SYSTEMS AND METHODS OF INFORMATION PROTECTION
СИСТЕМИ І МЕТОДИ ЗАХИСТУ ІНФОРМАЦІЇ

УДК 004.056.5 **DOI:10.30837/rt.2023.1.212.01**

А.О. ГАПОН, В.М. ФЕДОРЧЕНКО, канд. техн. наук,
О.В. СЄВЄРІНОВ, канд. техн. наук

МЕТОДИ ТА ЗАСОБИ СТАТИЧНОГО ТА ДИНАМІЧНОГО АНАЛІЗУ КОДУ

Вступ

Проблема якості коду є однією з базових, що впливають на якість програмного продукту. Існують засоби контролю та покращення його якості за допомогою як ручних, так і автоматизованих підходів. Процедура перегляду коду - це ручний підхід перевірки якості коду. Ця практика є корисною та необхідною під час розробки програмного продукту, але у той же час може займати багато часу та не прибирає людський фактор. Автоматизовані засоби аналізу коду є ключовими інструментами, які доповнюють ручний перегляд коду та забезпечують високу якість коду та, відповідно, його безпечність.

Статичний аналіз коду

Статичний аналіз програмного коду використовується з початку 1960-х років для оптимізації роботи компіляторів. Пізніше це виявилося корисним для інструментів налагодження, а також для фреймворків розробки програмного забезпечення. Зростає кількість інструментів, які дозволяють використовувати статичний аналіз коду, деякі з яких є інструментами з відкритим кодом і дозволяють аналізувати кілька різних мов програмування.

Інструменти статичного аналізу використовуються для створення звітів і виявлення певних відхилень від встановлених стандартів якості коду. Однак ці інструменти не дозволяють автоматично модифікувати вихідний код. Рішення змінити спосіб структурування раніше написаного коду залишається в руках розробників програмного забезпечення.

Інструменти статичного аналізу коду допомагають розробникам програмного забезпечення, створюючи звіт, де вказується причина певного дефекту, а також те, як цей дефект можна виправити.

Проте залишається відкритим питання, як підійти до недоліків, усунути виділені недоліки та переробити вихідний код.

Коли справа стосується інструментів статичного аналізу коду, великою проблемою є те, що існує велика кількість інструментів, які забезпечують цей тип аналізу. Кількість інструментів постійно збільшується, і необхідно класифікувати існуючі інструменти відповідно до мови програмування, яку вони підтримують, і типів дефектів, які вони виявляють.

Процес статичного аналізу коду корисний не лише для оптимізації роботи компілятора (що було початковою метою), але й для виявлення невідповідностей і можливих дефектів. Таким чином, можна створити інструменти, які допоможуть розробникам зрозуміти поведінку програми та виявити різні дефекти програми без її виконання. Інструментами, які використовуються для статичного аналізу коду, є програми, які пояснюють поведінку інших програм [1].

Статичний аналіз коду значно швидший, ніж звичайне тестування, і може виявити будь-який дефект, видимий у вихідному коді програми. Якщо порівняти інструменти для статичного аналізу коду з ручним переглядом коду розробниками програмного забезпечення, можна зробити висновок, що перегляд коду за допомогою інструменту також є набагато швидшим та ефективнішим. Однак щоб статичний аналіз коду виявив будь-який дефект, він повинен бути видимим у вихідному коді.

Стандартний цикл перевірки коду включає чотири основні фази:

1) встановити цілі;
2) запустити інструмент статичного аналізу;
3) переглянути код;
4) зробити рефакторінг.

Окрім стандартного циклу на рис. 1 показано кілька потенційних зворотних зв'язків або незначних ітерацій між стандартними кроками циклу, що робить цикл складнішим, ніж стандартна процедура.

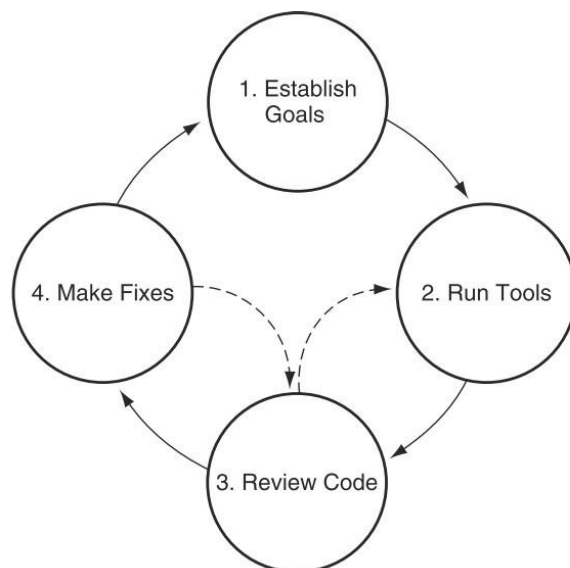

Рис. 1. Цикл огляду коду

Завдання, які вирішуються програмами статичного аналізу коду, можна розділити на три категорії:

1. Виявлення помилок у програмах.

