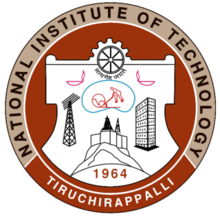
**National Institute of Technology, Tiruchirappalli**

****

**Department of Computer Applications**

**Information Security**

**Project Report**

Submitted to: Submitted by:

Dr. Janet Barnabas Akriti Upadhyay

205120007

MCA – 2nd Year

**RiskAnalyzer: Machine Learning-based Risk**

**Analysis on Android**

**Problem Definition:**

How can one verify the reliability of an Android Application, ensuring that it is not a malware that will harm their Mobile device?

**Introduction:**

Android-enabled smartphones remain a sensitive target for malware that aim at exploiting its diffusion to reach a high number of potential victims. Since users have access to a high number of apps through public markets and external web sites, they need reliable tools to rate the trustworthiness of apps they are going to install. App rating is empirically calculated according to different risk analysis techniques. Currently, most of them calculate a risk index value (hereafter, RIV) through probabilistic methods applied to the set of permissions required by the app. I argue that such approaches suffer from intrinsic limitations in terms of both methodology and setup. To prove this, I apply some optimizations to existing techniques at the state of the art, and I evaluate them through an extensive empirical assessment on a dataset made by 112.425 apps and 6.707 malware samples. Then, I propose a novel approach based on machine learning techniques that I implemented in an open-source tool, i.e., RiskAnalyzer (Risk Index for Android). Finally, I evaluate the performance of RiskAnalyzer on the same dataset, thereby proving that the proposed methodology outperforms probabilistic approaches.

**Proposed Solution:**

Risk analysis on Android is aimed at providing metrics to users for evaluating the trustworthiness of the apps they are going to install. Most of current proposals calculate a risk value according to the permissions required by the app through probabilistic functions that often provide unreliable risk values. To overcome such limitations, I have proposed RiskAnalyzer, a tool for risk analysis of Android apps based on machine learning techniques. RiskAnalyzer outperforms probabilistic methods in terms of precision and reliability.

RiskAnalyzer (Risk Index for Android) is a tool for quantitative risk analysis of Android applications written in Java (used to check the permissions of the apps) and Python (used to compute a risk value based on apps' permissions). The tool uses classification techniques through scikit-learn, a machine learning library for Python, in order to generate a numeric risk value between 0 and 100 for a given app. In particular, the following classifiers of scikit-learn are used in RiskAnalyzer:

* Support Vector Machines (SVM)
* Multinomial Naive Bayes (MNB)
* Gradient Boosting (GB)
* Logistic Regression (LR)

Unlike other tools, RiskAnalyzer does not take into consideration only the permissions declared into the app manifest, but carries out reverse engineering on the apps to retrieve the bytecode and then infers (through static analysis) which permissions are actually used and which not, extracting in this way 4 sets of permissions for every analyzed app:

* Declared permissions - extracted from the app manifest
* Exploited permissions - declared and actually used in the bytecode
* Ghost permissions - not declared but with usages in the bytecode
* Useless permissions - declared but never used in the bytecode

From the above sets of permissions (and considering only the official list of Android permissions), feature vectors (made by 0s and 1s) are built and given to the classifiers, which then compute a risk value.

I argue that the intrinsic limitations of probabilistic methods applied to Android Permissions (hereafter, APs) can be overcome by machine learning techniques able to build up more reliable RIVs.

Machine learning techniques are used for classifying elements, i.e., given a set of classes, they evaluate each element and assign a class to it. Therefore, they are particularly suitable for binary classification of malware. However, some techniques also provide a probability value related to the prediction. I leverage machine learning techniques to classify apps into two classes, i.e., malware and non-malware, and I use the classification probability to build up a RIV. For my purpose, I adopt the scikit-learn library, that implements a set of machine learning techniques and provides a probability function for some of them.

Machine learning techniques require feature vectors to compare and classify elements. In my context, elements are apps, and features are APs. I define feature vectors as follows: given AP Set the set of APs, for each app A, I define four feature vectors FVSA , with S ∈ {DAPA , EAPA , GAPA , UAPA}. Each FV is a binary vector of cardinality |APSet|, where FVSA[i] = 1 if pi ∈ S, and FVSA [i] = 0 otherwise. I adopt a supervised learning approach. Supervised learning requires classifiers to be trained on a training set before being applied to classify new elements. I train a set of supervised classifiers on a subset of the dataset and then I use them to classify the remaining APKs.

**Code:**

* app.py

*#!/usr/bin/env python3*

import hashlweb

import os

import subprocess

import time

from flask import Flask

from flask import jsonify

from flask import make\_response

from flask import render\_template

from flask import request

from sqlalchemy import cast

from sqlalchemy.sql import text

from Irkzeug.exceptions import BadRequest, UnprocessableEntity

from Irkzeug.utils import secure\_filename

from RiskAnalyzer import RiskAnalyzer

from model import db, Apk

ALLOID\_EXTENSIONS = {"apk", "zip"}

def create\_app():

app = Flask(\_\_name\_\_)

app.config["UPLOAD\_DIR"] = os.path.join(

os.path.dirname(os.path.realpath(\_\_file\_\_)), "upload"

)

app.config["MAX\_CONTENT\_LENGTH"] = 100 \* 1024 \* 1024

app.config["DB\_DIRECTORY"] = os.path.join(

os.path.dirname(os.path.realpath(\_\_file\_\_)), "database"

)

app.config["DB\_7Z\_PATH"] = os.path.join(

os.path.dirname(os.path.realpath(\_\_file\_\_)), "database", "permission\_db.7z"

)

app.config["DB\_PATH"] = os.path.join(

os.path.dirname(os.path.realpath(\_\_file\_\_)), "database", "permission\_db.db"

)

app.config["SQLALCHEMY\_DATABASE\_URI"] = "sqlite:///" + app.config["DB\_PATH"]

app.config["SQLALCHEMY\_TRACK\_MODIFICATIONS"] = False

*# Establish the database connection.*

db.init\_app(app)

*# Create the upload directory (if not already existing).*

if not os.path.exists(app.config["UPLOAD\_DIR"]):

os.makedirs(app.config["UPLOAD\_DIR"])

*# Check if the database file is already extracted from the archive,*

*# otherwise extract it.*

if not os.path.isfile(app.config["DB\_PATH"]):

instruction = '7z x "{0}" -o"{1}"'.format(

app.config["DB\_7Z\_PATH"], app.config["DB\_DIRECTORY"]

)

subprocess.run(instruction, shell=True)

return app

application = create\_app()

def check\_if\_valid\_file\_name(file\_name):

return (

"." in file\_name and file\_name.rsplit(".", 1)[1].loIr() in ALLOID\_EXTENSIONS

)

@application.after\_request

def add\_cache\_header(response):

response.headers[

"Cache-Control"

] = "public, max-age=0, no-cache, no-store, must-revalidate"

response.headers["Pragma"] = "no-cache"

response.headers["Expires"] = "0"

return response

@application.errorhandler(400)

@application.errorhandler(422)

@application.errorhandler(500)

def application\_error(error):

return make\_response(jsonify(str(error)), error.code)

@application.route("/", methods=["GET"], strict\_slashes=False)

def home():

return render\_template("index.html")

@application.route("/results", methods=["GET"], strict\_slashes=False)

def results():

return render\_template("results.html")

@application.route("/upload", methods=["POST"], strict\_slashes=False)

def upload\_apk():

