Lookey: Building a Public Key Database to Discover the Origin of Compromised Private Keys

Guðjón Ingi Valdimarsson Halla Margrét Jónsdóttir
Ísól Sigurðardóttir

Thesis of 12 ECTS credits submitted to the Department of Computer Science at Reykjavík University in partial fulfillment of the requirements for the degree of Bachelor of Science

May 13, 2022

Thesis Committee:

Dr. Gylfi Þór Guðmundsson, Supervisor
Adjunct, Reykjavik University, Iceland

Hjalti Magnússon, Advisor
Senior Security Engineer, Syndis, Iceland

Níels Ingi Jónasson, Advisor
Security Engineer, Syndis, Iceland

Dr. Grischa Liebel, Examiner
Assistant Professor, Reykjavik University, Iceland

Lookey: Building a Public Key Database to Discover the Origin of Compromised Private Keys

Guðjón Ingi Valdimarsson, Halla Margrét Jónsdóttir, Ísól Sigurðardóttir

May 13, 2022

Abstract

The lack of security online has been a growing concern in recent years with companies and individuals having a lot to lose in the case of a security breach. With increasing cyber security concerns, many services have opted for more secure methods of authentication than simple usernames and passwords. One such approach is using asymmetric cryptographic keys. Individuals tend to be more concerned with their passwords getting leaked than their private keys. This is of great concern since these keys have appeared in data leaks, and have been used for malicious purposes, e.g. to gain unauthorized access to sensitive data. However, given a private key found in a leak without relevant metadata, discovering the context of it is no trivial task. In this paper we aim to answer the research question: *What are the challenges in creating a system capable of identifying the owner and uses of a cryptographic private key, given only the key itself?* We evaluate methods of gathering and storing vast amounts of public keys along with their relevant metadata, while offering a fast lookup to match a leaked private key to a stored public key. To perform the lookup, we created Lookey: a system that uses the derived public key of a given private key to get information about the key's use or context. By using Lookey, one can possibly find and notify the owner of a leaked private key.

Lookey: Gerð dreifilykilsgagnagrunns til auðkenningar á eigendum útsettra einkalykla

Guðjón Ingi Valdimarsson, Halla Margrét Jónsdóttir, Ísól Sigurðardóttir

13. maí 2022

Útdráttur

Undanfarin ár hefur fjöldi netglæpa aukist gríðarlega en sú þróun er vaxandi áhyggjuefni þar sem fyrirtæki og einstaklingar hafa sífellt meiru að tapa ef þeir lenda í öryggisbrestum. Með aukinni meðvitund um mikilvægi öryggismála hafa margar þjónustur valið öruggari aðferðir til auðkenningar en hin hefðbundnu notendanöfn og lykilorð. Ein slík aðferð er að notast frekar við dreifilykla og einkalykla. Einstaklingar eiga það til að hafa meiri áhyggjur af því að lykilorð þeirra leki heldur en einkayklar þeirra. Í undanförnum lekum hefur hins vegar borið á því að einkalyklar séu þar á meðal og því þarf fólk að vera meðvitað um að þeir geti einnig lekið. Þar sem óprúttnir aðilar geta, og hafa, misnotað slíka lykla, t.d. til að komast yfir viðkvæm gögn, er afar mikilvægt að tryggja öryggi þeirra. Ef einkalykill finnst í leka er nær ómögulegt að vita í hvað hann er notaður ef engar nánari upplýsingar um hann eru fyrir hendi. Í þessari skýrslu stefnum við á að svara rannsóknarspurningunni: *Hvaða áskoranir koma upp við gerð kerfis sem býður upp á að bera kennsl á eiganda einkalykils ef engar nánari upplýsingar um lykilinn eru til staðar?* Til að takast á við þetta munum við bera saman aðferðir við að safna og geyma mikið magn dreifilykla í gagnagrunni með viðeigandi lýsigögnum, ásamt því að styðja hraðvirka leit að dreifilykli sem samsvarar gefnum útsettum einkalykli. Fyrir leitina bjuggum við til kerfið Lookey, sem tekur inn útsettan einkalykil og finnur samsvarandi dreifilykil og lýsigögn, gefið að dreifilykillinn sé í gagnagrunninum. Með notkun Lookey er möguleiki á að finna eigendur lekinna einkalykla og upplýsa þá um lekann.

We dedicate this to our families.

Acknowledgements

"Security is a journey, not a destination."

- Hjalti Magnússon

We would like to especially thank Dr. Gylfi Þór Guðmundsson for his invaluable support and guidance over the semester. Always ready to help us and tell us stories, some relative to the project and others not, though always resulting in a lesson or a good laugh.

Syndis, thank you for the the collaboration, providing us with an office space and lunches for the last three weeks of the project, a very interesting cyber security conference and of course the security socs. Níels Ingi Jónasson and Hjalti Magnússon, thank you for your valuable consultation and advisement throughout this project.

We want to give special thanks to all our families, for their unconditional support and teaching us the meaning and importance of education. If it were not for you, we would not be where we are today.

Last but not least, Davíð Stefán Reynisson and Guðrún Valdís Jónsdóttir, we want to thank you for your constant help, nit-picking and taking your time in reviewing this paper. We greatly appreciate it.

Contents

List of Figures

List of Tables

List of Abbreviations

BSc	Bachelor of Sciences
BC	Before Christ
CLI	Command Line Interface
DBMS	Database Management System
DH	Diffie-Hellman
DSA	Digital Signature Algorithm
ER	Entity Relationship
GiB	Gibibyte
MiB	Mebibyte
KiB	kibibye
ms	Milliseconds
RSA	Rivest-Shamir-Adleman
SQL	Structured Query Language

Chapter 1

Introduction

1.1 Overview

Security has always been a foundation on which our society is built. After the Internet became a big part of many people's lives, people have become more and more aware of the importance of cyber security and the privacy of their data. In recent years, many data leaks have occurred that exposed private and sensitive information [1, 2]. As a result of this development, people are highly concerned with their passwords getting leaked an rightly so. This has caused many tech-savvy people to resort to using asymmetric cryptographic keys rather than passwords, but they are used for encrypting and decrypting data, and for authentication among other things [3]. People are not as concerned about their asymmetric cryptographic keys getting leaked as they are with their passwords. This is of considerable concern as private asymmetric cryptographic keys are among the information one frequently sees in data leaks, along with e-mails, usernames, passwords etc. The leaking of private keys is as bad, in some cases even worse than the leaking of passwords. If the relevant information about the leaked key is at hand, private information and/or systems they are meant to secure become compromised and the victim can be impersonated online. However, if no additional information is available, deriving the key's use or ownership is no trivial task. This is due to the structure of cryptographic keys being a string of random characters with no relevant information about them, e.g. the owner of the keys. With no easy access to information about leaked private keys, finding the owner of the key to inform them of their leaked key is a near impossible task. For perspective, imagine finding a random key in the middle of Times Square. Without any further context, it would be incredibly hard to know whom it belongs to or what lock it unlocks. Imagine having a machine that knows how all the locks in New York City look like, such a machine would make the pairing of a key to its lock a possible task. This is what our system, Lookey, aims to achieve for asymmetric cryptographic keys.

Hence, the goal of making the Lookey lookup tool is to provide an answer to the following question:

> *What are the challenges in creating a system capable of identifying the owner and uses of a cryptographic private key, given only the key itself?*

In order to achieve this goal, asymmetric public keys, along with their metadata, will be scraped from websites and stored in a database. Other services that offer lookup of keys or certificates will be investigated in order to utilise them to increase the chance of finding a match for the private key. To perform the lookup, a tool will be created that allows deriving a public key from the private key.

1.2 Data leaks and consequences

As mentioned above, many data leaks have occurred in recent years. Malicious hacker groups are mostly responsible for these leaks, although leaks can also happen accidentally. As discussed in [4], thousands of cryptographic and API keys are unintentionally leaked everyday in public GitHub repositories by their owners. These secrets are leaked through the source code, as files in the repository or in configuration files. Exposing your private cryptographic key can have severe consequences. If a malevolent hacker group obtains one important private key, they can e.g. obtain unauthorised access to sensitive data and cause a lot of harm. When searching the Internet for big security breaches caused by stolen or leaked private keys, numerous incidents where found but three of them stood out. The companies/institutions involved were BitMart, The European Union (EU) and Mt. Gox.

BitMart: On December 4th 2021, the cryptocurrency trading platform BitMart alerted their users of an extensive security breach that occurred earlier that same day [5]. Two of their hot wallets (online cryptocurrency wallets) were compromised and the hackers responsible were able to steal about $150 million worth of assets. On December 6th, the BitMart Team updated their users on the cause of this security breach, namely that a private key had been stolen and used to compromise the two hot wallets.

The European Union: In late 2021 a private key from the EU, used to sign Digital COVID-19 certificates, was leaked. The key had been in circulation on online data breach marketplaces and messaging applications. The certificates, which the key was used to sign, allowed residents in the EU to travel outside of their country during the pandemic if certain requirements had been met. Some people even went so far as to create false certificates for people who had died or have never existed, e.g. for Adolf Hitler and Mickey Mouse, with the leaked private key [6].

Mt. Gox: Mt. Gox, founded in 2010, was one of the largest Bitcoin exchange in the world [7]. In 2011 the company was attacked where the attackers gained unauthorised access to an auditor's computer. In the attack the perpetrators changed the listed Bitcoin price to just 1 cent. Later on, in 2014, the company went bankrupt after a massive attack on their systems, where around $460 million worth of Bitcoin were stolen. When investigating the 2014 attack, investigators discovered that a private key had been stolen back in 2011. It is unknown whether the key was stolen through a cyber attack or with insider help.

Chapter 2

Background and related work

This chapter covers the necessary background material a reader needs to know in order to understand the context of this paper.

2.1 Cryptography

The best known historical use of cryptography is most likely by Gaius Julius Caesar (Rome, 101 BC - 44 BC), when he frequently used a substitution cipher in military applications [8]. However, as stated in [9], cryptography dates back to 1900 BC, when non-standard hieroglyphs were used in inscriptions in Egypt. With the computational power we have today, cryptographic algorithms like the ones used in Egypt and Rome are extremely easy to crack [8]. Today, cryptographic algorithms are far more secure and cracking them is computationally hard, if not impossible [9].

Cryptography is used e.g. to secure data and communications over networks, including the Internet, and for user authentication.

As discussed in [9], there are three main types of cryptographic algorithms: secret key cryptography, also known as symmetric key encryption; public key cryptography, also known as asymmetric key encryption; and hash functions. Of these three, only the asymmetric key cryptography is relevant to this project, which we will refer to as asymmetric key encryption for the rest of this paper.

2.1.1 Cryptographic keys

When encrypting and decrypting data, some secret values are needed which are usually referred to as keys [10]. Keys are strings of random characters following a certain format based on the cryptographic algorithm. Keys are used to encrypt data to make it unreadable and difficult to decrypt for unauthorised readers. Keys also allow authorised readers to decrypt the encrypted data. Cryptographic keys get their name from regular physical keys, since they encrypt data like regular keys lock locks, and then decrypt data like physical keys unlock locks.

Symmetric key encryption is when only one secret key is used to both encrypt and decrypt the data. Asymmetric key encryption is when a pair of keys are used for encryption and decryption, a private key and a public key.

