Федеральное государственное бюджетное образовательное учреждение высшего образования Московский государственный университет имени М. В. Ломоносова
Федеральное государственное бюджетное учреждение науки
Институт системного программирования Российской академии наук

На правах рукописи

Ермаков Михаил Кириллович

Методы повышения эффективности итеративного динамического анализа программ

05.13.11 — математическое и программное обеспечение вычислительных машин, комплексов и компьютерных сетей

Диссертация на соискание ученой степени
кандидата технических наук

Научный руководитель:
д. ф.-м. н., академик РАН
Иванников Виктор Петрович

Москва - 2016

Содержание

Введение

Актуальность

В настоящее время разработка программного обеспечения является активно развивающейся и востребованной областью. Сложность создаваемых программных комплексов повышается, а требования к их надёжности возрастают, что в значительной степени затрудняет сам процесс разработки. Для решения подобных проблем используются инструменты, позволяющие полностью или частично автоматизировать этапы процесса разработки. Среди подобных инструментов можно выделить группу программных средств, осуществляющих проверку качества программного кода.

Традиционно выделяют следующие две обширные группы подходов к исследованию качества программ: статический анализ и динамический анализ. При проведении статического анализа не производятся запуски исследуемой программы на выполнение — инструменты статического анализа автоматически создают структуры данных, описывающие код программы в различных представлениях (например, абстрактное синтаксическое дерево или граф потока управления) и осуществляют обработку этих структур.

Динамический анализ заключается в исследовании программ во время их выполнения. Это позволяет исследовать программы, если инструменту анализа доступен исполняемый код программы, но не доступен её исходный код.

Методы статического анализа хорошо масштабируемы, однако зачастую имеют высокий уровень ложных срабатываний. Методы динамического анализа позволяют исследовать поведение программы при её выполнении на конкретных наборах входных данных, что практически полностью устраняет ложные срабатывания. В то же время, динамический анализ позволяет проверять только те фрагменты кода программы, которые были реально выполнены при проведении анализа. Получение точной оценки качества кода программы с помощью методов динамического анализа обычно требует множественных запусков программы на

выполнение на различных наборах входных данных.

Данная особенность динамического анализа делает крайне актуальными методы, позволяющие автоматически строить наборы входных данных для исследования различных путей выполнения программы. Для реализации подобного подхода могут быть использованы принципы символьного исполнения программ. Символьное исполнение позволяет связывать пути выполнения программы с входными данными с помощью наборов булевых формул. Запуск программы на наборе входных данных позволяет исследовать путь выполнения *A* и получить для него трассу ограничений, состоящую из указанных выше формул. Полученная трасса может быть трансформирована таким образом, чтобы соответствовать потенциальному пути выполнения *B*. С помощью инструментов проверки выполнимости булевых формул возможно автоматически построить набор входных данных, запуск программы на котором приведёт к её выполнению по пути *B*.

Ключевым ограничением систем, осуществляющих автоматический обход путей выполнения программ на основе символьного исполнения, является чрезвычайно высокая вычислительная сложность этого подхода. Количество путей выполнения в программах растёт экспоненциально с увеличением объёма кода, а задача проверки выполнимости булевых формул не имеет в настоящее время полиномиальных алгоритмов решения.

Подобные ограничения делают крайне актуальными исследования, направленные на разработку механизмов повышения эффективности динамического анализа, включающего автоматический обход путей выполнения программы на основе символьного исполнения.

Добиться подобного ускорения можно путём обхода только части путей выполнения программы. При наличии знаний о структуре анализируемой программы возможно сформулировать критерии выбора путей выполнения программы таким образом, чтобы проводить целенаправленное исследование отдельных фрагментов кода. Существующие инструменты обхода путей

выполнения программы на основе символьного исполнения либо не предоставляют подобные возможности частичного анализа, либо проведение частичного анализа с их помощью накладывает дополнительные ограничения на анализируемые программы. В то же время, проведение частичного анализа представляется актуальным в процессе разработки программ — настройку анализа может производить разработчик или тестировщик, обладающий знаниями о внутреннем устройстве программы и особенностях входных данных.

Среди инструментов автоматического обхода путей выполнения программы на основе символьного исполнения для анализа непосредственного выполнения программы используются виртуальные машины и эмуляторы, системы предварительной инструментации исходного кода программы и системы динамической инструментации исполняемого кода. Инструментация кода программы подразумевает его модификацию путём внедрения кода, который не нарушает исходную функциональность программы и позволяет извлекать дополнительную информацию при выполнении программы.

Существующие методики предварительной инструментации исполняемого кода слабо представлены среди инструментов анализа программ, включающего автоматический обход путей выполнения программ на основе символьного исполнения.. В то же время предварительная инструментация исполняемого кода не требует наличия исходного кода программы и позволяет снизить накладные расходы по сравнению с динамической инструментацией исполняемого кода. В первую очередь, это достигается за счёт того, что разбор кода программы и его модификация производятся однократно, в то время как один и тот же фрагмент кода может выполняться на большом количестве путей. Во-вторых, предварительная инструментация проводится независимо от выполнения программы, что позволяет более эффективно распределять рабочую нагрузку при использовании нескольких вычислительных узлов. Возможность распределения нагрузки на несколько вычислительных узлов при использовании предварительной инструментации является крайне актуальной в настоящее время

при анализе программ, запускаемых на мобильных устройствах (например, платформ Android/ARM и Tizen/ARM), обладающих меньшим объемом ресурсов по сравнению с традиционными аналогами.

Целью диссертационной работы является разработка и реализация методов увеличения эффективности динамического анализа программ, включающего автоматический обход путей выполнения программ на основе символьного исполнения, с помощью ограничения количества исследуемых путей выполнения по пользовательским спецификациям и применения статической инструментации исполняемого кода для извлечения трасс выполнения программ.

Для достижения поставленной цели были определены **следующие задачи**:

1. Провести обзор методов динамического анализа программ на основе символьного исполнения и методов статической инструментации кода; выбрать инструментальные средства, на базе которых будет проводится разработка и реализация предлагаемых в работе методов.
2. Разработать и реализовать метод частичного анализа программ, включающего автоматический обход подмножества путей выполнения программы на основе символьного исполнения, где подмножество путей задаётся с помощью пользовательских спецификаций.
3. Разработать и реализовать схему параллельной обработки независимых путей выполнения в рамках выбранного средства динамического анализа с целью повышения эффективности использования вычислительных ресурсов.
4. Разработать метод статической инструментации исполняемого кода и реализовать программную систему, предоставляющей реализацию данного метода для платформы ARM/Linux. Разработать и реализовать на основе данной системы модули построения трасс выполнения программы для использования в рамках выбранного средства динамического анализа.
5. Провести оценку эффективности разработанных методов.

Научная новизна

В рамках диссертационной работы получены следующие результаты, обладающие научной новизной:

1. Предложен метод частичного динамического анализа программ, включающего автоматический обход подмножества путей выполнения программы, задаваемого пользовательскими спецификациями на источники входных данных и точки ветвления в коде программы. Предложенный метод не накладывает ограничений на формат входных данных программы и не требует непосредственного доступа к исходному коду программ.
2. Предложен метод статической инструментации исполняемого кода для извлечения трасс выполнения программы с целью построения наборов входных данных и оценки их приоритетности в рамках метода автоматического обхода путей выполнения программы.

Теоретическая и практическая значимость

Предложены методы, повышающие эффективность динамического анализа программ, включающего автоматический обход путей выполнения программ на основе символьного исполнения, и методы статической инструментации исполняемого кода.

Предложенные методы и полученные в процессе разработки методов наблюдения могут быть включены в курсы анализа программ, автоматизации тестирования и компиляторных технологий, преподаваемых в высших учебных заведениях, специализирующихся на исследовании информационных технологий. Реализация предложенных методов проведена в рамках инструментальных средств с открытым исходным кодом, что позволяет использовать данные методы при решении задач анализа программ независимым исследователям и при решении прикладных и промышленных задач анализа конкретных программных продуктов разработчикам и тестировщикам.

Апробация работы

Основные результаты диссертационной работы докладывались на следующих

конференциях и семинарах:

1. Третья международная научно-техническая конференция «Инструменты и методы анализа программ-2015» (Санкт-Петербург, Россия, 2015)
2. Открытая конференция по компиляторным технологиям (Москва, Россия, 2015)
3. Научно-исследовательский семинар Института системного программирования РАН

Публикации

Основные результаты опубликованы в работах [1] — [4]. Работы [1, 2, 3] опубликованы в изданиях перечня ВАК, работа [4] опубликована в сборнике трудов международной конференции. В работе [1] А.Ю. Герасимов выполнил написание вводного раздела статьи. В работе [2] С.П. Вартанов участвовал в создании программной реализации системы инструментации.

Личный вклад

Все представленные результаты были получены автором лично.

Структура диссертации

Диссертация состоит из введения, 4 глав и заключения. Работа изложена на 116 страницах. Список источников насчитывает 77 наименований. Диссертация содержит 5 таблиц и 31 рисунок.

Глава 1. Символьное исполнение и динамический анализ программ

1.1 Символьное исполнение программ

1.1.1 Базовые принципы исполнения программ

Для того, чтобы ввести понятие *символьное исполнение* рассмотрим базовые принципы процесса обычного исполнения программ и простые модели, в рамках которых можно описать данный процесс.

Исполнение программ обычно производится в рамках некоторой среды, обеспечивающей управление ресурсами, взаимодействие с физическими устройствами, производящими непосредственные вычисления, взаимодействие с пользователем, аппаратными средствами и другими программами. Для обработки внешних данных программе необходимо считать их из внешнего источника посредством системных вызовов, разместить полученные данные используя предоставленные средой ресурсы, произвести обработку данных и, при необходимости, записать результат обработки во внешний объект посредством системного вызова.

Приведём пример, иллюстрирующий рассмотренные выше объекты и принципы: операционная система является средой исполнения, выделяющей отдельным программам ресурсы виртуальной памяти. Для работы с внешним объектом — файлом, программа осуществляет запросы с помощью системных вызовов, позволяющие считать данные из файла в память и записать данные из памяти в файл.

Рассмотрим процесс исполнения программы на самом базовом уровне. Исполнение программы определяется непосредственно кодом программы и на самом общем уровне может быть описано в виде следующего процесса:

1. Инициализация ресурсов программы и выбор первой инструкции для начала работы (производится средой выполнения). Переход на шаг 2.
2. Разбор текущей инструкции и выполнение следующих действий:

- Осуществление запроса ко внешнему источнику, если это входит в функциональность инструкции.
- Изменение внутреннего состояния программы, к которому относится состояние всех элементов выделенных ресурсов.
- Переход на шаг 3.

3. Определение следующей инструкции для обработки на основе текущего состояния программы и параметров разобранной инструкции:
   - Если определение прошло успешно, переход на шаг 2.
   - В противном случае производится завершение исполнения программы.

