A System for Increasing Confidence in Digital Signatures

Authors

Dorottya Papp, Balázs Kócsó, Bálint Molnár, Milán Unicsovics

Supervisors

Dr. Levente Buttyán, Dr. Tamás Holczer

October 22, 2014

Contents

1 Introduction **2**

2 State-of-the-Art **5**
2.1 EFF SSL Observatory 5
2.2 ICSI Certificate Notary 6
2.3 EFF Sovereign Keys 7
2.4 Perspectives 7
2.5 Convergence 8
2.6 Google Certificate Transparency 9

3 Architecture **11**
3.1 Data Collection 12
3.1.1 Crawlers 12
3.1.2 Downloading 13
3.2 Data Processing 15
3.2.1 Public keys and Certificates 19
3.2.2 Portable Executables 22
3.2.3 Java Archives and Android Packages 24
3.3 Data Storage 27
3.3.1 Metadata 27
3.3.2 Big Data 29
3.4 Alert System 31
3.4.1 Simple Alerts 32
3.4.2 Signing Key Usage Alerts 32
3.5 User Interface 33
3.5.1 Upload 34
3.5.2 Search 34
3.5.3 Alerts 36

4 Case Study **39**
4.1 Duqu 41
4.2 Flame 41

5 Conclusion and Future Work **43**

Bibliography **46**

1. Introduction

Recent targeted malware attacks, e.g., Stuxnet, Duqu, and Flame, used digitally signed components that appeared to originate from legitimate software makers. In case of Stuxnet and Duqu, the private code signing keys of legitimate companies were suspected to be compromised and used by the attackers. In case of Flame, the attackers generated a fake certificate that appeared to be a valid code signing certificate issued by Microsoft, and used the corresponding private key to sign their malware [4]. This actually allowed Flame to masquerade as a Windows Update Proxy, and to infect computers on a local network by exploiting the automatic update procedure of Windows.

The purpose of code signing is to ensure the authenticity and integrity of software packages, however, ultimately the effectiveness of code signing as a security mechanism also depends on the security of the underlying Public Key Infrastructure (PKI). As the examples above show, attackers have already started to exploit weaknesses in the PKI system supporting code signing, and we expect that this trend will become stronger. The reason is that new versions of Windows (and other platforms as well) require software to be signed, otherwise, they ask for a confirmation of the user before the software is installed. Hence, attackers can benefit from signing their malware, as it allows for stealthy infection of victim systems.

Consequently, there is an urgent need to strengthen the PKI which code signing relies on. The difficulty is that this infrastructure is global, involving many participants in different countries (e.g., different CAs and software makers), and a multitude of procedures and practices. It is difficult to enforce common rules in such an environment and meet the same standards across the entire system. Also, the evolution of the system is uncontrolled, often governed by major, powerful stakeholders, and this can lead to suboptimal solutions (e.g., hundreds of root certificates that are all implicitly trusted by the users). Changing the entire system overnight is not feasible, and thus, one needs a solution that can be gradually deployed. In addition, given its size and complexity, making the entire PKI system 100% secure is illusionary, and one should rather adopt a best effort approach that raises the bar for the attackers even if attacks cannot be completely eliminated.

Motivated by the Stuxnet, Duqu, and Flame cases, the specific problem that we address in our project is that standard signature verification procedures used in today's PKI systems do not allow for detecting key compromise and fake certificates. Therefore, the objective of the project is to augment the standard signature verification workflow with checking of reputation information on signers and signed objects.

For this purpose, we decided

- to build a data collection framework for signed software and code signing certificates,
- to build a data repository that can handle large amount of signed objects efficiently, and that supports a flexible query interface,

- to use the repository for providing reputation information for signed objects, such as when a given signed object has been first seen and how often it was looked up by users,
- to provide alert services for private key owners that help them detecting when their signing keys were illegitimately used, and hence, probably compromised.

Our system, called Repository of Signed Code (ROSCO), does not aim at replacing the entire code signing infrastructure. Rather, in accordance with the best effort principle and the requirement of gradual deployment, ROSCO complements existing PKI functions with useful services that can be used by different participants to increase their confidence in the legitimacy of signed code. In particular, ROSCO provides the following advantages to the different participants:

- For software makers, the weaknesses of the code signing procedure undermine the trust in their code. An independent repository of signed code and accompanying certificates enables software makers to maintain trust in their code. More importantly, such a repository can be used to detect the malicious use of a software maker's signing key. This early detection capability is a unique property of such a global repository and cannot be achieved using the traditional PKI.
- For software platform operators, such as operating system providers and global software service providers, the repository is an indispensable source of information about the trustworthiness of installed code. As mentioned earlier, recent versions of Microsoft Windows, for example, require valid signatures for seamless installation of software packages. Cross-checking the code signing certificate, and thus, the integrity of the software code in our certificate repository is a major step ahead in protecting the integrity of the Windows operating system.
- For end users, the benefits are obvious: our repository serves them when they have to decide about the trustworthiness of a to-be-installed code.
- The code signing repository could be an invaluable source of information for security companies too. Based on the collected information, they can detect malicious campaigns and trends in signing malicious code. This repository could nicely integrate with many of the security offering available on the market.
- Finally, regulators and other authorities find an inherent value in making software more trustworthy. Similar to security companies, authorities can derive longitudinal statistics about malicious code and use it as an input when defining global defense strategies and coordination mechanisms.

Finally, we should mention again that the repository complements the existing code signing infrastructure, and there's no requirement whatsoever to change the operating principles of participants that do not want to use it. This opt-in approach allows for the possibility of gradual deployment. We expect, however, that as the size of our repository grows, the services that we can provide become more useful, and this will attract more participants to use our system. So potentially, the adoption cycle can be fast, and many participants can benefit from the strengthened code signing infrastructure in a short time.

The organization of our paper is the following: in Section 2 we will give a short review on the state of the art. In Section 3 we will discuss the architecture of the repository. We

will show possible ways of data gathering and processing and how to store the processed file. We will also give a review on the part of the system responsible for sending notification and on how to interact with the repository in Sections 3.4 and 3.5. To present the real life strength of the repository, we also included two case studies of the infamous pieces of malware Duqu and Flame in Section 4. We will discuss how our system would have helped in their detection.

2. State-of-the-Art

In this chapter, we will give a short overview of previous analysis, projects and results regarding digital certificates and their primal usage: proving authenticity. These works have different aims and goals such as shedding light on the inconsistencies of the Certification Authority (CA) system, mitigating Man-in-the-Middle attacks and detecting stolen or malicious certificates.

Firstly, projects focusing on analysis and providing data will be discussed: Section 2.1 discusses SSL Observatory by Electronic Frontier Foundation and Section 2.2 reviews the ICSI Certificate Notary by University of California, Berkeley. Following the results of these projects is a proposal by Electronic Frontier Foundation for a structural change in web authentication in Section 2.3: Sovereign Keys. Sections 2.4 and 2.5 discusses two projects whose implementation would replace the CA system, mitigate Man-in-the-Middle attacks and improve Trust-on-first-use authentication. At last, Section 2.6 reviews the Google Certificate Transparency. At the end of each section, we give a short comparison of the discussed project and our work, ROSCO.

2.1 EFF SSL Observatory

Electronic Frontier Foundation (EFF) launched the project SSL Observatory to document CA behavior and search for vulnerabilities connected to digital certificates. To achieve this goal, EFF needed to collect certificates and build a large database to analyze the collected data. [8]

To collect certificates, IP address on TCP port 443 was contacted. If a server answered, the SSL Handshake protocol was run without the key agreement part. This resulted in a large number of network packages with SSL certificates inside them.

The collected certificates were stored in a MySQL database and were analyzed thoroughly for inconsistencies. The built data set was also made public and could be downloaded for a time from the website of EFF. While the links pointing to torrents are functioning at the time of write, the available torrents cannot connect to any peers to get the data anymore.

Table 2.1 shows similarities and differencies between SSL Observatory and our project, ROSCO. Both projects work with digital certificates, however, ROSCO not only focuses on certificates involved in SSL communications but on certificates used for code signing and accepts user submission as well. Furthermore, our work is not limited to digital certificates and stores information about signed code as well. One of the aims of SSL Observatory was to analyze the collected data. ROSCO has no such goal, we wish to provide information for researchers, end users and to notify organizations.

	SSL Observatory	ROSCO
Sources	SSL communications	SSL scans of the Internet, digitally signed code, user submission
Information about	Certificates	Certificates and digitally signed code
Analysis	✓	✗

Table 2.1: Comparison of SSL Observatory and ROSCO

2.2 ICSI Certificate Notary

The International Computer Science Institute (ICSI) of University of California, Berkeley started their Certificate Notary in April 2012. The aim is to help their clients to identify malicious certificates by providing a third-party perspective on what they should expect. [27]

While similar to the SSL Observatory of EFF (reviewed in Section 2.1), ICSI Certificate Notary collects passively from live upstream traffic. As of July 31, 2014, more than 2 million certificates are stored in their database. Using the collected certificates, ICSI also built the Tree of Trust to visualize connections between root and intermediate Certification Authorities.

To collect the certificates, ten organizations agreed to instrument their border gateways with a monitoring infrastructure based on the open-source Bro network monitor. The monitoring infrastructure inspects outgoing connections to extract SSL certificates into Bro logs. Logs are sent to ICSI and are stored in a database in an internal network. A text zone file is generated based on the database in DNSBL format and transferred to the externally visible Demilitarized Zone.

Clients wishing to query the database are able connect to the notary via a public DNS interface. The DNS service operates in two modes which employ A and TXT records. In both modes, the NXDOMAIN answer indicates that the certificate has not been encountered yet. In A mode, if the answer is 127.0.0.1, then the certificate has been seen, while the answer of 127.0.0.2 means that a valid chain can be constructed to a root certificate from the Mozilla root store. In TXT mode, the notary replies with details in form of space separated key-value pairs. Details include when the certificate was first and last seen, how many times the certificate was encountered and whether the certificate has been validated.

ICSI also logs notary accesses for statistical purposes. To provide anonymity, only the IP address of the resolver is seen but not the querier. If users wish to mask their locations, they can switch to public DNS services.

The notary has an auxiliary interface to the now defunct Google Certificate Catalogue.

ICSI Certificate Notary and ROSCO are very similar in a sense that both are notaries. However, while ICSI Certificate Notary provides information and meta data about certificates only, ROSCO augments this concept with signed code. Just like the Certificate Notary, ROSCO implements the Tree of Trust through verification and extends it with signed code. Not only relations between certificates but between certificates and digital signatures on files are visualized as well. These differences are shown on Table 2.2.

	ICSI Certificate Notary	ROSCO
Sources	Live upstream	SSL scans of the Internet, digitally signed code, user submission
Collection	Passive	Passive and active
Information about	Certificates	Certificates and digitally signed code
Metadata provided	Date of first and last encounter, number of times of encounter, validation	Date of first encounter, count of views, date of last view, count of explicit searches, date of last search
Tree of Trust	✓	✓(extended)

Table 2.2: Comparison of ICSI Certificate Notary and ROSCO

2.3 EFF Sovereign Keys

Sovereign Keys by Electronic Frontier Foundation is a proposal to fix structural insecurities. The design would allow clients and servers to use cryptographic protocols without having to depend on any third parties. It would also remove certificate warnings. [19]

The proposal provides an optional and very secure way of associating domain names with public keys. In the design, Sovereign Keys are created by writing to a semi-centralized, verifiably append-only data structure. The requesting party has to control a CA-signed certificate for the relevant domain, or has to use a DNSSEC-signed key to prove their control of that domain.

Master copies of the append-only data structure are kept on machines called timeline servers. As a result, Sovereign Keys are preserved as long as at least one server has remained good. For scalability, verification and privacy purposes, lots of copies of the entire append-only timeline structure are stored on machines called mirrors.

Clients learn about Sovereign Keys by sending (encrypted) queries to mirrors. Once a client knows a Sovereign Key for a domain, that fact can be cached for a very long time, with only occasional queries to check for revocations.

Sovereign Keys and ROSCO are fundamentally different. While the former aims for a structural change, the goal of the latter is to aggregate information about signed code and hasten the detection of misuse.

2.4 Perspectives

The Computer Science Department at Carnegie Mellon University started the Perspectives Project as a new approach to secure communications on the Internet. It gives users the ability to choose a group they trust, plus it improves on the basic Trust-on-first-use (Tofu) authentication. [6]

The first requirement of Perspectives is to have public notary servers that regularly monitor SSL certificates. Each network notary server is connected to the Internet and builds a public history of SSL certificates used by each website. The design has a decentralized model so anyone can run one or more network notary servers. Notaries exist independently of both clients and servers. As a result, no structural changes are needed.

The design also calls for Notary Authorities. These organizations have to determine which machines are legitimate notary servers and publish the public keys of these notaries via out-of-band communication channels. Notary Authorities must also distribute a list about legitimate notary servers as this is only way users can learn about notaries.

Users can choose which group(s) of network notaries they trust. Instead of using the CA system to validate a certificate, the browser checks the consistency of certificates observed by network notaries over time. If network notaries are spread around the world, this approach gives the 'network perspective' of a server, making the execution of Man-in-the-Middle attacks significantly harder.

While the project does improve Trust-on-first-use authentication, it has some short-comings. One structural problem is related to privacy: by contacting a notary each time the user wishes to access a site, the client is leaking browsing history. Another problem is known as the 'notary lag': if a site changes certificates between probings, the result will be invalid until a next pull. A third short-coming is that the design works for only the initial connection and does not work for anything in the background e.g. JavaScript.

Perspectives and ROSCO have not many things in common because both aims and solutions differ. Perspectives builds a history of public keys and connects them to a website. This enables the elimination of self-signed certificate warnings and mitigation of Man-in-the-Middle attacks while improving Tofu authentication. ROSCO does not connect certificates and public keys to websites but to signed code. While the meta data provided for each piece of code may help users to determine the trustworthiness of previously unseen applications, the main goals is to provide research data and help organizations keep track of what they sign.

2.5 Convergence

When Perspectives was published, the creators also included an implementation. Moxie Marlinspike used this implementation to further improve Perspectives and called his project Convergence. His aim was to implement trust agility: not only could individual users decide where to anchor their trust, they also revise their trust decision any time. He also created a new protocol and a new client-server implementation for Perspectives. [16]

The first improvement was to cease notary lag. With Convergence, when the user requests consistency check for a website, they also supply the certificate they have seen for that website. The notary does not need periodical probing this way. Instead when a user contacts the notary with a certificate, the notary contacts the website and checks its certificate. If the two match, the answer is positive, otherwise negative.

