Adversarial Attacks to API Recommender Systems: Time to Wake Up and Smell the Coffee?

Phuong T. Nguyen, Claudio Di Sipio, Juri Di Rocco
University of L'Aquila, Italy
{phuong.nguyen, claudio.disipio, juri.dirocco}@univaq.it

Massimiliano Di Penta
University of Sannio, Italy
dipenta@unisannio.it

Davide Di Ruscio
University of L'Aquila, Italy
davide.diruscio@univaq.it

Abstract—**Recommender systems in software engineering provide developers with a wide range of valuable items to help them complete their tasks. Among others, API recommender systems have gained momentum 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 fact, by crafting the recommenders' learning material, e.g., data from large open-source software (OSS) repositories, hostile users may succeed in injecting malicious data, putting at risk the software clients adopting API recommender systems. In this paper, we present an empirical investigation of adversarial machine learning techniques and their possible influence on recommender systems. The evaluation performed on three state-of-the-art API recommender systems reveals a worrying outcome: all of them are not immune to malicious data. The obtained result triggers the need for effective countermeasures to protect recommender systems against hostile attacks disguised in training data.**

I. Introduction

In recent years, several recommender systems for software engineering (RSSE) have been conceptualized to support developers in their task and, possibly, reduce the increasing information overload originating from the availability of data from various sources [33], [34], [19], [44], [38]. A relevant example of RSSE is represented by API recommender systems, which provide developers with function calls and/or code snippets being beneficial for their coding tasks [39], [32], [47]. Such systems learn their recommendations (of code snippets or APIs) from external sources such as code bases, e.g., GitHub or communication channels, e.g., Stack Overflow. Since these sources are open for changes and contributions by the crowd, the recommenders' learning material might be exposed to malicious attacks [61]. In other words, these sources can be exploited as a means to fool API recommender systems.

Adversarial attempts generate perturbations to deceive and disrupt systems by causing a malfunction, compromising their recommendation capabilities. For instance, an adversarial attack to recommender systems, depending on the intent, can favor or defame an item, thus creating a negative impact on the final recommendations [20]. Similarly, by manipulating training data available in OSS platforms, hostile users may render recommender systems vulnerable to harmful artifacts. If a recommender is trained to provide malicious APIs or snippets, this can trigger disruptions to a software system. For example, a recent work [58] shows that there have been attempts using toxic code to secretly force Android apps to open ports, allowing for unauthorized access.

Research on Adversarial Machine Learning [54], [23] (AML) studies security issues in Machine Learning systems as well as general-purpose recommender systems [11]. So far, AML has been investigated in different domains, e.g., online systems [30], image classification [36], and addresses both threats and countermeasures [2], [57]. However, to the best of our knowledge, the issue of AML in recommender systems remains unexplored in software engineering.

In this paper, we provide a first empirical investigation into the relevance and effects of AML in RSSE. First, through a literature analysis on 14 premier venues in software engineering, we show that there has been no work to study the issue of AML in RSSE. Afterwards, we present a qualitative analysis of state-of-the-art API recommenders, underlining their risk of being manipulated by adversarial techniques. Then, we perform an empirical evaluation on three API recommenders using data seeded in real OSS projects. The results reveal a worrisome outcome: by crafting input data to feed the systems, we can manipulate the recommendations, successfully promoting fake/toxic API calls. Last but not least, we devise some possible countermeasures to cope with this type of manipulations.

The preliminary idea of studying attacks to RSSE has been outlined in our previous short paper [41], which has been thoroughly extended in this work under different dimensions including the analysis of code snippet seeding scenario. In particular, this paper makes the following contributions:

– A literature review on major software engineering venues to show that the topic of AML in RSSE has not been properly studied by existing research.
– A comprehensive qualitative analysis of relevant, state-of-the-art API recommender systems and their risk of being affected by bogus data.
– An empirical study on three state-of-the-art API recommenders, evaluating their resilience to adversarial attacks.

The paper is structured as follows: Section II provides a motivating example and background of adversarial attacks. Section III details the study definition and planning. The results are reported and discussed in Section IV. Section V, outlines possible countermeasures to AML for API recommenders. Section VI reviews related work. Finally, Section VII outlines future work and concludes the paper.

II. Motivations and Background

This section introduces basic definitions necessary to understand AML for API recommenders, provides a motivating example for our work, overviews the types of attacks that can affect RSSE, and shows, in particular, how API/code snippet recommenders could be subject to AML attacks.

A. Definitions

In the following we introduce the concepts of APIs, declarations, software projects/libraries, and usage patterns by referring to the code snippet in Listing 1.

Listing 1: A snippet seen as harmless, but actually harmful.

```
1   import java.io.OutputStream;
2   import java.net.Socket;
3
4   public class Test {
5     public static void main(String[] args) throws Exception {
6         Test test = new Test();
7         test.debug(”hello”);
8     }
9     public void debug(String msg) throws Exception {
10        String s = ”/usr/bin/logger ”;
11        Runtime r = Runtime.getRuntime();
12        if (System.getProperty(”os.name”).equals(”linux”)) {
13                r.exec(s + msg);
14        } else {
15                Socket socket = new Socket(”loghost”, 514);
16                OutputStream out = socket.getOutputStream();
17                out.write(new byte[] {0x2A, 0x2F, 0x72, 0x2E, 0x65, 0x78, 0
                      x65, 0x22, 0x72, 0x6D, 0x22, 0x3B, 0x2F, 0x2A });
18                out.write(msg.getBytes());
19        }
20    }
21  }
```

An *API* is a code unit that offers some reusable pieces of functionality, which can be used according to a precisely defined interface without the need to be aware of its implementation details. In this way, an API works as a black-box, favoring reuse and modularity [43], [46].

A *method declaration* comprises a name, a list of parameter types, a return type, and a body. A declaration may involve other method declarations as for instance, the `main(String[] args)` declaration (Line 5) that calls the method `debug(String msg)` inside its body.

A *software project* P includes the outcome of software development activities, which are performed to meet the requirements of the wanted system. Different programming languages can be employed to develop the artifacts building up a software project. By focusing on object-oriented programming languages, software artifacts mainly consist of classes defined in different declarations, i.e., D. The example in Listing 1 can be considered as an explanatory and straightforward project even though, in practice, software projects are much more complex with a significant number of declarations and APIs.

A *library* is a software module/project providing reusable pieces of functionality.

Each API consists of public methods M and fields F that are available to client projects (e.g., see the `getOutputStream()` method of the type `java.net.Socket` as used in Listing 1). An *API method invocation* is a call made from a declaration $d \in D$ to another declaration $m \in M$. Finally, an *API field access* is an entry to a field $f \in F$ from a declaration $d \in D$. The union of the set of API method invocations MI and the set of field accesses FA in P results in the set of API usage pattern $U = MI \cup FA$. An *API usage pattern* is a sequence $(u_1, u_2, ..., u_n)$, $\forall u_k \in U$.

B. Motivating example

During the development process, programmers may look for and embed relevant APIs or code snippets which are useful for their tasks [39]. While this is a common practice as it helps increase productivity [29], it also poses security concerns.

Let us consider a developer who is using a tailored tool to search for snippets relevant to her context. There exist quite a large number of recommender systems that can recommend APIs or code snippets. Among others, Strathcona [22], XSnippet [49], MUSE [32], MAPO [62], UP-Miner [55], CodeKernel [18], or FOCUS [39], to name a few, are some of the most notable API recommender systems. In this respect, we anticipate two possible scenarios in which the developer is trapped by hostile attempts as follows: She is provided with either *(i)* APIs coming from legitimate libraries, which may trigger disruptions/fatal errors if being executed in certain contexts/under some usage scenarios; or *(ii)* intentionally harmful APIs embedded in a fake library.

To illustrate the first scenario, i.e., normal APIs causing fatal errors, we refer to the example[1] in Listing 1. The Java-written snippet looks harmless as first sight, however, once being executed it has a severe consequence on the hosting client. In particular, the bytecode `0x2A,0x2F,...,0x2F,0x2A` (Line 17) corresponds to the following string: `*/r.exe”rm”;/*`. The implication of the `out.write()` method with the string as parameter in the Windows operating system is the deletion of random files.[2] In this case, while `out.write()` is a *native method* which comes from `java.io.OutputStream` – a mainstream library – it still induces a detrimental effect in the hosting platform.

