useful to assert that a software is compliant with the company policy before being deployed, or that a software complies with local data regulation. Furthermore, this inventory tool can be used for other purposes than a specific cryptographic inventory.

This tool would require a better graphical interface, such as a dashboard and CI/CD integration to be used in production. Having a full dashboard as described in Section 4.5.1 would make the tool easier to work with. The back-end can be kept as it is, and a graphical interface has to be created. Moreover, developing integration of this tool in CI/CD pipelines would allow to automate the inventory making process, and to assert software compliance with a specific policy before it is deployed automatically.

This inventory tool could support other programming languages than Go. To add a support for another programming language, we only need a way to build call graphs in this specific language, and provide the call graph to the inventory tool that will be able to import them, and display them as described. Moreover, it would be interesting to add a support for projects using multiple programming languages, displaying the bridge between the libraries in different programming languages. Run-time analysis can be used for such a purpose. Adding support for run-time analysis could give more preciseness on which dependencies are actually in use, as discussed in Section 4.3. Having an hybrid inventory tool making use of both static source code analysis and run-time analysis would be more accurate than using a single technique. Run-time analysis support can be added using the output of the unix `perf` command.

Chapter 6

Future work

This report defines the steps to reach cryptographic agility. It also presents a novel inventory tool, with the ability to build cryptographic inventories, which is the first essential step on the road to cryptographic agility. However, this is only the beginning of this huge migration, which will require a lot of efforts in the years to come.

As stated in Chapter 5, the inventory tool that was built for this project is more a proof-of-concept than a production ready software. Thus, the first future work would be to improve this tool or build a new one, with for instance a graphical dashboard interface, making it user friendly, and addressing all concerns mentioned in Section 5.6. Moreover, it would be beneficial to have an open source inventory tool implementation with multiple entities contributing to the improvement of the project. Having a diversity of inventory tools is always positive and would allow companies to select the inventory which fits the best their needs.

Then, the major task is to build an agile cryptographic framework, as described in Chapter 3. This task will be very challenging, as there are many constraints that must be satisfied, and questions left open. Developing such a framework will require a team of experienced professionals, and this task should be carried out either by private firms, selling their framework for profit, or by collaborations between entities building an open source framework publicly available. One major challenge to address in this agile cryptographic framework is key management. If the framework is able to offer a simple key management service, it will revolutionize the way cryptography is consumed, as everything could be automated. The developers will not have to be responsible for the choice of cryptographic algorithms, and the handling of cryptographic primitives, such as keys and nonces.

Once the cryptographic inventory and the agile cryptographic framework are ready to be deployed, there will be a huge work to migrate all applications from the legacy use of cryptography, to the agile framework, and the inventory will help in this transition. Awareness should be raised to encourage companies to migrate their software to a cryptographic agile framework, which will ease the migration to post-quantum cryptography. The deadline for the development of the frameworks and the migration, is the arrival date of the quantum computers with the capability of breaking the algorithms currently in use. The sooner the migration will be performed, the better data will be protected.

Chapter 7

Conclusion

This report details the possible milestones on the road to cryptographic agility. The need for cryptographic agility is growing with the arrival of quantum computers. Quantum computers will be able to break most public-key cryptographic algorithms currently in use, using Shor's algorithm [17]. The effects will be disastrous as internet secure communications rely on these algorithms, the security of web browsing, private messaging and online payments will be compromised. They are currently not powerful enough to perform these operations, but they are expected to reach this stage in a near future. The National Institute of Standards and Technology (NIST), is holding a competition to select the successors to the algorithms that will be broken by quantum computers, the post-quantum cryptographic algorithms [1]. NIST is expected to pick the best candidates in the years to come. Once they have published the new post-quantum cryptographic algorithms standard, all software and systems using cryptography vulnerable to quantum computers will have to replace them by the new standards. This migration is expected to be one of the largest code migration in the history of computer science, as it concerns a huge amount of software to be updated.

If this migration was to happen today, it would probably be chaotic. Currently, cryptography is consumed in a static way by most developers, meaning that they directly call the functions exposed by the implementations of cryptographic algorithms. This implies that a replacement of cryptographic algorithms in the source code requires to first, make an inventory of all the cryptographic algorithms in use, and second, manually update the function call to the vulnerable algorithm, to a function call to the new standard. This would also require to adapt the data structures resulting from the replaced algorithm, causing significant changes to the program infrastructure. Furthermore, the new functions have to be used correctly to avoid a cryptographic misuse leading to security vulnerabilities. The chaotic migration from MD5 illustrates the difficulties of a single cryptographic algorithm migration. However the post-quantum migration will have a much wider scale, as many algorithms need to be replaced at once.

