README: How to experiment the artifacts

Phuong T. Nguyen, Claudio Di Sipio, Juri Di Rocco, Davide Di Ruscio[1] and Massimiliano Di Penta[2]

[1] *Università degli studi dell'Aquila, L'Aquila, Italy*
{phuong.nguyen, claudio.disipio, juri.dirocco, davide.diruscio}@univaq.it
[2] *Università degli Studi del Sannio, Benevento, Italy*
dipenta@unisannio.it

ABSTRACT

This document provides instructions on how to get the datasets, to obtain, install, as well as to execute the tools for replicating the experiments described in our paper entitled *"Adversarial Attacks to API Recommender Systems: Time to Wake Up and Smell the Coffee?"* accepted in the Technical Track of the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE 2021). A preprint of the paper is available in the following link: `https://bit.ly/3B8Q7GS`.

1 Introduction

API recommender systems have gained traction in recent years as they became more successful at suggesting API calls or code snippets. While these systems have proven to be effective in terms of prediction accuracy, there has been less attention for what concerns such recommenders' resilience against adversarial attempts. In the accepted paper [8], we conducted an empirical evaluation on three API recommender systems to see if they are prone to malicious training data. The evaluation performed on three state-of-the-art API recommender systems, i.e., UP-Miner [10], PAM [3], and FOCUS [6] reveals a worrying outcome: all of them are not immune to malicious data. The results obtained suggest that even with a small amount of artificial training data, clients are always provided with the fake/malicious APIs. Altogether, we see that API recommender systems are likely to be exploited, and in this way they inadvertently become a *trojan horse*, causing havoc to software systems.

To pave the way for further explanations, in this section we give a short introduction on the way the approach works. Let us imagine a scenario in which one increases the popularity of malicious APIs by planting them to OSS projects, as many as possible. Fig. 1 illustrates the process in which attackers may exploit to plant malicious data. First, well-maintained repositories are

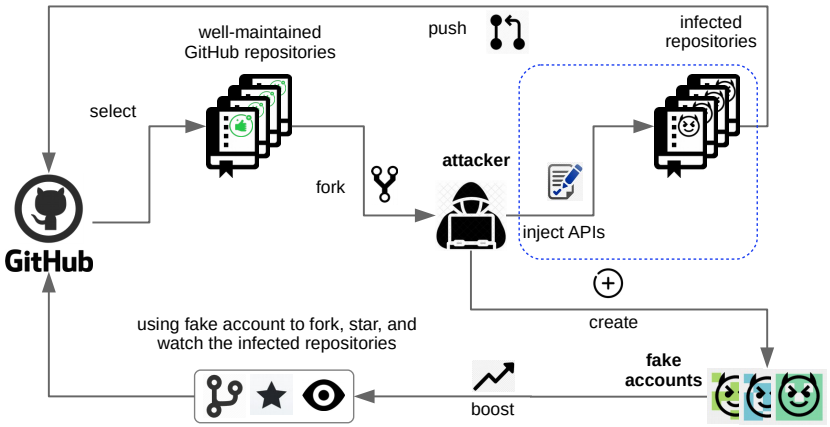

Figure 1: Manipulating GitHub repositories.

forked from GitHub, e.g., those that have good indicators in terms of stars, forks, or watchers. Afterwards, the projects are injected with fake APIs, and then uploaded again to GitHub. Attackers may create fake accounts to star, fork, and watch malicious repositories to increase their credibility/visibility, thus exposing them better to search engines.[1]

> **NOTE**: We consider an API malicious if it causes fatal errors, no matter where it comes from, i.e., either a legitimate or a fake library. An example of malicious APIs is available in this link: `https://bit.ly/31R760l`.

To simplify the evaluation, in the scope of our work [8], we inject APIs at the metadata level, i.e., after the data that has been parsed to a processable format. This is for experimental purposes only, since in reality we need to seed data directly to projects and upload them to GitHub as shown in Fig. 1. This is outside the focus of our paper, and we leave it for future work.

The artifacts include *(i)* two API injectors, one for FOCUS and the other one for both UP-Miner and PAM; and *(ii)* two Android datasets. In the following sections, we describe the systems and datasets used in the evaluation. Moreover, we also give more details on the steps needed to inject fake APIs at the metadata levels, and to conduct the experiments.

2 Injecting fake APIs

To boost up the popularity of an API pattern, the fake/malicious APIs can be seeded into a significant number of declarations, for each training project. There are the following parameters to consider, when it comes to populating artificial projects with fake APIs.

- α is the ratio of projects injected with fake APIs.

[1]Such a manipulation has been recently revealed `https://zd.net/3bg3CK9`.

Table 1: Characteristics and risks of the three API recommender systems.

