AndroidMalwareCrypto

Adam Janovsky

Jul 09, 2022

CONTENTS

This tool allows for an analysis of cryptographic API in Android applications. The tool was especially developed to compare cryptographic API usage in benign vs. malicious applications and contains (weak) malware classifier based purely on cryptographic API features. We strive to provide end-to-end solution, automating all steps in the analysis:

1. Decompilation of APKs (*Mining crypto API*),
2. collection of cryptographic API usage in the decompiled binaries (*Mining crypto API*, *APK Dataset*),
3. exploratory data analysis of crypto API in your dataset (*Data preparation*, *Exploratory data analysis*),
4. training and evaluation of malware classifier based on crypto API features (*Classifier training*),
5. explanations of the classifier using SHAP (*Classifier interpretation*).

This documentation serves as a protocol to allow full replication of our experiments. Use the links in the list above (or table of contents) to navigate to specific step in our research.

CHAPTER

ONE

INSTALL

1.1 Docker

```
docker pull adamjanovsky/cryptomlw \
& docker run -it adamjanovsky/cryptomlw
```

1.2 Python

```
git clone https://github.com/adamjanovsky/AndroidMalwareCrypto \
& python3 -m venv venv \
& source venv/bin/activate \
& python3 ./setup.py install
```

CHAPTER

TWO

CONTENTS

2.1 Installation

You can either run the tool using our resources, or try to make it working by yourself.

2.1.1 Docker image and MyBinder.org notebooks

If you don't want to analyze large dataset by yourself, you can use our pre-processed dataset from this Jupyter notebook. Would you insist on going through the decompilation, you can use our Docker image with

```
docker pull adamjanovsky/cryptomlw:latest \
& docker run -it adamjanovsky/cryptomlw
```

Inside the image, you can run our toy experiment with

```
cd AndroidMalwareCrypto \
&& ./cli_process.py sample_experiment/configs/dataset_processing/config_processing.yml
```

and then see the output in `/home/user/AndroidMalwareCrypto/sample_experiment/dataset/`.

For exporting outputs outside of the container, we recommend Docker volumes.

2.1.2 Dependencies

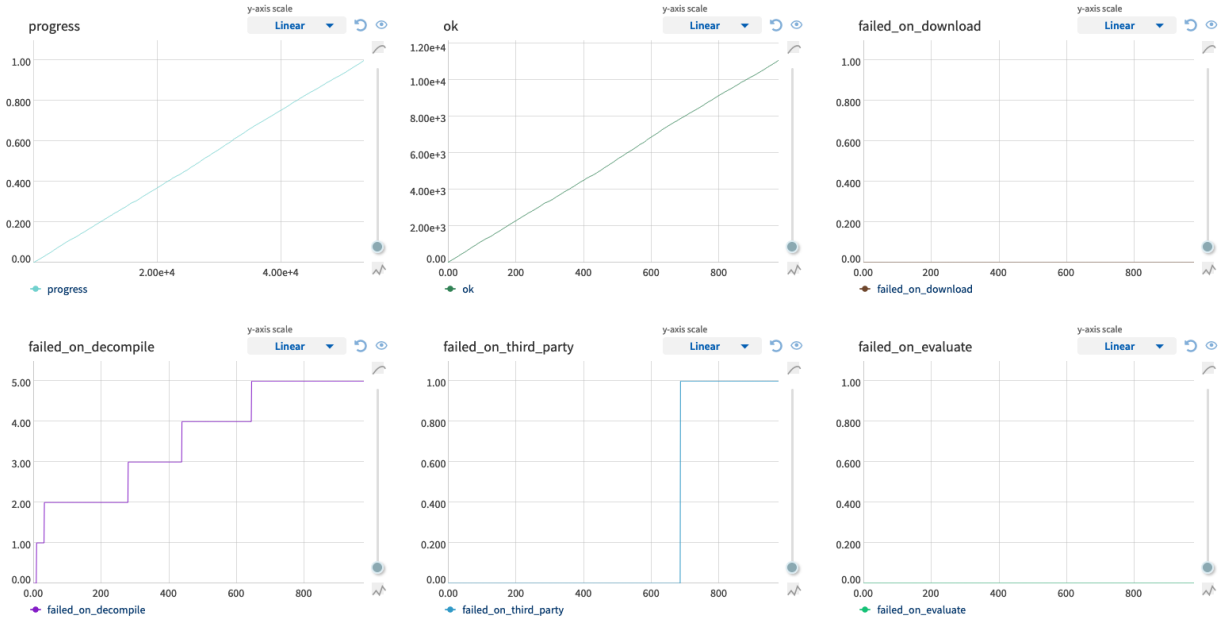
The rest of this document describes how to make our tool work on your local machine (unix). You're going to need Python 3.8. Start with the following

```
python3 -m venv venv \
& source venv/bin/activate \
& pip3 install -r requirements.txt
```

