Динамический анализ программ с целью поиска ошибок и уязвимостей при помощи целенаправленной генерации входных данных

Вартанов С. П., Герасимов А. Ю.
svartanov@ispras.ru, agerasimov@ispras.ru

Аннотация. В статье описываются принципы проведения динамического анализа программ с целью обнаружения в них дефектов различного рода при помощи целенаправленной генерации входных данных. Рассматриваются методы преобразования программ для извлечения трасс выполнения, механизмы отслеживания потоков данных и построения входных данных для покрытия путей выполнения. Показывается, как подобный подход позволяет проводить анализ полностью автоматически без привлечения разработчиков и тестировщиков, на основе исполняемого или интерпретируемого кода. Приводится описание инструментов динамического анализа, разработанных в Институте системного программирования РАН, — инструмента Avalanche, основанного на фреймворке Valgrind, и прототипа инструмента, производящего анализ программ на языке Java. В конце обзора приводятся результаты работы инструмента Avalanche на проектах с открытым кодом и результаты проведения динамического анализа Java-программ с целью поиска ошибок синхронизации.

Ключевые слова: динамический анализ, анализ программ

1. Введение

Современное программное обеспечение используется в различных областях человеческой жизни. Информатизация сфер, связанных с риском для жизни, предъявляет особые требования к качеству выпускаемых программных продуктов.

Известны различные подходы к проверке качества программного продукта:

- тестирование (ручное, полуавтоматическое, автоматическое) — проверка на соответствие спецификации,
- верификация — проверка на соответствие модели или доказательство верности,
- анализ программ на наличие дефектов и уязвимостей.

Все методы можно классифицировать по степени вовлечённости человека в процесс оценки качества программного продукта. Как правило, роль человека в проведении оценки качества связана с необходимостью понимания работы программы, определения критических мест в программе. Это означает требование высокой квалификации проверяющего, знаний об анализируемом продукте и в то же время приводит к значительным временным затратам и не исключает влияние человеческого фактора на оценку качества программ.

В связи с этим желательно, чтобы анализ проводился в полностью автоматическом режиме или роль человека при оценке качества программного продукта была сведена к минимуму. В полностью автоматическом режиме процесс поиска ошибок, связанных с несоответствием готового продукта спецификации, требует наличия спецификации в строго формализованной форме, процесс составления которой также требует участия человека.

При этом очевидно, что необходимо обеспечить отсутствие в программных продуктах ошибок, неизбежно приводящих к нежелательным последствиям, таких как критический отказ программы в случае выполнения недопустимой операции или порча данных, с которыми работает программа. Для поиска ошибок такого рода в настоящее время активно используются два основных метода полностью автоматического анализа программ — статический и динамический.

Структура статьи имеет следующий вид. В разделе 2 и 3 производится обзор методов анализа программ и имеющихся на сегодняшний день решений в области динамического анализа программ. В разделе 4 приведено описание подхода к решению задачи автоматического поиска дефектов в программах путём проведения итеративного динамического анализа программ для целенаправленной генерации входных данных. В разделе 5 рассматриваются особенности практической реализации данного подхода на основе инструментов Avalanche и прототипа инструмента для анализа Java-приложений. В заключении рассматриваются ограничения и сложности, возникающие в процессе динамического анализа программ и проводится обзор возможных направлений дальнейших исследований.

2. Сравнение статического и динамического анализа

Методы *статического анализа* программ предполагают проведение анализа без запуска программы на исполнение с помощью автоматического построения модели программы и последующей обработки построенной модели. Модель программы может быть построена таким образом, что это позволит её проводить частичный или параллельный анализ. Такой подход позволяет существенно сократить время анализа программы. Как правило, модель программы имеет некоторую степень приближения к реальному поведению программы в процессе запуска. В связи с этим поиск ошибок с помощью статического анализа может иметь как ложные срабатывания, так и не обнаруживать некоторые ошибки, присутствующие в программе.

Статический анализ может проводиться, в том числе, в процессе написания исходного кода программы в интегрированной среде разработки, что позволяет разработчикам обнаруживать и исправлять найденные ошибки в процессе написания программы и до её фактической компиляции в исполняемый код [1, 2]. В свою очередь это позволяет уменьшить стоимость исправления ошибки.

Динамический анализ основан на запуске исследуемого продукта на исполнение. Несомненным преимуществом динамического анализа является отсутствие каких-либо предположений о ходе исполнения программы и проверка её в процессе или сразу после исполнения. При этом одно из основных требований, предъявляемых к динамическому анализу — само проведение анализа должно насколько это возможно меньшим образом влиять на ход исполнения. При определённых условиях на детерминированность программы, динамический анализ позволяет избежать проблемы ложных срабатываний.

Главный минус динамического анализа состоит в том, что для получения качественного покрытия анализируемой программы, как правило, требуется неоднократный запуск программы на выполнение, что связано с большими временными затратами. Однако, при обнаружении ошибки в процессе динамического анализа, как правило, возможно сгенерировать входные данные для программы, на которых ошибка воспроизводится. Таким образом появляется возможность исключения ложных срабатываний анализатора.

3. Обзор инструментов динамического анализа программ

На сегодняшний день существует множество инструментов, осуществляющих динамический анализ приложений с целью обнаружения в них ошибок и уязвимостей. Эти инструменты могут быть классифицированы по следующим признакам:

- используемые методы инструментации и построения входных данных,
- зависимость от языков программирования, на которых могут быть написаны анализируемые приложения, или машинная зависимость,
- требование доступности исходного кода приложения,
- необходимость ручного внесения изменений в код,
- типы обнаруживаемых ошибок.