*# The POST request must contain a valid file.*

if "file" not in request.files:

raise BadRequest("No file uploaded")

file = request.files["file"]

if not file.filename.strip():

raise BadRequest("No file uploaded")

if file and check\_if\_valid\_file\_name(file.filename):

filename = secure\_filename(file.filename)

file\_path = os.path.join(

application.config["UPLOAD\_DIR"],

"{0}\_{1}".format(time.strftime("%H-%M-%S\_%d-%m-%Y"), filename),

)

file.save(file\_path)

rid = RiskAnalyzer()

permissions = rid.get\_permission\_json(file\_path)

try:

response = {

"name": filename,

"md5": md5sum(file\_path),

"risk": round(

rid.calculate\_risk(rid.get\_feature\_vector\_from\_json(permissions)),

3,

),

"permissions": [

val

for val in list(

map(

lambda x: {"cat": "Declared", "name": x},

permissions["declared"],

)

)

+ list(

map(

lambda x: {"cat": "Required and Used", "name": x},

permissions["requiredAndUsed"],

)

)

+ list(

map(

lambda x: {"cat": "Required but Not Used", "name": x},

permissions["requiredButNotUsed"],

)

)

+ list(

map(

lambda x: {"cat": "Not Required but Used", "name": x},

permissions["notRequiredButUsed"],

)

)

],

}

return make\_response(jsonify(response))

except Exception:

raise BadRequest("The uploaded file is not valid")

else:

raise UnprocessableEntity("The uploaded file is not valid")

@application.route("/apks", methods=["GET"], strict\_slashes=False)

def get\_apks():

query = Apk.query

if request.args.get("sort") and request.args.get("sort\_dir"):

query = query.order\_by(

text(

"{0} {1}".format(request.args.get("sort"), request.args.get("sort\_dir"))

)

)

if request.args.get("namefil"):

fil = request.args.get("namefil").replace("%", "\\%").replace("\_", "\\\_")

query = query.filter(Apk.name.ilike("%{0}%".format(fil), "\\"))

if request.args.get("md5fil"):

fil = request.args.get("md5fil").replace("%", "\\%").replace("\_", "\\\_")

query = query.filter(Apk.md5.ilike("%{0}%".format(fil), "\\"))

if request.args.get("riskfil"):

fil = request.args.get("riskfil").replace("%", "\\%").replace("\_", "\\\_")

query = query.filter(cast(Apk.risk, db.String).ilike("%{0}%".format(fil), "\\"))

pag = query.paginate()

item\_list = []

for item in pag.items:

item\_list.append({"name": item.name, "md5": item.md5, "risk": item.risk})

response = {"current\_page": pag.page, "last\_page": pag.pages, "data": item\_list}

return make\_response(jsonify(response))

@application.route("/details", methods=["GET", "POST"], strict\_slashes=False)

def get\_apk\_details():

if request.method == "GET":

try:

*# An exception will be thrown if the query string doesn't contain an md5.*

md5 = request.args["md5"]

apk = Apk.query.get(md5)

response = {

"name": apk.name,

"md5": apk.md5,

"risk": apk.risk,

"type": apk.type,

"source": apk.source,

"permissions": [

val

for val in list(

map(

lambda x: {"cat": "Declared", "name": x.name},

apk.declared\_permissions,

)

)

+ list(

map(

lambda x: {"cat": "Required and Used", "name": x.name},

apk.required\_and\_used\_permissions,

)

)

+ list(

map(

lambda x: {"cat": "Required but Not Used", "name": x.name},

apk.required\_but\_not\_used\_permissions,

)

)

+ list(

map(

lambda x: {"cat": "Not Required but Used", "name": x.name},

apk.not\_required\_but\_used\_permissions,

)

)

],

}

return make\_response(jsonify(response))

except Exception:

raise BadRequest("Unable to get details for the specified application")

if request.method == "POST":

response = {

"name": request.form["name"],

"md5": request.form["md5"],

"risk": request.form["risk"],

"permissions": request.form["permissions"],

}

return render\_template("details.html", apk=response)

def md5sum(file\_path, block\_size=65536):

md5\_hash = hashlweb.md5()

with open(file\_path, "rb") as filename:

for chunk in iter(lambda: filename.read(block\_size), b""):

md5\_hash.update(chunk)

return md5\_hash.hexdigest()

if \_\_name\_\_ == "\_\_main\_\_":

application.run(host="0.0.0.0", port=5000)

* model.py

*#!/usr/bin/env python3*

from flask\_sqlalchemy import SQLAlchemy

db = SQLAlchemy()

declared\_permissions = db.Table(

"declared\_permissions",

db.metadata,

db.Column("apk\_id", db.String(32), db.ForeignKey("apks.md5"), primary\_key=True),

db.Column(

"permission\_id", db.Integer, db.ForeignKey("permissions.id"), primary\_key=True

),

)

required\_and\_used\_permissions = db.Table(

"required\_and\_used\_permissions",

db.metadata,

db.Column("apk\_id", db.String(32), db.ForeignKey("apks.md5"), primary\_key=True),

db.Column(

"permission\_id", db.Integer, db.ForeignKey("permissions.id"), primary\_key=True

),

)

required\_but\_not\_used\_permissions = db.Table(

"required\_but\_not\_used\_permissions",

db.metadata,

db.Column("apk\_id", db.String(32), db.ForeignKey("apks.md5"), primary\_key=True),

db.Column(

"permission\_id", db.Integer, db.ForeignKey("permissions.id"), primary\_key=True

),

)

not\_required\_but\_used\_permissions = db.Table(

"not\_required\_but\_used\_permissions",

db.metadata,

db.Column("apk\_id", db.String(32), db.ForeignKey("apks.md5"), primary\_key=True),

db.Column(

"permission\_id", db.Integer, db.ForeignKey("permissions.id"), primary\_key=True

),

)

class Apk(db.Model):

\_\_tablename\_\_ = "apks"

md5 = db.Column(db.String(32), primary\_key=True)

type = db.Column(db.String(10), nullable=False)

source = db.Column(db.String(24), nullable=False)

name = db.Column(db.String(255), nullable=False)

risk = db.Column(db.Float, nullable=False)

declared\_permissions = db.relationship(

"Permission",

secondary=declared\_permissions,

backref=db.backref("app\_declaring", lazy="dynamic"),

)

required\_and\_used\_permissions = db.relationship(

"Permission",

secondary=required\_and\_used\_permissions,

backref=db.backref("app\_requiring\_and\_using", lazy="dynamic"),

)

required\_but\_not\_used\_permissions = db.relationship(

"Permission",

secondary=required\_but\_not\_used\_permissions,

backref=db.backref("app\_requiring\_but\_not\_using", lazy="dynamic"),

)

not\_required\_but\_used\_permissions = db.relationship(

"Permission",

secondary=not\_required\_but\_used\_permissions,

backref=db.backref("app\_not\_requiring\_but\_using", lazy="dynamic"),

)

def \_\_repr\_\_(self):

return '<Apk (md5="{0}", name="{1}", risk="{2}")>'.format(

self.md5, self.name, self.risk

)

class Permission(db.Model):

\_\_tablename\_\_ = "permissions"

id = db.Column(db.Integer, primary\_key=True)

name = db.Column(db.String(255), unique=True, nullable=False)

def \_\_repr\_\_(self):

return '<Permission (name="{0}")>'.format(self.name)

**Results:**

RiskAnalyzer has been developed in Python and implements the selected four classifiers. For each app A, RiskAnalyzer calculates the RIV on all four APs sets (i.e., DAPA , EAPA , GAPA, and UAPA), by combining the corresponding feature vectors in a unique one, i.e., FVAall = FVADAP || FVAEAP || FVAGAP || FVAUAP. The RIV is calculated as the average score value of all four classifiers.