and continue with resolving the dependencies below. You can also use the Dockerfile that illustrates which commands were needed to run our tool on vanilla Ubuntu (without Neptune and Androzoo integration, however).

Neptune integration

The project is capabple of being fully integrated with Neptune.ai. This allows the user to track all experiments performed with the project. Most notably, it is possible to monitor a progress of long-running experiment.



If you wish to integrate Neptune.ai with this tool, you simply have to register at Neptune.ai and specify the following options in the configuration file:

```
is_being_logged: True # If you want to log in Neptune.ai or not.
neptune_project_name: 'Team/YourProjectName' # Name of project in the Neptune.ai
experiment_name: 'MyLittleExperiment' # Name of experiment for the Neptune.ai
```

If you do not want to integrate with Neptune.ai, simply put `is_being_logged: False` to the experiment config and should be safe to go.

Androguard and Jadx integration

We decided to use Jadx decompiler instead of the Androguard's default DAD. That is because Jadx decompiler, from our experience, produces cleaner source code (don't forget to use patched version of androguard mentioned in requirements.txt). Naturally, you need to have Jadx installed and present in a path. To navigate `androidcrypto` to your `jadx` decompiler, specify the following option in the configuration file:

```
jadx_path: '/path/to/jadx/build/jadx/bin/jadx' # Path to the jadx binary in your system,␣
↪keep it 'jadx' if jadx is in your system path
```

Literadar integration

Detection of third-party libraries usage uses the fork of LiteRadar script under the hood. That script is being executed as python2 process, so you need to have python2 (2.7) functionable as well on your computer.

2.2 Mining crypto API

The whole process of mining cryptographic API is operated by `cli_process.py` that is in turn parametrized by config.yml. While the configuration is exhaustively described in the configuration file itself, here we recapitulate the most important steps of the process. There are several steps to be done:

1. Download dataset or load it from disk
2. Decompile the APKs
3. Detect all third-party libraries
4. Collect crypto API usage
5. Assign sample to malware family/category with Euphony

The output of `cli_process.py` is record.json that summarizes findings extracted from the dataset. This file is further processed by *data preparation* which further cleans it.

We comment on some problematic parts of the setup below.

2.2.1 Download dataset or load it from disk

This step is thoroughly described at *APK Dataset*.

2.2.2 Decompile the APKs

Should work seamlessly, provided that you *sorted out the Jadx dependency* and installed the patched androguard from `requirements.txt`.

2.2.3 Detect all third-party libraries

Provided that you *sorted out the LiteRadar dependency*, you just have to specify the path to the LiteRadar binary in the experiment config using the following parameter:

```
literadar_path: '/path/to/LiteRadar/LiteRadar/literadar.py' # Path to the LiteRadar␣
↪python2 script
```

2.2.4 Assign sample to malware family/category with Euphony

Possibly, you can also let Euphony label your malware samples with family names and types. Note that this functionality will probably fail when applied outside of the Androzoo dataset (more precisely it will result in `None` labels). In Androzoo dataset, it will only work with samples 2012-2017, `None` labels will be assigned elsewhere.