Отметим, что подобная модель исполнения программы является в значительной степени упрощённой и не учитывает большое количество факторов и возможностей; тем не менее, рассмотрение этой модели позволяет осуществить простой переход к описанию процесса символьного исполнения.

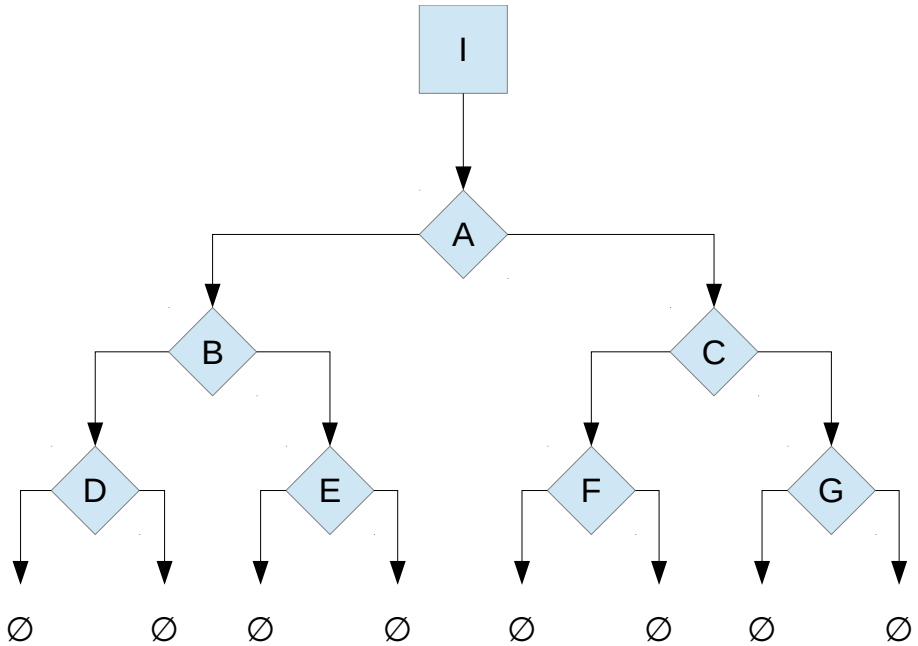
Для каждого отдельного запуска программы набор инструкций, начинающийся с инструкции, выбранной на первом шаге и заканчивающийся инструкцией, обработка которой привела к завершению программы на шаге 3, можно обозначить как **путь выполнения** программы.

Отметим дополнительно, что среди инструкций, из которых может быть составлена программа (например, набор инструкций процессорной архитектуры), можно выделить отдельную группу **условных** инструкций, отличающуюся поведением на шаге 3 процесса выполнения программ. Большинство инструкций программы являются линейными — результат шага 3 при обработке этих инструкций является фиксированным для любого состояния программы. Условные инструкции задают ряд возможных вариантов выполнения шага 3 в зависимости от значений элементов внутреннего состояния. Условные инструкции при выполнении программы задают **точки ветвления**.

В предположении, что у программы есть одна точка входа (то есть все пути выполнения начинаются с одной и той же инструкции), можно рассмотреть

множество всех возможных путей выполнения. Если бы в программе не было условных инструкций, то у неё был бы только один возможный путь выполнения — действительно, шаг 3 для каждой инструкции выполняется фиксированным образом, что однозначно задаёт вторую инструкцию пути выполнения по первой, третью — по второй и так далее.

Так как именно условные инструкции являются точками ветвления на шаге 3, то именно благодаря им и формируется множество путей выполнения. Подавляющее большинство условных инструкций описывают два возможных результата шага 3 (например, инструкции условного перехода основных процессорных архитектур). Для описания механизмов символьного исполнения будем рассматривать точки ветвления, соответствующие именно таким условных инструкциям. При данном рассмотрении множество путей выполнения программы можно представить **деревом путей выполнения** (рис. 1).

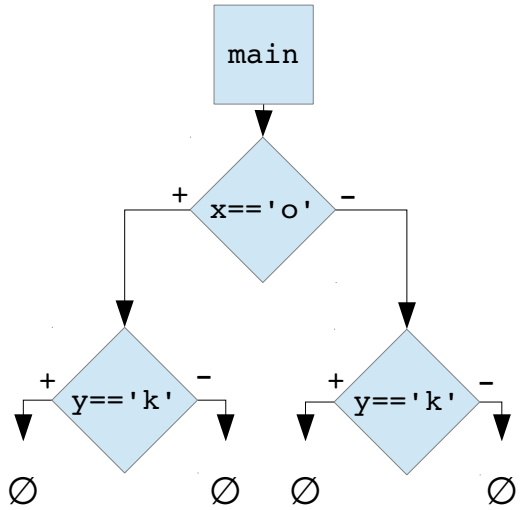


I — точка входа
A,B,...,G — точки ветвления
IABD, IABE, IACF, IACG — пути выполнения

Рисунок 1: Дерево путей выполнения

В общем случае точки ветвления A,B,...,G могут соответствовать различным инструкциям программы, однако часто соответствуют одним и тем же инструкциям программы. Приведём несколько примеров деревьев путей выполнения в рассмотренном выше графическом представлении на примере простых программ на языке C (рис. 2).

```
int main()
{
     int x, y, i = 0;
     //ввод x, y пользователем
     y = getc();
     x = getc();
     if (x == 'o')
          i ++;
     if (y == 'k')
          I ++;
     return i;
}
```




```
int main()
{
     int z, x, y, i = 0;
     //ввод x, y пользователем
     y = getc();
     x = getc();
     z = getc();
     if (x == 'o')
          if (y == 'k') {
               i ++;
               return i;
          }
     if (z == '!')
          i ++;
     return i;
}
```

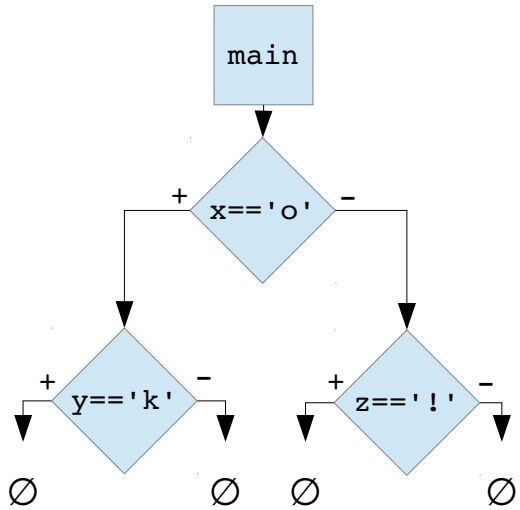


Рисунок 2: Деревья путей выполнения для модельных программ

В рамках данной работы будем рассматривать детерминированные программы. Детерминированной программой будем называть такую программу, выполнение которой зависит только от значений входных данных. Это означает,

что для любых двух запусков программы на одном наборе входных данных пути выполнения каждого запуска будут совпадать.

Ключевой особенностью обычного исполнения программы является то, что каждый отдельный запуск программы порождает единственный путь выполнения, так как получение конкретных значений из внешних источников устраняет неопределённость при вычислении результатов условных инструкций. Данная особенность иллюстрирует основную проблему различных методов динамического анализа программ — для того, чтобы получить достоверные данные о работе *всей* программы, необходимо произвести множество запусков программы на *различных* наборах входных данных.

1.1.2 Базовые принципы символьного исполнения программ

Символьное исполнение предлагает альтернативный метод исполнения программы, в рамках которого производится абстракция от конкретных значений. Элементы внутреннего состояния программы заменяются **символьными переменными**, описывающими все возможные значения элемента. Дополнительно символьными переменными заменяются все данные, считываемые из внешних источников. Вместо реального выполнения инструкций программы, происходит составление функционального ограничения (обычно в виде булевой формулы), описывающего связь между символьными переменными, соответствующими аргументам инструкции, и символьными переменными, соответствующими элементам, в которые помещается результат инструкции. Ограничения, соответствующие пути выполнения объединяются в **трассы ограничений**. Для иллюстрации отличий обычного исполнения (рис. 3) от символьного используем модельный фрагмент кода на языке C и будем рассматривать в качестве элементов состояния программы конструкции данного языка.

```
Исходный код          Обычное исполнение
1: int x, y, z;       1:
2: x = 5;             2: записать в переменную x число 5
3: y = 10;            2: записать в переменную y число 10
4: z = x + y;         4: извлечь значения переменных x и y
                         сложить значения переменных x и y
                         записать в переменную z число 15 (сумму)
```

Рисунок 3: Обычное исполнение модельного фрагмента кода

Как было замечено ранее, при обычном исполнении производится работа над конкретными значениями. Отметим, что после выполнения данного фрагмента значением переменной z будет именно число 15, а не формула, задающая сумму значений переменных x и y.

```
Исходный код       Символьное исполнение
1: int x, y, z;    1: завести символьные переменные sym_x, sym_y,
                   sym_z над доменом всех значений типа int
2: x = 5;          2: составить ограничение (sym_x == 5)
3: y = 10;         3: составить ограничение (sym_y == 10)
4: z = x + y;      4: составить ограничение (sym_z == sym_x + sym_y)
```

Рисунок 4: Символьное исполнение модельного фрагмента кода

При символьном исполнении (рис. 4) ситуация является обратной — значение суммы не определено, однако известно, что переменная z связана с переменными x и y соответствующей функциональной зависимостью.

Более важные отличия символьного исполнения от обычного проявляются при рассмотрении условных инструкций. Действительно, при обычном исполнении в рамках одного пути выполнения результат условной инструкции может быть вычислен, поскольку значения входных данных известны. При символьном исполнении конкретные значения не рассматриваются, поэтому вычислить точный результат условной инструкции напрямую нельзя. В рамках символьного исполнения для каждой точки ветвления необходимо рассмотреть все различные пути выполнения, порождаемые условной инструкцией в данной точке. Для иллюстрации данного эффекта расширим модельный пример, рассмотренный ранее (рис 5.).