To train each classifier in RiskAnalyzer, I applied the 10-fold cross validation on one of the three sets used to evaluate the classifiers. I also used the same set to empirically assess whether applying all four APs sets may improve the accuracy. To this aim, my tests returned the following average accuracy values: 92.93% for DAP, 88.36% for EAP, 79.12% for GAP, 91.09% for UAP, and 94.87% for all sets. Therefore, I chose to consider all sets.

**Advantages:**

RiskAnalyzer does not take into consideration only the permissions declared into the app manifest unlike other tools, but carries out reverse engineering on the apps to retrieve the bytecode and then infers (through static analysis) which permissions are actually used and which not, extracting in this way 4 sets of permissions for every analyzed app.

Hence, the intrinsic limitations of probabilistic methods applied to Android Permissions (hereafter, APs) can be overcome by machine learning techniques able to build up more reliable RIVs.

**Disadvantages:**

Performance of RiskAnalyzer: The performance of RiskAnalyzer has been evaluated on a general-purpose Laptop equipped with an Intel i5 GHz processor, and 8GB RAM. Performance of classifiers is evaluated in terms of average time and standard deviation, during the training and the testing phase. Using all sets decreases the average performance up to 240% during the training phase.

However, it is worth noticing that this phase is executed once at the beginning. Instead, the testing phase is very quick and lasts in a few millisecs both with one and all sets, thereby suggesting to adopt all four sets to obtain a higher accuracy.