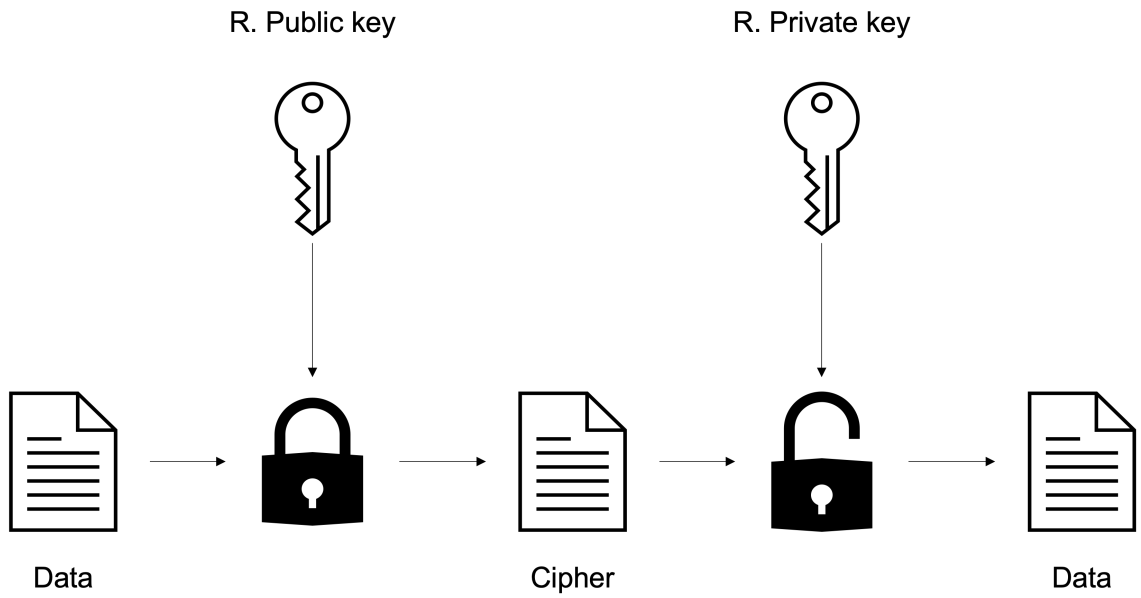

Figure 2.1: Demonstration of data encryption and decryption using asymmetric key encryption. R. indicates the receiver. The data is encrypted with the receiver's public key, resulting in a cipher which is sent to the receiver. The cipher is then decrypted with the receiver's private key.

2.2 Asymmetric key encryption

Asymmetric key encryption utilises a pair of keys, a public and a private key. The sender uses the receiver's public key to encrypt data, resulting in a cipher. The sender sends the cipher to the receiver, who uses their own private key to decrypt the cipher, resulting in the original data (see Figure 2.1). This way, the sender and the receiver are the only ones that are able to know the content of the data, given that the receiver's private key has not been compromised [11].

The receiver still needs to verify the sender and for that we have digital signatures. The data to be sent is hashed and the resulting hash is encrypted with the sender's private key, resulting in a digital signature. The sender sends the digital signature, their public key and the data to the receiver. This way, the receiver can apply the same hashing function on the data and use the sender's public key to decrypt the signature. Both result in hashes and now the receiver can verify the sender. If the hashes are equal the sender has been authenticated and the receiver can be sure that no one tampered with the message [3]. See Figure 2.2 and 2.3 for the signing and verification process, respectively.

To begin with, some preliminary research of common algorithms was conducted by looking at algorithms used on websites and in tools such as SSH. The most common asymmetric encryption algorithms noticed in the aforementioned research were Rivest-Shamir-Adleman (RSA), Digital Signature Algorithm (DSA), Diffie-Hellman (DH), ED25519 and derivations of these. From now on in this paper, these algorithms will be referred to by using their abbreviations.

2.3 Key formats

Asymmetric cryptographic keys can be of various algorithms and formats. As mentioned in the previous chapter, some of the most common key algorithms found were RSA, DSA, DH and

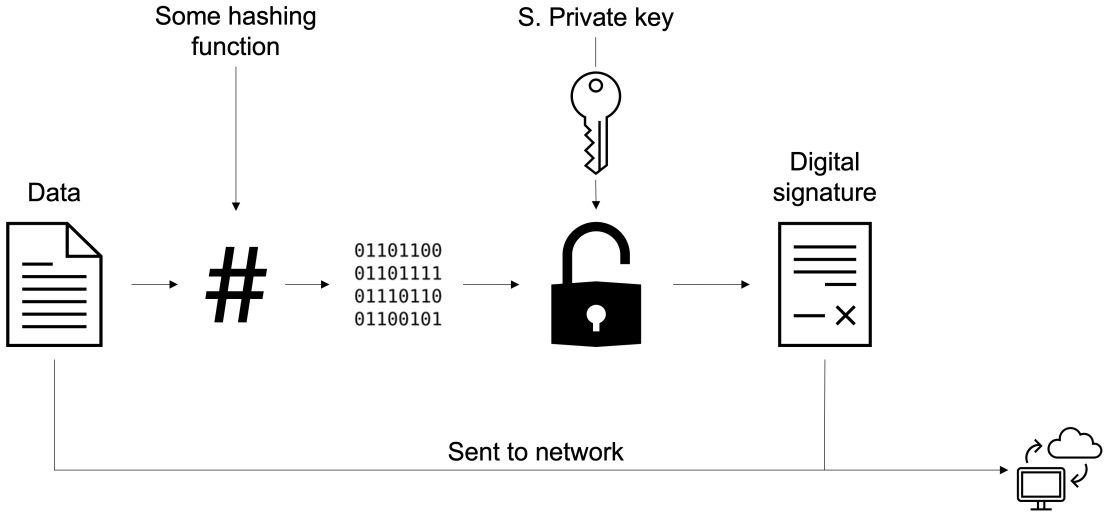

Figure 2.2: Digital signature - signing. S. indicates the sender. The data is hashed and the hash value is encrypted with the sender's private key to create a digital signature. The data and the signature are then sent to the receiver.

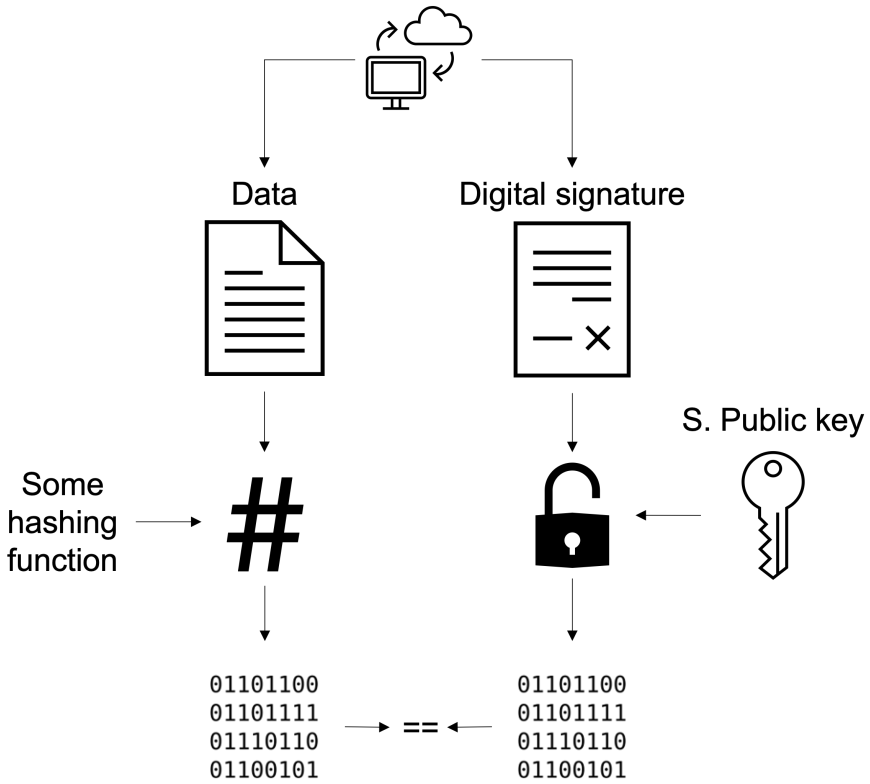

Figure 2.3: Digital signature - verification. S. indicates the sender. The data is hashed with the same hash function as was used for the signing. The digital signature is decrypted using the sender's public key, resulting in a hash. The hashes are then compared, if they are equal, the sender has been authenticated.

ED25519. These are all key algorithms that are used to generate different private and public key components and can be stored in various formats.

Two of the most used formats are the PEM format as described by [12] and [13], and the DER format as described in [14]. The PEM format is more familiar to users as it contains human readable text to identify the key or the digital certificate. Digital certificates are composed of a statement and a digital signature of the statement [15]. The PEM format contains two lines that define the beginning and end of the data, that is base64 encoded in between. An example of this key format is:

```
-----BEGIN PUBLIC KEY-----
base64 encoded data
-----END PUBLIC KEY-----
```

The DER format is a binary format that is used to encode a key or certificate. It is the original binary data that is base64 encoded in the PEM format. A more explicit example of a key in the PEM format is:

```
-----BEGIN PUBLIC KEY-----
base64(DER format)
-----END PUBLIC KEY-----
```

Knowing these formats exist is important for correctly identify a key for a program to derive the corresponding public key. In addition to the PEM and DER formats, two more formats are of special interest for this project. Those are the OpenSSH format used for SSH keys and X.509 certificates that are used for SSL and TLS certificates on the web. The following two chapters cover the details of these formats.

2.3.1 SSH keys

SSH (secure shell) is a network protocol that allows a user to communicate securely over an unsecured network and is predominantly used to gain command-line access on a remote machine. In SSH, asymmetric cryptographic keys are used to secure communications back and forth as well as to allow for authentication with keys instead of passwords [16].

For this project, the authentication part is more relevant since numerous websites and services now prefer authentication via SSH keys over password authentication. As an example, GitHub allows users to login with usernames and passwords for their website but requires SSH keys or access tokens for API and git authentication [17].

SSH uses the same key algorithms as other approaches that have been discussed and supports the most common formats. OpenSSH [18] generates SSH keys of a different format than the standard PEM format. SSH keys such as the ones generated by OpenSSH [18] are quite different then the standard PEM format. There is little to no information about the keys other than that the private key component is a PEM encoded key that contains both the private key and the public key. This special format is not a problem since it can be read into tools like OpenSSL, although it might require some special work in some cases. The public key is represented in a different way than the standard PEM format as listed above. The format of the public key is as follows:

```
type public-key [comment]
```

First is the type of the key, second is the public key component and third is an optional comment.

An example of a public key is:

```
ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAINGT4d2ggpB2Z3cLwFFnw
jVnOOUpBdi3FCdJdQi2K+Mc
```

Here, the algorithm is ssh-ed25519 and the key has no comment. This format of public keys is the OpenSSH standard for public keys. There is little to no information about these formats online, yet based on our preliminary analysis on existing literature, this format seems to be the one used today. There was another standard used before, described in [19], but based on our observations it seems to no longer favored in the OpenSSH community.

2.3.2 X.509 certificates

The X.509 certificates are used for the TLS and SSL protocols to encrypt data to and from a website that a user visits. Most users associate it with HTTPS, which extends the standard HTTP protocol with TLS or SSL. Anyone can see the X.509 certificates of websites that have HTTPS by clicking on the lock sign next to the URL in most browsers.

X.509 is simply a wrapper around any public key infrastructure like the ones that have been discussed earlier with RSA or ED25519 asymmetric cryptographic keys. The certificate contains fields that allow the user (or browser, in most cases) to verify that the website being visited is in fact the website it claims to be.

As described in section 4 of [14], which describes the X.509 v3 certificate, it contains a lot of fields to ensure that it is valid and authentic. The field Subject Public Key Info, described in section 4.1.2.7 of [14], contains the public key component of the certificate. This field will come in handy for the purpose of this project, since it will allow us to query public keys of X.509 certificates.

