Аннотация

Применение методов машинного обучения в статистическом анализе

Назаров Константин Олегович

Статические анализаторы широко используются в промышленности для нахождения и исправления уязвимостей в коде в процессе разработки, тем самым улучшая качество кода и упрощая дальнейшую разработку. Однако их применение ограничивается тем, что на больших и сложных кодовых базах статические анализаторы выдают большое количество false positives наряду с предупреждениями о действительных уязвимостях. Поэтому значительная часть данной работы посвящена тому, чтобы применить методы машинного обучения для классификации и приоритизации сообщений статического анализатора, что позволит сократить время, затрачиваемое разработчиками на ручную обработку всех сообщений анализатора. Основной проблемой при обучении подобного классификатора является составление тренировочного датасета, т.к. неочевиден алгоритм разметки данных и признаки для обучающей выборки. Также большим фактором является скорость генерации: признаки не должны вычисляться долго, т.к. сам статический анализ занимает очень большое время (для относительно простых анализаторов время работы примерно в

2 раза превышает время компиляции, но может быть и много больше [13]). В данной работе рассматривается способ генерации датасета при помощи тестовых сюит для статических анализаторов (т.е. репозиториев с "образцовыми"программами, специально написанными для тестирования статических анализаторов), а также показывается, что включение в обучающий датасет token-based представления кода помогает увеличить способность к распознаванию разных шаблонов ошибок, не увеличивая при этом сложности генерации датасета. Данные для обучения были получены путем применения нескольких статических анализаторов к кодовой базе Juliet C/C++ Test Suite v1.3. Разметка была произведена автоматически из метаданных тестовой сюиты. Полученный датасет был использован для обучения двух моделей: decision tree и gradient boosting, предоставляемых библиотекой xgboost для python3. Полученные классификаторы имеют высокую точность на тестовой выборке (≈ 96 %). Также в работе был предложен альтернативный метод генерации датасета, призванный решить проблемы использованного метода. Применение альтернативного метода является предметом дальнейших исследований.

Оглавление

Глава 1

Введение

Обозначения и сокращения

1. False Positive, FP - ошибка бинарной классификации, при которой классификатор ошибочно предсказывает положительный результат (например диагноз болен, когда болезни на самом деле нет)

2. Датасет - это обработанная и структурированная информация в табличном виде. Строки такой таблицы называются объектами, а столбцы – признаками (фичами)

3. Token-Based представление - представление кода в виде последовательности токенов, получаемых в результате лексического анализа

4. Тестовая сюита - специализированный набор тестов

5. Abstract Syntax Tree, AST - представлнение исходного кода в виде дерева абстрактного синтаксиса

6. Application Programming Interface, API - описание способов (набор классов, процедур, функций, структур или констант), кото-

рыми одна компьютерная программа может взаимодействовать с другой программой

7. Common Weakness Enumeration, CWE [20] - Список уязвимостей программного и аппаратного обеспечения. Является общепринятым языком для описания природы уязвимости

8. NIST - The National Institute of Standards and Technology

9. SATE - Static Analysis Tool Expositions

Введение

Статические анализаторы широко используются в промышленности для нахождения и исправления уязвимостей в коде в процессе разработки, тем самым улучшая качество кода и упрощая дальнейшую разработку. Однако их применение ограничивается Множеством факторов. Статья "A Few Billion Lines of Code Later. Using Static Analysis to Find Bugs in the Real World"[4] раскрывает множество проблем с применением статических анализаторов на практике, среди которых: большое количество false positives ($35\% - 91\%$ [11]) на сложных или больших кодовых базах; пользователи, помечающие непонятные им сообщения как ложные; а также сложность обработки большого количества ошибок пользователем, которая приводит к тому, что анализаторы выдают меньше сообщений, чем могли бы. В связи с этим, Moritz Marc Beller с коллегами [3] обнаружили, что на практике очень малое число проектов с открытыми исходным кодом интегрирует статические анализаторы в процесс разработки. Также большая часть проектов не поддерживается правила, что предупре-