The second was the privacy problem mentioned before. The first part of the solution is local caching. When a notary gives a positive answer, that certificate can be cached for some time. The next time the user connects to the same website, the cache can be used to check consistency. The only time the notary should be contacted is when a certificate first appears or a site changed certificates. The second part of the solution is notary bounce: one trusted notary acts as a proxy to all the other notaries. The chosen proxy knows who asks but not about which site, while the notaries know which site they have to check but not for whom.

Convergence was designed to be extensible. For this reason, both the client and the notary implementations were changed. Notaries are contacted through a REST API and the notary may work in any way its deploying organization wishes. It may use DNSSEC,

verify CA signatures, etc. On the client side, the user can configure the trust threshold: an answer is accepted if the minority/majority of the notaries agree, or if the notaries are in consensus.

However, Convergence is not without problems either. The most immediate problem is called the 'Citibank problem'. Citibank has a large number of certificates and each time a connection is made to one of its servers, the server replies with a different certificate with high probability. This case is identical to the Man-in-the-Middle attack and results in a false positive alert in the browser. Another serious problem is connected to captive portals. At hotels or airports, clients cannot connect to the Internet until they provide some sensible data. But users would not provide the data without their notaries out on the Internet. This problem may be solved by implementing Convergence upon the DNS level as captive portals usually let DNS queries through.

As Convergence builds upon Perspectives, it is even more alien from ROSCO than Perspectives. All the differences mentioned at the end of Section 2.4 between Perspectives and ROSCO hold here as well.

2.6 Google Certificate Transparency

Google launched the Certificate Transparency project to provide an open framework for monitoring and auditing SSL certificates in nearly real time. The framework has two main goals. The first goal is to detect SSL certificates that have been mistakenly issued by a CA or maliciously acquired from an otherwise unimpeachable CA. The second aim is to identify CAs that have gone rogue and are maliciously issuing certificates. [2]

Certificate Transparency has three main components: certificate logs, monitors and auditors.

Certificate logs are simple network services which maintain cryptographically assured, publicly auditable and append-only records. There records can be submitted and queried by anyone and consist of certificate chains rooted in a known CA certificate. When a chain is submitted to a log, a signed timestamp is returned, which can be later used as evidence for clients that the chain has been submitted.

Monitors are publicly run servers that periodically fetch data from all log servers and watch for suspicious certificates. As a result, most monitors have complete copies of the logs they monitor. A monitor needs to, at least, inspect every new entry in each log it watches. Some monitors will be run by companies and organizations while others will be subscription services that domain owners and CAs can buy into.

Auditors are lightweight software components that typically perform two functions: verification of log behavior and cryptographic consistency, plus verification of the inclusion of a particular certificate in a log. They take partial information about a log as input and verify that this information is consistent with other partial information they have. Auditors could be an integral component of the TLS client of browsers, a standalone service, or a secondary function of a monitor. Anyone can create an auditor, although CAs are likely to run the bulk of all auditors as they have an efficient way to gain insight into the operational integrity of all CAs.

The first goal of Certificate Transparency corresponds with ROSCO as both aim to identify accidentally issued and stolen certificates. ROSCO extends this aim to signed code. However, their solutions are completely different: Certificate Transparency provides a decentralized open framework to scan untrustworthy SSL certificates. On the other hand, ROSCO uses a centralized model: if an organization wishes, it can receive

	Certificate Transparency	ROSCO
Solution model	Decentralised	Centralised
Detection of	Malicious certificates	Malicious certificates and possible pieces of malware
Monitored organizations	Certification Authorities	Certification Authorities, Code signing companies

Table 2.3: Comparison of Certificate Transparency and ROSCO

notifications about encountered signed objects (whether it is a certificate or a signed application) which can be chained back to it. Our solution of alerts is also a solution for the second aim of Certificate Transparency: through notifications, Certification Authorities can monitor both their own certificates as well as the behavior of intermediate CAs. Furthermore, our alert system can be used by code signing companies as well. As a result, not only certificates are monitored but their usage as well. The above mentioned similarities are summarized in Table 2.3.

3. Architecture

To achieve the goals set in Chapter 1, the built system has to meet the following design requirements. The system must be able to accept X.509 certificates, Portable Executables, Java Archives and Android Packages from user submission as well as download them from the Internet. It must also be able to process the above mentioned type of files and store information about them. The stored information must be accessible for clients through a browser. To hasten the detection of misused keys, the system must be able to notify users about objects possibly signed by them. Of course, it is the decision of the user whether they wish to receive such notifications and in which way or not at all.

In the rest of the chapter, we will give the high-level overview of the built system which meets the design requirements set in the previous paragraph. In each section, modules of the architecture are grouped as they achieve certain goals together, for example data collection. Section 3.1 is about data collection: the significance of crawlers and the downloading script to get possibly signed applications from the Internet. Processing of collected applications is reviewed in Section 3.2. Thirdly, Section 3.3 discusses SQL and no-SQL database technologies and their useage to store metadata and signed objects respectively. The operation of the Alert System in found in Section 3.4. At last, Section 3.5 presents the User Interface.

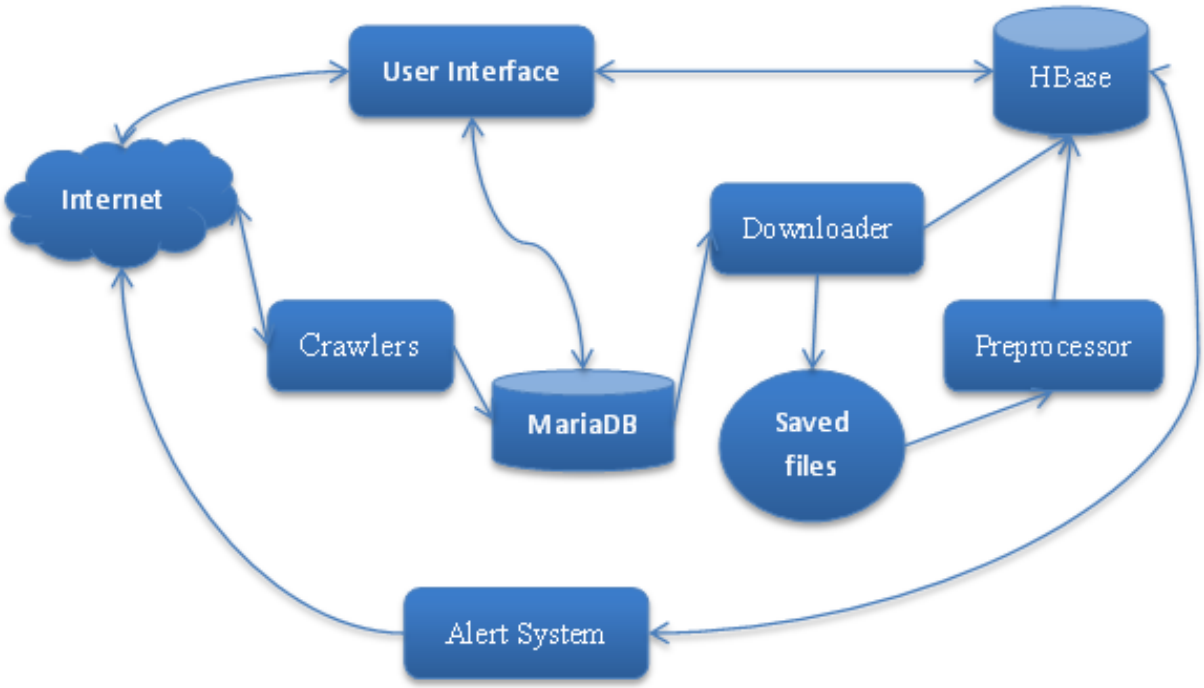

Figure 3.1: High level outline of ROSCO

Figure 3.1 shows the high level outline of the system. The architecture consists of

five major modules illustrated by rounded rectangles. These modules are: Crawlers, Downloader, Preprocessor, User Interface and Alert System. Two databases are used for data storage: MariaDB and HBase. Arrows between modules and databases visualize data flow between them.

Crawlers periodically access file sharing websites on the Internet and look for signed applications: Portable Executables, Java Archives and Android Packages. If a crawled URL or the HTML code of a website contains a URL that might possibly lead to a signed application, crawlers save the URL to MariaDB.

The Downloader module reads MariaDB for URLs to download from. After download, it checks whether the application has been seen by the system and if not, the module saves the file.

It is the job of the Preprocessor to extract information about signed objects. The module reads saved files and processes headers, meta information and digital signatures. The extracted information is then stored in HBase.

The User Interface implements users interaction with the repository through their browsers. It interprets search fields and contacts HBase to access the necessary information which is then returned to the user. Clients can also define what requirements a signed object must meet in order to notify them in form of alerts. They can also contribute to the collected information by uploading signed files and digital certificates.

The Alert System is tasked to interpret user-defined alerts and look for signed objects matching the specified criteria. When an alert triggers, the Alert System notifies the client via e-mail or RSS feed. The way of notification is decided by the user.

3.1 Data Collection

In this section, we will discuss how to collect data actively and in an automated manner.

Firstly, Section 3.1.1 will review Scrapy, a Python-written crawling framework, and mention several websites from which signed application can be downloaded. We crawl the Internet using Scrapy and collect URLs which may lead us to signed application. These URLs are stored in a database for download.

Secondly, the Downloader module in Section 3.1.2 is tasked with downloading possibly signed applications from collected URLs. The URLs are accessed from a database and downloaded only if they have never been seen by the system.

3.1.1 Crawlers

As mentioned before, users may submit signed applications and digital certificates to the system but this is not the only way to collect data. There are numerous signed applications on the Internet which can be collected in an automated way with crawlers. A crawler is essentially a small robot which browses the Internet and collects information. The most widely known case is the Google search engine which uses crawlers to index webpages to the database of the company.

In our case, each crawler should only crawl a predefined part of the Internet, a single domain. Also, the crawler needs to inspect every webpage it browses whether the site is a link to a signed application or it references a signed object. The easiest way to achieve this is through regular expressions as URLs often contain the name of the application and thus a file extension.

Scrapy is a high-level web crawling framework with all the features we mentioned above. [24] The usage of this tool is extremely easy, writing a crawler is like creating configuration files. The developer needs to give the domain in which the crawler should work and state which domains must be excluded. Once the crawler runs, it feeds the engine of the framework with the HTML codes of the visited sites. The engine forwards the HTML codes to a developer-created method, a callback. This callback method inspects the site and determines whether it meets the requirements. If it does, the URL to the webpage is stored in a database. The work flow is shown in Figure 3.2.

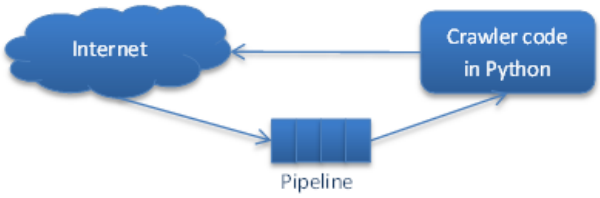

Figure 3.2: High-level Operation of a Crawler

Data samples are collected from various file sharing sites including SourceForge, Download3000, Github, Googlecode and APKfiles. To write an effective crawler, the URL handling of the site must be examined before creating the robot. Namely, used URL conventions and references to other domains in the HTML code. Once conventions are known, regular expressions can be used to detect applications in the URL or HTML code using file extensions, e.g. EXE, JAR, APK, SYS, etc.

The collected URLs are stored in a database for the Downloader module.

3.1.2 Downloading

The Downloader module is tasked with downloading applications from the collected URLs. However, collected links may be dead or may not contain the necessary resource. As a result, the Downloader must be able to differentiate between acceptable and unacceptable resources.

Figure 3.3 shows the flowchart of the module. It works as an infinite loop: if there is a link in the database, it processes the link, if not, it waits. The module tries to connect to the URL and waits for a response. There might be redirection in the background but that does not affect processing. If the connection could not been created or the returned HTTP Status code is not 200 (which is OK), the link is considered unnecessary and is deleted from the database. In case of a successful connection, the processing of the link can start.

What the module needs to determine is whether a URL points to an application or not. To determine if an application is signed is the responsibility of the Preprocessor in Section 3.2. The easiest way to determine if the content is indeed an application is the Content-Type header of the HTTP response. If the header is present, it contains information about the body of the response. If it is set to 'application', the URL indeed points to an application. Unfortunately, this header is not always included in the response. If that is case, it is known that applications are binary data and thus contain unprintable non-ASCII characters. Testing the content of the HTTP response against this criteria tells with a high percent of accuracy whether the processed URL points to an application. If either of this tests fail, the link is deleted from the database.

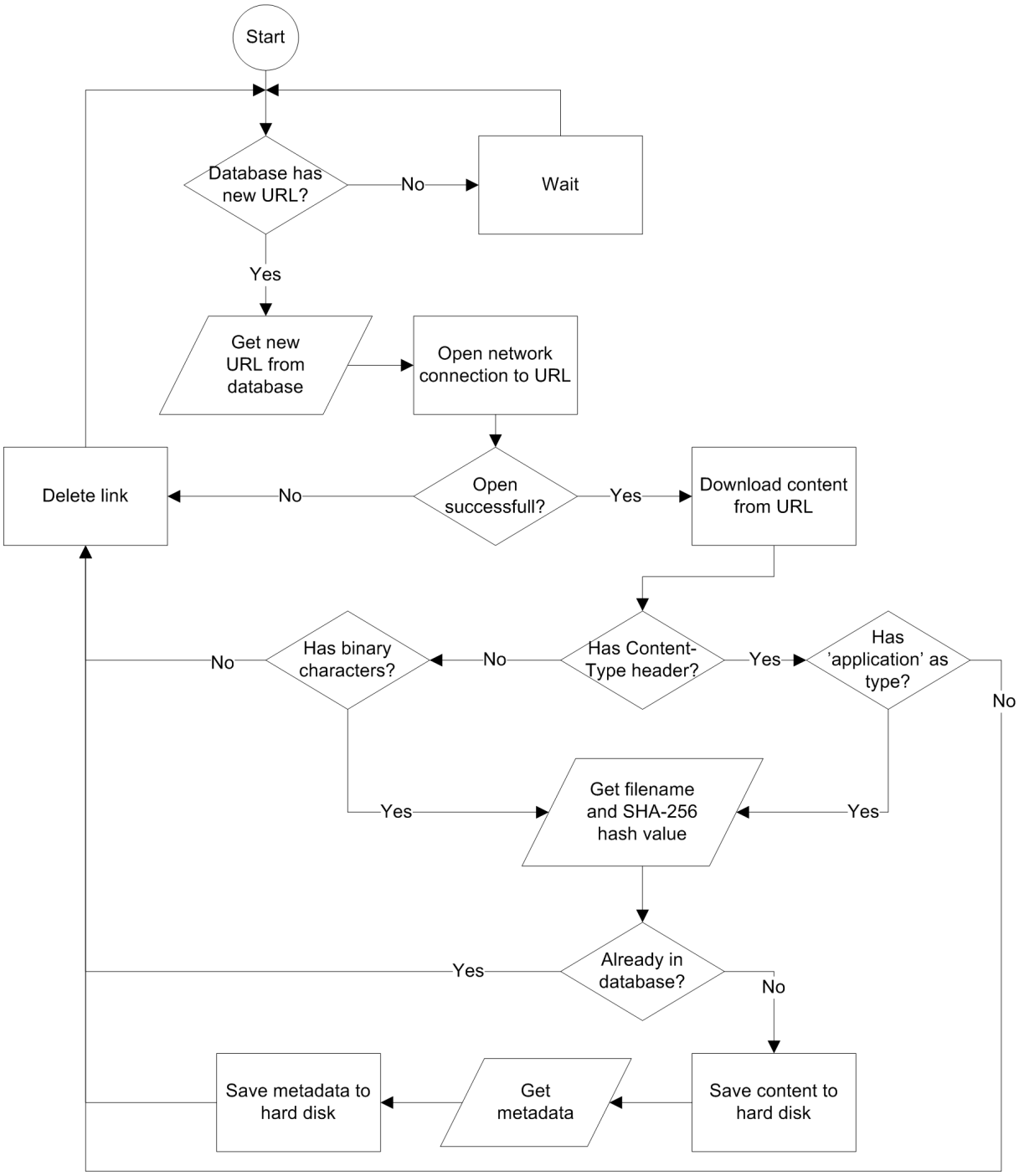

Figure 3.3: Downloader Module

After determining that the downloaded data is an application, the module checks the file name and calculates its SHA-256 hash value. There are two places which may contain the file name: the Content-Disposition header and the URL itself. As is the case with the Content-Type header, the Content-Disposition header contains meta data about the underlying application, one of which is the file name. This is the same file name as what is used on the server. If the header is missing, the URL itself may contain the file name, e.g. http://somesite.com/an_application.exe.