2.4 Certificate Transparency

Exploring the Internet without X.509 certificates would neither be fun nor secure. As mentioned in Chapter 2.3.2, such certificates give us secure access to websites with HTTPS. One of the problems with X.509 certificates is knowing which certificates are valid or falsely generated. Attackers often generate false certificates in an attempt to trick users since anyone can create and sign their own certificates. That is why Certificate Authorities (CAs) exist. CAs are trusted third parties that verify that a certificate is the correct and valid certificate for that particular website. This is similar to getting a referral from a trusted friend about a new employee. An example of such a CA is *Let's Encrypt* which is a free, automated, open and widely used CA announced in 2014 [20]. Before *Let's Encrypt* existed, getting TLS and SSL signed certificates was usually expensive and a tedious process. *Let's Encrypt* decided to give anyone that wanted a signed certificate for their website the service free of charge, greatly increasing the security of the Internet.

As of this writing, there are a total of 53 different trusted CAs that come with the Mozilla Firefox web browser [21]. Nonetheless, CAs are not without fault. This trust has been compromised before as mentioned in [22], where a certificate for Google was signed by a compromised CA, leading to a man-in-the-middle attack (MITM) against users in Iran. A MITM attack is when a malicious third party infiltrates and manipulates a communication between two parties that trust they are only communicating with each other [23].

This threat led to the creation of Certificate Transparency (CT) by Google. CT maintains an append-only log of certificates issued by CAs. This results in CT verifying and preventing

duplicates of issued certificates. Anyone can query the logs to see information about previously issued certificates. Monitors run against these logs to monitor and keep them correct and can alert site owners of fraudulent certificates issued for their domains [24]. The image in Figure 2.4 shows the process of requesting a new certificate from a CA and how it gets added to CT before being given to the domain owner.

2.4.1 Certificate Search

As mentioned in the previous chapter, anyone can query the logs that CT maintains, which means that valuable information can be obtained from CT. This represents a possible avenue of interest for finding information about leaked private keys since querying the CT logs with the corresponding public key might return a certificate the public key belongs to.

One option would be to implement our own monitor or program to get all the logs but we instead decided to utilise an already existing solution. Certificate Search, [25], offers querying CT logs for certificates based on different search parameters and was utilised since it satisfied our requirements. As described in Chapter 2.3.2, the X.509 certificates contain a Subject Public Key Info field containing the public key of the certificate. The sha256 or sha1 sum of this field can be utilised to query Certificate Search to find the certificate of that given public key. This offers looking up the public key component of a leaked private key on Certificate Transparency.

2.5 Driftwood

As part of our due diligence, research on previous work in this field was conducted. No notable papers, articles or other research into storing vast amounts of cryptographic keys for fast lookups were found. The most relevant work found was Driftwood by Truffle Security [26].

The Driftwood tool performs lookups on their own database of pubic keys. As stated in [27], the public keys in their database pair with sensitive private keys and that the private key will never leave where you run the tool. Driftwood utilised GitHub and Certificate Transparency to create a database of public keys.

Their database and documentations are not public so there is no information about the validity of the data they have. As of writing this, Driftwood has not been updated since November 2021 as of writing this report and the database querying is not public. Hence, there is not sufficient information about the tool or how it was implemented.

Our solution described in this paper will go beyond the capabilities of Driftwood. The system we propose has more optimised searching and the approach and tools developed are documented.

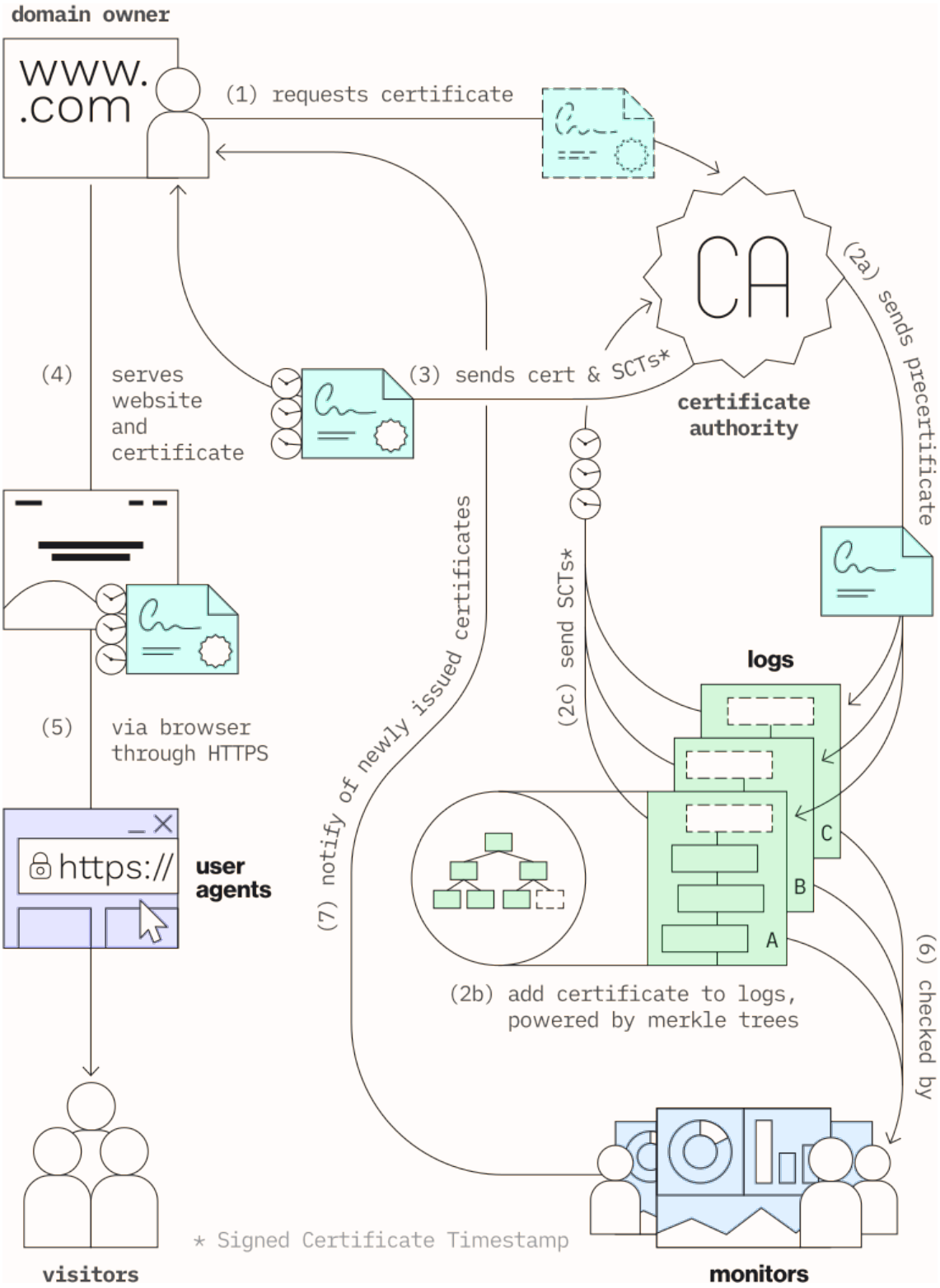

Figure 2.4: Diagram from Certificate Transparency showing the flow of a certificate request from a domain owner to a CA and how the new certificate gets added to the append only logs and monitored by the Monitors before being given back to the domain owner. Image retrieved from `https://certificate.transparency.dev/howctworks/img/with-ct-mix.svg`

Chapter 3

Design and implementation of Lookey

Lookey consists of three separate parts: a database, which stores public keys and their relevant metadata; a scraper, which collects the public keys and their relevant metadata; and a lookup tool, which searches for the corresponding public key to the compromised private key. If there is a match, the owner of the key pair is returned. These parts are all connected to form a complete solution, resulting in the lookup system Lookey.

3.1 The Database

The Lookey database needs to store vast amounts of public keys and their metadata. This requires a well designed database. It needs to hold all relevant information about the keys as well as allowing data to be queried in a reasonable time. A lot of thought, testing and decision making went into selecting the correct database and data structures for this project's needs.

The database of choice needs to satisfy a few base criteria which were specified for the task at hand. First, the database needs to be able to store all necessary data in a non-volatile form, such as on a hard disk or solid state disk. Secondly, it needs to be able to find a single public key quickly, with a direct matching query while returning the metadata of the key in question. These two base criteria are based on the intended use case of the tool: to attempt to match a single private key to the owner of the corresponding public key in our database. Consequently, the database does not need to handle any range queries or any complex or time consuming queries.

3.1.1 Database management system options

With these criteria specified, we began researching what database management system (DBMS) would satisfy our needs. Different kinds of DBMSs were considered such as: relational, key-value stores, document- and wide-column DBMSs. In this project's case, two kinds of DBMS fit our criteria best: relational DBMS and key-value stores.

Relational databases come from a time where data storage was expensive, while computing was relatively cheap. This prompted the invention of the relational model by Edgar F. Codd in 1970, which is what relational DBMSs are based on today. Relational DBMSs are well suited for complex access patterns, where data needs to be be queried in any way, shape or form [28].

Another option was to use a key-value store, which stores data in simple tuples of a key field, which can be searched for, and a values field, which holds the actual information. Key-value stores are highly scalable and the key searching is faster than when querying relational DBMSs. The disadvantage of key-value stores is the inability to query databases like relational DBMSs

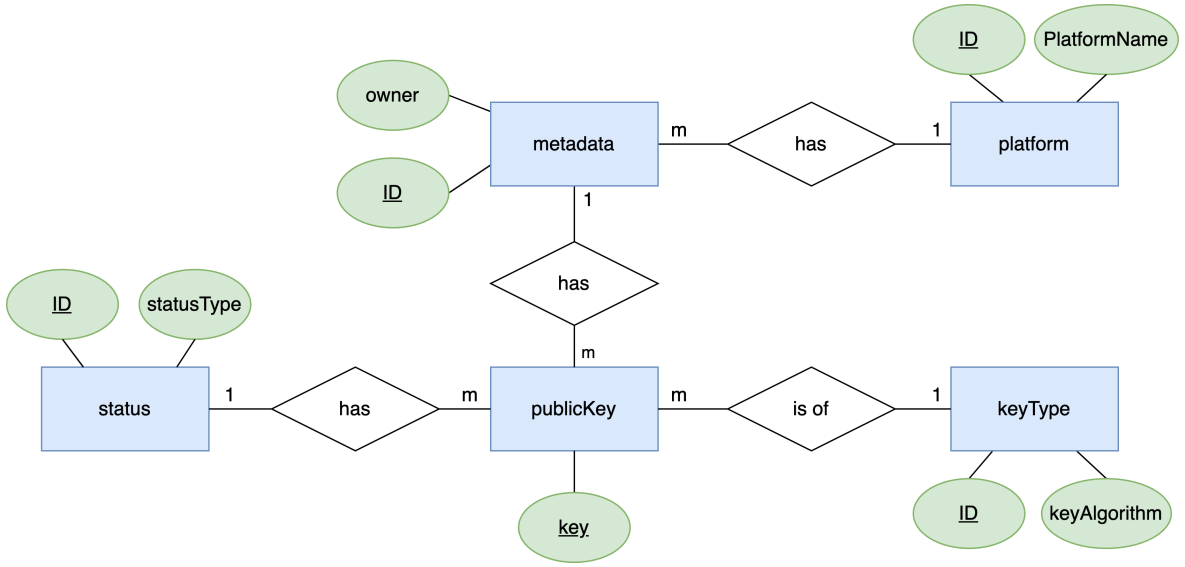

Figure 3.1: First version of the ER-diagram of the PostgreSQL database Lookey

do. Key-value stores need to store the same information multiple times, such as the platform a key is used on [29], whereas relational DBMSs use relations to avoid data redundancy.