ждения компиляции должны рассматриваться как ошибки. Поэтому значительная часть данной работы посвящена тому, чтобы применить методы машинного обучения для классификации и приоритизации сообщений статического анализатора, что позволит сократить время, затрачиваемое разработчиками на ручную обработку всех сообщений анализатора а также облегчит эффективное использование анализаторов на практике. Основной проблемой при обучении подобного классификатора является составление тренировочного датасета, т.к. неочевиден алгоритм разметки данных и признаки для обучающей выборки. Также большим фактором является скорость генерации: признаки не должны вычисляться долго, т.к. сам статический анализ занимает очень большое время (для относительно простых анализаторов время работы примерно в 2 раза превышает время компиляции, но может быть и много больше [13]). В промышленности эта проблема решается тем, что датасет зачастую имеется[16] [10] или может быть легко получен из статистики использования или из истории проекта[18]. В данной работе рассматривается способ генерации датасета при помощи тестовых сюит для статических анализаторов[9] (т.е. репозиториев с "образцовыми"программами, специально написанными для тестирования статических анализаторов), а также показывается, что включение в обучающий датасет token-based представления кода помогает увеличить способность к распознаванию разных шаблонов ошибок, не увеличивая при этом сложности генерации датасета. Данные для обучения были получены путем применения нескольких статических анализаторов к кодовой базе Juliet C/C++ Test Suite [1]. Разметка была произведена автоматически из метаданных тестовой сюиты. Полученный датасет был ис-

пользован для обучения двух моделей, которые впоследствии были протестированы на тестовой выборке.

Глава 2

Постановка задачи

Основной целью данной работы является сокращение времени, затрачиваемого разработчиком на обработку результатов статического анализа. Также важной целью является применение полученных результатов для анализа работы статического анализатора и его дальнейшей настройки. Для достижения обозначенной цели требуется решить следующие задачи:

1. Определить признаки для будущего датасета
2. Определить метод генерации датасета и его разметки. Собрать данные
3. Обучить классификатор сообщений статического анализатора
4. Проанализировать полученные результаты

Для того, чтобы результат работы считался удовлетворительным и применимым на больших масштабах, решения для поставленных задачи должны удовлетворять следующим требованиям:

1. Сложность извлечения признаков не должна превосходить сложности статического анализа

2. Разметка датасета должна быть автоматизированной, т.к. ручная разметка занимает слишком долгое время и требует наличия проанализированной большой кодовой базы [2]

3. Классификатор не должен быть ограничен конкретными анализаторами, т.е. должен иметь возможность обработать и классифицировать предупреждение от любого анализатора после необходимой обработки

Глава 3

Обзор существующих решений и литературы

Необходимость поддержки множества анализаторов обуславливается еще и тем, что каждый анализатор покрывает лишь конечное число уязвимостей, упуская остальные[7] [8] [4]. Поэтому для получения наиболее полного по типам ошибок датасета необходимо уметь обрабатывать сообщения от разных анализаторов. Решением данной проблемы является приведение всех кодов ошибок к кодам CWE [20] на этапе обработки сообщения и использование соответствующего кода CWE в датасете при обучении и классификации. Таким образом, добавление поддержки нового анализатора заключается в трансляции кода ошибок анализатора в CWE. Более подробно данный метод будет описан в дальнейшем в соответствующем разделе 4.1.4.

Sarah Heckman и Laurie Williams в своем исследовании[11] выделили следующие признаки, используемые в наиболее релевантных работах о классификаторах, подобных тому, который является целью данного диплома:

1. Характеристики ошибки - атрибуты предупреждения, сгенерированного статистиким анализатором, например: тип ошибки (double free, etc.), местоположение в коде (файл, класс, функция, строка,

etc.), а также приоритет, который анализатор присвоил ошибке

2. Характеристики кода - метрики кода, в котором содержится предупреждение. Эти метрики могут быть извлечены при помощи дополнительного анализа или из самого кода (цикломатическая сложность, количество строк в файле, etc.)

3. Метрики репозитория с исходным кодом - атрибуты репозитория с исходным кодом (история коммитов, частота изменения кода, история ревью, etc.)