In order to integrate Euphony with `cli_process.py`, you have to download the labels from Androzoo website and navigate the `cli_process.py` to these files using the following configuration keys:

```
euphony_names_path: '/path/to/euphony/names_proposed.json'
euphony_types_path: '/path/to/euphony/types_proposed.json'
```

2.3 Data preparation

The whole process of preparing the dataset is operated by `cli_prepare.py` that is in turn parametrized by config.yml. While the configuration is exhaustively described in the configuration file itself, here we recapitulate the steps of preparation pipeline. There are four steps:

1. Cleaning
2. Feature engineering
3. Records selection
4. Feature scaling
5. Feature selection

The input of the full pipeline are files processed during *crypto API mining*. The output of the full pipeline is usually a single hdf file per objective, which contains train and test dataframes of features and target labels. The currently supported objectives are:

- **malware labeling** – label malicious sample as one of the malware families
- **malware detection** – classify a sample as malicious or benign

The configuration can be tweaked to output results of the steps in the pipeline. For example, it might be beneficial to store the ouput of the cleaning. The pipeline's steps can be also skipped, but only from the beginning or the end. Specifically, each step expects the output of the previous step or paths to the data generated by previous step.

We further explain each of the pipeline steps in their order. We also provide an overview of tweaked configuration files.

2.3.1 Cleaning

During this step, the outputs of *crypto API mining* are merged together and cleaned. Each input file can be further specified as benign (or malicious) in the configuration. Additionally, the source of the sample can be added to possibly differentiate between different datasets during analysis.

Overall, the process of cleaning can be decomposed into five sequential steps, which are optional if not stated otherwise:

1. Clean missing values – mandatory step that replaces the missing values with appropriate constants
2. Clean labels:
    1. Rename family names, which are considered inaccurate, based on the rules in the configuration.
    2. Rename types based, which are considered inaccurate, based on the rules in the configuration.
    3. Adjust the mapping between family names and types to be 1:1. Specifically, change each sample's type to the most common type corresponding to the sample's family name.

3. Remove third party cryptography libraries from third party packages because of duplicit information
4. Remove similar classes for crypto API calls records because they are considered to be duplicites. Generally, classes with illegal characters in their names and another similar class are removed.
5. Clean crypto API calls:
    1. Remove crypto API calls records when crypto API imports are empty for the sample.
    2. Remove all the crypto API calls that are prefixes of another one on the same line.
    3. Remove crypto API calls false positives, for example when it is used as a substring in user-defined method, etc.

The output of this step can be optionally stored into a single `.json` file.

2.3.2 Feature engineering

During this step, features suitable for machine learning are engineered. This step's output can be optionally stored into a single `.csv` (hdf) file.

2.3.3 Records selection

First, the dataset is split into train and test with ratio specified in configuration. Next, the labels are adjusted for the specified objectives:

- **malware detection** – change target label `benign` for `malicious`
- **malware labeling** – take only the top family labels, optionally merge others into special label OTHER, optionally remove all the samples with UNKNOWN family tag

After this step, the dataset is ready for *classifier training*. This step's output per objective can be optionally stored into a single `.h5` (hdf) file. The dataframes are then stored under keys `X_train`, `X_test`, `y_train`, `y_test` where prefix X represents features and y represents labels.

2.3.4 Feature scaling

During this step, the features are optionally scaled according to configuration in the given order:

1. Normalize features by overall class count (`metadata_n_classes`)
2. Normalize features by total lines of code (`metadata_n_lines`)
3. Scale features using standard scaling In default configuration, only normalization by class count is used.

An important configuration is `columns_to_ignore`. The default columns should be left there because they are metadata columns and are not intended for use during training. However, more columns can be added to not scale them (in general only numeric columns are scaled).

This step's output per objective can be optionally stored into a single `.h5` (hdf) file. The dataframes are then stored under keys `X_train`, `X_test`, `y_train`, and `y_test` where prefix X represents features and y represents labels.