2. Рекомендації щодо оформлення коду. Деякі статичні аналізатори дозволяють перевіряти, чи відповідає вихідний код, прийнятий у компанії, стандарту оформлення коду. Мається на увазі контроль кількості відступів у різних конструкціях, використання пробілів/символів табуляції тощо.

3. Підрахунок метрик. Метрика програмного забезпечення – це міра, що дозволяє отримати чисельне значення певної якості програмного забезпечення або його специфікацій. Існує велика кількість різноманітних метрик, які можна підрахувати, використовуючи інші інструменти.

Серед недоліків статичного аналізу коду можна виділити:

1. Статичний аналіз, як правило, слабкий у діагностиці витоків пам'яті та паралельних помилок. Щоб виявляти такі помилки, фактично необхідно віртуально виконати частину програми. Це дуже складно реалізувати. Також подібні алгоритми вимагають дуже багато пам'яті та процесорного часу. Як правило, статичні аналізатори обмежуються діагностикою найпростіших випадків. Більш ефективним способом виявлення витоків пам'яті та паралельних помилок є використання інструментів динамічного аналізу.

2. Програма статичного аналізу попереджає про підозрілі місця. Це означає, що насправді код може бути абсолютно коректний. Це називається хибно позитивними спрацьовуваннями. Зрозуміти, чи вказує аналізатор на помилку чи видав хибне спрацювання, може лише програміст. Необхідність переглядати помилкові спрацьовування забирає робочий час і послаблює увагу до тих ділянок коду, де насправді містяться помилки.

Методи та засоби статичного аналізу коду

Одним із основних алгоритмів статичного аналізу є data-flow analysis (аналіз потоку даних). Завдання такого аналізу – визначити в кожній точці програми деяку інформацію про дані, якими оперує код. Інформація може бути різною, наприклад, за типом даних або значенням. Залежно від цього, яку інформацію потрібно визначити, можна сформулювати завдання аналізу потоку даних.

Наприклад, якщо потрібно визначити, чи є вираз константою, і навіть значення цієї константи, то вирішується завдання поширення констант (constant propagation). Якщо необхідно визначити тип змінної, можна говорити про завдання поширення типів (type propagation). Якщо необхідно зрозуміти, які змінні можуть вказувати на певну область пам'яті (зберігати ті самі дані), то йдеться про завдання аналізу синонімів (alias analysis). Існує безліч інших завдань аналізу потоку даних, які можуть використовуватись у статичному аналізаторі. Як і етапи побудови моделі коду, ці завдання також використовуються в компіляторах [2].

У теорії побудови компіляторів описані рішення задач внутрішньопроцедурного аналізу потоку даних (відстежити дані необхідно у межах однієї процедури/функції/методу). Рішення спираються на теорію алгебраїчних решіток та інші елементи математичних теорій. Вирішити задачу аналізу потоку даних можна за поліноміальний час, тобто за прийнятний для обчислювальних машин час, якщо умови задачі задовольняють умовам теореми про дозвіл, що на практиці відбувається далеко не завжди.

Symbolic execution (символьне виконання) – цей метод передбачає абстрактний рух шляхами програми, що імітує її виконання залежно від вхідних даних, що супроводжується зміною стану програми у різних точках. Суть методу символьного виконання полягає в розбитті безлічі вхідних даних на класи еквівалентності, що дозволяє оперувати при аналізі не окремими вхідними значеннями (число яких може бути дуже великим та експоненційно зростати залежно від кількості вхідних аргументів) та їх перебором, а цілими класами еквівалентності, число яких може бути і не кінцевим, але не перевищує загальну кількість комбінацій окремих вхідних значень. Однак, як правило, кількість класів еквівалентності комбінацій вхідних даних виявляється значно нижчою від числа всіх можливих комбінацій вхідних даних, що різко збільшує можливості аналізатора з обробки шляхів виконання.

Abstract interpretation (абстрактна інтерпретація) – це теорія звукової апроксимації семантики комп'ютерних програм, заснована на монотонних функціях над впорядкованими множинами, особливо решітками. Його можна розглядати як часткове виконання комп'ютерної програми, яка отримує інформацію про свою семантику (наприклад, потік управління, потік даних) без виконання всіх обчислень.