With these two DBMSs in mind, two database systems were considered: PostgreSQL and Redis on Flash. PostgreSQL is an open-source and highly customisable relational DBMS [30]. Redis, on the other hand, is a fast and popular key-value store that works in-memory. Redis provides a solution called Redis on Flash which offers storing parts of the data in non-volatile storage. This solution is an enterprise solution and, therefore, requires payment [31].

3.1.2 Design of the Lookey database

After deciding which DBMSs would be tested, the design process began [32].

When designing a relational database, ER-diagrams are an excellent way to get an overview of the structure of the database during the design process. All the entities that become tables in the database and the relationship between them, come into consideration when creating a diagram like this.

The Lookey database needs to store public keys, their cryptographic algorithm, their owners and on which platforms they are used. The first design of the database kept track of the keys' status, indicating whether they are active or inactive on the platform they are used on. The field storing the public keys needed to allow for both small and large amounts of data and thus the type structure chosen was bytearray. The tables in the first version of the database were publicKey, metadata, platform, keyType and status. The PostgreSQL database was implemented first from the ER-diagram, using the PostgreSQL version 14.2 [33]. After the implementation, testing was performed on the database with mock data as will be described in Chapter 3.1.4. The first draft of the ER-diagram of the PostgreSQL database can be seen in Figure 3.1.

After testing the initial implementation of the relational database, improvements on the design were made. The relation between the Metadata and PublicKey tables was changed to a many-to-many relationship. It was more appropriate in this case as each owner can own multiple public keys and each public key can be used on many platforms. This resulted in a new table, KeyMetadata. The initial version of the database had the default B-tree index for the public key. A B-tree index facilitates faster queries that contain the comparison operators <, <=, =, >=, >, while a hash index only offers support for the operator =, but even faster. Since a key would

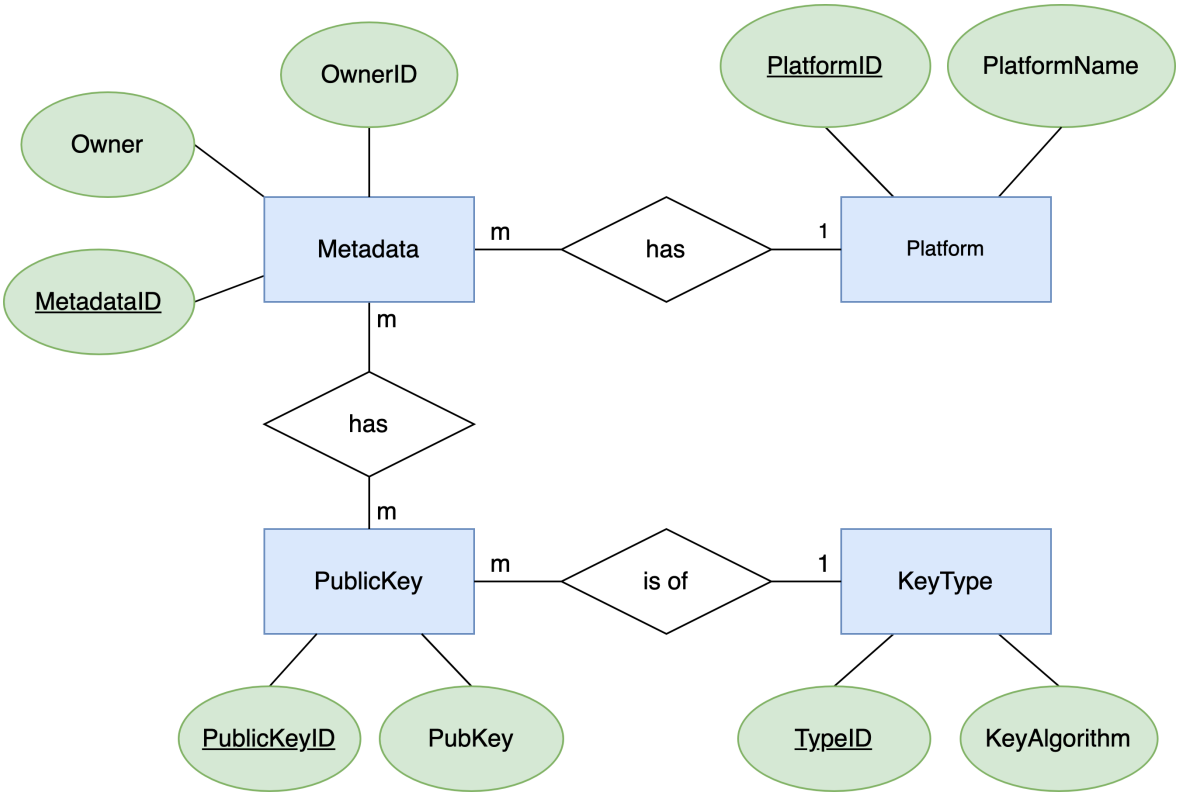

Figure 3.2: Final version of the ER-diagram of the PostgreSQL database Lookey

never be queried in any other way than with a direct match, a hash index was chosen in the final design. The primary key was changed from being the public key into a serial id instead to facilitate a smaller foreign key footprint. As mentioned in Chapter 3.1.4, the hash index resulted in a huge improvement to the table overhead, as it stores a lot less metadata than a B-tree index [34].

The tables in the final version of the PostgreSQL were PublicKey, Metadata, KeyMetadata, Platform and KeyType. The final version of the ER-diagram of the PostgreSQL database can be seen in Figure 3.2.

No need was for designing a Redis database since it is a key-value store, and does not need to be designed.

3.1.3 Generation of test keys

To determine which database option fit our needs better, we generated a set of test keys to test each database on fair grounds. A CLI tool was written in the Rust programming language, which offers fast and memory safe low-level programming [35]. The speed of Rust is well suited for key generation since generating some types of keys is quite time-consuming, as can be seen in Table 3.1.

The key set was generated utilising the OpenSSL library, which is a cryptography library with support for a lot of different key algorithms, such as the ones mentioned in Chapter 2.2 [36].

The tool generated a total of 5,100,000 keys using 6 different key algorithms and key sizes, giving a diverse set to test on. Table 3.1 shows information about the generation process, such as how many keys were generated, how time-consuming it was, and how big the test set turned out to be. This gave a better understanding of the complexity behind each algorithm.

Key type	Count	Output file size	Time to generate
RSA-4096	100,000	322MiB	17:05:33
RSA-2048	1,000,000	1.7GiB	20:54:05
DSA-1024	1,000,000	541MiB	12:34:48
DH-256	1,000,000	868MiB	0:06:24
DH-224	1,000,000	865MiB	0:07:02
DH-160	1,000,000	514MiB	0:02:02

Table 3.1: Information on the key generation process for different types of keys.

3.1.4 Testing the database

The PostgreSQL database that was designed and implemented initially, as seen in Figure 3.1, was tested for speed before Redis. The testing was done on the database with no optimisations or indices on columns, aside from the standard indices applied to primary keys. To begin with, the database was tested with the whole test key set of 5.1 million keys.

Table 3.2 shows data collected from the test on the initial version of the database. There, the results of different queries that were conducted can be seen. The queries included a direct query for both a stored key and an unknown key, with and without a filter on the key algorithm. Each query was run three times for each key algorithm, with three different keys to get a decent mean time for the query. All caches were cleared before each query, so each query is a worst case scenario where the data needs to be fetched from the disk. These results show that querying directly for the key is the fastest approach and adding a query on the key type only complicated the query, resulting in worse times.

However, one possible problem that was noted was the size of the table holding the keys became quite large, resulting in a total size of 11,367 MiB for 3,996 MiB of public key data. The reason for this abnormal size is that the index for the primary key, which in the case of PostgreSQL, is a B-tree index. This problem was later addressed in iteration two of the design, the final version of the database, which can be seen in Figure 3.2.

To check whether the query speed would rely on the amount of keys in the database, test with 10% of the generated mock data was conducted. Table 3.2 shows the mean times of the same queries on the smaller set of keys. The test implied that using a smaller set of keys had no effect on the time it took to search for a matching key, aside from taking a slightly longer time than the whole set. The reason for this is most likely due to the server being a virtual server with shared resources.

From the results of these tests, it was decided that testing Redis or any other DBMSs would not be beneficial due to the speed of PostgreSQL being sufficiently fast, and that any decrease in the time to query would be negligible. PostgreSQL's ability to access data in more ways than one gave another advantage over simpler databases. This is helpful, since gathering metadata about the keys being stored can be useful information and interesting to observe. Therefore, the project's database was chosen to be a PostgreSQL database.

Later on, the final version of the database, as seen in Figure 3.2, was tested with 5,100,000 test keys to ensure that it held up to the original design. The results of the test can be seen in Table 3.3. Since filtering by algorithm resulted in slower query times than the original tests, algorithm filtering was omitted from the testing on the final database and the direct matching of keys was the main focus. Looking at the results, it is clear that the final design works just as effectively as the original design, but results in a noticeably smaller public key table size.

The first design resulted in a public key table of 11,367 MiB, as mentioned earlier, but a total of 4,301 MiB in the final design. This is a huge improvement given that the total size of

Key type - count	Time to insert	Key that does not exists	Key that exists	Key that does not exist with algorithm filter	Key that exists with algorithm filter
RSA-4096 - 100,000	0:01:31	24.931 ms 20.772 ms 19.730 ms	37.787 ms 22.171 ms 44.889 ms	29.443 ms 30.014 ms 29.893 ms	57.166 ms 33.268 ms 43.983 ms
RSA-2048 - 1,000,000	0:16:33	24.931 ms 20.772 ms 19.730 ms	35.060 ms 25.176 ms 23.446 ms	29.700 ms 33.803 ms 31.771 ms	36.797 ms 45.874 ms 32.067 ms
DSA-1024 - 1,000,000	0:16:49	24.931 ms 20.772 ms 19.730 ms	27.456 ms 26.860 ms 25.710 ms	34.749 ms 28.206 ms 32.921 ms	30.435 ms 33.880 ms 32.481 ms
DH-160 - 1,000,000	0:17:15	24.931 ms 20.772 ms 19.730 ms	26.016 ms 27.614 ms 22.321 ms	30.361 ms 29.758 ms 30.695 ms	31.263 ms 28.897 ms 32.761 ms
DH-224 - 1,000,000	0:20:51	24.931 ms 20.772 ms 19.730 ms	29.198 ms 25.473 ms 26.037 ms	29.140 ms 25.398 ms 28.767 ms	31.240 ms 33.602 ms 29.731 ms
DH-256 - 1,000,000	0:24:01	24.931 ms 20.772 ms 19.730 ms	26.176 ms 23.175 ms 23.845 ms	29.140 ms 25.398 ms 28.767 ms	31.691 ms 31.033 ms 35.438 ms
Mean	**Total: 1:37:00**	**21.811 ms**	**27.689 ms**	**30.308 ms**	**35.089 ms**
Mean*	**Total: 0:08:07**	**28.853 ms**	**30.232 ms**	**34.089 ms**	**38.482 ms**

Table 3.2: Testing the initial version of the database. Insert and query times for different queries either utilising key types or not on the 5,100,000 keys set. The total size of the table was 11,367 MiB.
* Results for queries using 510,000 keys. The total size of the table was 1,137 MiB.

