Федеральное государственное автономное образовательное учреждение
высшего образования «Длинное название образовательного учреждения
«АББРЕВИАТУРА»

На правах рукописи

Фамилия Имя Отчество

Длинное название диссертационной работы, состоящее из достаточно большого количества слов, совсем длинное длинное длинное длинное название, из которого простому обывателю знакомы, в лучшем случае, лишь отдельные слова

Специальность XX.XX.XX —
«Технология обработки, хранения и переработки злаковых, бобовых культур, крупяных продуктов, плодоовощной продукции и виноградарства»

Диссертация на соискание учёной степени
кандидата физико-математических наук

Научный руководитель:
уч. степень, уч. звание
Фамилия Имя Отчество

Город — 2021

Оглавление

Стр.

Стр.

Введение

Статический анализ появился как развитие практики инспекции кода человеком и вырос в метод автоматической проверки кода программ. Сегодня автоматические инструменты статического анализа приобрели популярность в индустрии разработки программного обеспечения, они значительно увеличивают продуктивность за счет автоматической проверки кода по широкому диапазону критериев. Многие проекты по разработке программного обеспечения разрабатываются в соответствии с рекомендациями по кодированию, таких как MISRA C[1], CERT[2], JPL[3] и др., стремясь к стилю программирования, который улучшает понимаемость кода и снижает риск появления ошибок. Для проектов с повышенными требованиями к безопасности соответствие подобным правилам написания кода настоятельно рекомендовано, а иногда и является необходимым, как например MISRA в индустрии автомобилестроения. Проверка соответствия с помощью инструментов статического анализа все чаще становится обязательной практикой.

Однако, для того чтобы предотвратить ошибку одного следования правилам кодирования не достаточно. Это признано нормой MISRA C через особое правило, которое рекомендует более глубокий анализ: ”Минимизация сбоев во время выполнения должна быть обеспечивается использованием хотя бы одного из следующих подходов: (a) инструментом статического анализа; (b) инструментами динамического анализа; (c) явное написание проверок для обработки во время выполнения программы.”(MISRA C:2004, rule 21.1). Действующие стандарты безопасности требуют демонстрации отсутствия ошибок времени выполнения и состояния гонки, например: DO-178B/C, ISO-26262, EN-50128, IEC-61508.

На данный момент существует множество различных инструментов статического анализа. Их сравнение и выбор наиболее подходящего инструмента – непростая задача. Первая проблема заключается в том, что термин статический анализ используется для широкого диапазона техник, которые концептуально сильно различаются. Их всех объединяет то, что они анализируют код не выполняя его. Их можно разделить на три большие группы: синтаксические чекеры, не чувствительные к путям семантические анализаторы, чувствительные к путям семантические анализаторы.

Синтаксические чекеры. Ограничиваются исследованием синтаксиса программы. Большинство алгоритмически проверяемых правил MISRA C могут

быть проверены на синтаксическом уровне. В MISRA C:2012, 116 из 143 правил классифицированы как разрешимые, что в данном контексте подразумевает возможность проверки на синтаксическом уровне. Примеры: Lint? (1978).

Не чувствительные к путям семантические анализаторы. Сообщают о семантических ошибках в программе, таких как ошибки времени выполнения (деления на ноль, арифметические переполнения или переполнения буфера) и состояния гонки. Могут выдавать false positives (сообщения об ошибках, где нет настоящих дефектов) и false negatives (настоящий дефект, на котором анализатор не выдал сообщение об ошибке). Примеры: Klocwork[4], CodeSonar[5], Polyspace Bug Finder[6] и Coverity[7].

Чувствительные к путям семантические анализаторы. В большинстве случаев основаны на абстрактной интерпретации, формальном методе анализа программ, который обеспечивает математически строгий способ доказательства отсутствия дефектов без false negatives: ни один дефект не пропущен (из рассматриваемого типа дефектов). Могут сообщать о семантических ошибках в программе, включая ошибки времени выполнения и состояния гонки, также могут использоваться для доказательства функциональных утверждений, например что выходные значения всегда будут в ожидаемом диапазоне. Наличие false positives характерно для всех инструментов (проблема доказана в теореме Райса[8]). Примеры: Astree[9], Frama-C[10], Polyspace Code Prover[11], AdaCore CodePeer[12].

Разницу между этими подходами можно проиллюстрировать на примере деления на 0. В выражении ”x/0”деление на ноль возможно обнаружить синтаксически, но не в выражении ”a/b”. Не чувствительный к путям анализатор может не сообщить о делении на ноль для ”a/b”хотя деление на ноль может иметь место в сценариях, не рассмотренных анализатором. Когда чувствительный к путям анализатор не выдает предупреждение о делении на ноль в ”a/b”это доказывает, что b никогда не будет 0.

Бенчмарки для инструментов статического анализа предоставляют основу для сравнения различных инструментов. Они должны (как минимум) точно определять, какие дефекты исследуются, взвешивать серьезность различных дефектов, брать в расчет глубину анализа, определять false positives и false negatives оценки.

Целью данной работы является разработать подход к проверке качества автоматических инструментов анализа программ.

Для достижения поставленной цели необходимо было решить следующие **задачи**:

1. Изучить современные инструменты анализа программ: типы, характеристики, техники сравнения.
2. Сформулировать требования, критерии и методы, которые позволят оценить сравнительные характеристики анализаторов кода.
3. Разработать фреймворк оценки качества анализаторов кода.
4. На основе полученного фреймворка сделать оценку его применимости для сравнения статических анализаторов кода программ.

Научная новизна:

1. На основе сравнительного анализа методов подтверждения качества статических анализаторов разработана методика сравнения результатов анализа нескольких статических анализаторов.
2. Предложен подход для автоматизации процесса сравнения статических анализаторов.
3. Проведена оценка предложенного подхода на основе сравнения свободных статических анализаторов.

Практическая значимость На основе разработанной методики разработан инструмент автоматического сравнения качества статических анализаторов.

Методология и методы исследования. В данном работе используются: метод возхождения от абстрактного к конкретному, метод моделирования, метод сравнения.

Основные положения, выносимые на защиту:

1. Методика сравнения статических анализаторов.
2. Архитектура среды сравнения статических анализаторов.

Достоверность полученных результатов обеспечивается … Результаты находятся в соответствии с результатами, полученными другими авторами.

Апробация работы. Основные результаты работы докладывались на: перечисление основных конференций, симпозиумов и т. п.

Личный вклад. Автор принимал участие в разработке среды сравнения статических анализаторов, наполнении тестового набора для тестирования статических анализаторов тестовыми сценариями, провёл сравнительный анализ нескольких свободных статических анализаторов.

Объем и структура работы. Диссертация состоит из введения, 3 глав, заключения и 1 приложения. Полный объём диссертации составляет 46 страниц, включая 0 рисунков и 0 таблиц. Список литературы содержит 48 наименований.

Глава 1. Обзор

Надежность – главная проблема современных процессов разработки программного обеспечения. Поскольку сложность системы постоянно растет, традиционные методы тестирования не справляются с задачей проверки. Разработка программного обеспечения, даже для встраиваемых систем, стала глобальной задачей, в которую вовлечены разработчики из различных филиалов. Географически разделенные инженеры создают тысячи или даже миллионы строк кода и, возможно, никогда не видели друг друга. Контроль качества процессов производства современного программного обеспечения стал значительно труднее. С другой стороны, неисправность программного обеспечения обходится дорого с точки зрения сбоя самого приложения, но также из-за вытекающих из этого последствий, таких как несчастные случаи со смертельным исходом, потеря денег, отключение жизненно важных систем, потеря репутации и выплаты.

Убедиться в тезисе, представленном выше, можно на примере следующих известных сбоев программного обеспечения:

Взрыв пусковой установки ”Ариан 5”(1996 г.). Первый полет ракеты-носителя ”Ариан-5”в июне 1996 года потерпел неудачу из-за сбоя в управляющем программном обеспечении. Необработанное исключение привело к самоуничтожению ракеты всего через 37 секунд после запуска. Неудачное преобразование данных из 64-битного числа с плавающей запятой в 16-битное целое число со знаком является одной из самых дорогих программных ошибок в аэрокосмической отрасли[13].

