API Analytics for Curating Static Analysis Rules

Vineeth Kashyap
Roger Scott
Joseph Ranieri
David Melski
Lucja Kot
{vkashyap,rscott,jranieri,melski,lkot}@grammatech.com
GrammaTech, Inc.
USA

Abstract

Use of third-party library APIs is pervasive, but can be error-prone. API-usage errors can be detected via static analysis if specifications of correct usage are available, but manually creating such specifications is a bottleneck. We showcase a semi-automated "big code" solution, where we use large code corpora to mine patterns in API usage, and ask human experts to perform analytics on those patterns to create static analysis rules.

CCS Concepts: • **Software and its engineering → Software maintenance tools**.

Keywords: static analysis, big code, analytics

ACM Reference Format:
Vineeth Kashyap, Roger Scott, Joseph Ranieri, David Melski, and Lucja Kot. 2020. API Analytics for Curating Static Analysis Rules. In *Proceedings of the 11th ACM SIGPLAN International Workshop on Tools for Automatic Program Analysis (TAPAS '20), November 17, 2020, Virtual, USA*. ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3427764.3428318

1 Introduction

Modern programs are rarely written from scratch—they make heavy use of library API functions (henceforth referred to as APIs). The number of APIs is constantly growing, and it can be difficult for developers to use them correctly. Static analysis can help developers find such incorrect uses.

Static analysis tools check code for violations of a specified set of rules [1, 4, 7]; given a partial specification of correct usage for APIs, static analysis can help find errors in API usage. Unfortunately, detailed and unambiguous specifications are often not available. Because of the effort involved in creating specifications manually, most static analysis tools only check usages of a small percentage of popular APIs. This is a major source of false negatives in static analysis.

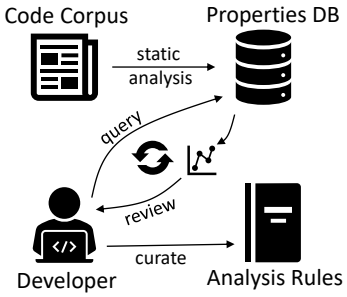

Figure 1. Proposed workflow for curating analysis rules.

"Big code" presents an exciting opportunity to mine API usage specifications from large, openly-available code corpora. The underlying assumption for such efforts is that most uses of APIs are correct, and that we can infer specifications from common usage patterns. Salento [6] demonstrates that simple specifications, such as valid API call sequences, can be automatically mined for a few Android APIs. However, it is difficult to automatically distinguish "useful" rules from spurious ones, and uncommon usage patterns are not necessarily incorrect. Consequently, to date, fully-automated approaches for mining API specifications—especially those that mine deeper specifications like pre- and post-conditions—suffer from high false-positive rates in practice (e.g., APISan [8] has an 88% false-positive rate).

We propose the workflow in Figure 1 to accelerate the curation of static analysis rules to check API usages. First, we run static analyses on a large corpus of programs to extract semantic properties (e.g., various predicates on argument and return values) of API usage sites, and store such information in a Properties DB. Then, instead of a fully-automated approach, analysis developers *interactively* perform "data analytics" on the Properties DB, and curate meaningful API usage rules. Because the rules have been manually vetted, violations of those rules are likely to be more meaningful, and have a lower effective false-positive rate (i.e., fewer false positives as determined by a human user). Learning from the results of such interactive analytics, analysis developers

TAPAS '20, November 17, 2020, Virtual, USA

ACM ISBN 978-1-4503-8189-5/20/11.
https://doi.org/10.1145/3427764.3428318

can conceive new—or refine existing—semantic properties on API usage.

2 Case Studies

JavaScript. We used the TypeScript compiler [5] to generate intra-procedural dataflow graphs of JavaScript programs. Based on these graphs, we statically inferred whether an argument (and any statically-known fields of an argument) to an API call can be a string with a statically-known value (i.e., a hard-coded string). We collected such data on projects in the node package manager (npm), for over 700 million lines of code. We performed analytics on this data to identify APIs which are often called with arguments (or any statically-known fields of arguments) that are not hard-coded. Table 1 provides a small sample of our derived rules. Next, we examined multiple npm projects for violations of our rules; we found over 140 true-positive bugs, some of which are shown in Listing 1.

```
// in  bleuapp-token-service/app.js
jwt.sign(/* payload */, "michaelwildchangethisplease");
// in qiangbibaokelvv/index.js
jwt.verify(config.code, 'shuabibao$#FF@@%GVD');
// in sabium-framework/models/funcoes.component.js
CryptoJS.RC4.encrypt(retorno, 'sabium');
```

Listing 1. Warnings on JavaScript code (different projects). Hard-coded sensitive information is underlined.

