Opleiding Informatica

Threat Intelligence Feed For

Mobile Applications

Dennis Buurman

Supervisors:
Olga Gadyatskaya & Yury Zhauniarovich

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

28/12/2021

Abstract

Malware is becoming an increasingly difficult problem with the increasing amount of malware variants and threat actors becoming more sophisticated. This problem extends to mobile platforms such as Android. Part of the solution is YARA, a tool to describe Indicators of Compromise (IOCs), which are widely used for malware detection. There are mobile security platforms that use YARA to scan Android applications for malware threats. However, with the existing implementations the IOCs from the YARA rules only exist in the rules themselves. Collecting and sharing the IOCs from YARA rules can help tackle the malware problem. This thesis aims to create a threat intelligence feed composed of IOCs found within YARA rules and metadata associated with the rules. The result is a collection of 9095 IOCs distributed over 78 different types.

Contents

1 Introduction

From 2016 to 2021 the amount of smartphone users has increased from 3.668 billion to 6.378 billion with a forecasted growth to 7.516 billion users in 2026[1]. Android, with around a 70% market share in mobile operating systems worldwide at the end of 2020, is positioned as the leading mobile operating system[2]. Due to the increasing amount of applications and application sources, Android users are more likely to be targeted by cyber attacks. Android allows downloads and installations from third-party sources, which makes the distribution of malicious software easy for attackers.

Android has a lot of security measures that can prevent malicious code from being installed or executed. This includes measures at the platform level as well as at the ecosystem level (e.g., Google Play Protect[3]). Unfortunately, mobile malware still has become a significant problem in recent years, especially Android malware, as it is the dominant and also open mobile platform.

Despite the constantly improving security, malware incidents still occur frequently. The latest Google Android security report[4] states that 0.04% of all downloads from Google Play were Potentially Harmful Applications (PHAs). PHAs refer to malware such as click fraud, trojans, SMS fraud, spyware, phishing, and hostile downloaders. Information about different malware types and their behavior can be found in [1, 2, 3].

Unlike its competitor IOS, Android allows the download and installation of software from third-party application markets. According to the report, the PHA rate for applications downloaded through these other sources was 0.92%. This is considerably higher than the 0.04% from Google Play downloads. Relatively speaking, the percentages do not seem significant. However, the report also mentions that 1.6 billion PHA installation attempts from outside Google Play were blocked, corresponding to 73% of total PHA installation attempts from outside sources. This means 27%, or roughly 600 million, PHA installation attempts were successful.

Present Android security measures prevent a lot of malicious software from being installed. Still, a large amount of malicious applications manages to slip through. Therefore, other measures are needed in order to help improve the security of the Android platform.

1.1 YARA and Threat Intelligence

One approach to improve malware detection is through malware analysis. With such an analysis, a strain of malware is broken down into a set of characteristics describing the behavior of the malware. These characteristics are called Indicators Of Compromise (IOCs). By collecting malware characteristics analysts can produce information known as threat intelligence. Threat intelligence can be combined into a data stream called a threat intelligence feed (TI feed). Nowadays, it is becoming more and more common for organizations to, often automatically, collect and consume

[1] https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/
[2] https://gs.statcounter.com/os-market-share/mobile/worldwide
[3] https://developers.google.com/android/play-protect
[4] https://source.android.com/security/reports/Google_Android_Security_2018_Report_Final.pdf

threat intelligence to fortify their defenses against cyber attacks. According to Mordor Intelligence[5], the threat intelligence market was valued at 5.28 billion USD in 2020 and is expected to reach 13.9 billion USD by 2026, with a compound annual growth rate of 12.9%. For this work, YARA rules are used to construct a TI feed. YARA is a multi-platform tool for file analysis created by VirusTotal[6]. Malware analysts use YARA notation to create textual or binary pattern descriptions of malware called a YARA rule. Creating these malware descriptions helps with identification and classification of malware and applying them can improve malware detection in an organization.

1.2 Koodous

Koodous[7], publicly accessible from May 2015, is a free online community platform for Android malware research. It offers tools for analysis on Android applications and social interaction between analysts. The platform uses YARA rules to identify malware within Android applications. Koodous keeps a vast and constantly updated APK repository that analysts can use. An APK is an Android Package, which is an archive file format used for distributing and installing Android applications. An applications APK contains all elements required to intall the application on an Android device. In Koodous, users can vote on the reputation of applications within the repository. This vote can either be a positive or negative. By following others and commenting on applications users can interact with each other. Users can use Koodous with the Android application[8] or the REST API[9]. The REST API allows for data requests from the platform. This data can be used for analysis and research.

Analysts have already collected IOCs and expressed them as YARA rules. However, not all organizations interested in mobile malware detection can integrate YARA in their environment. Moreover, many of the rules are repetitions or too basic to detect more complex malware. Therefore, it is illogical for organizations to consume YARA rules directly from Koodous and deploy them into their environment. The goal of this project is to use the publicly accessible YARA rulesets of the Koodous community to provide accessible and useful intelligence for a cyber threat intelligence feed for mobile Android applications.

1.3 Thesis overview

This section contains the introduction. The rest of this thesis is structured as follows. Section 2 covers background theory and related work. Section 3 describes the methods used in collecting and preprocessing the dataset. Section 4 outlines the intelligence sources and discusses how intelligence is extracted from them. Section 5 discusses the output. The projects limitations are discussed in Section 6. Finally, Section 7 concludes the thesis and discusses directions future work. This bachelor thesis is written as part of the computer science bachelor at LIACS (part of Leiden University). It is supervised by Olga Gadyatskaya & Yury Zhauniarovich.

[5]https://www.mordorintelligence.com/industry-reports/threat-intelligence-market#faqs
[6]https://www.virustotal.com/gui/home/upload
[7]https://koodous.com
[8]https://play.google.com/store/apps/details?id=com.koodous.android&hl=en&gl=US
[9]http://docs.koodous.com/

2 Background

This section discusses necessary theory and related work. Among others, Android security, threat intelligence, and YARA are discussed.

2.1 The Android platform security controls

Android is a multi-layered system, making it very complex. Because of this complexity, Android has multiple different security controls (e.g., memory isolation, biometric user authentication) that have their own models and do not have to be consistent with each other [4]. To generally describe Android security, René Mayrhofer et al. [4] present five rules expressing how Android's different security controls combine at the meta level:

1. *Multi-party consent.* Every executed action must be approved by all main parties. In the standard case these parties are the user, platform, and developer. This applies to both processes and objects (e.g., files, memory). Control over a data item is granted to the item's creator. For instance, user created files are controlled by the user.

2. *Open ecosystem access.* Android users are not limited to any single application source. Central developer vetting and user registration are not required.

3. *Security is a compatibility requirement.* The Android Compatibility Definition Document[10], or CDD, is a set of requirements that devices must met in order to be compatible for the latest Android version. This CDD is enforced by the Android Compatibility Test Suite[11] (CTS). Devices failing CTS, meaning they do not meet the CDD requirements, are not considered Android devices.

4. *Factory reset restores the device to a safe state.* "In the event of security model bypass leading to a persistent compromise, a factory reset, which wipes/reformats the writable data partitions, returns a device to a state that depends only on integrity protected partitions."[4] In other words, only Verified Boot system code is kept after a factory reset.

5. *Applications are security principals.* Android applications are not fully authorized agents for user actions. This means the applications cannot access all user-data even if executed by the user.

The above rules combined can be called a multi-party consent model. Actions only happen if all involved parties give their consent. This model has a significant benefit to the user. Because user privileges are separated, applications cannot access data from other applications, which helps protect the user's private data. The full implementation and more about Android security can be read in [4, 5, 6].

[10] https://source.android.com/compatibility/cdd
[11] https://source.android.com/compatibility/cts

2.2 TI and TI Feeds

Threat Intelligence (TI) is defined as evidence-based knowledge about threats to an environment. This includes context, mechanisms, indicators, implications, and actionable advice [7]. In the cyber security field this translates to cyber threat intelligence (CTI), which is aimed at detecting and preventing potential cyber attacks. TI is seen as a way to fortify the defense of an environment and prepare for known and unknown threats [8].

Indicators of Compromise (IOCs) are an important and widely used part of CTI. They consist of data elements found in malicious software. IP addresses, domain names, and file hashes are examples of IOCs. According to [9], IOCs are important because of three main reasons. Firstly, particular threats can be documented in a consistent manner. In addition, consistent documentation simplifies the sharing of information. Secondly, IOCs provide security personnel with a set of data that can be fed to automated processes. Thirdly, IOCs can provide context to threats by looking at their past occurrences. Commonly, three categories of IOCs are used: Network, Host-based, and Email indicators [10].

- Network indicators include IP addresses, URLs, and Domain names. These indicators could come from previously performed attacks like a Distributed Denial of Service attack (DDoS). A downside to this type of IOC is the usual short term effectiveness as threat actors have the ability to change it regularly.
- Host-based indicators include names of malware, file hashes, Dynamic Link Libraries (DLLs), and registry keys. They can be discovered by analyzing infected computers.
- Email indicators are found in socially engineered emails targeting organizations and individuals. These emails are crafted to resemble an email coming from a legitimate source. Oftentimes, they contain links to malicious sites or come attached with malicious software.

IOCs themself are not intelligence, but they are aspects that enable the production of it [10]. Accompanied by an organization's internal intelligence, IOCs can be used to provide intelligence. In [10], CTI is divided into four domains: strategic, operational, tactical, and technical threat intelligence.

- Strategic threat intelligence is high-level information aimed at understanding risk and impact of potential threats. It includes financial impacts, trends, historical data, and predictions regarding threats. This information is directed towards decision-makers, helping them allocate their effort and budget in order to mitigate these potential threats.
- Operational threat intelligence is information about imminent threats against the organization. This form of intelligence is rare, as most private entities do not have the right to legally access the right communication channels.
- Tactical threat intelligence, also referred to as Tactics, Techniques, and Procedures (TTPs), is information describing the modus operandi of specific threat actors. This kind of intelligence is used to understand attacks and adapt the defense accordingly.

- Technical threat intelligence is information fed to investigative or monitoring functions like firewalls or mail filtering systems. These functions can, for instance, use IOCs to block suspicious incoming traffic.

A TI feed is a way to automatically distribute threat intelligence information. Commonly, TI feeds consist of technical TI only, as this is the easiest type of information to share. Receivers of the threat information can use it for threat detection. An example of this is scanning a downloaded file using threat information provided by a TI feed. A TI feed can have multiple sources and its data needs to be analysed in order for it to be used optimally. This leads to many possibilities for research. TI feeds can be public, open-source, or private. Private feeds can be organization exclusive or locked behind a paywall. Public feeds and open-source feeds are free for usage, but open-source feeds allow contributions of third-parties.

2.3 Pyramid of Pain

Not all intelligence is of equal value. David Bianco's pyramid of pain [11] describes the "pain" or difficulty it causes if you deny a threat actor a type of indicator. The pyramid of pain is shown in Figure [1]. Each level, from bottom to top, increases the difficulty for a threat actor to adapt. As tools are harder to change than an IP address it is more "painful" for a threat actor to bypass a defense based on it. TTPs reside at the highest level, meaning a defense based on an attacker's TTPs is the toughest to bypass. An attack can be recognized at every level of the pyramid, but the higher up in the pyramid the more "painful" it becomes to bypass and the higher the value of the intelligence. Therefore, having intelligence is only the first step. Using it effectively to detect attacks on a higher level of "pain" comes after. Because of this, research and analysis on gathered threat intelligence can improve its value.