Проведём краткий обзор известных на данный момент решений.

3.1. Инструменты, использующие статическую инструментацию кода программ

Статическая инструментация предполагает предварительную (до начала исполнения) обработку анализируемой программы с целью внедрения в

программу дополнительных инструкций, которые позволяют собирать информацию о ходе выполнения программы, необходимую для анализа.

3.1.1 EXE

EXE [7] — инструмент для поиска ошибок в программах, автоматически генерирующий входные данные, на которых возникают ошибки. Для анализа программ при помощи EXE требуется модификация исходного кода программы. Программист должен указать участки памяти, которые должны восприниматься инструментом, как символьные данные — данные, значения которых изначально не имеют каких-либо ограничений. Например, для того, чтобы пометить 32-битовую переменную `i` как символьную, в код программы необходимо добавить вызов функции `make_symbolic(&i)`. Модифицированный код компилируется при помощи встроенного в EXE компилятора CIL. Полученный в итоге бинарный код используется для осуществления анализа.

EXE выполняет программу *символьно*, т. е. операции не возвращают конкретные значения, в зависимости от своих операндов, а создают ограничения, соответствующие возможности существования тех или иных значений. Если в программе присутствует недетерминизм, из-за подобной модификации EXE может иметь ошибки первого рода.

Если в программе встречается условный переход, зависящий от символьных данных, EXE делает в этом месте развилку, и в одном случае к имеющимся ограничениям добавляются те, что делают переход истинным, в другом — те, что делают переход ложным. Для вычисления выполнимости условий используется решатель Simple Theorem Prover (STP) [5].

В случае возникновения в программе ошибки, которая приводит к аварийному завершению программы, EXE генерирует набор входных данных — тест. EXE обеспечивает высокую степень покрытия исследуемой программы и предоставляет пользователю наборы входных данных для воспроизведения обнаруженных ошибок.

3.2. Инструменты, использующие исследование и обработку трасс выполнения программ без анализа исходного кода

3.2.1 KLEE

Идейное продолжение EXE — инструмент KLEE [8] — символьная виртуальная машина, основанная на компиляторной инфраструктуре LLVM [9] — низкоуровневой виртуальной машине.

Разработкой KLEE занималась та же группа Стенфордского университета, которая работала над созданием EXE. Инструмент возник в результате серьёзной переработки EXE. Главным отличием KLEE от EXE можно считать то обстоятельство, что для анализа приложения при помощи KLEE ручная

модификация кода не требуется. Но необходимо наличие исходного кода и его компиляция при помощи `llvm-gcc`. При этом для повышения эффективности анализа KLEE предоставляет интерфейс для модификации исходного кода без внесения в него побочных эффектов.

Процесс анализа приложения унаследован инструментом KLEE от EXE практически без изменений. Каждый символьный процесс имеет свои файл регистров, стек, кучу, счётчик команд и ограничения пути. KLEE интерпретирует набор инструкций, скомпилированный LLVM, и отображает инструкции в ограничения без аппроксимаций (с точностью до битов). Большое внимание в инструменте уделяется переменным окружения. Как и в EXE, для проверки выполнимости ограничений используется инструмент STP.

3.2.2 SAGE

SAGE [10] (Scalable, Automated, Guided Execution) — инструмент, разработанный компанией Microsoft совместно с Калифорнийским университетом в Беркли, для анализа Windows-приложений для систем x86.

Основным недостатком EXE, сохранившимся и в KLEE, можно считать необходимость наличия исходного кода программы для её анализа. SAGE лишён этого недостатка, поскольку работает на уровне машинных операций. Это делает инструмент независимым от языка программирования, на котором написано приложение, а также от используемого компилятора. При этом инструмент ориентирован на систему команд x86, что делает его машиннозависимым.

Подход, используемый в SAGE схож с подходом EXE и KLEE. SAGE поддерживает фактическое и символьное состояния программы, представленные парой — байтовое значение и символьный тег, — ассоциированной с каждой ячейкой памяти. Символьный тег — выражение, определяющее, какое значение может принимать данная ячейка.

Также в отличие от рассмотренных ранее инструментов, SAGE при осуществлении анализа не проводит целенаправленную проверку опасных операций, а фокусируется на построении как можно более полного покрытия возможных путей исполнения исследуемой программы.

3.3. Инструменты, использующие динамическую инструментацию программ

Динамическая инструментация означает, что все изменения программы производятся непосредственно перед исполнением соответствующего её блока на процессоре. Инструменты этой группы используют концепцию отслеживания потока «помеченных данных».

Следующие инструменты основаны на среде Valgrind [11]. Valgrind — свободное инструментальное программное обеспечение, предназначенное для осуществления динамической инструментации исполняемого кода. Valgrind

предоставляет средства для создания программных модулей, проводящих динамический анализ. В рамках самого проекта в числе прочих разработаны инструменты для автоматического обнаружения ошибок управления памятью Memcheck и потоками Helgrind.

3.3.1 Flayer

Инструмент Flayer [12] предназначен для динамического анализа бинарного кода программы. При помощи Valgrind производится динамическая инструментация кода с целью выявления выполнения условных операторов. Производится проверка на то, являются ли данные, от которых зависит переход, помеченными, т. е. зависящими от входных данных программы. Если это так, значения условий в операторах изменяются для обеспечения обхода различных путей выполнения программы.