2.3.5 Feature selection

During this step, a subset of features that is deemed to be useful for each objective is selected in three steps:

1. Remove features with low variance using Variance Threshold
2. Remove features that are linearly correlated above a threshold specified in configuration
3. Use Boruta to remove features with low predictive power

An important configuration is `columns_to_ignore`. The default columns should be left there because they are metadata columns and are not intended for use during training. However, more columns can be added to ignore them during feature selection.

This step's output per objective can be optionally stored into a single `.h5` (hdf) file. The dataframes are then stored under keys `X_train`, `X_test`, `y_train`, and `y_test` where prefix X represents features and y represents labels.

2.3.6 Configurations

Below are outlined various configurations with their simple description:

- full_with_clean_output.yml – whole pipeline with output of cleaned dataset
- without_feature_selection.yml – whole pipeline without feature selection
- clean_only.yml – clean only configuration
- from_clean.yml – pipeline without cleaning (starting from records selection)
- without_feature_scaling.yml – pipeline without feature scaling but with feature selection
- to_feature_scaling.yml – pipeline from cleaning up to records selection (without feature scaling and feature selection)
- full_with_feature_engineering_output.yml – full pipeline with output during feature engineering

2.4 Exploratory data analysis

There are generally two ways to explore the dataset in a semi-automatic way:

- using **Evaluator** on a raw dataset from *crypto API mining*.
- using **Automatic exploratory analysis** on prepared (train) dataset from feature engineering step of *data preparation pipeline* These two ways are outlined below.

2.4.1 Evaluator

Some useful information can be explored from the dataset using Evaluator. Evaluator can be used as an API inside own notebooks/scripts.

Even though the evaluator can use raw dataset, it is actually preferred to clean the dataset using *data preparation pipeline*. For this reason there is available explore_template.ipynb which deals with cleaning before using evaluator and all needed configuration can be set up in constants in the first cell. For examples of executed exploratory noteeboks see:

- explore_benign.ipynb
- explore_malicious.ipynb

However, these notebooks (but also evaluator) expect a single `records.json` file. If there are multiple `records.json` files, for example per year, it can be useful to merge them together. For this reason there is a simple utility in the repository, merge_jsons.py.

2.4.2 Automatic exploratory analysis

Automatic exploratory analysis is performed using pandas-profiling library. Our wrapper around this library, `cli_automatically_explore.py` can be used with specified input `.csv` file and output path where to store the report.

The input file should contain features after feature engineering. Please note that feature scaling and feature selection are not needed here. Generating the report for the whole dataset can be very time-consuming, so, there are two ways to deal with this:

- use minimal mode by specifying `--minimal_mode` flag – this mode does not calculate some information, for example correlation between all features
- use sample size by specifying `--sample_size`, which is by default 10 000 When minimal mode is used then sample size is completely ignored.

2.5 Classifier training

The whole process of preparing the dataset is operated by `cli_train.py` that is in turn parametrized by config.yml. While the configuration is exhaustively described in the configuration file itself, here we recapitulate some key takeaways.

The input of `cli_train.py` is the dataset prepared during *data preparation pipeline* atleast up to records selection (feature selection or feature scaling can be excluded). The output of is the stored trained classifier and optionally the prediction on the test dataset. The output can be further evaluated (*exploratory data analysis*) and interpreted (*classifier interpretation*).

2.5.1 Supported classifiers

Below are outlined the classifiers with all posible tags that can be used in the configuration:

- Random Forest: rf, random_forest, random forest
- LightGBM Gradient Boosted Decision Trees: lgbm
- Linear Support Vector Machines: svm, support_vector_machines, support_vector_machine, support vector machine, support vector machiens
- Gaussian Naive Bayes: gnb, gaussian_naive_bayes, gaussian naive bayes, gaussian_nb, gaussian nb
- Complement Naive Bayes: cnb, complement_naive_bayes, complement naive bayes, complement_nb, complement nb
- Linear Discriminant Analysis: lda, linear_discriminant_analysis, lienar discriminant analysis
- Explainable Boosting: eb, ebm, explainable_boosting, explainable boosting
- Decision Rule List: rules

Please note that, only Random Forest is currently fully supported. The other classifiers are considered experimental.