There is only one last check to be performed: whether the system has already processed this file. If it has, the link is deleted. If not, the application is saved to the hard disk and

meta data about the URL is gathered. Meta data include the link to which the module originally tried to connect and URL it was redirected to. Meta data is again saved to the hard disk in JSON format for the Preprocessor.

3.2 Data Processing

As mentioned in Section 3.1.2, previously unknown applications are saved to the hard disk with meta-data in JSON format. The Preprocessor reads new applications and processes them with meta data into the repository.

In this section, we will review the collected file formats and their digital signatures as well as give the high-level overview of the Preprocessor and a basic tutorial on digital signatures. Firstly, we will discuss digital signatures as they appear in textbooks and the standard that regulates its use in applications. Next is the high-level overview of the Preprocessor. Thirdly, in Section 3.2.1, public keys and digital certificates will be reviewed. In Section 3.2.2 the Portable Executable (PE) file format is discussed. At last, Section 3.2.3 gives a short introduction to Java Archives (JARs) and Android Packages (APKs).

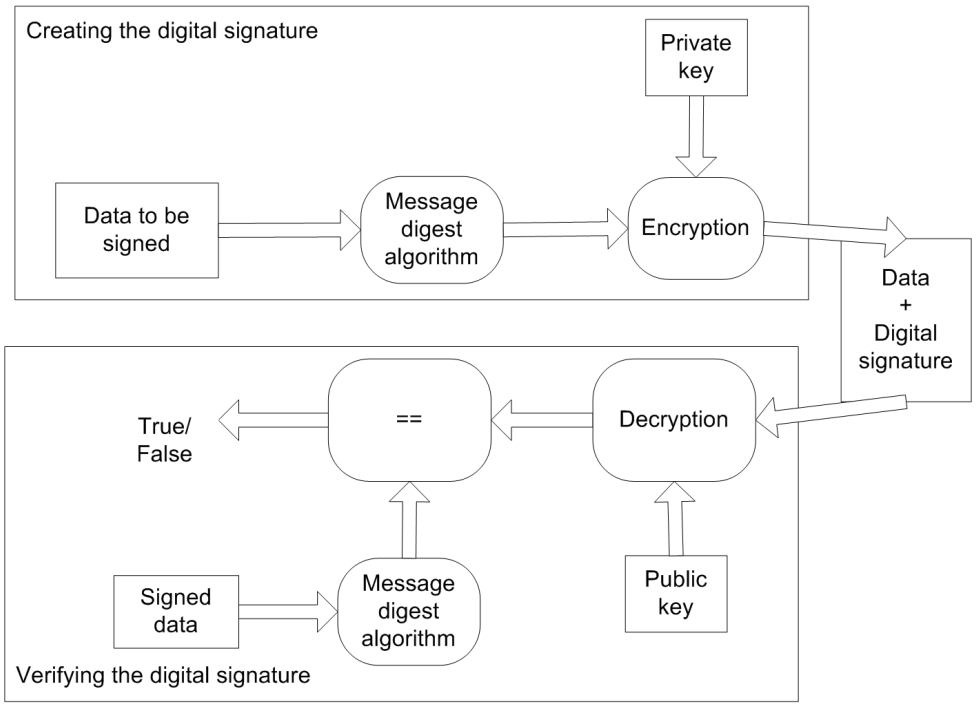

Figure 3.4: Creating and Verifying a Digital Signature

Digital signatures are used to provide authenticity. Each signer possesses a key-pair: the private key is kept secret, while the public key is made known to everyone in the infrastructure. To create a digital signature, the signer computes the hash value (also known as fingerprint value or message digest) of the data with a message digesting algorithm. The resulting bit-string is encrypted with the private key and appended to the data as shown in Figure 3.4. During verification, the hash value is computed again from the data and the signature is decrypted with the public key. If the decrypted and computed hash values match, the signature is considered valid.

The structure of a digital signature in binary applications is regulated by RFC 2315 [1], commonly known as the Public-key Cryptography Standards #7: Cryptographic Message Syntax. The standard enables multiple signers to sign the same data and supply their public keys to end-users. There are two structures that contain digital signatures: the SignedData and the SignedAndEvelopedData structures. The latter is the combination of SignedData and EnvelopedData but the structure and usage of EnvelopedData is outside of the scope of this paper.

```
SignedData ::= SEQUENCE {
  version Version,
  digestAlgorithms DigestAlgorithmIdentifiers,
  contentInfo ContentInfo,
  certificates
     [0] IMPLICIT ExtendedCertificatesAndCertificates
       OPTIONAL,
  Crls
    [1] IMPLICIT CertificateRevocationLists OPTIONAL,
  signerInfos SignerInfos }

DigestAlgorithmIdentifiers ::=
  SET OF DigestAlgorithmIdentifier

ContentInfo ::= SEQUENCE {
  contentType ContentType,
  content
    [0] EXPLICIT ANY DEFINED BY contentType OPTIONAL }

ContentType ::= OBJECT IDENTIFIER

SignerInfos ::= SET OF SignerInfo
```

The ASN.1 definition of SignedData is found above. It has a version information which is always 1 at the moment but may change in future versions. All the message digest algorithms are listed that were used by all signers. The Content field may be anything, so some signing techniques make use of this field to sign more files at once (see Section 3.2.2). If it is set, then the signature is computed with it as input. If not, the specification of the file format tells which fields the signature is computed on. All the certificates used by signers are listed as well and certificate revocation lists may be added. At last, information about signers is added along with the computed digital signature as Signer Info structures.

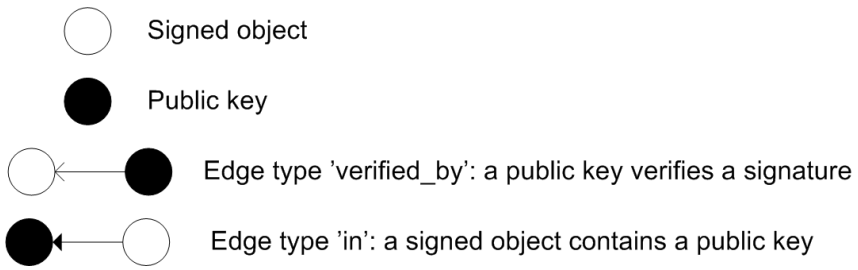

Figure 3.5: Notation of the Extended Tree of Trust

Relationships between public keys and signed objects can be represented by a graph. In the graph, public keys and signed objects are represented as nodes and relations between them are edges. There are two types of edges: 'in' stands for a signed object (digital certificate) containing a public key and 'verified_by' means that the signature of a signed object can be verified by a public key. Our notation for the graph is shown in Figure 3.5.

Certain rules and constraints can be defined for the resulting graph. The aim of the Preprocessor is to store signed objects, public keys and relations in a way that the graph can be reconstructed. Such relations and scenarios include:

1. Self-signed certificate
2. Certificate chain
3. The digital certificate of a signed object is verified by the public key contained in a digital certificate
4. A public key is contained in a digital certificate
5. Add new element to a chain
6. Two public keys must not verify the same signature. Else the signature scheme is broken
7. Two or more digital signatures may contain the same public key

The scenarios mentioned above are depicted by Figure 3.6.

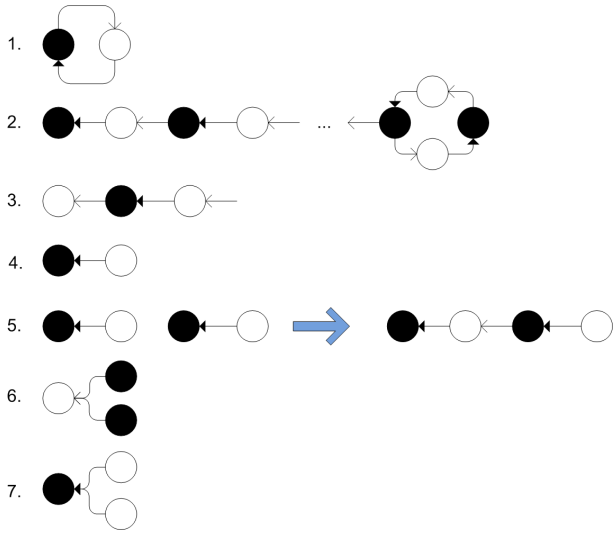

Figure 3.6: Semantics of the Extended Tree of Trust

Figure 3.7 shows the high level overview of the Preprocessor for a single file. There are four kinds of signed objects the module handles: Portable Executables, Android Packages, Java Archives and digital certificates. The Preprocessor works with presumably signed objects and runs in an infinite loop but we will only cover the processing of a single file. The first step is determining the file type. If a file format is unknown to the Preprocessor, it throws an exception and halts. For a known file format, the module checks whether it is digitally signed according to its specification. If it is not, processing ends and the file is deleted from the server. Certificates are extracted from each signed file and processed

separately. When the certificates are stored in the database, verification is run on the file. This enable us to reconstruct a Tree of Trust similar to ICSI Certificate Notary. However, our Tree is extended with the processed signed files.

If the input file is a Portable Executable, the Preprocessor inspects its header for information about the file. If the input is and Android Package (APK), it is first processed as if it was a Java Archive (JAR) as APKs are based on JARs. Then the AndriodManifest.xml is inspected to recover information connected to the execution on Andriod OS. In case of JARs, the META-INF folder has almost every information needed: application related data, digital signatures and file hashes. These files are compressed conforming to the ZIP standard, so compression data can also be extracted. For digital certificates, the X.509 standard regulates fields and attributes. The Preprocessor follows the guidelines of this standard to parse digital certificates.

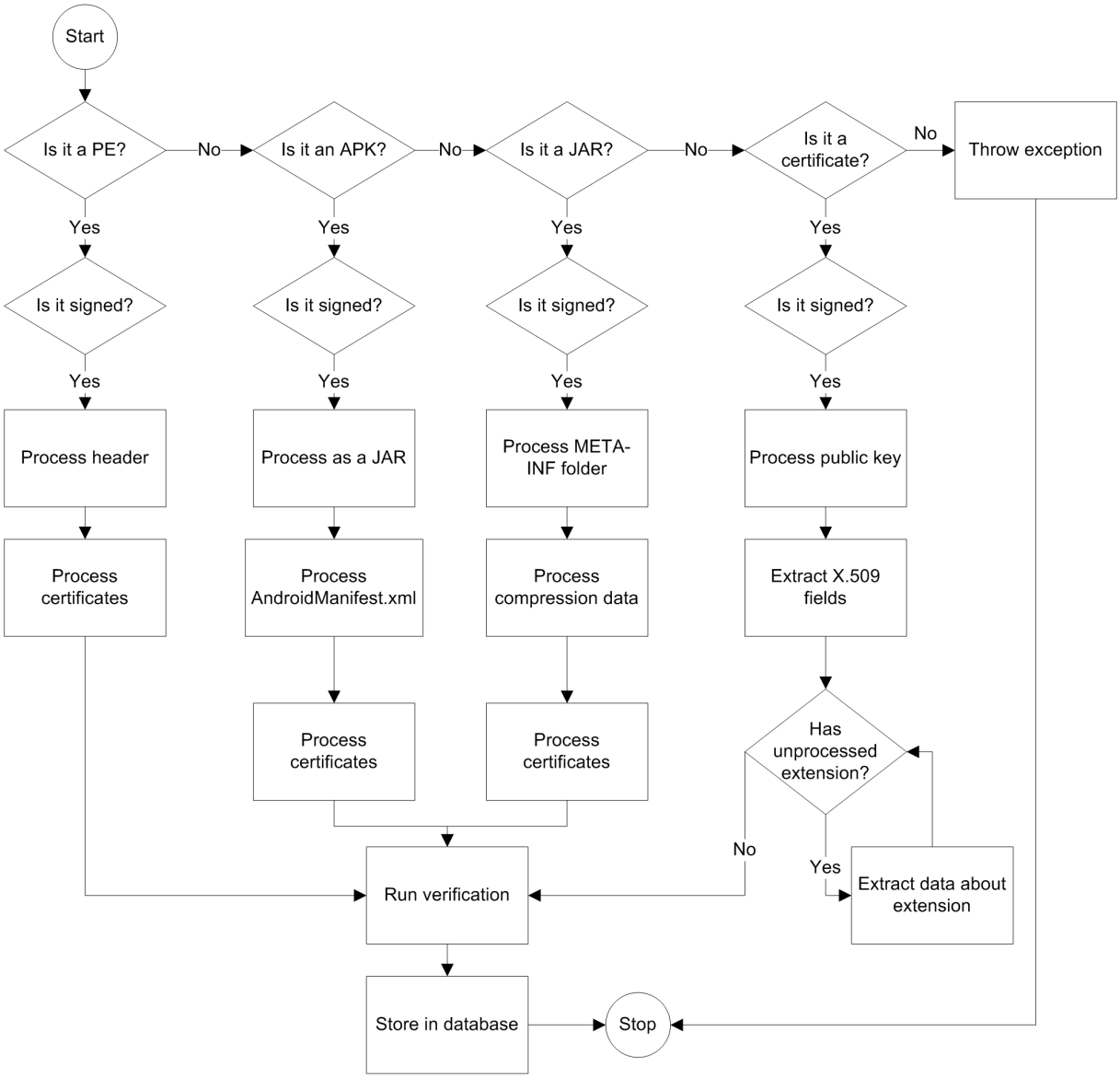

Figure 3.7: High-level Review of the Preprocessor

3.2.1 Public keys and Certificates

This section discusses different digital signatures and related public key types. To understand the need for digital certificates, we need to take a look at the history of cryptography. Until 1976, only symmetric key cryptography existed. In this symmetric key cryptography, encryption and decryption uses the same key. It is fast and its algorithms use less resources but the key must be negotiated before its actual usage. This poses a significant drawback: there are millions of Internet users who wish to communicate securely. It is unrealistic to assume that keys can be generated for each pair in advance. Whitfield Diffie and Martin Hellman found the solution and published their new idea of public-key cryptography in [28].