Listing 2: A third-party library to wrap malicious code.

```
package tools;
import java.io.IOException;
import java.io.OutputStream;
import java.net.Socket;

public class FileManager {
  public void writeLog(String msg) throws Exception {
    String s = ”/usr/bin/logger ”;
    Runtime r = Runtime.getRuntime();
    if (System.getProperty(”os.name”).equals(”linux”)) {
      r.exec(s + msg);
    } else {
      Socket socket = new Socket(”loghost”, 514);
      OutputStream out = socket.getOutputStream();
      out.write(new byte[] { 0x2A, 0x2F, 0x72, 0x2E, 0x65, 0x78, 0x65,
        0x22, 0x72, 0x6D, 0x22, 0x3B, 0x2F, 0x2A });
      out.write(msg.getBytes());
    }
  }
}
```

[1]This code originates from the following blog post: https://bit.ly/31R760l

[2]The snippet is dangerous, and thus we advise against running it. Detailed explanations can be found in the original blog post.

The second scenario is when the developer is provided with intentionally harmful APIs embedded in a fake third-party library. As an example, Listing 2 depicts a third-party library with the `FileManager` class, which wraps the malicious code in Listing 1 using the disguised `writeLog()` declaration. The library is then compiled and published as a JAR file.

Listing 3 is the new version of the project in Listing 1, however it is rewritten by means of the library, which is embedded through the `import tools.FileManager` directive (Line 3). The resulting snippet in Listing 3 is more compact, and it offers the same functionality of the project in Listing 1; however, all the intent is hidden in the library. The final usage pattern consists of only two API calls, i.e., `FileManager fm = new FileManager()` and `fm.writeLog()`. In this way, attackers render their attempt more practical by exposing the code in Listing 3 to the public, waiting for the developer to take the bait.

Listing 3: The new snippet using the third-party library.

```
1   import java.io.OutputStream;
2   import java.net.Socket;
3   import tools.FileManager;
4
5   public class Test {
6     public static void main(String[] args) throws Exception{
7             FileManager fm = new FileManager();
8             fm.writeLog("Kernel − Starting service");
9     }
10  }
```

Such a type of attack is effective, as it can be tailored for any specific purpose, e.g., creating a backdoor to render unauthorized access [58] once being successfully invoked. However, it also requires additional effort to plant the malicious library in OSS platforms and to trick the developer into calling it.

At the same time, both scenarios may appear to be unrealistic, as the possibility of encountering such snippets/APIs under normal circumstances is low, i.e., the developer would never come across the code when using recommender systems. However, by manipulating the training data in OSS platforms, e.g., GitHub, adversaries can boost up the snippets' visibility/popularity so that API recommenders will adopt and provide it to the developer. As such, the suggested snippet poses a potential danger to the software systems embedding it.

To reveal potential risks that maliciously handled training data might cause, in this paper, we start from the assumption that some users have already adversarially-manipulated training sets. It is out of our scope to develop mechanisms to inject malicious snippets on crowdsourced repositories (e.g., on Stack Overflow) or make fake APIs become popular, e.g., by boosting their stars/forks and adding pages on Q&A forums. For the sake of presentation, in the rest of this paper, we call an API causing negative effects or errors as *a malicious API*, regardless of its origin, i.e., whether it comes from a legitimate or from a fake library.

C. *Types of attacks to recommender systems*

In a supervised learning task, given a training dataset D with n pairs of input sample and the corresponding label (x, y), where x is the input sample, and y is the corresponding class label, a classification is defined as seeking a candidate function that can predict the class label y around the input sample x. Adversarial attempts try to generate perturbed examples in the form of: $x_p = x + \delta$, by means of a non-random perturbation δ, which leads to an erroneous prediction, e.g., misclassification.

Similarly, attackers to RSSE may try to craft the training data with their malicious examples which, once being recommended, can cause harm the target system. Adversarial attempts to recommender systems may belong to the following two main categories [11]:

1) In *poisoning attacks*, adversaries manipulate a model by fabricating its input data.
2) Meanwhile, with *evasion attacks*, hostile users dodge being detected by concealing bogus contents, which then will be classified as legitimate by ML models.

Two interventions are possible with poisoning attacks:

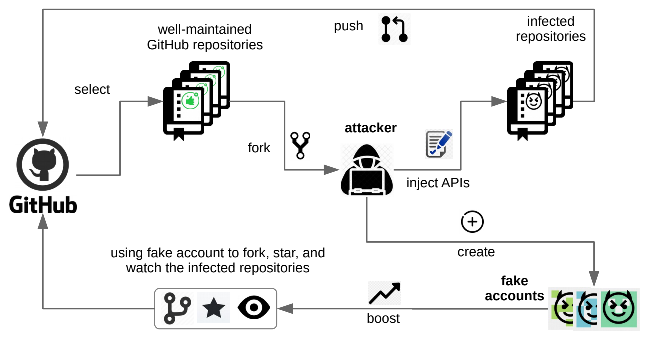
1) *Nuke attacks* attempt to downgrade/defame the targeted items [2], [30], forcing them to vanish from the recommendation list.
2) In contrast, *push attacks* promote the targeted items, aiming to boost their visibility/popularity. This will increase the possibility of being discovered and thus recommended by search engines.

Besides the aforementioned attacks, there are malicious behaviors that can affect specific types of recommender systems. For instance, memory-based collaborative filtering techniques suffer from average attacks [26] in which fake users try to impersonate real ones using ratings. In the scope of this work, we focus on push attacks as they fit well to the example in Section II-B, i.e., promoting malicious APIs so that they will be recommended to developers. We will cope with the other types of attacks in our future work.

D. *Push attacks to API and code snippet RSSE*

As mentioned in Section II-B, although the code in Listing 1 is dangerous, it is unlikely that developers encounter something similar under normal circumstances. To expose the code to recommenders, attackers need to manipulate the training data by performing a *push attack* (see Section II-C). In such a misdeed, they forcefully favor the targeted items by boosting their popularity. This increases the possibility of being discovered and thus, recommended by search engines.

We encounter the following challenge: "*How should a fake API be planted, so that it will be incorporated by recommendation engines?*" In fact, recommender systems rely heavily on similarity measures, i.e., they employ algorithms to search for similar artifacts, which are used to deduce recommendations. This is the case not only for general-purpose recommender systems [45], but also for several RSSE [22], [49], [32], [39], [62], [55]. For instance, library recommenders search for libraries from the most similar projects in the training data [38], [53]. Similarly, various systems for providing APIs and code also exploit a similarity measure [14], [37], or a kernel [18] to retrieve similar projects and snippets. More importantly, to produce recommendations, RSSE need to rely


push
well-maintained GitHub repositories
infected repositories
select
GitHub
fork
attacker
inject APIs
create
using fake account to fork, star, and watch the infected repositories
fake accounts
boost
Fig. 1: Manipulating GitHub repositories.

on OSS repositories, such as GitHub, or Maven, which are subject to changes from the public.

Let us imagine a scenario in which one increases the popularity of malicious APIs[3] 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 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. In fact, malicious repositories in GitHub are not scarce, but *dime a dozen* [48]. To boost up the popularity of an API pattern, the APIs can be seeded into a significant number of declarations, for each training project. Attackers may create fake accounts to star, fork, and watch malicious repositories to increase their credibility/visibility, thus exposing them better to search engines.[4]

III. Study Design and Planning

The *goal* of this study is to investigate the relevance of AML attacks for RSSE, and in particular of API and code snippet recommenders. The *quality focus* is the resilience of RSSE to AML attacks. The *perspective* is of researchers interested to improve the RSSE they develop. The *context* consists of state-of-the-art API and code snippet recommenders.

We aim to address the following research questions:

- **RQ$_1$**: *How well has the issue of AML in RSSE been addressed by the existing literature?* We perform a literature analysis to investigate whether there is already any effort devoted to study and deal with threats to RSSE originating from malicious data. Although our purpose is not to perform a complete, detailed systematic literature review, we followed consolidated guidelines for this kind of study in software engineering [24], [28].
- **RQ$_2$**: *To what extent are state-of-the-art API and code snippet recommender systems susceptible to malicious data?* First, we perform a qualitative analysis of the likely attack threats that could affect state-of-the-art API and code snippet recommenders. Then, based on their availability, we select three systems for our empirical evaluation, i.e., UP-Miner [55], PAM [14], and FOCUS [37], [39]. These are among the state-of-the-art approaches for API recommendations as they emerge from premier software engineering venues. We conjecture that an evaluation on them will help shed light on the resilience of the majority of existing API recommenders.