В связи с тем, что изменение значений данных в условных переходах происходит непосредственно в ходе выполнения программы, недостатком работы приложения является возможность ложных срабатываний. Это означает, что воспроизводимость найденной ошибки не гарантируется.

3.3.2 Catchconv

Инструмент Catchconv [13] по схеме своей работы схож с инструментом Flayer, однако лишён проблемы воспроизводимости ошибок.

Подобно инструментам EXE и KLEE, в Catchconv используется решатель STP для проверки выполнимости условий на возможные значения аргументов в условных переходах, составленных инструментом в ходе проведения анализа. Как и в инструменте Flayer, извлечение трассы и обнаружение условных переходов производится при помощи динамического анализа с использованием Valgrind. Среди недостатков Catchconv можно отметить обнаружение инструментом ограниченного класса ошибок. К ним относятся ошибки использования знаковых и беззнаковых типов данных.

3.4. Применение динамического анализа для языка Java

Рассмотрим применение динамических методов анализа для программ, написанных на языке Java. Основная особенность в свете рассматриваемой задачи анализа — трансляция языка в промежуточное представление, байт-код, и его интерпретация при помощи виртуальной машины.

3.4.1 Java PathFinder

Идеи символьных вычислений для проведения динамического анализа программ на языке Java нашли своё отражение в инструменте Java PathFinder [15]. В соответствии с этими идеями инструмент строит наборы булевых ограничений (булевых формул без кванторов над символьными переменными) для путей и на их основе формирует дерево символьного выполнения.

Для проведения символьных вычислений используется инструментация кода с целью получения дерева состояний анализируемого приложения. Состояния включают в себя конфигурацию кучи, ограничения пути и планирование потоков. Это дерево как пространство состояний исследуется при помощи основного механизма инструмента — проверки на модели (model checking).

Инструмент используется для генерации тестовых входных данных и построения контрпримеров для отдельных свойств модели.

3.4.2 Java ThreadSanitizer

Инструмент Java ThreadSanitizer [16], созданный в рамках проекта ThreadSanitizer, использует фреймворк ASM для динамической инструментации байт-кода Java, в ходе которой к функциональности анализируемого приложения добавляется генерация трассы событий. К событиям относятся доступы программы к памяти на чтение или запись, инструкции создания и управления потоками и операции управления синхронизацией потоков.

Получаемая после выполнения программы трасса используется экспериментальным инструментом ThreadSanitizer Offline, который на основе построения отношений предшествования осуществляет поиск возможных состояний гонки.

3.5. Выводы

Рассмотренные инструменты демонстрируют разнообразие подходов, применяемых для динамического анализа приложений. Общей чертой для всех программных средств является просмотр различных путей выполнения программы с целью обнаружения ошибок. Это производится при помощи инструментации или предварительного изменения кода и символьного выполнения программы. При этом одни средства нацелены на обеспечение наиболее полного покрытия всех путей, другие — на покрытие наибольшего числа потенциально опасных операций.

Другие различия состоят в том, основывается ли инструмент на исходном коде программ, на машинном коде или каком-то его промежуточном представлении. Также среди различий можно выделить типы обнаруживаемых ошибок: критические ошибки, связанные с аварийным завершением программы, ошибки, связанные с доступом к неинициализированной памяти или выполнением недопустимых операций, ошибки синхронизации в многопоточных приложениях или утечки выделяемой памяти.

4. Итеративный динамический анализ

Далее в статье более подробно рассматриваются аспекты методов проведения динамического анализа, исследованные в рамках разработки инструмента Avalanche и ряда прототипов для анализа программ на языке Java. В этом

разделе приведены используемые методы и решения и их сравнение с альтернативными способами.

4.1. Способы покрытия

При автоматическом анализе программ важно понимать, на каком этапе находится процесс анализа и когда его можно считать завершённым. На этот вопрос можно дать ответ при помощи определения полноты покрытия программы. Можно говорить, что исследуемый продукт проанализирован полностью, если были выполнены все

- вызовы функций или методов,
- операторы,
- условные переходы,
- или пути выполнения.

Некоторые из приведённых здесь вариантов покрытий целиком включают в себя другие. Например, выполнение всех операторов в программе автоматически означает выполнение и всех условных переходов хотя бы один раз.

Одним из наиболее полных покрытий, включающим в себя множество остальных, является покрытие всех достижимых путей выполнения программы. Такой критерий является очень тяжеловесным. Во-первых, для анализа потребуется совершить число запусков приложения, равное количеству достижимых путей, а оно растёт экспоненциально с ростом числа условных переходов. Во-вторых, возникает алгоритмически неразрешимая проблема останова анализируемой программы, когда для конкретного пути невозможно определить, следует ли прекратить анализ, поскольку программа зациклилась, или подождать её завершения.

Однако, несмотря на все издержки, даже при частичном покрытии программы на основе описанного критерия можно говорить о высоком качестве анализа просмотренных путей. Благодаря методам распараллеливания вычислений и применения распределенных вычислений можно частично решить проблему производительности анализа [17].

4.2. Определение путей выполнения

В дальнейшем в статье, если не оговорено обратное, речь будет идти об анализе программ или об анализе тех частей программы, выполнение которых детерминировано. Это означает, что на каждом шаге работы состояние программы полностью и однозначно (на основе имеющегося кода) зависит от её состояния на предыдущем шаге, а также, что последовательность шагов также детерминирована.