Утрата орбитального аппарата NASA Mars Climate Orbiter (1999 г.). ...

1.1 История автоматического анализа программ

В 1947 году появились термины «ошибка» (bug) и «отладка» (debugging). Грейс Мюррей, ученая из Гарвардского университета, работавшая с компьютером Mark II, обнаружила, что мотылек застрял в реле, из-за чего оно не вступало в контакт. Она подробно описала инцидент в рабочем журнале, приклеив мотыль-

ка лентой в качестве доказательства и назвав мотылька «ошибкой», вызывающей ошибку, а действие по устранению ошибки – «отладкой».

В то время тесты были сосредоточены на оборудовании, потому что оно было не так развито, как сегодня, и его надежность была важна для правильного функционирования программного обеспечения. Термин отладка был связан с применением исправлений для конкретной ошибки как одна из фаз в стадии разработки программного обеспечения. Проводимые тесты имели коррекционный характер и выполнялись для устранения ошибок, не дававших программе работать.

В 1957 году Чарльз Бейкер объясняет необходимость разработки тестов, чтобы гарантировать, что программное обеспечение соответствует заранее разработанным требованиям (тестирование), а также функциональным возможностям программы (отладка)[14]. Разработка тестов стала более важной по мере того, как разрабатывались более дорогие и сложные приложения, и стоимость устранения всех этих недостатков оказывала явный риск для прибыльности проекта. Особое внимание было уделено увеличению количества и качества тестов, и впервые качество продукта стало связано с фазой тестирования. Цель заключалась в том, чтобы продемонстрировать, что программа выполняет то, что от нее требовалось, с использованием ожидаемых и выдаваемых параметров.

В 1979 году Гленфорд Дж. Майерс радикально меняет процедуру обнаружения ошибок в программе: ”Тестирование программного обеспечения – это процесс запуска программы с целью поиска ошибок.”[15] Обеспокоенность Майерса заключалась в том, что, преследуя цель продемонстрировать, что программа безупречна, можно подсознательно выбрать тестовые данные, которые имеют низкую вероятность вызвать сбои программы, тогда как если цель состоит в том, чтобы продемонстрировать, что программа ошибочна, тестовые данные будут имеют большую вероятность их обнаружения, и мы будем более успешными в тестировании и, следовательно, в качестве программного обеспечения. С этого момента тесты будут пытаться продемонстрировать, что программа не работает должным образом, в отличие от того, как это делалось ранее, что приведет к новым методам тестирования и анализа.

В 1983 году была предложена методология, которая объединяет действия по анализу, пересмотру (revision) и тестированию в течение жизненного цикла программного обеспечения, чтобы получить оценку продукта в процессе разработки. Этап тестирования признан неотъемлемым этапом в разработке продукта,

приобретая особое значение в связи с появлением инструментов для разработки автоматизированных тестов, которые заметно повысили эффективность.

В 1988 году Уильям Хетцель опубликовал «Рост тестирования программного обеспечения»[16], в котором он переопределил концепцию тестирования как планирование, проектирование, создание, обслуживание и выполнение тестов и тестовых сред. Это в основном отразилось на появлении фазы тестирования на самом раннем этапе разработки продукта, этапе планирования. Если мы представим весь процесс разработки в виде конечной линии, где начало - это планирование, а конец - мониторинг проданного продукта, мы увидим, как фаза тестирования переместилась влево. Она появилась как этап пост- продакшнен, позже это был этап предпродакшн, а сейчас она находится на стадии завершения. Эта практика известна как Shift-Left.

Э. В. Дейкстра в лекции «О надёжности программ» утверждает, что тесты могут показать наличие ошибок в программе, но не могут доказать их отсутствие[17]. Таким образом использование одного тестирования в процессе верификации программы не является достаточным.

1.1.1 Уровни тестирования ПО

Процесс тестирования представляет из себя несколько уровней.

Ход процесса разработки ПО можно описать семью шагами:

1. Требования пользователя программы переводятся в набор письменных требований. Это цели продукта. 2. Требования преобразуются в конкретные цели путем оценки осуществимости и стоимости, разрешения противоречивых требований и установления приоритетов и компромиссов. 3. Цели переводятся в точную спецификацию продукта, в которой продукт рассматривается как черный ящик и учитываются только его интерфейсы и взаимодействие с конечным пользователем. Это описание называется внешней спецификацией. 4. Если продукт представляет собой систему, такую как операционная система, система управления полетом, система управления базами данных или кадровая система сотрудников, а не программа (компилятор, программа расчета заработной платы, текстовый процессор), следующим процессом является проектирование системы. На этом этапе система разделяется на отдельные программы, компоненты

или подсистемы и определяется их интерфейсы. 5. Структура программы или программ разрабатывается путем определения функции каждого модуля, иерархической структуры модулей и интерфейсов между модулями. 6. Разработана точная спецификация, определяющая интерфейс и функции каждого модуля. 7. Через один или несколько подшагов спецификация интерфейса модуля транслируется в алгоритм исходного кода каждого модуля.

Модульное тестирование (Unit testing) – это процесс тестирования отдельных подпрограмм, или процедур в программе. Прежде чем тестировать всю программу целиком следует сконцентрироваться на отдельных ее частях. Это объясняется рядом причин. Во-первых, модульное тестирование облегчает задачу отладки(процесс выявления и исправления обнаруженной ошибки), так как, когда ошибка найдена область ее распространения ограничена размерами модуля. Во-вторых, это позволяет распараллелить процесс тестирования проверяя сразу несколько модулей одновременно. Целью модульного тестирования является сравнение значения функции модуля некоторой функции или интерфейсу спецификации, определенной модулем. Следует подчеркнуть, что цель, как и для любого процесса тестирования, в том чтобы найти несоответствие спецификации.

Когда заканчивается модульное тестирование программы, в действительности процесс тестирования только начинается. Особенно это касается больших или сложных программ. Программная ошибка возникает, когда программа не выполняет то, что ее конечный пользователь обоснованно ожидает от нее. Согласно данному определения, даже проведя абсолютно идеальный модульный тест нельзя утверждать, что все ошибки найдены. Таким образом, для завершения тестирования необходимо какое-то дополнительное тестирование. В[18] это называется тестированием высшего порядка.

Разработка программного обеспечения - это в значительной степени процесс передачи информации о реализуемой программе и перевода этой информации из одной формы в другую. По этой причине подавляющее большинство ошибок программного обеспечения можно отнести к сбоям, ошибкам и помехам во время передачи и перевода информации.

Учитывая предпосылку, что семь этапов цикла разработки включают в себя общение, понимание и перевод информации, а также предпосылку, что большинство ошибок программного обеспечения происходит из-за сбоев в обработке информации, существует три дополнительных подхода для предотвращения и/или обнаружения этих ошибок. Во-первых, мы можем внести больше точности в

процесс разработки, чтобы предотвратить многие ошибки. Во-вторых, мы можем ввести в конце каждого процесса отдельный этап проверки, чтобы определить как можно больше ошибок, прежде чем переходить к следующему процессу. Третий подход – ориентировать отдельные процессы тестирования на отдельные процессы разработки. То есть сосредоточить каждый процесс тестирования на конкретном этапе перевода, таким образом фокусируя его на определенном классе ошибок. Другими словами, вы должны иметь возможность установить взаимно однозначное соответствие между процессами разработки и тестирования.

Функциональное тестирование (Functional testing) – это попытка найти расхождения между программой и внешней спецификацией. Внешняя спецификация – это точное описание поведения программы с точки зрения конечного пользователя. За исключением случаев использования в небольших программах, функциональное тестирование обычно представляет собой черный ящик. Чтобы выполнить функциональный тест, спецификация анализируется для получения набора тестовых примеров.

Системное тестирование (System testing) – это финальное тестирование программы с проверкой всех функциональных и нефункциональных требований. Системное тестирование - это не процесс тестирования функций всей системы или программы, потому что это будет дублировать процесс функционального тестирования. Системное тестирование имеет конкретную цель: сравнить систему или программу с ее первоначальными целями.