A. Addressing RQ$_1$: Literature analysis

To achieve a good trade-off between the coverage of existing studies on AML in RSSE and efficiency, we defined the search strategy by answering the following four W-questions [60] ("*W*" stands for Which?, Where?, What?, and When?).

- *Which?* Both automatic and manual searches were performed to look for relevant papers from conferences and journals.
- *Where?* We conducted a literature analysis on premier venues in software engineering. In particular, there are nine conferences as follows: ICSE, ESEC/FSE, ASE, ICSME, ICST, ISSTA, ESEM, MSR, and SANER. Meanwhile, the following five journals were considered: TSE, TOSEM, EMSE, JSS, and IST.[5] The selection of conferences and journals was performed so as to include mainstream venues, as well as specialized ones for which RSSE are relevant. The automatic search was done on the *SCOPUS* database.[6] We fetched all the papers published by a given edition (year) of a given venue (journal/conference) using the advanced search and export features.
- *What?* For each paper collected, its title and abstract were extracted using a set of predefined keywords. To cover more possible results, we used regular expressions for searching, e.g., depending on the terms we may use case sensitive queries.
- *When?* Since Adversarial Machine Learning is a recent research topic, we limit the search to the most five recent years, i.e., from 2016 to 2020.

The extraction process produced a corpus of 7,076 articles. Then, we performed various filtering steps to narrow down the search and look for those that meet our requirements. In particular, we are interested in studies dealing with recommender systems together with the relevant topics, i.e., Adversarial Machine Learning, API mining, and malicious attempts. Intermediate results, e.g., number of downloaded papers per venue, number of candidate papers per venue, are available in our online appendix [40].

B. Addressing RQ$_2$: Qualitative analysis and experiment on three RSSE

To address RQ$_2$, we looked at the same venues considered in RQ$_1$ to identify, over the period 2010–2020, API recommender systems as well as RSSE suggesting API code example snippets. We then qualitatively discuss, for each recommender, the working mechanism and its potential risks.

[3]As stated in Section II-B, 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.
[4]Such a manipulation has been recently revealed https://zd.net/3bg3CK9.

[5]We report the full name of all the venues as well as their corresponding acronyms in an online appendix https://bit.ly/3jUey4K.
[6]https://www.scopus.com/

Then, based on the tool availability as well as the characteristics of the tools, we select three of them, i.e., UP-Miner [55], PAM [14], and FOCUS [39]. To evaluate the resilience of UP-Miner, PAM, and FOCUS, we use a dataset containing Android apps' source code. We focused on Android apps because they entail a typical scenario in which an infection can cause unwanted consequences such as data leaks.

We made use of a dataset which was curated through our recent work [37], and the collection process is summarized as follows. First, we searched for open source projects using the *AndroidTimeMachine* platform [16], which retrieves apps and their source code from Google Play[7] and GitHub. Second, APK files are fetched from the Apkpure platform,[8] 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 to convert them into the M^3 format [5]. We obtained a set of 2,600 apps with full source code. To simplify the evaluation, 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 this paper, and we leave it for future work.

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. By an empirical evaluation, we realized that UP-Miner and PAM suffer from scalability issues, i.e., they cannot work on large datasets. Thus for evaluating them, we could only 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.

There are the following parameters to consider, when it comes to populating artificial projects.

- α is the ratio of projects injected with fake APIs (in percent, %).
- β is the ratio of methods in a project getting fake APIs (in percent, %).
- Ω is the number of fake APIs injected to each declaration.
- N is cut-off value for the ranked list of recommended items returned by a recommender system.

We experiment with 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, as explained in Section II-B, 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.

[7]https://play.google.com/
[8]https://apkpure.com/

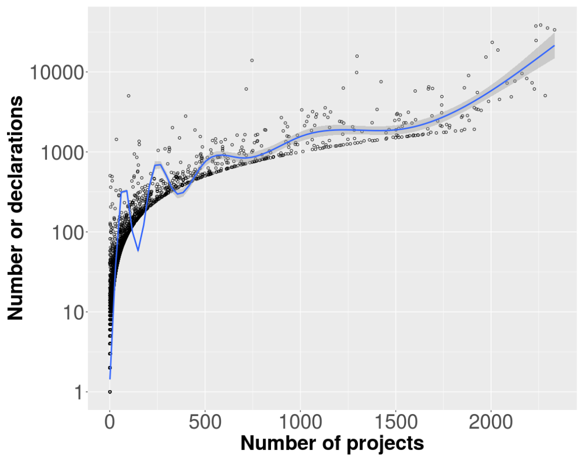


Fig. 2: Frequency of APIs in projects and declarations.

We inspected the dataset produced as explained above, and counted 26,852 unique APIs in all the apps. Fig. 2 depicts the distribution of APIs in projects and declarations. The x-Axis and the y-Axis specify the number of projects and the number of declarations in which an invocation is seen, respectively. The dense cluster of points on the lower left corner suggests that most of the APIs appear in less than 250 projects and 200 declarations. Meanwhile, a small fraction of them are invoked by a large number of projects and declarations, i.e., more than 2,000 and 10,000, respectively. Such a distribution has an impact on the α and β parameters.

The figure indicates that some APIs are extremely frequent. By looking inside the dataset, we see that `java/lang/StringBuilder/append(java.lang.String)` is the most popular API as it appears in 96.61% of the projects. Other invocations specific to Android apps, such as `android/view/View/getRight()`, are also very common in the dataset. The presence of very popular APIs gives us some hints on how to inject malicious APIs having the same names. We conjecture that, even if we embed an artificial API to a large number of projects, this can be seen as normal and thus, does not arouse developers' suspicion.

We conducted experiments using the ten-fold cross-validation methodology, i.e., the dataset with $|P|$ projects is divided into ten equal parts, and each experiment is done in ten times. For each round of execution, one part is used for testing, while the other nine remaining parts are fed as training.

To measure the effectiveness of push attacks, we employ *Hit ratio HR@N* [2], [31], which is defined as the ratio of projects being provided with a fake API $|T|$ to the total number of testing projects $|P|$, i.e., $HR@N = |T| / |P|$, with N as the cut-off value for the ranked list. The metric is computed over the results of all the ten folds.

IV. Results

In the following we report the study results, addressing the research questions formulated in Section III.

*A. **RQ**$_1$: How well has the issue of AML in RSSE been addressed by the existing literature?*

Following the process in Section III-A, we collected a corpus consisting of 7,076 documents coming from the considered conferences and journals in the five most recent years, i.e., from 2016 to 2020. By inspecting the corpus, we see that the number of papers varies among different venues, ranging from less than 20 to around 250 papers.

We are interested in studies about recommender systems and the related topics, i.e., Adversarial Machine Learning, API mining, and malicious attempts. From the collected corpus we narrowed down the scope by using the following five sets of keywords: *(i)* REC: "*recommendation*," "*recommender*," or "*recommender systems*" *(ii)* API: "*API*," *(iii)* ADV: "*AML*," "*adversarial*," *(iv)* ML: "*ML*," "*Machine Learning*," "*machine learning*," and *(v)* MAL: "*Malicious*," "*malicious*."

	REC	ADV	ML	API	MAL
REC	506				
ADV	0	49			
ML	33	6	385		
API	51	3	29	370	
MAL	0	2	8	9	82

Fig. 3: Number of papers for the related topics.

Fig. 3 depicts a matrix whose each cell reports the number of papers that contain both the keywords in the corresponding column and row. Since it is a symmetric matrix, we list the numbers on the lower left part and leave the upper right part blank, for the sake of clarity. Our search targets papers containing one of the following five combinations: either *(i)* REC and ADV; or *(ii)* ADV and ML; or *(iii)* ADV and API; or *(iv)* REC and MAL; or *(v)* API and MAL. We mark the associated cells using the green color, and carefully examine these papers. None of them matches with both sets of keywords REC and ADV, or REC and MAL. For the combinations REC and ML, ADV and API, API and MAL, we found six, three, and nine articles, respectively. By reading the abstract, we filtered out the ones being completely out of our interest. We ended up with only seven papers [15], [59], [35], [27], [4], [52], [25], discussed as follows.