C/C++. We used the CodeSonar [3] static analysis tool's intermediate representations to infer various semantic properties[1] at API callsites in a C/C++ program. We computed, for each API callsite considered, whether:

- P1. the return value is used across every program path
- P2. an argument is (a) non-negative, (b) non-zero
- P3. a "pointer" argument, when dereferenced, has been (a) initialized, (b) initialized to a non-empty string.

We collected such data on over 400 million lines of C/C++ programs, taken from the Fedora SRPM projects. We then analyzed this data to curate API usage rules. For example, we identified those APIs with callsites where P1 is often true, and then manually verified whether the obtained rules are valid. Through this mechanism, we obtained P1-style rules for over 1500 APIs, which we integrated into CodeSonar. We also integrated a subset of these rules into the open-source tool Clang-Tidy [2]. When we manually vetted violations of our rules on open-source code, they typically pointed to inefficient code or memory leaks. Sometimes, as shown in Listing 2, they were security errors. Here, `g_strescape` escapes unsafe characters in `param_str`, and returns a safe string[2]; however, the programmer discards the return value and uses the old, unsafe `param_str` later used in constructing an SQL command.

[1]Only a subset of these semantic properties are discussed in this paper.
[2]In most uses of `g_strescape`, P1 is true, which is how we mined this rule.

Table 1. Rules inferred from JavaScript programs: for a given API, the "Applicable Argument" should not be hard-coded.

API	Applicable Arg
`mysql.createConnection`	1 (`password` field)
`ssh2.Client.connect`	1 (`password` field)
`meshblu.createConnection`	1 (`token` field)
`jsonwebtoken.sign`	2
`CryptoJS.CipherHelper.encrypt`	2
`secp256k1.publicKeyCreate`	1

Table 2. Rules inferred from C and C++ programs: for a given API, the "Property" should be true for the "Arg" argument.

API	Property	Arg
`memset`	P2(a)	3
`strcat`	P2(b)	1
`pthread_mutex_unlock`	P3(a)	1
`xmlNewChild`	P3(b)	3

```
if (param_str)
  g_strescape(param_str, NULL);
// later, an sql cmd is built using param_str
```

Listing 2. Warning on C code from the "Query Object Framework" project, based on a curated rule.

In Table 2, we provide a sample of the rules we mined and vetted based on the other semantic properties we extracted.

Acknowledgments

This research was supported by DARPA MUSE award #FA8750-14-2-0270 and DHS STAMP award #HHSP233201600062C. The views, opinions, findings, and recommendations contained herein are solely those of the authors.

References

[1] Google. 2020. *ErrorProne Bug Patterns.* https://errorprone.info/bugpatterns
[2] GrammaTech Inc. 2020. *Clang Tidy Checker Enhancement.* https://reviews.llvm.org/D76083
[3] GrammaTech, Inc. 2020. *CodeSonar.* Retrieved May 11, 2020 from https://www.grammatech.com/products/codesonar
[4] LLVM Project. 2020. *Clang Static Analyzer: Available Checkers.* https://clang-analyzer.llvm.org/available_checks.html
[5] Microsoft. 2020. *TypeScript Compiler API.* https://github.com/microsoft/TypeScript/wiki/Using-the-Compiler-API
[6] Vijayaraghavan Murali, Swarat Chaudhuri, and Chris Jermaine. 2017. Bayesian Specification Learning for Finding API Usage Errors. In *ESEC/FSE* (Paderborn, Germany) *(ESEC/FSE 2017).* ACM, New York, NY, USA, 151–162. https://doi.org/10.1145/3106237.3106284
[7] University of Maryland. 2015. *FindBugs Bug Patterns.* http://findbugs.sourceforge.net/bugDescriptions.html
[8] Insu Yun, Changwoo Min, Xujie Si, Yeongjin Jang, Taesoo Kim, and Mayur Naik. 2016. APISAN: Sanitizing API Usages through Semantic Cross-Checking. In *Proceedings of the 25th USENIX Conference on Security Symposium* (Austin, TX, USA) *(SEC'16).* USENIX Association, USA, 363–378.