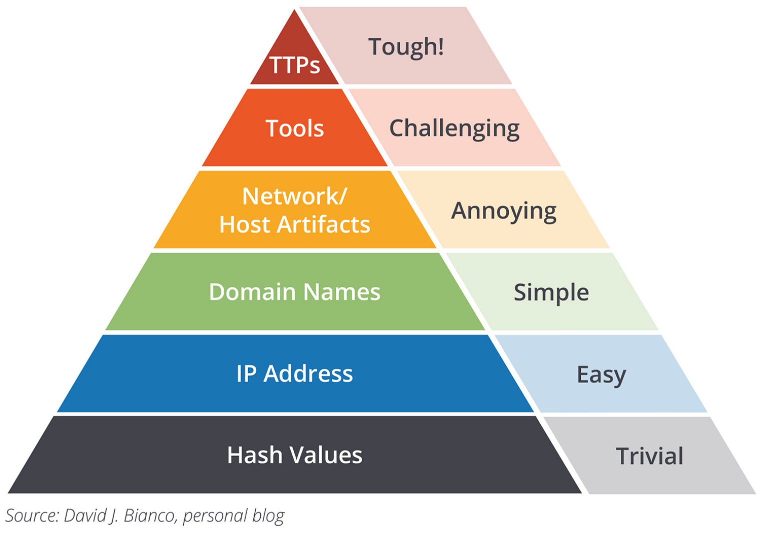


Figure 1: Pyramid of pain

2.4 Previous Work With TI Feeds

Multiple approaches to Android malware detection and analysis already exist. Park et al. [12] introduced a mobile malware threat intelligence evaluation based on situational awareness. As stated by Park et al., “Situational awareness means recognizing environmental factors in the time and space where a situation occurs and responding to future threats” [12]. Using machine learning, their model performs malware detection aimed at recognizing malicious application behaviour, threat assessment to rank the malicious behaviour, and decision-making to optimize rating. Standard situational awareness is based on the perspective of analysts or decision makers. By combining situational awareness with machine learning, decision support will be more objective. Malware detection approaches like machine learning detection rely on previously occurred incidents, making them reactive approaches. Grisham et al [13] present a model that aims to proactively identify mobile malware and associated key authors from hacker forum assets. It uses neural networks and recurrent neural networks to identify mobile malware attachments. This is followed by a social network analysis to determine the key authors. This proactive approach allows for threat intelligence before an attack has happened, which in turn helps security personnel with defending their organizations infrastructure. DREBIN [14] is another project aimed to improve Android security. It can be described as a lightweight method for Android malware detection that can run directly on smartphones. DREBIN uses a combination of static analysis and machine learning to achieve up-to-date malware detection with little runtime overhead. Interested readers can find more information about Android malware detection approaches in [3, 15, 16, 17]

Although the above mentioned projects are relevant to this work, the nature of them is different as they cover malware detection, not producing threat intelligence. Producing threat intelligence feeds is not a new field of research, but to the best of my knowledge, the topic of producing threat intelligence feeds specifically for Android applications, or mobile applications in general, has not yet been investigated extensively in the literature. However, some of the data from general threat intelligence can be relevant for mobile applications. For instance, Tounsi et al [10] discuss network indicators such as IP addresses and URLs. Both are also relevant for mobile applications.

Many general indicators are relevant to Android security, but there are some indicators specific to the Android platform. According to Gregor Robinson and George R. S. Weir [18], the primary security risk associated with the Android operating system is the misuse of permissions. With certain permissions, applications are allowed to read user data or even install software. Some applications need a subset of permissions to function properly. For instance, navigation applications need access to the device location in order to function properly. However, threat actors can abuse this system by creating applications that perform malicious activity such as spying on the user or reading their data. Therefore, application permissions are an important IOC to Android threat intelligence.

Other works about Android malware detection, including above mentioned works, use features like permissions, system calls, network activity, file hashes, DLLs, and more. Assisting malware detection approaches with a TI feed reporting such features may help improve detection. This makes research into threat intelligence an important part in the battle against cyber attacks.

2.5 Malware Detection Techniques

Malware detection approaches use different techniques. The techniques can be divided in two broad categories: anomaly-based detection and signature-based detection [19].

Anomaly-based detection is based on comparing what is regarded as normal behavior to the behavior of an inspected program. Specification-based detection is a special type of anomaly-based detection. This type uses a set of rules considered to be valid behavior. If a program violates this set of rules it is considered to be anomalous, which may indicate malicious activity.

Instead of looking at desired behavior, signature-based detection looks for known malicious characteristics within a program in order to decide if the program is malicious. Examples of such characteristics are file hashes, certain strings, permissions, and behavioral profiles. In other words, the characteristics include IOCs and TTPs. Most antivirus software implementations use the signature-based method for malware detection. The effectiveness of this method strongly depends on the known malicious behavior characteristics. Therefore, sharing information about these characteristics or indicators is important as it enables better malware detection for organizations. Figure [2]

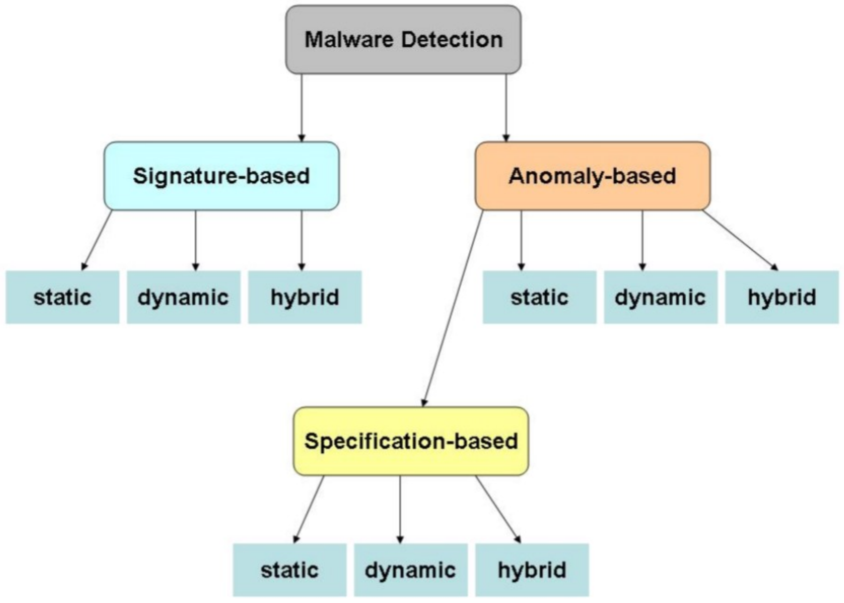
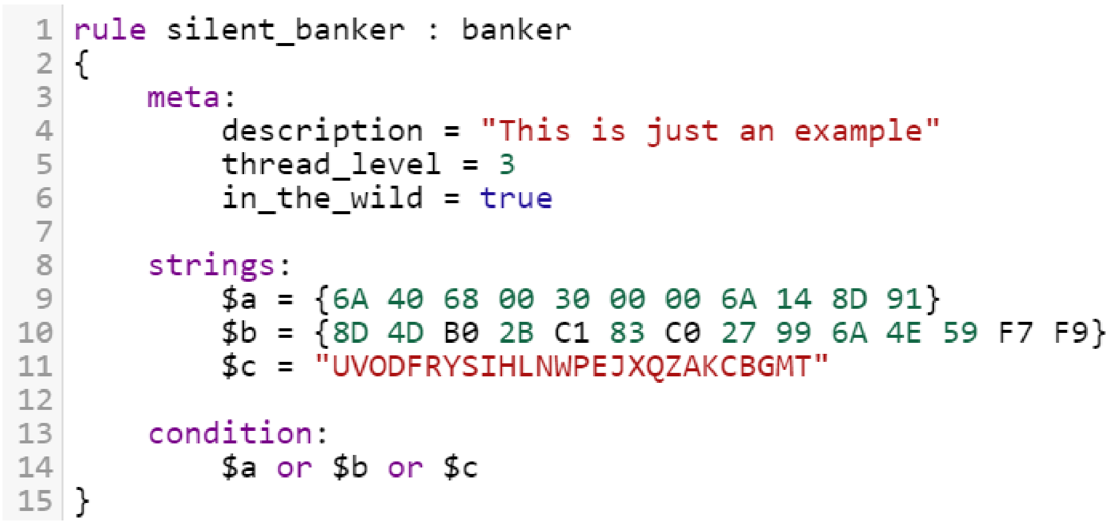


Figure 2: Malware detection techniques classification. Source: [19]

shows the relation between the detection techniques. As seen in the figure, every technique can have one of three analysis approaches: static, dynamic, or hybrid. The analysis approach refers to how data is acquired from the target program. Static analysis scans the syntax of the program files to gather data. The program is not executed. Dynamic analysis collects the behavior of the program at runtime. Hybrid approaches combine static and dynamic analysis. More information about the three analysis approaches can be found in [20].

2.6 YARA explanation

YARA was publicly released in 2013 by VirusTotal[12]. The tool is multi-platform, meaning it runs on Windows, Mac OS, and Linux. As previously mentioned in Section 1, a YARA rule is a malware pattern description. In YARA, a rule consists of a set of strings with a boolean expression containing the rule's logic. Strings are the variables containing the information describing the malware. They are essentially the IOCs describing the threat. A string value can be defined in three types: textual, hexadecimal, or a regular expression. However, some strings are only meant as comments. For these so-called meta strings there is a special section called meta. Strings declared in the meta cannot be referenced in the condition. The condition is a boolean expression that determines whether a file satisfies, or matches with, the rule. In the condition, previously defined strings are used to build the rule's logic. The order in which the sections should be is meta, strings, condition. This is a fixed order. However, only the condition is required. Rules without (meta) strings containing only a condition are thus valid. Figure [3] shows a simple rule showcasing the general structure of a YARA rule. In depth information on YARA's features and capabilities can be found in the YARA documentation[13].



```
rule silent_banker : banker
{
    meta:
        description = "This is just an example"
        thread_level = 3
        in_the_wild = true

    strings:
        $a = {6A 40 68 00 30 00 00 6A 14 8D 91}
        $b = {8D 4D B0 2B C1 83 C0 27 99 6A 4E 59 F7 F9}
        $c = "UVODFRYSIHLNWPEJXQZAKCBGMT"

    condition:
        $a or $b or $c
}
```

Figure 3: Example ruleset from Koodous. Source: https://koodous.com/rulesets/10.

YARA's main function is scanning files based on a description. This is a static analysis approach as the actual data of the file is scanned. In contrast to static analysis, dynamic analysis is performed at run-time. Instead of looking at the data itself, a dynamic analysis approach looks at the behaviour during execution. Combining static and dynamic analysis results in a hybrid approach allowing for a higher IOC coverage. Therefore, it is important to cover both approaches. Obviously, dynamic analysis is not supported with native YARA. However, YARA supports the use of modules to extend its functionality. With modules, developers can define data structures and functions that allow for more complex conditions in rules. This means that, among others, dynamic analysis tools can be integrated. Some modules, like Cuckoo, are officially distributed with YARA. Other modules need to be added by the user or platform. Koodous added three modules to it's YARA

[12]https://github.com/VirusTotal/yara

[13]https://yara.readthedocs.io/en/v4.1.2/index.html

implementation: Androguard, Droidbox, and File. Documentation on these modules can be found in the documentation of Koodous[14] and YARA[15]. Cuckoo and Droidbox are examples of modules that use dynamic analysis tools.