Key type - count	Time to insert	Key that does not exists	Key that exists
RSA-4096 100,000	0:01:24	35.571 ms 31.445 ms 36.999 ms	34.022 ms 28.718 ms 27.485 ms
RSA-2048 1,000,000	0:13:19	35.571 ms 31.445 ms 36.999 ms	26.514 ms 33.473 ms 23.507 ms
DSA-1024 1,000,000	0:13:01	35.571 ms 31.445 ms 36.999 ms	48.968 ms 26.834 ms 27.471 ms
DH-160 1,000,000	0:13:18	35.571 ms 31.445 ms 36.999 ms	25.057 ms 29.740 ms 25.450 ms
DH-224 1,000,000	0:14:05	35.571 ms 31.445 ms 36.999 ms	31.317 ms 28.873 ms 27.139 ms
DH-256 1,000,000	0:14:14	35.571 ms 31.445 ms 36.999 ms	26.895 ms 24.511 ms 25.789 ms
Mean	**Total: 1:09:21**	**34.672 ms**	**28.995 ms**

Table 3.3: Testing the final version on the database. Insert and query times for different queries either where the key exist or does not exist on the 5,100,000 keys set. The total size of the table was 4,301 MiB with the keys totaling 3,996 MiB.

the keys was 3,996 MiB, meaning the data for the index was reduced from 7,371 MiB down to 305 MiB with the use of a Hash index.

3.2 The key scraper

When the Lookey database had been designed, implemented and tested, real data needed to be gathered. The final version of the PostgreSQL database stores public keys, their owners and metadata, the platform on which the keys are used and the key algorithms.

The following subchapters describe how the public keys and their metadata where gathered for the Lookey database.

3.2.1 Sources of public cryptographic keys

When looking for websites that have millions of users that use asymmetric cryptographic keys, GitHub, GitLab and BitBucket were considered good candidates.

Git is a version control system that enables users to track changes in their code and easily revert back to their older versions [37].

GitHub is a very popular git-based version control system, used by millions of developers [38]. GitHub is mainly used to store code and its different versions, but also offers many development tools. When working with GitHub repositories, users often need to identify themselves to GitHub, using either their username and password, or an SSH key [17]. In addition to

version control and development tools, GitHub provides a REST API that was utilised in this project.

GitLab is another very popular git-based version control system like GitHub, but offers Continuous Integration (CI) Tools, whereas GitHub allows users to integrate such tools themselves [39]. GitLab offers two APIs, a REST API and a GraphQL API, which were both utilised in this project. More information on the APIs can be found in [40, 41].

BitBucket is yet another git-based version control system that uses SSH keys for authentication.

E-mails were sent to GitHub, GitLab and BitBucket asking if they were able to provide their users' usernames for this project for research purposes, instead of having to gather them by sending numerous requests to their APIs.

BitBucket answered the e-mail, stating that they are unable to provide this information and that it is not possible to retrieve someone's public key from them at the moment. However, they mentioned that they have an existing feature request to have those features added to BitBucket.

Neither GitHub nor GitLab answered the e-mails. These answers resulted in the need of implementing a scraper to gather real public keys and their metadata from these websites on the Internet.

In conclusion, three APIs were utilised in this project when gathering public keys, one from GitHub and two from GitLab.

3.2.1.1 GitHub REST API

The API's url is `https://api.github.com/`, which offers many endpoints to which requests can be sent in order to retrieve data. However, the only endpoint relevant to this project is the */users* endpoint that lists all GitHub users in pages, ordered by their user IDs in ascending order. It accepts two query parameters, *per_page* and *since*, which should both be integers. The *per_page* parameter indicates how many users should be returned; this number can not exceed 100. The *since* parameter indicates that the users returned should all have IDs larger than *since*.

The GitHub API has a rate limit of 60 requests per hour without authentication, but 5,000 requests per hour with authentication. The authentication consists of a GitHub username and the associated personal access token. More information about the API can be found in [42].

3.2.1.2 GitLab REST API

GitLab provides a REST API with the url `https://gitlab.com/api/v4/` and, similar to the GitHub REST API, offers multiple endpoints. The only endpoint relevant to this project is the */users/<username>/keys*. The endpoint returns all public keys owned by the given user, if they have any.

The API has a rate limit of 2,000 requests per minute and authentication is required. The authentication consists only of a personal access token.

There exists a */users* endpoint in the REST API that lists GitLab users paginated, 100 users per page. This endpoint did not prove useful for this project because the users are returned in descending order by their user IDs. This means that every time someone creates a new user, that user is the first user on the first page in the response from the REST API. This makes it harder to retrieve all users from GitLab using the REST API because the users listed on each page continually change while the endpoint returns the data. Each time a new user registers to GitLab, all previously registered users are pushed back one spot, moving users between pages. Fortunately, GitLab also provides a GraphQL API where users are returned in ascending order by their user ID.

```
{
  users(sort: CREATED_ASC, after: <afterCursor>) {
    pageInfo {
      endCursor
      hasNextPage
    }
    nodes {
      id
      username
    }
  }
}
```

Figure 3.3: GitLab GraphQL query that returns 100 users per page in ascending order by their user ID.

3.2.1.3 GitLab GraphQL API

The GitLab GraphQL API is located at `https://gitlab.com/api/graphql` [41]. All GitLab users can be retrieved by querying the API with the query in Figure 3.3.

This query returns 100 users (their user IDs and usernames) ordered by their user ID in ascending order as well as an *endCursor* and the boolean value *hasNextPage*. These values returned make it possible to query the next 100 users by setting the *afterCursor* as the *endCursor* returned from the previous query. This is repeated while *hasNextPage* remains true.

3.2.2 Design of the scraper

After figuring out how to scrape GitHub and GitLab for usernames and their public keys, sequence diagrams were created to get a better overview of how the scraper would operate. Figure 3.5 demonstrates how scraping for users is performed and Figure 3.7 demonstrates how collecting the public keys for the usernames in the database is done. High level diagrams of the user scraping and the key scraping can be seen in Figures 3.4 and 3.6 respectively.

3.2.3 Implementation of the scraper

Python was used for the initial implementation of the scraper. After some more thought, the programming language Go was deemed a better fit for this purpose than Python. Go was preferred for implementing the scraper since it would be a concurrent program [43]. As the scraper's implementation progressed, it became even clearer that changing programming languages was a good decision. The Go language has Goroutines that allow you to start a concurrent thread by simply writing "go" in front of the function call. Go is also very convenient when using channels, as both sending values into and receiving values from a channel is quite straightforward.

3.2.3.1 GitHub users

To gather the GitHub users, we used the */users* endpoint in the GitHub REST API 3.2.1.1. The scraper enters a loop that sends a GET request in every iteration to the endpoint with the *since*