4. Метрики базы данных с багами - информация о багах может быть связана с изменениями в исходном коде для того, чтобы идентифицировать уязвимость и необходимые исправления

5. Метрики динамического анализа - гибридное использование статического и динамического анализа может помочь устранить затраты, связанные с исполнением каждой техники анализа

В данной работе внимание было сосредоточено целиком на первых двух пунктах: характеристики ошибки и характеристики кода. Изучив соответствующие статьи, было выяснено, что основными характеристиками кода для последующего анализа являются метрики, такие как число вложенности функции, число условных переходов, цикломатическая сложность, etc [9]. Однако все эти метрики являются метаданными, которые лишь характеризуют код в целом, не давая понятия о его структуре. Для того, чтобы каким либо образом анализировать структуру кода обычно используется представление AST[17]. Такой подход хорош для таких языков как Java, Python, где

AST может быть легко получено. Для получения же AST в C/C++ необходима полная компиляция проекта. Т.к. каждый проект имеет свои систему для сборки, то невозможно описать данную процедуру единым простым путем. Поэтому в данной работе был опробован подход на основе token-based представления кода, предложенного в [17] для задач определения стиля проекта и вывода правил использования API. Идея состоит в том, что хоть представление в виде токенов и является неточным, но отфильтровав его, и анализируя наряду с метаданными кода, может быть получена более полная картина, нежели чем при использовании только метаданных. Включение токенов в обучающую выборку производилось в предположении, что все false positives следуют определенным шаблонам, которые могут быть определены при помощи предварительно обработанного token-based представления кода. Данное предположение было подробно изучено в работе Zachary P. Reynolds[15], в которой была произведена классификация этих шаблонов для нескольких статических анализаторов. Более подробно о token-based представлении в качестве признаков будет рассказано в соответствующем разделе 4.1.6.

В большинстве рассмотренных работ для классификации используются либо решающие деревья, либо LSTM нейронные сети[9] [18]. В данной работе будет рассмотрен только классификатор на основе решающих деревьев, и решения на основе рекуррентных нейронных сетей являются предметом дальнейших исследований, т.к. могут помочь находить более сложные шаблоны ошибок.

Для разметки датасета предлагаются разные способы. Anshul Tanwar с коллегами в своей работе [18] предлагают способ разметки false positive на основе истории коммитов репозитория. В своей статье

авторы брали исправления разработчиков в течение жизни проекта и сопоставляли их с отчетами статического анализатора. После чего производилась разметка: если исправление касалось куска кода, в котором анализатор находил уязвимость, то ставилась метка true positive, тоесть верное предсказание. Иначе, если разраобтчик помечал данный кусок кода как безопасный, то сообщение анализатора помечалось как false positive. Этот подход обладаем многими преимуществами, такими как больший охват различных сценариев срабатывания анализатора, тоесть обученный на подобных данных классификатор будет более гибким, т.к. запомнит более реалистичные шаблоны, нежели классификатор, обученный на рукописных тестах. Также преимуществом данного подхода является относительная простота реализации. Минусами же данного подхода являются три вещи. Первая - это необходимость наличия сторого формализованной истории коммитов в репозитории, чтобы суметь эффективно извлечь информацию об исправлениях. Вторая - необходимость наличия сообщений, помеченных разработчиками как false positive, что возможно только при должном уровне интеграции статического анализа в процесс разработки, что, как показал предыдущий анализ литературы, является большой редкостью. Третья же вещь - необходимость запускать анализ проекта много раз на разных этапах жизни для корректной разметки, что является очень ресурсоемким и долгим процессом. Другой подход, предложенный в работе Lori Flynn с коллегами [9], избегает многих минусов первого подхода за счет уменьшения охвата кодовой базы. В этой статье авторы использовали Juliet Test Suite - набор рукописных тестов, предназначенных для валидации эффективности различных статистиких анализаторов[1]

для составления и разметки своего датасета. Подход к генерации датасета, используемый в данной работе был позаимствован из этой работы.

Глава 4

Исследование и построение решения задачи

Здесь надо декомпозировать большую задачу из постановки на подзадачи и продолжать этот процесс, пока подзадачи не станут достаточно простыми, чтобы их можно было бы решить напрямую (например, поставив какой-то эксперимент или доказав теорему) или найти готовое решение.

После анализа литературы и уже существующих решений стало ясно, как реализовывать каждую из поставленных ранее задач. Далее будет рассказано о том, какие методы были выбраны для выполнения этих задач. Порядок, в котором они были объявлены в главе 2 сохранен.

4.1 Признаки для датасета

Обсудим признаки, которые были выбраны для обучения классификатора. В предыдущей главе 3 было сказано, что из всех признаков, указанных в [11], данная работа сфокусируется лишь на первых двух - характеристики ошибки и характеристики кода, опуская последние три - метрики репозитория с исходным кодом, метрики базы данных с багами, метрики динамического анализа. Далее будет

описано, почему при составлении датасета был опущен каждый из перечисленных классов признаков, а также будет описано то, какие признаки все же были представлены в датасете в настоящей работе.

