ScrAPK: A Web Scraping Tool that Collects Potentially Fake and Rogue APKs from Alternative App Distributors

Thang Le, Warren Wang
University of Kansas
minhthang14900@ku.edu, warrenw@ku.edu

I. Introduction

A big technology company's online presence is one of its most significant competitive advantages in today's digital world. This makes it a prime target for malicious attackers trying to impersonate or tarnish these companies' brands for their own personal profit. Through a semi-related research paper: How Did That Get Into My Phone? Unwanted App Distribution on Android Devices [1], we learned how these malicious individuals/groups are capable of doing these attacks. They do it by developing fake applications that impersonate the company's own applications by using the same or a similar package name and distributing them through alternative app distribution vectors. This is a problem because impersonation allows fake applications to inherit a company's application's good reputation and brand recognition. In the worst case, it could even harm the end-user while tarnishing the company's reputation. For instance, an attacker can use the Facebook brand to direct users to their fake and rogue application. This allows them to potentially do all sorts of attacks, such as collecting the user's information for a phishing attack or having them click a link that distributes malware. To prevent app impersonation, we developed ScrAPK, a web scraping tool built on top of Scrapy that collects potentially fake and rogue Android applications from alternative app distribution vectors based on the metadata of the company's original application.

The rest of the paper is outlined as follows: In Section 2 we outlined the background of the technologies we use, In Section 3 we outline our approach in building ScrAPK and our data collection mechanism. In Section 4 we outlined the findings by collecting the APK impersonators of three popular applications: Facebook, YouTube, and Spotify. In Section 5, we went over the limitations of our tools and potential future work. We concluded the paper in Section 6.

II. Background

In this section, we will go over some of the backgrounds towards the framework and technologies we use and our decision towards utilizing them.

A. Scrapy
We did not want to build the web crawler from scratch. Therefore, we researched a few open-source scraping libraries/frameworks to build ScrAPK on top of. We came across a few scraping libraries/frameworks such as Scrapy, Beautiful Soup, and Selenium. We ended up choosing Scrapy, an open-source and collaborative framework for extracting the data you need from websites in an extensible way [2]. It uses what they refer to as spiders to scrape and crawl the web. Spiders are also capable of downloading HTML and parsing and processing its data.

We chose Scrapy because it is very well-structured, which allowed us to have better flexibility and adaptability when developing our specific use case. Scrapy also comes with a larger ecosystem of developers contributing to projects on Github and support on StackOverflow than the other two libraries. A con to picking Scrapy is that it has a slightly higher learning curve than the other two.

B. XPath
XPath stands for XML Path Language. It uses a “path-like” syntax to identify and navigate nodes in an XML document. This is an essential concept to our development because the XPath allows us to learn the structure of the DOM and locate the APK data that we want to extract from the alternative app distributor.

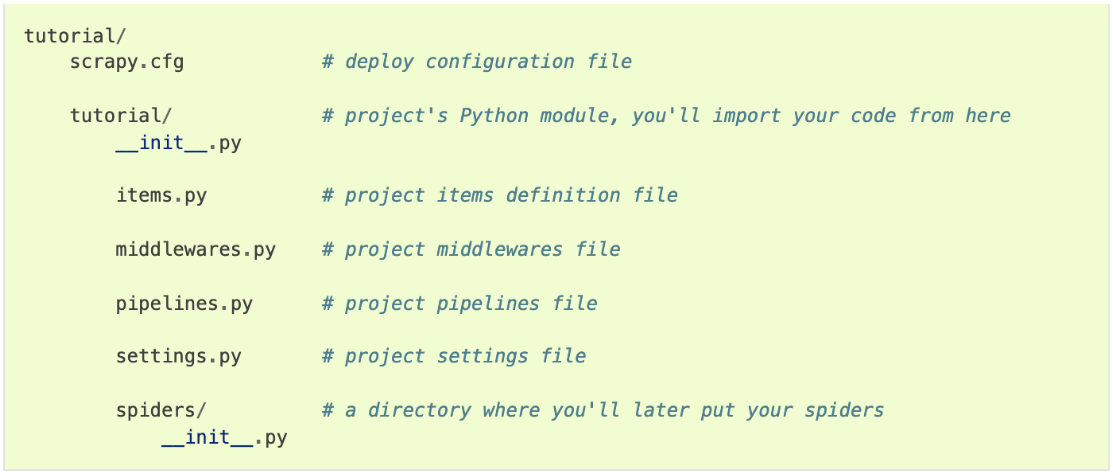
Absolute XPath:
The absolute XPath has the complete path from the root node of the XML file to the node that we identify. An absolute XPath starts with the / symbol. One drawback with the absolute XPath is that if there is any change in attributes beginning from the root to the element, our absolute XPath will become invalid. An example of an Absolute XPath to a title of an app in the Google Play store looks like: /html/body/div/div/div[2]/div/main/article/header/h1.