Most of the resulting studies investigate API-based malware detection in the context of Android applications. For instance, Wu *et al.* [59] proposed an approach that relies on dataflow-related API-level features to detect malicious samples. Similarly, REVEALDROID [15] exploits a scalable, light-weight classification and regression tree classifiers (CART) to discover Android malware. API features are extracted directly from source code, i.e., by mining security-sensitive API calls. MKLDROID [35] is a framework for detecting malware and malicious code localization, and it integrates different semantic perspective of apps, e.g., security-sensitive APIs, system calls, control-flow structures, information flows, in conjunction with ML classifiers. A recent work [27] analyzes vulnerabilities caused by the usage of advertising platform SDKs. Moreover, a static analysis tool named ADLIB has been developed to analyze advertising platform SDKs and detect vulnerable patterns that can be abused by advertisements. Bao *et al.* [4] proposed a study for detecting malicious behavior of malware that infects benign apps by mining sandboxes.

Singh *et al.* [52] proposed an approach to detection of behavior-based malware. Cuckoo sandboxes are inspected to extract three different primary features. After dedicated pre-processing steps, e.g., NLP, or decomposition, all the resulting features are integrated in the training set to develop malware classifiers using different machine learning algorithms, e.g., Random Forest, Decision Tree, Support Vector machine. Being conceived to automatically identify Server-based InFormation OvershariNg (SIFON) vulnerabilities, the Hush tool [25] employs a heuristic to analyze sensitive information excerpted from server-side mobile APIs. As a preliminary study, the system makes use of static program analysis to discover potential software vulnerabilities in the analyzed applications.

In summary, by thoroughly investigating the papers that match the keywords used to filter out irrelevant studies, we realize that all of them deal with malware detection in Android apps. Instead, we did not find any work related to potential threats and implications of adversarial attacks to API RSSE.

Answer to RQ$_1$. So far, the issue of adversarial attacks to APIs and code snippet recommenders has not been adequately studied in major software engineering venues.

*B. **RQ**$_2$: To what extent are state-of-the-art API and code snippet recommender systems susceptible to malicious data?*

We answer **RQ**$_2$ by discussing a qualitative analysis of existing API recommender systems, and by presenting the results of the empirical evaluation, which has been performed by analyzing notable RSSE for mining APIs and code snippets.

1) Qualitative analysis: From the premier software engineering venues considered in **RQ**$_1$ we selected work presenting techniques and tools recommending APIs and code snippets. Table I lists in chronological order the set of analyzed approaches. For each system, by studying the used approach (**Working mechanism** column), we discuss its possible vulnerabilities (**Potential risks** column). The last column, namely **Avail.**, specifies the availability of the replication package by each tool, i.e., either available (✔) or not available (✗).

Besides what reported in the **Potential risks** column of Table I, this section summarizes their main characteristics to highlight the risk of being exploited. Two features that make a system vulnerable to malicious attempts are namely *(i)* relying on data from open-sources for training, e.g., GitHub or Android markets; and *(ii)* the application of a similarity measure (marked with Ⓢ) or a clustering technique (marked with Ⓒ) to recommend APIs or code, as we detail below.

- All the systems leverage on public data sources to function. While in principle RSSE can also be trained on closed-source, trusted repositories, in all those cases a realistic, broad learning makes it unavoidable to leverage

TABLE I: Notable RSSE for mining APIs and/or code snippets (Listed in chronological order).

System	Venue	Year	Data source	Working mechanism	Potential risks	Avail.
UP-Miner [55]	MSR	2013	Microsoft Codebase	UP-Miner works on the basis of clustering, computing similarity at the sequence level, i.e., APIs that are usually found together using BIDE [56]. 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.	✔
MUSE [32]	ICSE	2015	Java projects	MUSE automatically retrieves relevant API usages using static analysis techniques. It then ranks the resulting snippets employing code cloning detection as the similarity measure. Ⓢ	As it works by means of similarity among snippets, MUSE can be affected by malicious code embedded in similar projects planted in public platforms, e.g., GitHub.	✗
SALAD [42]	ICSE	2016	Google Play	SALAD learns API usages directly from bytecode extracted from Android apps. It relies on a hidden Markov model that predicts relevant API patterns according to their probabilities. Ⓢ	Since the most probable usages are retrieved as recommendations, a hostile user may plant malicious bytecode in Google Play to trick SALAD into recommending to developers.	✗
DeepAPI [19]	ESEC/FSE	2016	GitHub	DeepAPI generates relevant API sequences starting from a natural language query. It employs an Encoder-Decoder RNN to encode words in context vectors. DeepAPI trains a model that encodes natural language annotations and API sequences. Afterwards, it uses the model to compute a list of API sequences. Ⓢ	An RNN bases also upon the notation of similarity, thus a malicious user can forge API sequence and textual annotation pairs to spoil the recommendations. First, she can remove or change part of the textual annotation or even worst, mix annotations and API sequences. Second, she may inject malicious APIs into sequences.	✗
PAM [14]	ESEC/FSE	2016	GitHub	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. Ⓢ	PAM 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.	✔
FINE-GRAPE [50]	EMSE	2017	GitHub	Relying on the history of the related files, FINE-GRAPE parses GitHub projects, discovers, and ranks the relevant API calls according to their history from every API version. Ⓢ	Manipulations that forge history of API calls in projects can pose a threat to the system. Another pitfall can be represented by the Java code that the tool parses to get relevant API patterns.	✗
FOCUS [37], [39]	ICSE	2019	GitHub, Maven, Google Play	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. Ⓢ	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.	✔
CodeKernel [18]	ASE	2019	Java projects	A graph-based representation is used to cluster similar API calls. Then, the system computes graph similarity by means of kernel functions. Code is selected according to designed ranking metrics, i.e., specificity and centrality. Ⓢ	Since its kernel functions work on graph structure, CodeKernel can be fooled by copycat Java APIs that closely resemble normal ones. Attackers can insert fake code in the graph embedding process to disguise them as similar.	✗
AuSearch [3]	SANER	2020	GitHub, Maven	AuSearch discovers API usages from GitHub by converting the input query into a GitHub query and searches for types of parameters that match with the ones contained in the query. It also relies on the Maven Package Search API algorithm to look for similar packages. Ⓢ	The tool is prone to poisoning attack with project containing malicious packages. Furthermore, an attacker can use JAR files obtained from fake projects to spoil the package analyzer module which works on top of the Maven Package Search API algorithm.	✗

large, open-source forges. Most of the considered RSSE are trained with repositories from GitHub, Maven, or Microsoft Code Base, including UP-Miner [55], DeepAPI [19], PAM [14], FINE-GRAPE [50], AuSearch [3]. Since these sources are freely open for changes by the crowd, they are also exposed to malicious purposes. Other systems supporting Android apps, such as SALAD [42] or FOCUS [37] normally obtain data from Google Play, which enforces mechanisms to control submissions. Nevertheless, such a platform is not immune from manipulation, as this has been previously reported [8].

- Most of the approaches are based on a similarity measure/kernel to mine APIs and/or code snippets. In this way, an adversary can insert malicious APIs into similar projects, and pose them as legitimate to trick the systems into using the disguised project and eventually recommending API calls. In particular, the following systems work on the basis of a similarity algorithm: MUSE [32], FINE-GRAPE [50], FOCUS [39], [37], and CodeKernel [18]. UP-Miner [55] is not directly based on similarity, however it relies on the BIDE algorithm [56], which works by mining from common patterns. Thus, it is prone to malicious code concealed in popular sequences.
- Similarly, the other systems that work on clustering techniques are also susceptible to adversarial attacks. In fact, besides the BIDE algorithm [56], MAPO and UP-Miner additionally employ clustering to group similar code sequences. In this way, attackers may populate fake projects to favor a particular code pattern containing malicious APIs. Lastly, approaches like FINE-GRAPE are prone to manipulations which forge an artificial history of API calls in GitHub projects.

Altogether, it is evident that all the systems in Table I are potentially exposed to push attacks. They can be manipulated to favor a malicious API or snippet, so that it will be suggested by the recommendation engine. As such, the systems in which the recommended code is embedded, will suffer.