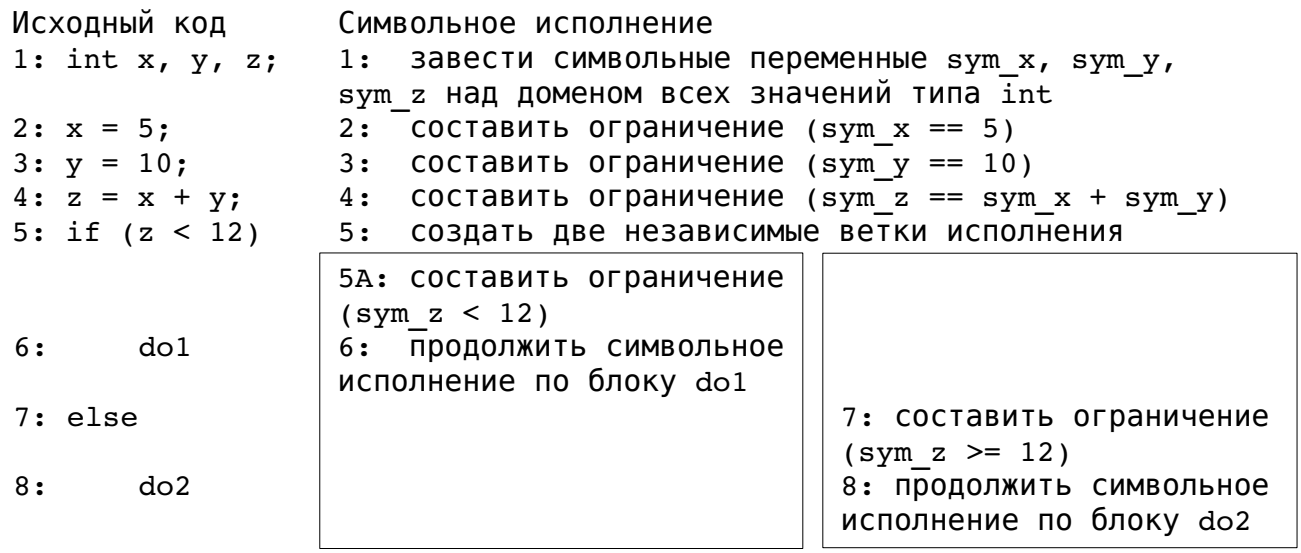


Рисунок 5: Символьное исполнение модельного фрагмента кода с точкой ветвления

Легко заметить, что несмотря на работу с символьными переменными, а не конкретными значениями, часть путей выполнения можно отбросить. Действительно, путь выполнения, соответствующий левой ветке и блоку `do1` является **несовместным** по входным данным, так как при ограничениях, наложенных на переменные `sym_x`, `sym_y` и `sym_z`, формула `(sym_z < 12)` всегда принимает значение *ложь*. Обработка данного пути является бессмысленной, так как программа не может по нему выполняться. Проведение подобной проверки может быть осуществлено автоматически с помощью инструментов, решающих задачу выполнимости булевых формул.

Рассмотрим влияние внешних источников данных на символьное исполнение на основе ещё одного модельного фрагмента кода (рис. 6).

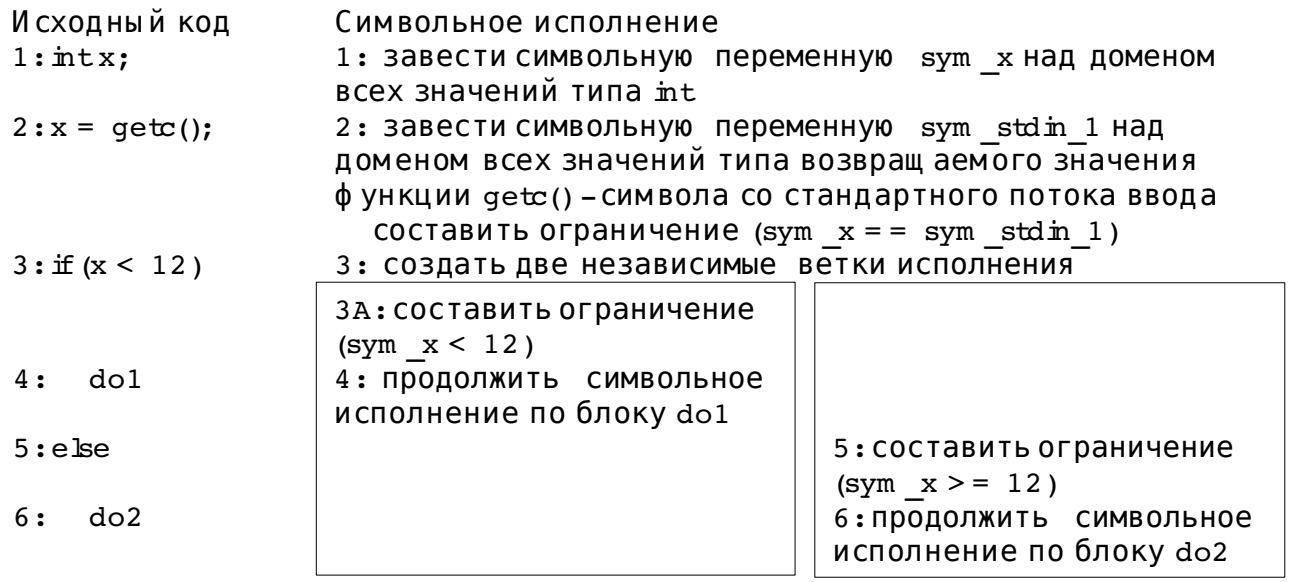


Рисунок 6: Символьное исполнение модельного фрагмента кода, обрабатывающего внешние данные

В данном случае оба пути выполнения являются совместными, так как со стандартного потока могут быть считаны символы, удовлетворяющие как условию на строке 3, так и отрицанию данного условия. Отметим одну из наиболее важных особенностей символьного исполнения — символьные переменные, соответствующие данным, считываемых извне, являются **свободными** в создаваемых трассах ограничений. Все остальные переменные имеют ограничения, связывающие их с конкретными значениями (в случае данного примера переменная `sym_x` ограничивается условной инструкцией). Нахождение значений данных, считываемых из внешних источников, удовлетворяющих трассе ограничений пути, и запуск программы на этом наборе значений позволит обойти путь выполнения. Это позволяет добиться следующего эффекта — символьное исполнение программ порождает наборы внешних данных, соответствующие различным путям выполнения.

1.1.3 Автоматический обход путей выполнения

На основе рассмотренных выше особенностей символьного исполнения опишем метод автоматического анализа программ, включающий построение наборов входных данных для обхода возможных путей выполнения. Этот метод

основывается на следующих принципах:

1. Разрабатывается модель символьной обработки инструкций на основе особенностей анализируемых программ и среды их исполнения. Данная модель описывает:
    - отображение элементов внутреннего состояния программы и среды исполнения в символьные переменные;
    - описание эффектов инструкций, поддерживаемых средой исполнения, в терминах ограничений, накладываемых на символьные переменные;
    - описание отображения интерфейсов доступа ко внешним источникам в символьные переменные;
2. Производится запуск программы на символьное исполнение. Среда символьного исполнения интегрируется с инструментом проверки выполнимости булевых формул для отбрасывания несовместных путей и поиска значений входных данных.
3. Среда символьного исполнения обрабатывает программу, проверяя потенциальные пути выполнения в точках ветвления, отбрасывая несовместные и создавая реальные наборы входных данных для совместных.
4. Символьное исполнение производится пока не пройдены все возможные пути выполнения или пока не достигнуто некоторое условие остановки анализа.

Итогом работы системы, реализующей подобный метод, является анализ путей выполнения программы при символьном исполнении и накопление множества наборов входных данных для использования другими средствами динамического анализа.

1.1.4 Совмещение обычного и символьного исполнения программ

Рассмотрим модель исполнения программы, основывающуюся на совмещении символьного исполнения с обычным. Применение данной модели позволяет в значительной степени снизить затраты на обход путей выполнения и

работу инструментов, осуществляющих проверку выполнимости булевых формул.

В рамках данной модели исполнение инструкции может производится символьно, обычным образом или в режиме сопряжения. Как правило, методы совмещенного исполнения основываются либо на механизме точного переключения между символьным и обычным исполнением, либо на выполнении **всех** инструкции в режиме сопряжения. Оптимизация анализа для обоих механизмов достигается за счёт уменьшения числа символьных переменных и числа точек ветвления, в которых происходит проверка выполнимости и создание новых наборов входных данных. Рассмотрим особенности совмещения символьного и обычного исполнения на модельном примере при использовании механизма точного переключения (рис. 7).

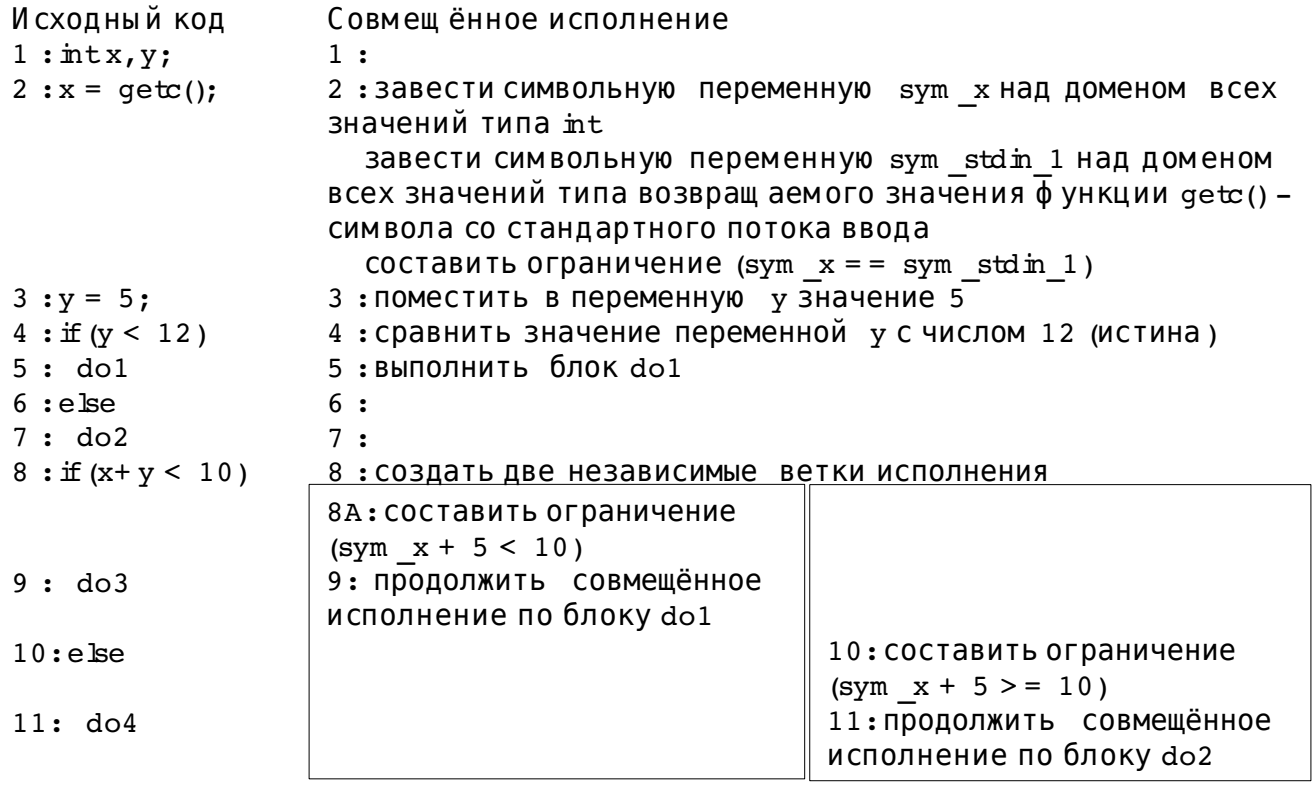


Рисунок 7: Совмещённое символьное и обычное исполнение модельного фрагмента кода

В рамках данного примера сопряжённое исполнение позволяет избежать символьной обработки переменной `y` (при обработке строк 3,8), не зависящей от входных данных, и создания дополнительных путей выполнения для условия, проверяющего значение `y` (при обработке строки 4).