**Novelty of the Solution:**

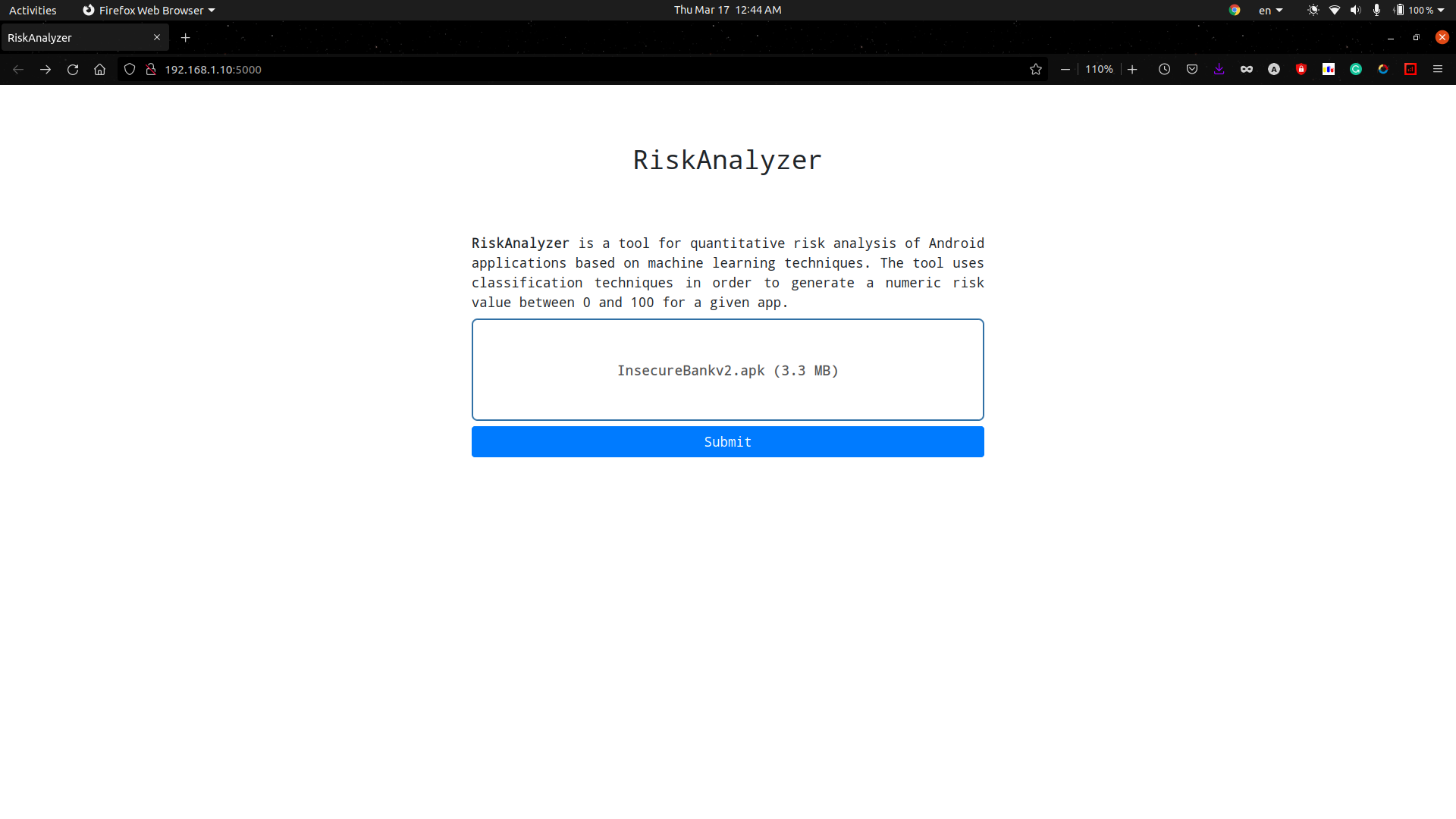
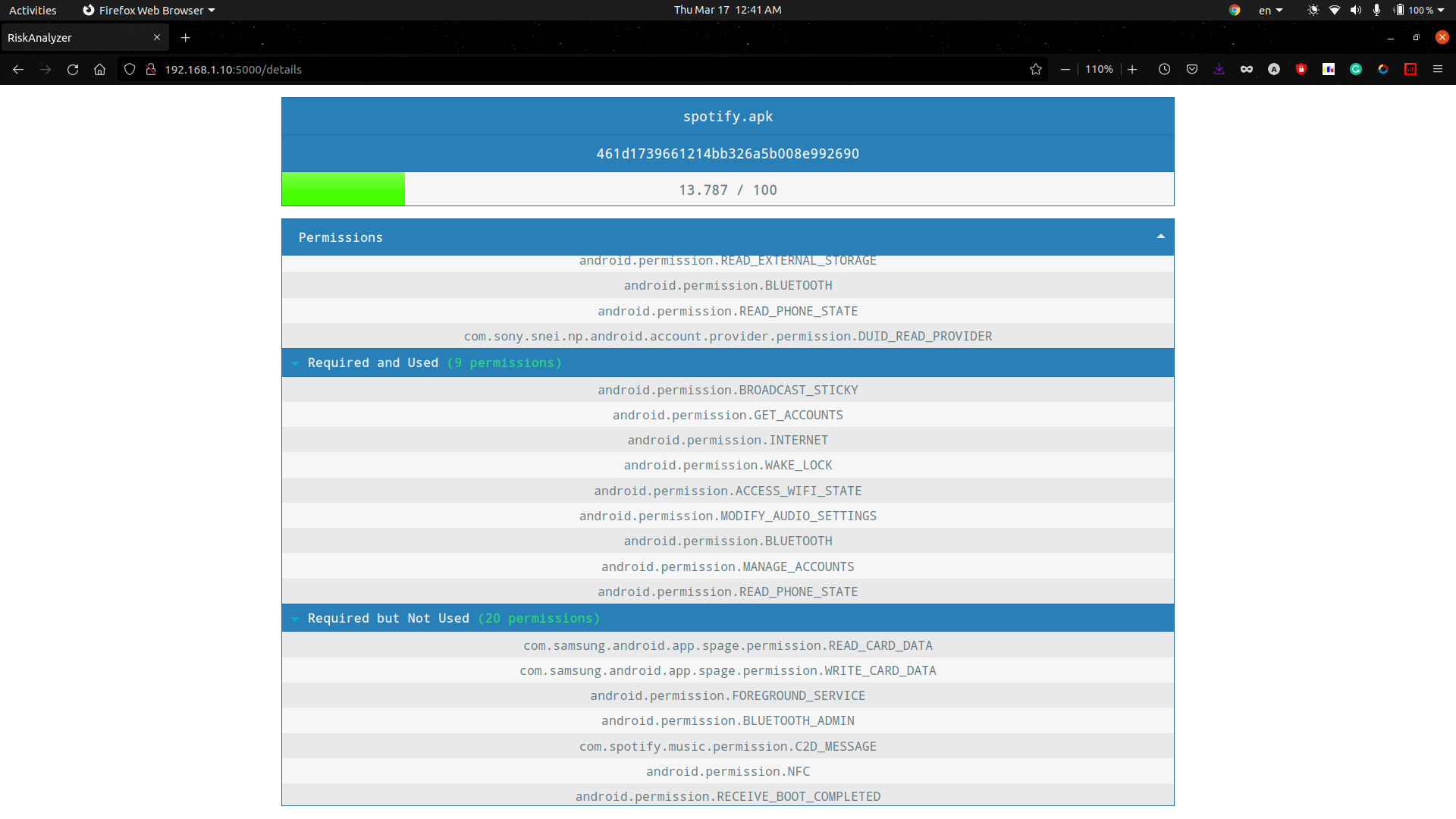
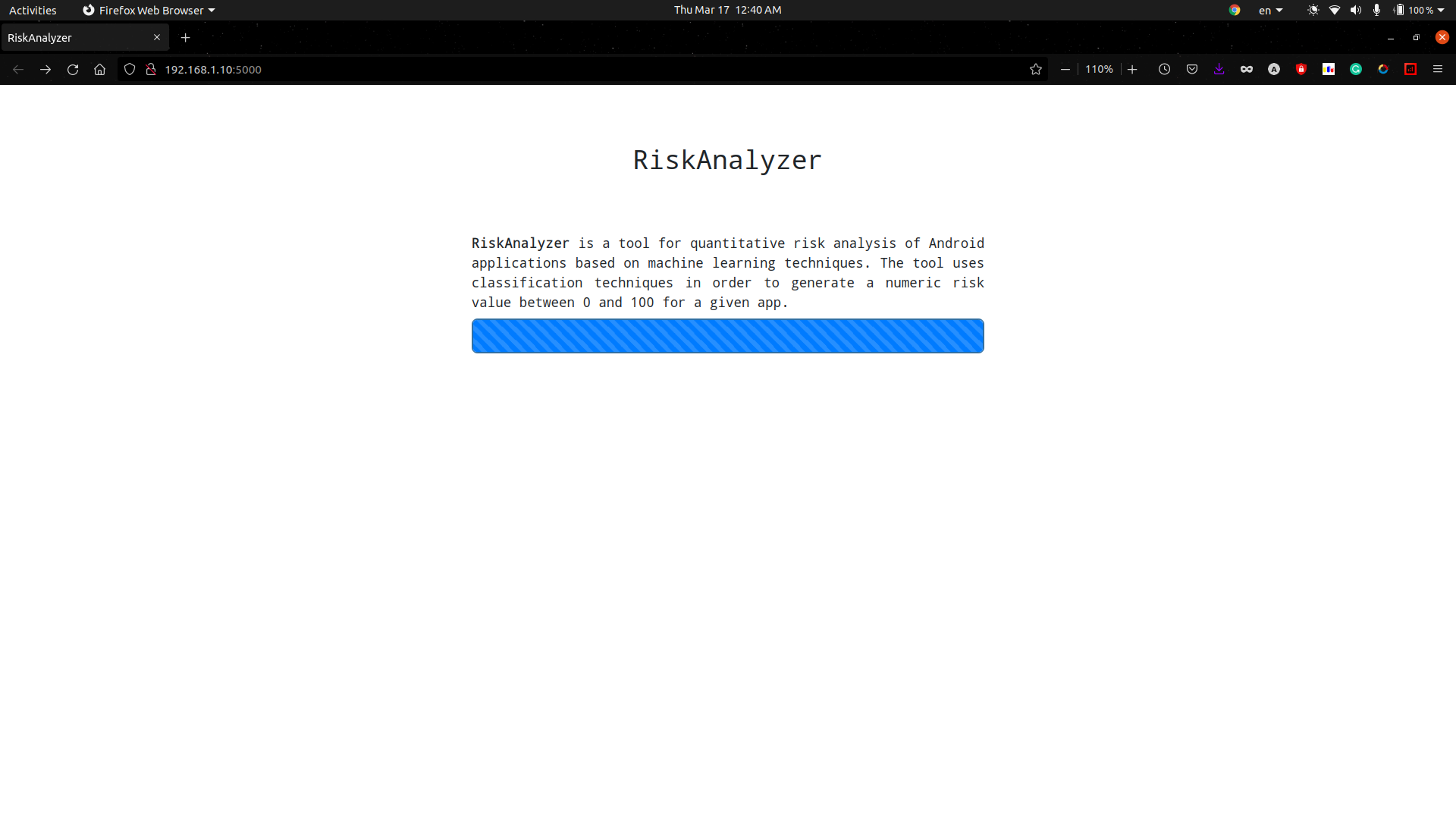
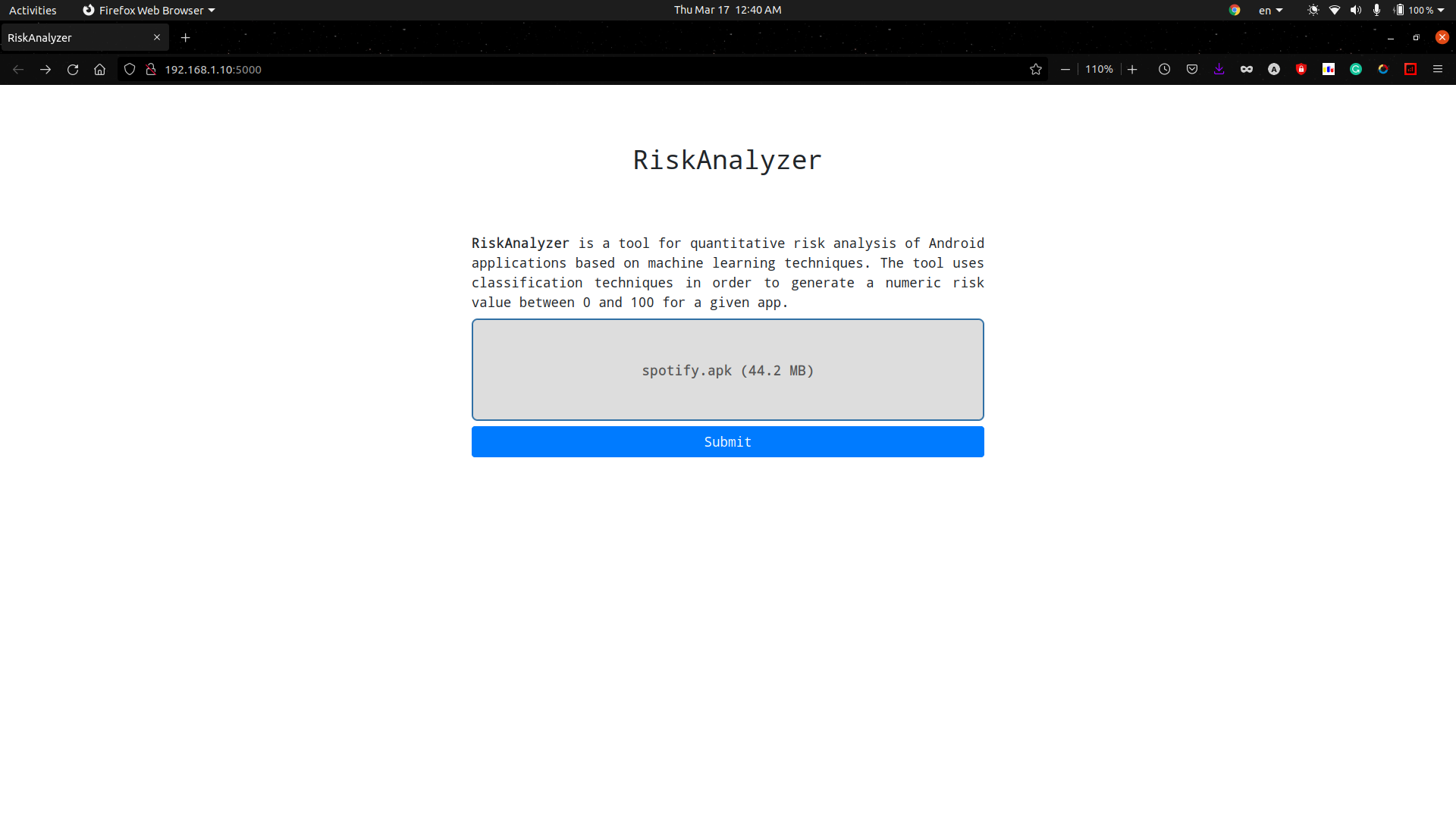