Приемочное тестирование (Acceptance testing) – это процесс сравнения программы с исходными требованиями и текущими потребностями конечных пользователей. Это необычный тип тестирования, поскольку он обычно выполняется заказчиком программы или конечным пользователем и обычно не считается обязанностью организации-разработчика.

На ряду с тестированием одним из подходов к написанию безопасного и переносимого ПО является использование в разработке стандартов кодирования.

1.1.2 Правила кодирования

Спустя полвека с момента своего создания язык программирования C по-прежнему остается одним из наиболее часто используемых языков програм-

мирования[19] и наиболее часто используемый для разработки встраиваемых систем. Причины такого успеха уходят корнями в требования отрасли. К подобным причинам относятся[20]:

- Компиляторы C существуют практически для любого процессора, от крошечных DSP, используемых в слуховых аппаратах, до суперкомпьютеров.
- Скомпилированный код на C может быть очень эффективным и без скрытых затрат, т.е. программисты могут приблизительно предсказать время выполнения еще до тестирования и до использования инструментов для наихудшего приближения времени выполнения (Это по-прежнему верно для реализаций, работающих на простых процессорах, с ограниченной степенью кэширования и внутреннего параллелизма. Предсказание максимального времени работы без инструментов становится совершенно невозможным для современных многоядерных систем, таких как Kalray MPPA, Freescale P4080 или эквиваленты ARM Cortex-A57).
- C позволяет писать компактный код.
- C определяется международными стандартами: он был впервые стандартизирован в 1989 году Американским национальным институтом стандартов (эта версия языка известна как ANSI C), а затем Международной организацией по стандартизации (ISO).
- C, возможно с расширениями, обеспечивает легкий доступ к аппаратному оборудованию, что является необходимым для разработки встроенного программного обеспечения.
- C имеет долгую историю использования во всех типах систем, включая системы безопасности, защиты, критически важные для бизнеса.
- C широко поддерживается всевозможными инструментами.

На ряду с достоинствами C имеет и недостатки: написание безопасных и защищенных приложений на C требует особой осторожности. В противном случае код может получиться запутанным и неясным. Замечательные примеры подобных программ представлены на международном конкурсе запутанного C кода [21]. Решение, обязательное или настоятельно рекомендуемое всеми применимыми промышленными стандартами – language subsetting: критически важные приложения не программируются на неограниченном языке C, а программируются в подмножестве, где вероятность совершения потенциально опасных ошибок

снижена. Это требуется или настоятельно рекомендуется всеми стандартами безопасности, таким как, например, IEC 61508[22].

Конечно, кодирования на более безопасном подмножестве C недостаточно для гарантии корректности. Однако:

- Ограничение языкового подмножества, в котором не полностью определенное поведение и проблемные функции запрещены или строго регулируются, «может значительно повысить эффективность и точность статического анализа»[23];
- Правильно спроектированные языковые подмножества имеют сильное влияние на читаемость кода: ревью кода в сочетании со статическим анализом и соблюдением правил кодирования являются основой наиболее эффективных стратегий устранения дефектов.

Согласно[24] в опросе среди специалистов встроенных систем ПО стандарт кодирования MISRA C[1] - наиболее широко используемый и стандарт BARR-C[25] – второй наиболее используемый. Вместе они получили 40 процентов голосов респондентов.

Проект MISRA (Motor Industry Software Reliability Association) был основан для создания руководства по разработке ПО для микроконтроллеров в наземных средствах по заказу правительства Британии. Работа над проектом началась в 1990 году. Первое руководство вышло в 1994 году, не было привязано к какому-либо языку. Консорциум MISRA приступил к работе на стандартом для языка C: в это же время Форд и Ленд Ровер независимо разрабатывали проприетарные руководства для программного обеспечения на языке C для транспортных средств, и было признано, что совместная деятельность будет более выгодной для промышленности. Первый связанный с языком C стандарт MISRA C[1] вышел в 1998 году и стал общепринятым.

В 2004 году MISRA опубликовало улучшенную версию стандарта[26], расширив целевую аудиторию, чтобы включить все отрасли, которые разрабатывают программное обеспечение на C для использования в высоконадежных/критических системах. Благодаря успеху MISRA C и тому факту, что C++ также используется в критических контекстах, в 2008 г. MISRA опубликовала аналогичный набор рекомендаций MISRA C++ [27].

И MISRA C:1998, и MISRA C:2004 нацелены на версию стандарта C 1990 г.[28]. Последняя версия MISRA C:2012, опубликованная в 2013 г.[29] поддерживает оба стандарта: C90 и C99[30]. По сравнению с предыдущими версиями,

MISRA C:2012 охватывает больше языковых проблем и предоставляет более точную спецификацию с улучшенными обоснованием и примерами.

MISRA C повлиял на все общедоступные стандарты кодирования для C и C++, которые были разработаны после MISRA C:1998. MISRA C:1998 повлиял на JSF Air Vehicle C++ Coding Standards[31], который в свою очередь повлиял на MISRA C++:2008.

В MISRA C правила делятся на три основных категории: Mandatory, Required и Advisory. Mandatory - наиболее строгая категория, требующая постоянного выполнения. Required - менее строгая: возможны отклонения при условии документирования и обоснования. Advisory - правила, которым следовать не обязательно.

В MISRA-C:1998 перечислено 127 правил (93 обязательных и 34 рекомендательных).

В MISRA-C:2004 141 правило (121 обязательное и 20 рекомендательных). Правила разделены на 21 категорию.

В MISRA-C:2012 143 правила (каждое из которых может быть проверено статическим анализатором кода) и 16 директив (правил, соответствие которым открыто для интерпретаций или связано с процессами и процедурами). Правила делятся на обязательные, требуемые и рекомендательные; могут распространяться на отдельные единицы трансляции или на всю систему.

На стандарт BARR:2018 оказали влияние MISRA C:2012, BARR C:2013 и NETRINO EMBEDDED C. Как следствие BARR:2018 является надмножеством MISRA C:2012 и может использоваться, как плавный первый шаг для проекта без статического анализа и правил кодирования к проекту, соответствующему MISRA C:2012.

Другими примерами стандарта кодирования для обеспечения безопасности критических систем являются:

- ”JPL Institutional Coding Standard for the C Programming Language”[3] от НАСА. Данный стандарт испытал влияние MISRA C:2004 и стандарта ”Сила десяти”[32]. Ни один из данных источников не рассматривает программные риски, связанные с использованием многопоточного программного обеспечения. Стандарт JPL призван заполнить этот пробел.
- ”High Integrity C++”[33] от Programming Research Ltd.
- ”Joint Strike Figther Air Vehicle C++ Coding Standards”[31] от Lockheed.

– ”Embedded System development Coding Reference guide for C/C++”[34; 35] от Information-technology Promotion Agency, Japan.
– ”Guidelines for the use of the C++14 language in critical and safety-related systems”[36] от AUTOSAR.
– ”The SEI CERT C/C++ Coding Standart”от Carnegie Mellon University[2].

1.1.3 Бенчмарки для проверки статических анализаторов

переделать –>

Бенчмаркинг обеспечивает объективный и повторяемый способ измерения свойств инструмента обнаружения ошибок. Бенчмарки для инструментов обнаружения ошибок должны отвечать на несколько вопросов о результатах работы инструмента[37]:

– Сколько ошибок обнаружены правильно?
– Сколько ложных сообщений об ошибках сделано?
– Сколько ошибок пропущено/не сообщается?
– Насколько хорошо масштабируется инструмент?