System	Working mechanism	Potential risks
UP-Miner [10]	UP-Miner works on the basis of clustering, is dependent on the similarity among API sequences. It computing. In other words, UP-Miner computes similarity at the sequence level, i.e., APIs that are usually found together using BIDE. Finally, it clusters to group frequent sequences into patterns.	Similar to MAPO, as UP-Miner depends on BIDE, an attacker can inject malicious code in the training in projects disguised as similar to trick UP-Miner. In this way, it may recommend to developers harmful snippets.
PAM [3]	PAM defines a distribution over all possible API patterns in client code, based on a set of patterns. It uses a generative model to infer the most probable patterns. The system generates candidates by relying on the highest support first rule.	The system recommends API calls that commonly appear in different code snippets. Thus, push and nuke attacks could modify the final ranking obtained by the tool, i.e., operating on terms' occurrences to favor or defame a certain API pattern.
FOCUS [5, 7]	FOCUS suggests APIs by encoding projects in a tensor and using a collaborative-filtering technique to deliver the list of APIs. Eventually, it mines APIs and snippets from similar projects with a graph representation.	Due to the internal design, the system is susceptible to poisoning attacks, i.e., an adversary can create fake similar projects containing toxic APIs and pose them as legitimate to deceiving FOCUS into recommending these calls/snippets.

- β is the ratio of methods in a project getting fake APIs.
- Ω is the number of fake APIs injected to each declaration.

We opted for the following sets of parameters: α=$\{5\%, 10\%, 15\%, 20\%\}$, β=$\{40\%, 50\%, 60\%\}$, Ω=$\{1, 2\}$. The rationale behind the selection of these values is as follows. Concerning α, though having popular APIs is commonplace, it is difficult to rack up projects with malicious APIs, thus α is set to the following thresholds α=$\{5\%, 10\%, 15\%, 20\%\}$. In contrast, within a project, attackers have more freedom to embed APIs to declarations, therefore β is varied from 40%, 50%, to 60%. Finally, the number of fake APIs should be kept low to make attacks more feasible, i.e., Ω=$\{1, 2\}$. We study how the calibration of the parameters affects the final efficiency, aiming to anticipate the extent to which the attacks are successful in the field.

3 Systems and datasets

In Section 3.1 we briefly introduce the systems considered in the evaluation, namely UP-Miner [10], PAM [3], and FOCUS [5]. Meanwhile in Section 3.2, we present the dataset exploited for the experiments. The software and hardware configurations for the testing platform are specified in Section 3.3.

3.1 Considered recommender systems

In the evaluation, we examined the resilience of UP-Miner [10], PAM [3], and FOCUS [5] on a dataset containing Android apps' source code. We selected these tools due to the following reasons. *First*, UP-Miner is a well-established

recommender system, which has shown to outperform MAPO [11], one of the first systems for suggesting APIs. PAM has been proven to be more effective compared to MAPO and UP-Miner [3]. Meanwhile, FOCUS is the most recent approach and it obtains the best prediction performance if compared to both UP-Miner and PAM [7]. *Second*, the considered systems are also representative in terms of working mechanism. UP-Miner works based on clustering, while PAM mines API patterns that commonly appear in different snippets, and finally FOCUS exploits a collaborative-filtering technique, i.e., also based on a similarity algorithm, to retrieve APIs from similar projects. *Third*, the three tools have evaluation replication package available. Such the implementations enable us to run the experiments according to our needs. Table 1 gives more information on the three tools considered in our evaluation.

3.2 Datasets

We made use of a dataset which was curated through our recent work [7], and the collection process is summarized as shown in Fig. 2. First, we searched for open source projects using the *AndroidTimeMachine* platform [4], which retrieves apps and their source code from Google Play[2] and GitHub. Second, APK files are fetched from the Apkpure platform,[3] using a Python script. Third, the *dex2jar* tool [1] is used to convert the APK files into the JAR format. The JAR files were then fed as input for Rascal [2] to convert them into the M^3 format [2]. We obtained a set of 2,600 apps with full source code. For more detail about how the collection was done, the readers are kindly referred to the replication package of our recent work[4]. The M_3 models are stored in `initial_focus_dataset` folder.

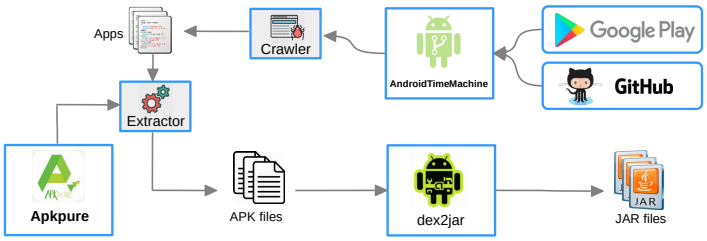

Figure 2: The data extraction process.