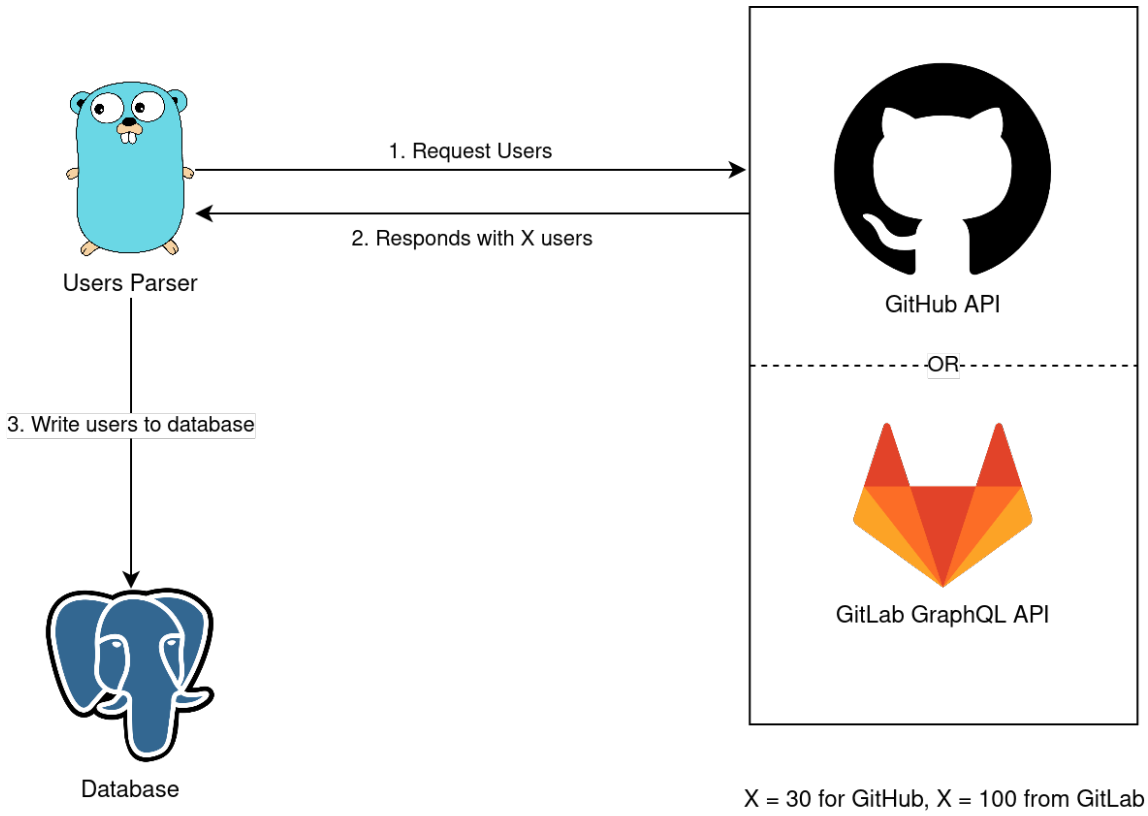

Figure 3.4: High level diagram of users scraping

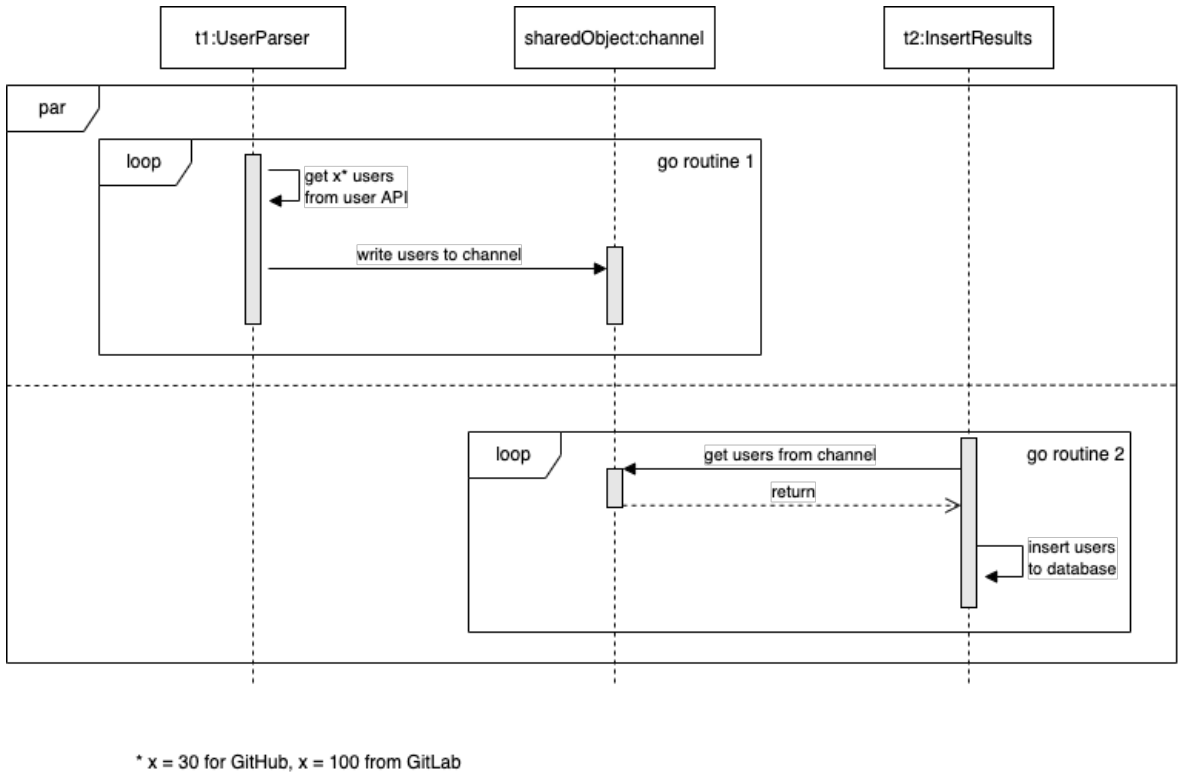

Figure 3.5: Detailed Sequence diagram of user scraper

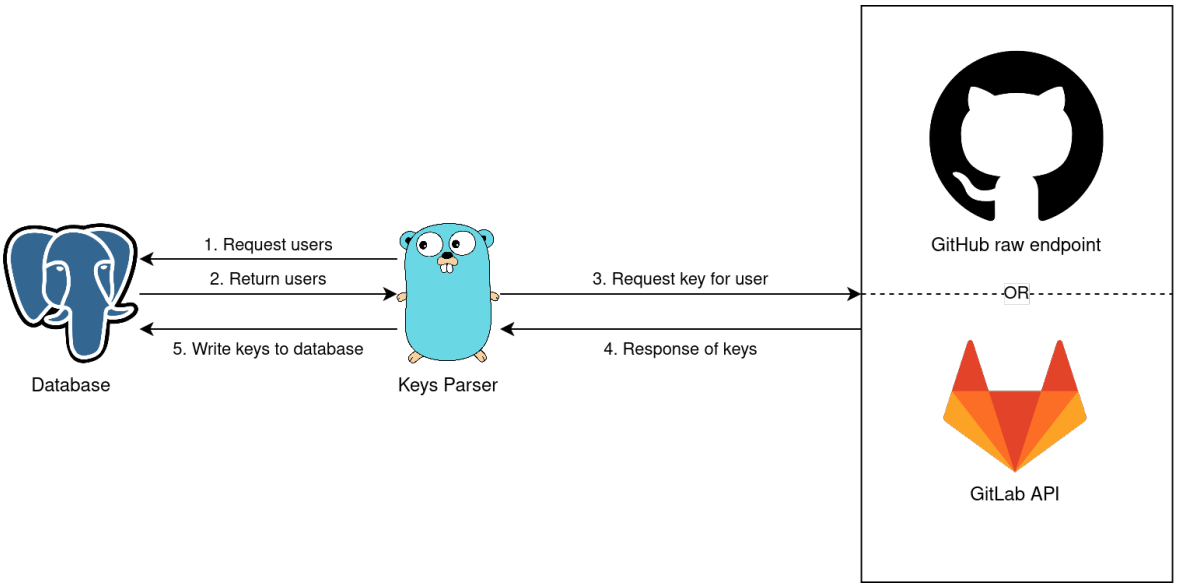

Figure 3.6: High level diagram of keys scraping

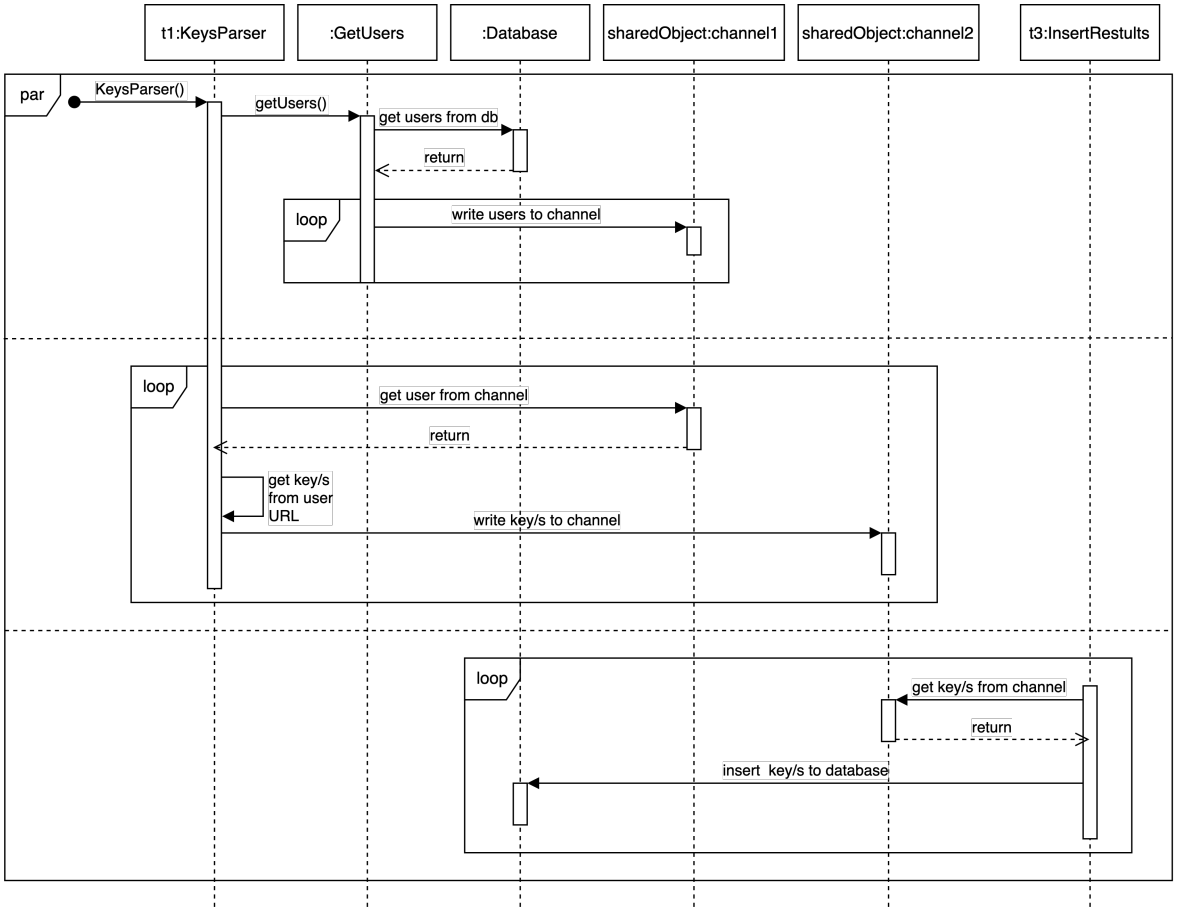

Figure 3.7: Sequence diagram of key scraper

parameter set to 0 initially. The *since* parameter is then updated to the last user's user ID in the previous response. This returns 30 users each time and runs until there are no more users.

The *per_page* parameter was not used in this project and the reason why is we did not know about it until after we had collected all GitHub usernames. If we had known, *per_page* had been set to 100.

When each GET request had returned a response, all users were sent into a channel. This channel was then used in another Goroutine that added the users into our database.

3.2.3.2 GitLab users

The GitLab GraphQL API was used to collect all GitLab usernames, with the query from 3.3. The *after* parameter is initially an empty string. As soon as the first request is sent, the *after* parameter is set as the *endCursor*'s value from the previous response. All usernames are then added into the database as is done with the GitHub usernames. This is repeated until *hasNextPage* is false.

3.2.3.3 GitHub/GitLab keys

There are two ways, that we know of, to access users' public keys both from GitHub and GitLab. One way is to use their REST APIs and the other way is via the following raw endpoints:

```
GitHub: \url{https://github.com/<username>.keys}
GitLab: \url{https://gitlab.com/<username>.keys}
```

The latter approach was used for GitHub since their APIs have a rate limit but these endpoints did not. On the other hand the API was used for GitLab since both were rate limited but the API had a higher limit.

The same logic is used when retrieving public keys from both GitHub and GitLab. The first thing the scraper does is to retrieve all GitHub usernames from the database and send them into a channel. The scraper then creates multiple Goroutines, depending on the arguments given at run-time, all issuing GET requests to the URL with the next username from the channel. All keys are sent into another channel and that channel is then used in another Goroutine that inserts the keys to the database, similar to when inserting users to the database.

3.3 The Lookey lookup tool

Gathering public keys and finding ways to gather information about public keys from sources like Certificate Transparency has no real value unless private keys can be matched to their respective public keys and metadata. That is where the Lookey lookup tool comes in. It is a simple CLI tool written in Rust that accepts a private key or a public key as an argument. In the case of a private key, it attempts to parse it using multiple different formats. Once it is successfully parsed, it derives its matching public key.

With the public key in hand, the tool then computes the sha256 hash of the public key in DER format, which is the same as what is done with X.509 certificates, which were discussed in Chapter 2.3.2. This hash is what Certificate Transparency can be queried with. The public key and the sha256 hash are then sent to a REST API that resides on the same server as the database. The API is written in Python using the FastAPI [44] library. The reason for using this API was to avoid the need for a direct database connection from the lookup tool to the database, by only offering one way to use the database for any authorised users. This API has only a single endpoint that takes in the public key and hash and queries the database and

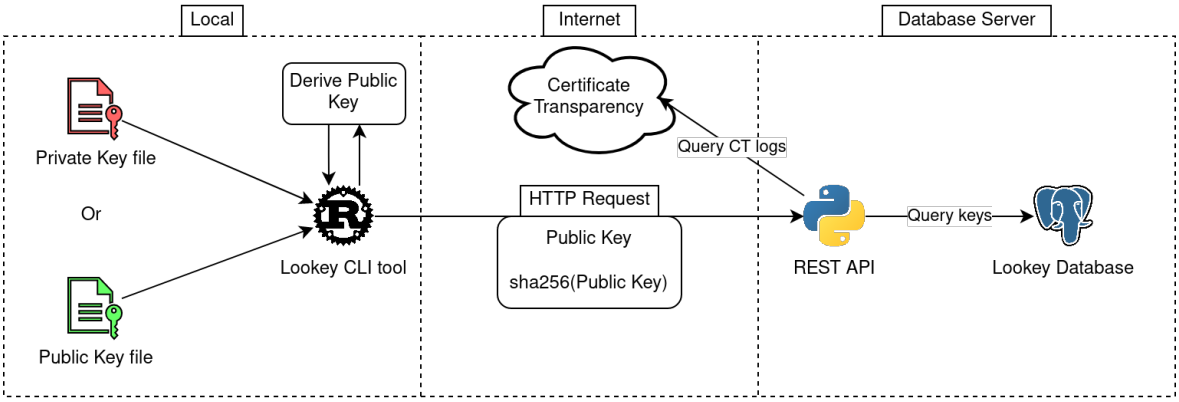

Figure 3.8: Overview of how the Lookey lookup tool works on a high-level.

Certificate Transparency for potential matches. Figure 3.8 shows a high-level overview of how a lookup via Lookey is performed by taking in a Private or Public key and sending the necessary information to the API to query either CT or the database.

Chapter 4

Evaluation and results

The intent of the research conducted was to analyse if a private key lookup tool could be built using public keys collected online and stored in a database. Thus far, the background and design has been explained in previous chapters. In this chapter, the evaluation and results of gathering real data and testing the database with real private keys will be discussed.

4.1 Scraping public keys

As previously discussed, Lookey had three main sources of keys: Certificate Transparency and SSH keys from GitHub and GitLab. Since a suitable solution to find certificates from CT was found and used, scraping CT for data was not necessary. Scraping GitHub and GitLab was however necessary to gather users' public keys for the lookup, which will be discussed below. As mentioned in Chapter 3.2.1, e-mails were sent beforehand to ask for the data instead of needing to scrape for it, but no response was received from GitHub or GitLab. Table 4.1 shows an overview of the scraping performed on GitHub and GitLab.

4.1.1 Users scraping

The users scraping was the first step taken and was done before any keys scraping was started. We started scraping users from GitHub. It was out first choice since GitHub has significantly more users than GitLab as can be seen in Table 4.1. The time taken to scrape is not quite representative of the entire process since in the beginning, the scraping was slower than expected. This was due to the database not managing to keep up and causing a bottleneck. Later on, we managed to solve this issue by using a separate Goroutine to insert users into the database as well as optimising the inserts. This accelerated the scraping process without putting too much load on the API.