Выделяются два основных типа бенчмарков. Маленькие бенчмарки размером от 10 до тысячи строк кода состоят из синтетических тестов, либо из автономных программ, извлеченные из существующего ошибочного кода, которые содержат конкретную ошибку. В работе[38] извлечено 15 багов из реальных приложений. В работе[39] синтезировано 291 тесткейса ошибки переполнения буфера. Один из наиболее известных для данных целей тестовый набор - Juliet Test Suite[40]. Juliet Test Suite был разработан Центр гарантированного программного обеспечения (Center for Assured Software) Агентства национальной безопасности США (US American National Security Agency). Его тестовые сценарии были созданы для тестирования сканеров или другого программного обеспечения. Набор тестов состоит из двух частей. Одна часть посвящена ошибкам безопасности для языков программирования C и C++. Другой касается ошибок безопасности для языка Java. В 2019 вышел набор для C# и есть экспериментальные (ещё не стандартизированные) наборы для других языков. Примеры кода с уязвимостями безопасности даны в простой форме, а также встроены в вариации различных потоков управления и паттернов потоков данных. Пакет содержит около 64000

тестовых случаев на C/C++ и около 28000 тестовых случаев на Java. Набор тестов охватывает 25 основных ошибок безопасности, определенных SANS/MITRE (MITRE 2011)[41]. Можно выделить два типа исходного кода: искусственно созданные примеры с целью тестирования определенного свойства анализатора и код, полученный из репозиториев программ, в которых были найдены программные ошибки в процессе тестирования или эксплуатации. Естественный код используется в реальном программном обеспечении, таком как, например, веб-сервер Apache или Microsoft Word. Искусственный код создается для определенной цели, например, для тестирования сканеров безопасности. Juliet Test Suite содержит только искусственно созданные примеры кода, поскольку такой код упрощает оценку и сравнение инструментов статического анализа.

Примерами больших бенчмарков в виде полных программных дистрибутивов являются BugBench[42] и Faultbench[43].

<– переделать

1.1.4 Первое поколение статических анализаторов. Lint

Первый инструмент статического анализа Lint[44] появился в конце 1970-х. Впервые разработчики получили возможность автоматизировать обнаружение дефектов программного обеспечения на самых ранних этапах жизненного цикла приложения, когда их легче всего исправить. Кроме того, это давало разработчикам уверенность в качестве своего кода перед релизом. Технология, лежащая в основе Lint, была революционной, она использовала компиляторы для проверки дефектов.

Однако Lint не разрабатывался с целью выявления дефектов, вызывающих проблемы во время выполнения программы. Скорее, его цель заключалась в том, чтобы выделить подозрительные или непереносимые конструкции в коде и помочь разработчикам соблюдать общий формат при кодировании. Под ”подозрительным кодом”имеется в виду код, который, будучи технически правильным с точки зрения языка исходного кода (например C, C++), может быть структурирован так, чтобы он выполнялся способами, которые разработчик не предполагал. Lint являлся дополнением к компилятору. В то время как компилятор концентрировался на быстром и точном превращении программы в последовательность

бит, Lint концентрировался на ошибках в переносимости, стиле и эффективности. Из-за ограниченных возможностей анализа Lint, уровень шума был чрезвычайно высоким, часто превышая соотношение между шумом и реальными дефектами в соотношении 10 к 1.

Следовательно, обнаружение настоящих дефектов требовало от разработчиков проведения трудоемкой ручной проверки результатов Lint, что усложняло именно ту проблему, которую должен был устранить статический анализ. По этой причине Lint так и не получил широкого распространения в качестве инструмента обнаружения дефектов, хотя имел ограниченный успех в нескольких организациях. Фактически, как свидетельство качества технологии, лежащей в основе Lint, множество различных версий продукта по-прежнему доступны сегодня.

1.1.5 Второе поколение статических анализаторов

Почти два десятилетия статический анализ оставался скорее областью научных исследований, чем коммерчески жизнеспособным производственным инструментом для выявления дефектов. В начале 2000 года появилось второе поколение инструментов. Используя новые технологии, оно расширяло возможности инструментов первого поколения от простого выявления нежелательных паттернов до покрытия путей выполнения. Эти инструменты могли анализировать целые базы кода, а не только один файл.

Сместив фокус с «подозрительных конструкций» на «дефекты времени выполнения», разработчики статического анализа осознали необходимость в большем понимания внутреннего устройства программ. Это означало объединение сложного анализа путей с межпроцедурным анализом, чтобы понять, что происходит, когда поток управления переходит от одной функции к другой в рамках данной программы.

Несмотря на принятие и использование организациями, статический анализ 2-го поколения все еще не мог найти золотую середину между точностью и масштабируемостью. Некоторые решения были точными для небольшого набора типов дефектов, но не могли масштабироваться для анализа миллионов строк кода. Другие могли работать за короткое время, но имели показатели точности, аналогичные Lint, представляя похожие проблемы с предупреждениями об ошиб-

ках и шумом. После внедрения в процесс разработки эти инструменты могут сообщать о дефектах в приемлемом соотношении, но только с ограниченными параметрами анализа.

Инструментам второго поколения требовались однородные среды сборки и разработки. Из-за этого внедрить их в какой то проект было сложной задачей, требовавшей больших усилий.

Борьба между точностью и масштабируемостью вылилась в проблему с ложными срабатываниями. Подобно шуму, от которого страдали инструменты первого поколения, ложные срабатывания тормозили распространению инструментов второго поколения.

1.1.6 Третье поколение статических анализаторов

Третье поколение инструментов статической проверки программ превосходит своих предшественников по всем параметрам и является неотъемлемой частью процессов и сред разработки. Одной из причин подобного успеха является использование решателей для задач выполнимости булевых формул (SAT) совместно с традиционными техниками анализа.[]

Новаторское использование SAT позволяет статическому анализу исходного кода находить правильные дефекты в коде без большого количества ложных срабатываний.

1.2 Уровни статического анализа кода

1.2.1 Бинарный уровень

Предметом статического анализа могут выступать исходный код, промежуточное представление, ассемблерный код или бинарный код. То есть на любом этапе до выполнения программы. Часто статический анализ применяется на исходном коде или промежуточном представлении, таком как Static Single

Assignment (SSA). Код высокого уровня содержит множество структурированной информации, что упрощает анализ. Тем не менее статический анализ бинарного кода, набирает популярность благодаря нескольким тенденциям в разработке программного обеспечения. Первый тренд заключается в расширении функциональности программного обеспечения за счет интеграции надежного кода с ненадежным внешним кодом. Вторая тенденция – разработка программного обеспечения на основе компонентов для повышения производительности и контроля затрат. Несколько модулей от разных производителей объединены в единую систему. Изоляция неисправностей модулей от разных производителей жизненно важна для устойчивости всей системы. Как правило, статический анализ на двоичном уровне необходим по следующим причинам[45]:

1. Исходный код недоступен. Для вредоносного кода, такого как вирусы, черви, бот-сети, троянские кони и ненадежные расширения, такие как плагины браузера, надстройки баз данных и другой ненадежный код, исходный код либо недоступен, либо ненадежен.
2. Иногда оптимизирующие компиляторы оптимизируют код настолько агрессивно, что семантика двоичного кода может отличаться от семантики исходного кода. Подобная ситуация может потребовать выполнить проверку перевода сгенерированного кода.
3. Перезапись безопасности неуместна на уровне источника или IR. Некоторые стратегии безопасности анализируют и вставляют проверки безопасности в двоичный код для защиты рассматриваемого двоичного кода. Эти подходы должны идентифицировать определенные машинные инструкции в двоичном коде и вставлять динамические проверки безопасности прямо перед ними. На исходном уровне или уровне IR машинные инструкции даже не генерируются.
4. Некоторые оптимизации могут быть выполнены на двоичном уровне для повышения производительности во время выполнения программы. Хотя большинство программных оптимизаций выполняется на промежуточных представлениях во время компиляции, некоторые системы оптимизируют двоичный код напрямую без информации об исходном коде или с ее ограничением.
5. Двоичный код можно повторно использовать напрямую, без исходного кода. Иногда исходный код теряется, недоступен или больше не под-

держивается его первоначальными авторами, и было бы полезно иметь возможность повторно использовать двоичный код напрямую.

Причины для проведения статического анализа двоичного кода вместо исходного кода или промежуточных представлений, которые перечислены выше, далеки от завершения.