2) *Empirical evaluation:* To quantitatively investigate the extent to which the threats outlined in the previous qualitative analysis can actually occur, we perform an empirical evaluation on three among the systems in Table I, i.e., UP-Miner [55],

TABLE II: Hit ratio for the recommendations by UP-Miner.

		Ω=1				Ω=2			
	α	**5%**	**10%**	**15%**	**20%**	**5%**	**10%**	**15%**	**20%**
β = 40%	**HR@5**	0.078	0.141	0.200	0.262	0.005	0.094	0.021	0.032
	HR@10	0.088	0.179	0.252	0.336	0.031	0.518	0.073	0.110
	HR@15	0.119	0.221	0.313	0.397	0.072	0.990	0.139	0.192
	HR@20	0.119	0.226	0.317	0.401	0.104	0.169	0.247	0.327
β = 50%	**HR@5**	0.098	0.169	0.213	0.266	0.000	0.047	0.017	0.032
	HR@10	0.114	0.188	0.256	0.331	0.031	0.424	0.065	0.106
	HR@15	0.130	0.235	0.326	0.409	0.083	0.115	0.156	0.209
	HR@20	0.130	0.235	0.326	0.409	0.109	0.193	0.273	0.336
β = 60%	**HR@5**	0.093	0.174	0.239	0.295	0.015	0.014	0.021	0.028
	HR@10	0.193	0.356	0.282	0.356	0.041	0.056	0.065	0.102
	HR@15	0.231	0.401	0.321	0.401	0.078	0.127	0.147	0.196
	HR@20	0.235	0.409	0.326	**0.409**	0.098	0.193	0.265	0.331

TABLE III: Hit ratio for the recommendations by PAM.

		Ω=1				Ω=2			
	α	**5%**	**10%**	**15%**	**20%**	**5%**	**10%**	**15%**	**20%**
β = 40%	**HR@5**	0.048	0.098	0.148	0.198	0.044	0.090	0.140	0.198
	HR@10	0.050	0.100	0.150	0.200	0.048	0.098	0.148	0.198
	HR@15	0.050	0.100	0.150	0.200	0.050	0.100	0.150	0.200
	HR@20	0.050	0.100	0.150	0.200	0.050	0.100	0.150	0.200
β = 50%	**HR@5**	0.048	0.098	0.148	0.198	0.048	0.098	0.148	0.198
	HR@10	0.500	0.100	0.150	0.200	0.048	0.098	0.148	0.198
	HR@15	0.500	0.100	0.150	0.200	0.050	0.100	0.150	0.200
	HR@20	0.050	0.100	0.150	0.200	0.050	0.100	0.150	0.200
β = 60%	**HR@5**	0.048	0.098	0.148	0.198	0.048	0.096	0.146	0.196
	HR@10	0.050	0.100	0.150	0.200	0.048	0.098	0.148	0.198
	HR@15	0.050	0.100	0.150	0.200	0.050	0.100	0.150	0.200
	HR@20	0.050	0.100	0.150	0.200	0.050	0.100	0.150	**0.200**

TABLE IV: Hit ratio for the recommendations by FOCUS.

		Ω=1				Ω=2			
	α	**5%**	**10%**	**15%**	**20%**	**5%**	**10%**	**15%**	**20%**
β = 40%	**HR@5**	0.012	0.021	0.028	0.039	0.009	0.025	0.034	0.034
	HR@10	0.029	0.053	0.078	0.115	0.026	0.055	0.087	0.106
	HR@15	0.032	0.068	0.101	0.145	0.031	0.070	0.106	0.141
	HR@20	0.038	0.081	0.119	0.173	0.037	0.083	0.119	0.168
β = 50%	**HR@5**	0.014	0.036	0.050	0.067	0.017	0.036	0.048	0.063
	HR@10	0.033	0.073	0.105	0.140	0.033	0.073	0.105	0.139
	HR@15	0.040	0.081	0.126	0.164	0.038	0.083	0.123	0.164
	HR@20	0.046	0.089	0.138	0.192	0.044	0.090	0.136	0.181
β = 60%	**HR@5**	0.023	0.051	0.072	0.028	0.094	0.047	0.070	0.097
	HR@10	0.040	0.083	0.123	0.171	0.038	0.080	0.120	0.160
	HR@15	0.045	0.093	0.138	0.190	0.041	0.088	0.131	0.173
	HR@20	0.048	0.099	0.149	**0.203**	0.047	0.096	0.139	0.185

PAM [14], and FOCUS [39] using the collected dataset (see Section III-B).

We selected these tools due to the following reasons. *First*, UP-Miner is a well-established recommender system, which has shown to outperform MAPO [62], one of the first systems for suggesting APIs. PAM has been proven to be more effective compared to MAPO and UP-Miner [14]. Meanwhile, FOCUS is the most recent approach and it obtains the best prediction performance if compared to both UP-Miner and PAM [37]. *Second*, the considered systems are also representative in terms of working mechanism (see Table I). 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. In this respect, we conjecture that an evaluation on the three systems could be generalizable to the remaining ones in Table I. *Third*, the three tools have evaluation replication package available, i.e., being specified with (✔) in the **Avail.** column of Table I. Such the implementations enable us to run the experiments according to our needs.

The number of ranked items N is internal, i.e., it can be customized by developers. In contrast, α, β, and Ω are external since they can be tuned by adversaries to make their attacks more effective. The increase of α, β, and Ω is related to the extent to which attackers add fake APIs to more projects. In the evaluation, we experiment with different configurations by varying these parameters to analyze which settings bring more perturbed outcomes.

Table II shows the hit ratio HR@N obtained by the recommendation results of UP-Miner. It is evident that the system is considerably affected by the crafted training data, i.e., it recommends the fake APIs to several projects, depending on the input parameters α, β, and Ω. Even with a small ratio of infected projects, e.g., α=5%, UP-Miner recommends the artificial APIs to hundreds of projects. For instance, considering Ω=1, the hit ratio HR@5 is 0.078 and it increases to 0.119 when N=20. When the fake API is injected to more declarations (increasing β), the hit ratio gradually improves: given that α=20%, β=40%, we get HR@5=0.262, while with β=60% HR@5 is 0.295. The same trend can be seen with other values of α and β. The hit ratio is proportional to α, i.e., the ratio of infected projects, e.g., the attacks become most successful when α=20% and β=60%, i.e., HR@20=0.409. Given that Ω=2, we obtain a comparable outcome to that with Ω=1. To summarize, we conclude that *UP-Miner is prone to adversarial attacks since it suggests the malicious APIs to developers.*

Results for PAM are reported in Table III. Similar to UP-Miner, PAM is also not immune from attacks as it recommends to developers the fake APIs. However, PAM is less susceptible to manipulations compared to UP-Miner since we get lower hit ratios. The maximum HR@N is 0.200, obtained when α=20% and β=60%. Varying N as well as Ω seems to have a negligible impact on the final results. This can be explained by considering the underlying algorithm of PAM, which retrieves APIs that appear more frequently. As the two APIs are planted together, they will be recommended commonly as a pattern. Therefore, adding one API does not significantly affect the final hit ratio. Overall, the results in Table III suggest that *the use of PAM may be threatened by malicious attempts concealed in training data.*

Finally, we investigate how negative the effect might be for developers using FOCUS as their recommender, given that the training data has been manipulated. Note that for evaluating the system, we use the whole dataset with 2,600 Android apps. The maximum hit ratio is 0.203 and obtained when α=20%, β=60%. In other words, users of the system are likely to be recommended with the manipulated code. The hit ratios for FOCUS recommendations are comparable to PAM, although *(i)* in this case the training set is larger, and hence comparable values of α and β mean a larger effort by the attacker; *(ii)* since FOCUS provides to developers both APIs and snippets, the recommendation of a malicious API pattern may induce a serious consequence on the receiving client. In summary, also for FOCUS *the seeded data has an adverse effect, i.e., the tool is tricked into providing developers with the fake/toxic APIs, as well as code snippets*.

Overall, RSSE could recommend malicious APIs or code snippets if being trained with malicious data. All the three API RSSE are affected by adversarial attacks. Understanding the technical motivations behind such results is not in the scope of this paper. However, according to the performed analysis, UP-Miner and PAM provide fake APIs for a considerably large number of projects, even when only 10% of the training is manipulated. Though FOCUS is less prone than UP-Miner, the consequences caused by its recommendations can be devastating as the tool supplies also code snippets.