Дополнительно к этому следует оговорить, что любая информация, которая будет использоваться в размышлениях в контексте анализируемой программы

— будь то используемые в программе переменные или входные данные — является дискретной и конечной.

4.3. Формализация задачи построения входных данных

Покажем далее, что задача построения входных данных в указанных ограничениях для неинтерактивных программ может быть сведена к задаче проверки выполнимости булевых ограничений.

Поскольку мы считаем, что программа не интерактивна, её входными данными могут быть файлы, аргументы командной строки и т. п. Вся входная информация дискретна, а следовательно может быть представлена в виде булевого вектора — набора булевых переменных:

$$x = (x_1, \ldots, x_n)$$

Будем также считать, что программа работает с ограниченной дискретной памятью, которая состоит из *k* булевых переменных. Будем рассматривать граф потока управления программы — граф, в котором вершинам соответствуют инструкции программы, а рёбрам — возможные переходы.

Любое выполнение программы можно представить как путь (конечный, если программа остановилась, или бесконечный, в случае, если программа зациклилась) в графе потока управления. Рассмотрим определённый путь π. Пусть его длина есть *t*. Для каждой вершины с номером *i* введём вектор

$$z_i = (z_{i_1}, \ldots, z_{i_k}), \forall i = \overline{1,t}$$

переменных, соответствующих используемым в программе переменным.

Поскольку программа является детерминированной, каждое z_{i_j} зависит только от входных данных и предыдущих состояний программы, т. е. представляет собой булеву функцию от переменных

$$x_1, x_2, \ldots, x_n,$$
$$z_{1_1}, z_{1_2}, \ldots, z_{1_t},$$
$$z_{2_1}, z_{2_2}, \ldots z_{2_t},$$
$$\ldots$$
$$z_{i-1_1}, z_{i-1_2}, \ldots, z_{i-1_t}.$$

Индукцией по номеру вершины пути можно доказать, что любая переменная z_{i_j} зависит только от входных данных. В самом деле, это верно для всех переменных первой вершины — $z_{1_j}, j = \overline{1,t}$. Если это верно для всех вершин с номером, меньшим *l*, то каждая переменная этих вершин имеет вид

$$z_{i_j} = g_{i-1_j}(x), i = \overline{1,l-1}, j = \overline{1,t}.$$

Для вершины *l* каждая переменная имеет вид

$$z_{l_i} = g_{l_i}(x_1, \ldots, x_n, z_{l_1}, \ldots, z_{1_t}, \ldots, z_{l-1_i} \ldots z_{l-1_t}) =$$
$$= g_{l_i}(x_i, \ldots, x_n, g_{1_1}(x), \ldots, g_{1_t}, g_{1_t}(x), \ldots g_{l-1_1}(x), \ldots g_{l-1_t}(x)) =$$
$$= g'_{l_i}(x), j = \overline{1,t}.$$

Это означает, что любая переменная в каждой вершине пути зависит *только* от входных данных. Отметим, что путь однозначно определяется выбором следующего ребра в каждой вершине графа, из которой выходят несколько рёбер. Такие вершины соответствуют условным переходам в программе.

Каждый условный переход определяется условием. В нашем случае условие можно записать как некоторую логическую функцию от переменных из множества *x*. Значение этой функции определяет, будет ли выполнен данный условный переход, т. е. какое из рёбер, исходящих из вершины, соответствующей этому переходу, будет выбрано.

Обозначим множество всех условных переходов в программе как $B = \{B_1, \ldots, B_m\}$ и запишем для каждого перехода B_i функцию $f_i(x)$, соответствующую его условию, то есть переход совершается тогда и только тогда, когда значение функции истинно.

Снова рассмотрим путь π. Как было отмечено, он однозначно определяется всеми выборами в условных переходах. Поставим в соответствие π последовательность всех переходов, встречающихся при обходе пути — $B_\pi = \{B_{i_1}, \ldots, B_{i_r}\}$.

Для каждого перехода B_{i_l} из последовательности определим, какое ребро следует за соответствующей вершиной в пути π. Если ребро имеет пометку, что переход выполнился, поставим ему в соответствие функцию f_{i_1}, иначе — функцию $\overline{f_{i_l}}$. Если функция не тождественно истинна, она может быть представлена в виде конъюнктивной нормальной формы f'_{i_l}.

Выполнимость каждого перехода B_{i_l} определяется равенством истине функции f'_{i_l}. Весь путь однозначно определяется выполнимостью конъюнктивной нормальной формы $F = f'_{i_1} \wedge \ldots \wedge f'_{i_t}$.

Задачу можно формализовать следующим образом. Необходимо найти такие значения переменных из набора *x*, чтобы значение функции *F* было истинно. Эта формулировка полностью соответствует формулировке задачи определения выполнимости булевых формул. Согласно теореме Кука — Левина, эта задача является NP-полной [3].

Для решения задачи проверки выполнимости логических формул на сегодняшний день существует множество инструментов, так называемых решателей. В их число входит, например, MiniSat, эффективно решающий сформулированную выше задачу [4], или STP Constraint Solver, который оперирует с битовыми массивами и поддерживает

- арифметические операции,
- операции сравнения,
- логические операции,
- а также конкатенацию, сдвиги и пр. [5]

4.4. Инвертирование условий

Вернёмся теперь к основной задаче — построению всех допустимых путей в программе. Очевидно, не обязательно все возможные пути в терминах графа потока управления программы являются допустимыми путями выполнения программы. Гарантированно полный проход по всем допустимым путям даёт перебор всех возможных входных значений программы. Одним из возможных решений может быть генерация случайных входных данных и запуск программы на них с последующим запоминанием пройденных путей для подсчёта покрытия. Однако, для каждого пути достаточно получить хотя бы один набор входных данных. Рассмотрим далее способ перебора возможных путей, вместо перебора входных данных.

