1 Root Causes of MD's False Positives and False Negatives

For reasons of space, in the paper, we present only the most-prevalent root causes of MD's false positives (Experiment P) and false negatives (Experiment RUB).

1.1 Experiment P

The following is the full list of the root causes of all MD's false positives in Experiment P.
***FP1:* Uncommon Usages.** 84 (73.7%) of the false positives are uncommon-but-correct usages, i.e., they are instances of a usage pattern, which, however, is not mined due to low frequency of its occurrences. These usages are flagged as violations, because they deviate from the patterns that MD learned.

In nine cases, e.g., a loop calls `Iterator.hasNext()` again after calling `next()`, to check whether there will be a subsequent iteration. MD reports a missing call to `next()` after this second call to `hasNext()`. This illustrates two problems: (1) the heuristic for identifying pure methods by looking for methods with the prefix `get` misses cases such as this `hasNext()` call. (2) MD does not consider alternative non-frequent patterns. A future solution might be *a probabilistic model of API usage* that considers the likelihood of different usages and reports no violation if one usage is only slightly more likely than another, or if an API's usages generally vary a lot.

Almost as prevalent (seven violations) is a scenario where the pattern ($p_s = 10$) expects the primitive value of a `PDFObject` to be retrieved using the `floatValue()` method, while the usage calls the `intValue()` method. Both usages are valid alternatives, but the latter has only a support of seven in the project, which is why MD does not learn a respective pattern.

Another reoccurring scenario (five violations) again uses an `Iterator`, checking for the presence of sufficient elements by either `size()` or `isEmpty()` on the underlying collection, instead of the expected `hasNext()` method. Again, these alternative usages occur infrequently for MD to learn a respective pattern.
***FP2:* Intra-procedural Analysis.** 18 (15.8%) of the false positives are due to our intra-procedural analysis. In seven cases, MD reports missing usage elements that occur in transitively called methods. Using an inter-procedural analysis, e.g., to filter such false positives as proposed by Li and Zhou [1], might help to mitigate this problem. Future work should investigate whether the additional computational cost pays off.
***FP3:* Dependent States.** Five (4.4%) of the false positives are due to implicit dependencies. In three case, for example, the code ensures that two collections have the same size and then iterates over one, while checking the size of the other. This usage is safe, but MD does not capture such implicit dependencies between object states. This problem was also discussed in previous work [2] as the least prevalent cause for false positives. Since it is very difficult, if not impossible, to capture inter-dependencies of object states using static analysis, we did not address this problem with MD.
***FP4:* Mapping Choice.** Another five (4.4%) of the false positives are due to MD's detection algorithm choosing a non-optimal mapping between a pattern and a target. In these cases, the target is actually an instance of the pattern, but MD does not recognize

this, due to the greedy extension strategy we introduced to scale the detection. We consider this a reasonably small cost, considering that we could not perform an exhaustive search on many projects in our experimental environment.

***FP5:* Cross-method Usages.** One false positive (0.9%) is a cross-method usage, which our filtering misses, because the respective object is initialized as a local variable and only later assigned to a field.

***FP6:* Combined Patterns.** The last false positive (0.9%) is a case where a violation is an instance of a combination of two alternative patterns, which our alternative-pattern filtering cannot detect.

1.2 Experiment RUB

The following is the full list of the root causes of all MD's false negatives in Experiment RUB.

***FN1:* Representation.** In 15 cases, an illegal parameter value (constant or literal) is passed as a call parameter. MD cannot detect this, because AUGs do not capture concrete values.

***FN2:* Self-Usages.** Eight cases are due to our removal of self-usages to counteract their potential to cause false positives. In three cases, the misuses contain method calls on the API being implemented and in another three cases, the misuses contain calls on fields. When creating AUGs, we explicitly disregard both these scenarios to avoid false positives caused by partial usages. This strategy successfully reduced the impact of partial usages in Experiment P. However, these eight false negatives show that we traded precision for recall. By *capturing inter-procedural usages*, we might make filtering self usages and usages on fields unnecessary and enable us to identify misuses in them. The Chronicler [3] detector mines usages from an inter-procedural call graph, which might mitigate the problem. However, it is unclear how to adapt this approach from considering only method calls to all usage elements we encode in AUGs. Furthermore, such an approach duplicates evidence, if methods are called multiple times, which might bias the mining.

***FN3:* Matching.** In seven cases, MD cannot match the respective pattern and the target usages, because they contain only a single, distinct call each, because the respective API favors the use of one method over the use of another, e.g., due to deprecation. Our detection algorithm expects usages to consist of at least two usage elements and, by design, does not match AUGs that have no method call in common. None of the other other detectors can identify these misuses either.

***FN4:* Redundant.** Seven cases are misuses where the usage has a redundant element that should be removed. Since all detectors are designed to detect missing elements, none can detect these misuses. It is worth noting that DroidAssist [4] uses a *probabilistic approach* that might find superfluous method call, but the technique has never been evaluated.

***FN5:* Pure Methods.** In six cases, our mining excludes elements from the pattern when heuristically determining semantically irrelevant nodes (calls to getter methods in

all cases). The remaining pattern can no longer be matched to the usage.
***FN6:* Multiplicity.** In one case, our mining excludes subsequent calls to the same method due to our heuristic of addressing call multiplicities. The remaining pattern no longer shows the difference between the misuse and the correct usage.
***FN7:* Analysis.** In one case, our analysis misses data-flow relations, because it assumes single static assignment. In the correct usage `if (v == null) { v = ... } v.m()`, which ensures the initialization of `v` before its use, it regards `v` before and after the assignment as two different objects. Therefore, our pattern mining sees no data flow between the `null` check and the subsequent usage and excludes the check from the pattern. Consequently, MD cannot identify the missing check in the respective misuse.
***FN8:* Operator Abstraction.** In one case, the misuse is an accidentally inverted condition, which we miss because we abstract from concrete operators in representing conditions.

References

[1] Z. Li and Y. Zhou, "PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code," in *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESEC/FSE '13. ACM Press, 2005, pp. 306–315.

[2] S. Amann, H. A. Nguyen, S. Nadi, T. N. Nguyen, and M. Mezini, "A systematic evaluation of static API-misuse detectors," *IEEE Transactions on Software Engineering*, 2018. [Online]. Available: https://arxiv.org/abs/1712.00242

[3] M. K. Ramanathan, A. Grama, and S. Jagannathan, "Static specification inference using predicate mining," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '07. ACM Press, 2007, pp. 123–134.

[4] 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*, ser. ICSE '16. ACM Press, 2016.