4.1.1 Метрики репозитория с исходным кодом

Как было обозначено в обзоре литературы 3, для того, чтобы хоть сколько нибудь эффективно анализировать историю коммитов проекта, она должна быть строго организована, иначе поиск нужных мест в коде сводится к полному анализу всего проекта на каждом из этапов его разработки. В той же главе было объяснено, что в открытом доступе очень малое число проектов соблюдает подобные правила. Плюс, даже при их соблюдении, множественный анализ проекта очень дорог вычислительно и по времени. Плюс, для того, чтобы размечать false positives, необходимо иметь доступ к уже собранной статистике использования анализатора, которой не существует в открытом доступе. Таким образом, анализ истории коммитов и других метрик репозитория отпадает.

4.1.2 Метрики базы данных с багами

Как и в предыдущем пункте, применение данных метрик ограничивается их отсутствием в открытом доступе.

4.1.3 Метрики динамического анализа

Для проведения динамического анализа требуется полная сборка всего проекта, что добавляет сложности к и без того затратному процессу анализа. Также, основным предметом изучения данной работы

является именно статический анализ, поэтому рассмотрение метрик динамического анализа было решено оставить как предмет для будущих исследований.

4.1.4 Характеристики ошибки

Для того, чтобы удовлетворить требованию языковой независимости, все сообщения анализаторов требуется приводить к общему виду. Таким общим видом было выбрано представление Common Weakness Enumeration (CWE)[20]. Таким образом, добавление нового статического анализатора к списку поддерживаемых сводится к добавлению интерфейса, транслирующего код ошибки анализатора в код CWE. К счастью, для большинства распространенных статических анализаторов существуют таблицы соответствия или настройки, позволяющие выводить код ошибки сразу в формате CWE.

4.1.5 Характеристики кода

4.1.6 Token-based представление кода

4.2 Метод генерации и разметки датасета

4.3 Сбор данных

4.4 Обучение классификатора

4.4.1 Decision tree

4.4.2 Gradient boosting

Рассмотрим пример кода Juliet Test Suite:

```
void CWE415_Double_Free__malloc_free_char_08_bad()
```

```
{
    char * data;
    /* Initialize data */
    data = NULL;
    if(staticReturnsTrue())
    {
        data = (char *)malloc(100*sizeof(char));
        if (data == NULL) {exit(-1);}
        /* POTENTIAL FLAW: Free data in the source - the bad sink
        free(data);
    }
    if(staticReturnsTrue())
    {
        /* POTENTIAL FLAW: Possibly freeing memory twice */
        free(data);
    }
}
```

The Juliet Test Suite contains two kinds of metadata that are relevant for determining the validity of alerts: a manifest file: This is an XML file that provides precise flaw information including line number, CWE, and filepath. function names: Documentation for the test suite says that if the function name includes the string GOOD then the particular CWE does not occur in it, but if it includes the string BAD then the CWE does occur in the function. We gathered information about filepath and line numbers covered by each function name that contains GOOD or BAD, as well as the CWE indicated (usually by filename). Note that both the

manifest file and the function names provide only CWE-specific flaw information. In general, a line of code marked BAD for one code flaw type could be flawless with respect to all other code flaw types. Thus, we can use the metadata flaw information to determine the validity of an alert only when we can establish that the alert's checkerID is for a flaw of the same type as a flaw referenced in the metadata. The test suite metadata does not identify every CWE weakness in the Juliet code nor all locations of the CWE weaknesses, so an alert that doesn't map to the test suite metadata cannot be automatically labeled using the metadata. In other words, if an alert's CWE doesn't match the test suite metadata's CWE, the metadata can't be used to label the alert true or false. Publicly-available mappings between checkerIDs and CWEs are available for many of the FFSA tools that we tested. We fused alerts from this set of tools, producing a set of fused alerts with known CWE designations. We then determined verdicts (i.e. classifier ground truth labels) for each fused alert as follows: If the manifest includes a record of a flaw for the same filepath, line number, and CWE as the alert, then set verdict=True, indicating that the alert is a true positive. If the defect alert matches any line within a function name with GOOD and the alert's CWE matches the CWE associated with the function, then set verdict=False, indicating a that the alert is a false positive.