Одним из таких способов является *инвертирование условий*. Суть метода заключается в следующем. Для первого выполнения программы используется некоторый набор входных данных. Концептуально это может быть абсолютно случайный набор. Однако, эффективность метода повышается, если выполнение программы на начальных входных данных будет, например, содержать как можно больше условных переходов.

Во время первого выполнения программы на начальных входных данных для каждого встретившегося условного перехода строится набор условий, при истинности которых выполнение в этой точке сворачивает по альтернативной ветке. Для каждого набора условий строятся (если возможно) удовлетворяющие им входные данные, происходит запуск программы и описанные действия повторяются для инвертирования оставшихся условий.

Такой подход теоретически даёт возможность обойти все возможные пути в программе.

4.5. Отслеживание потока помеченных данных

Для указанного метода инвертирования условий нужен, во-первых, некоторый механизм построения условий в ходе выполнения программы и создания наборов условий, отвечающих новым путям; во-вторых, механизм преобразования построенных ограничений в булевы формулы и проверки их выполнимости; в-третьих, механизм выбора на каждом новом шаге некоторого нового пути.

Задача преобразования набора логических выражений в булевы формулы, преобразование в конъюнктивную нормальную форму, решение проблемы проверки выполнимости и оптимизации, связанные с каждым из этих процессов, выходят за рамки статьи и не будут рассмотрены здесь. Остановимся на проблеме построения набора ограничений, определяющего каждый отдельный путь на основе запусков анализируемой программы.

Рассмотрим понятие помеченных данных. *Помеченными данными*, используемыми в программе, будем называть любые данные, значения которых зависят от входных данных (в [6] помеченные данные определяются иначе: это либо входные данные, либо результат операции, в которой участвуют помеченные данные). Следует отметить, что свойство «помеченности» данных вообще говоря не является постоянным в ходе выполнения программы. Изначально и постоянно помеченным является исходный набор входных данных. Любые же переменные, используемые в программе могут становиться помеченными или терять это свойство в ходе выполнения, в зависимости от того, в каких операциях они участвуют.

Концепцию помеченных данных легко проиллюстрировать на простом примере:

```
int main(int argc, char** argv)
{
    // Arguments checking

    int a = read_from_file(0);
    int b = 1;

    int c = a + 5;
    int d = b + 5;

    if (c == 6)
    {
        // First block
    }
    else
    {
        // Second block
    }
    if (d == 6)
    {
        // Third block
    }
    else
    {
        // Fourth block
    }
}
```

Переменной a вызовом функции `read_from_file` присваивается значение первого байта некоторого файла, который является частью набора входных данных, следовательно, эта переменная «помечается» в данной точке. Переменная b, очевидно, помеченной не является — её значение в данной точке программы не зависит от того, какие данные подаются программе на вход. Переменная c после присваивания, зависящего от значения переменной a, станет косвенно зависящей от входных данных, и значит, станет помеченной. Иначе дело обстоит с переменной d, значение которой от входных данных не зависит. Таким образом, результат операции считается помеченным, если в ней *существенным* образом участвуют помеченные данные.

Для каждого условного перехода на основе помеченности данных аналогичным образом определяется, зависит ли его условие транзитивно от входных данных. В условии второй if — then — else конструкции в примере переменная d не помечена, а значит условие заведомо не зависит от входных данных. В данном случае всегда будет выполнен третий блок программы, тогда как четвёртый блок — недостижимый код. Условие в первой конструкции существенно зависит от помеченной переменной c и это означает, что, возможно, существуют различные наборы входных данных, при которых в данной точке выполнение программы пойдёт по разным веткам, однако их существование не гарантировано.

Следует отметить, что в реализациях обычно снимается требование *существенного* участия помеченных переменных в операциях для того, чтобы результат считался помеченным. Допустим, имеется помеченная переменная a. Обе конструкции `int b = a - a;` и `int c = a * 0;` осуществляют присваивание нуля переменным b и c, однако в обеих операциях участвует помеченная переменная. Для проверки, является ли это участие существенным, необходимо производить разбор правой части выражений и определять области возможных значений, что требует значительных накладных расходов.

Концепция помеченных данных в рассмотренном подходе позволяет определить условные переходы, условия в которых на конкретном пути не зависят от входных данных и, следовательно, на этом пути не могут быть инвертированы путём изменения входных данных. На практике такое отсечение позволяет существенно сократить количество переходов, условия которых требуют проверки выполнимости, что, в свою очередь, важно, поскольку это NP-полная задача и её решение может требовать значительного времени.

4.6. Поиск ошибок и уязвимостей

Описанные методы позволяют генерировать наборы входных данных программы для покрытия различных путей выполнения. Повышение эффективности анализа программы в этом случае может обеспечиваться

использованием эвристических методов выбора нового пути на каждом последующем шаге. Перебор путей может проходить в глубину, в ширину, или основываться на различных метриках: увеличения покрытия числа условий, инструкций и т. д.

Выполнение определённого пути программы (т. е. запуск программы на конкретных входных данных) позволяет извлечь информацию о ходе выполнения. Эта информация может использоваться как для простой проверки хода выполнения и состояния, в котором программа завершилась, на наличие ошибок, так и для построения различного рода моделей и проверки свойств этих моделей.