Relative XPath:
The relative XPath starts by referring to the element that we want to identify and not from the root node. A relative XPath starts with the // symbol. It is mainly used for automation since the relative XPath is not impacted even if an element is removed or added in the DOM. The relative XPath should be identical for every app within the same distributor. An example of a Relative XPath to a title of an app in the Google Play store looks like: //h1[contains(@itemprop, "name")]/span.

III. Approach
This section will go over the development we did, the alternative app distributor we explored, and our collection mechanism.

A. Development and ScrAPK workflow

Once we installed Scrapy, we had to set up a new Scrapy project. We were then given a starter file containing the files as seen in the image below [2]:



```
tutorial/
    scrapy.cfg            # deploy configuration file

    tutorial/             # project's Python module, you'll import your code from here
        __init__.py

        items.py          # project items definition file

        middlewares.py    # project middlewares file

        pipelines.py      # project pipelines file

        settings.py       # project settings file

        spiders/          # a directory where you'll later put your spiders
            __init__.py
```

To build ScrAPK on top of this framework, we had to utilize and edit the files: items.py, pipelines.py, settings.py, and spiders. We also initialized a SQLite database to store our scraped APK data.

Items.py is where we define the Item object for our scraped APK. Therefore, to build ScrAPK, we had to define an Item object that consists of data representing an APK. This Item object contains the distributor, URL, title, path to the title, developer, and the name of the package.

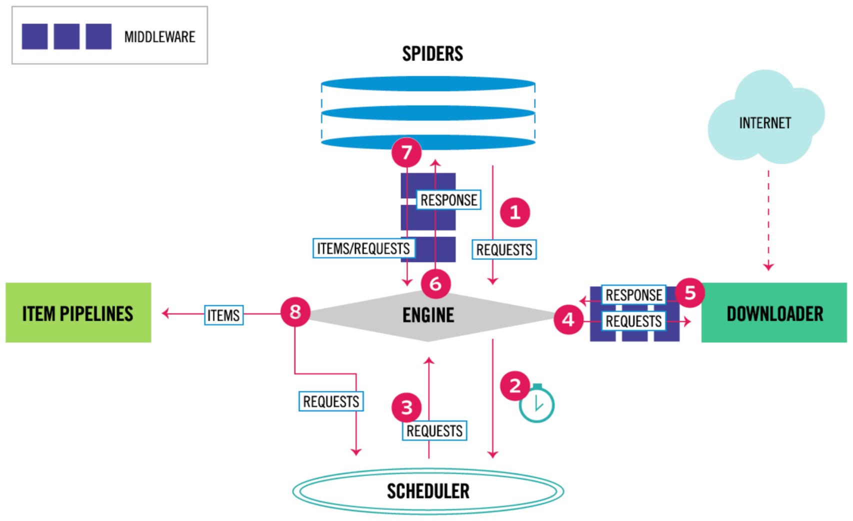
Pipelines.py is the file that handles filtering and saving the scraped item to the database. To build ScrAPK, we had to develop two pipelines. The first is a duplicate handler pipeline where it is able to detect if there are any incoming duplicates based on the URL of the scraped APK. If there is, we will drop the APK. The second is a save data pipeline, whereafter it goes through our filtering process, it gets reformatted to our model and is saved to our SQLite database.