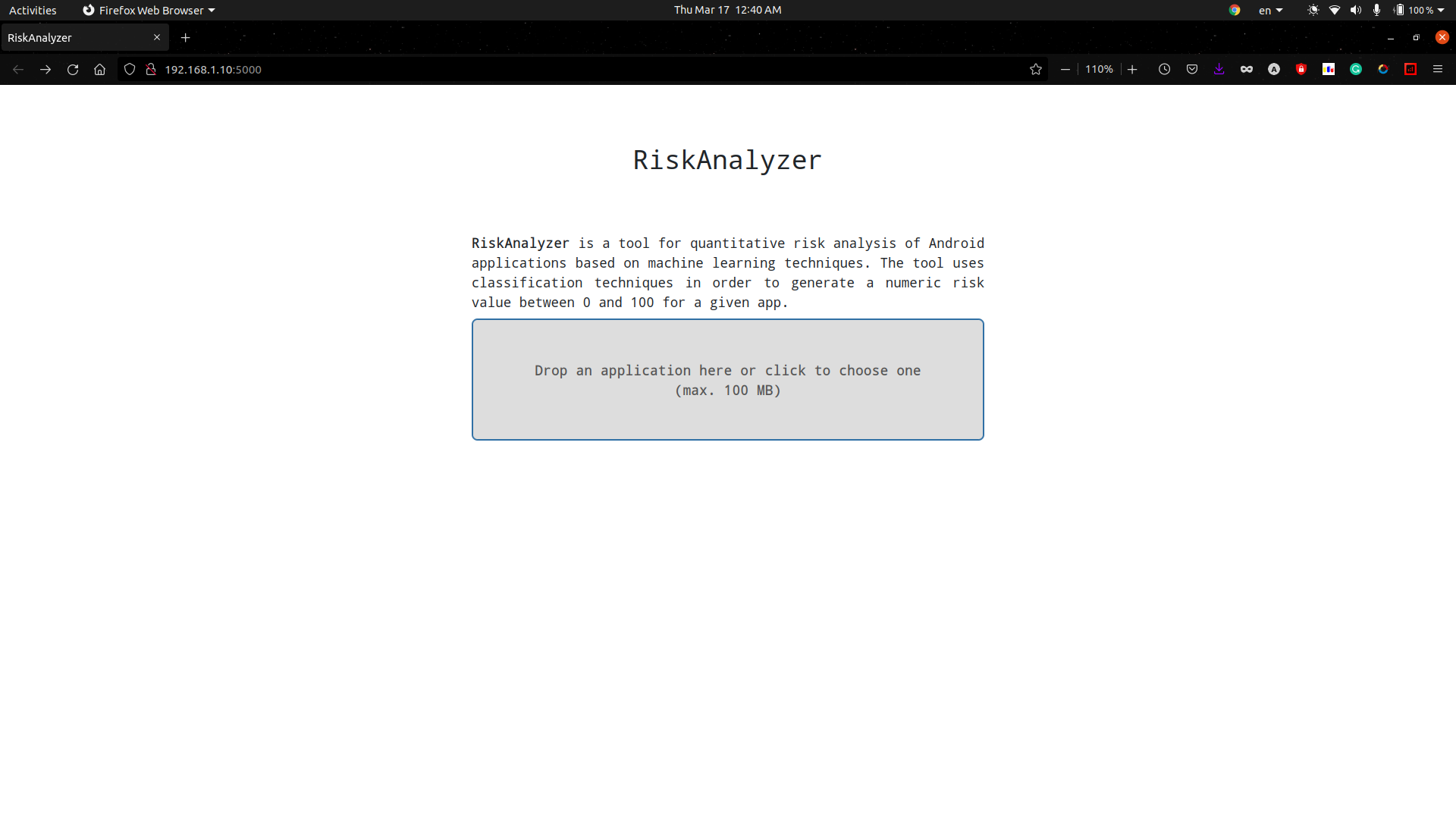
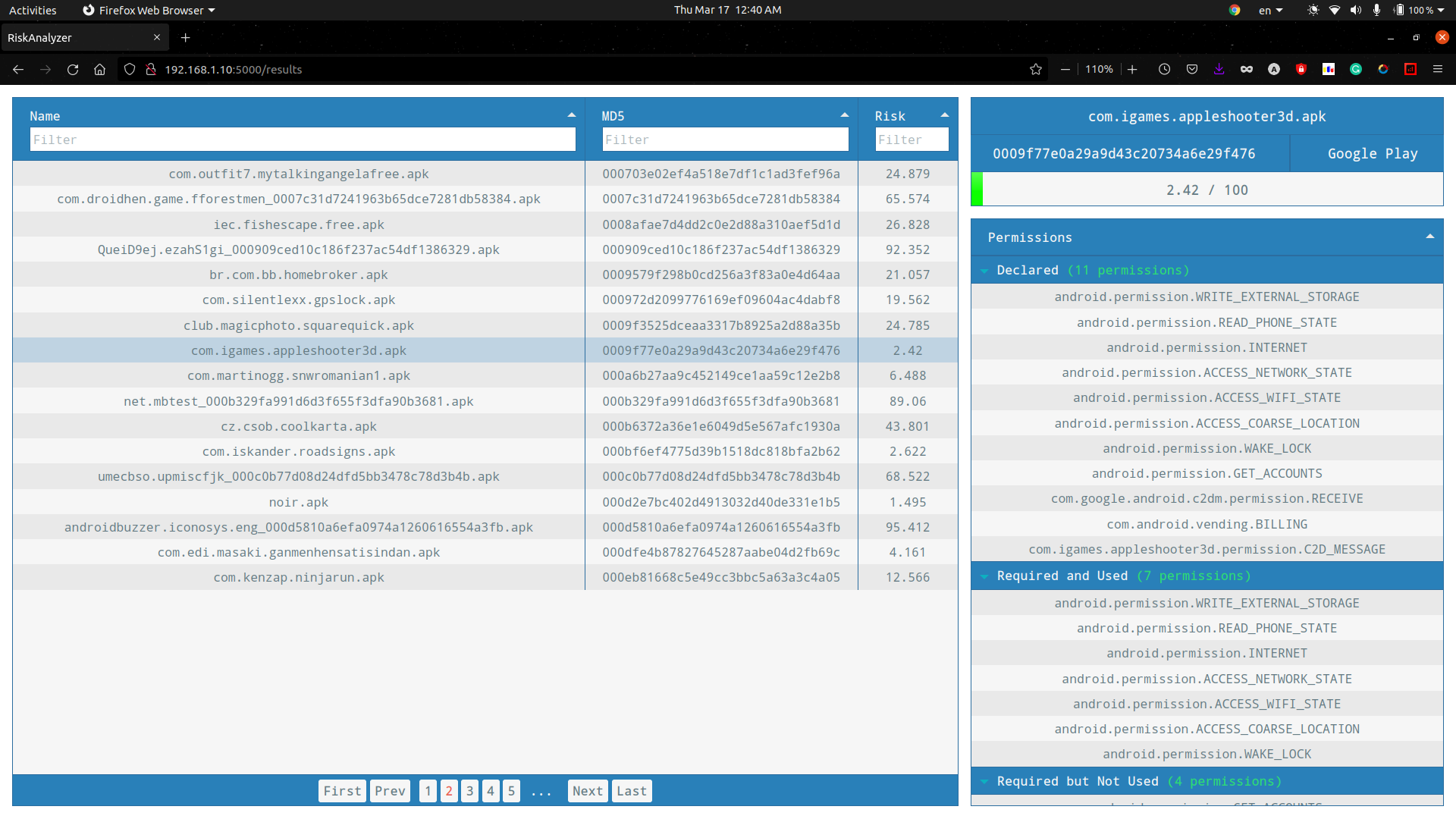
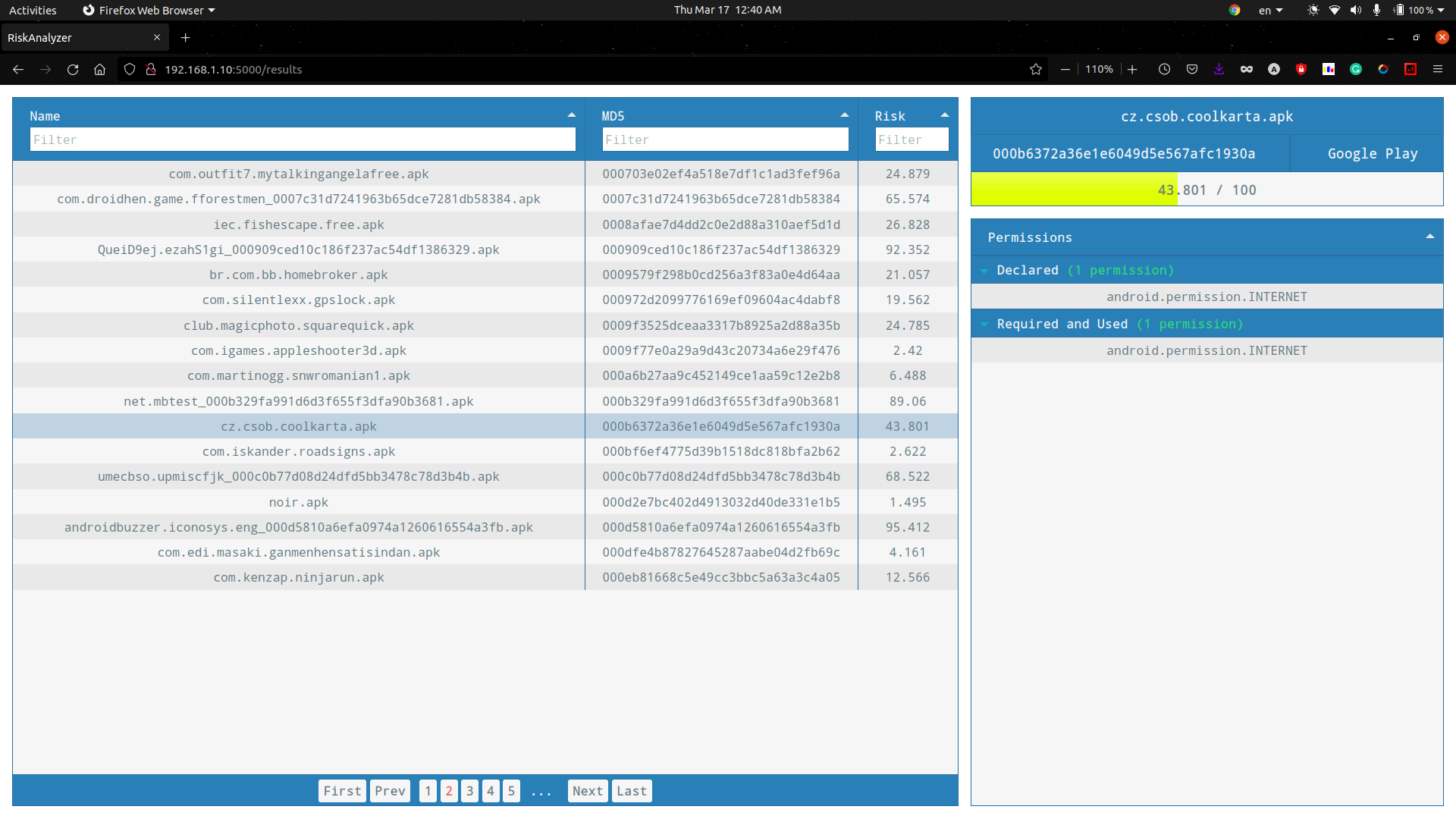
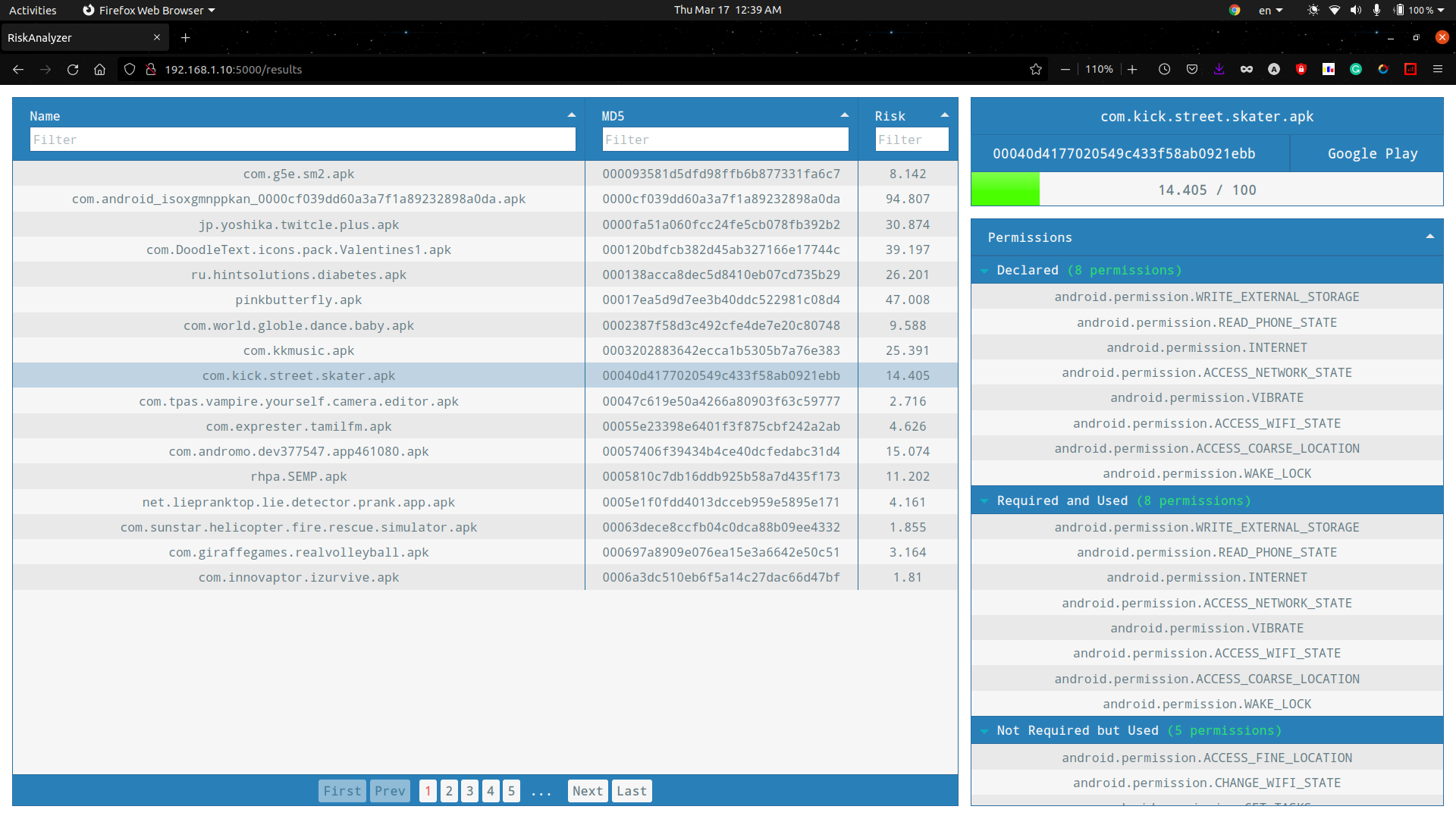
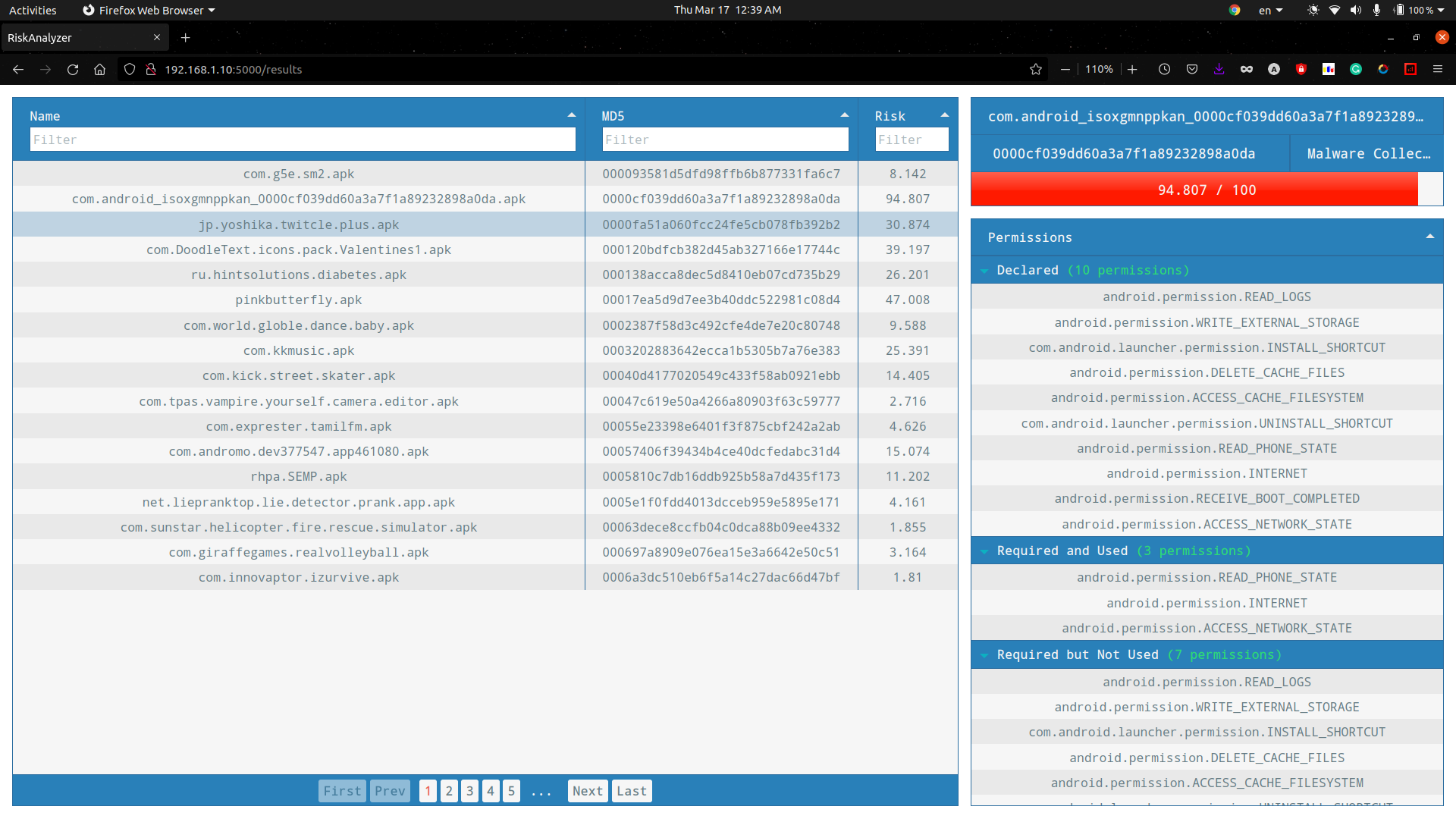