Platform	Scrape type	Time to scrape	Total gathered	Total covered	Users with keys
GitHub	Users	171:53:31	101,306,024	All*	-
GitLab	Users	134:54:33	10,226,814	All*	-
GitHub	Keys	184:36:35	4,802,438	55,052,911	2,734,005
GitLab	Keys	81:36:38	1,966,740	2,666,115	905,717

Table 4.1: Information on the users and keys scraping process. *Users registered before May 4th 2022.

Shortly after starting the GitHub scraping, scraping of GitLab users began. GitLab, however, does not allow scraping their APIs. This will be discussed in Chapter 5.2.

Looking at these results, it is interesting to see the time difference in gathering all the users from each platform. Even though GitHub has roughly 10 times more users than GitLab, the total time difference between scraping each platform was not substantial. This was due to the fact that the GraphQL API used for GitLab took a lot longer to respond to each request than the GitHub REST API. To begin with, the GitLab GraphQL API returned a response every 6 seconds on average, but later on it escalated up to roughly 3 responses each second. It is also worth mentioning that the GitHub scraping was only done with 30 users per request, but was later discovered that 100 users could be retrieved in each request. Using 100 users per request would have sped up the scraping even more for GitHub. These time results are almost representative of a worst case scenario of gathering all users for these platforms. If any unscrupulous party would want to do this it would not take them long to have a database of all the users from both platforms.

4.1.2 Keys scraping

After gathering all the users of both platforms and storing them in the database, it was possible to query the whole set of users from the database and start scraping for each individual user's public keys. This was done with multiple Goroutines in the scraper, to allow scraping multiple users' keys at a time, since each individual request was independent of all others. Not all users' keys were scraped. This is because it takes a lot longer to scrape keys than users. There was simply not enough time to scrape all users' keys nor was it necessary for the evaluation of Lookey. A total of 55,052,911 GitHub users were scraped for their keys and 2,666,115 GitLab users. The time it took for each platform can be seen in Table 4.1.

The gathering of the keys was not entirely the same for GitHub and GitLab. GitHub did not rate limit their raw endpoint, `https://github.com/[user].keys`, which was used within a reasonable margin, to not overload their servers. GitLab, on the other hand, did both rate limit all of their raw endpoints and APIs. This is why it was decided to use their API, `https://gitlab.com/api/v4/users/[user]/keys` instead, since it offered more requests per minute than the raw endpoint [45].

During the GitLab keys scraping, the API was first used with an authentication token allowing a total of 2,000 requests per minute, but adding the token resulted in almost a third of the requests being rejected with a "401 Unauthorized status." This was most likely a bug so the scraping was run with no access token and thus 500 requests per minute could be sent.

During the scraping of keys and while looking at the overview of keys gathered, which can be seen in Table 4.2, only a small amount of users had configured SSH keys for GitHub while proportionally more GitLab users had configured keys. Only 5% of the scraped GitHub users had SSH keys configured for their account, while 34% of GitLab users had SSH keys configured. This might have been the case due to the scraping going through the users in ascending order of user IDs so earlier users might not be active or GitLab might enforce SSH keys more than GitHub.

Another interesting observation of the keys scraping was how many user keys requests returned a 404 error code, indicating that they no longer exist. This suggests that the API returned users even though they had been deleted and could not be accessed anymore.

Table	Rows	Table data size	Table total size
Key Types	8	8 KiB	0 40 KiB
Public Keys	6,580,604	3 GiB	3.4 GiB
Metadata	111,532,838	7.2 GiB	14 GiB

Table 4.2: Overview of the collected data in the relevant database tables. The total size is the size of the data and any other metadata used by PostgreSQL. Platform is omitted due to only containing 2 already known rows.

Key type	Count
ssh-rsa	5,072,271
ssh-ed25519	1,459,575
ecdsa-sha2-nistp256	23,251
sk-ssh-ed25519@openssh.com	2,029
ecdsa-sha2-nistp521	14,703
ssh-dss	5,928
sk-ecdsa-sha2-nistp256@openssh.com	1,631
ecdsa-sha2-nistp384	1,216

Table 4.3: Overview of the key types found after scraping both GitHub and GitLab.

4.2 Data analysis

Table 4.2 shows an overview of the whole data collected during the scraping steps. The keys collected and their count are shown in Table 4.3. The count of keys in the table confirms the preliminary research of websites and SSH keys mentioned in 2.2 about the most common keys in use being RSA, ED25519 and DSA. No keys of the key algorithm Diffie-Hellman were scraped, to no surprise as they are not used for SSH.

The total number of gathered keys found in Table 4.1 does not match the total number of keys in the keys table as is shown in Table 4.2. This is due to the fact that 376,926 public keys had more than one owner, resulting in only one key existing in the database. This is to be expected since users often have multiple accounts on different platforms and might even share the same key for different accounts on the same platform.

4.3 Lookey lookup times

To test the effectiveness of the lookup tool, an evaluation was conducted using real world private keys of the from the researchers as well as some public keys of X.509 certificates. The test was conducted using the Lookey lookup tool described in 3.3 from the remote server running the database. This was done to simulate a real world scenario in the best way possible, since the tool would be used from a remote machine. The result of these tests can be seen in Table 4.4 and includes private key lookup both with and without Certificate Transparency lookup. This was done to test the database in isolation as well as in conjunction with CT, to show the speed of the database. The database can be managed while the CT search is reliant on an external service and can not be managed.

The results show that running without CT took each request a shorter amount of time, which is to be expected, but it had no real significant effect on the usefulness of the solution. Querying for certificates that do not exist seems to be a bit faster as well, since the data does not need

Test type	Result exists	Time
SSH key with CT	Yes	1.008 s
SSH key with CT	No	0.616 s
SSH key without CT	Yes	0.922 s
SSH key without CT	No	0.235 s
X.509 public key	Yes	0.403 s
X.509 public key	No	1.088 s
SSH key directly on database	Yes	0.568 s
SSH key directly on database	No	0.045 s

Table 4.4: Table showing speeds of querying either existing or non-existing keys using the whole system (as was shown in Figure 3.8) as well as directly with the database.

to be retrieved and prepared to be sent back. It also shows that the lookup is well within the margin for a fast lookup, since a 1 second lookup for a key is not a lot of time for a human to wait.

Chapter 5

Discussion

This chapter will cover discussions regarding the research conducted in this paper such as ethics, privacy and legal concerns, along with some practical concerns on the implementation.

5.1 Ethical concerns

The data collected for this project is, as mentioned before, very sensitive. If the data collected in this project would end up in the wrong hands, it could be used for malevolent purposes. If an unscrupulous person got a hold of it and used it to find the owner of a private key, that person would be able to do a number of malicious things, e.g. impersonate the owner. Instead of alerting the owner that their private key has been compromised, as ethical hackers would do, they could use it to infiltrate systems, access classified data, etc. We realise that it is vital to handle the collected data with great care, and proper security measures must be taken.

5.1.1 Privacy concerns and securing the Lookey system

Privacy has been a prevalent topic in recent years as more and more people have become aware of their private information online. Legislators have also been adamant in protecting users and their data online with new laws and legislation, such as the GDPR [46], that companies need to follow. Storing hundred of millions of usernames and public keys clearly falls under these laws and regulations, which is why this topic needed to be addressed to ensure the legality of this research. Even though our intentions are good, the laws apply to us as well.

To secure the data collected for this project, it is only stored on a Digital Ocean machine, provided by Syndis. Furthermore, the machine's firewall was configured accordingly. It only allows connections to port 5432 (the default PostgreSQL port) from the IP addresses of team members and advisors. This prevents unauthorised people from trying to gain access to the database.

5.2 Legal concerns

Since our database stores usernames and public keys from various platforms, we used APIs provided by the platforms themselves to gather the data. All the APIs have a Terms of Use section that we examined, although after data collection. We realise that this should have been done before collecting the data. The GitHub REST API's Terms of Use states in section 7 of [47] that for research purposes, scraping of information is allowed. However, the GitLab REST API's Terms of Use states in section 1.3.9 in [48] that bulk collection and scraping of

information is prohibited. Even though the scraping of the GitLab APIs is prohibited, this was an important step in order to understand how the data would be handled, given the sensitivity.

Since scraping information from the GitLab API is prohibited, we erased all information gathered from that API after all research and evaluation were finalised. However, it is important to note that since we, with good intentions, were able to collect significant amounts of such sensitive information, people with bad intentions are capable to do so as well. Using someone's private key to do harm is much more serious than violating the Terms of Use of an API. With that in mind, a person who intends to misuse someone's private key will most certainly look past such API rules.

If Syndis were to use Lookey with data from GitLab, they would have to establish some kind of an agreement with GitLab. Furthermore, they would also have to do the same with GitHub and other platforms they intend to use, since Lookey will not continue being a research project.

5.3 Practical concerns in the implementation

As mentioned in Chapter 3.2.1.1 the GitHub REST API has a rate limiting of 5,000 requests per hour per authentication. This created a problem for this project since it was on a deadline.

We were not sure, whether or not we should use all three team members' authentication tokens - allowing us to issue 15,000 requests per hour, or just one token. After discussing this further, we decided to use all three tokens alternately in the same running process. Doing so would have the same effect on the API as using three computers simultaneously collecting the data with one different authentication token each - which we would have done otherwise. However, we realise that this is not a viable solution for scalability as the rate limiting is there for a reason. Circumventing the rate limit any further would be unethical. The optimal solution would be to come to an agreement with the platforms instead of scraping their APIs which are not built for such usage. This is something that Syndis needs to keep in mind if they intend to further develop the Lookey lookup system.

Chapter 6

Future work

The future of Lookey will be in the hands of the Icelandic Cyber Security company Syndis, who collaborated with us on this project. Syndis plans to use Lookey when performing security audits on clients and to notify any owners of leaked private keys they might find online or in data leaks.

There are, however, a few extensions and considerations to the research in this paper that can be looked into to extend the solution previously described.

6.1 Other sources of keys

As described before, the scraping of keys was only done on two websites, GitHub and GitLab. There are, although, a lot more websites that offer or might offer SSH or key based authentication, which could be scraped for even more data. One example is BitBucket that we originally planned on scraping, but later discarded due to their API not offering users queries. In the e-mail response, as mentioned in Chapter 3.2.1, BitBucket stated that there was an existing feature request to have the features we need added to BitBucket.

Other sources of keys were also considered such as keys for Cryptocurrency wallets, which store private keys that protect the currency of their owner. [49] discusses this topic nicely in its introduction and how having the private key of a bitcoin wallet gives anyone complete control over it.

Another possible avenue of interest is to look into keys used for PGP [50]. PGP is used a lot these days to encrypt e-mails and digital signatures, as well as for any general applications that might want the information to be private. Keys could then possibly be queried from online databases of PGP keys such as [51].

The final approach considered to gather asymmetric keys would be to gather keys directly from other hosts on the Internet. This could be done in two different ways: by connect to other hosts or by allowing incoming connections. The first approach can be achieved by scanning the internet and trying to connect via SSH to the hosts in order to gather their public keys. This can be easily achieved with an already existing tool in Linux called ssh-keyscan [52], which gathers public keys from SSH servers. The second approach can be achieved by setting up a so called honeypot. A honeypot is a decoy system that is expected to be broken into while allowing the owner to monitor it in detail. [53] describes a virtual honeypot that can be setup on any machine and mimic any operating system and network architecture. Setting up such a virtual honeypot could offer the ability to wait for incoming SSH connections to gather the “attackers” public key used for authentication.

All of these approaches offer extending the solution discussed in this paper beyond its initial implementation and it is up to Syndis or other interested parties to continue researching these approaches.

