УДК 004.415.5
DOI: 10.18523/2617-3808.2020.3.27-30

Бабич Т. А., Гороховський С. С.

ПОБУДОВА ТА ЗБЕРЕЖЕННЯ У ГРАФОВІЙ БАЗІ Neo4j АБСТРАКТНОГО СЕМАНТИЧНОГО ГРАФА ВИХІДНИХ КОДІВ PHP-ДОДАТКІВ

Здійснено реалізацію можливості багаторазового використання графів потоку керування для проведення над ними повторного статичного аналізу (можливість проведення пошуку недоброякісного коду як за стилістикою, так і щодо наявності вразливостей, що їх він створює у результаті). Проведено аналіз декількох граматик для підбору оптимальної, а також порівняння баз даних для конкретної цільової задачі (вибудови графа з великою кількістю точок).

Ключові слова: графові бази даних, статичний аналіз коду, PHP, вразливості, інформаційна безпека.

Вступ

Однією з якісних характеристик програмного забезпечення є наявність різноманітних вразливостей, а також недоброякісного коду, що у результаті може завдавати фінансових та репутаційних збитків як компанії-розробнику, так і кінцевому користувачеві [4]. Існує два підходи до тестування програмного забезпечення:

• Статичний аналіз коду – метод можливий лише за наявності вихідних кодів, адже знаходження помилок у програмному забезпеченні відбувається шляхом автоматичного аналізу вихідного коду, без його безпосереднього виконання.

• Динамічний аналіз програмного забезпечення – визначає дефекти після запуску програми (наприклад, під час модульного тестування). Однак деякі помилки кодування можуть не виникати під час тестування пристрою або бути недоступними за поверхневого тестування і потребуватимуть значно більших часових витрат на виявлення.

Розбір вихідних кодів

Вибудова абстрактного синтаксичного дерева

Вихідним кодом програми є послідовність інструкцій, сформульована у мові програмування як текст. Граматики формальних мов – це набір правил, які описують те, що компілятор вважає правильним вводом: як створити правильні інструкції або набір інструкцій з алфавіту мови (синтаксису). Об'єкт оброблення вихідного коду (трансформатор або компілятор) призначає значення (семантика) і перетворює інструкцію на іншу мову (зазвичай проміжну мову або байт-код) [1].

Які вхідні дані компілятор вважає корисною інформацією, залежить від семантики мови, для якої компілятор будується. Вихідні коди містять набагато ширше розмаїття даних, ніж компілятор потребує для перетворення, аналізуючи додаток: коментарі, порядок декларування функцій, відступ, рядок, що допомагає читачеві коду, але не несе додаткової інформації [5].

Першу фазу процесу розбору називають лексичним розбором. У цій першій фазі програмне забезпечення сканує вхідний потік символів і сегментує їх у послідовність груп, токени, рядки зі «значенням».

Вона також класифікує ці групи в різні класи маркерів, типи маркерів. Оброблення вхідного файлу у значення (перетворення рядка «2» на число 2) також може відбуватися на цій фазі.

Семантика мови програмування допомагає класифікувати різні рівні поведінки. Одним із ключових понять є семантичне поле. Це сукупність понять, зв'язки яких визначають ціле поле споріднених значень. На лексичному рівні ці поняття відображають набір термінів, пов'язаних парадигматичними відносинами в «лексичному полі».

Формальні граматики

Розрізняють два види контекстно-вільних граматик за принципом виконання: розбір згори вгору – LL(k) і згори вгору – LR(k).

LR(k) належить до найбільш загального детермінованого методу синтаксичного аналізу, за якого вхідний рядок обробляється в одному зліва-направо-скануванні, виробляє правий розбір,

із використанням вигляду вершини довжини k. LR(k) парсери є узагальненням недетермінованого парсера зменшення зсуву і простого аналізатора пріоритету.