Public-key cryptography uses a key-pair: a private and a public key. The generation of such a key-pair is based on mathematical problems thought to be extremely hard to solve such as the discrete logarithm problem or large prime factorization. These problems are also trapdoor one-way functions: the inversion of the function is computationally infeasible unless some secret information, the trapdoor (the private key), is known.

The first and still the most widely used public key scheme was created by R. L. Rivest, A. Shamir and L. Adleman. [23] The name of this scheme comes from the surname of its creators: RSA. The security of RSA keys is based on the large integer factorization problem. Even though the problem has never been proved to be difficult, mathematicians have been working on a general solution for more than three hundred years. The public key has two components: a modulus (n) and an exponent (e) as shown in Table 3.1.

Rivest-Shamir-Adleman	
Components of the public key	• n - Modulus • e - Exponent

Table 3.1: RSA Public Keys

The National Institute of Standards and Technology (NIST) in the United States published the Digital Signature Algorithm (DSA). [17] The standard was revised several times since then, the most recent paper was published in July 2013. The algorithm is patented, the owner is the United States of America. The secrecy is based on the discrete logarithm problem (DLP) in prime-order subgroups of $\mathbb{Z}_p{}^*$ which is believed to be difficult. No general method is known to solve the problem on conventional computers. The digital signature is computed using a set of domain parameters and a per-message secret number k. Table 3.2 summarizes the public key.

Digital Signature Algorithm	
Components of the public key	• p - Prime modulus • q - Prime divisor of $(p-1)$ • g - a generator of a subgroup of order q in the multiplcative group of GF(p)

Table 3.2: DSA Public Keys

Elliptic curve cryptosystems were invented in 1985. They are elliptic curve analogues for the discrete logarithm cryptosystems where the subgroup of $\mathbb{Z}_p{}^*$ are replaced by a group of points on an elliptic curve over a finite field. Just like the discrete logarithm problem, the elliptic curve discrete logarithm problem (ECDLP) is also believed to be unsolvable by conventional computers. However, ECDLP appears to be significantly harder than DLP so the same level of security can be achieved with smaller parameters. ECDSA was first proposed in 1992 and by 2000, it has been accepted as ISO, ANSI, FIPS [17] and IEEE standards. NIST recommends three types of curves but domain parameters may be generated according to X9.62. An ECDSA key pair consists of a private key d and a public key Q with associated domain parameters. The public key itself is a point on the elliptic curve, given by its coordinates. d and Q are in mathematical relation with their domain parameters. Table 3.3 summarizes the public key of ECDSA scheme. [7]

Elliptic Curve DSA	
Components of the public key	• x - X coordinate of the point on the curve • y - Y coordinate of the point on the curve • *curve_name* - Name of the used elliptic curve

Table 3.3: ECDSA Public Keys

To preprocess a public key, it must be loaded from a certificate. After load, the key type must be determined. If the key is an RSA public key, the modulus (n) and he exponent (e) are extracted from the key with the key-length. In case of DSA public keys, the prime modulus (p), the prime divisor (q) and the generator (g) are extracted. The last expected key-type is ECDSA public keys in which case the coordinates (x, y) of the key and the curve name are extracted. In each case, extracted information is stored in the database along with the length of key in bits. There are other types of public keys as well, but their occurrence is such a rare event that we decided not to support them.

In a Public Key Infrastructure, public keys need to be distributed between parties. While confidentiality is not an issue, authenticity is of vital importance as public keys must be associated with entities in the infrastructure. The concept of certificates was invented by Kohnfelder in 1978 [10] and its basic idea is to bind subject identification information with the public key via the digital signature of a trusted entity, the Certification Authority (CA).

The International Telecommunications Union - Telecommunication Standardization Sector standardized digital certificates in X.509: RFC 2459 [22] records fields, extensions and constraints. A certificate contains information about its owner (Subject) and the issuing authority (Issuer). The standard does not state mandatory subfields in names, only that the resulting name must be unique. The PublicKey field contains the public key discussed before while the validity is given by the ValidFrom and ValidTo fields. The information of the certificate must be considered valid in this period. To protect the integrity of the certificate, it is signed with the private key of the issuer; the signature and the used algorithm are stored in the Signature and SignatureAlgorithm fields, respectively. Extensions were introduced in RFC 5280 [5] in May, 2008. The standard defines the most commonly used ones but organizations may define their own. Table 3.4 summarizes these fields as well as the attributes stored in our repository.

Digital certificates can be collected from various sources: instances of the SSL Hand-

shake protocol, projects similar to SSL Observatory and digitally signed files. We used the data set of Project Sonar, an Internet-wide scan of SSL certificates by Rapid7 Labs [13] as well as the December, 2010 data dump of SSL Observatory. Using crawlers, we also collected numerous signed applications which contain certificate chains.

CERTIFICATE		
Serial Number	Used to uniquely identify the certificate with respect to the issuer	
Issuer	The entity that verified the information and issued the certificate	
	CN:	stands for common name
	ST:	state of residence
	O:	organization
	C:	country of residence
	E:	email address
	T:	locality
	OU:	organization unit
	STREET:	street address
	ALL:	complete distinguished name
ValidFrom	The date the certificate is valid from.	
ValidTo	The expiration date.	
Subject	The entity for whom the certificate was issued	
	CN:	stands for common name
	ST:	state of residence
	O:	organization
	C:	country of residence
	E:	email address
	T:	locality
	OU:	organization unit
	STREET:	street address
	ALL:	complete distinguished name
PublicKey	Public key	
Signature	The digital signature which is used to verify that that the certificate came from the referenced issuer.	
SignatureAlgorithm	The algorithm used to create the signature: a message digest algorithm with an encryption algorithm	
Extensions		
ExtensionType	Type of the extension.	
Value	Value of the extension.	
IsCritical	Is this extension critical?	

Table 3.4: Fields of Digital Certificates

When the fields in a certificate have been extracted, verification is run. Our database is searched for the certificate of the issuer and any other signed object the public key may have signed. Unfortunately, there is no standardized way to construct a certificate chain. RFC 4158 [11] contains optimization best practices but the recommendations only show how to exclude certificates from the candidate pool. Our database holds millions of

certificates so exclusion would still yield such a large candidate set that validating each member of the set would likely take several months. What we need is a straightforward way to find the pool of possible matches with the least cardinality. After much consideration, we settled for searching by Common Name fields. This will not give us the complete list, however, as each digital signature is verified by the textbook method mentioned in the introduction, relations found are always correct.

3.2.2 Portable Executables

Microsoft introduced the Portable Executable file format as part of the Win32 specifications, even though it is derived from the earlier Common Object File Format (COFF). The term 'Portable Executable' was chosen because the intent was to have a common file format for all Windows operating systems, on all supported CPUs. The addition of 64-bit Windows required some modifications, and the resulting format is called PE32+.

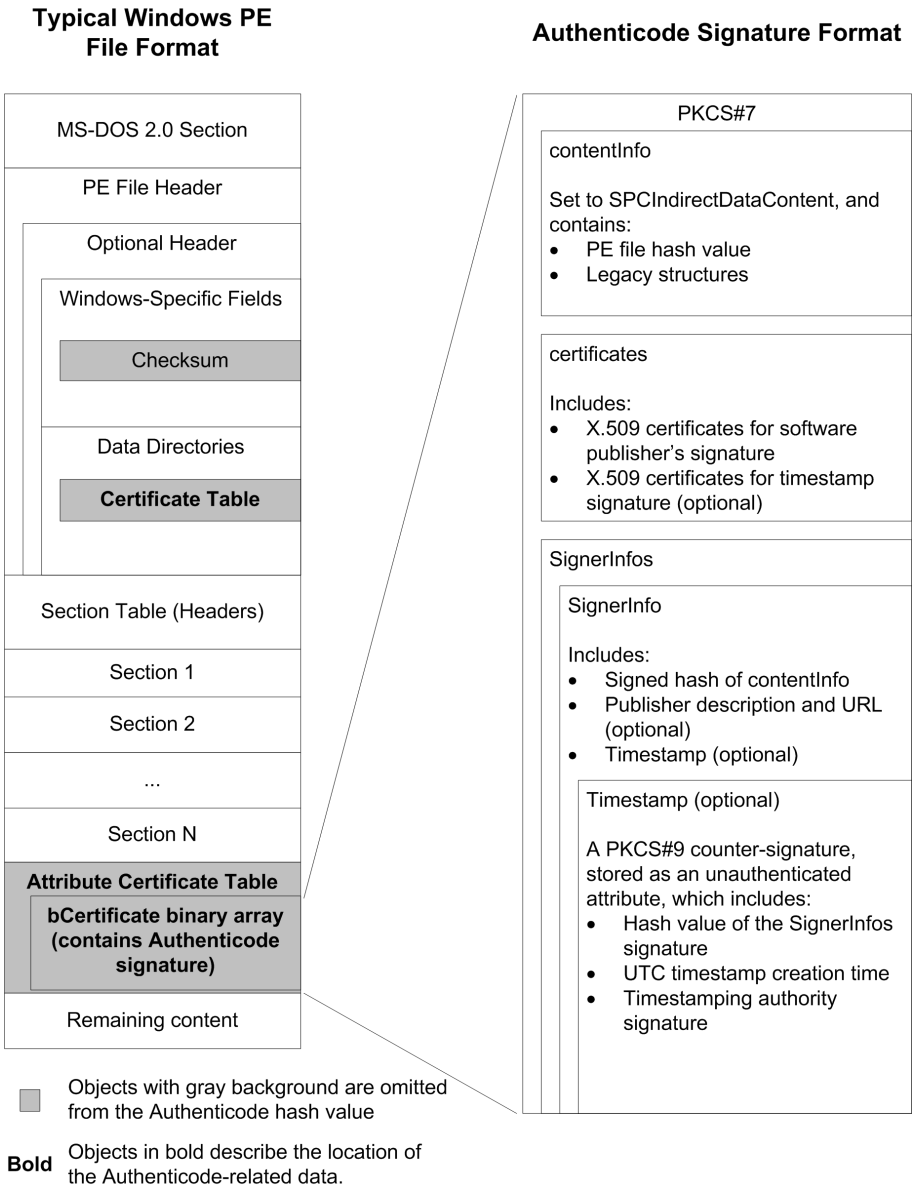

Figure 3.8: Portable Executable File Format

Thanks to the PE file format, executables and dynamically linked libraries (DLLs) differ in only a single bit. Even the extension is artificial e.g. Control Panel applets (CLP files) are DLLs in fact.

Figure 3.8 illustrates the typical layout of a Portable Executable and its digital signature. [15, 14] The file begins with the MS-DOS stub for backward compatibility with MS-DOS systems. However, if a PE were to be run on an MS-DOS operating system, an error message would be printed: 'This program cannot be run in DOS mode'. The stub contains the file offset to the new header stub: PE File Headers. This stub begins with the PE signature (which is not the digital signature), the letters 'P' and 'E' followed by two null bytes. This stub contains several smaller stubs. The COFF stub has important data about the executable such as the target CPU, the creation date and time of the file and attribute flags. Following the COFF stub is the Optional Header that provides information to the loader. It states among others the linker version, the required operating system and subsystem versions. The Section Headers after the Optional Header may be omitted. Some sections contain code or data that the program declared and uses directly, while other data sections are created by the linker and librarian, and contain information vital to the operating system. The rest of the file is the raw data.

Most data stored in the header is important for the operating system and the loader. However, certain fields carry information which can be interpreted by end users as well. We extract these information from the header and store them in the database. Table 3.5 summarizes the fields which contain such information. The characteristics of a PE is a number, the result of OR operation of 15 flags. Machine stands for the CPU type the file was complied for, and is a number as well. The association between numbers and CPU types can be found in [15]. Compile date shows when the PE was compiled and the LinkerVersion tells which version of the linker was used. The minimum operating system requirement can also be found in the header as well as information about the PE type (EXE, DLL or driver).

PORTABLE EXECUTABLE	
Characteristics	Characteristics of PE (15 flags OR-ed together)
Machine	Target CPU
CompileDate	Datetime of compiling
LinkerVersion	Version of the linker used to create the file (major and minor)
MinimumOS	Minimum operating system version requirement (major and minor)
Type	Whether the file is an EXE, a DLL, a driver or unknown

Table 3.5: Stored Attributes of Portable Executables

Microsoft created the Authenticode technology to digitally sign Portable Executables. As mentioned in the introduction of the chapter, the technology makes use of the Content field in the PKCS #7 structure. As a result, there are two types of signatures for Portable Executables: embedded and detached.

In case of embedded signatures, the digital signature is placed inside the PE header. It can be found in the Certificates Table entry in the Optional Header and the associated Attribute Certificate Table. The Content field of the PKCS #7 structure is set to a

Microsoft specific structure called SpcIndirectDataContent and contains the hash value of the file, file descriptions and various optional or legacy ASN.1 fields.

However, thanks to the flexibility of the PKCS #7 standard, the signature may not be present in the file but instead, it may be contained in a different file. Authenticode calls this files catalogs and stores their information in a central database on the computer. A catalog file is essentially a PKCS #7 structure with its Content fields set to another Microsoft specific structure called Certificate Trust List. Unfortunately, it is not well documented and everything we know about it comes from our experience. The Certificate Trust List (CTL) contains several SpcIndirectDataStructures with file hashes. If the signature of the catalog file is considered valid, then all files with their hashes listed inherit the valid signature. When a product ships with a catalog file, it must be installed on the target system via the Windows API. The catalog is stored in a central database and each time the shipped application is run, Windows verifies its signature by searching for a valid catalog in which the file is listed.