Глава 5

Описание практической части

Если в рамках работы писался какой-то код, здесь должно быть его описание: выбранный язык и библиотеки и мотивы выбора, архитектура, схема функционирования, теоретическая сложность алгоритма, характеристики функционирования (скорость/память).

5.1 Структура проекта

5.2 Пример работы

Глава 6

Заключение

Здесь надо перечислить все результаты, полученные в ходе работы. Из текста должно быть понятно, в какой мере решена поставленная задача.

Список литературы

[1] NSA Center for Assured Software. *Juliet Test Suite*. 2022. URL: `https://samate.nist.gov/SARD/test-suites/112` (дата обр. 28.04.2022).

[2] Nathaniel Ayewah и William W. Pugh. "The Google FindBugs fixit". В: *ISSTA '10*. 2010.

[3] Moritz Marc Beller и др. "Analyzing the State of Static Analysis: A Large-Scale Evaluation in Open Source Software". В: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)* 1 (2016), с. 470—481.

[4] Al Bessey и др. "A few billion lines of code later". В: *Communications of the ACM* 53 (2010), с. 66—75.

[5] bright-tools. *CCSM repository*. 2022. URL: `https://github.com/bright-tools/ccsm` (дата обр. 28.04.2022).

[6] Clang. *Clang Static Analyzer documentation*. 2022. URL: `https://clang-analyzer.llvm.org/` (дата обр. 28.04.2022).

[7] Aurélien Delaitre, Vadim Okun и Elizabeth N. Fong. "Of Massive Static Analysis Data". В: *2013 IEEE Seventh International Conference on Software Security and Reliability Companion* (2013), с. 163—167.

[8] Aurelien Delaitre и др. “Evaluating Bug Finders: Test and Measurement of Static Code Analyzers”. en. В: Complex FaUlts и Failures in LargE Software Systems (COUFLESS), Firenze, -1, 2015-05-23 2015. URL: https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=918370.

[9] Lori Flynn, William Snavely и Zachary Kurtz. “Test Suites as a Source of Training Data for Static Analysis Alert Classifiers”. В: (май 2021).

[10] Lori Flynn и др. “Prioritizing alerts from multiple static analysis tools, using classification models”. В: (май 2018). DOI: 10.1145/3194095.3194100.

[11] Sarah Heckman и Laurie Williams. “A systematic literature review of actionable alert identification techniques for automated static code analysis”. В: *Information and Software Technology* 53.4 (2011). Special section: Software Engineering track of the 24th Annual Symposium on Applied Computing, с. 363—387. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2010.12.007. URL: https://www.sciencedirect.com/science/article/pii/S0950584910002235.

[12] Ugur Koc и др. “Learning a Classifier for False Positive Error Reports Emitted by Static Code Analysis Tools”. В: MAPL 2017 (2017), с. 35—42. DOI: 10.1145/3088525.3088675. URL: https://doi.org/10.1145/3088525.3088675.

[13] David Malcolm. *Static analysis in GCC 10*. 2022. URL: https://gcc.gnu.org/wiki/DavidMalcolm/StaticAnalyzer (дата обр. 28.04.2022).

[14] Michael Pradel и Koushik Sen. "DeepBugs: A Learning Approach to Name-Based Bug Detection". В: *Proc. ACM Program. Lang.* 2.OOPSLA (окт. 2018). DOI: 10.1145/3276517. URL: https://doi.org/10.1145/3276517.

[15] Zachary Reynolds и др. "Identifying and Documenting False Positive Patterns Generated by Static Code Analysis Tools". В: май 2017, с. 55—61. DOI: 10.1109/SER-IP.2017..20.

[16] Joseph R. Ruthruff и др. "Predicting Accurate and Actionable Static Analysis Warnings: An Experimental Approach". В: 2018.

[17] Andrey Y. Shedko и др. "Applying probabilistic models to C++ code on an industrial scale". В: *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops* (2020).

[18] Anshul Tanwar и др. "Assessing Validity of Static Analysis Warnings using Ensemble Learning". В: (апр. 2021).

[19] Cppcheck Team. *cppcheck static source code analysis tool for C and C++ code*. 2022. URL: http://cppcheck.net/ (дата обр. 21.05.2022).

[20] CWE Team. *Common weakness enumeration: A community-developed dictionary of software weakness types*. 2022. URL: http://cwe.mitre.org/documents/schema/index.html (дата обр. 28.04.2022).