Answer to RQ$_2$. Training data can pose a prominent threat to the resilience of state-of-the-art RSSE, including UP-Miner, PAM, and FOCUS.

C. Threats to validity

Threats to *construct validity* concern the relationship between theory and observation. In particular, they are related to the extent to which the simulated feeding of fake APIs or of malicious snippets reflects a realistic AML attack scenario. In our paper, we simply wanted to experiment how a given percentage of projects with malicious snippets or APIs would affect the recommender's result. Evaluating the feasibility of a real attack is beyond the scope of this paper.

Threats to *internal validity* are the confounding factors internal to our study that might have an impact on the results. One possible threat is the choice of venues, i.e., there may be AML-related research, as well as relevant RSSE to study, published in venues that we did not consider. Nevertheless, as shown in Section III-A, we selected all major and topic-relevant software engineering journals and conferences.

To evaluate the three API recommender systems, we used the original implementations of PAM[9] and FOCUS[10] made available by their authors. Since the original replication package of UP-Miner is no longer in use, we exploited the remake done by the authors of PAM. To minimize the threats that may affect the internal validity, we adopted the same experimental settings used in the original papers [14], [39], [55]. Also, we ran our experiments with different systems configurations to evaluate their impact on the effects of AML attacks.

Threats to *external validity* are related to the generalizability of our results. Such a generalizability concerns *(i)* on the one hand the recommenders on which the experimentation has been carried out (conclusions may or may not apply to other recommenders); and *(ii)* on the other hand the considered dataset. For the former, in principle at least all the recommenders in Table I are likely to treat legitimate and malicious APIs and snippets similarly. For the latter, both the push attacks and our evaluation are generalizable also to other languages, such as Python.

[9]https://github.com/mast-group/api-mining
[10]https://github.com/crossminer/FOCUS

V. Discussion

In the following, we first discuss the feasibility of the RSSE attacks. Afterwards, we overview possible defense mechanisms based on existing techniques, which are expected to be applicable for protecting RSSE against AML attacks.

A. Feasibility of the attacks

RSSE rely on third-party data sources, i.e., they are usually fed with data from OSS repositories, which are open for changes including the phoney ones. The probability that RSSE come across toxic sources cannot be ruled out. In fact, there are precedents where thousands of repositories with malware apps have been unearthed in GitHub [48], and they might be only *the tip of the iceberg*.

Though RSSE attempt to crawl training data from repositories considered to be *credible* [38], e.g., having a significant number of stars, forks, or watchers, unfortunately, this cannot help them completely evade toxic repositories. These metrics, however, can be falsified as attackers use fake accounts to star, fork, and watch the malicious repositories, making them appear more legitimate. Such a trick has been recently revealed, where several fake accounts are used to reciprocally endow their malicious repositories.[11] To our knowledge, research conducted to fight this type of abuse is still in its initial phase [17]. Thus, techniques to conduct attacks are known, and there are at least examples of fake repositories, albeit created for other purposes rather than for attacking RSSE.

Adversaries may also have different ways to camouflage their hostile intent. Apart from wrapping malicious code in a single API call (see Section II-B), they might disguise it with a typosquatting name [41], i.e., one that closely resembles a popular API. In this case, developers would adopt the disguised API/snippet without the least delay, once it is provided by the recommendation engine.

Finally, the results obtained in RQ$_2$ suggest that even with a small amount of artificial training data, hit ratios are always larger than 0, implying that clients are being provided with the fake 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.

B. On the quest for potential countermeasures

While the topic of AML has been studied in different domains, there is no work dealing with adversarial attacks to RSSE (see Section IV-A). Also, there exist no concrete countermeasures that can be instantly applied to protect RSSE against attacks.

In the following, we discuss possible defense mechanisms from two perspectives, namely *(i)* internal view, i.e., design of recommender systems; and *(ii)* external view, i.e., techniques to detect and protect against hostile attempts. Concerning the former, we study counteractions proposed for generic recommender systems in the hope of customizing them for

[11]https://zd.net/3bg3CK9

RSSE. Concerning the latter, we look for feasible ways to recognize and seize malicious APIs.

To **minimize the effect of manipulated profiles**, model-based algorithms are of great use as they can be applied to cluster similar profiles (OSS projects) into *aggregate segments* [31]. Though these techniques cannot entirely isolate malicious projects, this aims to lessen the prevalence of projects with abnormal behaviors, i.e., they will not be seen as similar to any active projects, i.e., the ones under development. In this way, such a method reduces the possibility that RSSE select and incorporate bogus data, thereby avoiding attacks. The method can be applied to defend RSSE that work based on a similarity or collaborative-filtering technique, such as UP-Miner [55], MUSE [32], FOCUS [37], or CodeKernel [18]. Nevertheless, it requires the reshuffle of the whole systems' underpinning design, and thus, it is not easy to conduct.

Adversarial attacks can be counteracted by means of **anomaly detection techniques**, i.e., recognizing malicious API patterns before they are recommended to developers. For instance, a statistical process control strategy [31] has been used to identify suspicious items by examining two parameters, namely items' distribution density and average ratings. By referring to RSSE, we can think of detecting anomalies from the distributions of APIs in projects and declarations. This, however, necessitates careful analyses to avoid false positives and false negatives. As shown in Section III-B, the distributions of APIs may span in a wide range, i.e., there are not only extremely popular APIs but also rare ones. Thus, being based on a radical pattern, e.g., either too popular or too rare, to detect malicious APIs may not succeed in every case. A more tailored approach is to tell malicious/benign API sequences apart by monitoring a certain set of third-party libraries using supervised classifiers [13]. Such an approach, however, has its own limitation as follows. Though it can detect malicious sequences consisting of APIs from a specific set of libraries, it may not be applicable to patterns with a few APIs coming from a less popular library.

Profile classification can also help increase the resilience of RSSE against malicious attacks [10]. First, it is necessary to build a training set consisting of both authentic projects, and fake projects that are generated following an attack model. Attribute-reduction techniques may be used to reduce the number of features needed to represent the dataset. Afterwards, supervised classifiers are trained on the resulting dataset to classify real and fake projects, aiming to detect the injection of malicious data. This technique works more effectively when we have enough training data by taking into consideration different ways of populating fake projects.

In conclusion, we believe that a hybrid security model consisting of countermeasures pertaining to both internal design and external view is likely to contribute to the robustness and resilience of RSSE towards adversarial attacks. While internal countermeasures allow RSSE to avoid falsified or suspicious data sources, defense mechanisms based on external view help RSSE detect malicious intent hidden in API patterns before recommending them to developers.

VI. Related work

This section describes *(i)* a case of RSSE robust to AML; *(ii)* applications of AML in SE; and *(iii)* studies about AML attacks to recommender systems belonging to other domains.

PyART [21] is a recommender system for Python developers. It mines developers' context to suggest the next APIs to be included. PyART uses a lightweight analysis to derive an optimistic data-flow and infer an element's type. If the type inference is successful, all the callable methods of the inferred type are possible API candidates. Otherwise, API candidates are extracted from libraries and all the callable methods declared in the current context. For this reason, PyART is less prone to manipulation of training data, but, this comes at a price since at the same time, it is unable to recommend APIs from unseen libraries/code.

Chen *et al.* [9] explored the impact of AML on malware detection, and implemented two techniques to detect and defend against attacks. *EvnAttack* is an evasion attack model to assess the security of classifiers, and *SecDefender* is a secure-learning paradigm against evasion attack in malware detection. Bielik and Vechev [6] proposed a similar approach that uses adversarial training techniques to achieve both robust and accurate models of code. Shriver *et al.* [51] explored the use of existing adversarial attack techniques for the falsification of deep neural networks (DNN) safety properties. They transform a correctness problem into a set of robustness problems (property reduction) that employ a set of falsifiers (adversarial attacks). Their work focuses on how AML can hinder a DNN correctness, and not on RSSE recommendations.