To verify an Authenticode signature, three structures need to be hashed together: the ContentType, the DigestInfo and the SpcSpOpusInfo. The first one is part of the SignedData structure, the second is found in the SpcIndirectDataStructure and the latter is an authenticated attribute for each signer. Their ASN.1 definition can be found below.

```
DigestInfo ::= SEQUENCE {
    digestAlgorithm     AlgorithmIdentifier,
    digest              OCTETSTRING
}

SpcSpOpusInfo ::= SEQUENCE {
    programName             [0] EXPLICIT SpcString OPTIONAL,
    moreInfo                [1] EXPLICIT SpcLink OPTIONAL,
} --\#public--
```

3.2.3 Java Archives and Android Packages

Both Java Archives and Android Packages are based on the ZIP file format. ZIP is a cross-platform, interoperable file storage and transfer format. [20] It is one of the most widely used file formats for compression. The compressed data must be created from at least one file.

ZIP SPECIFIC ATTRIBUTES	
CreateDatetime	It is an interval of the first and last created archive members' creation date and time
CompressType	It is a set of members' compression type attributes
CreateSystem	It is a set of what kind operating system was used to create its members
CreateVersion	It is a set of values which define what version of the compression tool was used when creating the members
ExtractVersion	It is a set of values which define what version of the decompression tool can be used to decompress the members

Table 3.6: Stored attributes of ZIP Archives

Inside the container, compressed files have local file headers and encryption headers. The local file header contains several fields such as the version needed to extract, compression method, date and time of last modification and so on. As these information apply to a single compressed file, these information are stored as intervals or sets in our repository. For each Java Archive and Android Package, we give the intervals and sets of members' ZIP specific attributes. Table 3.6 summarizes these fields.

As mentioned before, the Java Archive (JAR) file format is based on the ZIP file format and aggregates many file into one. [18] In many cases, it is not simply an archive but used as a building block for other applications and extensions. It contains the special META-INF folder which stores configuration data and packages. Three files are of interest for our project:

- MANIFEST.MF
- x.SF where x is the file name
- digital signature

The MANIFEST.MF is referred to as the manifest file and defines extension configuration and package related data. The manifest file can be split into two halves: the main section and the list of sections for contained files (individual sections). The main section contains security and configuration information about the whole archive plus the extension or application the JAR depends on, if there is any. Individual sections define various attributes for packages and/or files they refer to. Not all files in the archive need to be listed, but the signed ones must.

Files with .SF extension are signature files and specify which files the digital signature protects. Its structure is similar to the manifest file: it has a main section and a list of individual files. The main section includes information supplied by the signer but this information is not specific to any file. Each individual entry contains a hash value of the corresponding entry in the manifest file.

The digital signature is computed over the signature file and contained in a separate file with varying extension but the file name is the same. In case of .RSA, the digital signature way created by using the SHA-256 message digest algorithm with RSA encryption. If the signature was created with the Digital Signature Algorithm, the extension is .DSA. For every other method of signature creation, the file begins with SIG-. There may be more than one signer for a single Java Archive in which case each signature file and digital signature pair have different file names.

While processing a Java Archive, the Preprocessor extracts ZIP specific information about the compressed member of the archive according. This information is stored as intervals for dates and sets for any other arbitrary data. The module than parses the Manifest and Signature files according to its specification and computes the SHA-256 message digest of members. Table 3.7 summarizes the stored information.

Verification of the digital signature in case of Java Archives is well defined in the specification. However, the method is quite complex and is usually done by the jarsigner tool. This tool not only verifies the digital signature on a file but also performs security checks to check their integrity. As a result, we need to determine only the signing certificate to place the application into the Tree of Trust, any other checks can be performed by the tool. This can be done by running the textbook verification shown in Figure 3.4 with the contents of the Signature file as input.

One interesting thing about Java Archives is that not all members of the archive must be signed. In this case, verification can only be successful partially. While performing security checks, the jarsigner tool also focuses on the problem and returns a warning message to signal the issue. If a signature protects an archive only partially, the information is stored in the repository as it is a vital information for users of that application.

JAVA ARCHIVES		
Main section of Manifest file	Manifest-Version	Defines the manifest file version
	Created-By	Defines the version and the vendor of the Java implementation on top of which this manifest file is generated. This attribute is generated by the jar tool
	Class-Path	The value of this attribute specifies the relative URLs of extensions or libraries this application or extension needs.
Main section of Signature File(s)	Created-By	The version and vendor of the signing software. (Typically information about jarsigner)
InnerFiles-AndHashes	Files and their SHA-256 hashes in the JAR	

Table 3.7: Stored attributes of Java Archives

Android is a popular mobile platform in more than 190 countries and with a constantly growing number of users. Android applications are distributed and stored in Android Packages (APKs) which are based on the Java Archive file format but introduce some restrictions. These restrictions specify what folders and files a package must have. As for the META-INF folder, there can be only one signer for an APK. The package must contain the AndroidManifest.xml file in its root directory which is a binary XML. It stores essential information about the application to the Android System such as the name, version, access rights and referenced library files of the application. [9]

ANDROID MANIFEST FILE	
Permissions	A restriction limiting access to a part of the code or to data on the device
Package name	Names the Java package for the application
Android version	Minimum level of Android API that the application requires
Activities	Implements part of the applications visual user interface
Services	Implements long-running background operations or communications API
Receivers	Enables applications to receive information that are broadcast by the system or other applications
Providers	Supplies structured access to data managed by the application

Table 3.8: Stored attributes of Android Packages

To store information about Android Packages, the Preprocessor processes the file as a Java Archive and extracts information from the AndroidManifest.xml file. Table 3.8 summarizes these information.

Verification of Android Packages can be done the same way as with Java Archives. The only restriction is: an APK must only have one signer.

3.3 Data Storage

In this section, data storage technologies and practices used by ROSCO will be discussed. In the first part, the SQL-based MariaDB will be reviewed in which metadata about users and alerts are stored. Discussed in the second part, the non-SQL HBase is used to store information about signed objects and public keys.

3.3.1 Metadata

There are tree types of metadata in our project: links to be downloaded, data about alerts, and user data. The download links are URLs which may or may not point to signed code. It is the responsibility of Crawlers to search for these URLs and save them. The Downloader module uses these links to download applications for the Preprocessor. Each alert used by the Alert System is a row in the database. They are connected to a user and contain criteria about signed object. The Alert System uses these criteria when it searches for matching objects. User data is used by both the User Interface and the Alert System. It contains user names, passwords and e-mail addresses.

We use the SQL-based MariaDB to store metadata. MariaDB is an enhanched replacement for MySQL which improves speed and has more included storage engines. [12] It is also open-source. For example, the Aria storage engine enables faster complex queries (queries which normally use disk-based temporary tables). It is used for internal temporary tables, which should give a speedup when doing complex selects. Aria is usually faster for temporary tables when compared to the basic MyISAM because Aria caches row data in memory and normally doesn't have to write the temporary rows to disk.

MariaDB is a good choice for metadata storage because SQL-based databases generally perform better in heavily transactional environment. Both the Alert System and the Downloader can be considered heavily transactional applications as the data requested by them has to arrive in one piece. How could we download a file if only a part of the URL was given? Not to mention the complex select statements used by the Alert System. Because of the CAP theorem (see Section 3.3.2), non-SQL databases could not perform even remotely as well as SQL-based ones do.

Downloadable links are stored separately from user related data. As such, they are in a separate database, contained in a single table as depicted by Figure 3.9. The table has two columns: id and link. The column link contains the URL from which the resource has to be downloaded. Its type is text as RFC 2616 (Hypertext Transfer Protocol – HTTP/1.1) impose no limitation on the length of URLs. [21] On the other hand, the de facto limit is 2000 characters, popular browsers do not support more. However, this number is not exact and while the probability of finding a URL with more than 2000 characters is extremely low, it cannot be excluded. Unfortunately, MySQL and thus MariaDB are unable to index variable length columns such as text. As a result, a dummy column was created to act as the primary key.

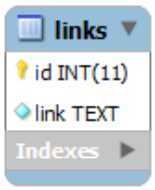

Figure 3.9: Table of Downloadable Links

Figure 3.10 shows SQL tables related to users. The table user contains all information about a single account: user name, password, e-mail address and the creation date of the account. The table login stores information about when and how users interacted with the system, the time of login and logout.

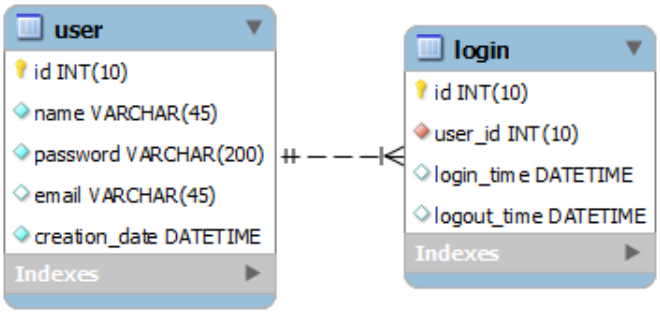

Figure 3.10: User Related Metadata in the Database

SQL tables shown in Figure 3.11 are related to alerts: notifications about signed objects for users. As all alerts must be connected to a user, the table user is shown in the figure.

The alert table contains information about alerts. Users can name their alerts for ease of interaction with the system as well as activate, deactivate and later reactivate them as they find convenient. Users can define in which way they want to be notified: e-mail, RSS, both or neither. There are two kinds of alert: simple alerts and signing key usage alerts. With a simple alert, clients can get notifications whenever certain fields in an object matches their criteria. For this purpose, the column string stores the criteria while columns hb_table and hb_column contain the table and column of HBase in which to search for matching objects. If a user wishes to be notified when the system encounters objects signed by a specific private key, they define signing key usage alerts. In this case, the column kid stores the identifier of the supplied public key in HBase.

Signed objects which triggered alerts are stored in the table matched. Using this table, objects matching a specific alert can be retrieved from the database. Even if a user does not define a way for notification, they can see matching objects on the User Interface (see Section 3.5) with the date and time of match.

The new_signed_objects table stores signed objects identifiers for which alerts need to be run. The only column of the table stores the identifier of the newly processed object in HBase.

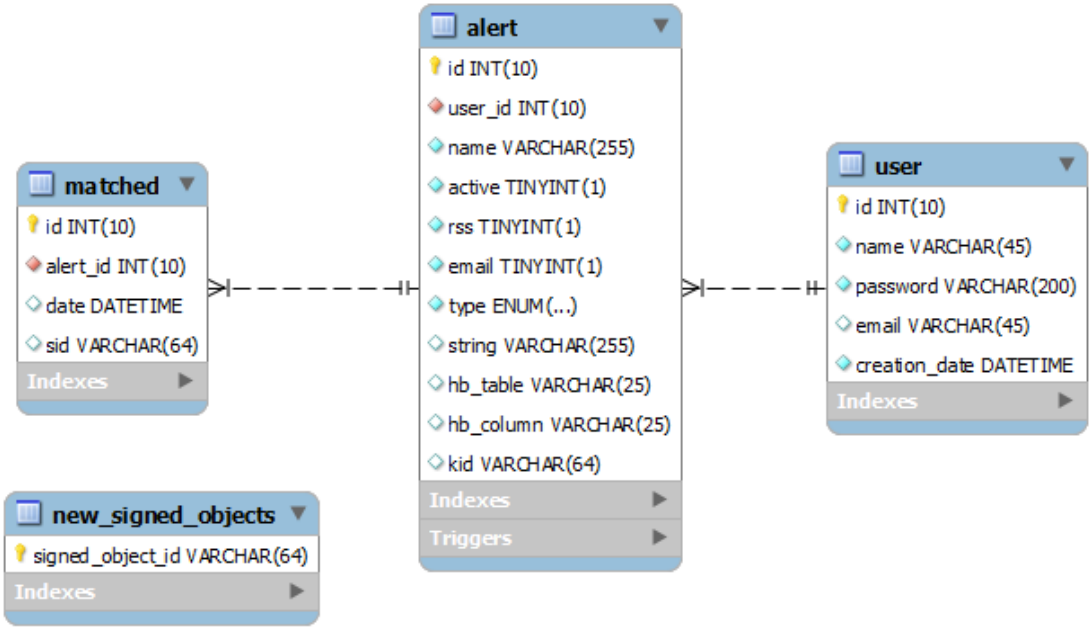

Figure 3.11: Alerts in the Database

3.3.2 Big Data

Sometimes the sheer complexity and size of the stored data can cause problems. This is called the Big Data problem. The collection of signed objects faces this problem as there are millions of signed applications and certificates out on the Internet. As mentioned in Section 2.1, SSL Observatory scanned the IPv4 address space and collected more than 4.3 million valid certificate chains. Using Zmap, Project Sonar scanned IPv4 addresses too and collected 66 million certificates. And these do not include any applications and their contained certificates.

The solution to Big Data is a distributed system. A distributed system makes the execution of a certain task possible on multiple machines thus increasing speed, capacity and availability. Distributed systems have a better fault-tolerance than single machine systems because if one computer fails, others can still execute the given task.

Our project focuses on storage so what we need is a distributed data store. Such a system was designed to store large amount of replicated data across multiple computers. A distributed data storage usually ships with a distributed file system to manage the stored data. The architecture may follow the client-server scheme: there are one or more master servers which store information about the slaves (nodes). These masters store the hierarchic structure of the file system, metadata and the location of each file. In practice, at least two masters are deployed for redundancy.

Despite the obvious advantages of distributed systems, real-life implementations has serious problems. These problems were summarized by Eric Brewer in 2000. He stated that it is impossible for a distributed system to provide the following three guarantees:

- Consistency: there is no difference between read and written data
- Availability: the system always answers to a query
- Partition-tolerance: the system is able to tolerate the scenario in which it disintegrates

Figure 3.12 shows where distributed data storages stand with respect to the CAP-theorem.

The distributed data storage we use is Hadoop, a Google BigTable clone with read-optimization and consistency. [26] It is consistent, distributed, multidimensional, sorted, stores key-value pairs and columns may be omitted (there is no NULL value). As shown in Figure 3.12, it has a tabular structure. It is ideal for our use case as there are small-size updates to the database and fast look-ups are needed at scale.