Його основне конкретне застосування – формальний статичний аналіз, автоматичне вилучення інформації про можливе виконання комп'ютерних програм; такі аналізи мають основні застосування:

1) всередині компіляторів для аналізу програм, щоб вирішити, чи можуть бути застосовані певні оптимізації або перетворення;

2) для налагодження чи навіть сертифікації програм проти класів помилок.

Для мови програмування чи специфікації абстрактна інтерпретація складається із надання кількох семантик, пов'язаних відносинами абстракції. Семантика – це математична характеристика можливої поведінки програми. Найбільш точна семантика, що дуже точно описує фактичне виконання програми, називається конкретною семантикою.

Мета статичного аналізу полягає в тому, щоб в якийсь момент отримати семантичну інтерпретацію, що обчислюється. Наприклад, можна вибрати уявлення про стан програми, що маніпулює цілісними змінними, забувши фактичні значення змінних і зберігши тільки їх знаки (+, – або 0). Для деяких елементарних операцій, таких як множення, така абстракція не втрачає точності: щоб отримати знак твору достатньо знати знак операндів. Для інших операцій абстракція може втратити точність: наприклад, неможливо дізнатися знак суми, операнди якої відповідно позитивні і негативні.

Необхідно знайти компроміс між точністю аналізу та її розв'язаністю (обчислюваністю) чи зручністю читання (обчислювальними витратами).

Насправді певні абстракції адаптуються як до властивостей програми, які потрібно проаналізувати, так і до набору цільових програм.

Динамічний аналіз коду

Динамічний аналіз коду – це метод аналізу програми безпосередньо під час виконання. Звідси випливає, що з вихідного коду в обов'язковому порядку має бути отриманий файл, що виконується, тобто не можна в такий спосіб проаналізувати код, що містить помилки компіляції або збірки. Динамічний аналіз виконується за допомогою набору даних, що подаються на вхід досліджуваної програми. Тому ефективність аналізу безпосередньо залежить від якості та кількості вхідних даних для тестування. Саме від них залежить повнота покриття коду, яка буде одержана за результатами тестування [3].

Використовуючи динамічне тестування, можна отримати такі метрики та попередження:

1. Ресурси, що використовуються: час виконання програми в цілому або її окремих модулів, кількість зовнішніх запитів (наприклад, до бази даних), кількість використовуваної оперативної пам'яті та інших ресурсів.

2. Ступінь покриття коду тестами та інші метрики програми.

3. Програмні помилки: розподіл на нуль, розіменування нульового покажчика, витоку пам'яті, "стан гонки".

4. Детектування деяких вразливостей.

До основних переваг динамічного аналізу коду відносять:

1. Можливість проводити аналіз програми без необхідності доступу до вихідного коду. Тут варто зробити застереження, так як програми для динамічного аналізу розрізняють за способом взаємодії з програмою, що перевіряється. Наприклад, поширений спосіб проведення динамічного аналізу шляхом попереднього інструментування вихідного коду. У цьому випадку доступ до коду програми, що перевіряється, буде необхідний.

2. Можливість виявлення складних помилок, пов'язаних із роботою з пам'яттю: вихід за обсяги масиву, виявлення витоків пам'яті.

3. Можливість проводити аналіз багатопоточного коду безпосередньо в момент виконання програми, тим самим виявляти потенційні проблеми, пов'язані з доступом до ресурсів, що розділяються; можливі deadlock ситуації.

4. У більшості реалізацій поява хибних спрацьовувань неможлива, оскільки виявлення помилок відбувається під час виконання програми, таким чином, виявлена помилка не є передбаченням, зробленим з урахуванням аналізу моделі програми, а констатацією факту її виникнення.

Перерахуємо недоліки, які притаманні динамічному аналізу коду:

1. Не можна гарантувати повного покриття коду, тобто, швидше за все, відсоток коду програми, який був проаналізований у процесі динамічного тестування, не буде рівним ста відсоткам.

2. Практично не виявляються помилки логічного типу. Наприклад, з погляду динамічного аналізатора, завжди справжня умова не є помилкою, оскільки така некоректна перевірка просто зникає ще на етапі компіляції програми.

3. Тяжко локалізувати місце з помилкою у вихідному коді.

4. Більш висока складність використання порівняно зі статичним аналізом, оскільки для досягнення більшої ефективності динамічного аналізу програмі, що тестується, потрібно подання достатньої кількості вхідних даних, щоб отримати більш повне покриття коду.