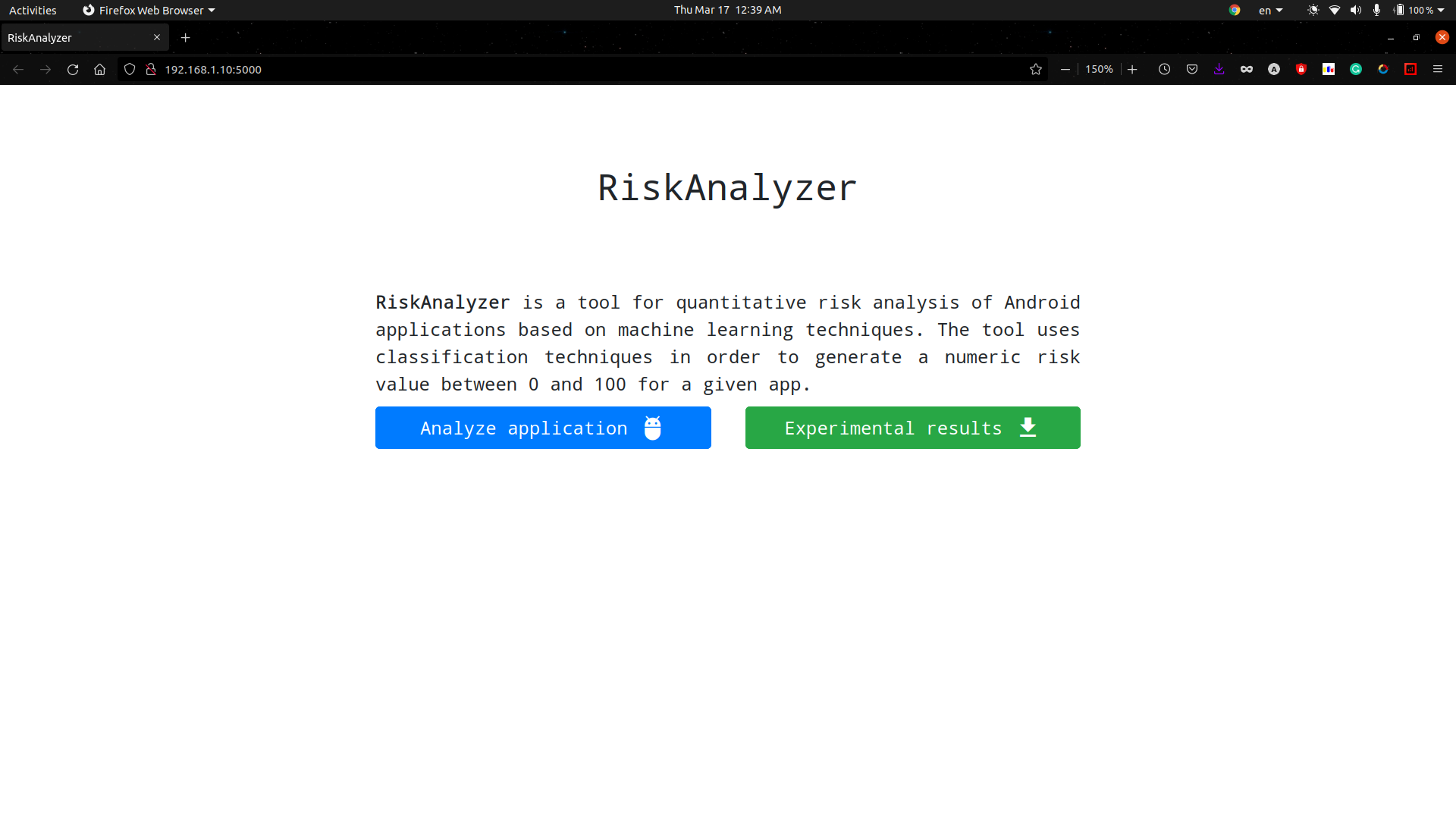
In this solution, I empirically assessed the reliability of probabilistic risk index approaches for Android apps, and I proposed a novel methodology based on machine learning aimed at overcoming the shortcomings of the probabilistic solutions. I implemented the methodology in a tool, RiskAnalyzer, that I empirically evaluated.