Settings.py acts as a scheduler that determines which pipeline takes precedence over another. In our case, because we wanted to filter the data before saving it, we had the duplicate handler pipeline come before the save data pipeline.

Lastly, we created a new file called spider.py in the spiders folder. This is where we define our scraping logic and collection mechanism. However, before we run spider.py we will have to first inspect the DOM structure of a given alternative app distributor on the web. We do this to manually generate a relative XPath to the attribute of an APK that we defined in our model. Now, the spider makes a GET request to the provided root URLs. If the request returns a 200

HTTP Response object (which represents a successful response), this gives the spider permission to start extracting the data from the webpage. Now that the spider has permission, it will use the generated relative XPath along with a keyword (which represents the APKs name in our case) to be passed into the response.xpath function to start extracting the data. Once the data is extracted, it will be loaded into an Item object and sent to the pipeline for data processing. In addition to this scraping process, our spider also adds other URLs that contain a given keyword to a "to_be_visited URL" list, so it continues crawling those URLs afterward. After we are done with the scraping process, the visited URL will be added to a "visited URL" list to let our spider know not to visit the same URL twice.

The high-level project architecture and workflow can be viewed through the diagram below [3]:



B. Alternative app distributor

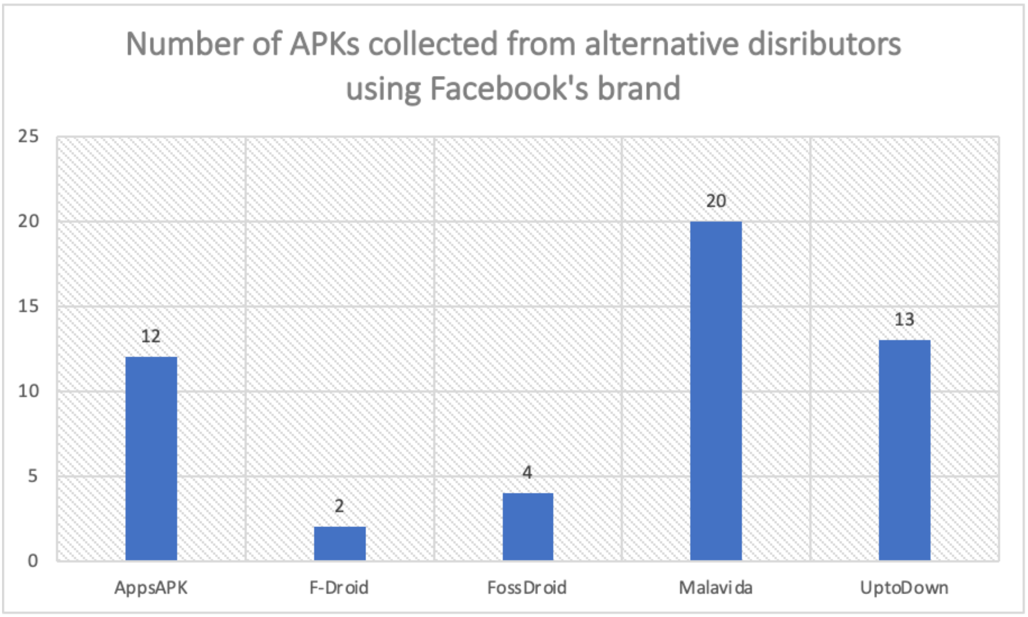
To utilize ScrAPK, we had to compile a list of alternative app distributors to explore. After doing a brief investigation, we found a list of 25 alternative app distributors on the web [4]. We then ran ScrAPK on all 25 alternative app distributors. However, due to limitations discussed in section 4, we were only successful in analyzing the DOM and scraping for potentially fake and rogue Android applications from five distributors. These five distributors are AppsAPK through appsapk.com, F-Droid through f-droid.org, FossDroid through fossdroid.com, Malavida malavida.com, and UpToDown through uptodown.com.