Отдельно стоит обратить внимание на методы моделирования для поиска ошибок при помощи проверки операций, потенциально способных приводить к аварийному завершению программы при определённых значениях аргументов. К таким операциям можно отнести деление или разыменование указателя. Для таких операций на каждом пути, в котором они присутствуют, необходимо определить, существуют ли входные данные, на которых значения аргументов для них приводят к ошибке. Для этого используются те же принципы, что и при построении входной информации для инвертирования переходов, с тем лишь отличием, что условие инвертирования заменяется ограничением на значение указанных аргументов.

5. Реализация методов динамического анализа

5.1. Инструмент Avalanche

Описанные выше принципы и методы проведения динамического анализа были реализованы в инструменте Avalanche [6].

Инструмент основан на фреймворке Valgrind. В основе инструмента лежит динамическая инструментация кода, проводимая двумя различными компонентами — программными модулями Valgrind — Tracegrind и Covgrind. Первый программный модуль используется для инструментации программы с целью отслеживания помеченных данных и построения наборов ограничений для инвертирования условных переходов и построения входных данных. Таким образом, инструмент реализует описанный подход к построению различных путей выполнения программы.

Второй программный модуль — Covgrind — инструментирует анализируемую программу с целью определения покрытия базовых блоков. Для каждого нового пути, построенного при помощи инвертирования условных переходов, Covgrind вычисляет количество новых базовых блоков, которые будут покрыты после анализа. На основе такой метрики происходит выбор новых путей выполнения анализируемой программы.

Поиск ошибок в Avalanche основан на определении статуса завершения программы, описанном способе проверки опасных операций, а также

использовании различных программных модулей фреймворка Valgrind, таких как Memcheck для проверки операций работы с памятью и Helgrind для проверки управления потоками.

Также в Avalanche есть механизм анализа сетевых приложений. В этом случае помеченными изначально считаются все данные, поступающие анализируемой программе от сервера.

В дополнение к сказанному, Avalanche предоставляет возможности для повышения эффективности анализа:

- Выделение во входных файлах набора байт, которые необходимо считать изначально помеченными. При этом возникает возможность, например, зафиксировать заголовок входного файла с целью подбора его различного содержимого.
- Указание списка сигнатур функций, выражения в условных операторах которых должны быть инвертированы в ходе анализа. Это позволяет производить локальных анализ некоторых участков программы.

В последней версии программы были добавлены возможности проведения параллельного и распределённого анализа. Как было сказано, после каждой итерации анализа, в зависимости от количества встретившихся условных переходов, строится множество наборов ограничений для их инвертирования. Распараллеливание анализа основано на независимости задач определения выполнимости имеющихся ограничений и запуска анализируемой программы с инструментацией для определения прироста покрытия.

Проведение с помощью Avalanche распределённого анализа на нескольких вычислительных единицах или в распределённой вычислительной среде осуществляется на основе деления построенного дерева достижимых путей программы на ряд поддеревьев и проведения их независимого анализа с последующим объединением результатов. Результаты работы в распределенном и параллельном режиме приведены в работе [17].

5.2. Динамический анализ программ на языке Java

Описанные методы применимы для широкого класса императивных языков программирования. Как следует из обзора существующих решений, эффективные средства решения поставленной задачи методами динамического анализа используются и для интерпретируемых языков, таких, как язык Java.

В рамках исследований авторами был создан прототип средства динамического анализа программ на языке Java. В основу были положены, как и в инструменте Avalanche, принципы отслеживания помеченных данных и построения новых входных данных с помощью решения булевых ограничений.

Основная особенность языка Java, как уже отмечалось, — трансляция в байт-код и интерпретация при помощи виртуальной машины. Формат байт-кода

определяется конкретной машиной, при помощи которой будет производиться интерпретация. В связи с этим, в отличие от Avalanche, в прототипе используется статическая инструментация байт-кода с помощью библиотеки BCEL [14]. Это означает, что преобразование кода с целью внесения в него дополнительной функциональности осуществляется до проведения итеративного анализа, а не повторяется на каждой его итерации.

Такое изменение способа инструментации, хотя и может означать инструментацию частей программы, которые никогда не будут выполнены в ходе анализа (например, это может касаться частей используемых библиотек), сокращает время, затрачиваемое на проведение отдельной итерации, а также позволяет проводить анализ на платформах, поддерживающих Java-машины с иным форматом байт-кода, путём предварительного конвертирования инструментированного байт-кода в требуемый формат. Так, прототип был реализован для инструментации байт-кода формата Java Virtual Machine, однако при наличии средства конвертирования в формат DEX позволяет анализировать программы на платформе Android, использующей для интерпретации байт-кода виртуальную машину Dalvik.

Для обнаружения ошибок данный прототип использует механизм исключений языка Java в том смысле, что любое выброшенное и не отловленное программно исключение считается дефектом. В прототипе отсутствует отдельный механизм инструментации для подсчёта полноты покрытия. Вместо этого информация о числе инструкций оценивается на основе предыдущих запусков. Результаты исследования приведены в статье [18].

6. Заключение

Несмотря на тяжеловесность методов динамического анализа, существующие решения уже позволяют автоматически обнаруживать ошибки в программах.