2.7 Related work about YARA

YARA rules provide a great way to identify malware. However, they suffer from performance penalties as the run time grows linearly with the number of input files. As a solution Brengel et al. [21] present YARIX, a methodology to scale YARA searches efficiently. Using an inverted n-gram index and transforming YARA rules into index lookups they drastically reduce scanning time. This makes YARA searches a more viable option to use in large scale organizations.

Another point of interest is the large amount of time and effort needed to manually produce high quality YARA rules. AutoYara [22] is a YARA automation tool aimed at reducing the workload of analysts. This tool can generate simple YARA rules describing specific malware families. The research shows a decrease of 44-86% on analysts time spent constructing YARA rules. As a result, the analysts have more time to spend on complex malware families too advanced for tools.

Conditions in YARA rules are boolean expressions. Executing a YARA rule results in a match or no match. Possible information gathered during execution is lost. Naik et al. [23] propose embedding fuzzy rules with YARA rules to optimize information output. The embedded fuzzy rules are designed to capture all information generated by YARA rules. The major benefit of this concept is that it can be applied to existing YARA rules without requiring AI or other enhanced techniques in the rule generation process.

YARA is becoming an increasingly interesting point of research. Papers mentioned above show promising results. Still, many research possibilities for YARA remain. This work proposes the use of YARA rule elements to provide threat intelligence. To the best of my knowledge, this is the first research project regarding the subject.

[14] `http://docs.koodous.com/`
[15] `https://yara.readthedocs.io/en/v4.1.2/index.html`

3 Dataset Collection and Preprocessing

Before any TI can be extracted, the dataset needs to be collected and preprocessed. This section covers the process of collecting and preprocessing the dataset along with the methods used. Figure [4] shows a quick overview of the project workflow.

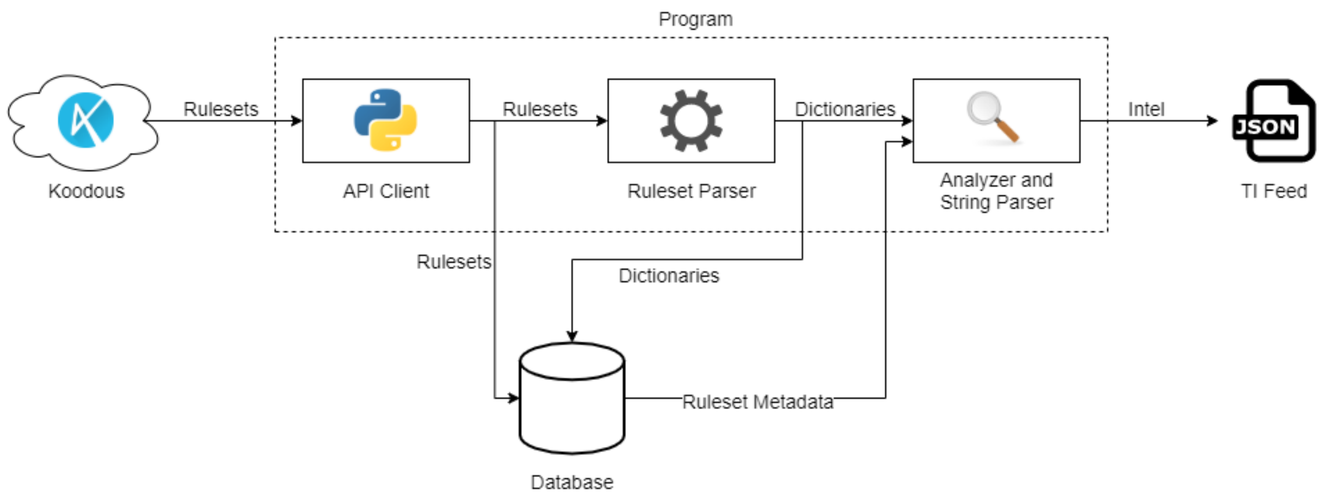


Figure 4: Project Workflow

3.1 Methods

This project uses several tools for gathering, processing, and analysing the public YARA rules from the Koodous platform. A Koodous REST API client is used to connect with the Koodous database and to query the rules. Program code is written in Python 3.8.6. Within Python, the requests[16] and certifi[17] modules are used to create a REST API client. Collected and processed data is stored in a NoSQL database from MongoDB[18]. MongoDB offers the pymongo module[19] to interact with their databases. The ply[20] module is used for the rule parsing process.

3.2 Data Collection and Storage

The data available on the Koodous platform can be divided into three categories: APKs, YARA rulesets, and analyst information. The amount of rulesets and APKs in the repository changes on a daily basis. At the time of writing there are 2208 public rulesets available and the APK repository contains a total of 83.7 million applications. APKs from the repository can be used to test YARA rules. When an application matches a ruleset it is added to that rulesets detection list. This way each rulesets receives a detection level based on the number of applications it detects. Rulesets can be mutated by their creator, meaning not only new rulesets could be added, but existing ones

[16]https://github.com/psf/requests
[17]https://github.com/certifi/python-certifi
[18]https://www.mongodb.com/
[19]https://github.com/mongodb/mongo-python-driver
[20]http://www.dabeaz.com/ply/

could be changed or even deleted.

To create the dataset, the public rules are requested from the Koodous database using a REST API client. The response on an API request is a JSON object containing the requested data and some extra metadata. All rules are collected using a for loop requesting rulesets within an ID range. This method has some limitations. Namely, the IDs of public rulesets in the Koodous database are not consecutive numbers and the lowest and highest public IDs are unknown. For this reason a static range of [0-8000] is used. The upper boundary of the range is determined by looking at the ID when creating a ruleset and then rounding it up to thousands. Collecting the rulesets is slow. This is due to a requests per minute limitation of the API which limits clients to 60 ruleset requests per minute. With 8000 IDs to request, this can add a lot of time to the collection process. Figure [5] shows the response codes when requesting IDs from range [0, 8000] and Table [1] shows the response code descriptions. As seen in the figure, the limit was reached 148 times. This means that collection was halted 148 times. Given a requests per minute limit of 60, the process takes $8000/60 = 133\frac{1}{3}$ or roughly 134 minutes. To avoid repeating this process, the collected data is stored.

Code	Description
200	Successful
404	Not found
429	Too many requests

Table 1: Response codes

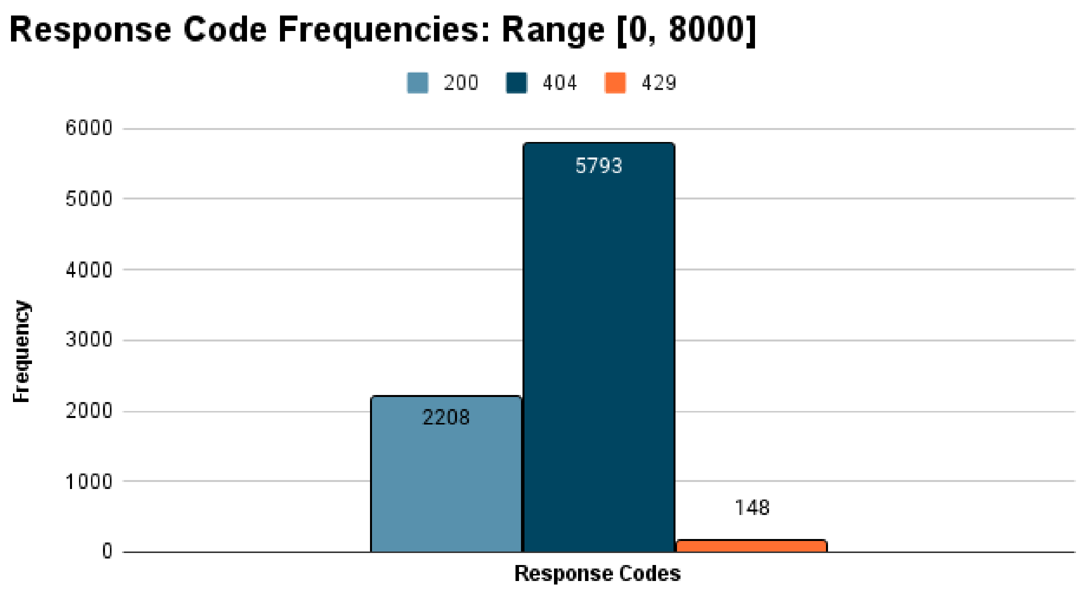


Figure 5: Dataset collection: Frequency of response codes

As the data is received in JSON, it is convenient to store it in JSON as well. The MongoDB database stores data as JSON documents. The result of every successful response is processed and stored.

After a successful response, the data goes through a preprocessing step. Unimportant data is cleaned and a SHA-256 hash of the ruleset content is added. This hash allows for quick lookups on for instance duplicate rulesets. After preprocessing, the data is stored in the MongoDB database. Figure [6] shows the structure of a ruleset database document. As seen in the figure, a ruleset document contains an ID, the ruleset itself, data about the author of the ruleset, and some metadata. Incorporating analyst data and other metadata can yield some interesting results. An example of this will be given in Section 4.3. With some modifications, other database implementations could also be used to store the rulesets. A NoSQL implementation is chosen as it fits best with the tools used.

3.2.1 Preprocessing: Lexing and Parsing

Before any threat intelligence can be extracted, the dataset, or more specifically the rulesets of the dataset, need to be preprocessed with a parser. A parser compatible with Python is best suited for this project as this fits best with the rest of the code. Multiple Python compatible YARA parsers are available. Some of the existing parser options are Plyara[21] and Yaramod[22]. Another option is a custom parser created specific for this project. Each parser has its pros and cons. The main objective of the parser is to visualize all components of a rule and make them accessible so they can be used to extract information after analysing them. Full compilation of YARA is not required. This results in the following requirements:

The parser must be able to:

(a) parse YARA syntactically

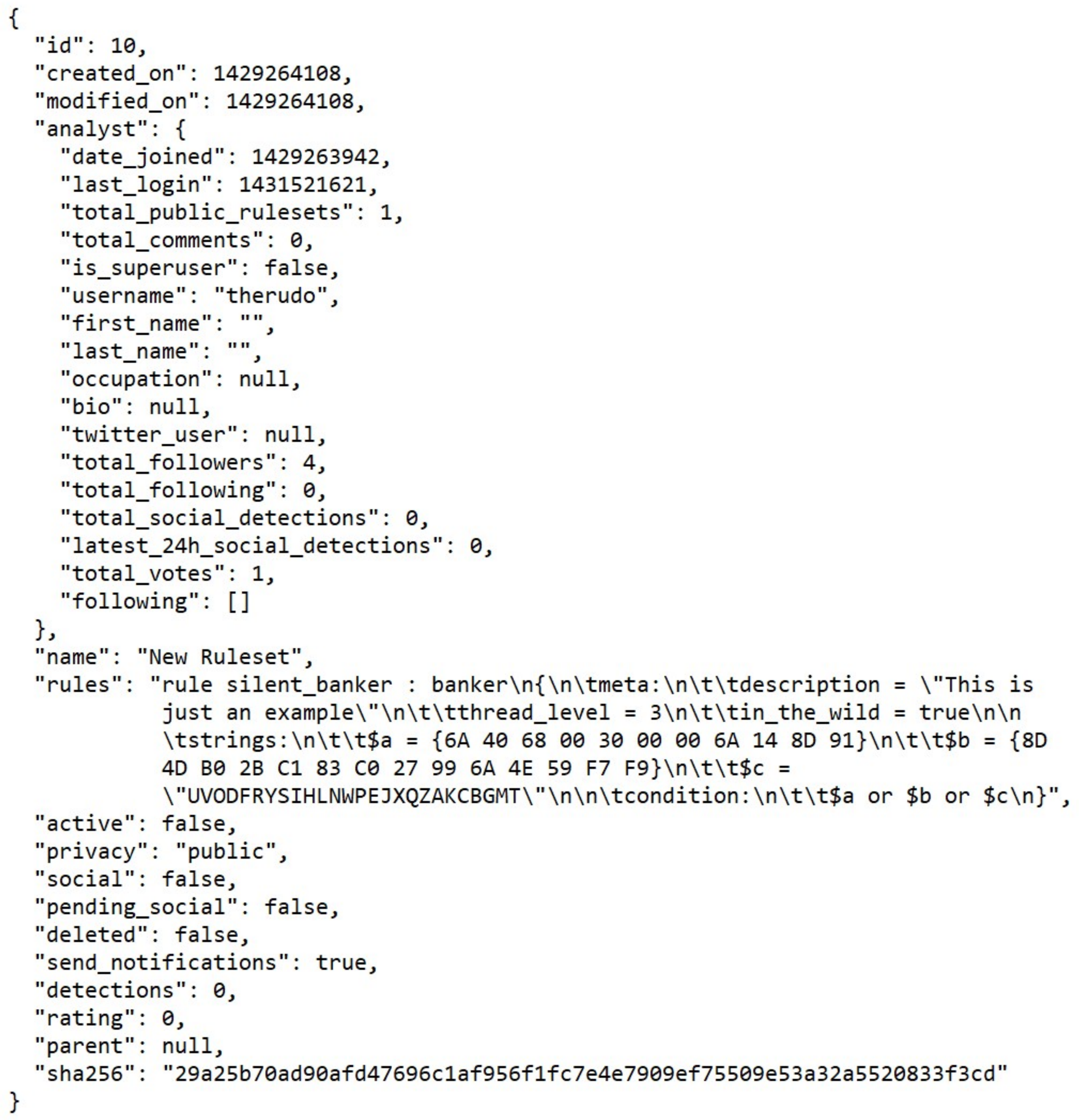
(b) transform the YARA Rule into a structure with accessible elements

(c) accept the modules integrated in Koodous' YARA implementation

The first option, Plyara, is a relatively simple parser. It only checks the syntax. This means that any module or function can be accepted as long as it is syntactically correct. Plyara outputs a dictionary of the parsed YARA rule. This can easily be transformed into a JSON object. Thus, Plyara satisfies the requirements for the scope of this project. The downside of Plyara is that it does not create a syntax tree, meaning condition analysis, or boolean analysis, is limited. This limits possibilities for future work.

The second option, Yaramod, is a more complex parser. It checks both syntax and semantics. Parsed rules are represented as a traversable abstract syntax tree (AST). However, a traversing program needs to be written in order to create an accessible output to satisfy requirement b. Because Yaramod also performs semantic checks it needs a description on external modules before they can

[21] https://github.com/plyara/plyara
[22] https://github.com/avast/yaramod



```
{
  "id": 10,
  "created_on": 1429264108,
  "modified_on": 1429264108,
  "analyst": {
    "date_joined": 1429263942,
    "last_login": 1431521621,
    "total_public_rulesets": 1,
    "total_comments": 0,
    "is_superuser": false,
    "username": "therudo",
    "first_name": "",
    "last_name": "",
    "occupation": null,
    "bio": null,
    "twitter_user": null,
    "total_followers": 4,
    "total_following": 0,
    "total_social_detections": 0,
    "latest_24h_social_detections": 0,
    "total_votes": 1,
    "following": []
  },
  "name": "New Ruleset",
  "rules": "rule silent_banker : banker\n{\n\tmeta:\n\t\tdescription = \"This is
           just an example\"\n\t\tthread_level = 3\n\t\tin_the_wild = true\n\n
           \tstrings:\n\t\t$a = {6A 40 68 00 30 00 00 6A 14 8D 91}\n\t\t$b = {8D
           4D B0 2B C1 83 C0 27 99 6A 4E 59 F7 F9}\n\t\t$c =
           \"UVODFRYSIHLNWPEJXQZAKCBGMT\"\n\n\tcondition:\n\t\t$a or $b or $c\n}",
  "active": false,
  "privacy": "public",
  "social": false,
  "pending_social": false,
  "deleted": false,
  "send_notifications": true,
  "detections": 0,
  "rating": 0,
  "parent": null,
  "sha256": "29a25b70ad90afd47696c1af956f1fc7e4e7909ef75509e53a32a5520833f3cd"
}
```

Figure 6: Example of extracted ruleset from Koodous REST API

be accepted. The biggest advantage of using Yaramod is the possibility of boolean analysis.

However, in this project I developed a custom parser specific to the goals of this work. Similar to Plyara, it creates a JSON/dictionary representation of the parsed ruleset and does not check for semantics. Because of this it satisfies all given requirements. With a custom parser, the functionality could be changed based on the needs of the project. In other words, the output can be controlled. If needed, the parser could be extended with AST functionality. With a custom parser there is full control over the way rulesets are parsed and what data needs to be saved.

The parser is a LALR(1) parser, which is short for Look-Ahead Left-to-Right parser. The "(1)" denotes a one-token look-ahead while parsing. It is an efficient bottom-up parsing method that parses text according to a set of production rules. More on the principles of parsing can be found in [24]. Following is a brief explanation of the parsing process.

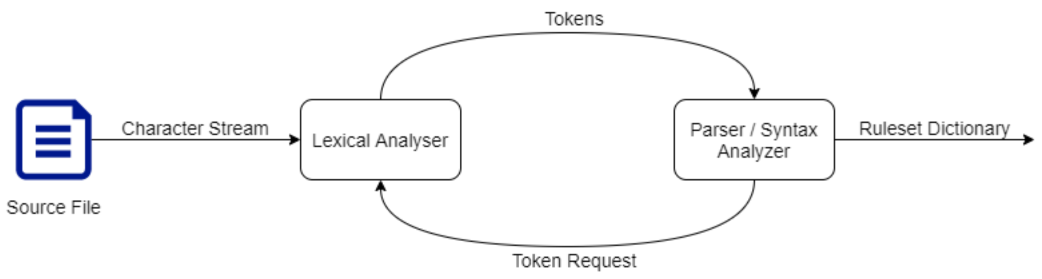


Figure 7: Parsing Process

Before the production rules can be applied, the input character stream needs to be separated into tokens representing terminals. This is done by the lexical analyser, or simply lexer. Every defined token is assigned a regular expression (regex). A list of tokens is created based on regex matches with the input. These tokens have a precedence that determines the order in which the lexer tries matching a piece of the input to a certain regex and token. The order is based on the definition in the code. Earlier defined tokens are checked earlier in matching. The characters of the input are scanned from left to right. Scanned characters are appended to a string buffer and checked against the regular expressions. On a match the corresponding token is appended to the token list. Its value is the matched string. The string buffer is then cleared and scanning continues. This results in a token stream that is fed to the parser.

The parser requests one token at a time from the lexer and can look ahead for an extra token as specified by LALR(1). The lookahead token is used to choose the production rule best fitting the token stream. The token stream should be a production of the grammar described by the production rules. When this is not the case, a syntax error occurs and parsing stops. There is no output in this case. For this project, the production rules are created based on the YARA documentation[23]. While parsing, a dictionary representation of the ruleset is created. This format can easily be converted to JSON. The structure of such a dictionary is shown in Figure [8]. All tokens and their values are added to the dictionary. All elements within the ruleset easily accessible. This allows for better analysis and gathering of statistics. A token is added to the dictionary after it has been handled. In other words, a token is added to the dictionary after the production of up to and including this token is complete.

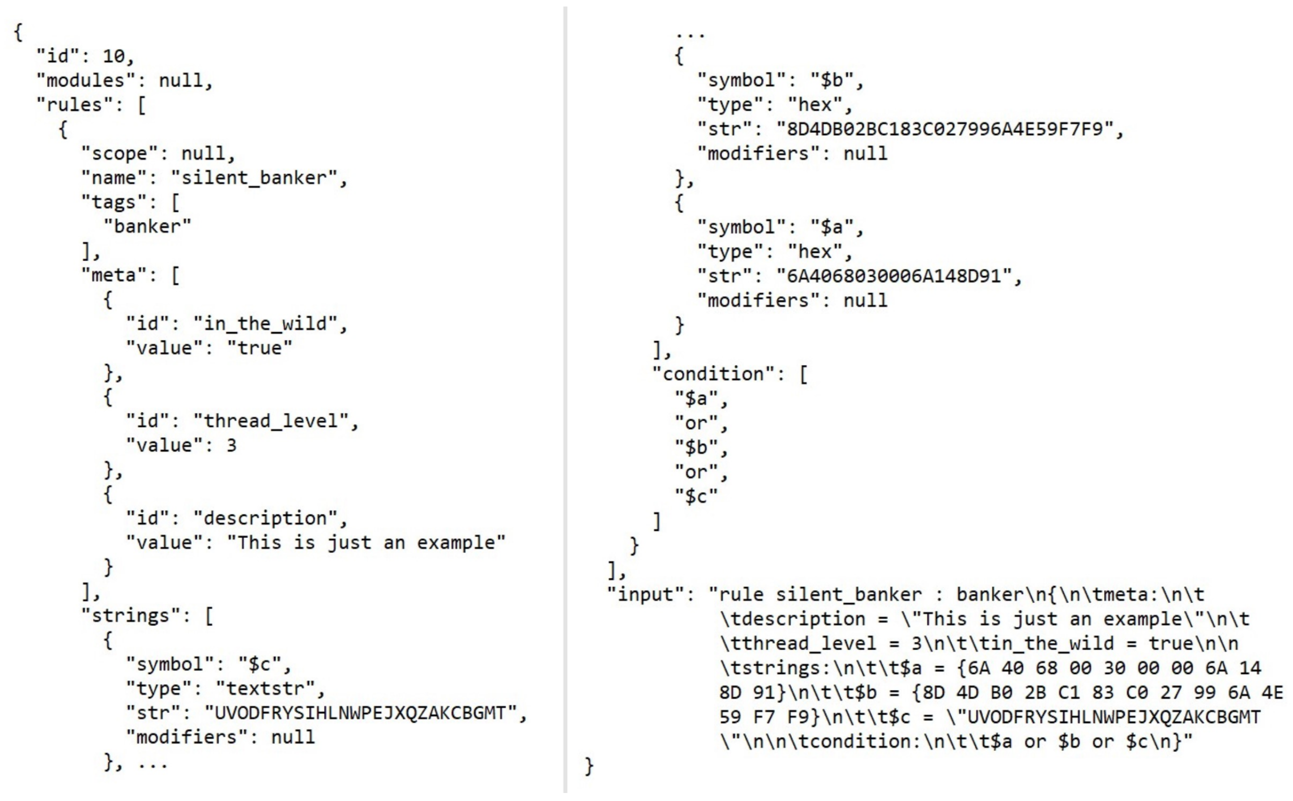
3.2.2 Parser example

In short, a character stream is inputted into the lexical analyser, which transforms it into a token stream. Then, one token at a time, the parser tries to apply a received token into the best fitting

[23]https://yara.readthedocs.io/en/v4.1.2/index.html

production rule. After a token is handled, the parser requests a new one from the lexical analyser until all tokens are handled or a syntax error occurs. This process is visualised in Figure [7].