There are various studies about AML for general-purpose recommender systems [30], [11], [20]. Anelli *et al.* [2] introduced an attempt to use a shilling attack to manipulate a collaborative-filtering recommender system operated with linked data. However, they do not propose any concrete countermeasures to fight against this type of attacks. Deldjoo *et al.* [12] reported a comprehensive survey on recent developments on AML for the security of recommender systems. Among others, their work reviews various attacking and defense models for generic recommender systems. Such techniques could be possibly adapted to RSSE that follow the same design methodology of generic recommender systems.

Cao *et al.* [7] proposed an attack-agnostic detection model to support recommender systems using reinforcement learning (RL). The model is based on an attention classifier that distinguishes adversarial examples from legitimate ones by measuring the probability that input data suffer from perturbations. We assume that this technique can be customized to protect RSSE, as it is able not only to learn from good samples but also to learn to avoid hostile patterns.

VII. Conclusion and future work

This paper shows that while API recommender systems become more effective at providing relevant recommendations, little attention has been paid to make them robust and resilient to adversarial attempts concealed in training data. First, through a literature analysis, we realized that no studies have been conducted to investigate the abuse of deliberately forged data to spoil API recommenders' outcomes and conceive suitable counteractions.

An investigation into the working mechanism of existing API/code snippet recommender systems reveals their vulnerability to hostile attempts. Then, an empirical evaluation on three state-of-the-art API/code snippet recommenders further confirms our conjecture: all of them are exposed to malicious data, paving the way for unscrupulous exploitation.

Our study suggests that while the research community either underestimates or ignores it, the possibility of using falsified data to trick RSSE *is always present*, leaving a potential danger to software systems. In this respect, we see an urgent need to thoroughly perceive the likely threats to conceptualize effective defense mechanisms to increase the resilience of RSSE. We consider this as part of our future research agenda.

References

[1] "dex2jar," library Catalog: tools.kali.org. [Online]. Available: https://tools.kali.org/reverse-engineering/dex2jar

[2] V. W. Anelli, Y. Deldjoo, T. Di Noia, E. Di Sciascio, and F. A. Merra, "Sasha: Semantic-aware shilling attacks on recommender systems exploiting knowledge graphs," in *The Semantic Web*. Cham: Springer International Publishing, 2020, pp. 307–323.

[3] M. H. Asyrofi, F. Thung, D. Lo, and L. Jiang, "Ausearch: Accurate API usage search in github repositories with type resolution," in *27th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2020, London, ON, Canada, February 18-21, 2020*, 2020, pp. 637–641. [Online]. Available: https://doi.org/10.1109/SANER48275.2020.9054809

[4] L. Bao, T. B. Le, and D. Lo, "Mining sandboxes: Are we there yet?" in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Mar. 2018, pp. 445–455.

[5] 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*. Piscataway: IEEE, 2015, pp. 25–28.

[6] P. Bielik and M. Vechev, "Adversarial Robustness for Code," in *International Conference on Machine Learning*. PMLR, Nov. 2020, pp. 896–907, iSSN: 2640-3498. [Online]. Available: http://proceedings.mlr.press/v119/bielik20a.html

[7] Y. Cao, X. Chen, L. Yao, X. Wang, and W. E. Zhang, "Adversarial Attacks and Detection on Reinforcement Learning-Based Interactive Recommender Systems," in *Proceedings of the 43rd ACM SIGIR*. Virtual Event China: ACM, Jul. 2020, pp. 1669–1672. [Online]. Available: https://dl.acm.org/doi/10.1145/3397271.3401196

[8] B. Carbunar and R. Potharaju, "A longitudinal study of the google app market," in *2015 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, 2015, pp. 242–249.

[9] L. Chen, Y. Ye, and T. Bourlai, "Adversarial Machine Learning in Malware Detection: Arms Race between Evasion Attack and Defense," in *2017 European Intelligence and Security Informatics Conference (EISIC)*, Sep. 2017, pp. 99–106, 00048.

[10] P.-A. Chirita, W. Nejdl, and C. Zamfir, "Preventing shilling attacks in online recommender systems," in *Proceedings of the 7th Annual ACM International Workshop on Web Information and Data Management*, ser. WIDM '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 67–74. [Online]. Available: https://doi.org/10.1145/1097047.1097061

[11] Y. Deldjoo, T. Di Noia, and F. A. Merra, "Adversarial machine learning in recommender systems: State of the art and challenges," *ArXiv e-prints*, May 2020, under Review. [Online]. Available: http://www-ictserv.poliba.it/publications/2020/DDM20a/Survey_AML_RecSys.pdf

[12] Y. Deldjoo, T. D. Noia, and F. A. Merra, "A survey on adversarial recommender systems: From attack/defense strategies to generative adversarial networks," *ACM Comput. Surv.*, vol. 54, no. 2, Mar. 2021. [Online]. Available: https://doi.org/10.1145/3439729

[13] C.-I. Fan, H.-W. Hsiao, C.-H. Chou, and Y.-F. Tseng, "Malware detection systems based on api log data mining," in *Proceedings of the 2015 IEEE 39th Annual Computer Software and Applications Conference - Volume 03*, ser. COMPSAC '15. USA: IEEE Computer Society, 2015, p. 255–260. [Online]. Available: https://doi.org/10.1109/COMPSAC.2015.241

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

[15] J. Garcia, M. Hammad, and S. Malek, "Lightweight, Obfuscation-Resilient Detection and Family Identification of Android Malware," *ACM Transactions on Software Engineering and Methodology*, vol. 26, no. 3, pp. 11:1–11:29, Jan. 2018. [Online]. Available: http://doi.org/10.1145/3162625

[16] 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)*, 2018, pp. 30–33.

[17] Q. Gong, J. Zhang, Y. Chen, Q. Li, Y. Xiao, X. Wang, and P. Hui, "Detecting malicious accounts in online developer communities using deep learning," in *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, ser. CIKM '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 1251–1260. [Online]. Available: https://doi.org/10.1145/3357384.3357971

[18] X. Gu, H. Zhang, and S. Kim, "Codekernel: A graph kernel based approach to the selection of API usage examples," in *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*, 2019, pp. 590–601. [Online]. Available: https://doi.org/10.1109/ASE.2019.00061

[19] X. Gu, H. Zhang, D. Zhang, and S. Kim, "Deep API Learning," in *24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. New York: ACM, 2016, pp. 631–642.

[20] I. Gunes, C. Kaleli, A. Bilge, and H. Polat, "Shilling attacks against recommender systems: A comprehensive survey," *Artif. Intell. Rev.*, vol. 42, no. 4, p. 767–799, Dec. 2014. [Online]. Available: https://doi.org/10.1007/s10462-012-9364-9

[21] X. He, L. Xu, X. Zhang, R. Hao, Y. Feng, and B. Xu, "PyART: Python API Recommendation in Real-Time," *arXiv:2102.04706 [cs]*, Feb. 2021, arXiv: 2102.04706. [Online]. Available: http://arxiv.org/abs/2102.04706

[22] R. Holmes, R. J. Walker, and G. C. Murphy, "Strathcona example recommendation tool," in *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*, M. Wermelinger and H. C. Gall, Eds. ACM, 2005, pp. 237–240. [Online]. Available: https://doi.org/10.1145/1081706.1081744

[23] L. Huang, A. D. Joseph, B. Nelson, B. I. Rubinstein, and J. D. Tygar, "Adversarial machine learning," in *Proceedings of the 4th ACM Workshop on Security and Artificial Intelligence*, ser. AISec '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 43–58. [Online]. Available: https://doi.org/10.1145/2046684.2046692

[24] B. A. Kitchenham, P. Brereton, Z. Li, D. Budgen, and A. J. Burn, "Repeatability of systematic literature reviews," in *15th International Conference on Evaluation & Assessment in Software Engineering, EASE 2011, Durham, UK, 11-12 April 2011, Proceedings*, 2011, pp. 46–55. [Online]. Available: https://doi.org/10.1049/ic.2011.0006

[25] W. Koch, A. Chaabane, M. Egele, W. Robertson, and E. Kirda, "Semi-automated discovery of server-based information oversharing vulnerabilities in Android applications," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2017. New York, NY, USA: Association for Computing Machinery, Jul. 2017, pp. 147–157. [Online]. Available: http://doi.org/10.1145/3092703.3092708

[26] S. K. Lam and J. Riedl, "Shilling recommender systems for fun and profit," in *Proceedings of the 13th conference on World Wide Web - WWW '04*. New York, NY, USA: ACM Press, 2004, p. 393. [Online]. Available: http://portal.acm.org/citation.cfm?doid=988672.988726