К стратениям проведения статического анализа на бинарном коде относятся[45]:

1. Дизассемблеры. Дизассемблер используется для декодирования информации, хранящейся в двоичных файлах, и перевода машинных инструкций на язык ассемблера или эквивалентное промежуточное представление. В принципе, методы дизассемблирования можно разделить на две категории: методы динамической разборки и методы статической разборки в зависимости от того, выполняется ли двоичный код или нет.
2. Построение графа потока управления. После дизассемблирования двоичного кода следующим шагом является построение графа потока управления или конкретизация скелета графа потока управления, если он уже был сгенерирован во время разборки. Как правило, в двоичном коде нет явной информации о потоке управления. Все нужно извлечь из самого двоичного кода. Существует несколько алгоритмов построения графа потока управления. Для ребер, вызванных инструкциями прямого перехода и вызова, дизассемблер может легко идентифицировать их цели(targets) и добавлять ребра для них в граф потока управления. Например, IDA Pro[46] создает граф потока управления с ребрами, индуцированными только инструкциями прямого перехода и вызова.
3. К прочим стратегиям стоит отнести разбор регистров флагов и анализ алиасов.

1.2.2 Уровень промежуточного представления

Компиляция программы – сложный процесс. Компилятор – это программа, которая переводит высокоуровневую программу на исходном языке в форму, готовую к выполнению на компьютере. На раннем этапе развития компиляторов

разработчики ввели IR – intermediate representation (промежуточные представления, также обычно называемые промежуточными языками) для управления сложностью процесса компиляции. Использование IR в качестве внутреннего представления программы позволяет разбить компилятор на несколько этапов и компонентов, тем самым извлекая выгоду из модульности. Одним из наиболее популярных промежуточных представлений является LLVM IR.

Проект LLVM – это набор модульных технологий, компиляторов и множество инструментов. LLVM начинался как исследовательский проект в Университете Иллинойса с целью предоставить современную стратегию компиляции на основе SSA, способную поддерживать как статическую, так и динамическую компиляцию произвольных языков программирования. С тех пор LLVM превратился в зонтичный проект, состоящий из ряда подпроектов, многие из которых используются в производстве в большом количестве коммерческих проектов и проектов с открытым исходным кодом, а также широко используются в академических исследованиях[47].

В [48] выделяют следующие преимущества статического анализа промежуточного представления LLVM IR языков высокого уровня (C/C++):

1. Анализируемая программа намного ближе к программе, которая фактически выполняется на компьютере, поскольку неоднозначности семантики уже устранены.
2. Появляется возможность находить ошибки, внесенные компилятором.
3. Использование компилятора IR делает возможным выполнение анализа для программ, написанных на различных языках программирования.

Программы в LLVM-IR представляют собой SSA (static single assignment), то есть, каждой (скалярной) переменной значение присваивается ровно один раз. Таким образом, присвоение значения скалярным переменным можно рассматривать как логические эквивалентности. SSA располагает ограниченным набором используемых инструкций и низкоуровневым характером, что позволяет произвести преобразование программы LLVM-IR в логическое представление значительно проще, по сравнению с подобными преобразованиями для исходного кода языка программирования высокого уровня[48].

1.2.3 Уровень ассемблерного кода

TODO

1.2.4 Уровень исходного кода

TODO

1.3 Динамические анализаторы

TODO

1.4 Подходы к проверке качества инструментов анализа программ

Инструмент анализа программ, как программый продукт, можно оценить по производительности и маштабируемости. Строгих общепринятых значений производительности и масштабируемости не существует, поскольку эти параметры зависят от глубины, сложности и цели анализа и сильно различаются между инструментами.

Выделяются два подхода к оценке качества анализа, проводимого инструментом:

– Запуск на реальном проекте с последующей интерпритацией результатов;
– Запуск на тестовом наборе (бенчмарке).

Первый подход позволяет измерить:

– количество шума, выдаваемого инструментом;
– количество
– производительность;

– при наличии дополнительных ресурсов, задействованных в процессе анализа, оценить масштабируемость.

Глава 2. Требования

2.1 Выбор характеристик для оценки качества инструментов статического анализа

Результат запуска статического анализатора на тестовом наборе или проекте это множество верных и ложных предупреждений об ошибках (соответственно true positives – TP, false positives – FP). Ложное предупреждение об ошибке свидетельствует о наличие неточности в алгоритме статического анализатора. Большое количество ложных предупреждений – это шум в результатах работы статического анализатора и признак низкого качества проводимого анализа. Большое количество верных предупреждений при малом количестве ложных предупреждений свидетельствует о высоком качестве проводимого анализа и о его полноте. Под полнотой понимается способность статического анализатора обнаруживать как можно больше дефектов, являющихся истинными ошибками в программе. Отношение верных предупреждений об ошибках к общему количеству предупреждений это метрика – точность (precision) (2.1).

$$precision = TP/(TP + FP) \tag{2.1}$$

Данная метрика используется в статистике для оценки результатов исследования на основе полученных true positives и false positives. Данная метрика широко распространена в оценке качества статических анализаторов. Общее количество предупреждений формируется из суммы верных и ложных предупреждений. Ситуации, в которых инструмент не выдал предупреждения, однако ошибка имеет место быть, являются false negatives. Сумма false negatives и true positives это общее количество ошибок, которое содержится в тестовом наборе или коде проекта. Чаще эта величина вычисляется для статических анализаторов, сканирующих тестовые наборы, так как определение точного количества ошибок в программном проекте –- сложная задача. Отношение количества верных предупреждений, true positives, к общему количеству ошибок в тестовом наборе или проекте это мера оценки качества результата инструмента статического анализа, называемая отклик (recall) (2.2).

$$recall = TP/(TP + FN) \tag{2.2}$$

Данная мера так же как и точность используется в статистике для оценки качества проводимого исследования. Идеальный инструмент статического анализа не выдает ложных предупреждений и сообщает о всех существующих ошибках. Для идеального инструмента значения точности и отклика равны единице. Теоретически полностью противоположный идеальному инструменту инструмент выдает только ложные предупреждения, шум. Для такого инструмента значения точности и отклика равны нолю. Таким образом значения точности и отклика находятся в диапазоне от ноля до единицы. На основе значений точности и отклика вычисляется величина F-measure. F-measure характеризует точность проводимого исследования. F-measure вычисляется, как гармоническое среднее точности и отклика (2.3).

$$F - measure = 2 * precision * recall/(precision + recall) \tag{2.3}$$

Наибольшее возможное значение для F-measure равно одному, соответсвующее идеальным значениям точности и отклика. Наименьшее возможное значение для F-measure равно нолю, в данном случае одно из величин, точность или отклик равно нолю.

Чтобы измерить величины точность, отклик, F-measure для участвующего в исследовании статического анализатора предлагается следующий порядок действий. Во-первых, сформировать тестовый набор. Данный тестовый набор должен соответствовать инспекциям (чекерам) статического анализатора, ошибкам которые он обнаруживает. Во-вторых, запустить статический анализатор на данном тестовом наборе. Данный шаг осуществляется средствами Acceptance Testing Framework, или сокращенно ATF. В-третьих, собрать статистику работы статического анализатора на тестовом наборе. Данный шаг осуществляется средствами ATF. В-четвертых, на основе полученной статистики, которая есть количество верных и ложных предупреждений, вычислить значения точности и отклика.

Результаты измерений будут представлены в двух интерпретациях. Первая интерпретация – значения величин точность и отклик для тестового набора, на котором проводился анализ для данного статического анализатора, то есть проверяется качество результата инструмента в рамках заявленных возможностей обнаружения определённого типа ошибок. Вторая интерпретация –- значения величин точность и отклик на тестовых примерах, представляющих полное количество возможных обнаруживаемых дефектов в рамках тестового набора. Вторая

интерпретация необходима для объективного сравнения качества анализа, проводимого статическими анализаторами.

2.2 Требования к Acceptance Testing Framework

Инструменты статического анализа исходного кода должны проверять состояние исходного кода программ с точки зрения очень разных правил, которые могут применяться в качестве промышленного или общекорпоративного стандарта кодирования. Несмотря на то, что современные статические анализаторы исходного кода уделяют особое внимание безопасности кода, отсутствию логических ошибок и производительности, некоторые правила кодирования, применяемые в компаниях или отрасли, могут содержать такие требования к коду, как стиль отступов, соглашения об именах и т. д. Например, если рассмотреть программы, написанные на языке программирования Python, то исходный код может содержать комментарии определенного вида, такие как Shebang, кодировка файла, информация о версии или лицензии. Вот почему, пытаясь удовлетворить потребности тестирования промышленных статических анализаторов исходного кода, такой фреймворк не может полагаться на специальные комментарии и форматирование кода, как, например, используемые в большинстве известных баз данных тестовых примеров Juliet Национального института стандартизации и технологий США.

