Using API-Embedding for API-Misuse Repair

Sebastian Nielebock
University of Magdeburg
Germany
sebastian.nielebock@ovgu.de

Robert Heumüller
University of Magdeburg
Germany
robert.heumueller@ovgu.de

Jacob Krüger
University of Toronto &
University of Magdeburg
Canada & Germany
jacob.krueger@ovgu.de

Frank Ortmeier
University of Magdeburg
Germany
frank.ortmeier@ovgu.de

ABSTRACT

Application Programming Interfaces (APIs) are a way to reuse existing functionalities of one application in another one. However, due to lacking knowledge on the correct usage of a particular API, developers sometimes commit misuses, causing unintended or faulty behavior. To detect and eventually repair such misuses automatically, inferring API usage patterns from real-world code is the state-of-the-art. A contradiction to an identified usage pattern denotes a misuse, while applying the pattern fixes the respective misuse. The success of this process heavily depends on the quality of the usage patterns and on the code from which these are inferred. Thus, a lack of code demonstrating the correct usage makes it impossible to detect and fix a misuse. In this paper, we discuss the potential of using machine-learning vector embeddings to improve automatic program repair and to extend it towards cross-API and cross-language repair. We illustrate our ideas using one particular technique for API-embedding (i.e., API2Vec) and describe the arising possibilities and challenges.

CCS CONCEPTS

• **Software and its engineering → Error handling and recovery**; **Software defect analysis**.

KEYWORDS

API Misuse, API Embeddings, Program Repair

ACM Reference Format:
Sebastian Nielebock, Robert Heumüller, Jacob Krüger, and Frank Ortmeier. 2020. Using API-Embedding for API-Misuse Repair. In *IEEE/ACM 42nd International Conference on Software Engineering Workshops (ICSEW'20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3387940.3392171

1 INTRODUCTION

Application Programming Interfaces (APIs) describe how functionalities of a program, for example, a library, can be accessed and used in a different program. However, due to missing or outdated documentation, or simply due to a lack of understanding of a particular API, developers may misuse it [7]. We denote an API misuse as any deviant usage of an API compared to the usages intended by the API developers, causing faulty behavior in the code, such as software crashes or performance losses. One way to detect and fix misuses are API usage patterns that describe how an API is correctly used. Usage patterns are inferred from real-world source code using temporal specification-mining techniques [1, 3, 10]. A contradiction of a pattern is then considered as a misuse. Eventually, these patterns can then be used to automatically fix the misuses [9, 11]. The success of this technique strongly depends on the quality of the patterns and the source code from which patterns are mined [5]. If the code does not contain exactly the patterns of a particular API, existing techniques cannot fix its misuses.

However, other researchers showed that source code exhibits regularity, similar to natural languages [4]. So, well-known machine-learning models like Word2Vec [6] could also be applicable to source code. In particular, Word2Vec denotes words as high-dimensional vectors that allow conducting arithmetic operations to express semantic relations, such as $V(Athens) - V(Greece) + V(Norway) \approx V(Oslo)$. Nguyen et al. [8] introduced API2Vec and showed that such relations can be found for APIs as well. Consequently, it is possible to find semantic relationships between the elements of one API that are also present in other APIs, which do not necessarily share syntactical similarities. Therefore, when lacking sufficient examples using a particular API, it can be possible to adapt known patterns from other APIs. Moreover, Nguyen et al. also show the possibility of using this technique for cross-language migrations, allowing to transfer patterns from one programming language to another. Building on such ideas, we propose to use machine learning—and the example of API2Vec—for automatic API-misuse repair (cf. Section 2); and discuss open challenges, especially for cross-API and cross-language support (cf. Section 3).

2 USING API2VEC FOR API-MISUSE REPAIR

API2Vec [8] uses the continuous bag-of-words (CBOW) model from Word2Vec to create API embeddings. This network consists of an input, a hidden, and an output layer. The input is the context of an API element w_i (i.e., $w_{i-n}, \cdots, w_{i-1}, w_{i+1}, \cdots, w_{i+n}$) with a fixed size (i.e., $2n$). The network is then trained to determine the missing API element w_i. After training, API embeddings for each API element are inferred as vectors, spanning a high-dimensional API2Vec space. For training, Nguyen et al. [8] obtained API call sequences statically from real-world source code. One trained network based on Java projects and the JDK API is available on GitHub.[1]

As Nguyen et al. [8] show, API elements frequently used together also cluster in the API2Vec space, which we expect to identify as API usage patterns. Patterns mined from code are usually represented

ICSEW'20, May 23–29, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7963-2/20/05…$15.00
https://doi.org/10.1145/3387940.3392171

[1] https://github.com/pdhung3012/api2vec

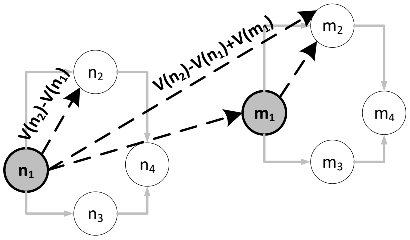

Figure 1: Mapping pattern nodes based on API2Vec vectors.

as data structures describing the relations of API elements to each other (e.g., a sequence or a graph) [10]. We apply API usage graphs as introduced by Amann et al. [2] to repair misuses, since these effectively decrease the false positive rate in API misuse detection.