IV. Findings

This section will explore the findings collected by utilizing ScrAPK to scrape across the five alternative app distributors. In particular, we were focusing on collecting potentially fake and rogue apps that are impersonating the brand of Facebook, Spotify, and YouTube. ScrAPK is able to scrape for any application brand. However, we chose to explore Facebook, YouTube, and Spotify for this project because we believe these types of household consumer social apps to be the biggest targets for malicious attackers. In the future, we may explore more consumer social apps and also other types of applications in other various industries.

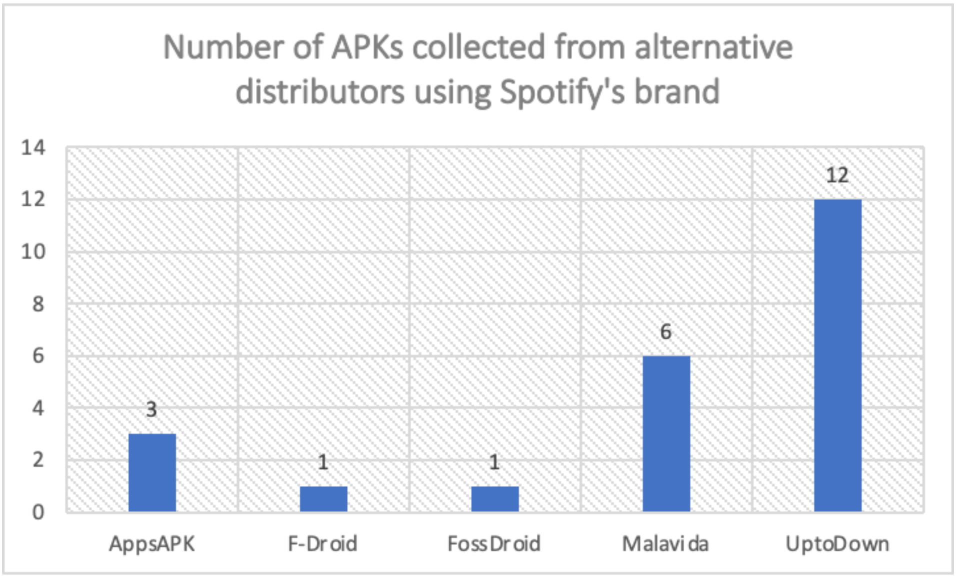
A. Facebook

We used the keyword ' facebook' to extract and collect the potentially fake or malicious apps using Facebook's brand from the five app distributors. Our results show that ScrAPK was able to collect a total of 51 unique apps from the five distributors. From the 51, 12 came from AppsAPK, two came from F-Droid, four came from FossDroid, 20 came from Malavida, and 13 came from UptoDown. A visualization of the distribution can be seen below.