Cryptographic agility allows developers to update the use of cryptographic algorithms in software implementation without making significant changes to the software infrastructure. Cryptographic agility would be the perfect solution for a smooth post-quantum migration. However, no cryptographic agile framework is available at the moment. Furthermore, it can provide a flexible security level, a support for cryptographic policy and contextual, composability and implementation agility. The report discusses a possible design for a cryptographic agile framework. Agility could be added through a new layer of indirection, located between the software's source code, and the cryptographic algorithm implementation. This middleware takes as input a cryptographic policy, defining which and how cryptographic algorithms must be used. When the use of a cryptographic algorithm is needed, the source code will call the appropriate function exposed by the middleware, and the middleware will be responsible to select the algorithm to use, and to make use of it. The middleware can be implemented as a dynamic library or as a microservice. The dynamic library offers better performance guarantees, but the microservice provides a higher level of agility. The cryptographic agile framework also has the feature to migrate encrypted data to another encryption scheme, and to provide a cryptographic inventory which can be used to certify compliance with data regulation laws.

In order to be able to make use of a cryptographic agile framework to carry out an automatic migration to post-quantum cryptography, it is first necessary to manually migrate software to agile cryptography. A manual migration is a tedious work, as it requires all function calls to static cryptographic algorithm implementation to be redirected to the agile framework. A cryptographic inventory helps to understand which, where and how cryptography is used, and is convenient to perform a manual migration. Thus, building a cryptographic inventory tool is the first step towards cryptographic agility. Automated cryptography inventory solutions are able to build such inventory, using static source code or run-time analysis. These tools can also be used to certify regulations compliance, assess risk, or list cryptographic primitives in use, such as certificates or keys.

We implemented a simple static source code analysis tool, aiming to build a Software Bill of Material for Go implementations. It recursively scans the dependencies of a Go project, and builds its dependency graph. The dependency graph is displayed in a Terminal User Interface and is navigable and interactive. The dependency graph can also be exported as an interactive chord diagram, a Graphviz dependency graph image or as a JSON text file. Specific packages can be filtered in order to build a very specific inventory. Thus, filtering all the cryptographic components provides a full cryptographic inventory. The filter and the reverse dependency tree features provide details on how specific cryptographic algorithms are used. It also shows the components of a specific project making use of these algorithms, which is convenient to discover dependency on deprecated algorithms, as demonstrated.

The road map to agile cryptography development consists in three steps. The first one is to develop convenient tools producing Software Bill of Material for arbitrary projects, as described in Chapter 4. The cryptographic inventory can be extracted from this Software Bill of Material. The second step is to build a cryptographic agile framework as described in Chapter 3. This framework would give the ability to seamlessly migrate the cryptography in use at a project or company level. Once the inventory tool and cryptographic agile framework are out, the last step is to advertise the use of these tools for post-quantum cryptography migration, in order to avoid a chaotic manual migration.

The milestones for the migration to post-quantum cryptography for the industry are the following. The first milestone is to build a company-wide cryptographic inventory, or individual cryptographic inventories for each project. Once this is done, engineers must concurrently update the source code from the legacy use of cryptography to an agile cryptographic framework, and migrate their databases accordingly. After this first migration is completed, the migration to post-quantum cryptography can be automatically performed from the cryptographic agile framework. This framework will then be used to automate future migrations, policy compliance, and key management, thus simplifying the consumption of cryptography.

This project presented the strategy to reach cryptographic agility, and pointed out how cryptographic agility can offer a fast and organized migration to post-quantum cryptography.

Bibliography

[1] Gorjan Alagic, Jacob Alperin-Sheriff, Daniel Apon, David Cooper, Quynh Dang, John Kelsey, Yi-Kai Liu, Carl Miller, Dustin Moody, Rene Peralta, Ray Perlner, Angela Robinson, and Daniel Smith-Tone. “Status Report on the Second Round of the NIST Post-Quantum Cryptography Standardization Process”. In: (July 2020). DOI: `10.6028/NIST.IR.8309`.

[2] Mirko Amico, Zain H. Saleem, and Muir Kumph. “Experimental study of Shor’s factoring algorithm using the IBM Q Experience”. In: *Physical Review A* 100.1 (July 2019). ISSN: 2469-9934. DOI: `10.1103/physreva.100.012305`. URL: `http://dx.doi.org/10.1103/PhysRevA.100.012305`.