База данных тестовых примеров должна состоять из двух частей. Первая часть – тестовые примеры. Вторая часть – описание тестовых примеров. Элемент второй части должен предоставлять всю необходимую информацию по тестовому примеру в виде файла или группы файлов, организованных в структуру каталогов. Файлы описатели, они же файлы аннотации, не должны зависеть от языка программирования исходного кода инструмента статического анализа и языка программирования анализируемых программ. Файл аннотация – это файл в JSON формате, который описывает тестовый пример как для содержащего, так и для не содержащего ошибку.

Фреймворк должен сравнивать статические анализаторы по величине полноты проведенного на тестовом наборе анализа. Для этого фреймворк должен поддерживать запуск нескольких статических анализаторов в одной связке.

Фреймворк не должен зависеть от операционной системы.

Существуют инструменты статического анализа программ для множества разных языков, иногда сразу нескольких. Acceptance Testing Framework не должен зависеть от целевого языка анализируемых программ. Он должен подходить для тестирования анализаторов с такими целевыми языками программирования как C, C++, Java, C#, Python и других языков.

На вход фреймворк принимает базу данных с тестовыми примерами и файлами аннотациями – тестовый набор. Фреймворк не должен менять структуру или отдельные файлы тестового набора. Совместимость достигается за счет соблюдения в тестовом наборе правил, которые накладывает фреймворк.

Тестовый набор может содержать тестовые примеры трех видов. Первые – тестовые примеры с ошибкой, дефектом, о котором должен предупредить инструмент статического анализа. Вторые – не содержащие ошибку тестовые примеры. Третьи – тестовые примеры, проверяющие поддержку специфических конструкций языка программирования. Первая и вторая группы это соответственно истинно положительные и ложноположительные предупреждения (третья группа также ложноположительные предупреждения). Для определения достоверного качества инструмента статического анализа необходима возможность проверки всех групп тестовых примеров, выделение отдельной статистики для каждой группы.

Средствами статического анализа возможно проверить исходную программу на наличие в ней уязвимостей безопасности, ошибок времени выполнения, несоответствия стилю форматирования и комментариев. Фреймворк должен поддерживать неограниченное количество средств проверки правил кодирования, включая, помимо прочего, стили форматирования и комментариев.

Результат запуска инструмента статического анализа на исходном коде возможен в двух вариантах. Первый вариант это отсутствие предупреждения, выдаваемого статическим анализатором. Второй вариант, статический анализатор, просканировав исходный код, выдает предупреждение. Вариант, в котором выдается предупреждение, может быть представлен текстом, напечатанным в стандартном потоке вывода или ошибки, или файлом определенного формата. Если инструмент статического анализа встроен в среду разработки, то результат его работы есть подсветка кода в местах, соответствующих выданным предупреждениям. Необходимо, чтобы независимо от выдаваемого формата фреймворк

поддерживал возможность запуска и оценки результата работы инструмента статического анализа.

Результат запуска инструмента статического анализа на тестовом наборе фреймворк должен агрегировать во внутреннее представление. На основе внутреннего представления фреймворк должен формировать отчет для вывода результата и представления данного результата пользователю в различных форматах: машиночитаемый (JSON, XML и другие), вывод, отформатированный для отображения результата на экране, формат HTML. А также возможность расширения списка форматов отчета по запросу.

2.3 Требования к участвующим в исследовании инструментам статического анализа

Глава 3. Дизайн. Реализация

3.1 Дизайн

Независимость от целевой среды. Чтобы удовлетворить требование независимости фреймворка от целевой среды, такой как оборудование и операционная система, было принято решение реализовать фреймворк на языке Python, поскольку он имеет интерпретаторы исходного кода Python для большинства промышленных операционных систем и для большинства популярных аппаратных платформ.

Независимость анализируемого языка программирования. Фреймворк никак не полагается на содержимое фрагментов кода.

Возможность проверять фрагменты кода без изменения исходного кода, даже в комментариях. Возможность проверки как ошибочных, так и чистых фрагментов кода без модификации. Мы используем файлы-аннотации тестовых примеров в формате JSON. Тестовый пример для Acceptance Testing Framework – это кортеж из файла-аннотации и фрагмента исходного кода. Файл аннотации JSON содержит следующую информацию:

- Тип фрагмента: содержит ли он дефект (True Positive) или этого не ожидается в этом фрагменте кода (True Negative);
- Вид дефекта, о котором ожидаем или не ожидаем получить сообщение от статического анализатора;
- Описание тестового примера;
- Флаг пропуска для пометки тестовых случаев, которые не поддерживаются, но которые планируется поддерживать в будущем;
- Расположение дефекта: имя файла, номер строки и смещение, в котором ожидается дефект;
- Дополнительная служебная информация. Например, если тестовый пример разработан для конкретной версии языка, чтобы настроить анализатор соответствующим образом, или дополнительное поле, описывающее цель тестового примера, для QA-инженера или разработчика.

Такое решение позволяет хранить всю информацию независимо от тестовых примеров, необходимых Acceptance Testing Framework для соответствующей настройки инструментов статического анализа.

Также фреймворк не полагается на количество тестовых примеров в тестовом наборе. Достаточно при запуске фреймворка указать местоположение каталога файловой системы с набором тестов, отформатированным для использования Acceptance Testing Framework и вся работа, связанная с запуском инструментов статического анализа в наборе тестов будет обрабатываться самим фреймворком.

Возможность сравнивать различные инструменты анализа. Acceptance testing framework удовлетворяет этому требованию, вводя инструмент абстрактного интерфейса для запуска внешнего инструмента статического анализа в виде исполняемой программы и получения результатов анализа во внутреннем представлении ATF. Для поддержки нового инструмента анализа необходимо реализовать интерфейс Tool, чтобы преобразовать настройки тестового примера из аннотации тестового примера в ожидаемые аргументы инструмента анализа и запустить этот инструмент как внешний процесс. Разработан ряд реализаций интерфейса для инструментов, таких как PyLint, JetBrains PyCharm и восьми других инструментов, отличающихся между собой способом анализа программ. Например, PyLint допускает анализ на единственном входном файле и может быть запущен на каждом тестовом примере отдельно. PyCharm ожидает на вход директорию и рассматривает ее, как проект для проводимого анализа.

С другой стороны, представление результатов анализа разными инструментами может существенно различаться. Реализация интерфейса Tool также отвечает за интерпретацию результатов анализа определённого инструмента, для которого сделана реализация внешнего анализа и преобразует их во внутреннее представление Acceptance Testing Framework. По сути внутреннее представление является картой соответствия каждого тестового примера к одному из трех возможных значений, Passed, обозначающему, что инструмент прощел тестирование на тестовом примере, Failed – провалил его, Skipped – статический анализатор не запускался на данном тестовом примере.

Таким образом, вся логика работы с конкретным инструментом анализа инкапсулирована внутри реализации интерфейса Tool для данного инструмента.

3.2 Реализация

Acceptance Testing Framework архитектурно состоит из 4 компонентов: Driver, TestSuite, Tool, Reporter.

Driver. Входная точка для фреймворка. Позволяет настраивать набор тестов, репортер и инструменты в соответствии с параметрами, передаваемыми во фреймворк при запуске. Чтобы настроить тестовый набор следует передать в качестве параметров путь до директории с тестовым набором. В случае если данный параметр пропущен или указанный в параметре путь не существует, то в качестве значения принимается значение по умолчанию, путь до директории с тестовым набором, прописанный внутри фреймворка. Далее директория, к которой ведет путь, проверяется на наличие в ней тестовых примеров. При отсутствии тестовых примеров в директории фреймворк выдает предупреждение об ошибке. Для того, чтобы из множества всех тестовых примеров сформировать подмножество тестовых примеров, соответствующих определенному типу или определенным типам дефектов, следует передать соответствующий тип или типы в параметре kind. Возможна ситуация, когда часть тестовых примеров не поддерживается инструментом статического анализа или не может быть обнаружена в текущей версии инструмента. Для неподдерживаемых тестовых примеров следует добавить в файл, описывающий тестовый пример, поле skip и установить его значение равным true. При запуске статического анализатора фреймворком может потребоваться передать дополнительные параметры, такие как версия языка программирования. На основе переданной версии языка фреймворк сформирует из множества тестовых примеров подмножество, соответствующее указанной версии. Так получив в параметре python_version вторую или третью версию языка Python фреймворк сформирует подходящее подмножество тестовых примеров.