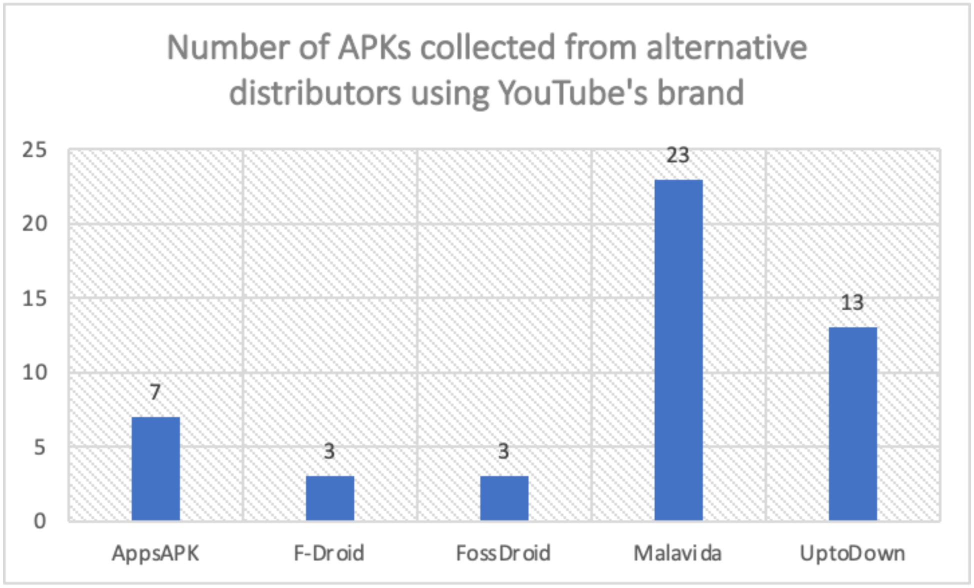
B. Spotify

We repeated the steps from part A but used the keyword ‘spotify’. Our results show that ScrAPK was able to collect a total of 23 unique apps from the five distributors. From the 23, three came from AppsAPK, one came from F-Droid, one came from FossDroid, six came from Malavida, and 12 came from UptoDown. A visualization of the distribution can be seen below.



C. YouTube

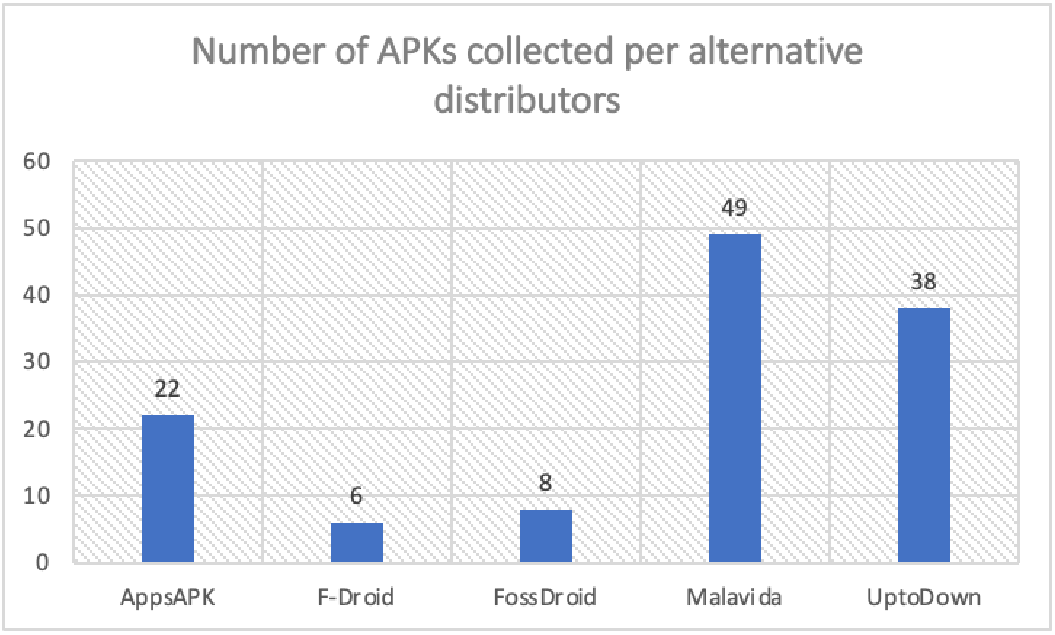
We repeated the steps from parts A and B but used the keyword 'youtube'. Our results show that ScrAPK was able to collect a total of 49 unique apps from the five distributors. From the 49, seven came from AppsAPK, three came from F-Droid, three came from FossDroid, 23 came from Malavida, and 13 came from UptoDown. A visualization of the distribution can be seen below.