Динамічне тестування найбільш важливо в тих областях, де головним критерієм є надійність програми, час відгуку або споживані ресурси. Це може бути, наприклад, система реального часу, що управляє відповідальною ділянкою виробництва, або сервер бази даних. У таких областях будь-яка допущена помилка може бути критичною.

Методи та засоби динамічного аналізу коду

Метод *інструменталізації.* Контрольно-вимірювальні прилади виконують вимірювання програми. Це та сама основа, на якій ми можемо відслідковувати, усувати неполадки, налагоджувати, профільтрувати та розуміти, як працюють програми та чому вони працюють певним чином.

На практиці інструменталізація може бути такою ж простою, як запис часу, в який відбувається виконання функції, та його реєстрація, щоб допомогти звузити місце, де існує вузьке місце у продуктивності [4].

Якщо програма споживає більше пам'яті, ніж очікувалося, отримання зразків використання пам'яті з часом (або зразків розподілу пам'яті та показників складання сміття) також може надати цінну інформацію для відстеження витоку пам'яті. Запис вхідних та вихідних даних функцій або цілих систем (наприклад, корисного навантаження запиту та відповіді служби HTTP) може допомогти у налагодженні програм з несподіваними вхідними даними або неправильною логікою. Це лише деякі приклади використання, які показують, наскільки важливим є інструментування для виявлення та вирішення проблем за допомогою моніторингу та усунення несправностей.

Є два способи додати інструменталізацію у програми: вручну чи автоматично. На продуктивність працюючого коду впливає як сам факт впровадження інструментування, так і те, як саме команда реалізує інструментування.

Один із способів написання коду приладу – ручна інструменталізація. Написати додатковий код для виконання вимірювань разом із рештою коду програми. Однак інструментування фрагментів коду вручну може бути важким завданням. І в міру того, як проєкти та команди збільшуються в розмірах, стає дедалі важче стандартизувати, що і як потрібно використовувати. Крім того, все, що робиться вручну і повторюється, схильне до помилок: або про цей метод забувають у деяких місцях, або реалізовують неправильно.

Більшість служб моніторингу та усунення несправностей надають спеціалізовані автоматизовані засоби інструментування коду: для мов, середовищ виконання та фреймворків.

Автоматичне інструментування зазвичай реалізується шляхом додавання проміжного програмного забезпечення, яке обертає певні важливі фрагменти коду логікою інструментування. Типовим прикладом є проміжне програмне забезпечення для HTTP-запиту, яке вимірює час витрат на отримання відповіді, а також інформацію про запит та відповідь, таку як код стану та корисні дані.

Підтримка автоматичного інструментування на різних мовах програмування може відрізнятися. Динамічні інтерпретовані мови часто використовують такі методи для додавання автоматичного інструментарію, у той час як мови, що компілюються за допомогою байт-коду, такі як Java, дозволяють модифікувати байт-код під час виконання для досягнення того ж ефекту.

Рекомендується використовувати автоматичну інструменталізацію. Це часто забезпечує набагато краще охоплення з розумними значеннями за умовчанням, ніж самостійна робота, і практично не вимагає часу для реалізації та обслуговування. Ручну інструменталізацію варто залишити для окремих випадків, якщо такі є, які не охоплені автоматичним підходом.

Серед засобів інструменталізації можна виділити:

1. Трасування коду – отримання інформаційних повідомлень про виконання програми протягом його роботи.

2. Налагодження програми та (структурована) обробка винятків – відстеження та виправлення помилок програмістів у додатку ще на стадії його розробки.

3. Профілювання – набір методик відстеження продуктивності коду, включаючи вимірювання.

4. Лічильники продуктивності – це компоненти, що дозволяють відстежувати рівень продуктивності програми.

5. Реєстратори подій – компоненти, які дозволяють отримувати сповіщення та відстежувати ключові події під час виконання програми.

Інший метод це – *динамічне тестування*. Динамічний аналіз відноситься до вивчення фізичної реакції системи на змінні, які не є постійними та змінюються з часом. При динамічному тестуванні програмне забезпечення має бути скомпільоване та запущене. Воно включає роботу з програмним забезпеченням, введення вхідних значень і перевірку відповідності виводу очікуваним шляхом виконання конкретних тестових випадків, які можна виконати вручну або з використанням автоматизованого процесу. Це відрізняється від статичного тестування.

Процес та функції динамічного тестування у розробці програмного забезпечення можна розділити на модульне тестування, інтеграційне тестування, системне тестування, приймальне тестування та регресійне тестування.