Figure [8] shows the parser output obtained from parsing the YARA rule shown in Figure [3]. The output dictionary contains the ruleset ID, a list of imported modules, and a list of rules. The 'rules' list consists of dictionaries representing the rules present in the ruleset. Within such a rule dictionary the rules strings are classified and put into their according section, 'meta' or 'strings'. The condition is split and put into a list of elements. Every created dictionary is stored and can be used for analysis later.



```
{
  "id": 10,
  "modules": null,
  "rules": [
    {
      "scope": null,
      "name": "silent_banker",
      "tags": [
        "banker"
      ],
      "meta": [
        {
          "id": "in_the_wild",
          "value": "true"
        },
        {
          "id": "thread_level",
          "value": 3
        },
        {
          "id": "description",
          "value": "This is just an example"
        }
      ],
      "strings": [
        {
          "symbol": "$c",
          "type": "textstr",
          "str": "UVODFRYSIHLNWPEJXQZAKCBGMT",
          "modifiers": null
        }, ...
```

```
          ...
        {
          "symbol": "$b",
          "type": "hex",
          "str": "8D4DB02BC183C027996A4E59F7F9",
          "modifiers": null
        },
        {
          "symbol": "$a",
          "type": "hex",
          "str": "6A4068030006A148D91",
          "modifiers": null
        }
      ],
      "condition": [
        "$a",
        "or",
        "$b",
        "or",
        "$c"
      ]
    }
  ],
  "input": "rule silent_banker : banker\n{\n\tmeta:\n\t
            \tdescription = \"This is just an example\"\n\t
            \tthread_level = 3\n\t\tin_the_wild = true\n\n
            \tstrings:\n\t\t$a = {6A 40 68 00 30 00 00 6A 14
            8D 91}\n\t\t$b = {8D 4D B0 2B C1 83 C0 27 99 6A 4E
            59 F7 F9}\n\t\t$c = \"UVODFRYSIHLNWPEJXQZAKCBGMT
            \"\n\n\tcondition:\n\t\t$a or $b or $c\n}"
}
```

Figure 8: Parser output example

4 Threat Intelligence Feed

The YARA rules from the dataset contain useful threat intelligence data. In this section we discuss the transformation of a YARA rule into threat intelligence elements. To achieve this transformation, we need to understand how the rule elements are structured. The most important sources of threat intelligence in YARA rules are the module funtion calls and the (non-meta) string values. Therefore, the rest of the section will focus on handling these two element types.

4.1 Parsing data from modules

Within YARA rules, module function calls are annotated with their type, which can be inferred from the module name and the function name, e.g., `androguard.package name(/com.video.uiA/i)` can be classified as a package name. The argument of the function, `/com.video.uiA/i`, represents the element's value, type, and optional modifiers. In this case the type is a regular expression, the value `is com.video.uiA`, and the modifier `i` is used, which means the value is case sensitive. In general, the structure of a module function call can be roughly described with the regular expression:

$$module(.function\_name)^{+}\backslash(argument(,argument)^{*}\backslash)$$

With *module* and *function_name* being regular text strings and *argument* being a value between two characters, optionally followed by a modifier. The two outer characters of the value denote its type. Forward slashes, "`/.../`", indicate that the value is a regular expression, like in the example: `/com.video.uiA/i`. Curly brackets, "`{...}`", indicate a hexadecimal value and double quotes, "`"..."`" indicate a textual string. The available modules in Koodous YARA are Androguard[24], Droidbox[25], File[26], and Cuckoo[27].

There are a lot of module functions available, meaning that a lot of indicators can be extracted from them. Different host-based and network indicators are present. Part of the host-based indicators are application components. Known components of an application can be used to identify the application. Table [2] shows application components that can be found in the function calls.

Component	Description
Activity	UI screen within the application.
Library	Android app module that can be included in other app modules.
Service	Performs long-running operations in the background without a UI.
Receiver	Enables applications to receive intent messages broadcasted by the system or other applications. Keeps enabled when other app components are not running.
Filter	Specifies the types of intent messages that an activity, service, or receiver can respond to. Filters out irrelevant intent messages.

Table 2: Application component indicators found in function calls

[24] http://docs.koodous.com/yara/androguard/
[25] http://docs.koodous.com/yara/droidbox/
[26] http://docs.koodous.com/yara/file/
[27] https://yara.readthedocs.io/en/stable/modules/cuckoo.html

Following are two lists detailing the supported function calls and indicators found in them:

Host-based indicators:

- `androguard.package_name`: Package name. Android applications should have a unique package name. However, malware developers often use the same or similar package names for multiple applications.
- `androguard.app_name`: Application name. Fake apps can have the same name as the official. The name itself is not useful. However, keeping track of the application names that are used in fake apps combined with other IOCs from that fake app can provide valuable information.
- `androguard.activity`: Activities. Known activity names can be used as indicators.
- `androguard.permission`: Permissions. Applications need a set of permissions to work properly. Therefore, detecting the declared permissions of an application can be used to identify it.
- `androguard.certificate.(sha1,issuer,subject)`: Certificate. Each application has a certificate. This certificate contains a hash and information about the developer (issuer/subject of the certificate). Malicious developers can use the same certificate for many application samples. Therefore, it can be used to indicate a potential threat.
- `androguard.receiver`: Receivers. The receiver component name can be used to identify certain applications.
- `androguard.sevice`: Services. Services are used to run tasks in the background. Malware can use this to secretly send and receive data. Detecting specific services names can therefore help malware detection.
- `androguard.filter`: Filters. As with receivers, intent filter component names can be used to identify an application.
- `droidbox.library`: Libraries. Applications can use compromised libraries that contain malware. It is therefore important to check for known compromised libraries.
- `droidbox.(written, read).(filename, data)`: Written/Read files. Some malware write and/or read specific files. Detecting a malware's file write/read patterns can help detecting it.
- `file.(md5, sha1, sha256)`: Hash. The File module supports the detection of MD5, SHA1, and SHA256 hashes. Using the hashes, malicious files can be stored and used for detection.

Network indicators:

- `androguard.url`: URL. There are a lot of compromised URLs. Their intentions range from delivering malware to stealing credentials. Applications referring you to such websites can be considered malicious.

- `droidbox.sendsms`: Send SMS. In Android, applications can send an SMS. This can be exploited by sending an SMS to a premium, paid, destination in the background. For example, an executed Android trojan typically sends an SMS to a list of premium numbers, stealing money in the process. Tracking such activity is important to prevent this.

- `droidbox.phonecall`: Phone calls. As with unauthorised SMS sending, phone calls could also be performed in the background without the user initiating the call. As one can imagine, this can be used with malicious intent.

- `cuckoo.network.http_request`: HTTP request. Instead of looking at the URLs specifically, requests to compromised URLs could also be used for malware detection.

- `cuckoo.network.dns_lookup`: DNS lookup. A request to a DNS server for a compromised domain indicates malicious behaviour.

4.2 Parsing string values

In contrast to module function calls, strings do not have naming conventions to help infer it's values data type. The string identifier and value can mean anything ranging from a random character stream to a valuable piece of information. There is no naming convention for strings. Because of this, it is not very useful to look at the identifier of a string to determine the data type of the string's value. Strings are often named `$a`, `$b`, `$string1`, `$a0`, etc. This leaves inferring the data type with regular expressions as the only feasible option. But what kind of data can be inferred from string values? As mentioned in Section 2.6, strings represent IOCs that describe a particular malware strain. This means that string values represent a form of technical TI. It is therefore important to look at common indicators found in (Android) malware and create patterns matching with them.

4.2.1 Network indicators

First, there are the general network indicators. As mentioned in Section 2.2, this includes IP addresses, URLs, and domain names. They are all structured data types that can be matched with a regular expression. An IP address can have two types: IPv4 or IPv6. An IPv4 address can easily be described with the following regex:

$$\text{IPv4: “}\backslash d\{1,3\}\backslash.\backslash d\{1,3\}\backslash.\backslash d\{1,3\}\backslash.\backslash d\{1,3\}\text{”}$$

IPv6 addresses are significantly longer than IPv4 addresses: 128 bits compared to 32 bits. In addition, a set of zeros can be omitted with a special abbreviated notation. Because of this, an IPv6 regex is very complex. An example of an IPv6 regex can be viewed at Stackoverflow[28]. Matched values need to be validated in order to make sure the values are indeed IP addresses. Creating a socket with the matched value is a good way to validate potential IP addresses. If the operation fails the value is not a valid IP address. If it succeeds, the value is a valid IP address. URLs are characterized by a protocol (http) followed by a domain name and optionally a resource name. Domain name and resource names can be useful threat information. Within a URL, the protocol is a very distinctive sequence of characters (http(s)://), making it a suitable sequence to match values

[28] https://stackoverflow.com/questions/53497/regular-expression-that-matches-valid-ipv6-addresses

on. This also holds for the top-level domain (TLD). Common TLDs are 'com', 'org', 'net', 'info', and 'ru'. If a dot followed by a TLD name is encountered, it is very likely that the value is a domain name.

Referring back to the pyramid of pain, the above handled network indicators fall in the lower half of the pyramid. This means that threat actors can relatively easily adapt these values to circumvent a defense based on these indicators. Therefore, the lifetime of these indicators is short. However, this does not mean that they have no value. They can serve as a 'first line of defense' until higher level threat information is acquired. After that, they can serve as a supplement to the higher level threat information and they can be used in system monitoring.

4.2.2 Host-based indicators

Second, there are host-based indicators. File hashes are very common in this type of indicator. However, given only a string value, there is no way to determine if it is a hash or what hash algorithm is used. The value could just as much be a random character stream. The only clue as to what hash algorithm is used is the fixed length of the output. However, there are multiple hash functions with the same fixed output length. The only option is to check the length of the string value and compare it the output length of commonly used hash algorithms.

Application resources can be extracted from strings more reliably. This is because they are externalized in a special resource folder[29] (`Project/res/`). Therefore, string values can be matched with `res/` to match resource files. Resources include image files, xml files, arbitrary raw files, and more. In addition to looking specifically at the resources folder, it is also possible to match resources based on their file extension. Table [3] shows the file extensions that where encountered while analyzing the dataset. More extensions could be added in the future based on findings in the dataset. Some files are too general to be considered an indicator. The Android application manifest "`AndroidManifest.xml`" is a very good example of this. This file is required to be at the root of every project. Therefore, it is too general and should be excluded from the TI feed.

Host-based indicators, except hash values, fall in the Host Artifacts category of the pyramid of pain. Hash values are very easy to change and therefore reside at the lowest level of the pyramid. Hashes are still important for detection. However, their lifetime is even shorter than the lifetime of the earlier mentioned network indicators. Resource files in specific locations are harder to change. They can be location dependent for example. Basing a defense on a malware's (injected) resources is therefore harder to circumvent. This makes resource information more valuable.