1.2 Обзор инструментов динамического анализа на основе символьного исполнения

Термин *символьное исполнение* применительно к выполнению программного кода появился в середине 1970-х годов и был развит исследователями, имевшими значительный опыт в вопросах формальной верификации программного обеспечения, в ряде работ, среди которых можно отметить [5,6]. Уже для ранних реализаций метода символьного исполнения была актуальной проблема масштабируемости, возникающая из-за отсутствия эффективных алгоритмов решения задачи проверки выполнимости набора булевых ограничений. Это накладывало серьёзные ограничения на применение методов символьного исполнения в промышленных целях.

Различные исследователи соглашаются [7,8], что современная популярность метода анализа программ на основе символьного исполнения вызвана успехами в области решателей булевых формул и повышением производительности вычислительных устройств.

В середине 2000-х годов появился целый ряд работ, описывающих схемы анализа и архитектуру инструментов реализующих данные схемы. Схемы, предлагаемые в данных работах, различаются по нескольким характеристикам, из которых можно выделить наиболее значимые:

- организация порядка обхода путей выполнения программы;
- используемая среда исполнения программы;

Данные характеристики определяют основные особенности работы инструмента и его область применимости. Дополнительные характеристики, такие как оптимизации отдельных модулей анализа, применяемые стратегии выбора ветвлений и работы с наборами булевых формул, обычно определяют общую скорость работы инструмента.

1.2.1 Классификация по порядку обхода путей выполнения

Можно выделить две основные схемы контроля за исполнением программы и работы с точками ветвления в программе: режим **online** и режим **offline**.

В рамках режима **online** инструмент анализа осуществляет полный контроль за выполнением целевой программы. Производится единственный запуск программы на выполнение с начальным набором входных данных. При достижении точки ветвления, учитывающей символьные данные, текущее состояние программы сохраняется в некоторую внутреннюю структуру данных, а трасса ограничений дублируется с точностью до результата проверки условия в точки ветвления. Для обоих трасс осуществляется проверка выполнимости и построение набора входных данных (если проверка была успешной). Далее, инструмент анализа выбирает одно из направлений ветвления и продолжает выполнение программы и анализ в выбранном направлении. В определённый момент времени (например, при завершении анализа выполнение в выбранном ранее направлении или по результатам модуля балансировки выбора путей) осуществляется откат к состоянию в точке ветвления и анализ производится уже по другому направлению ветвления.

Данная схема имеет ряд преимуществ, приводящих к высокой скорости обхода путей выполнения. В первую очередь, за счёт сохранения состояния программы в памяти и отката к сохранённым состояниям отсутствует необходимость заново выполнять код программы до точки ветвления. Согласно [9], затраты на повторное выполнение данного кода выше, чем затраты на восстановление состояния программы при откате. Во-вторых, поддержка и откат состояний эффективно сочетаются с использованием решателей булевых ограничений, поддерживающих пошаговое накопление формул (incremental solving [10]).

В то же время, хранение состояний программы во время анализа требует значительных ресурсов памяти и наличия механизма контроля, обеспечивающего поддержание высокой скорости работы на всём протяжении анализа. Схемы инструментов, осуществляющих анализ в режиме online, являются менее гибкими из-за повышенного уровня зависимости по данным между модулями анализа.

В рамках режима **offline** инструменты анализа производят многократные

запуски целевой программы на выполнение от точки входа на создаваемых наборах входных данных. При выполнении программы производится отслеживание потока данных и построение трасс булевых ограничений, однако обработка трасс осуществляется после завершения выполнения программы. При таком подходе поддержка множества состояний программы в точках ветвления и откат к данным состояниям является необязательными - в данной роли выступают наборы входных данных, снабжённые дополнительной информацией для поддержания уникальности соответствующих им путей выполнения программы.

В противоположность режиму online режим offline требует ограниченного объёма ресурсов памяти (фактически, вместо описания состояния программы необходимо хранить значения входных данных), однако повторное выполнение исполняемого кода приводит к замедлению обхода путей выполнения программы.

1.2.2 Классификация по схеме взаимодействия со средой выполнения

Проведение всех необходимых действий по составлению трассы ограничений, сохранению и восстановлению состояния программы напрямую привязано к выполнению отдельных инструкций программы. В связи с этим можно предложить два возможных варианта реализации системы, осуществляющие данные действия:

1. Модификация среды исполнения, включающая расширение функциональности обработчиков отдельных инструкций.
2. **Инструментация** — модификация кода программы, нацеленная на внедрение дополнительных инструкций, извлекающих нужные данные во время выполнения программы.

Первый вариант обеспечивает:

- минимальное влияние на внутреннее состояние программы — ресурсы среды исполнения и ресурсы обрабатываемой программы разделены;
- возможность обработки внутреннего состояния программы встроенными функциями среды исполнения — например, сохранение текущего состояния программы при проведении анализа в режиме online;

- фиксированные накладные расходы на выполнение дополнительных действий при обработке инструкций.

Второй вариант обеспечивает:

- прямое влияние на внутреннее состояние программы — дополнительный код разделяет ресурсы программы;
- возможности учёта специфики конкретной программы при проведении модификации с целью оптимизации последующего анализа;
- лёгкость интеграции с разными средами исполнения.

Инструментация может проводиться в двух режимах:

- статическая инструментация, проводимая однократно перед непосредственным выполнением программы, обеспечивает:
  - низкие накладные расходы на повторное выполнение кода;
  - ограниченное покрытие кода;
- динамическая инструментация, проводимая непосредственно во время выполнения программы, обеспечивает:
  - высокие накладные расходы;
  - полное покрытие кода;

При проведении инструментации также актуально представление кода, с которым осуществляется работа:

- исходный код программы, обработка которого имеет следующие особенности:
  - позволяет проводить только статическую инструментацию;
  - позволяет формировать модификации кода малыми затратами — основные изменения производятся системами компиляции и компоновки.
  - инструментация чувствительна к оптимизациям, используемым при компиляции и сборке;
- исполняемый код программы в инструкциях целевой процессорной

архитектуры, обработка которого имеет следующие особенности:

- точная привязка к выполнению кода обеспечивает высокую точность;
- наборы инструкций процессора сильно варьируются, что задаёт высокую сложность непосредственной модификации;

- исполняемый код программы в специализированном промежуточном представлении, имеющий следующие особенности:
  - более низкоуровневый, чем исходный код, - больше точность модификации;
  - более высокоуровневый, чем исполняемый код в инструкциях целевой процессорной архитектуры, - проще проведение модификаций;
  - требует использования специализированной среды исполнения.

1.2.3 Обзор существующих инструментов анализа

Современные инструменты автоматического обхода путей выполнения активно используются как для теоретических исследований задач динамического анализа программ, так и непосредственно при промышленной разработке свободного и проприетарного программного обеспечения. В данном разделе рассматриваются инструменты, которые внесли наиболее значимый теоретический и практический вклад в развитие области, а также приведён краткий обзор последних разработок, иллюстрирующих широкое распространение базового подхода среди различных программных и аппаратных платформ.

Подход, основывающийся на сопряжении символьного исполнения программы с непосредственным исполнением программы, был впервые предложен авторами системы DART [11]. Инструмент DART (режим offline, инструментация) основывается на проведении статического анализа исходного кода целевых программ с целью выявления источников внешних данных, определяющих дерево путей выполнения программы. В качестве подобных источников рассматриваются внешние переменные и функции, определённые в

подключаемых в процессе выполнения библиотеках. На основе полученной информации осуществляется автоматическое создание исходного кода модуля тестирования, включающегося в состав исследуемой программы. Модуль тестирования производит повторное выполнение следующего набора действий:

- инициализация внешних для целевой программы переменных и подмена внешних функций на функции, возвращающие некоторые конкретные значения
- передача управления в точку входу целевой программы

Конкретные значения внешних переменных и возвращаемые значения внешних функций определяются согласно символьным ограничениям для обхода путей выполнения. Для извлечения символьных ограничений используется статическая инструментация исходного кода.

В рамках работ по созданию систем CUTE и jCUTE (режим offline, инструментация) [12,13] подход, предложенный авторами инструмента DART, был расширен и формализован для более общего множества целевых программ. Данные системы предлагали поддержку автоматического обхода путей выполнения для программ на языках C и Java соответственно. Дополнительно, был введён принцип символьных зависимостей по указателю, не рассматриваемый авторами инструмента DART. В рамках данного принципа операции работы с памятью рассматривали в качестве символьных переменных не только содержимое ячеек памяти, но и непосредственно адреса памяти, что порождало построение дополнительных ограничений при выполнении программы. Это, в свою очередь, позволяло повысить точность проверки выполнимости ограничений для построения наборов входных данных. Авторы инструмента CUTE также предложили ряд оптимизаций наборов ограничений, позволяющих сократить время, требуемое на проверку их выполнимости; некоторые из данных оптимизаций широко используются и в более современных инструментах автоматического обхода путей выполнения. Реализация инструмента CUTE была основана на статической инструментации исходного

кода программы на языке C, в то время как инструмент jCUTE был предназначен для исследования программ на языке Java.

Инструмент KLEE (режим online, виртуальная машина) [14] предоставляет возможности анализа исполняемого кода программ во внутреннем представлении низкоуровневой виртуальной машины LLVM [15]. Инструмент KLEE реализует стандартный режим online автоматического обхода путей выполнения на основе точной трассировки состояния регистров и памяти виртуальной машины при выполнении программы. Для поддержания состояний программы используются встроенные возможности виртуальной машины LLVM, дополнительно осуществляется сохранение значений символьных переменных. В инструменте KLEE присутствует возможность ручной разметки источников входных данных в программном коде и возможность автоматической обработки файлов, аргументов командной строки и переменных окружения. Авторами реализованы дополнительные оптимизации трасс ограничений и алгоритмы выбора приоритетных путей выполнения для анализа, нацеленные на максимизацию покрытия исследуемого кода. Дополнительно стоит отметить более ранние работы авторов в области, результатами которых стали инструменты EXE [16] и STP [17]. Инструмент EXE был технологическим предшественником KLEE, нацеленным на работу с исполняемым кодом на базе процессорной архитектуры x86 без использования виртуальной машины LLVM. Инструмент STP представляет собой решатель булевых ограничений, использующий ряд оптимизаций для наборов ограничений, получаемых в процессе работы метода автоматического обхода путей выполнения.