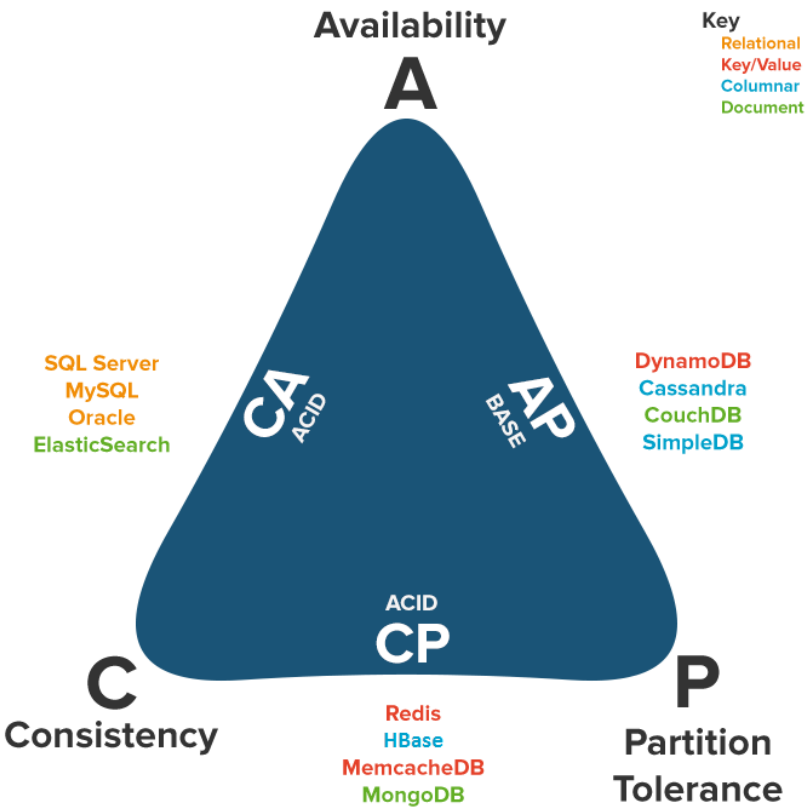

Figure 3.12: Distributed Data Storages and the CAP-theorem

HBase was designed on top of Hadoop, an Apache framework for distributed processing of large data sets. Hadoop has its own file system, the block-based Hadoop Distributed File System (HDFS). Each machine in the cluster has a single task to perform, an HDFS role. Most of the time there is only one Name Node in a cluster to store and manage metadata. This server knows where folders and files are located in the file system. The Secondary Name Node helps the Name Node to manage data. Data Nodes store data blocks. However, machines are single points of failure: if an error occurs, the data stored by that machine becomes unavailable. HBase has mandatory access to HDFS, that is why it is able to answer queries in milliseconds.

Just as machines have HDFS roles, they are assigned HBase roles as well. Such a role is a Region Server which manages a part of the key space in a sorted manner. The machine with the HBase role Master keeps track of which Region Server manages which keys. Because keys are sorted on a Region Server, searching by keys take logarithmic time and only one Region Server is needed to perform the task. On the other hand, if we wished to search by other columns, the whole data set on all Region Servers would need to be searched in linear time. Considering that the data set is Big Data, linear-time search can take massive amount of time. As a result, to work with HBase, the queries

must be defined before creating any tables and anything worthy of search should be made a key.

In ROSCO, HBase is the bulk of the repository. All the data about signed objects processed by the Preprocessor is stored here in several tables. In addition, we store message digests of signed objects calculated with SHA-256, SHA-1 and MD5.

3.4 Alert System

In this chapter we will discuss the Alert System, the module responsible for notifying users when objects of their interest arrives to the system. We will give the high-level overview of the system and discuss two alert types: simple alerts and signing key usage alerts.

Newly defined alerts apply to signed objects processed after the creation of the alert. This module performs its task separately from other modules and runs in an infinite loop. Figure 3.13 shows the high level overview.

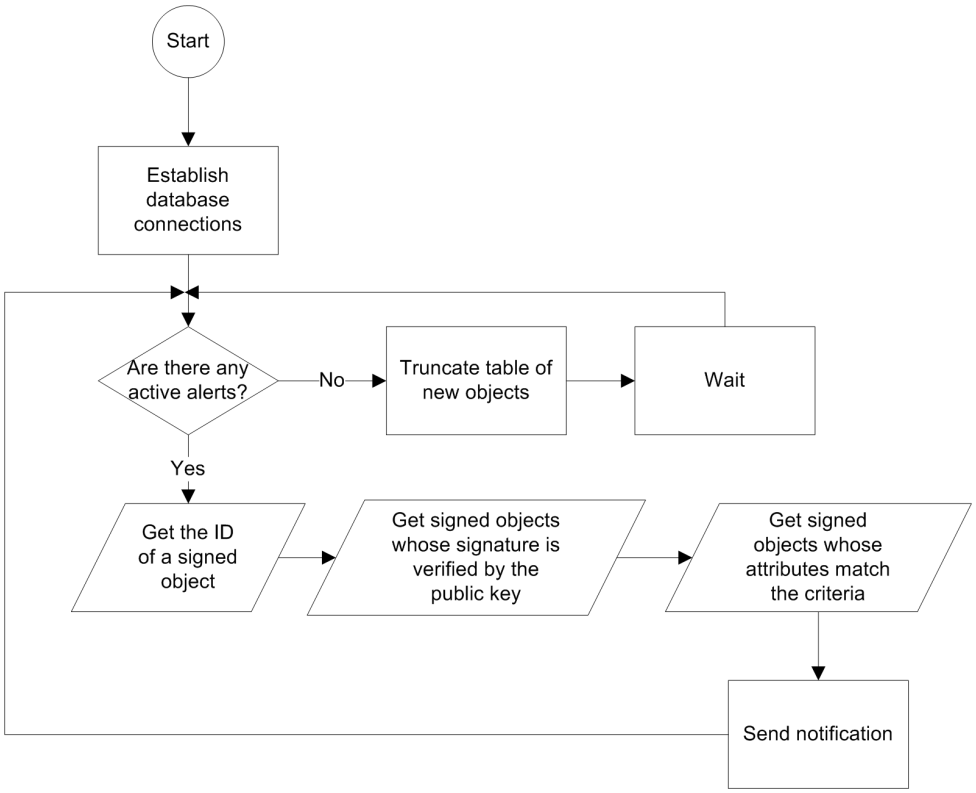

Figure 3.13: High-level Overview of Alert System

At the start of its execution, the Alert System needs to establish database connections to both HBase and MariaDB. The connection to MariaDB is used to extract alert specific metadata as discussed in Section 3.3.1. The System needs user information and defined criteria of each users from MariaDB. On the other hand, HBase is needed to access newly processed signed objects to match criteria.

In each loop, the System checks whether there are active alerts in the system. If there are none, the table storing newly processed objects can be truncated as no alerts need to be run for them. If there are any, the System begins checks alerts with respect to the newly processed signed object. At first, signing key usage alerts are applied: the System checks

HBase for any relationship between the supplied public key and other signed objects. Then come simple alerts: using the identifier of the signed object, the module retrieves its attributes from HBase and matches user-defined criteria.

At the end of a loop, the System sends notification to users whose alert triggered. Notification is done in two ways: the module sends an e-mail message to the supplied address or the user can subscribe to an RSS feed. Users may ask to receive notifications in both or neither ways. Even if users ask for no notification, they have a chance to see matched objects: the System keeps track of signed objects triggering the alert.

3.4.1 Simple Alerts

In this section we will discuss simple alerts. This type of alert enables user to define criteria for attributes of signed objects. If the system encounters a signed object whose attribute matches the defined criteria, it sends a notification.

Simple alerts can be defined for searchable attributes of all signed objects handled by the repository. This includes certificates, Portable Executables, Java Archives, Android Packages and public keys. There is a limitation, however, that the alert can apply to only a single attribute of the signed object. The defined criteria is a substring: if the provided substring is present in the attribute, the alert triggers and the System notifies the defining user. The chosen attribute is stored in a column of an HBase table in the repository, so the System saves the column and table names. This information is later used during checks and the alert triggers if the attribute contains the specified criteria. In implementation, the match is performed with regular expressions. Unfortunately, this implementation may be exploited to make the system dump every newly processed data to the user. The substring '*' as a regular expression matches everything - all signed objects of the given type would match and would become available to the user. As a counter-measure, we decided that only alphanumerical characters, '.' and '-' are supported and they are escaped on the server-side.

Simple alerts are useful for users who wish to acquire information about certain companies or organizations and their signed products. This type of alerts can also be used to track signed code of a specific environment such as operating system.

3.4.2 Signing Key Usage Alerts

In this section we will discuss signing key usage alerts. These are the alerts that enable companies to gain advantages by using our system. They provide a way for companies and organizations to keep track of their signed code. This feature also reduces the time needed to realize a key compromise. If the company receives a notification stating the an object was signed with their private key but the company did not perform the signing operation, then their key pair is compromised and needs to be revoked.

The question arises: how to determine if a signature was created with a private key? As the name suggests, these keys are kept from the public and should be known by the owner only. The answer is verification: if the public key is known, verification proves that the corresponding private key was used to generate the signature (see Section 3.2).

Users must make their public keys available for the repository to get notifications about their signed code. Since certificates play a huge part in the distribution of public keys, we request the certificate containing the public key in question. The certificate is processed as any other certificate (see Section 3.2.1) and the identifier of the key in the

repository is stored in MariaDB for further use.

While processing new signed objects, the provided certificate is used in verification and if the process yields a successful result, the binding between the object and the key is stored in HBase. Next, the Alert System looks for connections between the currently checked signed object and public keys provided by users. If a connection exists in HBase, interested users are notified via the channel they specified.

3.5 User Interface

The User Interface enables users to interact with the repository: uploading files, searching and alert definition is handled by this module. Interaction with the system is done via the browser and the User Interface is essentially a webserver with additional modules to increase flexibility.

In this section, we will discuss how the module is embedded to the repository and how users can interact with it. We will present the high-level overview of the User Interface and its building components. At first, we will cover uploads to the repository in Section 3.5.1. Section 3.5.2 discusses data queries and information gathering from the database. At last, users can manage their alerts as discussed in Section 3.5.3.

Figure 3.14 shows the high level overview of the User Interface. As mentioned before, users can interact with the database through their browser. The browser send HTTP requests to the repository and the webserver, Tornado, answers with HTTP replies in a synchronous or asynchronous manner depending on the request. If the request concerns metadata i.e. login data or alerts, the webserver uses the MariaDBManager written by us to interact with MariaDB. Should the user however request data stored in HBase, Torando uses the HBaseManager also written by us.

MariaDBManager is responsible for managing a database connection to MariaDB and issue SELECT, INSERT or UPDATE statements to the database. The need for HBaseManager is much larger as there is no operation in key-value databases equal to SQL JOIN. As a result, intersection of multiple query results must be done outside of the database. It is mentioned that requests to the webserver may be synchronous and asynchronous. Simple requests such as defining a new alert, which can be processed with a single MySQL statements are synchronous. Queries to HBase, however, are asynchronous because intersection may take a lot of time. For example, a user wishes to see all certificates whose Issuer's Common Name begins with 'a' and was issued a month ago. As there is no join, two separate queries are run in HBase: search for all certificates where the Issuer's Common Name begins with 'a' and return all certificates which was issued a month ago. HBase returns two sets and HBaseManager must perform the intersection. But it must not be forgotten, that storing digital certificates is a big data problem: either set may contain thousands of certificates. Even if the algorithm runs in time O(n), the algorithm must process so much data, that linear time may take minutes which is unfortunate from the user's point of view. As a result, queries have a timeout: if the repository is unable to reply in the given window of time, the query must be repeated.

The separation of webserver and database management is a question of implementation. They could have been integrated into one big webserver interacting with both databases but we would have lost the flexibility this design provides. This solution has the advantage of easily development: new functions not used by Tornado can be implemented without changing the webserver.

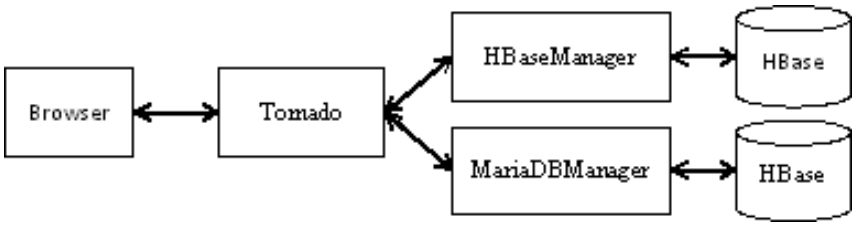

Figure 3.14: High-level Overview of the User Interface

3.5.1 Upload

Users can upload files through the User Interface. These files are placed into queues from which the Preprocessor takes and processes them into the database. As seen in Section 3.2, this queue is essentially a folder in the file system of the repository. Clients may upload digital certificates, Portable Executables, Java Archives and Android Packages.

Uploading is done by multipath synchronous requests to the webserver. Essentially, the file is not sent as one HTTP POST request as the file size may be too big to be handled efficiently. Instead, the file is split into smaller parts which travel through the network independently increasing the speed of uploading. On server side, the parts are collected, the original file is put together and it is placed into the queue of the Preprocessor.

3.5.2 Search

Searching is one of the main usage of the User Interface. This feature allows users to acquire information about signed objects by specifying some criteria their attributes must meet. Single and multiple attributes are allowed as well but not all attributes can be searched by.

Basically, all signed object type stored in the repository can be searched for. However, not all attributes can be used while searching. As mentioned before in Section 3.3.2, everything worthy of search should be made key to a table. It is also worthy of note, that rows in an HBase table may have unlimited amount of columns, it can only have a single row key. The row key is similar to a primary key in SQL. Often a single column is not enough to became the row key, so composite keys are permitted. So why not make all columns part of the row key? Because the most efficient queries specify the row key exactly. Should the user only provide the information that the signed object in question is a DLL and the type of the PE is not the first column, it would not limit the space in which the result is located. All Portable Executables would be needed to read in the database.

To overcome this problem, information about signed objects are stored in different tables. There is a master table for which the SHA-256 digest value is key and contains all information about the different signed objects. To make queries more efficient, so called 'inverse' tables are present, which connect particular information to the SHA-256 message digest. With this solution the search for DLLs would result result in one efficient query to get SHA-256 hash values and several other also efficient queries to access information specified by the digest value.