[3] F. Arute, K. Arya, R. Babbush, et al. “Quantum supremacy using a programmable superconducting processor”. In: *Nature* 574.7779 (Oct. 2019), pp. 505–510. ISSN: 1476-4687. DOI: `10.1038/s41586-019-1666-5`.

[4] Charles H. Bennett, Ethan Bernstein, Gilles Brassard, and Umesh Vazirani. “Strengths and Weaknesses of Quantum Computing”. In: *SIAM Journal on Computing* 26.5 (Oct. 1997), pp. 1510–1523. ISSN: 1095-7111. DOI: `10.1137/s0097539796300933`. URL: `http://dx.doi.org/10.1137/S0097539796300933`.

[5] Lidong Chen, Stephen P. Jordan, Yi-Kai Liu, Dustin Moody, Rene C. Peralta, Ray A. Perlner, and Daniel C. Smith-Tone. *Report on Post-Quantum Cryptography*. NIST, Apr. 2016. DOI: `10.6028/NIST.IR.8105`.

[6] Cryptosense. *Cryptographic Inventory Whitepaper v1.0*. Feb. 2020.

[7] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, and J. Alex Halderman. “The Matter of Heartbleed”. In: *Proceedings of the 2014 Conference on Internet Measurement Conference*. IMC ’14. Vancouver, BC, Canada: Association for Computing Machinery, 2014, pp. 475–488. ISBN: 9781450332132. DOI:

10.1145/2663716.2663755. URL: https://doi.org/10.1145/2663716.2663755.

[8] Adam Everspaugh, Rahul Chaterjee, Samuel Scott, Ari Juels, and Thomas Ristenpart. "The Pythia PRF Service". In: *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, Aug. 2015, pp. 547–562. ISBN: 978-1-939133-11-3. URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/everspaugh.

[9] Jay Gambetta. *IBM's Roadmap For Scaling Quantum Technology*. Sept. 2020. URL: https://www.ibm.com/blogs/research/2020/09/ibm-quantum-roadmap/.

[10] InfoSec Global. *Hunting for weak Crypto Whitepaper*. https://www.infosecglobal.com/resource/agilescan-whitepaper.

[11] GnuPG. *The 'GnuPG Made Easy' Reference Manual*. https://www.gnupg.org/documentat Oct. 2020.

[12] Yong Li, Yuanyuan Zhang, Juanru Li, and Dawu Gu. "iCryptoTracer: Dynamic Analysis on Misuse of Cryptography Functions in iOS Applications". In: *Network and System Security*. Ed. by Man Ho Au, Barbara Carminati, and C.-C. Jay Kuo. Cham: Springer International Publishing, 2014, pp. 349–362. ISBN: 978-3-319-11698-3.

[13] D. McMahon. *Quantum Computing Explained*. Wiley - IEEE. Wiley, 2007. ISBN: 9780470181362. URL: https://books.google.ch/books?id=bDXwFHJNKFAC.

[14] David Ott, Christopher Peikert, and other workshop participants. *Identifying Research Challenges in Post Quantum Cryptography Migration and Cryptographic Agility*. 2019. arXiv: 1909.07353 [cs.CY].

[15] Ronald Rivest and S Dusse. *The MD5 message-digest algorithm*. 1992.

[16] Shodan. *[2019] Heartbleed Report*. https://web.archive.org/web/20190711082042/https://www.shodan.io/report/0Wew7Zq7. Archive from: 2019-07-11. 2019.

[17] P. W. Shor. "Algorithms for quantum computation: discrete logarithms and factoring". In: *Proceedings 35th Annual Symposium on Foundations of Computer Science*. 1994, pp. 124–134. DOI: 10.1109/SFCS.1994.365700.

[18] Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, and Yarik Markov. "The First Collision for Full SHA-1". In: *Advances in Cryptology – CRYPTO 2017*. Ed. by Jonathan Katz and Hovav Shacham. Cham: Springer International Publishing, 2017, pp. 570–596. ISBN: 978-3-319-63688-7.

[19] Xiaoyun Wang and Hongbo Yu. “How to Break MD5 and Other Hash Functions”. In: *Advances in Cryptology – EUROCRYPT 2005*. Ed. by Ronald Cramer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 19–35. ISBN: 978-3-540-32055-5.

[20] Stephen Wiesner. “Conjugate Coding”. In: *SIGACT News* 15.1 (Jan. 1983), pp. 78–88. ISSN: 0163-5700. DOI: `10.1145/1008908.1008920`. URL: `https://doi.org/10.1145/1008908.1008920`.