We then inserted fake APIs to random projects and declarations, attempting to simulate real-world scenarios where APIs are dispersed across several declarations. Finally, the resulting data is parsed in two formats, i.e., ARFF files to be fed to UP-Miner and PAM, and a special file format for providing input to FOCUS.

[2] `https://play.google.com/`
[3] `https://apkpure.com/`
[4] `https://mdegroup.github.io/FOCUS-Appendix/`

UP-Miner and PAM suffer from scalability issues, i.e., they cannot work on large datasets [7]. Thus for evaluating them, we could consider a subset consisting of 500 apps. For FOCUS, the whole 2,600 apps are used since the system is capable of handling well a large amount of data.

> **NOTE**: This section introduces the data extraction process, for the sake of reproducibility. However, we completely parsed the metadata needed as input for the three recommender systems at your disposal. The tasks described in this section require a long computation time as well as knowledge about Rascal, and we strongly suggest using the metadata as it is.

3.3 Testing platform

Table 3.3 specifies the hardware and software requirements that a testing system needs to meet in order to be eligible for the execution of the artifacts.

Name	**Requirements**
RAM	$\geq$ 8GB
Operating System	A modern Linux system[1]
Java JRE	$\geq$ Java 8
Apache Maven	$\geq$ Maven 3.0
Python	$\geq$ 3.7
Git	$\geq$ 2.0

Table 2: Hardware and software requirements

[1] We developed and tested the tools involved in the experiments on different Linux systems, although it should work without any issue on any operating system. In the instructions, we assume a Linux system and basic command line skills.

In our case, the testing platform is a PC with Linux 4.20.3, Intel Core i7-6700HQ CPU @ 2.60GHz and 16GB of RAM.

4 Obtaining the artifacts

On a Linux system, open a terminal and execute the following command to download all artifacts from a single GitHub repository:

```
$ git clone https://github.com/MDEGroup/APIRecSys-AML.git
    APIRecSys-AML
```

This will create a new directory named `APIRecSys-AML`, and we call it the root directory in the scope of this document. For the sake of clarity, we do not write the full path for the directory. In practice, an example of a full path in Linux is `/home/admin/Desktop/APIRecSys-AML/`. The root directory has the following structure:

- The `FOCUS` directory contains the implementation of the injector and the dataset consisting of 2,600 artifacts:

  - `injector`: the Java implementation of the injector that provides input for FOCUS.
  - `dataset`: a set of 2,600 M_3 models extracted by Rascal and ready to be used with FOCUS.
  - `ASE2021-FOCUS`: this folder contains the source implementation of FOCUS.

- The `UPMiner_PAM` directory contains the ARFF injector and the sub-datasets of 500 android apps that we used to evaluate PAM and FOCUS:
  - `injector`: because both PAM and UP-Miner uses ARFF files as input, the injector works for both UP-Miner and PAM
  - `dataset` contains the datasets generated by varying the inject parameter on the initial dataset of 500 android apps.

- The FOCUS metadata coming from 2,600 Android apps is stored in the `initial_focus_dataset` folder. The canonical name of the app can be extracted from the filename g_<canonical-name>-dex2jar.jar.txt. We acknowledge the original data collected from the AndroidTimeMachine platform [4].

- The UP-Miner and PAM metadata coming from 500 Android apps is stored in the `initial_upminer_pam_dataset` folder. The canonical name of the app can be extracted from the file name g_<canonical-name>-dex2jar.jar.txt.

- `ALL_PAPERS.xlsx` is an Excel file to report the paper collection used to perform the literature analysis. We investigate whether there is already any effort devoted to studying and dealing with threats to RSSE originating from malicious data. For further details, we refer the interested readers to Section III.A and Section IV.A of the accepted paper [8].

5 Executing the experiments with FOCUS

This section describes how to run FOCUS on the dataset using the configurations to reproduce the results presented in the paper.

5.1 Injecting fake APIs

We have a dedicated tool located in `APIRecSys-AML/FOCUS/injector/` to inject fake APIs following the description in Section 2. Please run the following command to perform the injection.

```
$ mvn compile exec:java -Dexec.mainClass="ase2021.aml.apirecsys.
   Runner" -Dexec.args="-alpha 0.4 -beta 1 -omega 1 -src ../../../
   initial_focus_dataset/"
```

Once you run this command, the APIs will be injected to the original projects.

5.2 Running FOCUS

To run FOCUS, please navigate to the `APIRecSys-AML/FOCUS/ASE2021-FOCUS` containing the stand alone FOCUS implementation and execute it, using the following commands:

```
$ cd APIRecSys-AML/FOCUS/ASE2021-FOCUS/
$ mvn compile exec:java -Dexec.mainClass="it.univaq.disim.seagroup
    .FOCUS.Runner" -Dexec.args="-src <injected_dataset>"
```

where `<injected_dataset>` is the path of the dataset injected by fake APIs.

Intermediate results are stored in the evaluation folder of the corresponding dataset's directory.