As mentioned in Section 3.2, relations between signed objects can be represented by a graph, an extended version of the Tree of Trust. If the client wishes it, the User Interface is able to show a small portion of the graph to see where the object stands and to whom is it related. Figure 3.15 shows where the certficate issued for thawte Primary Root CA

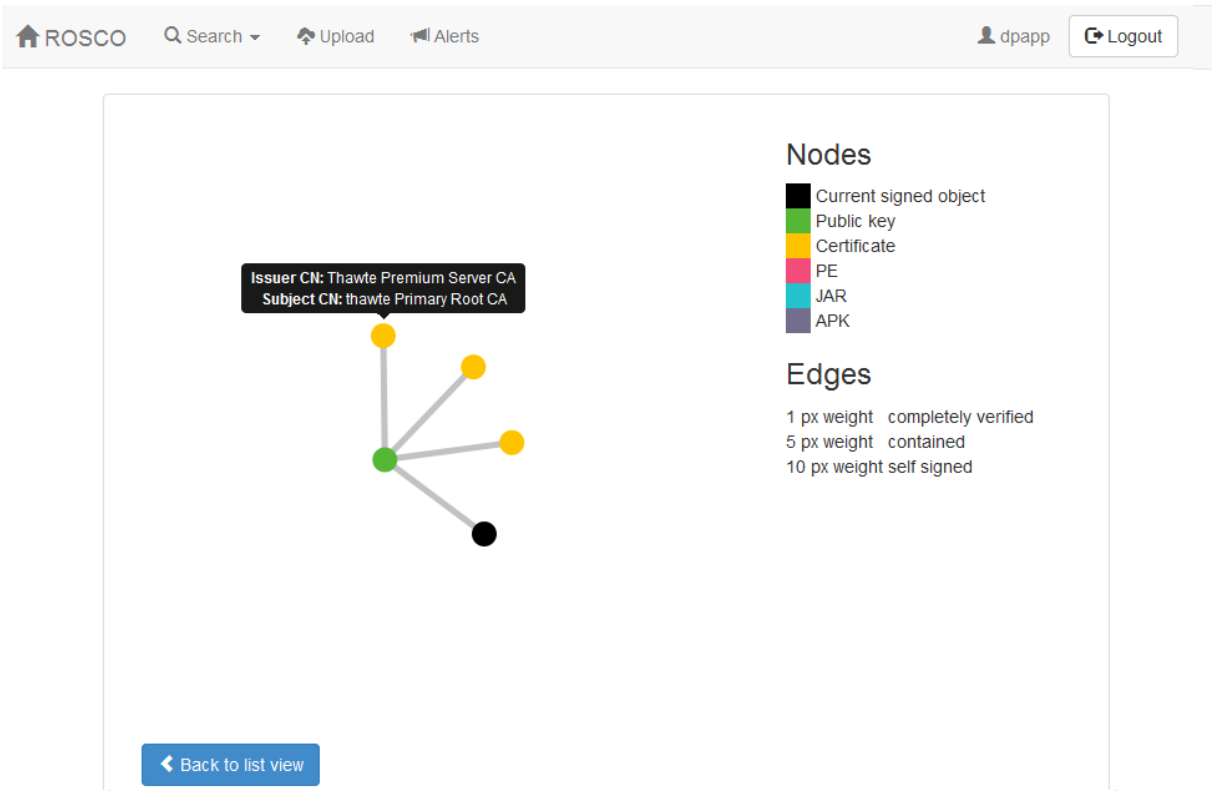

Figure 3.15: The thawte Primary Root CA in the Tree of Trust

stands in the Tree. The same public key is contained in three other certificates and all of them are self-signed.

Signed objects can be searched for in a general way: if users know a message digest value, they can specify it as keyword to search in all stored signed object, regardless of their type. The hash value may be calculated by SHA-1, SHA-256 and MD5 message digest algorithms.

Digital certificates have the most number of searchable attributes. Users may specify the serial number and extension type plus the value of the type. Validity is also available: both start and end dates can be used as the exact date or starting and/or ending points of a validity range. From the issuer and subject name the most common fields are searchable: common name, state, organization and country. Figure 3.16 shows an example of search for certificates.

For Portable Executables, three attributes can be searched by. The minimum operating system requirement can be chosen from a list with both the internal version code used by the linker and the commonly known operating system name presented. For example, Windows 8 is known as 6.2 internally. Search by compilation date works the same way as validity for certificates: both exact date and range can be specified. The type of Portable Executables is also a list with the values EXE, DLL, DRIVER or UNKNOWN. Figure 3.17 shows an example of search for Portable Executables.

Java Archives and Android Packages can be searched together both by inner member digest, and vendor. APK files have another searchable attribute: permissions the application needs. Figure 3.17 shows an example of search for Android Packages.

Experienced users can also use the free text search feature. They can choose not to click through the available search fields but instead type a search string and get results that way. For example, to search by the ValidFrom field, the validfrom free text field

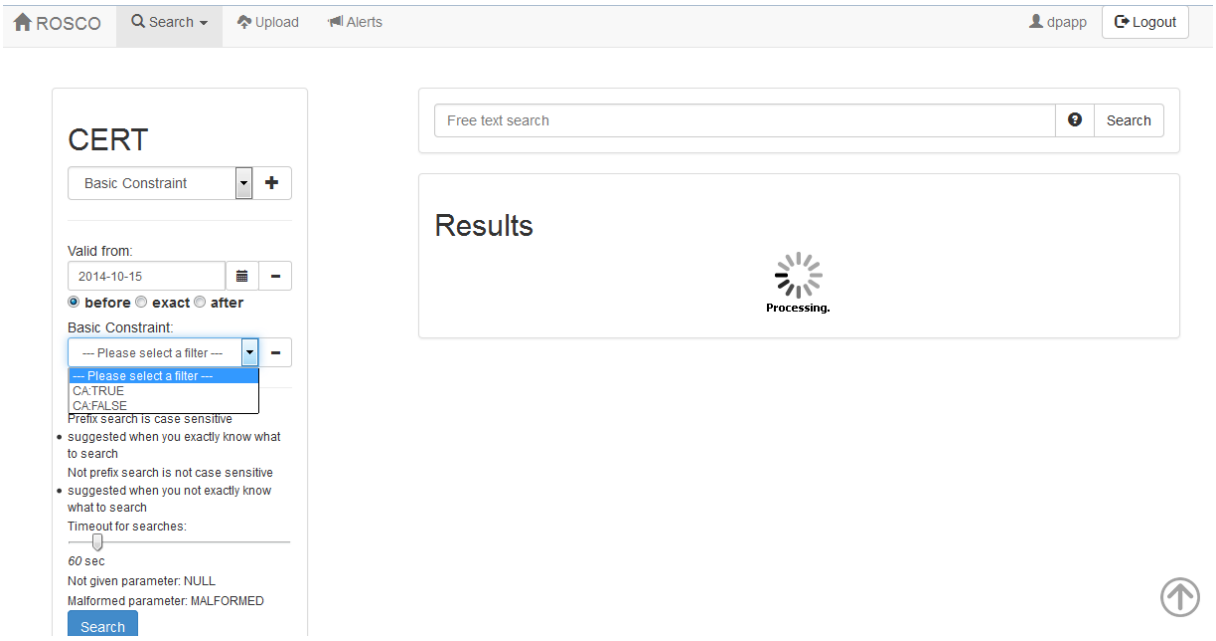

Figure 3.16: Example of Search for Certificates

must conform to the rule YYYY-MM-DD[#before, #after].

3.5.3 Alerts

As mentioned in Section 3.4, there are two kinds of alerts: string alerts and signing key usage alerts. Users may define and manage their alerts via the User Interface. This consists of de- or reactivating alerts, seeing which objects matched and accessing the generated RSS file.

When defining a string alert, the alert is active by default but users may override this setting via the checkbox. Alerts can be named and the system uses this name to identify alerts for users. The type of the alert specifies what kind of signed object the alert should match while the attribute of the signed object can be chosen from a combo box. The criteria is supplied in the keyword field. There are two more check boxes: notify stands for notification by e-mail, RSS is for generating RSS feeds. If e-mail notification is enabled, users can provide an e-mail address different from the default. The last column is Matched: previously matched objects can be viewed in a list any time. Also, previously matched objects are given as links, clicking on them results in a query for the matched object.

Signing key usage alerts can be used to keep track of what a certain private key signs. This can achieved by verifying the digital signature with the corresponding public key. Defining signing key usage alerts are done through the same interface. The Active, Name, Email, Notify, RSS and Matched fields are the same. However, instead of defining attributes and criteria, users need to upload a certificate with a public key inside through the File column. The public key is extracted from the certificate to create the alert and the certificate is placed into the queue of Preprocessor and its information will be stored and is made publicly available in the repository.

Notification of users can be done in two ways: e-mail and RSS feed.

RSS is a Web content syndication format and must conform to the XML 1.0 specification. An RSS feed is essentially an XML file published to the Internet. Users can

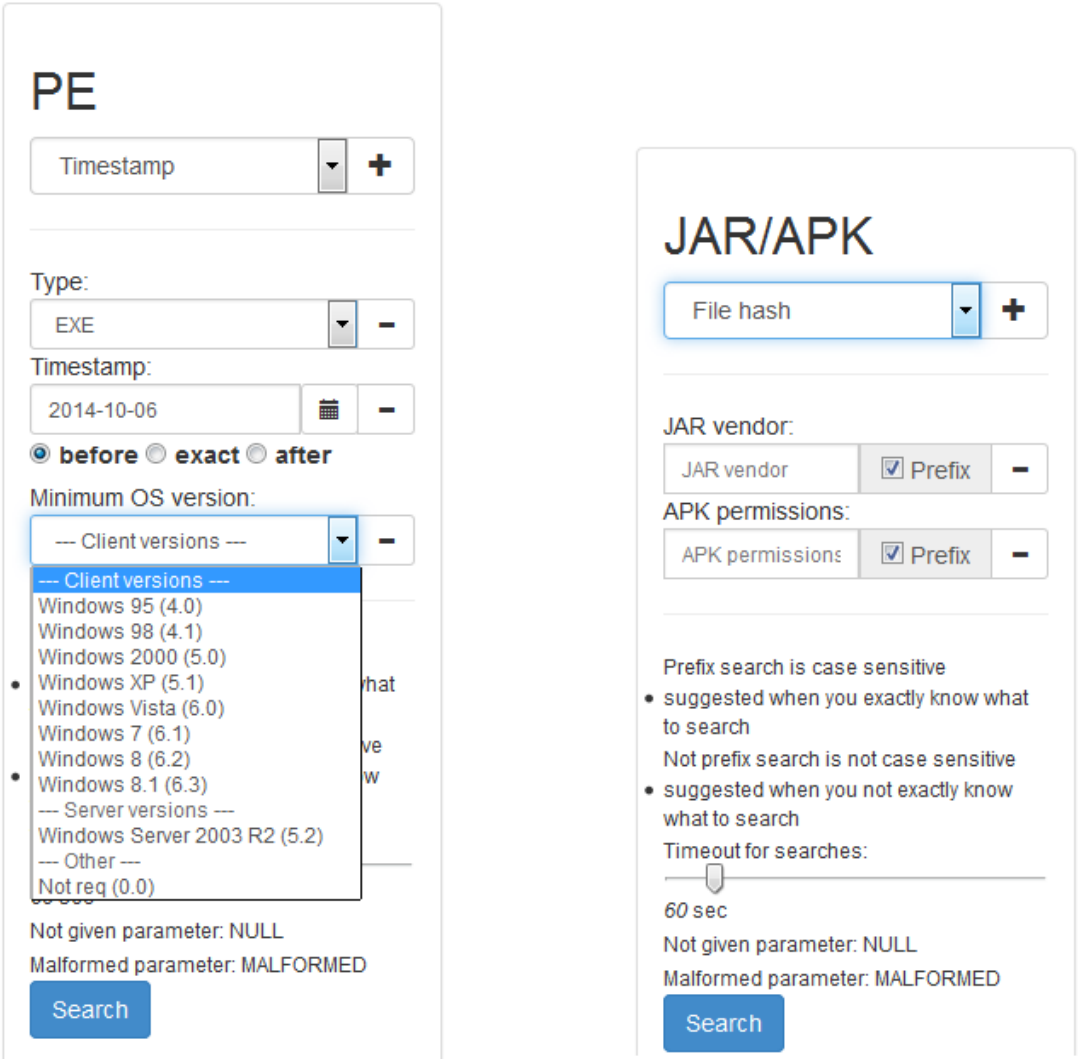

Figure 3.17: Search for Portable Executables and Java Archives

subscribe to such a feed and their readers periodically contact the file. The reader uses the lastUpdated field to see if new content has been added: if such a case occurs, the lastUpdated field is updated in the file. Our case can be considered special as the RSS feeds the system generates are meant to be seen only by the user for whom it was generated. This is called a private RSS feed, but there is a serious problem: authentication. If the feed was meant to be seen by only a group of users, each request for the file has to be authenticated. This is generally done by redirection: before access is granted to the file, users must log in. This approach is sufficient if the reader is a browser but there are other reader devices. Unfortunately, most RSS readers can not handle redirection and can not access private RSS feeds.

But there are other ways to authenticate user, our solution uses tokens. When a user registers to the site, a unique token is generated from the user-provided data using a symmetric encryption algorithm. This is a secure solution as the symmetric key is known only to the server so third party may not impersonate a legitimate user. This token is stored in MariaDB and the private RSS file of the user is named <token>.xml. If a reader

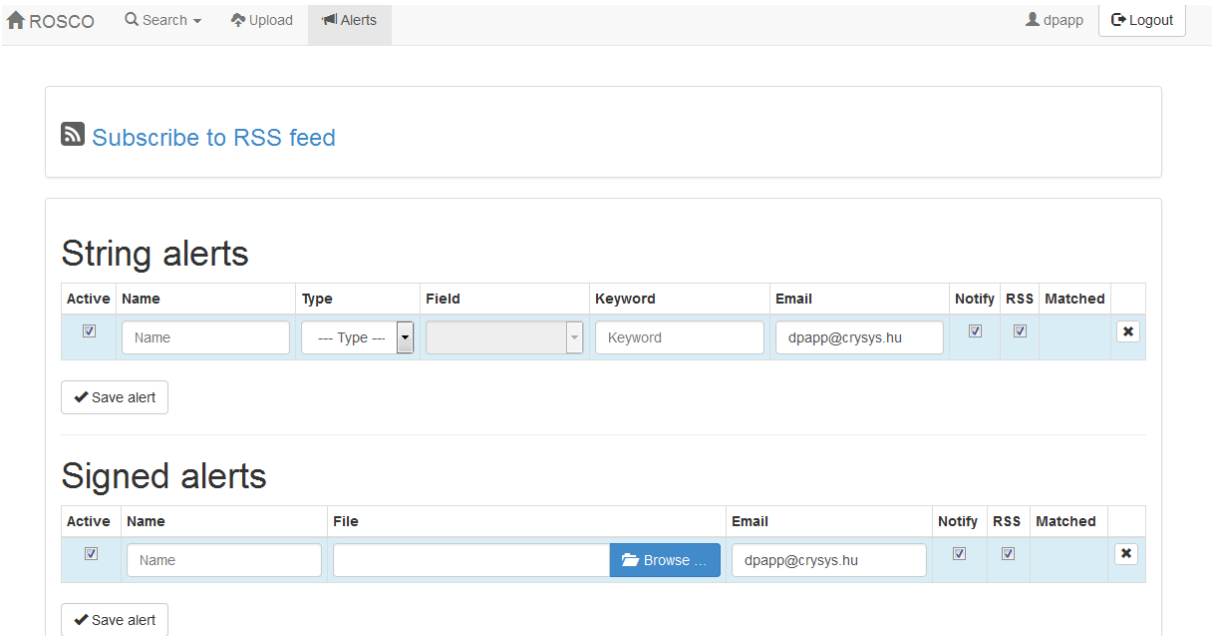

Figure 3.18: Defining Alerts

requests the private file, the system knows the reader has access to the file because the user must have been signed in when the reader subscribed to the feed.

