ПРАКТИЧЕСКАЯ РАБОТА. АНАЛИЗАТОРЫ КОДА

1.1. Цель и задачи практической работы

Цель работы: ознакомиться с основными принципами и методами использования статических и динамических анализаторов кода для раннего выявления ошибок и потенциальных уязвимостей, что позволит повысить качество, безопасность и надёжность программного обеспечения.

Для достижения поставленной цели работы студентам необходимо выполнить ряд **задач**:

1. Изучить теоретические основы статического и динамического анализа кода.
2. Ознакомиться с популярными инструментами статического анализа (например, ESLint, Pylint, Checkmarx, SonarQube, FindBugs, TSLint, Cppcheck) и динамического анализа (например, Valgrind, DynamoRIO, Java VisualVM, Burp Suite, OWASP ZAP).
3. Применить выбранные анализаторы к ранее разработанным учебным проектам на разных языках программирования.
4. Провести анализ исходного кода до и после внесения целенаправленных ошибок, оценить адекватность обнаружения дефектов.
5. Сформировать детальный отчёт с критическим анализом результатов, выводами о преимуществах и ограничениях каждого подхода.

1.2. Теоретический раздел

1.2.1. Статический анализ кода

Статический анализ — это метод анализа программного кода без его исполнения. Его основная задача — обнаружить потенциальные ошибки, нарушения стандартов кодирования, неэффективные или опасные конструкции и утечки ресурсов непосредственно в исходном коде.

Основные характеристики:

1. Позволяет выявить ошибки ещё до запуска программы, что снижает затраты на их исправление.
2. Результаты анализа оформляются в виде отчётов, включающих список найденных проблем, ссылки на документацию, описание потенциальных рисков и рекомендации по исправлению.

Ниже представлен ряд инструментов для проведения статического тестирования, относящихся как к open source, так и нет.

ESLint — проверяет JavaScript и TypeScript-код на предмет соблюдения стандартов, синтаксических ошибок и логических недочётов. Является проектом с открытым исходным кодом, распространяемым под лицензией MIT. ESLint разработан международным сообществом.

Pylint анализирует Python-код, оценивая стиль, структуру и потенциальные ошибки в логике программы. Является проектом с открытым исходным кодом, распространяемым под лицензией GNU GPL. Pylint разработан международным сообществом.

Checkmarx представляет собой многофункциональную платформу, фокусирующуюся на обнаружении уязвимостей безопасности в коде на различных языках. Это коммерческий продукт, не являющийся проектом с открытым исходным кодом. Checkmarx разработан международной компанией.

SonarQube — универсальный инструмент, предоставляющий детальные отчёты по качеству кода для множества языков программирования. Имеет как версию с открытым исходным кодом (Community Edition), так и коммерческие версии с расширенными возможностями. SonarQube разработан международной компанией SonarSource.

FindBugs — специализированный анализатор для Java, выявляющий типичные проблемы, такие как возможные null-указатели и утечки ресурсов. Является проектом с открытым исходным кодом и распространяется под лицензией LGPL. FindBugs разработан международным сообществом.

TSLint — инструмент для TypeScript, следящий за соблюдением стиля и выявляющий специфичные для языка ошибки. Является проектом с открытым исходным кодом, распространяемым под лицензией Apache 2.0. TSLint разработан международным сообществом (на сегодняшний день его функциональность постепенно интегрируется в ESLint).

Cppcheck — анализатор для C и C++, проверяющий код на наличие ошибок, связанных с управлением памятью и неверными указателями. Является проектом с открытым исходным кодом, распространяемым под лицензией GPL. Cppcheck разработан международным сообществом.

1.2.2. Динамический анализ кода

Динамический анализ представляет собой процесс изучения поведения программы во время её исполнения. Этот метод позволяет выявить ошибки, которые

не обнаруживаются статическим анализом, например, утечки памяти, ошибки исполнения и проблемы с производительностью.

Основные характеристики:

1. Позволяет оценить, как программа работает в условиях, приближенных к боевым, выявляя проблемы, связанные с взаимодействием модулей и ресурсами системы.

2. Результаты динамического анализа оформляются в виде отчётов, включающих данные о профилировании, использовании памяти, времени выполнения и других аспектах работы программы.

Примеры инструментов:

1. Valgrind выполняет динамический анализ кода, в основном для C и C++ приложений, обнаруживая утечки памяти, ошибки работы с памятью, некорректное использование указателей и другие проблемы, возникающие во время исполнения программы. Является проектом с открытым исходным кодом, распространяемым под лицензией GPL. Valgrind разработан международным сообществом.

2. DynamoRIO представляет собой платформу для динамического анализа и инструментальную среду для создания собственных инструментов анализа исполняемого кода. Является проектом с открытым исходным кодом, распространяемым под лицензией BSD. DynamoRIO разработан международным сообществом.

3. Java VisualVM — инструмент для динамического анализа и профилирования Java-приложений, позволяющий отслеживать использование памяти, время выполнения и другие параметры работы виртуальной машины Java. Является проектом с открытым исходным кодом, доступным бесплатно. Java VisualVM разработан международным сообществом (поддерживается Oracle/OpenJDK).