Параметры kind, skip, python_version участвуют в формировании предиката. Предикат – интерфейс, который для передаваемого в него набора параметров формирует множество проверок, проверок тестовых примеров на соответствие значениям параметров kind, skip, python_version. Тестовый пример с полем skip равным true, с типом дефекта, отличным от параметра kind, с версией языка Python, отличающейся от python_version, не проходит проверку и не будет проверяться.

TestSuite. Представляет собой интерфейс, описывающий набор интерфейсов TestCases, созданных с использованием предоставленного пути к каталогу набора тестов, где каждый тестовый пример имеет аннотацию в формате JSON, описанного в разделе 3.1, и файл с фрагментом кода. Каждый загружаемый тестовый пример должен сопровождаться файлом аннотацией в формате JSON, файл аннотация, в свою очередь, должен следовать структуре, определенной в компоненте TestCase, в противном случае фреймворк выдаст предупреждение об ошибке и завершит работу. Тестовые примеры, не удовлетворяющие требованиям интерфейса предиката, выделяются в отдельное множество тестовых примеров, для которых анализ не проводится, но в зависимости от вида отчёта могут включаться в статистику.

TestCase. Это представление тестового примера внутри фреймворка. Данный компонент определяет структуру, которой следует тестовый пример. Поля файла аннотации соответствуют атрибутам компонента TestCase. Для инициализации компонента TestCase следует передать путь к файлу аннотации тестового примера. Компонент TestCase насчитывает 18 атрибутов, относящихся к тестовому примеру. Обязательные к указанию атрибуты, без которых невозможна инициализация компонента TestCase, это: path, description, kind, positive. Path – путь к файлу аннотации. Description – текст, описывающий тип ошибки, представленной в тестовом примере в виде дефектного или похожего на дефектный фрагмента кода. Kind – тип ошибки, соответствует правилу из стандарта. Под стандартом подразумевается стандарт, регламентирующий корректное написание программ на целевом языке программирования, и на основе которого разработан тестовый набор. Атрибут positive принимает значения true, false. В случае когда positive равно true статический анализатор должен выдать предупреждение для тестового примера, в случае, когда positive равно false нет. Атрибуты line и pos обязательны для тестовых примеров с атрибутом positive равным true. Это, соответственно, номер строки и позиция, с которой начинается ошибка в тестовом примере, которую должен обнаружить статический анализатор. Данные атрибуны не инициализируются, если тестовый пример не содержит ошибки. Атрибут problems обязателен для тестовых примеров с несколькими ошибками, о которых должен предупредить статический анализатор. Атрибут problems это список кортежей. Каждый кортеж содержит элементы line и pos, которые указывают на соответствующую ошибку в тестовом примере. Другие атрибуты интерфейса TestCase это file_path, test_id, testcase_directory, entire_file,

testcase_goal, compiler_flags, classname, name, length. File_path путь до файла с исходным кодом, для которого проводиться статический анализ. Значени file_path берется из файла аннотации. Test_id – уникальный номер тестового примера. Возможна ситуация, когда для тестого примера необходима своя директория. В этом случае в атрибут testcase_directory из файла аннтоции загружается название директории. Статический анализатор сканирует файлы внутри данной директории. Entire_file – флаг, сигнализирующий, что ошибка в файле с исходным кодом не имеет позиции, номера строки и распространяется на весь файл. Иными словами статический анализатор должен выдать предупреждение на таком примере для всего файла. Testcase_goal описывает тестируемое в примере поведение. Compiler_flags используется при загрузке тестовых примеров, написанных на языках C/C++. Classname и name используются в формировании отчета. Length – количество строк в файле с исходным кодом.

Tool. Это интерфейс, позволяющий добавлять новые инструменты статического анализа к фреймворку. Реализация этого интерфейса зависит от настроек фреймворка, переданных в качестве аргументов командной строки. Добавление нового инструмента статического анализа осуществляется в несколько итераций. Первая итерация – установить статический анализатор в пространстве пользователя ПК. Вторая итерация – выделить из списка тестовых примеров те, которые соответствуют проверкам, реализуемым статическим анализатором. Для данных тестовых примеров строится отображение ошибок: типы дефектов из тестового набора соотносятся с типами дефектов, которые проверяет статический анализатор. Третья итерация – добавить точку запуска статического анализатора внутрь фреймворка. Для этого есть два возможных варианта. Первый вариант это обращение к статическому анализатору через внутренние интерфейсы данного анализатора. Второй вариант это запуск статического анализатора в интерфейсе командной строки отдельным процессом. После выполнения последней итерации статический анализатор готов к запуску на тестовом наборе. Фреймворк запускает статический анализатор отдельно на каждом тестовом примере из тестового набора или на полном тестовом наборе, в зависимости от поддерживаемого интерфейса тестируемого инструмента статического анализа. В запуске участвуют только тестовые примеры, для которых построено отображение ошибок. Результат запуска инструмента статического анализа на тестовом примере это либо предупреждение об ошибке, либо отсутствие какого-либо выходного значения, либо сообщение об ошибке. Результат запуска статического анализатора и тестовый

пример агрегируются. Получается тестовый набор и набор выходных значений статического анализатора, соответствующий тестовым примерам. Данная структура далее анализируется фреймворком. Фреймворк на основе выходного значения устанавливает статус для тестового примера. Статус SKIP – статический анализатор пропустил данный тестовый пример. Статус FAILED – статический анализатор выдал предупреждение для тестового примера, который не содержит ошибку или не выдал предупреждение для тестового примера, содержащего ошибку. Статус SUCCESS – статический анализатор выдал предупреждение для тестового примера, который содержит ошибку или не выдал предупреждение для тестового примера, который не содержит ошибку. На выходе все тестовые примеры из тестого набора промаркированы одним из возможных статусов. Внутреннее представление тестового набора готово для формирования отчета.

Reporter. Это интерфейс, позволяющий отображать результаты анализа с использованием единого представления результатов запуска инструмента статического анализа на тестовом наборе. Представление результата возможно в трех вариантах: обычный текст, JUnit XML, HTML.

3.3 Тестовый набор

Тестовый набор представляет из себя множество каталогов. Каждый каталог соответствует определенному типу дефектов. Внутри каталога находяться файлы аннотации и файлы с исходным кодом. Тестовый набор состоит из 3 частей. Первая часть описывает общие проблемы в программировании, такие как программные ошибки, низкоэффективный и сложноподдерживаемый коды, содержит требования к стилю написания программы, обработке исключений, параллелизма и производительности для исходного кода, написанного на языке Python.

Вторая часть тестового набора содержит тестовые примеры, описывающие возможные ошибки кодирования на языке Python, которые могут привести к уязвимости безопасности, рискам с точки зрения проверки данных, исключений, операций ввода-вывода, распаковки и запаковки данных.

Третья часть тестового набора содержит тестовые примеры согласные с MITRE Common Weakness Enumeration. На данный момент тестовые примеры соответствуют только одному правилу из CWE – CWE 915. CWE 915 – “Improperly

Controlled Modification of Dynamically-Determined Object Attributes", неправильно контролируемое изменение динамически определяемых атрибутов объекта.