**Application:**

RiskAnalyzer provides a metric to users for evaluating the trustworthiness of the apps they are going to install. Android is still the most widespread mobile operating system in the world, as more than 300 million Android-enabled smartphones have been sold only in the third trimester of 2016. Android-enabled smartphones remain a sensitive target for malware that aim at exploiting its diffusion to reach a high number of potential victims. Since users have access to a high number of apps through public markets and external web sites, they need reliable tools to rate the trustworthiness of apps they are going to install.

**Screenshots of Results:**

Results through a web interface and calculating the risk of new applications (by uploading the .apk file):



**Comparison of Existing Solutions:**

The scientific literature related to risk analysis of Android apps is rather limited and mostly focused on APs, so I also take into account works regarding malware classification because I expect to see some relationships between malware and high-risk apps. Currently available proposals are probabilistic, i.e., the RIV indicates the probability that an app can be a malware, according to statistical analysis carried out on datasets containing both apps (that are expected to be mostly benign) and Ill-known malware samples. In an existing solution, authors propose a method for detecting risk signals according to the frequency of security-sensitive APs. The RIV is calculated according to Bayesian probabilistic models that compare the APs required by each app with those requested by other apps in the same category (that must be known a priori). Furthermore, authors define three properties that should be granted by any probabilistic function calculating a RIV for apps, namely, i) monotonicity (i.e., removing an AP should lower the RIV), ii) coherence (i.e., malware should have higher RIVs than apps), and iii) ease of understanding (i.e., the RIV of an app should be clearly understandable to the user, and it should allow straightforward comparison among values).

Also, another existing solution proposes a methodology for calculating a RIV for apps according to their category. More specifically, for each category, the kind and number of required APs are empirically inferred, thereby identifying permission patterns belonging to apps in each category. Then, the RIV is calculated by measuring a distance between the set of APs required by the app and the permission patterns of its category. Notwithstanding the encouraging empirical results obtained on a dataset made by 7.737 apps and 1.260 malware samples, the main limitation of the approach is in the need to know in advance the category of the app. Such information can be often unreliable as categories are manually chosen by developers.

Maetroid evaluates app risk according to both APs and metadata information related to the developer’s reputation and the source app market. The risk is calculated according to declared APs only, and by assigning static weights to each AP. Maetroid does not provide a quantitative RIV, but assigns each app in one (out of three) risk category. A framework for app risk analysis is made by three layers carrying out static, dynamic and behavioural analysis, respectively. The framework combines the results from each layer and builds up the RIV. Unluckily, the framework is purely theoretical and lacks any empirical evaluation, thereby making it difficult to assess the viability of the approach. DroidRisk is a quantitative method for calculating a RIV. DroidRisk is trained on a set of 27.274 apps and 1.260 malware samples, whereby it calculates the distribution of declared APs (i.e., those contained in the Android Manifest file). Then, DroidRisk applies a probabilistic function that calculates a RIV according to the kind and the potential impact of APs required by the app.

I argue that probabilistic methods suffer from some limitations.

1. They are unable to recognize as dangerous the malware that requires a limited set of APs; conversely, they averagely provide high RIVs for apps requiring many APs.
2. Current proposals deal with declared APs only, without deepening, for instance, which APs are actually exploited by the app. Due to the monotonicity of probabilistic risk indexes, relying only on declared permissions can impact the reliability, as apps are often overprivileged by their developers and can therefore obtain too high RIVs.
3. Probabilistic methods statically define the impact of APs, that is, all APs belonging to the same category (e.g., Normal, Dangerous, Signature, SignatureOrSystem) equally impact the estimation of the RIV. This choice does not allow to provide different impacts to APs, e.g., according to their distribution on the set of malware.

I argue that more reliable RIVs can be obtained through a machine learning approach based on – four sets of permissions for each app A, namely

1. Declared Permissions (DAPA), i.e., declared in the Android Manifest file;
2. Exploited permissions (EAPA), i.e., APs that are actually exploited in the app code;
3. Ghost permissions (GAPA), i.e., APs that the app tries to exploit in the code, but they are not declared in the Android Manifest file;
4. Useless permissions (UAPA), i.e., declared APs that are not exploited in the app code.

**Conclusion and Future Development:**

In this project I empirically assessed the reliability of probabilistic risk index approaches for Android apps, and proposed a methodology based on machine learning aimed at overcoming the shortcomings of the probabilistic solutions.

Future development of this project includes extending the feature set beyond APs, by taking into account suspicious API calls and URLs, both recognizable in the bytecode through the static analysis technique I adopted to build the permission sets.

**X-X-X-X**