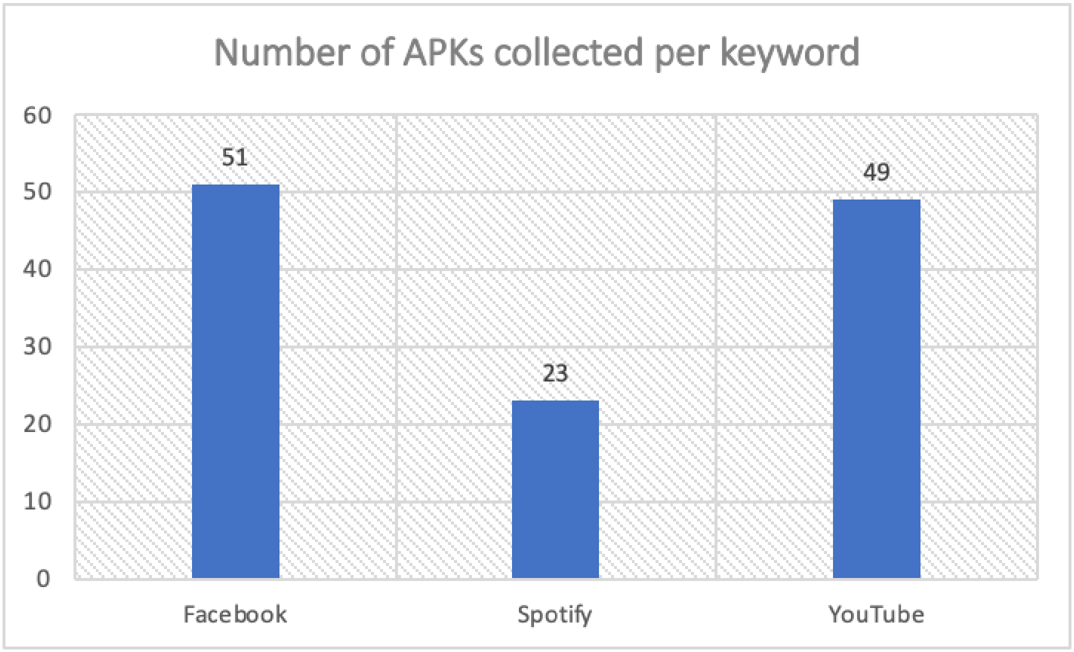


D. Cumulation

As a whole ScrAPK collected a total of 123 unique apps from five distributors using three keywords. The distributor Malavida had the most APKs extracted while F-Droid had the least.

Based on the three different keywords, the ‘facebook’ keyword led to the most APKs collected, with ‘youtube’ coming in second and ‘spotify’ coming in last. The entire raw data exported from our SQLite database including the specific app title, developer name, and URL can be found on the GitHub link in section 7. Visualization showcasing these insights can be viewed below.





V. Limitations and Future Work

This section will go over some of the limitations in our project and future works that combat these limitations.

A. Dark web

The dark web is a part of the internet that is not indexed and therefore not accessible by search engines. This makes it a prime hotbed for criminal activity. According to research in 2016 done by Daniel Moore and Thomas Rid of King's College [5], they found that 57% of 2723 live dark websites over a five-week period hosted illicit material. This gives us a hypothesis that many more fake and rogue mobile applications live on the dark web. However, our scraping tool is limited to only traversing the DOM of a public, searchable website given its root URL. Therefore, this constrains our ability to scrape any dark web content. We will leave the development of a more sophisticated and feature-rich tool that has the ability to traverse through the dark web as future work.

B. Sign-up blockage

A few of the alternative app distributors we encountered require signing up and logging in before unlocking access to its contents. This sign-up page blocks the DOM containing the keyword our scraping tool is searching for. Therefore, it prevents our tool from automatically traversing through the DOM and collecting the path of the potentially fake and rogue mobile application. In the future, we hope to create a form-filling bot that could automatically sign-up/log in on our behalf when the scraping tool encounters a similar sign-up page so that it does not require any human intervention.