В настоящее время инструмент KLEE активно развивается и распространяется свободно. Тем не менее, необходимость наличия исполняемого кода программы в бит-коде виртуальной машины LLVM является существенным ограничением данного инструмента. Существуют подходы, позволяющие автоматически переводить исполняемый код в инструкциях целевой платформы в бит-код LLVM, однако их применение может влиять на точность (для статических

методов трансформации кода [18]) и скорость (для динамических методов трансформации кода [19]) анализа.

Инструмент SAGE (режим offline, инструментация) [20], созданный для внутреннего использования в компании Microsoft, предоставляет возможности анализа программ в режиме offline. Для сбора данных о выполнении программы в инструменте SAGE используется система трассировки программ iDNA [21], позволяющая воссоздавать полный набор инструкций, составляющих путь выполнения. Запуск программы на выполнение производит подобную трассу, которая уже в свою очередь обрабатывается дополнительными модулями. Подобная архитектура системы позволяет снизить накладные затраты анализа в случае использования дополнительных модулей, нацеленных на исследование процесса выполнения программы.

Инструмент SAGE активно используется для тестирования стандартных модулей и программ операционной системы Windows. При проведении подобного тестирования активно применяются методы распределения и распараллеливания анализа для эффективного использования доступных вычислительных кластеров. Приведённая в [22] статистика включает в себя результаты более чем, 300 машинных лет выполнения анализа с помощью инструмента SAGE. Инструмент SAGE является проприетарной разработкой и не распространяется на коммерческой основе.

Инструмент PEX [23] реализует идеи применения символьного исполнения и автоматического обхода путей для программ на языке C#. Данный инструмент был интегрирован в среду разработки Visual Studio и предназначен для анализа отдельных фрагментов кода, обычно соответствующих модульным тестам, для которых входными данными являются параметры функций. Подобный подход позволяет сократить количество различных тестов, нацеленных на проверку одного множества условий корректности, - граничные значения входных данных будут построены автоматически.

Платформа BitBlaze (режим online, эмулятор) [24] предоставляет широкие

возможности проведения анализа исполняемого кода, включая также модули символьного исполнения и интеграции с решателями булевых ограничений. Платформа включает в себя три основных компонента - среда статического анализа исполняемого кода Vine, среда исполнения и динамического анализа кода TEMU и модуль управления взаимодействием между компонентами и механизмами анализа символьных данных Rudder. Каждый из трёх компонентов предоставляет программный интерфейс для реализации инструмента анализа, решающего необходимую задачу.

Среда Vine осуществляет автоматический перевод исполняемого кода во внутреннее представление и построение таких структурных элементов, как граф потока управления, граф потока данных и т.д. Среда TEMU (на базе эмулятора QEMU) осуществляет выполнение инструкций исполняемого кода и всех дополнительных действий по извлечению данных и отслеживанию операций над символьными данными согласно спецификациям инструмента анализа. Модуль Rudder осуществляет контроль за процессом анализа, взаимодействием с решателями булевых ограничений, механизмами выбора путей для анализа.

На основе платформы BitBlaze были реализованы инструменты анализа программ с целью выявления потенциальных уязвимостей, фрагментов кода, осуществляющих несанкционированные действия над пользовательскими данными и средой выполнения, восстановления алгоритмов и протоколов и т.д. Работа в рамках полносистемного эмулятора TEMU позволяет проводить исследование взаимодействия отдельных процессов и учёт действий, производимых операционной системой в привилегированном режиме. В то же время, использование полносистемной эмуляции приводит к высоким накладным расходам и делает затруднительным взаимодействие с реальными устройствами целевой архитектуры или использование вычислительных мощностей параллельных и распределённых систем.

Инструмент Avalanche (режим offline, инструментация) [25] предоставляет возможности проведения анализа в режиме offline. Помимо стандартного обхода

путей выполнения в инструменте происходит целенаправленный поиск критических дефектов в программе путём внедрения дополнительных ограничений в трассу ограничений. Для каждой потенциально опасной операции (например, доступ к содержимому динамической памяти или операции деления) осуществляется попытка построения входных данных, приводящих к использованию некорректных значений операндов (нулевой адрес ячейки памяти и нулевой делитель соответственно).

В инструменте Avalanche используется динамическая инструментация на основе системы Valgrind и решатель STP. Также присутствуют модули удалённого выполнения, обеспечивающие возможность использования нескольких вычислительных узлов, ответственных за разные этапы анализа (выполнение программы, разрешение булевых ограничений).

Инструмент S2E [19] позволяет проводить автоматический обход путей выполнения целевой программы совмещая два инструмента — эмулятор QEMU [26], осуществляющий перевод инструкций целевой архитектуры в бит-код виртуальной машины LLVM во время выполнения программы, и инструмент KLEE, рассмотренный ранее. Подобный подход позволяет снять ограничение на наличие исполняемого кода программы в формате бит-кода LLVM и предоставляет возможности полносистемного анализа. Дополнительно в инструменте S2E присутствуют возможности добавления пользовательских модулей для определения стратегии обхода путей выполнения и обработки трасс инструкций с целью обнаружения различных классов дефектов (например, обнаружения ситуаций некорректной работы с динамической памятью).

Авторы инструмента предложили несколько моделей совмещения символьного и реального исполнения программы; в рамках проведения анализа S2E возможен выбор одной из этих моделей. Данные модели определяют степень строгости соответствия символьных переменных и конкретных значений в ячейках памяти и регистрах при выполнении программы. Использование различных моделей позволяет варьировать масштаб исполняемого кода, для

которого производится ветвление и, соответственно, масштаб необходимых вычислений и обрабатываемых путей выполнения, в зависимости от целей анализа, задаваемых пользователем.

Инструмент Mayhem (режим offline/online, инструментация) [9] рассматривает возможность проведения гибридного offline- и online-анализа путём динамического контроля доступных ресурсов. При наличии достаточного количества ресурсов осуществляется запись состояний программы в точках ветвления; в условиях ограниченности ресурсов анализ продолжается без увеличения количества сохранённых состояний в режиме offline. В отличие от других инструментов, реализующих режим online, инструмент Mayhem не использует виртуальные машины и эмуляторы для работы с состояниями программы (в Mayhem используется динамическая инструментация кода на основе системы Pin [27]). Откат к некоторому состоянию осуществляется путём исполнения программы с соответствующим набором входных данных без проведения обработки символьных переменных. При достижении целевой точки ветвления осуществляется восстановление информации о символьных переменных из сохранённого ранее состояния и включается основной механизм анализа. Дополнительно в составе инструмента Mayhem действуют модули, позволяющие обнаруживать не только дефекты, приводящие к аварийному завершению программы, но и потенциальные уязвимости безопасности; для каждой подтверждённой уязвимости проводится автоматическое построение кода для её эксплуатации.

На основе системы Mayhem был проведён наиболее масштабный анализ открытого кода - более 10 тысяч отдельных программ из операционной среды Debian - который выявил более тысячи различных дефектов. При этом инструмент Mayhem является проприетарным.

1.3 Направления оптимизации итеративного динамического анализа

При разработке и реализации схем автоматического обхода путей выполнения на основе символьного исполнения требуется учитывать значительное количество особенностей задачи анализа программ. Практически все актуальные проблемы данной задачи сводятся к сложности исследуемых программ и объёму работы, которую необходимо произвести. Сложность программ, выраженная через количество различных путей выполнения, растёт экспоненциально с увеличением объёма кода (так называемая проблема “экспоненциального взрыва”). Подобный рост усугубляется увеличением объёмов обрабатываемых данных, применением специализированных схем передачи и обработки данных (например, схемы “клиент-сервер”), использованием параллельных потоков. Ключевая для метода задача проверки выполнимости булевых формул не имеет полиномиальных алгоритмов решения [28].

Сформулируем следующие основные группы проблем:

A) экспоненциальный рост количества путей при увеличении сложности исследуемого программного обеспечения;

B) отсутствие полиномиальных алгоритмов решения задачи выполнимости булевых ограничений, являющегося ключевым этапом при построении входных данных;

C) накладные расходы на сбор информации о символьных переменных и создание трассы ограничений;

D) наличие особенностей исследуемых программ, понижающих точность работы с символьными переменными и корректность создаваемых зависимостей выполнения от входных данных.

Указанные проблемы приводят к невозможности проведения абсолютно полного анализа программ высокой сложности даже при наличии больших объёмов вычислительных и достаточного количества времени. В свою очередь, становится актуальным исследование методов оптимизации отдельных

компонентов систем анализа и сокращения или трансформации потоков обрабатываемых данных без значительной потери качества.

Среди наиболее перспективных направлений исследований в данной области можно выделить следующие:

1. сокращение размера обрабатываемого дерева путей выполнения программы (решение проблемы **A**);
2. трансформация трасс булевых ограничений и использование дополнительной информации при проверке выполнимости (решение проблемы **B**);
3. управление порядком выбора путей выполнения программ (решение проблем **A**, **B**, **D**);
4. расширение и уточнение модели символьного исполнения (решение проблем **A**, **D**);
5. снижение накладных расходов на создание трасс ограничений (решение проблемы **C**);
6. повышение эффективности использования предоставляемых ресурсов (решение проблем **A**,**B**,**C**);

1.3.1 Сокращение числа путей выполнения

Данный подход позволяет напрямую снизить вычислительную сложность обхода путей выполнения и уменьшить время, необходимое для анализа программы. Часто данный эффект достигается за счёт снижения общей полноты анализа.

В работах [29,30] рассматривается возможность более эффективного анализа программ, входные данные которых описываются формальными грамматиками; метод позволяет уменьшить число исследуемых путей выполнения и рассматривать в трассах ограничений соответствие символьных переменных элементам грамматики, ускоряя проверку выполнимости данных трасс.

В работе [31] предлагается возможность оценки различий состояния программы в разных точках ветвления. На основе метода некоторые пути

выполнения считаются эквивалентными, если они приводят к эквивалентным состояниям программы — обход всех путей одной группы эквивалентности объявляется излишним.

В работах [32,33] предлагается возможность создания описаний отдельных фрагментов кода; при проведении символьного исполнения обход путей по точкам ветвления внутри данных фрагментов не производится, а их описание напрямую переводится во фрагмент трассы ограничений, что позволяет избежать излишних издержек по обработке инструкций программы для формирования трассы ограничений.

1.3.2 Трансформация трасс ограничений

Проведение оптимизаций трасс ограничений, не поддерживаемых инструментами проверки выполнимости, позволяет ускорить работу данных инструментов. Среди предлагаемых оптимизаций можно отметить как общие оптимизации наборов булевых формул, так и оптимизации, характерные для задачи анализа программ.

В работах [14,19,20] предлагается ряд методов оптимизации работы инструментов проверки выполнимости с помощью разбиения трасс ограничений на независимые подтрассы и кэширования результатов данных инструментов при проведении обхода путей выполнения.