[29]https://developer.android.com/guide/topics/resources/providing-resources

Extension	**File type**
.zip	Zip compressed folder
.jpg	Compressed image file
.png	Portable Network Graphic image file
.txt	Plain text file
.xml	XML data file
.mp3	Compressed audio file
.cc	C++ source code file
.so	Linux/Android shared library file
.jar	Java Archive file
.pw	Pathetic Writer document file
.tr	TomeRaider 2 eBook file
.xyz	Molecule Specification File
.php	PHP source code file
.mobi	Mobipocket reader eBook file
.info	Texinfo formatted information document

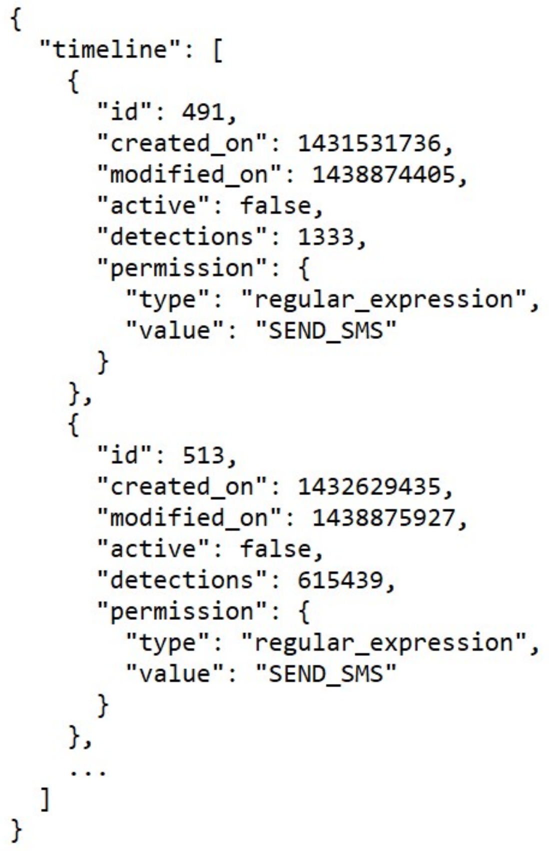
Table 3: Matched file extensions and their file type.

4.3 Deriving higher level threat intelligence

Individual indicators or indicator sets, as discussed in Section 4.1 and 4.2, apply to specific malware strains or threat actors. This low-level intelligence can be used directly in malware detection (e.g., scanning). The indicators are obtained by analyzing individual YARA rules. Apart from using individual YARA rules, it is also possible to obtain intelligence by combining different rules and the metadata associated with them. An example of this is a timeline of permission requests found in the rules (and thus in malware). This can be a full timeline or in periods. Figure [9] shows a sample of the full timeline. As seen in the figure, every permission request is saved together with some metadata of the corresponding YARA ruleset. The '`created_on`' and '`modified_on`' are epoch timestamps denoting the creation date and last modification date of the ruleset. The '`detections`' refer to the amount of APKs from the Koodous repository the corresponding YARA ruleset matches (on the date of gathering the dataset).

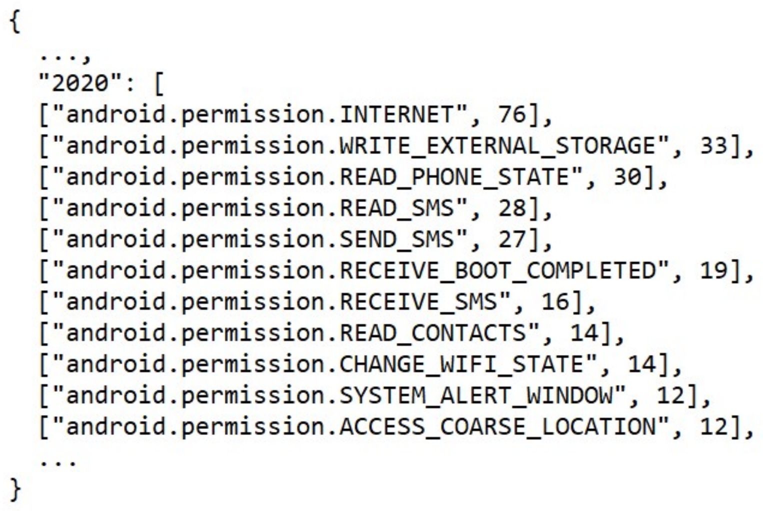
Using the full timeline, it is possible to create timelines with specific periods. Figure [10] shows a sample of the timeline denoting the frequency of different permission requests within a year. Unlike individual indicators and indicator sets, the permission timeline cannot be used directly in malware detection. However, it's value comes from the high-level information it provides about risks associated with certain permissions. It can therefore be classified as strategic intelligence.

The permission timeline is only the first intelligence obtained from combined YARA rulesets. There are multiple possibilities for further research regarding combined YARA rulesets. There may be a way to find and use similarities between different rules. This includes similar IOCs or sets of IOCs from different rules often found together.



```
{
  "timeline": [
    {
      "id": 491,
      "created_on": 1431531736,
      "modified_on": 1438874405,
      "active": false,
      "detections": 1333,
      "permission": {
        "type": "regular_expression",
        "value": "SEND_SMS"
      }
    },
    {
      "id": 513,
      "created_on": 1432629435,
      "modified_on": 1438875927,
      "active": false,
      "detections": 615439,
      "permission": {
        "type": "regular_expression",
        "value": "SEND_SMS"
      }
    },
    ...
  ]
}
```

Figure 9: Timeline for each permission request



```
{
  ...,
  "2020": [
  ["android.permission.INTERNET", 76],
  ["android.permission.WRITE_EXTERNAL_STORAGE", 33],
  ["android.permission.READ_PHONE_STATE", 30],
  ["android.permission.READ_SMS", 28],
  ["android.permission.SEND_SMS", 27],
  ["android.permission.RECEIVE_BOOT_COMPLETED", 19],
  ["android.permission.RECEIVE_SMS", 16],
  ["android.permission.READ_CONTACTS", 14],
  ["android.permission.CHANGE_WIFI_STATE", 14],
  ["android.permission.SYSTEM_ALERT_WINDOW", 12],
  ["android.permission.ACCESS_COARSE_LOCATION", 12],
  ...
}
```

Figure 10: Permission requests per year

5 Results

This section covers the IOC collection that is extracted from the dataset. The collected intelligence is presented in the form of JSON-files. Each type of information is put into a separate file. This results in a total of 80 files. Two of these files are the timeline files discussed in Section 4.3. The dataset has a timeline that starts in 2015 (with the release of Koodous) and continues to November 2021. Table [4] and Figure [11] show the amount of permission requests encountered in the dataset for each year. As can be seen in the figure, the amount of permissions requested differs greatly per year. A spike in permission requests can mean that there are more malware strains that need permissions in order to operate. The frequency of permission requests in general may indicate a trend in permission abuse. While this is useful at a high level, looking at individual permission frequencies provides better information about specific permission trends. Figure [12] and [13] show the top 10 and top 5 most requested permissions per year. As can be seen in the figures, some permissions see a spike in requests in certain years. Take 2020 for example. In this year, the permission `android.permission.INTERNET` was requested 76 times compared to 16 times a year earlier. That is an increase of 375%. Trends like these indicate the level of caution needed before granting the permissions.

Year	**Permission Requests**
2015	49
2016	109
2017	328
2018	285
2019	94
2020	501
2021	132
Total	**1498**