C. Inconsistent DOM structure within alternative app distributor

Industry-standard front-end development has it that every data of the same type lives within the same HTML element. For example, we expect every application title within the alternative app distributor to live in the same HTML tag <h1 class=" app_AppName">. This is the case for the Google Play Store. However, not every alternative app distributor we have encountered has their front-end HTML structured this way. The worst we have seen is each application living in various tags with different class names. The inconsistent tags and class names make it incredibly hard to search and traverse through the DOM. To solve this, we were required to manually inspect the inconsistent DOM structure, identify any patterns that live within the DOM, and adjust our code and path-finding logic so that it is able to work through the inconsistency. When we could not identify any patterns, we had to resort to hard coding the path.

D. Language barrier

According to w3Techs [6], a web technology usage survey platform, 36.4% of the top 10 million websites are not in English. This statistic gives us the conviction that there are many more alternative app distributors containing fake and rogue applications in a different language. This is a limitation towards our scraping tool as it is currently only able to search for keywords in English. We leave the development of a more robust scraping algorithm that can search for keywords in different languages as future work.

E. Dynamic and static analysis of fake applications

We currently assume that the fake applications collected through the alternative app distributors to be potentially rogue or malicious. However, we have no way of actually determining if it is actually malicious and how it is malicious. Therefore, as part of our future work, we plan on including dynamic and static analysis to monitor the behavior of these applications to help make the determination. For dynamic analysis, we can use TaintDroid, an Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones [7]. By using TaintDroid we can further understand if any of these applications exhibit suspicious handling of sensitive data. For static analysis, we can use FlowDroid, a highly precise static taint analysis tool for Android applications [8]. By using FlowDroid, we can quickly detect data leaks found in any of these applications.

F. Scraping more apps

As mentioned in Section 4, due to time and development limitations, we were only able to scrape for three brands across five distributors. In the future, we plan to explore more consumer social apps and also other types of applications in other various industries.

VI. Conclusion

Big technology companies developing mobile applications with millions of users make it a prime target for digital brand impersonation. Malicious developers can create fake applications that impersonate a different company or brand by using the same package name and distributing them through an alternative app distribution vector for their own personal gains. Hence, we presented the design and implementation of ScrAPK, a web scraping tool built on top that collects potentially fake and rogue Android applications from alternative app distribution vectors. We ran ScrAPK across five alternative app distributors to collect applications impersonating Facebook, Spotify, and YouTube and found a total of 123 unique apps. In the future, we plan on improving ScrAPK to handle different automation and expand its coverage to the dark web and websites of a different language. We also plan to do static and dynamic analysis on collected apps to determine if it is malicious and how it is malicious.

VII. Artifact

https://github.com/Thang-sudo/ScrAPK

VIII. References

[1] How Did That Get Into My Phone? Unwanted App Distribution on Android Devices https://drew.davidson.cool/eecs700/kotzias_2021.pdf

[2] Scrapy https://scrapy.org/

[3] Scrapy architecture https://docs.scrapy.org/en/latest/_images/scrapy_architecture_02.png

[4] 25+ Similar app stores https://alternativeto.net/software/aptoide/

[5] Cryptopolitik and the Darknet https://www.tandfonline.com/doi/full/10.1080/00396338.2016.1142085

[6] Usage statistics of content languages for websites https://w3techs.com/technologies/overview/content_language#:~:text=English%20is%20used%20by%2063.5%25%20of%20all%20the,used%20by%20less%20than%200.1%25%20of%20the%20websites

[7] TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones https://www.usenix.org/legacy/event/osdi10/tech/full_papers/Enck.pdf

[8] FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps https://www.bodden.de/pubs/far+14flowdroid.pdf