1.3.3 Управление порядком выбора путей выполнения

Методы данной группы нацелены на повышение качества результатов, получаемых при проведении автоматического обхода путей выполнения программы за фиксированное время. Применение данных методов не приводит к уменьшению общего числа путей выполнения, которые необходимо проверить, а позволяет изменить порядков обхода путей выполнения программы согласно некоторым критериям, актуальным для пользователя инструментов анализа.

В работах [34,35] предлагается использовать граф потока управления программы и дополнительные структуры, основывающиеся на данном графе, для выделения путей выполнения, потенциально позволяющих покрыть большое

количество кода. Результатом применения данных методов является возможность построить актуальное покрытие кода по базовым блокам и точкам ветвления на основе обхода меньшего количества путей выполнения, т.е. организовать направленный обход наиболее перспективных путей выполнения.

В работе [36] предлагается интегрировать в систему автоматического обхода путей выполнения внешний модуль оценки качества путей выполнения. Рассматриваются несколько возможных алгоритмов, положенных в основу подобного модуля; наиболее интересным из них является алгоритм, основанный на предоставлении преимущества путям выполнения, включающим граничные значения условий в точках ветвления (например, для условия $x<5$ граничное значение x равняется 4, если переменная x принимает только целые значения). Задача проверки подобных значений при проведении тестирования программ является актуальной [37].

В работах [38,39] предлагается совмещение генетических алгоритмов построения наборов входных данных и методов автоматического обхода путей выполнения на основе символьного исполнения. Генетические алгоритмы позволяют с малыми затратами строить тестовые наборы для оценки различных фрагментов кода программы, но не позволяют подробно обходить пути выполнения внутри данных фрагментов. Символьное исполнение обрабатывает программу с определённой точки входа, что не позволяет за ограниченное время покрыть различные фрагменты, которые могут иметь разную глубину. Совмещение методов в рамках единой системы создания тестов позволяет снизить влияния данных недостатков.

1.3.4 Расширение и уточнение модели символьного исполнения

Методы данной группы нацелены на проведение символьного исполнения, включающее учёт косвенных связей между данными программы при создании трассы ограничений. Учёт данных связей позволяет отбросить часть путей выполнения на основе вывода об отсутствии практического смысла их проверки повышая тем самым эффективность анализа.

В работах [40,41,42] предлагаются модели символьного исполнения, которые учитывают связи между данными, участвующими в циклах программы. Данные методы позволяют избежать построения путей выполнения, которые отличаются только количеством итераций циклов при работе программы. Дополнительно, их применение позволяет получать пути выполнения для достижения фрагментов кода, требующих определённого количества итераций некоторого цикла в программе.

В работах [43,44] предлагаются методы повышения точности символьного исполнения программ, проводящих вычисления с плавающей точкой. Подобные вычисления в значительной степени повышают затраты на проверку выполнимости ограничений из-за сложности представления операций в виде булевых формул. Авторы работ предлагают более эффективные алгоритмы преобразования операций над значениями с плавающей точкой к операциям обработки целочисленных значений для представления их в трассе ограничений.

В работах [45,46,47] предлагаются модели символьного исполнения, учитывающие использование параллельных потоков в программах. Так как символьное исполнение базируется на построении трассы ограничений по ходу выполнения инструкций программ, параллельные потоки могут вносить неопределённость в данную трассу. Эта неопределённость может приводит к повторной обработке одних и тех же путей выполнения или, наоборот, к потере части путей выполнения.

1.3.5 Сокращение накладных расходов на создание трасс ограничений

Методы данной группы нацелены на снижение затрат на символьное исполнение инструкций.

В работах [24,33] предлагаются методы, основанные на проведении статического анализа исполняемого кода программы перед проведением анализа программ, с целью исключения фрагментов, обработку которых заведомо не нужно проводить символьно, и с целью выделения фрагментов, для которых можно создать предварительное описание. Вместо непосредственной обработки

инструкций данных фрагментов во время символьного исполнения программы в трассу ограничений добавляется целый блок, соответствующий функциональности фрагмента. Применение данного метода позволяет снизить затраты на повторную обработку данных фрагментов при исследовании различных путей выполнения, для которых эти фрагменты являются общими.

1.3.6 Повышение эффективности использования вычислительных ресурсов

Методы данной группы позволяют повысить эффективность анализа программ при использовании оборудования, предоставляющего поддержку параллельных и распределённых вычислений. Подробный обзор работ в данном направлении приведён в главе 3.

1.3.7 Предлагаемые в работе методы

В рамках данной работы предлагаются методы, которые можно отнести к группам 1, 5 и 6. Выбор данных направлений обуславливается их потенциальной областью применимости и стремлением повысить применимость инструментов автоматического обхода путей выполнения в области промышленной разработки программного обеспечения.

1.4 Инструмент Avalanche

В качестве базового инструмента для проведения исследований в рамках данной работы был выбран инструмент Avalanche.

1.4.1 Предпосылки выбора инструмента Avalanche

Выбор инструмента проводился исходя из указанных целей и задач данной работы для снижения затрат на исследование и реализацию предлагаемых методов и минимизации технических ограничений. В таблице 1 представлены особенности инструментов, которые активно развиваются разработчиками и предоставляют поддержку актуальных на данный момент версий операционных систем и процессорных архитектур. Рассматриваемые особенности являются ключевыми с точки зрения практического применения инструмента.

Таблица 1: Характеристики инструментов автоматического обхода путей выполнения программ

		Avalanche	BitBlaze	KLEE	Mayhem	SAGE	S2E
Проприетарный		нет	нет	нет	да	да	нет
Поддержка платформ		Linux: x86,x64, ARM	Linux: x86,x64, ARM	LLVM	Linux: x86, x64	Windows: x86, x64	Windows, Linux: x86,x64, ARM
Открытый исходный код		да	да	да	нет	нет	да
Работа с исполняемым кодом в инструкциях целевой архитектуры		да	да	нет	да	да	да
Механизм обработки выполнения программы: Э — эмуляция И — инструментация В — виртуальная машина		И	Э	В	И	И	Э
Поддержка удалённого выполнения		да	нет	нет	неизвестно	да	частичная
Поддержка источников входных данных	аргументы командной строки	да	да	да	да	да	да
	файлы	да	да	да	да	да	да
	сетевые соединения	да	нет	нет	да	неизвестно	да
	поток ввода	да	да	да	да	да	да
	ручная разметка исходного кода	нет	нет	да	нет	нет	да

Инструмент Avalanche распространяется свободно, может быть свободно модифицирован, не требует применения систем эмуляции и интерпретации и

позволяет осуществлять анализ приложений, выполнение которых производится на удалённом устройстве. Последняя особенность является в значительной степени значимой для исследования предлагаемого метода статической инструментации исполняемого кода согласно аргументам, рассмотренным во введении.

1.4.2 Схема работы инструмента Avalanche

Инструмент Avalanche реализует метод автоматического обхода путей выполнения на основе символьного исполнения, сопряжённого с обычным исполнением. Обход путей в рамках инструмента Avalanche производится в режиме offline.

Во время анализа для всех создаваемых наборов входных данных производятся запуски программы на выполнение, независимые от задач символьного исполнения. Данные запуски позволяют обнаруживать критические дефекты анализируемых программ, приводящие к аварийному завершению. Дополнительно данные запуски осуществляют оценку перспективности отдельных путей выполнения с целью скорейшего увеличения покрытия базовых блоков кода программы.

Инструмент состоит из четырёх компонентов, каждый из которых решает отдельную подзадачу анализа:

- модуль tracegrind, осуществляющий выполнение программы с целью создания трассы ограничений;
- модуль covgrind, осуществляющий проверку корректности работы программы на создаваемых наборах входных данных и оценку приоритетности путей выполнения с точки зрения прироста покрытия кода;
- модуль STP, осуществляющий проверку выполнимости наборов ограничений;
- управляющий модуль Driver, контролирующий обмен данными между остальными модулями и ответственный за сохранение результатов и взаимодействие с пользователем.

Схема взаимодействия компонентов представлена на рисунке 8.

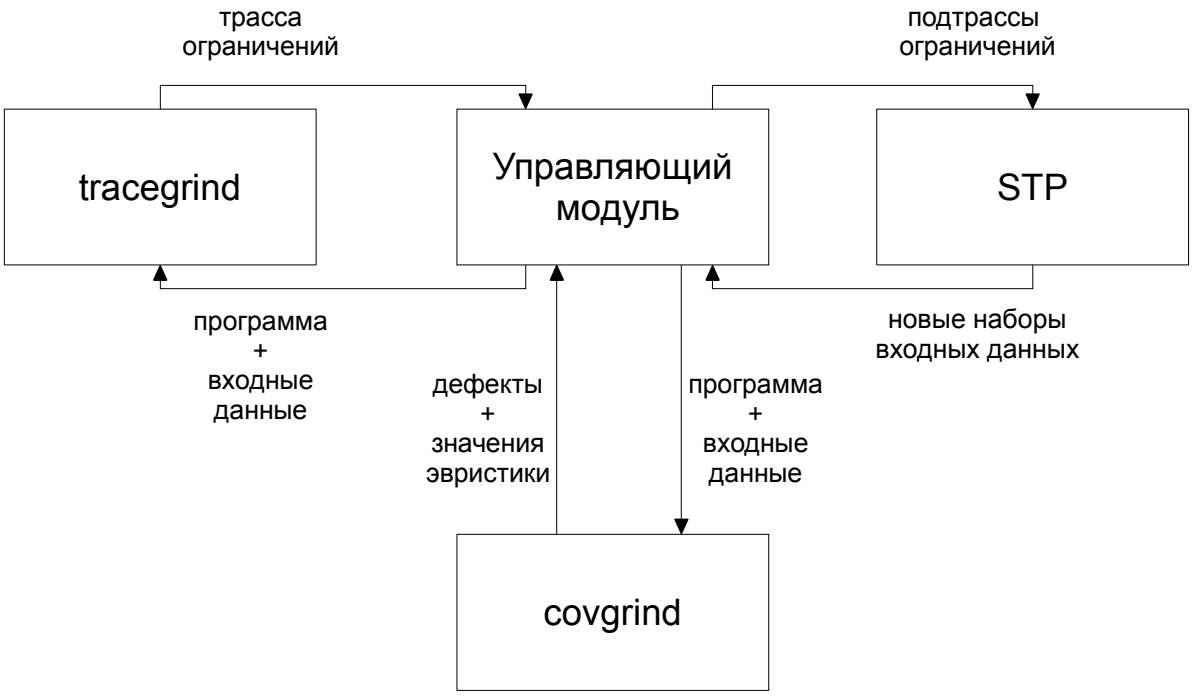


Рисунок 8: Схема взаимодействия компонентов инструмента Avalanche

Рассмотрим более подробно алгоритм работы инструмента Avalanche:

1. Выполнить запуск программы под контролем модуля covgrind на начальном наборе входных данных *InitialInput*; результатом работы модуля является множество адресов базовых блоков *BBSet*, выполненных на данном запуске.
2. Инициализировать рабочее множество *Inputs*, состоящее из троек <*Input*,*Score*,*Depth*>, элементом <*InitialInput*,|*BBSet*|,0>. Элемент *Score* используется в качестве метрики перспективности наборов входных данных. Элемент *Depth* задаёт глубину точки ветвления, породившей путь выполнения, в дереве путей выполнения. Пополнить множество *GlobalBBSet*, хранящее все базовые блоки программы, выполненные хотя бы на одном пути во время анализа, элементами *BBSet*.
3. Выбрать из множества *Inputs* элемент с наибольшим значением *Score*. Если таких элементов несколько, среди них выбрать элемент с наименьшим значением *Depth*. Зафиксировать значение *Depth* выбранного элемента как

CurrentDepth.