2.5.2 Correct input format

The input is a single `.h5` (hdf) file which must contain dataframes under keys:

- `X_train` - training features
- `y_train` - training labels
- `X_test` - testing features

While it is desired to use the output of *data preparation pipeline*, the input can be generated by different tools. The input file only has to have the specified dataframes as keys with correct dimensionality and corresponding data quality, for example no missing values.

2.5.3 Classifier training

The classifiers are trained using cross-validation with the count of splits specified in configuration as `cv_splits`, which is by default 5. The metric which is optimized is usually f1-score, which is macro-averaged during multi-class classification. Additionally, inverse class weights are used when possible to deal with any class imbalances.

Some of the columns in the dataset can be optionally ignored during training. Please see `columns_to_ignore` in configuration. However, take care to leave the default columns there because they are only intended as metadata.

2.5.4 Training for multiple objective and in parallel

Currently, the classifiers specified in a single configuration file are trained **sequentially**. Still, the training of a single classifier can be parallelized by specifying `threads`, which is by default 8, in configuration. If the goal is to train multiple classifiers in parallel then the solution is to have a configuration file for each classifier.

Additionally, when training classifiers for more objectives (or just using different input datasets for the same objective) new configuration files have to be specified.

2.6 Classifier interpretation

The process of interpreting of classifiers happens generally in two steps:

1. Calculate SHAP values using `cli_explain.py`
2. Use the calculated SHAP values to interpret classifiers' local and global decisions

`cli_explain.py` is parametrized by config.yml. While the configuration is exhaustively described in the configuration file itself, here we recapitulate some key takeaways. The tool requires two inputs:

- dataset prepared during *data preparation pipeline*
- trained model during *classifier training*

2.6.1 Input format

Please note that the input dataset in hdf format should correspond to dataset that is output of *data preparation pipeline*. Specifically, it has to be output of records selection, feature scaling, or feature selection steps. Furthermore, the trained model should be trained on data with same features as the dataset.

2.6.2 Supported models

The tool, `cli_explain.py`, is a simple wrapper around SHAP library. Currently, only TreeSHAP is supported and tested for Random Forest.

2.6.3 Configuration options

Below are explained some configuration options and their interactions.

It is desired to not use the whole dataset. There are two ways to do so:

- for global interpretability use `sample_size` of, for example 10 000 samples, specifying negative values is regarded as whole dataset
- for local interpretability use `indices` to specify list of indices to interpret Additionally, these two approaches can be combined because first the indices are used to select the data and then the sample is drawn from it.

Additional three options can be specified which are explained in TreeSHAP documentation:

- `feature_perturbation`
- `model_output`
- `approximate` Please note that when `interventional` method is used for `feature_perturbation` then only 10 000 samples are used as background data even when `sample_size` is not specified! This is done for the results to be consistent when using different `indices` over the same dataset.

The configuration `columns_to_ignore` specifies which columns to ignore during calculation. This **must** correspond to the value used during *classifier training*.

2.6.4 Calculation speed and parallelization

Calculating SHAP values is generally time-consuming process. Thus, as was previously mentioned it is recommended to not use the whole dataset. It is also beneficial to not calculate interaction values if they are not needed (usually only SHAP values are desired).

The calculation of SHAP values can be also parallelized by creating more configurations with different `indices` and then using the calculated values together.