Table 4: Permission requests per year

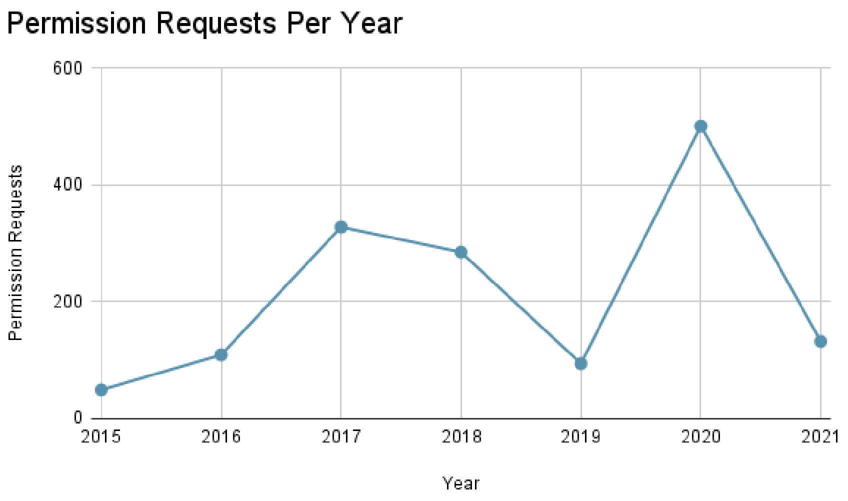


Figure 11: Permission Requests Per Year

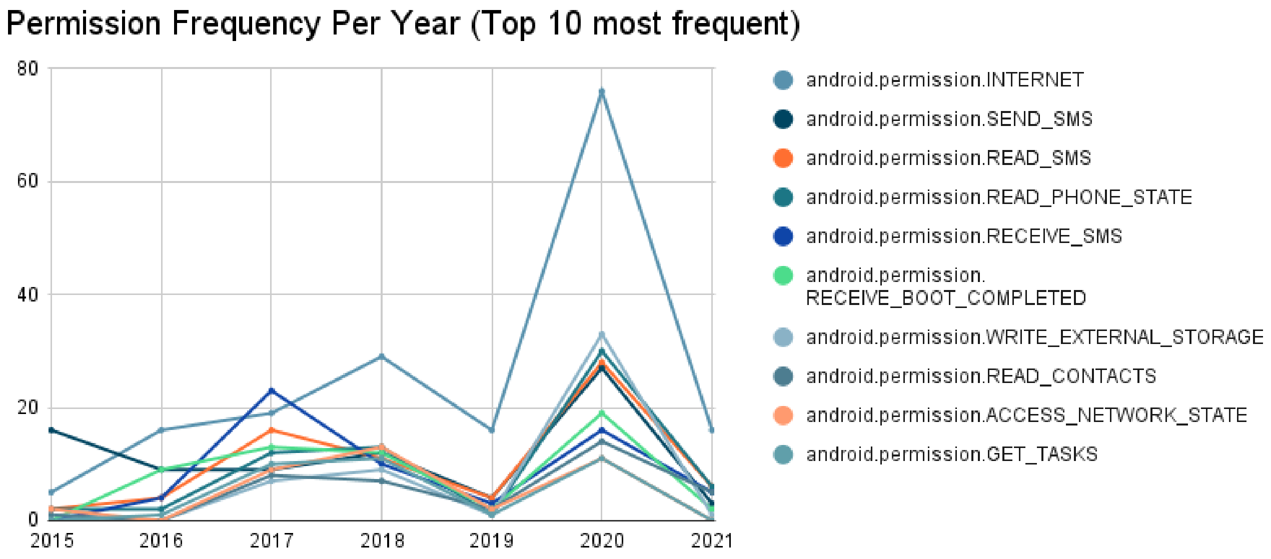


Figure 12: Permission (Top 10) Frequency Per Year

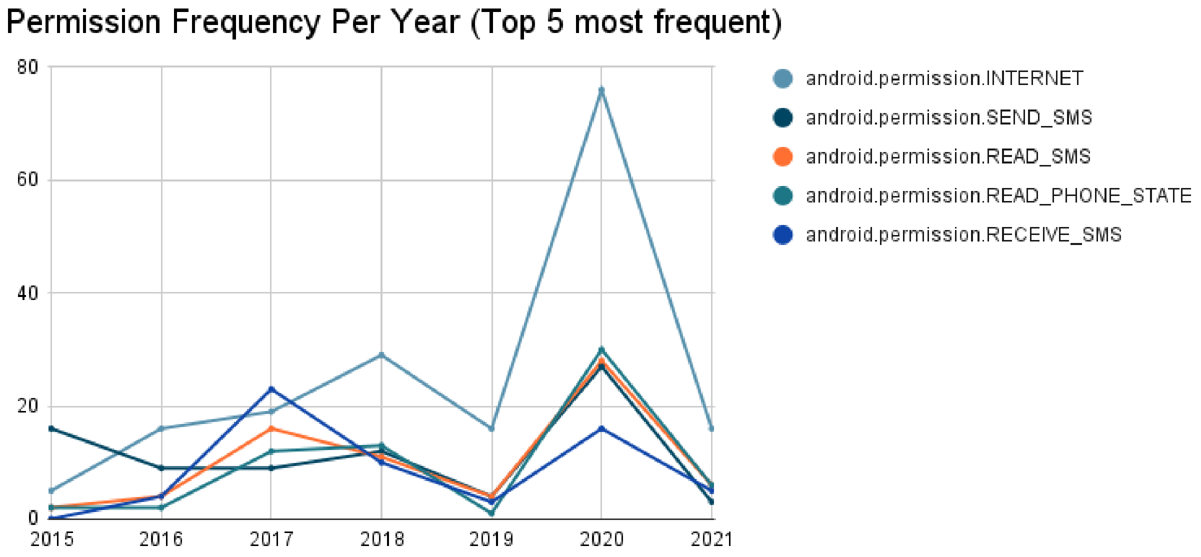


Figure 13: Permission (Top 5) Frequency Per Year

The remaining 78 files contain IOCs. In total, a collection of 9095 IOCs is extracted from the dataset. Figure [14] shows the amount of IOCs of specific types. As can be seen in the figure, this approach allows the collection of many URLs, DNS lookups, domain names, IP addresses, etc. These network indicator values can be used to configure the security operations to detect connections to or from these network locations. Apart from network indicators, this approach also allows the collection of a lot of host-based indicators like permissions, package names, resource files (file), hashes, etc. Host-based indicators can be used to identify their corresponding application or application component. This means these values can be used to support both static and dynamic malware detection approaches.

In Figure [14], the hash type denotes a collection of values that is probably a hash. The type of hash is either estimated by a prefix, such as `"SHA-1:"`, or by the length of the value. As these values are less reliable than confirmed hashes, they are separated from the confirmed hashes.

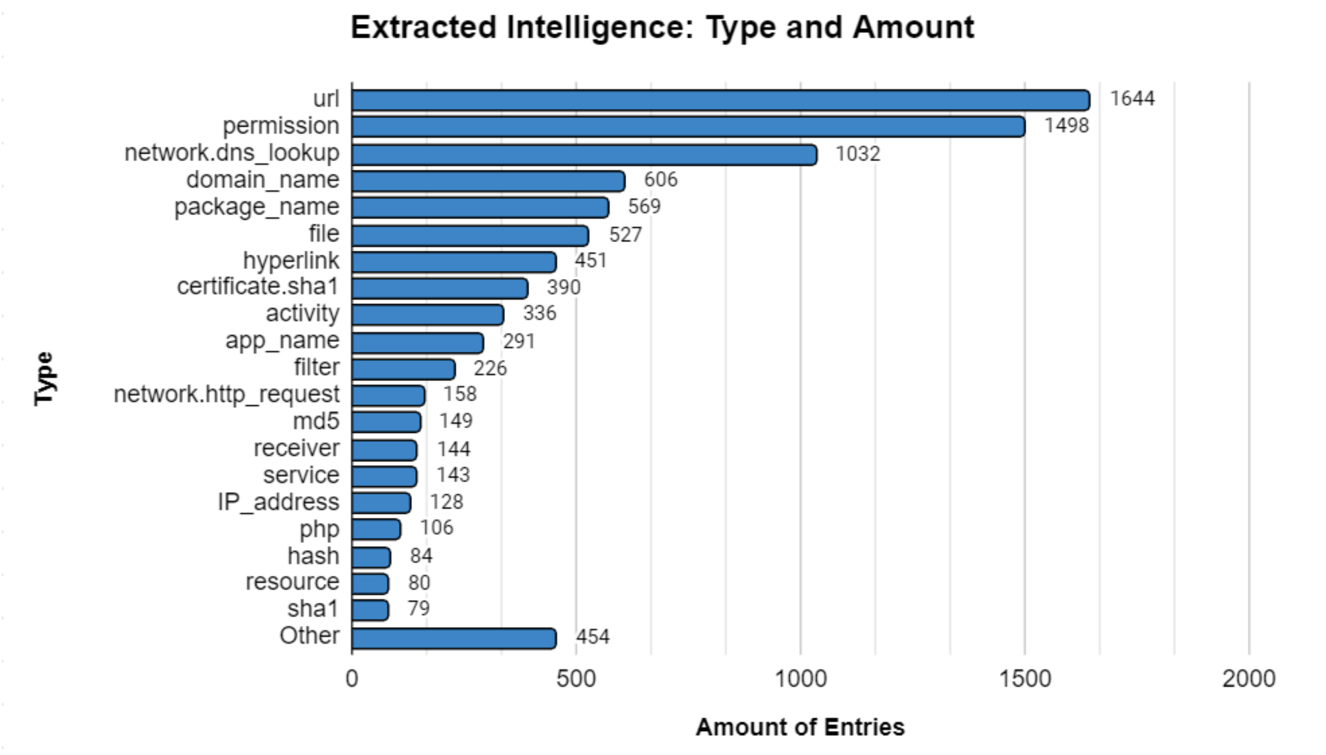


Figure 14: Extracted intelligence type and amount

The frequency of IOC types changes from year to year. This can be seen in Figure [15] and Table [5]. Again, the trends of IOC types indicate what is more frequently used in discovered malware strains. However, unlike permissions, most value types do not have a relatively small predefined set of possible values that cannot be altered without changing its result. A different permission has a different result while a different filename may just be an obfuscation through renaming. This makes individual value trends less reliable. However, IOC type trends do not suffer from this as altered values will still be included. Therefore, IOC type trends give a more reliable indication of what is used in malware strains. Individual value trends can also be useful, but only if the value's type has a set amount of values where each value has a set result (like a permission).

Figure [16] and Figure [17] show the top 5 and top 5-10 most frequent IOC types. Here, the same holds as with permissions. The trend of an IOC type shows how often this type is used to identify discovered malware strains. Therefore, it indicates the level of caution needed when encountering such values. For example, in 2018, where a lot of URL values are discovered in malware, it is logical to validate encountered URLs more thoroughly than usual.

Year	**IOCs**
2015	1069
2016	1121
2017	1590
2018	3171
2019	516
2020	1271
2021	357
Total	**9095**