4. Burp Suite — набор инструментов для динамического анализа веб-приложений, ориентированный на обнаружение уязвимостей, таких как SQL-инъекции и XSS-атаки. Имеет как бесплатную (Community Edition), так и коммерческую версию с расширенными функциями. Burp Suite разработан международной компанией PortSwigger.

5. OWASP ZAP — динамический анализатор для веб-приложений, предназначенный для автоматического обнаружения уязвимостей. Является проектом с открытым исходным кодом, распространяемым под лицензией Apache 2.0, и разработан международным сообществом в рамках проекта OWASP.

Необходимо чётко различать инструменты для статического и динамического анализа. Например, ESLint предназначен только для статического анализа, и его нельзя использовать для мониторинга работы кода в режиме исполнения.

1.3. Описание работы

Подготовительный этап работы включает следующие шаги:

1. Выбрать два ранее разработанных учебных проекта, написанных на разных языках программирования (например, один на Python и другой на Java или C/C++). Можно использовать проекты с более ранних курсов.
2. В отчёте кратко описать функциональность каждого проекта, указать его особенности и прикрепить исходный код (либо непосредственно в документе, либо посредством ссылок на репозитории).

1.3.1. Часть 1. Статический анализ кода

Для выполнения 1 части работы необходимо выполнить следующие шаги:

1. Провести анализ каждого приложения с использованием не менее трёх статических анализаторов, выбирая инструменты в зависимости от языка проекта (например, для проекта на Python — Pylint, SonarQube и Checkmarx; для проекта на C/C++ — Cppcheck, SonarQube и, возможно, специализированный анализатор для языка).
2. Сохранить полученные отчёты, в которых подробно описаны найденные ошибки, предупреждения и рекомендации по их устранению.
3. Провести сравнительный анализ отчётов, оценив, какие дефекты обнаружены всеми инструментами, а какие — только отдельными, а также проанализировать ложные срабатывания.
4. Внести в каждый проект пять целенаправленных ошибок (например, нарушения стандартов кодирования, логические ошибки, ошибки управления памятью, неправильную обработку исключений, ошибки синтаксиса).
5. Провести повторный статический анализ с использованием тех же инструментов.
6. Сравнить результаты: зафиксировать, какие из внесённых ошибок были обнаружены всеми инструментами, а какие — пропущены. Это позволит оценить адекватность каждого анализатора.

7. Сформулировать вывод о том, насколько статический анализ эффективен для раннего выявления ошибок, указать его преимущества (быстрота, интеграция в CI/CD) и ограничения (невозможность обнаружения ошибок, возникающих только при выполнении программы).

1.3.2. Часть 2. Динамический анализ кода

Для выполнения 2 части работы необходимо выполнить следующие шаги:

1. Проанализировать работу каждого приложения с использованием двух динамических анализаторов, подбирая инструменты в зависимости от особенностей проекта (например, для C/C++ — Valgrind, для Java — Java VisualVM, для веб-приложений — Burp Suite или OWASP ZAP).
2. Получить отчёты, фиксирующие ошибки, возникающие во время исполнения, такие как утечки памяти, сбои выполнения, проблемы с производительностью.
3. Внести в каждый проект по три ошибки, которые проявляются только при выполнении кода (например, ошибки управления памятью, некорректная обработка исключений или неправильное взаимодействие между компонентами).
4. Провести повторный динамический анализ и сравнить результаты с первичным запуском.
5. Оценить, насколько обнаруженные ошибки соответствуют внесённым, и выявить возможные недочёты анализаторов.
6. Сформулировать вывод о полезности динамического анализа для обнаружения ошибок, возникающих в реальных условиях исполнения, а также указать его сильные стороны и ограничения (например, высокая нагрузка на систему, необходимость в настройке тестовой среды).

1.4. Итоговый отчёт

По результатам индивидуальной работы отчёт должен содержать:

1. Титульный лист, включающий в себя наименование работы, автора, дату выполнения, название учебной дисциплины.
2. Краткое описание каждого проекта, используемых анализаторов и применённых методов анализа.
3. Таблицы или графики, отображающие количество и тип обнаруженных ошибок до и после внесения намеренных дефектов.
4. Сравнительный анализ результатов до и после внесения ошибок.

5. Сравнение отчётов статического и динамического анализа, обсуждение ложных срабатываний, пропущенных ошибок и возможных причин таких результатов.

6. Обоснованные выводы о целесообразности применения статического и динамического анализа в различных условиях, рекомендации по выбору инструментов для разных типов проектов и интеграции анализа в процессы разработки.

7. Приложения (при необходимости): дополнительные материалы (коды, логи тестирования, скриншоты и т. д.).

Отчёт должен быть оформлен в соответствии с ГОСТ 19.401-78 и ГОСТ 34.602-2020. Требования включают стандарты на титульный лист, использование единообразного шрифта, оформление таблиц и диаграмм, наличие нумерации страниц.

Итоговая оценка работы будет зависеть от полноты отчёта, качества выполнения задания, соответствия оформления стандартам и презентации результатов на защите.