2.6.5 Output format

The output of `cli_explain.py` is a single `.h5` (hdf) file specified by `output_path`. It contains 5 keys:

- `shap_values` - dataframe of shap values if `calculate_shap_values` was set to True
- `shap_interaction_values` - dataframe of shap values if `calculate_interaction_values` was set to True
- `values` - dataframe of values used if `save_values` was set to True
- `shap_expected_values` - dataframe of expected value for each class

- `targets` - dataframe of targets used if `save_targets` was set to True
- `hashes` - dataframe of hashes used if `save_hashes` was set to True

2.6.6 Using calculated values

The calculated values can then be loaded in a Jupyter notebook and used with SHAP library to explain the classifier decisions. For loading the values we provide a function androidcrypto.analysis.explainability.api.load_explainability_output. For an example of using these values see notebook.

2.7 Classifier Evaluation.

The classifiers can be easily evaluated using evaluation API or notebooks.

Evaluation API is available in evaluation.py.

A template of evaluation notebook which expects `.csv` predictions is available in evaluation_template.ipynb.

2.8 APK Dataset

There are two ways of supplying the dataset of APK binaries for analysis:

- Load own APKs from a disk,
- command the `download` task of `cli_process.py` to provide a dataset from Androzoo.

2.8.1 Load own APKs from disk

Just prepare a folder that looks like this

```
dataset
├── data
    ├── apk
        ├── first.apk
        ├── second.apk
        ├── third.apk
        ├── fourth.apk
        ├── fifth.apk
```

and navigate the tool to the folder by specifying

```
dataset_path: "/path/to/folder/with/dataset"
```

in the config file.

2.8.2 Download samples from Androzoo

Apart from providing your own dataset, this tool is capable of leveraing the Androzoo dataset to download malicious APKs directly from their database. This is done as a part of the `download` task in `cli_process.py`. In order to be able to create datasets from Androzoo, you have to:

- register in the Androzoo service and obtain your API key,
- download the list of APKs in the Androzoo (csv) and possibly shuffle the lines of it (if you want to ensure random sampling),
- in the configuration file, provide path to Androzoo csv_file as

```
csv_path: '/path/to/androzoo/androzoo_shuffled.csv' # path to your csv file of Androzoo
```

- include the Androzoo token as a command-line parameter when running `cli_process` as

```
./cli_process.py -a my_androzoo_token
```

It should be noted that the parameters for the Androzoo sampling (how many samples to download, should they be malicious?, etc.) can be found in the `download` task configuration in the configuration file.

How the dataset looks like after processing

The folder sample_dataset shows the form of a dataset (and is used in the sample experiment). The directory structure is shown in the listing below.

```
dataset
├── data
│   ├── apk
│   │   ├── 0d9e99e79d9a04e41da65fed037b8560612c12d559581664275 57f78bc887585.apk
│   │   ├── 3d44791ccd1d5b871a1e4d4fcfc90b667f219c7979ca02d220e442552ec2b357.apk
│   │   ├── 570232b8f1b197fcf31a91a924f0fa2b0480a7b1cd98c2f0ab79b1110eb94e3c.apk
│   │   ├── 58d75f26c56e9f91b06513cf9a990289222d85641d338be6d9cb3967583d3727.apk
│   │   ├── 63890df926f97eb10370a0fb5f667ce9610887128a8cc69e0e53c57ffe06c05d.apk
│   │   ├── 9fb6230d41fa2a9c7b89d61ab61e77079dd415b481810f320aa267758ced59e9.apk
│   │   ├── a002fecefcd5e26b14e4221c1c59a2e60751034e8f9cd2a36c4e2a4267d03521.apk
│   │   ├── a5513cc8ed6bac8ec3252a51b550ee38c1b3a463e5f21303a1af0b7d83e0fb7a.apk
│   │   ├── d9e5c58dced69e209b5fc8721cc249ef3cd8b9f425469a3d8e3a1c829b0932a9.apk
│   │   └── f26a759ff6b136ba291494b0cbf42f06b240092662ab78d2f7d6369d685b5a4a.apk
│   └── dx
├── meta.yml
└── readme.md
```

Meta.yml file

The `meta.yml` file is automatically created for each supplied dataset. You don't have to explicitly provide it, it will be created automatically.