6.2 Capabilities of Lookey for mass scans

The original design of the solution discussed in previous chapters was to offer fast single searches to match a few leaked private keys at a time. It is however a possibility that later on this solution might see a more widespread use if Syndis decides to offer the service to more people. This will change the precondition of this solution and might require a faster or more distributed approach to the database component. This is a topic that goes beyond the scope of this paper and might require additional research on its own. It is however worth considering, to scope the capabilities of this solution for future uses.

The solution described in this paper can handle roughly one query per key each second as was shown in Table 4.4. This is not an effective batch querying method, since only a single public key gets sent to the API and queried. In the case of handling mass scans of keys, such as in the case of a huge data leak, a new endpoint could be defined to handle multiple incoming keys. The tests conducted in Chapter 3.1.4 were always a worst case scenario where there was no caching so all queries resulted in reading from the disk. This could be completely avoided with a suitable amount of RAM on the database server, to be able to cache the entire set of public keys. This is not an unreasonable task since the total amount of public keys collected previously resulted in a table size of 3.4 GiB as shown in Table 4.2. This means that storing the entire set of public keys in-memory can easily be achieved, since as of writing this report, 4 GiB of memory is on the lower end of server hardware specs. Therefore, if all keys could be stored in memory, each query would go straight to the cache and based on short testing of those queries, each would give a result in around 0.5 ms. This is only to query the key in the public key table without any metadata connected to it. On the other hand, querying for the metadata as well, which is done in the API, usually took around 500 ms, as can be seen in Table 4.4. This is due to the amount of table joins needed. This means that a total of 2,000 keys could be queried from the database each second without metadata while querying two public keys with takes the same time. This is not considering Certificate Transparency lookups which take a lot longer.

Considering a theoretical example that all private keys of the stored public keys, totalling 6,580,604 keys, would leak, querying only the database for the matching keys would take a total of 54.8 minutes. In order to get metadata for each key as well, one would need to wait 38 days, which is very inefficient. This is the absolute worst case scenario but it shows the time needed for both types of queries. If Lookey would be used for such mass scans the implementation of the database would need to be optimised, distributed, or replaced.

Chapter 7

Conclusion

This paper demonstrated the challenges faced in the design and implementation of a system capable of finding the owner of a cryptographic private key. Creating such a system was achieved by scraping different sources online for public keys and storing them in a database. Certificate Transparency was also introduced and how it could be used to find public keys associated with SSL/TLS certificates of websites. The data gathered and Certificate Transparency was then used to get information and context for leaked private keys.

Before querying the database, a public key was derived from the given private key and a sha256 hash of the public key was calculated. Then, the public key and the hash were used to query the database and Certificate Transparency for possible matches. The time to find a match only took around 1 second in the worst case, greatly improving the efficiency of trying to find the use of a key manually. Extending the solution to add more sources of keys was also proposed to increase the effectiveness of the tool even further.

Furthermore, it was shown that scraping data online was not extremely time consuming, despite limitations and rules in place to block it. Thus, a person with malicious intent could create a tool, such as the one described in this paper, in a relatively short time and use it for immoral purposes.

This demonstrates the importance of having this type of a solution in the hands of a trusted cyber security company, such as Syndis. That allows cyber security professionals to always be one step ahead of malicious hackers and being able to alert users of potential breaches or the leaking of their private keys.

Bibliography

[1] A. T. Tunggal, “Biggest data breaches in recent years,” April 2022, accessed: May 2, 2022. [Online]. Available: https://www.upguard.com/blog/biggest-data-breaches

[2] K. Bocek, “Attack on trust threat bulletin: Sony breach leaks private keys, leaving door open,” December 2014, accessed: May 2, 2022. [Online]. Available: https://www.venafi.com/blog/attack-on-trust-threat-bulletin-sony-breach-leaks-private-keys-leaving-door-open

[3] R. Kaur and A. Kaur, “Digital signature,” in *2012 International Conference on Computing Sciences*, 2012, pp. 295–301.

[4] M. Meli, M. R. McNiece, and B. Reaves, “How bad can it git? characterizing secret leakage in public github repositories.” in *NDSS*, 2019.

[5] “Bitmart security breach update,” accessed: May 5, 2022. [Online]. Available: https://support.bmx.fund/hc/en-us/articles/4411998987419-BitMart-Security-Breach-Update

[6] A. Sharma, “Eu investigating leak of private key used to forge covid passes,” Oct 2021, accessed: May 5, 2022. [Online]. Available: https://www.bleepingcomputer.com/news/security/eu-investigating-leak-of-private-key-used-to-forge-covid-passes/

[7] J. Tuwiner, “What was the mt. gox hack?” April 2022, accessed: May 5, 2022. [Online]. Available: https://www.buybitcoinworldwide.com/mt-gox-hack/

[8] S. Selleri, “The roots of modern cryptography: Leon battista alberti’s "de cifris",” *URSI Radio Science Bulletin*, vol. 2020, no. 375, pp. 55–63, 2020.

[9] G. C. Kessler, “An overview of cryptography,” *Handbook on Local Area Networks,*, 2003.

[10] D. G. Amalarethinam, J. S. Geetha, and K. Mani, “Analysis and enhancement of speed in public key cryptography using message encoding algorithm,” *Indian Journal of Science and Technology*, vol. 8, no. 16, p. 1, 2015.

[11] S. Chandra, S. Paira, S. S. Alam, and G. Sanyal, “A comparative survey of symmetric and asymmetric key cryptography,” in *2014 International Conference on Electronics, Communication and Computational Engineering (ICECCE)*, 2014, pp. 83–93.

[12] J. Linn, “Privacy enhancement for internet electronic mail: Part i: Message encryption and authentication procedures,” RFC 1421, IAB IRTF PSRG, IETF PEM WG, Tech. Rep., 1993.

[13] S. Josefsson and S. Leonard, “Textual encodings of pkix, pkcs, and cms structures,” Internet Engineering Task Force (IETF), Tech. Rep., 2015.

[14] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk, "Internet x. 509 public key infrastructure certificate and certificate revocation list (crl) profile," Network Working Group, Tech. Rep., 2008.

[15] L. Harn and J. Ren, "Generalized digital certificate for user authentication and key establishment for secure communications," *IEEE Transactions on Wireless Communications*, vol. 10, no. 7, pp. 2372–2379, 2011.

[16] T. Ylonen, "Ssh–secure login connections over the internet," in *Proceedings of the 6th USENIX Security Symposium*, vol. 37, 1996.

[17] B. Balter, "Token authentication requirements for api and git operations," july 2020, accessed: May 8, 2022. [Online]. Available: https://github.blog/2020-07-30-token-authentication-requirements-for-api-and-git-operations/

[18] "Openssh," accessed: May 8, 2022. [Online]. Available: https://www.openssh.com/

[19] J. Galbraith and R. Thayer, "The secure shell (ssh) public key file format," *Request for Comments*, vol. 4716, 2006.

[20] J. Aas, "Let's encrypt: Delivering ssl/tls everywhere," 2014, accessed: May 5, 2022. [Online]. Available: https://letsencrypt.org/2014/11/18/announcing-lets-encrypt.html

[21] "Ca certificates in firefox," accessed: May 5, 2022. [Online]. Available: https://ccadb-public.secure.force.com/mozilla/CACertificatesInFirefoxReport

[22] "Certificate transparency," accessed: May 6, 2022. [Online]. Available: https://certificate.transparency.dev/

[23] A. Mallik, "Man-in-the-middle-attack: Understanding in simple words," *Cyberspace: Jurnal Pendidikan Teknologi Informasi*, vol. 2, no. 2, pp. 109–134, 2019.

[24] "Certificate transparency monitors," accessed: May 12, 2022. [Online]. Available: https://certificate.transparency.dev/monitors/

[25] "Certificate search," accessed: May 5, 2022. [Online]. Available: https://crt.sh

[26] D. Decker, D. Ayrey, and J. R. Ty, "Driftwood," 2021, accessed: May 6, 2022. [Online]. Available: https://github.com/trufflesecurity/driftwood

[27] D. Ayrey, "Driftwood: Know if private keys are sensitive," November 2021, accessed: May 6, 2022. [Online]. Available: https://trufflesecurity.com/blog/driftwood

[28] E. F. Codd, "A relational model of data for large shared data banks," in *Software pioneers*. Springer, 2002, pp. 263–294.

[29] A. Davoudian, L. Chen, and M. Liu, "A survey on nosql stores," *ACM Computing Surveys (CSUR)*, vol. 51, no. 2, pp. 1–43, 2018.

[30] "Postgresql," accessed: May 4, 2022. [Online]. Available: https://www.postgresql.org/

[31] "Redis on flash," accessed: May 4, 2022. [Online]. Available: https://redis.com/redis-enterprise/technology/redis-on-flash/

[32] R. Ramakrishnan, J. Gehrke, and J. Gehrke, *Database management systems*. McGraw-Hill New York, 2003, vol. 3.

[33] "E.2. release 14.2," accessed: May 12, 2022. [Online]. Available: https://www.postgresql.org/docs/14/release-14-2.html

[34] "11.2. index types," accessed: May 12, 2022. [Online]. Available: https://www.postgresql.org/docs/14/indexes-types.html

[35] "Rust," accessed: May 3, 2022. [Online]. Available: https://www.rust-lang.org/

[36] "Openssl," accessed: May 5, 2022. [Online]. Available: https://www.openssl.org/

[37] "1.2 getting started - a short history of git," accessed: May 12, 2022. [Online]. Available: https://git-scm.com/book/en/v2/Getting-Started-A-Short-History-of-Git

[38] "Github," accessed: May 5, 2022. [Online]. Available: https://github.com/

[39] "Gitlab," accessed: May 11, 2022. [Online]. Available: https://about.gitlab.com/

[40] "Gitlab api," accessed: May 11, 2022. [Online]. Available: https://docs.gitlab.com/ee/api/

[41] "Graphql api," accessed: May 11, 2022. [Online]. Available: https://docs.gitlab.com/ee/api/graphql/

[42] "Github rest api," accessed: May 11, 2022. [Online]. Available: https://docs.github.com/en/rest

[43] "Go documentation," accessed: May 12, 2022. [Online]. Available: https://go.dev/doc/

[44] S. Ramírez, "Fastapi," accessed: May 8, 2022. [Online]. Available: https://fastapi.tiangolo.com/

[45] "Gitlab.com-specific rate limits," accessed: May 11, 2022. [Online]. Available: https://docs.gitlab.com/ee/user/gitlab_com/#gitlabcom-specific-rate-limits

[46] "General data protection regulation," accessed: May 10, 2022. [Online]. Available: https://gdpr-info.eu/

[47] "Github acceptable use policies." 2022, accessed: May 11, 2022. [Online]. Available: https://docs.github.com/en/site-policy/acceptable-use-policies/github-acceptable-use-policies#7-information-usage-restrictions

[48] "Gitlab api terms of use." 2022, accessed: May 11, 2022. [Online]. Available: https://about.gitlab.com/handbook/legal/api-terms/

[49] M. Guri, "Beatcoin: Leaking private keys from air-gapped cryptocurrency wallets," in *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*. IEEE, 2018, pp. 1308–1316.

[50] S. Garfinkel *et al.*, *PGP: pretty good privacy*. "O'Reilly Media, Inc.", 1995.

[51] "keys.openpgp.org," accessed: May 10, 2022. [Online]. Available: https://keys.openpgp.org/

[52] D. Mazieres and W. Davison, *ssh-keyscan - gather SSH public keys from servers*, November 2019, accessed: May 11, 2022. [Online]. Available: https://man.openbsd.org/ssh-keyscan.1

[53] N. Provos *et al.*, "A virtual honeypot framework." in *USENIX Security Symposium*, vol. 173, no. 2004, 2004, pp. 1–14.