Побудова абстрактного семантичного графа

Після розбору вихідного коду починається наступна частина аналізу – побудова абстрактного синтаксичного дерева. Абстрактне синтаксичне дерево (АСТ) [3] – це дерево представлення синтаксичної структури результату розбору за граматикою, і, як відомо з побудови компіляторів, забезпечує представлення програм, придатних для аналізу, нечутливого до потоку, наприклад, аналіз типу, аналіз потоку управління і аналіз вказівника. Такий аналіз ігнорує порядок виконання операцій у функції або блоці, що робить АСТ зручним поданням для подальшої роботи над вихідними кодами. Вузли дерева позначають конструкт, що відбувається на вході, тоді як ребра представляють зв'язок між цими вузлами на основі правил граматики. Вихідний код містить набагато більше деталей (наприклад, відступ, пробіл, коментарі), ніж це необхідно, і потребує додаткової уваги під час процесу розбору. Отже, це подання є більш компактним, абстрактним синтаксичним деревом.

Однак ця інформація не є вичерпною у контексті пошуку вразливостей, адже недостатньо знати лише порядок виконання програми, необхідно також враховувати взаємодію з користувачем і даними, що він подає у програму. Аналізатор ділянки або аналізатор контексту виробляє структуру даних, яка представляє інформацію про обсяг програми, розширюючи інформацію, доступну АСТ. Елемент, що представляє схему в цій структурі даних, може містити, серед іншого, метадані про саму структуру, конкретний вузол АСТ, що належить до зони застосування, доступні змінні та функції у цій ділянці.

Абстрактний семантичний граф (АСГ) відрізняється від абстрактного синтаксичного дерева двома основними поняттями:

1. АСГ не обов'язково є деревами.

2. Виражаючи більше, ніж синтаксичну інформацію, АСГ несуть семантичну інформацію, виражену додатковими ребрами та вершинами.

Приклад цього типу краю з'єднує посилання змінних на їх декларації. АСГ – це графічне представлення результату розбору; вузли являють собою не тільки вираз, а й контексти виразу. Компілятори зазвичай працюють на АСГ внутрішньо, оскільки представлена не тільки синтаксична, а й поведінкова інформація.

Використання графових баз даних

У сучасному світі все більше поширюються тенденції використання підходу Not Only SQL (не тільки реляційних баз даних), зокрема, популярності набули графові бази даних. Базуючись на концепції математичного графа, база даних графів містить набір вузлів і вершин.

Вершина являє собою об'єкт, а ребро – з'єднання або зв'язок між двома об'єктами [3]. Кожна вершина у базі даних графів ідентифікується унікальним ідентифікатором, який виражає ключові пари значень. Крім того, кожне ребро визначається унікальним ідентифікатором, який деталізує початкову або кінцеву вершину, а також набір властивостей.

Однією з основ теорії про реляційні бази є наявність зв'язку між сутностями – відношень, а також функцій, що дають змогу ними керувати. У графових базах даних відношення представлені як ребра між вершинами.

Відтворення абстрактного семантичного графа у графовій базі даних

Визначення складових елементів

Для відтворення абстрактного семантичного графа було створено чотири типи сутностей:

- File (файл) – інформація про файл, який було проаналізовано;
- Point (операція) – абстрактне синтаксичне дерево;
- Variable (змінна) – транспорт для роботи між операціями і станами змінних;
- VariableVersion (стан змінної) – екземпляр змінної, що може бути створено операцією присвоєння (кожне нове присвоєння – новий стан змінної).

Першоосновою для змінних слугує змінна Variables, а для операцій – Entry_Point, далі від них ідуть різноманітні розгалуження ребр і вершин, також було розроблено такі зв'язки:

- HAS_ENTRY_POINT – перша інструкція;
- NEXT_OPERATION – наступна ключова інструкція;
- NEXT_SUBOPERATION – інструкція поточного виразу;
- HAS_ARGUMENT – має аргумент, що не використовує змінні;
- CALLED_VARIABLE – звернення до змінної;
- HAS_NAME – назва змінної під час звернення;
- USED_VARIABLE – виклик стану змінної;
- HAS_VARIABLES – ключова змінна, із якою пов'язані всі інші;
- HAS_VARIABLE – зв'язок змінних із першоосновою Variables;

- ASSIGNED – присвоєння змінній нового значення;
- HEAD – перший стан змінної;
- TAIL – останні стан змінної;
- PREV – попередній стан змінної.