Модульне тестування – це тест, який фокусується на правильності основних компонентів програмного забезпечення. Модульне тестування відноситься до категорії тестування білої скриньки. У всій системі контролю якості модульне тестування має бути завершено групою продуктів, а потім програмне забезпечення передається до відділу тестування.

Інтеграційне тестування використовує визначення того, чи правильно підключені інтерфейси між різними модулями у процесі інтеграції всього програмного забезпечення.

Тестування програмної системи, яка завершила інтеграцію, називається системним тестом, і мета тесту – переконатися, що правильність та продуктивність програмної системи відповідають вимогам, зазначеним у її специфікаціях. Тестувальники повинні дотримуватися встановленого плану тестування. При тестуванні надійності та простоти використання програмного забезпечення його введення, виведення та іншу динамічну робочу поведінку слід порівнювати зі специфікаціями програмного забезпечення. Якщо специфікація програмного забезпечення неповна, системний тест більше залежить від досвіду роботи та суджень тестувальника, такого тесту недостатньо. Системний тест – це тестування «чорної скриньки».

Приймальне тестування – це останній тест перед введенням програмного забезпечення в експлуатацію. Це пробний процес перевірки програмного забезпечення покупцем. У реальній роботі компанії це зазвичай реалізується шляхом звернення до замовника із проханням спробувати чи випустити бета-версію програмного забезпечення. Приймальне тестування – це тестування методом «чорного ящика».

Метою регресійного тестування є перевірка та зміна результатів приймального тестування на етапі обслуговування програмного забезпечення. Наприклад, обробка скарг клієнтів є здійсненням регресійного тестування.

Роль статичного та динамічного аналізу у безпеці

Зараз безпека є ключовим питанням розробки програмного забезпечення для різноманітних пристроїв та є важливим аспектом конкурентоспроможності бізнесу. Розробка безпеки в продукті на ранніх етапах має вирішальне значення для зменшення як ризику, так і вартості подальших загроз безпеки. Інструменти статичного та динамічного аналізу коду відіграють важливу роль у повному наборі інструментів програмного забезпечення та допомагають прискорити розробку безпечного програмного забезпечення.

Атака на підприємство може знизити продуктивність, зв'язати ресурси, зашкодити довірі та знизити прибутки. Оскільки більшість сучасних загроз спрямовані на прикладний рівень, аналіз безпеки коду є обов'язковим для будь-якої конкурентоспроможної організації. Аналіз додатків шукає в програмному забезпеченні такі вразливості, як бекдори додатків або зловмисний код, щоб їх можна було виправити, перш ніж їх виявлять і використають хакери.

Висновки

Статичний та динамічний аналізи коду є важливими інструментами аналізу коду, які доповнюють один одного та виводять рівень безпеки коду на новий рівень. При поєднанні аналізу цих підходів формується новий підхід – гібридний аналіз коду, оскільки поєднуючи

ці підходи, можна з'ясувати більш повну картину, ніж використовуючи їх окремо, бо аналізується повна поведінка програмного забезпечення.

Список літератури:

1. OWASP [Електронний ресурс]. Режим доступу: //owasp.org/www–community/controls/Static_Code_Analysis
2. Citeseerx[Електронний ресурс]. Режим доступу: https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.394.5540
3. Techopedia [Електронний ресурс]. Режим доступу: https://www.techopedia.com/definition/30958/dynamic–application–security–testing–dast
4. Contrast security[Електронний ресурс]. Режим доступу: https://www.contrastsecurity.com/glossary/dynamic–application–security–testing

Надійшла до редколегії 04.03.2023

Відомості про авторів:

Гапон Андрій Олександрович – Харківський національний університет радіоелектроніки, аспірант кафедри безпеки інформаційних технологій; Україна; e-mail: gapon.andrei@gmail.com; ORCID: https://orcid.org/0000-0003-2560-7426

Федорченко Володимир Миколайович – канд. техн. наук, доцент, Харківський національний університет радіоелектроніки, доцент кафедри електронних обчислювальних машин, факультет комп'ютерної інженерії та управління, Україна; e-mail: volodymyr.fedorchenko@nure.ua, ORCID: https://orcid.org/0000-0001-7359-1460

Сєвєрінов Олександр Васильович – канд. техн. наук, доцент, Харківський національний університет радіоелектроніки, доцент кафедри безпеки інформаційних технологій, факультет комп'ютерної інженерії та управління, Україна; e-mail: oleksandr.sievierinov@nure.ua, ORCID: https://orcid.org/0000-0002-6327-6405