Однако, стоит отметить, что в области исследований методов динамического анализа программ есть ряд известных проблем, которые не позволяют в настоящее время использовать средства динамического анализа программ в каждодневной работе программистов и сотрудников отделов контроля качества программного обеспечения:

- Проблема экспоненциальной зависимости количества запусков анализатора от количества условных переходов в программе.
- Проблема целенаправленного анализа определенной части программы.
- Проблема влияния инструкций инструментации на общую производительность выполнения и анализа программы.

В связи с этим можно сформулировать ряд задач для дальнейших исследований в области анализа программ.

6.1. Преобразование среды выполнения как альтернатива инструментации

В рамках исследований в ИСП РАН был создан прототип средства, производящего анализ выделения и использования памяти в приложениях на платформе Android. Для интерпретации Java-приложениях Android использует виртуальную машину Dalvik. Так как поставленная задача предусматривает определение использования памяти с точностью до байтов, т. е. требует отслеживания всех операций с памятью, изменение хода выполнения анализируемого приложения при помощи инструментации оказывается неэффективным. В связи с этим было принято решение об изменении самой среды выполнения байт-кода для извлечения необходимой информации, вместо инструментации программы.

Технически решение состоит в создании клиент-серверного приложения, отправляющего и принимающего информацию от анализируемого приложения при помощи отправки сообщений, и изменении кода виртуальной машины Dalvik путём добавления функциональности по отправке сообщений с извлечённой информацией о выделении, высвобождении и использовании памяти.

Подобный подход также может быть использован для динамического анализа с целью поиска ошибок, уязвимостей или неэффективностей любого рода. Несмотря на то, что решение зависит от используемой виртуальной машины, оно может быть значительно эффективнее инструментации.

6.2. Интеграция со статическим анализом

Как отмечалось ранее, один из существенных недостатков статического анализа — необходимость проверки воспроизводимости обнаруженных дефектов. Для решения этой проблемы может быть использовано совмещение двух подходов для устранения ряда недостатков каждого из них.

В случае, если существует возможность средствами статического анализа сделать предположение о возможной трассе выполнения программы, приводящей к ошибке, динамический анализ может подтвердить обнаруженный дефект путём построения входных данных, на которых данный дефект воспроизводится, либо опровергнуть его с помощью доказательства невыполнимости набора утверждений, определяющих соответствующий путь и состояние системы.

В таком случае решается две задачи:

- воспроизведение дефекта, найденного статическим анализатором путём вычисления входных данных в динамическом анализаторе,
- решение проблемы анализа определенной части программы динамическим анализатором.

Список литературы

[1] Савицкий В. О., Сидоров Д. В. «Ленивый» анализ исходного кода на языках C и C++. Труды Института системного программирования РАН, том 23, 2012 г. ISSN 2220-6426 (Online), ISSN 2079-8156 (Print), DOI: 10.15514/ISPRAS-2012-23-8, 133–141 с.

[2] Савицкий В. О., Сидоров Д. В. Инкрементальный анализ исходного кода на языках C/C++. Труды Института системного программирования РАН, том 22, 2012 г. ISSN 2220-6426 (Online), ISSN 2079-8156 (Print), DOI: 10.15514/ISPRAS-2012-22-8, 119–130 с.

[3] Новикова Н. М. Основы оптимизации. М.: МГУ, 1998. 17–22 с.

[4] Eén N., Sörensson N. An Extensible SAT-solver. SAT 2003. P. 502-518.

[5] Ganesh V., Dill D. L. A Decision Procedure for Bit-Vectors and Arrays // In Proceedings of Computer Aided Verification. 2007. P. 524–536.

[6] Исаев И. К., Сидоров Д. В. Применение динамического анализа для генерации входных данных, демонстрирующих критические ошибки и уязвимости в программах // Программирование. 2010. № 4. С. 16.

[7] Cadar C., Ganesh V., Pawlowski P., Dill D., Engler D. EXE: Automatically Generating Inputs of Death // Computer System Laboratory Stanford University. P. 14.

[8] Dunbar D., Cadar C., Engler D. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs // Stanford University. 2008.

[9] Lattner C. The LLVM Compiler Infrastructure [HTML] (http://llvm.org/)

[10] Goldefroid P., Levin. M. Y, Molnar D. SAGE: Whitebox Fuzzing for Security Testing // Communications of the ACM. 2012. 55. P. 40–44.

[11] Valgrind. Instrumentation Framework for Building Dynamic Analysis Tools [HTML] (http://valgrind.org/)

[12] Drewry W., Ormandy T. Flayer: Taint analysis and flow alteration tool [HTML] (http://code.google.com/p/flayer/)

[13] Molnar D., Wagner D. Catchconv: Symbolic execution and run-time type inference for integer conversion errors // UC Berkeley, 2007.

[14] Apache Commons Byte Code Engineering Library [HTML] (http://commons.apache.org/bcel)

[15] Visser W., Păsăreanu C. S., Khurshid S. Test Input Generation with Java PathFinder // Proceeding ISSTA '04 Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis. P. 97–107.

[16] Serebryany K., Iskhodzhanov T. ThreadSanitizer—data race detection in practice. WBIA '09, New York City, NY, USA, 2009

[17] Ермаков М. К., Герасимов А. Ю. Avalanche: применение параллельного и распределенного динамического анализа программ для ускорения поиска дефектов и уязвимостей. Труды Института системного программирования РАН, том 25, 2013 г. ISSN 2220-6426 (Online), ISSN 2079-8156 (Print), DOI: 10.15514/ISPRAS-2013-25-2, стр. 29-38.