To explain the general notion of the repair mechanism, we assume a two-dimensional projection of the API2Vec space (cf. Figure 1). Our intent is to find a pattern consisting of nodes m_i (destination pattern) that is similarly structured as the pattern consisting of nodes n_i (source pattern). To find this pattern, we can apply a similar technique as Nguyen et al. [8]. First, we identify two nodes, one from each pattern, serving as "hook nodes" (e.g., n_1 and m_1 in Figure 1). A possible way to identify the hooks is to choose n_1 randomly from the known pattern and to determine m_1 as the closest node in the API2Vec space that belongs to a different API than n_1. Based on these nodes, we can compute the likely position of API elements that, for instance, represent node n_2 from the source pattern in the destination pattern (i.e., node m_2 by $V(m_2) \approx V(n_2) - V(n_1) + V(m_1)$). Usually, the position of a node is only an approximation, so that we have to check the closest API elements—ranked by a distance metric. Note that we only need one hook node per pattern, since we can always compute the vector from the hook node to any other node in the source pattern.

This process yields patterns that are likely similar in their functionality. For example, adding an element to a list shares some similar API calls with adding an element to a stack. In case we found such a pair of patterns (i.e., source and destination patterns), the destination pattern can then be used to detect and repair API misuses. Usually, one would repair API misuses by the source pattern. However, in case we cannot infer this pattern, we are not able to fix the misuse. Still, we can leverage the existing API embeddings, to map a known and frequent destination pattern to the source API, forming a source pattern. For evaluation, we plan to use a set of well-known API misuses (e.g., the MuBench dataset used in previous studies by Amann et al. [1]) and generate the source patterns to fix the misuses based on similar destination patterns. To check whether we can really infer destination patterns, we prepare source-code files that have an over-represented amount of usages of the similar API compared to the "required" API of the misuse. So, we will re-train the existing CBOW model and implement a technique for transforming graph-based patterns into code patches.

3 CHALLENGES FOR API-MISUSE REPAIR

Even though very promising, we faced several challenges using the API2Vec technique, which are arguably transferable to other machine learning techniques for automatically repairing API misuses.

(1) We require a minimum number of example usages for both the source and destination patterns. Otherwise, it is impossible to compute the mappings, even if they exist. For finding destination patterns, we also envision applying API2Vec in its cross-language configuration, to find similar patterns in other languages. This requires a transformation of the two vector spaces [8]. Therefore, we have to show whether and between which paradigms of programming languages such relations exist.

(2) We assume that API elements frequently applied together are not only in local proximity, but are structured similarly. So, the vectors between nodes of a pattern are similar to the vectors of another, comparable pattern. Currently, this has only been shown for simple patterns, such as API method-call pairs. Whether more complex patterns are similar is an open question for future work.

(3) Some patterns describe the same concept but need more or fewer API elements. Our technique would then fail to find a one-to-one mapping of nodes. A possible solution could be to split and merge nodes in source patterns to find more similar patterns.

(4) We will extend the existing data set with third-party APIs. While JDK APIs indicate good results, we do not know whether these relations also exist among third-party libraries.

4 SUMMARY

In this paper, we proposed the idea of combining API2Vec with API misuse repair. API2Vec introduced the idea of using a vector representation to find similar API usages across different APIs (and even programming languages), where APIs must not necessarily share syntactical similarities. While promising, we also identified important challenges, for example, learning appropriate neural networks or collecting empirical evidence on whether similar patterns exhibit similar structures in the API2Vec space. Arguably, other machine-learning techniques can also be adopted to facilitate API misuse repair, but face similar problems.

Acknowledgments. This research has been supported by the German Research Council DFG (grant no. SA 465/49-3).

REFERENCES

[1] Sven Amann, Hoan Anh Nguyen, Sarah Nadi, Tien N Nguyen, and Mira Mezini. 2018. A Systematic Evaluation of Static API-Misuse Detectors. *IEEE TSE* 45, 12 (2018).

[2] Sven Amann, Hoan Anh Nguyen, Sarah Nadi, Tien N Nguyen, and Mira Mezini. 2019. Investigating next Steps in Static API-Misuse Detection. In *16th MSR*. IEEE.

[3] Glenn Ammons, Rastislav Bodík, and James R Larus. 2002. Mining Specifications. In *29th POPL*. ACM.

[4] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the Naturalness of Software. In *34th ICSE*. IEEE.

[5] Claire Le Goues and Westley Weimer. 2012. Measuring Code Quality to Improve Specification Mining. *IEEE TSE* 38, 1 (2012).

[6] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality. In *26th NeurIPS*. Curran Associates.

[7] Sarah Nadi, Stefan Krüger, Mira Mezini, and Eric Bodden. 2016. Jumping through Hoops: Why Do Java Developers Struggle with Cryptography APIs?. In *38th ICSE* (Austin, Texas). ACM, 12.

[8] Trong Duc Nguyen, Anh Tuan Nguyen, Hung Dang Phan, and Tien N Nguyen. 2017. Exploring API Embedding for API Usages and Applications. In *39th ICSE*. IEEE.

[9] Sebastian Nielebock. 2017. Towards API-Specific Automatic Program Repair. In *32nd ASE*. IEEE.

[10] Martin P Robillard, Eric Bodden, David Kawrykow, Mira Mezini, and Tristan Ratchford. 2013. Automated API Property Inference Techniques. *IEEE TSE* 39, 5 (2013).

[11] Westley Weimer. 2006. Patches as Better Bug Reports. In *5th GPCE*. ACM.