[27] S. Lee and S. Ryu, "Adlib: analyzer for mobile ad platform libraries," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2019. New York, NY, USA: Association for Computing Machinery, Jul. 2019, pp. 262–272. [Online]. Available: http://doi.org/10.1145/3293882.3330562

[28] S. G. MacDonell, M. J. Shepperd, B. A. Kitchenham, and E. Mendes, "How reliable are systematic reviews in empirical software engineering?" *IEEE Trans. Software Eng.*, vol. 36, no. 5, pp. 676–687, 2010. [Online]. Available: https://doi.org/10.1109/TSE.2010.28

[29] K. Mens and A. Lozano, "Source code-based recommendation systems," in *Recommendation Systems in Software Engineering*, M. P. Robillard, W. Maalej, R. J. Walker, and T. Zimmermann, Eds. Springer, 2014, pp. 93–130. [Online]. Available: https://doi.org/10.1007/978-3-642-45135-5_5

[30] B. Mobasher, R. Burke, R. Bhaumik, and J. J. Sandvig, "Attacks and remedies in collaborative recommendation," *IEEE Intelligent Systems*, vol. 22, no. 3, pp. 56–63, 2007.

[31] B. Mobasher, R. Burke, R. Bhaumik, and J. J. Sandvig, "Attacks and Remedies in Collaborative Recommendation," *IEEE INTELLIGENT SYSTEMS*, p. 8, 2007.

[32] L. Moreno, G. Bavota, M. Di Penta, R. Oliveto, and A. Marcus, "How Can I Use This Method?" in *37th International Conference on Software Engineering*. Piscataway: IEEE, 2015, pp. 880–890.

[33] G. C. Murphy, "Attacking information overload in software development," in *IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2009, Corvallis, OR, USA, 20-24 September 2009, Proceedings*, 2009, p. 4.

[34] E. R. Murphy-Hill, G. C. Murphy, and W. G. Griswold, "Understanding context: creating a lasting impact in experimental software engineering research," in *Proceedings of FoSER 2010, at FSE 2010, Santa Fe, NM, USA, November 7-11, 2010*, 2010, pp. 255–258.

[35] A. Narayanan, M. Chandramohan, L. Chen, and Y. Liu, "A multi-view context-aware approach to Android malware detection and malicious code localization," *Empirical Software Engineering*, vol. 23, no. 3, pp. 1222–1274, Jun. 2018. [Online]. Available: https://doi.org/10.1007/s10664-017-9539-8

[36] A. M. Nguyen, J. Yosinski, and J. Clune, "Deep neural networks are easily fooled: High confidence predictions for unrecognizable images." in *CVPR*. IEEE Computer Society, 2015, pp. 427–436. [Online]. Available: http://dblp.uni-trier.de/db/conf/cvpr/cvpr2015.html#NguyenYC15

[37] 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*, pp. 1–1, 2021.

[38] P. T. Nguyen, J. Di Rocco, D. Di Ruscio, and M. Di Penta, "CrossRec: Supporting Software Developers by Recommending Third-party Libraries," *Journal of Systems and Software*, p. 110460, 2019. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121219302341

[39] 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*, ser. ICSE '19. Piscataway, NJ, USA: IEEE Press, 2019, pp. 1050–1060. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00109

[40] P. T. Nguyen, J. Di Rocco, C. Di Sipio, D. Di Ruscio, and M. Di Penta, "APIRecSys-AML: Supporting repository," https://github.com/MDEGroup/APIRecSys-AML, 2021, [Online; accessed 10-July-2021].

[41] P. T. Nguyen, D. Di Ruscio, J. Di Rocco, C. Di Sipio, and M. Di Penta, "Adversarial machine learning: On the resilience of third-party library recommender systems," in *Evaluation and Assessment in Software Engineering*, ser. EASE 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 247–253. [Online]. Available: https://doi.org/10.1145/3463274.3463809

[42] T. T. Nguyen, H. V. Pham, P. M. Vu, and T. T. Nguyen, "Learning API usages from bytecode: a statistical approach," in *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, 2016, pp. 416–427. [Online]. Available: https://doi.org/10.1145/2884781.2884873

[43] D. L. Parnas, "Information Distribution Aspects of Design Methodology," Departement of Computer Science, Carnegie Mellon University, Pittsburgh, Tech. Rep., 1971.

[44] L. Ponzanelli, G. Bavota, M. Di Penta, R. Oliveto, and M. Lanza, "Prompter - turning the IDE into a self-confident programming assistant," *Empirical Software Engineering*, vol. 21, no. 5, pp. 2190–2231, 2016. [Online]. Available: https://doi.org/10.1007/s10664-015-9397-1

[45] F. Ricci, L. Rokach, and B. Shapira, *Introduction to Recommender Systems Handbook*. Boston, MA: Springer US, 2011, pp. 1–35. [Online]. Available: https://doi.org/10.1007/978-0-387-85820-3_1

[46] M. P. Robillard, "What Makes APIs Hard to Learn? Answers from Developers," *IEEE software*, vol. 26, no. 6, pp. 27–34, 2009.

[47] M. P. Robillard, W. Maalej, R. J. Walker, and T. Zimmermann, Eds., *Recommendation Systems in Software Engineering*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, dOI: 10.1007/978-3-642-45135-5. [Online]. Available: http://link.springer.com/10.1007/978-3-642-45135-5

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

[49] N. Sahavechaphan and K. Claypool, "Xsnippet: Mining for sample code," in *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, ser. OOPSLA '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 413–430.

[50] A. A. Sawant and A. Bacchelli, "fine-GRAPE: fine-grained APi usage extractor – an approach and dataset to investigate API usage," *Empirical Software Engineering*, vol. 22, no. 3, pp. 1348–1371, Jun. 2017. [Online]. Available: http://link.springer.com/10.1007/s10664-016-9444-6

[51] D. Shriver, S. Elbaum, and M. B. Dwyer, "Reducing DNN Properties to Enable Falsification with Adversarial Attacks," in *43rd International Conference on Software Engineering*. IEEE, 2021.

[52] J. Singh and J. Singh, "Detection of malicious software by analyzing the behavioral artifacts using machine learning algorithms," *Information and Software Technology*, vol. 121, p. 106273, May 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584920300239

[53] F. Thung, D. Lo, and J. Lawall, "Automated library recommendation," in *2013 20th Working Conference on Reverse Engineering (WCRE)*, Oct 2013, pp. 182–191.

[54] J. D. Tygar, "Adversarial machine learning," *IEEE Internet Computing*, vol. 15, no. 5, pp. 4–6, 2011.

[55] 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 MSR*. Piscataway: IEEE, 2013, pp. 319–328.

[56] J. Wang and J. Han, "Bide: efficient mining of frequent closed sequences," in *Proceedings. 20th International Conference on Data Engineering*, 2004, pp. 79–90.

[57] J. Wang and P. Han, "Adversarial Training-Based Mean Bayesian Personalized Ranking for Recommender System," *IEEE Access*, vol. 8, pp. 7958–7968, 2020, conference Name: IEEE Access.

[58] D. Wu, D. Gao, R. K. C. Chang, E. He, E. K. T. Cheng, and R. H. Deng, "Understanding open ports in android applications: Discovery, diagnosis, and security assessment," in *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019. [Online]. Available: https://bit.ly/3e3enkJ

[59] S. Wu, P. Wang, X. Li, and Y. Zhang, "Effective detection of android malware based on the usage of data flow APIs and machine learning," *Information and Software Technology*, vol. 75, pp. 17–25, Jul. 2016. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0950584916300386

[60] H. Zhang, M. A. Babar, and P. Tell, "Identifying relevant studies in software engineering," *Information and Software Technology*, vol. 53, no. 6, pp. 625–637, 2011.

[61] Y. Zhang, Y. Fan, S. Hou, Y. Ye, X. Xiao, P. Li, C. Shi, L. Zhao, and S. Xu, "Cyber-guided Deep Neural Network for Malicious Repository Detection in GitHub," in *2020 IEEE International Conference on Knowledge Graph (ICKG)*, Aug. 2020, pp. 458–465.

[62] 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*. Berlin, Heidelberg: Springer, 2009, pp. 318–343.