[18] Вартанов С. П., Герасимов А. Ю. Применение динамического анализа для поиска дефектов в программах на языке Java. Труды Института системного программирования РАН, том 25, 2013 г. ISSN 2220-6426 (Online), ISSN 2079-8156 (Print), DOI: 10.15514/ISPRAS-2013-25-1, стр. 9-28.

Dynamic program analysis for error detection using goal-seeking input data generation

Vartanov S. P., Gerasimov A. Y.
svartanov@ispras.ru, agerasimov@ispras.ru

Abstract. This paper describes the principles of program dynamic analysis for defect detection using input data generation. Presented comparison of dynamic vs static analysis od programs, overview of existing tools for dynamшс analysis such as EXE, KLEE, SAGE, Flayer, Catchconv, Java PathFinder, Java ThreadSanitizer. Techniques of program transformation allowing execution trace extraction, data flow tracing and input data generation for execution path coverage approaches are considered. We clarify in what way such an approach allows us to perform fully automatic analysis using executable or interpretable code based on iterational dynamic analysis with automatic conditional branches alternation through input data generation for target program. This paper also presents dynamic analysis tools developed at Institute for System Programming RAS---Avalanche (Valgrind-based tool) and a prototype tool for Java applications. These tools allows to find critical defects programs which lead to program crash and generate input data sets for reproducing found defects. The paper concludes with an evaluation of practical results of applying Avalanche tool to a set of open source projects as well as results of applying Java analysis tool to detect concurrency defects and describes possible directions for future research.

Keywords: dynamic analysis, program analysis

References

[1]. V. O. Savitsky, D. V. Sidorov. «Lenivyj» analiz iskhodnogo koda na yazykakh C i C++ [Lazy source code analysis for C/C++ languages] Trudy ISP RAN [The Proceedings of ISP RAS], 2012, vol 23, pp. 133-141. DOI: 10.15514/ISPRAS-2012-23-8. (in Russian)

[2]. V. O. Savitsky, D. V. Sidorov, Inkremental'nyj analiz iskhodnogo koda na yazykakh C/C++[Incremental source code analysis for C/C++ languages] Trudy ISP RAN [The Proceedings of ISP RAS], 2012, vol 22, pp. 119-129. DOI: 10.15514/ISPRAS-2012-22-8. (in Russian)

[3]. Novikova N. M. Osnovy optimizatsii [The Basics of Optimization]. M.: MGU [MSU], 1998. 17–22 p. (in Russian)

[4]. Eén N., Sörensson N. An Extensible SAT-solver. SAT 2003. P. 502-518.

[5]. Ganesh V., Dill D. L. A Decision Procedure for Bit-Vectors and Arrays // In Proceedings of Computer Aided Verification. 2007. P. 524–536.

[6]. Isaev I. K., Sidorov D. V. Primenenie dinamicheskogo analiza dlya generatsii vkhodnykh dannykh, demonstriruyushhikh kriticheskie oshibki i uyazvimosti v programmakh [The Use of Dynamic Analysis for Generation of Input Data that Demonstrates Critical Bugs and Vulnerabilities in Programs]. Programmirovanie [Programming and Computer Software]. 2010. # 4. P. 1-16. (in Russian)

[7]. Cadar C., Ganesh V., Pawlowski P., Dill D., Engler D. EXE: Automatically Generating Inputs of Death // Computer System Laboratory Stanford University. P. 14.

[8]. Dunbar D., Cadar C., Engler D. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs // Stanford University. 2008.

[9]. Lattner C. The LLVM Compiler Infrastructure [HTML] (http://llvm.org/)

[10]. Goldefroid P., Levin. M. Y, Molnar D. SAGE: Whitebox Fuzzing for Security Testing // Communications of the ACM. 2012. 55. P. 40–44.

[11]. Valgrind. Instrumentation Framework for Building Dynamic Analysis Tools [HTML] (http://valgrind.org/)

[12]. Drewry W., Ormandy T. Flayer: Taint analysis and flow alteration tool [HTML] (http://code.google.com/p/flayer/)

[13]. Molnar D., Wagner D. Catchconv: Symbolic execution and run-time type inference for integer conversion errors // UC Berkeley, 2007.

[14]. Apache Commons Byte Code Engineering Library [HTML] (http://commons.apache.org/bcel)

[15]. Visser W., Păsăreanu C. S., Khurshid S. Test Input Generation with Java PathFinder // Proceeding ISSTA '04 Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis. P. 97–107.

[16]. Serebryany K., Iskhodzhanov T. ThreadSanitizer—data race detection in practice. WBIA '09, New York City, NY, USA, 2009

[17]. Ermakov M. K., Gerasimov A. Y. Avalanche: adaptation of parallel and distributed computing for dynamic analysis to improve performance of defect detection [Avalanche: primenenie parallel'nogo i raspredelennogo dinamicheskogo analiza programm dlya uskoreniya poiska defektov i uyazvimostej] Trudy ISP RAN [The Proceedings of ISP RAS], 2013, vol 25, pp. 29-38. DOI: 10.15514/ISPRAS-2013-25-2. (in Russian)

[18]. Vartanov S. P., Gerasimov A. Y. Applying dynamic analysis for defect detection in Java-applications [Primenenie dinamicheskogo analiza dlya poiska defektov v programmakh na yazyke Java] Trudy ISP RAN [The Proceedings of ISP RAS], 2013, vol 25, pp. 9-28. DOI: 10.15514/ISPRAS-2013-25-1. (in Russian)