4. Осуществить запуск программы под контролем модуля tracegrind на текущем наборе <*Input*, *Score*, *Depth*> с зафиксированной максимальной глубиной просмотра точек ветвления *InvertDepth*. Результатом данной операции является трасса ограничений *PathTrace*, содержащая *N* точек ветвления, причём *N*<*InvertDepth*. Каждая точка ветвления имеет глубину $Depth_i$, равную порядковому номеру точки в трассе.
5. Разбить *PathTrace* на *N* подтрасс, каждая из которых оканчивается ограничением на результат условной операции в точке ветвления. Изменить данное ограничение на противоположное.
6. Для каждой подтрассы выполнить проверку выполнимости с помощью модуля STP; если трасса выполнима, результатом выполнения будет являться новый набор входных данных.
7. Для каждого полученного набора входных данных *NewInput* выполнить запуск программы под контролем модуля covgrind (результатом является множество *NewBBSet*). Добавить в множество *Inputs* элемент <*NewInput*,| *NewBBSet**GlobalBBSet*|, *CurrentDepth*+$Depth_i$>, где $Depth_i$ соответствует глубине точки ветвления, породившей набор входных данных. Пополнить множество *GlobalBBSet* элементами *NewBBSet*.
8. Осуществить переход на шаг 4, пока множество *Inputs* не пусто или пока не закончилось время, выделенное на анализ.

Работа модуля tracegrind на рабочем наборе <*Input*, *Score, Depth*> с максимальной глубиной просмотра точек ветвления *InvertDepth* осуществляется следующим образом:

1. Инициализировать отображения *IsTainted*(*Addr*): Memory → Boolean, *IsTainted*(*Reg*): Registers → Boolean. Изначально *IsTainted*(*Arg*)=False для любого аргумента. Инициализировать счётчик глубины *CurDepth* значением 0.
2. Запустить программу и для каждой инструкции пути выполнения совершить

следующие действия:

1. Если инструкция осуществляет запись в ячейку памяти по адресу *Addr* или регистр *Reg* из внешнего источника (файла, сетевого интерфейса и т.д.) размера *Size*, обновить отображения: *IsTainted*(*Addr*)=True и *IsTainted*(*Reg*)=True. В трассу ограничений добавить запись, фиксирующую связь символьной переменной, соответствующей *Addr* или *Reg*, и символьной переменной, соответствующей блоку данных из внешнего со смещениями [*Offset*, *Offset*+*Size*].
2. Если для каждого аргумента инструкции *IsTainted*(*Arg*)=False, обновить отображения для элемента (ячейки памяти или регистра) *Target*, в который записывается результат выполнения инструкции: *IsTainted*(*Target*)=False.
3. Если хотя бы для одного из аргументов инструкции *Arg* *IsTainted*(*Arg*)=True, обновить отображения для элемента *Target*, в который записывается результат выполнения инструкции: *IsTainted*(*Target*)=True. Записать в трассу *PathTrace* ограничение, связывающее *Target* и **все** аргументы инструкции, причём если для аргумента *IsTainted*(*Arg*)=False, то в ограничение помещается его непосредственное значение, в противном случае — символьная переменная соответствующего размера.
4. Если инструкция осуществляет ветвление, обладает свойством, указанным в пункте 3, и $CurDepth > Depth$ — добавить в трассу ограничений маркер точки ветвления и увеличить значение *CurDepth* на единицу.
5. Если $CurDepth \geq Depth + InvertDepth$, остановить выполнение программы.

Использование ограничителя глубины *InvertDepth* позволяет предотвратить обход путей в глубину, характеризующийся слишком интенсивным ростом размера трассы ограничений и сложности проверки выполнимости. Текущая глубина набора входных данных *Depth* отсекает создание маркеров точек ветвления для

ранних условных инструкций, зависящих от входных данных — эти точки ветвления уже были обработаны на предыдущих итерациях анализа.

Рассмотрим пример работы инструмента Avalanche, используя модельную программу (рис. 9), считывающую два символа со стандартного потока ввода. Предположим, что начальный набор входных данных включает два символа со значением `'\0'`. Для простоты описания вместо подсчёта базовых блоков для оценки прироста покрытия будем проводить подсчёт покрытых строк кода.

```
1 : int main()
2 : {
3 :   char s1, s2;
4 :   int i = 0;
5 :   s1 = getc();
6 :   s2 = getc();
7 :   if (s1 == 'o')
8 :     i++;
9 :   if (s2 == 'k')
10:     i++;
11:   return 1;
12: }
```

Рисунок 9: Модельный пример

Первый шаг работы Avalanche (рис. 10) заключается в запуске программы с помощью инструмента covgrind для инициализации рабочего множества *Inputs*.

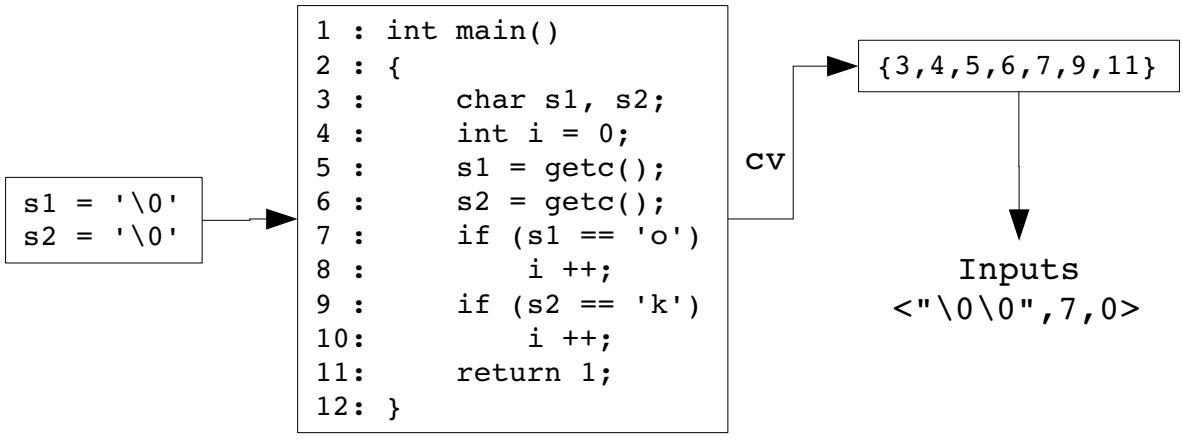


Рисунок 10: Первый шаг работы инструмента Avalanche

На первой итерации (рис. 11) проводится запуск программы на начальном наборе данных для создания трассы, разделяющейся на две подтрассы. В каждой подтрассе инвертируется последнее условие и с помощью инструмента STP строятся наборы входных данных. Обработка наборов с помощью covgrind пополняет рабочее множество Inputs.

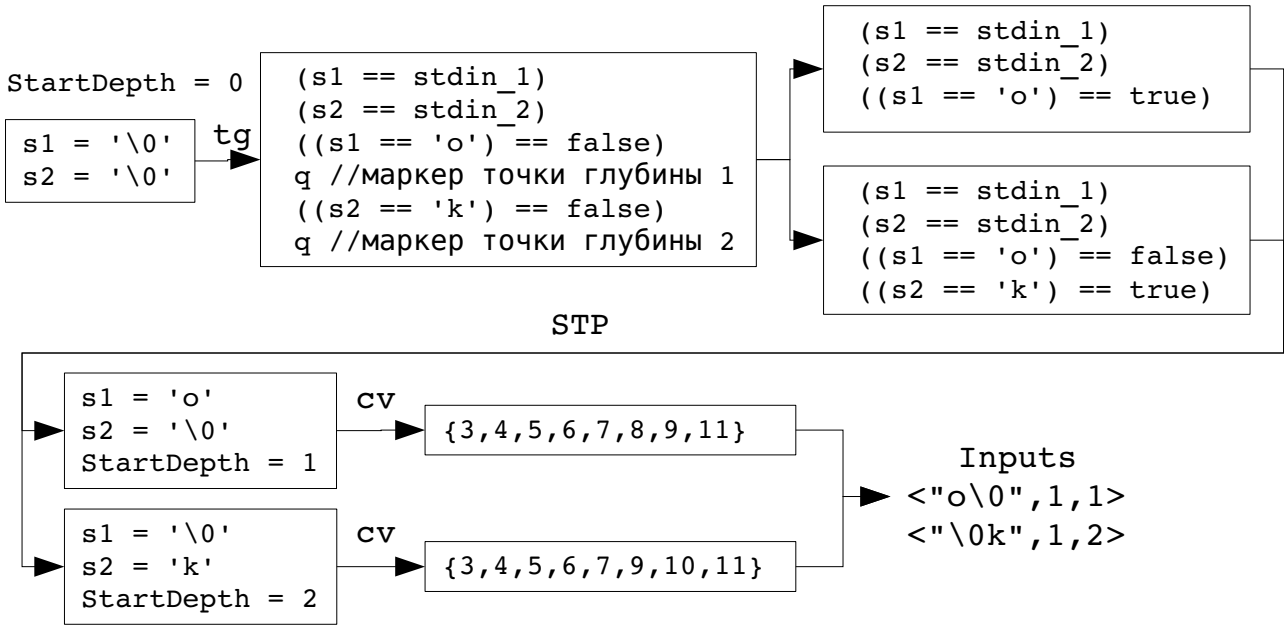


Рисунок 11: Первая итерация анализа

Для второй итерации (рис. 12) выбирается набор `"o\0"`, имеющий такую же перспективность, как и набор `"\0k"`, но меньшую глубину. При обработке tracegrind этот набор порождает одну новую подтрассу (первая точка ветвления отсекается значением *StartDepth*). Соответствующий подтрассе новый набор данных имеет значение метрики прироста покрытия 0, так как к этому моменту анализа все строки уже были покрыты.