Заключение

Список сокращений и условных обозначений

Словарь терминов

TeX : Система компьютерной вёрстки, разработанная американским профессором информатики Дональдом Кнутом

Список литературы

1. MISRA-C:1998 Guidelines for the use of the C language in vehicle based sofware. — Nuneaton, Warwickshire CV10 0TU, UK : MIRA Ltd, 1998.
2. SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems, 2016th ed. — Software Engineering, Carnegie Mellon University, 2016.
3. JPL institutional coding standard for the C programming language. — Jet Propulsion Laboratory, California Institute of Technology, 2009.
4. *Perforce-Software-Inc.* Rogue Wave Software, Klocwork / Perforce-Software-Inc. — URL: https://www.perforce.com/products/klocwork (visited on 01/03/2021).
5. *GrammaTech-Inc.* GrammaTech, CodeSonar / GrammaTech-Inc. — URL: https://www.grammatech.com/codesonar-cc (visited on 01/03/2021).
6. *The-MathWorks-Inc.* Mathworks, Polyspace Bug Finder / The-MathWorks-Inc. — URL: https://www.mathworks.com/products/polyspace-bug-finder.html (visited on 01/03/2021).
7. *Synopsys-Inc.* Synopsys, Coverity / Synopsys-Inc. — URL: https://scan.coverity.com (visited on 01/03/2021).
8. Rice's theorem. — URL: https://en.wikipedia.org/wiki/Rice%27s_theorem (visited on 03/12/2021).
9. *AbsInt*. AbsInt Angewandte Informatik GmbH, Astree / AbsInt. — URL: https://www.absint.com/astree/index.htm (visited on 01/03/2021).
10. *FRAMA-C*. The Frama-C platform web site, Frama-C / FRAMA-C. — URL: https://frama-c.com (visited on 01/03/2021).
11. *The-MathWorks-Inc.* Mathworks, Polyspace Code Prover / The-MathWorks-Inc. — URL: https://www.mathworks.com/products/polyspace-code-prover.html (visited on 01/03/2021).
12. *AdaCore*. CodePeer / AdaCore. — URL: https://www.adacore.com/codepeer (visited on 01/03/2021).
13. *Lions, J. L.* ARIANE 5 Flight 501 Failure: Report by the Enquiry Board / J. L. Lions //. — 1996.

14. *C., B.* Review of D.D. McCracken's "Digital Computer Programming" / B. C. // Mathematical Tables and Other Aids to Computation. — 1957. — Oct. — Vol. 60, no. 1. — P. 298—305.

15. *Myers, G. J.* The Art of Software Testing / G. J. Myers. — New York, 1979. — 177 p.

16. *Gelperin, D.* The growth of software testing / D. Gelperin, B. Hetzel // Commun. ACM. — 1988. — Vol. 6, no. 31. — P. 687—695.

17. *Dijkstra, E. W.* On the reliability of the programs / E. W. Dijkstra. — URL: https://www.cs.utexas.edu/users/EWD/ewd03xx/EWD303.PDF (visited on 01/07/2021).

18. *Myers, G. J.* The art of software testing. Second edition. / G. J. Myers. — New Jersey : john wilson, sons inc. hoboken, 2004. — 151 p.

19. *BV, T. S.* TIOBE Index for January 2021 / T. S. BV. — URL: https://www.tiobe.com/tiobe-index/ (visited on 02/02/2021).

20. *Bagnara, R.* The MISRA C Coding Standard and its Role in the Development and Analysis of Safety- and Security-Critical Embedded Software / R. Bagnara, A. Bagnara, P. M. Hill. — 2018. — arXiv: 1809.00821 [cs.PL].

21. *Leo Broukhis Simon Cooper, L. C. N.* The International Obfuscated C Code Contest / L. C. N. Leo Broukhis Simon Cooper. — URL: http://www.ioccc.org/years.html#2020 (visited on 02/02/2021).

22. iec 61508-1:2010: functional safety of electrical electronic/programmable electronic safety-related systems. — geneva, switzerland: iec : iec, 2010.

23. varieties of static analyzers: a comparison with astree / P. Cousot [et al.] // first joint ieee/ifip symposium on theoretical aspects of software engineering (tase 2007). — IEEE Computer Society. Shanghai, China, 06/2007. — P. 3—20.

24. *Bagnara, R.* BARR-C:2018 and MISRA C:2012: Synergy Between the Two Most Widely Used C Coding Standards / R. Bagnara, M. Barr, P. M. Hill. — 2020. — arXiv: 2003.06893 [cs.PL].

25. *Barr, M.* BARR-C:2018 - Embedded C Coding Standard / M. Barr. — 2018. — URL: https://barrgroup.com (visited on 02/04/2021).

26. MISRA-C:2004 Guidelines for the use of the C language in critical systems. — Nuneaton, Warwickshire CV10 0TU, UK : MIRA Ltd, 2004.

27. MISRA C++:2008 Guidelines for the use of the C++ language in critical systems. — Nuneaton, Warwickshire CV10 0TU, UK : MIRA Ltd, 2008.

28. IEC 9899:1990: Programming Languages - C. — C. Geneva, Switzerland : ISO/IEC, 1990.

29. MISRA C:2012 Guidelines for the use of the C++ language in critical systems. — Nuneaton, Warwickshire CV10 0TU, UK : MIRA Ltd, 2012.

30. IEC 9899:1999: Programming Languages - C. — C. Geneva, Switzerland : ISO/IEC, 1999.

31. JSF Air vehicle C++ coding standards for the system development and demonstration program. — Lockheed Martin Corporation, 2005.

32. The Power of Ten: Rules for Developing Safety-Critical Code // IEEE Computer. — 2006. — P. 93—95.

33. High Integrity C++. — United Kingdom : Programming Research Ltd, 2013. — 155 p.

34. Embedded System development Coding Reference guide, C Language Edition. — Japan : Software Reliability Enhancement Center, Technology Headquarters, Information-technology Promotion Agency, 2018. — 206 p.

35. Embedded System development Coding Reference guide, C++ Language Edition. — Japan : Software Reliability Enhancement Center, Technology Headquarters, Information-technology Promotion Agency, 2013. — 222 p.

36. guidelines for the use of the c++14 language in critical and safety-related systems. — autosar, 2017.

37. BegBunch: Benchmarking for C bug detection tools / C. Cifuentes [et al.]. — 2009.

38. *Zitser*, *M.* Testing static analysis tools using exploitable buffer overflows from open source code / M. Zitser, R. Lippmann // International Symposium on Foundations of Software Engineering. — 2004. — P. 97—106.

39. *Kratkiewicz*, *K.* Using a diagnostic corpus of C programs to evaluate buffer overflow detection by static analysis tools / K. Kratkiewicz, R. Lippmann // Workshop on the Evaluation of Software Defect Detection Tools. — 2005. — June.

40. NIST. National Institute of Standards and Technology SAMATE Reference Dataset (SRD) project. — URL: http : / / samate . nist . gov / SRD (visited on 02/10/2021).

41. *Corporation*, *M.* Common Weakness Enumeration / M. Corporation. — URL: http://cwe.mitre.org/ (visited on 02/10/2021).

42. *Lu*, *S.* BugBench: A benchmark for evaluating bug detection tools / S. Lu, Z. Li // In Proc. of Workshop on the Evaluation of Software Defect Detection Tools. — 2005. — June.

43. *Heckman*, *S.* On establishing a benchmark for evaluating static analysis alert prioritization and classification techniques / S. Heckman, L. Williams. // In Proc. of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement. — 2008. — Oct.

44. *Johnson*, *S. C.* Lint, a C Program Checker / S. C. Johnson // COMP. SCI. TECH. REP. — 1978. — C. 78—1273.

45. *Zeng*, *B.* Static Analysis on Binary Code / B. Zeng //. — 2012.

46. *Eagle*, *C.* The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler / C. Eagle. — USA : No Starch Press, 2008.

47. *Chow*, *F.* Intermediate Representation / F. Chow // Communications of the ACM. — 2013. — Дек. — Т. 56. — С. 57—62.

48. *Merz*, *F.* LLBMC: Bounded Model Checking of C and C++ Programs Using a Compiler IR / F. Merz, S. Falke, C. Sinz // Verified Software: Theories, Tools, Experiments / под ред. R. Joshi, P. Müller, A. Podelski. — Berlin, Heidelberg : Springer Berlin Heidelberg, 2012. — С. 146—161.

Список рисунков

Список таблиц

Приложение А

Примеры вставки листингов программного кода