Inter-Procedural Graph-Based API Misuse Detection

Ruben Opdebeeck, Camilo Velázquez-Rodríguez
Software Languages Lab, Vrije Universiteit Brussel, Belgium
{ropdebee,cavelazq}@vub.be

I. Extended Abstract

The vast majority of programs written today use functionality provided by one or more libraries, and thus integrate with numerous Application Programming Interfaces (APIs). However, programming with these APIs can be difficult due to extensive APIs, implicit usage contracts, and missing or outdated documentation. Misusing such APIs may lead to severe bugs such as program crashes, data loss, or security vulnerabilities.

To alleviate the difficulties in using APIs, a lot of effort has gone into the research of API misuse detectors. Many of these detectors use the wisdom of the crowd to flag likely API misuses. Such tools first mine patterns in library usage across a corpus of programs, and then perform pattern matching in order to find imperfect instances, or violations, of the pattern. Due to space considerations, we refer to the main author's thesis dissertation [1] for a literature study of the field.

Amann et. al. [2] recently performed a systematic study of existing API misuse detectors and found a large number of common shortcomings. Using insights from this study, they developed MUDetect and its accompanying *API Usage Graph* (AUG) representation [3]. MUDetect improves upon the results of previous research by a factor of 2 in precision and recall. However, MUDetect's pattern mining and pattern matching are completely intra-procedural, which is a significant root cause for both false positives and false negatives. For example, in the presence of helper methods, intra-procedural pattern mining can only infer partial patterns, and intra-procedural pattern matching reports missing program elements which are present in the helper method's body.

We propose a tool that builds upon the work of Amann et. al. by incorporating *function inlining* into the construction of API usage graphs. This is made difficult due to the extent of information captured in API usage graphs, such as control and data flow. Pattern mining is performed using an apriori frequent subgraph mining algorithm, whereas pattern matching is done using exploratory growth of common subgraphs. However, the subgraph isomorphism problem is NP-complete, and thus our approach suffers from combinatorial explosion. In order to limit the size of the mined graphs, we devise four different inlining parameters: The maximum depth of inlining, whether or not recursive calls should be eliminated, whether or not duplicate calls should be eliminated, and whether or not calls to methods that contain no direct usage of a targeted API should be inlined.

In pattern matching, our approach introduces an additional inter-procedural filtering step. This step is intended to remove common false positives, such as partial patterns which are further instantiated by helper methods, and duplicate violation reports due to pattern elements that are missing at function boundaries. Mitigating the latter case is done by blaming either the caller or callee in the violation, based on the fraction of callers that contain the same misuse. Intuitively, if the callee would be missing an element, all of its callers would too, since the callee is inlined into the callers.

We evaluated our work using the MUBench [4] misuse benchmark by running experiments on nine open-source projects for different combinations of the inlining parameters, and one instance of our approach without inlining enabled as a baseline. In these experiments, we limit the pattern mining to the target project. We find that the inter-procedural filter succeeds at removing false positives related to inter-procedural pattern instances, leading to an increase in precision over the baseline. However, this is offset by a minor loss in both recall and precision compared to MUDetect, because the resulting graphs cannot represent first-party method calls after inlining. Inlining also comes at a much higher execution time, since the constructed graphs are larger in size, which mainly affects the pattern matching phase. Finally, we find that more elimination through different parameters leads to diminishing results, as expected, but leads to an immense decrease in running times.

It is our immediate future work to extend this evaluation to the full MUBench dataset, as well as perform pattern mining in a cross-project setting. We expect the latter to produce a higher precision and recall because cross-project mining cannot mine patterns of first-party APIs, yet exhibits promising results in MUDetect's evaluation. Further future work includes the adoption of a richer representation that allows capturing first-party patterns and focuses more on the behaviour of a program, as well as performance improvements through clustering and slicing and more efficient graph pattern matching, as opposed to excessive elimination of inlining which hinders the ability to mine accurate patterns.

References

[1] R. Opdebeeck, "Exploring Static Inter-Procedural API Misuse Detection Using Graph Inlining," Master's thesis, Vrije Universiteit Brussel, Brussels, Belgium, 2019.
[2] S. Amann, H. A. Nguyen, S. Nadi, T. N. Nguyen, and M. Mezini, "A Systematic Evaluation of API-Misuse Detectors," *IEEE Transactions on Software Engineering*, 2018.
[3] ——, "Investigating Next-Steps in Static API-Misuse Detection," in *Proc. MSR '19*. IEEE, 2019, pp. 265–275.
[4] S. Amann, S. Nadi, H. A. Nguyen, T. N. Nguyen, and M. Mezini, "MUBench: a Benchmark for API-Misuse Detectors," in *Proc. MSR '16*. ACM, 2016, pp. 464–467.