Розробка запитів для пошуку

Для перевірки теорії про можливість пошуку та відображення вразливостей було створено спеціально сформовану програму, а також вебдодаток, що дає змогу проаналізувати абстрактний синтаксичний граф.

Першим прикладом слугуватиме пошук вразливості введення стороннього скрипту – XSS. XSS – це ін'єкційна атака, яка дає змогу виконувати сценарії у браузері користувача. Користувач стає жертвою, коли відвідує вебсторінку, яка реалізує злий скрипт. Вебдодаток діє як постачальник сценарію атаки до браузера. Мовою сценаріїв, що в основному використовується, є JavaScript, оскільки вона є ядром, або необхідністю максимального якісного досвіду користування вебсторінками. XSS здебільшого застосовують для знешкодження вебпрограми. Запит, який у своєму алгоритмі використовує вивід користувацьких даних із GET запиту:

```
OPTIONAL MATCH direct_call = (:Point {type:
«Stmt_Echo»})-[:NEXT_SUBOPERATION]->(a:Point
{type: «Expr_ArrayDimFetch»})-[:CALLED_
VARIABLE]->(:Point {name: «_GET»})
OPTIONAL MATCH call_via_variables = (:Point
{type: «Stmt_Echo»})-[:CALLED_VARIABLE]->()-
[:USED_VARIABLE]->(:VariableVersion)<-
[:ASSIGNED]-(:Point)<-[*]-(:Point {type:
«Expr_Assign»})-[*]->(:Point {type: «Expr_
ArrayDimFetch»})-[:CALLED_VARIABLE]->(:Point
{name: «_GET»})
WHERE NONE(oneNode IN NODES(direct_call)
WHERE oneNode:Point AND oneNode.type =
«Expr_FuncCall» AND oneNode.name IN [«strip_
tags», «htmlspecialchars», «filter_var»])
OR NONE(oneNode IN NODES(call_via_variables)
WHERE oneNode:Point AND oneNode.type =
«Expr_FuncCall» AND oneNode.name IN [«strip_
tags», «htmlspecialchars», «filter_var»])
RETURN direct_call, call_via_variables
```

Завдяки цьому запиту можна отримати всі вразливі шляхи.

Також акцент було зроблено на пошуку коду, що не відповідає стилістичним правилам написання. Ці правила формалізовано у конвенціях стилю написання коду, як приклад було взято одне з правил статичного аналізатора коду на якість – Sonar. Вимога програмування випадку за замовчуванням для інструкції перемикання (switch) – це приклад захисного програмування. Програміст у своїх вихідних кодах має вживати відповідних заходів або написати відповідний коментар про те, чому не зроблено жодних дій. Навіть якщо перемикач охоплює всі поточні значення переліку, випадок за замовчуванням все одно має використовуватися, оскільки немає гарантії, що перелік не буде розширено. Запит на виявлення недоброякісного коду, що потрапляє у цю ситуацію, такий:

```
MATCH (switch:Point {type: «Stmt_Switch»})
WHERE NOT (switch)-[:NEXT_SUBOPERATION]->
(:Point {type: "Stmt_Case", default: true})
RETURN switch
```

Висновки

Розроблене рішення може перетворити досить велике сховище вихідного коду в цілому на графічне представлення і підтримувати його згодом. Виявлено, що підхід є придатним для виконання перевірок відповідності кодовим умовам і для виконання тестів статичного аналізу на графічному поданні. Цей підхід також використовує додаткове оброблення з деталізацією на рівні файлів, прискорюючи статичний аналіз. Дані замірів швидкодії щодо створення та використання такого рішення підтверджують, що швидкість виконання усіх дій, від аналізу коду до побудови у базі даних, є достатньою, щоб надавати змогу швидко аналізувати вихідний код.