Table 5: IOCs Per Year

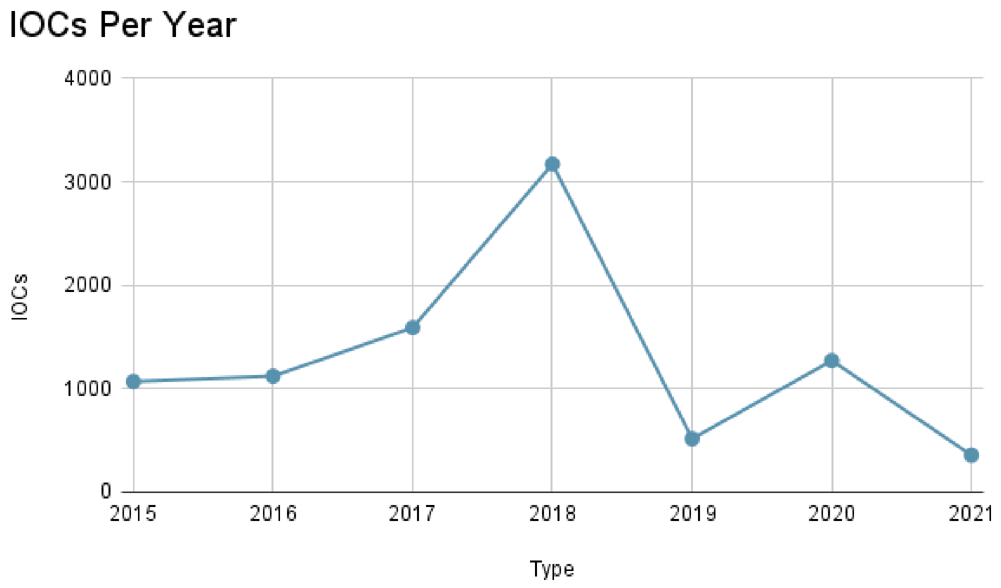


Figure 15: IOCs Per Year

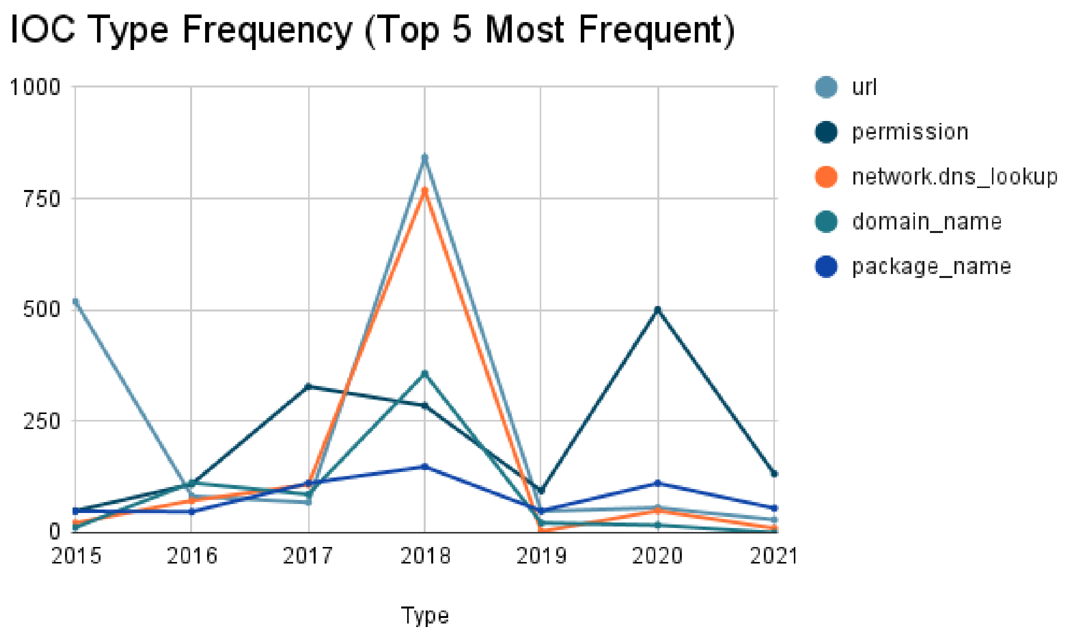


Figure 16: IOC Type (Top 5) Frequency

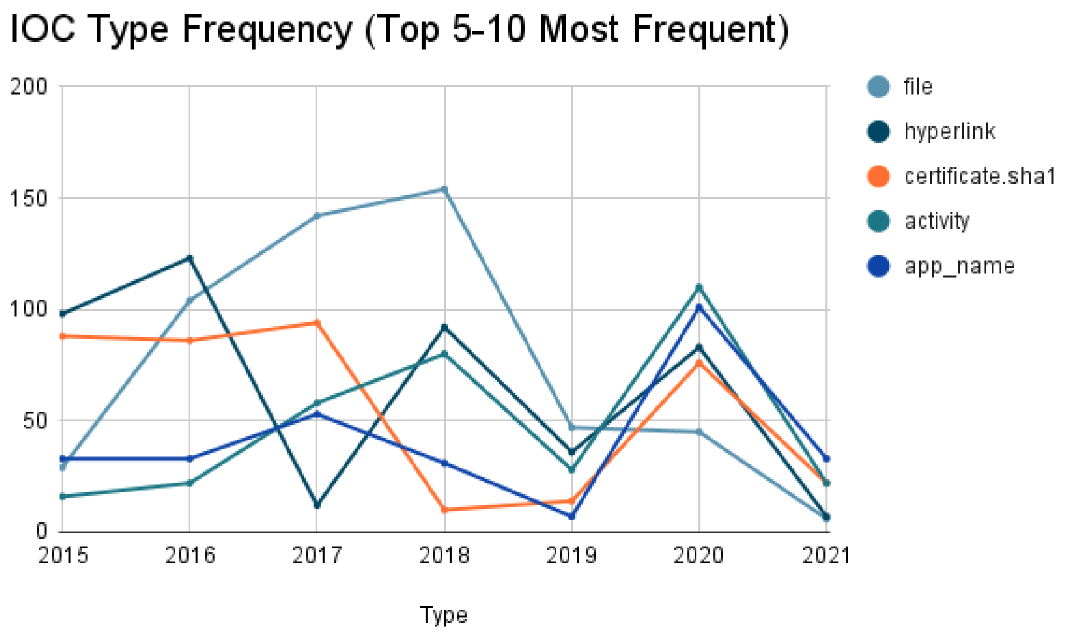


Figure 17: IOC Type (Top 5-10) Frequency

6 Limitations

This section will discuss the limitations of this project. First, the Koodous platform. The REST API of Koodous does not provide a list of valid ruleset IDs. In addition, the amount of requests to the server is limited. Because of this, collecting and updating the dataset is a long process. Second, the IOC extraction technique used only supports syntactical validation of the YARA rules, not semantic validation. As a result, only individual IOC values can be extracted. Boolean operations, like AND or OR, are not used in the value extraction process. This is reflected in the results, or Section 5, where only single IOC values and types are discussed. In addition to this, with string value parsing, as described in Section 4.2, regular expression or hexadecimal values are not evaluated. The values are only matched with predefined string patterns. In other words, regular expression and hexadecimal string values are treated as regular string values. The lack of semantic validation and regular expression and hexadecimal evaluation results in missed and invalid values. In Figure [18] the distribution of IOC values is shown. The invalid IOCs account for 2.66% of all values (242 out of 9095). The amount of missed IOCs cannot be accurately shown as a lot of the regular expressions do not contain any operators and is thus semantically equivalent to a regular string value. However, as can be seen in Figure [19], with certain value types the proportion of regular expression values can be significantly high. It is therefore logical to assume that a lot of IOCs are either incorrect or missing.

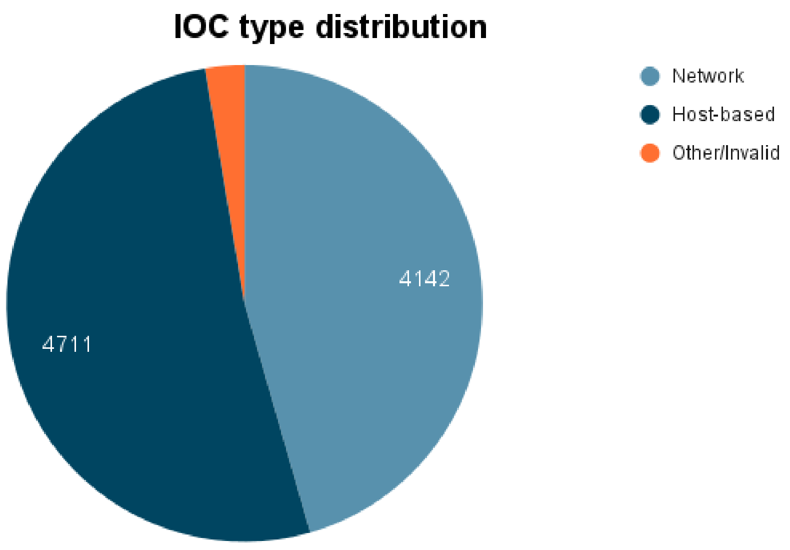


Figure 18: IOC type distribution

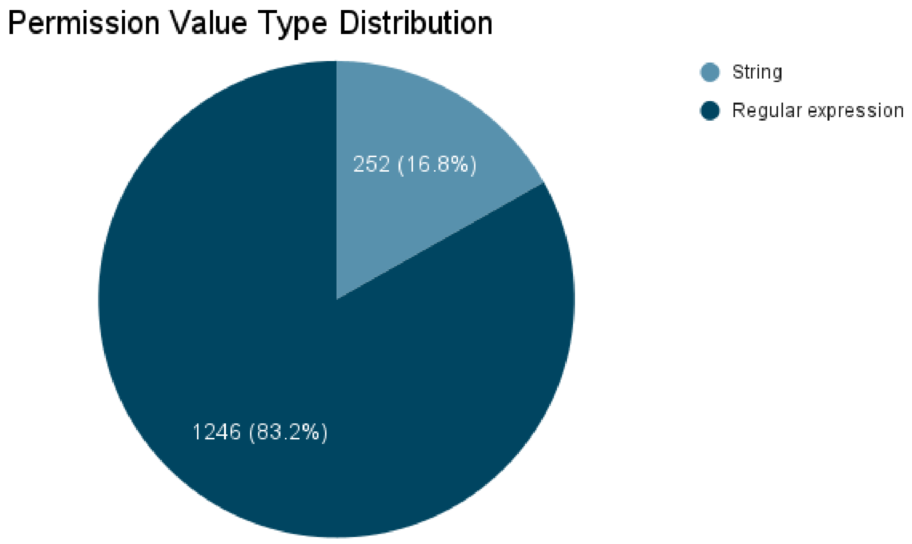


Figure 19: Permission value type distribution

7 Conclusion And Future Work

Mobile malware and malware in general has been a rising threat for years. Sharing threat intelligence is very important to combat these malware threats. This paper introduced the idea of extracting TI from YARA rules and sharing it in the form of a TI feed. I showed that it is possible to extract different types of IOCs and that there are possibilities for higher level intelligence extraction. The result of this work is a proof of concept TI feed containing IOCs and a permission request timeline. Currently, the project only supports basic functionalities. As discussed in Section 6, a significant proportion of IOCs is missed. In addition, a small proportion of IOCs is invalid. A logical next step will be to extend the IOC extraction capabilities in order to increase the amount of extracted IOCs and improve their quality. Another direction for future work is to explore the possibilities of boolean tree analysis. A lot of intelligence lies in the relation of IOCs. Incorporating techniques to define the relations between IOCs may yield some useful intelligence.

References

[1] S. Arshad, M. A. Shah, A. Khan, and M. Ahmed, "Android malware detection & protection: a survey," *International Journal of Advanced Computer Science and Applications*, vol. 7, no. 2, pp. 463–475, 2016.

[2] F. Wei, Y. Li, S. Roy, X. Ou, and W. Zhou, "Deep ground truth analysis of current android malware," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 252–276, Springer, 2017.

[3] P. Faruki, A. Bharmal, V. Laxmi, V. Ganmoor, M. S. Gaur, M. Conti, and M. Rajarajan, "Android security: a survey of issues, malware penetration, and defenses," *IEEE communications surveys & tutorials*, vol. 17, no. 2, pp. 998–1022, 2014.

[4] R. Mayrhofer, J. V. Stoep, C. Brubaker, and N. Kralevich, "The android platform security model," *ACM Transactions on Privacy and Security (TOPS)*, vol. 24, no. 3, pp. 1–35, 2021.

[5] W. Enck, M. Ongtang, and P. McDaniel, "Understanding android security," *IEEE security & privacy*, vol. 7, no. 1, pp. 50–57, 2009.

[6] W. Enck, D. Octeau, P. D. McDaniel, and S. Chaudhuri, "A study of android application security.," in *USENIX security symposium*, vol. 2, 2011.

[7] R. McMillan, "Definition: Threat intelligence."

[8] M. Bromiley, "Threat intelligence: What it is, and how to use it effectively," *SANS Institute InfoSec Reading Room*, vol. 15, p. 172, 2016.

[9] J. Andress, "Working with indicators of compromise," *Journal Information Systems Security Association (ISSA)*, vol. 5, pp. 14–20, 2015.

[10] W. Tounsi and H. Rais, "A survey on technical threat intelligence in the age of sophisticated cyber attacks," *Computers & security*, vol. 72, pp. 212–233, 2018.

[11] D. J. Bianco, "The pyramid of pain."

[12] M. Park, J. Seo, J. Han, H. Oh, and K. Lee, "Situational awareness framework for threat intelligence measurement of android malware.," *J. Wirel. Mob. Networks Ubiquitous Comput. Dependable Appl.*, vol. 9, no. 3, pp. 25–38, 2018.

[13] J. Grisham, S. Samtani, M. Patton, and H. Chen, "Identifying mobile malware and key threat actors in online hacker forums for proactive cyber threat intelligence," in *2017 IEEE International Conference on Intelligence and Security Informatics (ISI)*, pp. 13–18, IEEE, 2017.

[14] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. Siemens, "Drebin: Effective and explainable detection of android malware in your pocket.," in *Ndss*, vol. 14, pp. 23–26, 2014.

[15] M. Odusami, O. Abayomi-Alli, S. Misra, O. Shobayo, R. Damasevicius, and R. Maskeliunas, "Android malware detection: A survey," in *International conference on applied informatics*, pp. 255–266, Springer, 2018.

[16] R. Zachariah, K. Akash, M. S. Yousef, and A. M. Chacko, "Android malware detection a survey," in *2017 IEEE international conference on circuits and systems (ICCS)*, pp. 238–244, IEEE, 2017.

[17] M. S. Rana, C. Gudla, and A. H. Sung, "Evaluating machine learning models for android malware detection: A comparison study," in *Proceedings of the 2018 VII International Conference on Network, Communication and Computing*, pp. 17–21, 2018.

[18] G. Robinson and G. R. Weir, "Understanding android security," in *International Conference on Global Security, Safety, and Sustainability*, pp. 189–199, Springer, 2015.

[19] N. Idika and A. P. Mathur, "A survey of malware detection techniques," *Purdue University*, vol. 48, no. 2, 2007.

[20] B. Baskaran and A. Ralescu, "A study of android malware detection techniques and machine learning," 2016.

[21] M. Brengel and C. Rossow, "Yarix: Scalable yara-based malware intelligence," in *USENIX Security Symposium*, 2021.

[22] E. Raff, R. Zak, G. Lopez Munoz, W. Fleming, H. S. Anderson, B. Filar, C. Nicholas, and J. Holt, "Automatic yara rule generation using biclustering," in *Proceedings of the 13th ACM Workshop on Artificial Intelligence and Security*, pp. 71–82, 2020.

[23] N. Naik, P. Jenkins, N. Savage, L. Yang, K. Naik, and J. Song, "Embedding fuzzy rules with yara rules for performance optimisation of malware analysis," in *2020 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE)*, pp. 1–7, IEEE, 2020.

[24] A. Aho, *Compilers: Principles, Techniques, and Tools*. Always Learning, Pearson, 2014.