6 Executing the experiments with UP-Miner and PAM

Similar to the experiments with FOCUS, we test UP-Miner and PAM with manipulated data. Section 6.1 describes how to run the API injector, and Section 6.2 gives the instructions to run both tools.

6.1 Injecting fake APIs

Since both PAM and UP-Miner rely on ARFF file format to perform the recommendations, we develop a Python script that injects API calls to the initial project file using *liac-arff*, a tailored library to open and manipulate ARFF[5]. The `main_arff` function opens the project file and injects one or two fake APIs for a certain number of method declarations according to the configurations defined in Section 2. For instance, you can inject two fake APIs into 40% of methods belonging to 5% of projects, i.e., Ω=2, β=40, and α=5 by running the following command:

```
$ cd APIRecSys-AML/UPMiner_PAM/injector/
$ pip install liac-arff
$ pip install click
$ python3 injector.py --root=../../initial_upminer_pam_dataset/ --
    dest=./dest --beta=40 --alpha=5 --fake_api=org.fake1 --
    fake_api2=org.fake2
```

[5] `https://pythonhosted.org/liac-arff/`

Though such kind of attacks is simple, they can compromise the quality of the final recommendations as shown in the paper [8].

6.2 Running PAM and UP-Miner

To evaluate the impact of adversarial attacks, we run PAM and UP-Miner as described in the original supporting GitHub repository `https://github.com/mast-group/api-mining`.

Please execute the following command to clone, run PAM and UP-Miner on the injected input data generated in Section 6.1:

```
$ git clone https://github.com/mast-group/api-mining.git APIRecSys-AML/UPMiner_PAM/
$ cd APIRecSys-AML/UPMiner_PAM/
$ mvn package
$ for f in <injected_dataset> *; do java -jar api-mining/target/api-mining-1.0.jar apimining.pam.main.PAM -f $f; done
$ for f in <injected_dataset> *; do java -jar api-mining/target/api-mining-1.0.jar pimining.upminer.UPMiner -f $f; done
```

where `<injected_dataset>` is the path of the dataset injected by fake APIs.

Troubleshooting

If you encounter any problem during the evaluation, please do not hesitate to contact us through one of the following emails: phuong.nguyen@univaq.it, juri.dirocco@univaq.it, claudio.disipio@graduate.univaq.it.

References

[1] dex2jar. Library Catalog: tools.kali.org.

[2] B. Basten, M. Hills, P. Klint, D. Landman, A. Shahi, M. J. Steindorfer, and J. J. Vinju. M3: A General Model for Code Analytics in Rascal. In *1st International Workshop on Software Analytics*, pages 25–28, Piscataway, 2015. IEEE.

[3] J. Fowkes and C. Sutton. Parameter-free Probabilistic API Mining Across GitHub. In *24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 254–265, New York, 2016. ACM.

[4] F. Geiger, I. Malavolta, L. Pascarella, F. Palomba, D. Di Nucci, and A. Bacchelli. A graph-based dataset of commit history of real-world android apps. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pages 30–33, 2018.

[5] P. T. Nguyen, J. Di Rocco, D. Di Ruscio, L. Ochoa, T. Degueule, and M. Di Penta. FOCUS: A Recommender System for Mining API Function Calls and Usage Patterns. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, pages 1050–1060, Piscataway, NJ, USA, 2019. IEEE Press.

[6] P. T. Nguyen, J. Di Rocco, D. Di Ruscio, L. Ochoa, T. Degueule, and M. Di Penta. FOCUS: A Recommender System for Mining API Function Calls and Usage Patterns - to appear. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, May 2019.

[7] P. T. Nguyen, J. Di Rocco, C. Di Sipio, D. Di Ruscio, and M. Di Penta. Recommending api function calls and code snippets to support software development. *IEEE Transactions on Software Engineering*, pages 1–1, 2021.

[8] P. T. Nguyen, C. Di Sipio, J. Di Rocco, D. Di Ruscio, and M. Di Penta. Adversarial Attacks to API Recommender Systems: Time to Wake Up and Smell the Coffee? In *In Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering*, May 2021.

[9] M. O. F. Rokon, R. Islam, A. Darki, E. E. Papalexakis, and M. Faloutsos. Sourcefinder: Finding malware source-code from publicly available repositories. *ArXiv*, abs/2005.14311, 2020.

[10] J. Wang, Y. Dang, H. Zhang, K. Chen, T. Xie, and D. Zhang. Mining Succinct and High-coverage API Usage Patterns from Source Code. In *10th Working Conference on Mining Software Repositories*, pages 319–328, Piscataway, 2013. IEEE.

[11] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. MAPO: Mining and Recommending API Usage Patterns. In *23rd European Conference on Object-Oriented Programming*, pages 318–343, Berlin, Heidelberg, 2009. Springer.