Список літератури

1. Aho A. Compilers: Principles, Techniques, and Tools / A. Aho, M. Lam, R. Sethi, J. Ullman. – Boston : Pearson/Addison Wesley, 2007.
2. Dullien T. REIL: a platform-independent intermediate representation of disassembled code for static code analysis / T. Dullien, S. Porst. – CanSecWest, 2009.
3. Grune D. Parsing Techniques: a practical guide / D. Grune, C. Jacobs. – New York; London : Springer, 2008.
4. Stamelos I. Code quality analysis in open source software development / I. Stamelos, L. Angelis, A. Oikonomou, Georgios L. Bleris // Information Systems Journal. – No. 12. – P. 43–60.
5. Wichmann B. Industrial perspective on static analysis / B. Wichmann, A. Canning, D. Clutterbuck, L. Winsborrow, N. Ward, R. Marsh // Software Engineering Journal. – 1995. – No. 10. – P. 69–75.

References

Aho, A., Lam, M., Sethi, R., Ullman, J., & Aho, A. (2007). Compilers: principles, techniques, & tools. Boston: Pearson/Addison Wesley.

Dullien, T., & Porst, S. (2009). REIL: A platform-independent intermediate representation of disassembled code for static code analysis. CanSecWest.

Grune, D., & Jacobs, C. (2005). Parsing techniques: a practical guide. New York; London: Springer.

Stamelos, Ioannis, Angelis, Lefteris, Oikonomou, Apostolos, & Bleris, Georgios L. (2002). Code quality analysis in open source software development. *Information Systems Journal*, *12*, 43–60.

Wichmann, B., Canning, A., Clutterbuck, D., Winsborrow, L., Ward, N., & Marsh, W. (1995). Industrial perspective on static analysis. *Software Engineering Journal*, *10*, 69–75.

T. Babych, S. Gorokhovskyi

BUILDING AND STORING IN GRAPH DATABASE Neo4j ABSTRACT SEMANTIC GRAPH OF PHP APPLICATIONS SOURCE CODE

Static code analysis is a very important stage in the development and implementation of software, and it needs to be used to obtain a better code. The most complicated in static analysis is the analysis of source code, and further analysis. Analysis of the source code can be done multiple times and expand the set of necessary grammar. The main goal was and remains to provide a solution to reduce the time needed for a global reassessment of static analysis at code level after a change.

The main problem when using static code analysis is to build an abstract semantic graph, because each software solution is provided with a separate data warehouse. The proposed solution proposes to use as a database repository using graphs. Thus, the storage mechanisms of the created abstract semantic graph have been simplified, which in turn, in addition to increasing the clarity of the information we store, provides convenient ways to further work with the stored information.

The developed solution can transform a rather large source repository into a graphic representation and maintain it later. It is found that the approach is suitable for carrying out code compliance checks and for performing static analysis tests on a graphic representation. This approach also uses advanced file-level detailing, accelerating static analysis. Based on my measurements, the frameworks are fast enough to help their users quickly change the repository of codes.

This article confirms the thesis about the possibility of storing an abstract semantic graph in a graph database, and after refinement, if it contains sufficient transformations and requests for language processing, can become a complete transport for communication between various static analysis tools that usually perform one of two the functions of verification either for quality or for vulnerability, thereby making a unified creation of an abstract semantic graph.

As an improvement it is necessary to consider the possibility of incremental analysis – the analysis of changes in the code, in order to minimize resource costs for a rather resource intensive operation of the structure of the abstract semantic graph.

Keywords: graph database, static code analysis, php, vulnerability, information security.

Матеріал надійшов 09.06.2020