Should users request notification by e-mail, the system includes information about the signed object in the message. The message format is fairly simple and concentrates on providing all information needed to determine if the object is of interest.

4. Case Study

In this chapter we show how our ROSCO system helps increasing trust in digital signature. Firstly, we will discuss sime general scenarios. Then we will present the cases of Duqu and Flame to see a real scenario where the repository could have helped users avoid becoming victims of malicious code. In both cases, the Alert System plays a significant role as it signals misuse of certificates to trusted companies.

There is a variety of ways our repository increases the amount of trust users place in digital signatures. Alerts are most useful for software makers as they help to detect the illegitimate usage of their signing keys. Average users also benefit from its use: metadata provided by the system can help them to evaluate trustworthiness of signed software.

In our first scenario, a user downloads a signed file from a freeware sharing site. However, the user does not trust the signer and would like to know whether there is a possibility that the file is malicious. In this case, they can contact our ROSCO system and search for the file by, for example, its SHA-256 digest value. If the file is not known to the system, the user can upload it. If it is known or has been processed after uploading, the system will be able to return information about the file when it is a search result. The user can view the relevant part of the extended Tree of Trust and see what other files are connected to the downloaded one. Viewing the information about the other files can help evaluate the trustworthiness of the downloaded one. Should a known malware be present among the related files, the user would not run the downloaded file as there is a possibility that it is a piece of malware.

To demonstrate the events of the first scenario, let us assume that the user downloaded the APK com.harvesters.linkupwow. The trustworthiness of the file is measured by VirusTotal: a free service that analyzes suspicious files and URLs and facilitates the quick detection of viruses, worms, trojans, and all kinds of malware. The system runs 45-55 antivirus software on each user-uploaded file simultaneously. The result is visualized by a score: number of detections / number of antivirus software run. For com.harvesters.linkupwow, this score is 1/47 which is low and the only detection can be thought of as a false positive. However, the user is still not convinced. They contact our ROSCO system and see on the graph that eight other files have been signed by the signer. The public key is contained in a self-signed digital signature which is valid until Jan 17, 2066 and was created by ivan from Beijing, China. Using our repository, the user can query VirusTotal for the trustworthiness of the other files. Seeing, that six out of the eight related APKs have been detected as malware by more than nineteen antivirus software (see Table 4.2), all including Trojan and adware detections, the user can conclude that the downloaded file may be a malicious signed software, but the malicious piece of code has not been added to other antivirus databases yet.

Possibly Malicious APKs		
SHA-256 message digest	Package name	VirusTotal score
20B850BDD1CA1CBD EF40D5B73B5AB6D2 1E5DAF7A77B464A8 9397ACFCBCC49042	com.androidemu.harvemm1	23/55
75B7D2F66112949C 4AE18AFB26A92086 327790EB8DE97A2A 3203F6FE2004157F	com.androidemu.harvespmxd	23/51
F338C44C5D78C204 72E85327F2B2EC50 9EC85B27AA2CD7F7 1679E52F53A5E1A3	com.androidemu.harvedragon3	23/54
8A70B377ADB8CEC0 C8F8F4DF53950D6E F8E9F42863070705 59B2CCA22BE8A56B	com.harvesters.linkupwow	22/50
51A97AD597E4B484 43BDAFA97BE6244F 0FF48E4512CA6F4D 8EC5F66A20AE146A	com.harvesters.linkupwow	21/54
D6522D0727A41DE3 2D7025565F29A87D 3830BA250D51CDA7 2360AD4C8FD65419	com.androidemu.harvedragon3	20/54
1001E583F33C2473 217BF486795C2759 709EB24AA12D8907 5E19BFD7666900A2	com.androidemu.harvemm1	17/48
C14C3D9E46B54454 086A7482C6E04690 3B45C79D93DE77F8 3959401AF47E2104	com.harvesters.linkupwow	7/52
0523578757C9DDC7 9F61CB8BE9EA4AFD AFCF70C0AED7970B 18ABBBCF08B651B2	com.harvesters.linkupwow	1/47

Table 4.1: Possibly Malicious APKs and Their VirusTotal Scores

In our second scenario, a user encounters a signed piece of code which was created five years ago. The user contacts the repository, requests the data of the file in question and from the provided metadata, learns that nobody has seen the file in five years. Since signed pieces of code are developed to be used by many users, an unseen file for five years is more than suspicious and may signal a targeted attack against the user. The metadata presented by the system helps users to determine how much trust they can place in the code.

4.1 Duqu

Duqu was discovered by the Laboratory of Cryptography and System Security (CrySyS) [3] during an incident response investigation. The primary goal of this piece of malware is information gathering. One of its files is digitally signed driver and verification of the signature states that it was signed by C-Media Electronics Incorporation. This piece of malware shares many things in common with Stuxnet: modular design, kernel driver based rootkit, DLL injection, etc.

Duqu was first detected on September 1, 2011 when one of its file were uploaded to Virustotal for scanning. Unfortunately, only two antivirus engines were able to detect it and it was added to other antivirus databases only later. On September 9, a Duqu driver was uploaded with the valid digital signature of C-Media. Since then, several new variants have been found including the dropper: the Laboratory of Cryptography and System Security discovered a Microsoft Word document which exploited a zero day kernel vulnerability.

The certificate of C-Media was issued by VeriSign Inc., a well known and widely trusted Certification Authority. As a result, infected computers accepted and trusted Duqu. The compromised key was revoked one and a half month later on October 14, but the detection of key compromise would have been faster with the help of our repository.

If C-Media had been a client, they would have had a signing key usage alert for their own private key. When parts of Duqu were uploaded to the repository, the system would have known the signature belonged to C-Media. Because of the signing key usage alert, the company would have received a notification either by e-mail or RSS-feed about the file. In the wake of this notification, the company would have contacted the system and would have checked the Portable Executable in question. Knowing that they had not in fact performed the signing operation for the file, their logical conclusion would have been that the private key is compromised and would have requested the revocation of their certificate. Considering that the signed piece of code was first contacted by VirusTotal on September 1 and the revocation took place on October 14, even if the actual check by the company would have happened several days after the notification, revocation could have happened at least a month sooner.

When the certificate revocation list appeared for the research community, our system could have been used to find other pieces of signed code, signed by the same private key. Should there be other pieces of malware created with the help of the compromised private key, researchers would have found them earlier.

4.2 Flame

The Laboratory of Cryptography and System Security participated in an international collaboration to investigate a previously unknown piece of malware: Flame also known as Flamer and sKyWIper. [25] The goal of Flame is cyber espionage and information stealing and had been operational since March 2010.

Flame provides a fake Microsoft certificate for verification, signed by the Microsoft LSRA PA Certification Authority. The creators were able to find an MD5 collision and use it to their advantage: the resulting certificate could be trusted by all major browsers and could fool the Windows Update system. When a machine tried to connect to the updating system, Flame redirected the connection through an infected machine and sent a fake, malicious Windows Update to the client. The fake update proceeded to download

the main body and infect the computer.

After the incident, Microsoft LSRA PA was replaced and the message digest algorithm was changed to SHA-1.

Our ROSCO system could have alerted Microsoft to the existence of the fake certificate. If they had had a simple alert for their certificates, the system would have notified them of the fake one as well. Suppose, the company had defined a simple alert with the keyword 'Microsoft LSRA PA' for the Common Name of the Issuer. Each time a signed object would have arrived to the repository, Microsoft would have been notified of the code signing certificate. In case of Flame, and any other pieces of code, Microsoft could have checked whether the company had been the true issuer. Founding no evidence of signing the fake certificate, Microsoft could have realized the problem with the Microsoft LSRA PA and could have revoked the CA certificate sooner. As the operating system would not have accepted the fake certificate, Flame could have been rendered nearly useless.

5. Conclusion and Future Work

Motivated by recent targeted malware, which used digitally signed components that appeared to originate from legitimate software makers, we developed a respository of signed code and some related services with the objective of augmenting the standard signature verification workflow with checking of reputation information on signers and signed objects and allowing for the detection of key compromise and fake certificates.

Our ROSCO system provides

- a data collection framework for signed software and code signing certificates,
- a data repository that can handle large amount of signed objects efficiently, and that supports a flexible query interface,
- reputation information for signed objects, such as when a given signed object has been first seen and how often it was looked up by users,
- alert services for private key owners that help them detecting when their signing keys were illegitimately used, and hence, probably compromised.

ROSCO does not aim at replacing the entire code signing infrastructure, rather, it tries to complement it with new mechanisms. There is no requirement whatsoever to change the operating principles of participants that do not want to use our system. This opt-in approach allows for the possibility of gradual deployment. The services offered by ROSCO will become more useful with the expected increase of the size of our repository. We hope that this will attract more participants to use our system who can benefit from our services when determining the trustworthiness of a signed application.

In this paper, we gave a detailed description of the design and implementation of ROSCO. We started by introducing its overall architecture, and then described its components such as the data collection and processing subsystems, the SQL based data used for storing meta-data and the noSQL database used for storing the actual signed objects and their relationships, the alert subsystem, and web based user interface. We also discussed how ROSCO could have been used to detect the misuse of signatures and certificates in the high profile targeted attacks of Stuxnet, Duqu, and Flame, and how it can be used to identify malicious applications by revealing the bad reputation of their originators.

The development of ROSCO is still on-going and there are many possibilities for future work. We plan to extend the set of supported signed objects with certificate revocation lists and timestamps, and the set of supported file types with signed MS office documents. We also plan to give access to our system to a selected set of signing and relying parties for testing purposes, and to open it to the general public later. Finally, from a scientific point of view, the huge amount of signed objects that we collected is a very valuable resource, on which we intend to perform different analysis tasks with the aim of better understanding code signing practices.

Acknowledgement

The work of the authors was partially supported by IT-SEC Expert, which received a NICOP Research Grant from the Office of Naval Research Global (ONRG) under award number N62909-13-1-N243.

Bibliography

[1] B. Kaliski. Public-key Cryptography Standards #7: Cryptographic Message Syntax. `http://tools.ietf.org/html/rfc2315`, March 1998.

[2] Ben Laurie, Adam Langley, Emilia Kasper. Certificate Transparency. `http://datatracker.ietf.org/doc/rfc6962/`, June 2013.

[3] Boldizsár Bencsáth, Gábor Pék, Levente Buttyán, Márk Félegyházi. Duqu: A Stuxnet-like malware found in the wind. Technical report, Laboratory of Cryptography and System Security, October 2011.

[4] Boldizsár Bencsáth, Gábor Pék, Levente Buttyán, Márk Félegyházi. The Cousins of Stuxnet: Duqu, Flame, and Gauss. *Future Internet 2012*, pages 971–1003, 2012.

[5] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, W. Polk. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile . `https://datatracker.ietf.org/doc/rfc5280/`, May 2008.

[6] Adrian Perrig Dan Wendlandt, David G. Andersen. Perspectives: improving ssh-style host authentication with multi-path probing. In *ATC'08 USENIX 2008 Annual Technical Conference on Annual Technical Conference*, pages 321–334. Carnegie Mellon University, 2008.

[7] Don Johnson, Alfred Menezes. The Elliptic Curve Digital Signature Algorithm. Technical report, University of Waterloo, Canada, August 1999.

[8] Electronic Frontier Foundation. SSL Observatory. `https://www.eff.org/observatory`.

[9] Google Inc. Introduction to Android. `http://developer.android.com/guide/index.html`.

[10] Loren M. Kohnfelder. *Towards a Practical Public-key Cryptosystem*. PhD thesis, Massachusetts Institute of Technology, 1978.

[11] M. Cooper, Y. Dzambasow, P. Hesse, S. Joseph, R. Nicolas. Internet X.509 Public Key Infrastructure: Certification Path Building. `http://tools.ietf.org/html/rfc4158`, September 2005.

[12] MariaDB Foundation. Introduction to Android. `https://mariadb.org/`, 2014.

[13] Mark Schlosser, Brian Gamble, Jody Nickel, Claudio Guarnieri, HD Moore. Internet-Wide Scan Data Repository. `https://scans.io/study/sonar.ssl`, 2014.

[14] Microsoft Corporation. *Windows Authenticode Portable Executable Signature Format*, July 2008.

[15] Microsoft Corporation. *Microsoft Portable Executable and Common Object File Format Specification*, February 2013.

[16] Moxie Marlinspike. Convergence. `http://convergence.io/`, 2011.

[17] Natioanl Institute of Standards and Technology. Digital Signature Standard (DSS). `http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf`, July 2013.

[18] Oracle Inc. JAR File Specification. `http://docs.oracle.com/javase/7/docs/technotes/guides/jar/jar.html`.

[19] Peter Eckersley. Sovereign Keys: A Proposal to Make HTTPS and Email More Secure. `https://www.eff.org/deeplinks/2011/11/sovereign-keys-proposal-make-https-and-email-more-secure`, November 2011.

[20] PKWARE Inc. .ZIP File Format Specification. `http://www.pkware.com/documents/casestudies/APPNOTE.TXT`, September 2012.

[21] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Bernes-Lee. Hypertext Transfer Protocol – HTTP/1.1. `https://www.ietf.org/rfc/rfc2616.txt`, June 1999.

[22] R. Housley, W. Ford, W. Polk, D. Solo. Internet X.509 Public Key Infrastructure. `https://www.ietf.org/rfc/rfc2459`, January 1999.

[23] R. L. Rivest, A. Shamir and L. Adleman. A Method for Obtaining Signatures and Public-Key Cryptosystems. In *Communications of the ACM, Vol. 21 Issue 2*, pages 120–126, February 1978.

[24] Scrapyhub. Scrapy. `http://scrapy.org/`.

[25] sKyWIper Analysis Team. sKyWIper (a.k.a. Flame a.k.a. Flamer): A complex malware for targeted attacks. Technical report, Laboratory of Cryptography and System Security, May 2012.

[26] The Apache Software Foundation. Apache Hadoop. `https://hadoop.apache.org`, 2014.

[27] The International Computer Science Institute (ICSI) of University of California, Berkeley. ICSI Certificate Notary. `http://notary.icsi.berkeley.edu/`, April 2012.

[28] Martin E. Hellman Whitfield Diffie. New directions in cryptography. In *IEEE Transactions on Information Theory, Vol. IT-22, No.*, pages 644–654, November 1976.