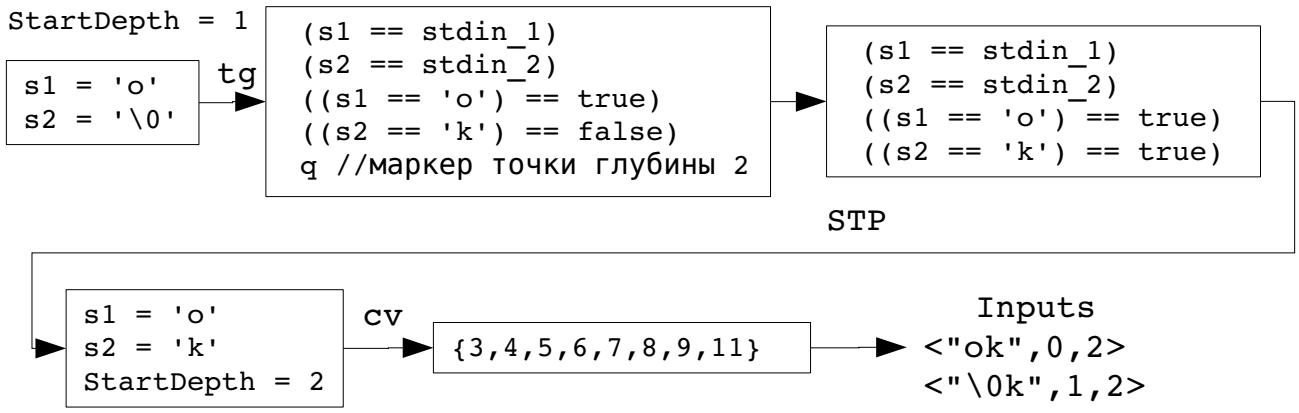


Рисунок 12: Вторая итерация анализа

На последующих двух итерациях (рис. 13) наборы данных не порождают новые подтрассы, так как значения глубины *StartDepth* для них отсекают обе точки ветвления. Рабочее множество *Inputs* исчерпывается и анализ завершается.

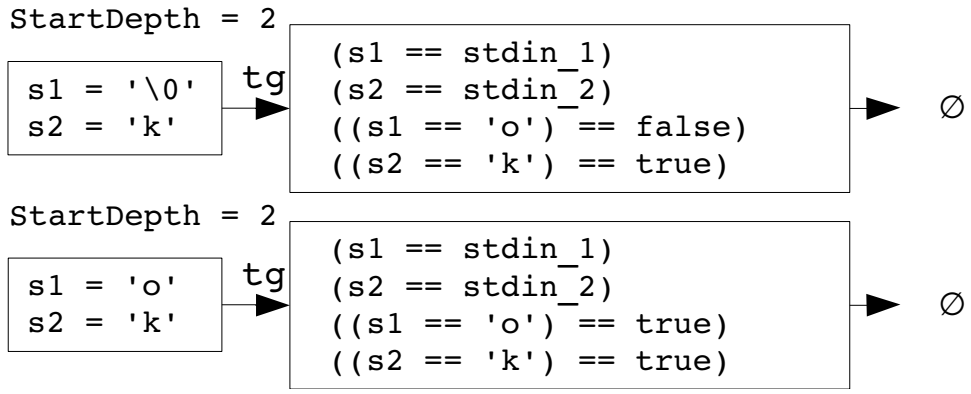


Рисунок 13: Третья и четвертая итерации анализа

Итак, на основе исполняемого файла программы и начального набора данных `"\0\0"` инструмент Avalanche автоматически построил три дополнительных набора данных (`"o\0"`,`"\0k"`,`"ok"`) для обхода всех возможных путей выполнения программы.

1.5 Выводы к главе 1

Рассмотренные в разделе 1.1 методы автоматического обхода путей выполнения программы, основанные на принципах символьного исполнения, реализованы в значительном количестве инструментов, активно применяющихся в теоретических и практических целях. В то же время, методы обладают набором ограничений различного рода; в настоящий момент исследования, направленные

на частичное или полное снятие данных ограничений, являются в достаточной степени актуальными.

Согласно положениям, рассмотренным в вводной части работы, были выбраны два перспективных направления, связанные с ограничением количества рассматриваемых путей выполнения на основе экспертных знаний об исследуемой программе и особенностях входных данных, которые она обрабатывает, и с понижением накладных расходов на обработку непосредственного выполнения программы на путях, проводимую с целью извлечения трасс выполнения.

По результатам обзора инструментов, приведённого в пункте 1.2, для реализации методов, предлагаемых в данной работе, был выбран инструмент Avalanche, разработанный в Институте системного программирования РАН.

Тем самым, глава 1 содержит подробное описание решения задачи 1, выделенной во вводной части.

Глава 2. Частичный анализ программ

Глава посвящена режиму проведения частичного анализа, в рамках которого проводится направленный обход путей выполнения согласно спецификациям, сформированным экспертом. Рассматривается влияние входных данных программы и внутренней структуры программы на проведение анализа и предлагаются сценарии, при которых экспертные знания о данных особенностях программы могут быть использованы для настройки анализа.

2.1 Вычислительная сложность полного анализа программ

В главе 1 были обозначены особенности программ, приводящие к высокой вычислительной сложности задачи автоматического обхода различных путей выполнения. Рассмотрим более подробно данные факторы:

1. объём кода целевой программы;
2. объём входных данных, обрабатываемых программой;
3. использование схем обработки данных, сводящихся к циклическому выполнению;
4. использование параллельных потоков обработки данных.

2.1.1 Увеличение объёма кода программы

Увеличение объёма кода, участвующего в обработке внешних данных, приводит к появлению дополнительных путей выполнения и увеличению размера трасс ограничений, соответствующих путям выполнения. Рассмотрим следующие простые примеры программ (рис. 14).

```
Программа 1
1: int main()
2: {
3:     int x = getc();
4:     if (x < 10)
5:         do1();
6:     else
7:         do2();
8:     return 1;
9: }
```

```
Программа 2
1 : int main()
2 : {
3 :    int x = getc();
4 :    if (x < 10)
3 :        do1();
4 :    else
5 :        do2();
6 :    if (x % 2)
7 :        do3();
8 :    else
9 :        do4();
10:    return 1;
11: }
```

```
Программа 3
1 : int main()
2 : {
3 :    int x = getc();
4 :    x = x + 5;
5 :    if (x < 10)
6 :        do1();
7 :    else
8 :        do2();
9 :    return 1;
10: }
```

Рисунок 14: Влияние объёма кода на пути выполнения программы

Все три программы считывают символ со стандартного потока ввода и проводят некоторую его обработку. Предположим, что функции do1, do2, do3, do4 не работают с входными данными — выполнение этих функций не порождает новых путей выполнения и не влияет на трассу ограничений. В данных предположениях программа 1 имеет два пути выполнения и для каждого из них трассы ограничений состоят из двух формул по выполнению инструкций на строках 3 и 4. Программа 2 отличается от программы 1 наличием ещё одного условного блока обработки значения переменной x. Это увеличивает число путей выполнения в два раза и добавляет в трассу ограничений каждого пути дополнительную формулу. Программа 3 отличается от программы 1 наличием дополнительной инструкции преобразования внешних данных, что увеличивает размер трасс ограничений.

На основе примера легко оценить потенциальный эффект расширения кода. Внесение линейных участков кода замедляет создание трассы ограничений и увеличивает время работы инструментов проверки выполнимости булевых формул. Добавление N условных блоков, обрабатывающих входные данные, увеличит число путей выполнения в 2^N раз в худшем случае.

2.1.2 Увеличение объёма входных данных

Влияние данного фактора в общем случае можно свести к фактору 1 или фактору 3. Действительно, предположим, что входные данные включают

фрагменты A,B,C,D,...,F, каждый из которых обрабатывается отдельной частью программы. Если необходимо с помощью программы обрабатывать данные, включающие фрагмент X, это потребует расширения функциональности программы и добавления новых блоков кода. Если необходимо с помощью программы обрабатывать данные, включающие несколько повторяющихся элементов $A_1,...A_m$, то в программе должны быть циклы, в рамках которых повторно работает модуль, обрабатывающий индивидуальные фрагменты A_i.

2.1.3 Проведение циклической обработки данных

Наличие в программе циклов, производящих обработку данных, увеличивает сложность проведения анализа эквивалентно увеличению объёма кода. Действительно, циклические конструкции, выполняющиеся фиксированное количество раз, можно развернуть в линейную последовательность инструкций. Циклические конструкции, количество повторений которых зависит от параметров запуска программы, т.е., фактически, от входных данных, вносят неограниченное количество возможных путей выполнения. Примером подобной ситуации могут служить программы, работающие в интерактивном режиме, основанном на ожидании некоторых команд и проведении обработки данных команд, — в таком режиме объём входных данных является неограниченным.

2.1.4 Использование параллельных потоков обработки данных

Отсутствие полного контроля со стороны программы над работой координатора вычислительных потоков современных операционных систем, контролирующего порядок и интервал предоставления процессорного времени вычислительным потокам, дополнительно увеличивает вычислительную сложность задачи динамического анализа для многопоточной программы.

Если в программе присутствуют блоки обработки входных данных, не имеющие точно определённых отношений следования [48], в рамках обхода путей необходимо обработать все возможные варианты порядка выполнения данных инструкций. Рассмотрим пример, иллюстрирующий подобный эффект (рис. 15).

```
1 : int main()
2 : {
3 :     int res;
4 :     thread t1, t2;
5 :     int x = getc();
6 :     t1.run(x, &res);
7 :     t2.run(x, &res);
//ожидание завершения потоков
8 :     await_threads(t1, t2);
9 :     if (res < 5)
10:         do1();
11:     else
12:         do2();
13:     return 1;
14: }
```

```
Поток t1
t1.1: void run(int x, int *r)
t1.2: {
t1.3:    *r = min(x,1);
t1.4: }

Поток t2
t2.1: void run(int x, int *r)
t2.2: {
t2.3:    *r = x + 2;
t2.4: }
```

Рисунок 15: Влияние параллельных потоков на пути выполнения программы

Рассмотренная программа создаёт два независимых потока управления, которые преобразуют символ, считанный со стандартного потока ввода, и записывают результат в переменную `res`, которая участвует в условной инструкции. Приведённый пример безусловно демонстрирует классическую ошибочную ситуацию гонки, однако можно предположить программы, в которых неопределённость работы потоков обрабатывается корректно без введений явных отношений следования.

В рамках данного примера выполнение инструкции t1.3 после инструкции t2.3 приведёт к несовместности пути выполнения по ветке *ложь* для условия на строке 9, т. к. минимум из значения переменной x и числа 1 не может превышать число 4. При проведении символьного исполнения это приведёт к тому, что инструмент проверки выполнимости не сможет подобрать новый набор входных данных. Если бы инструкция t2.3 была бы выполнена после инструкции t1.3, путь выполнения через блок do2 был бы исследован. Отметим, что стандартная схема символьного исполнения исследует только непосредственные инструкции программы, но не порядок их выполнения. Для управления порядком обработки инструкций необходимо расширение модели символьного исполнения и обеспечение контроля над координатором параллельных потоков.

2.2 Возможности частичного анализа программ

2.2.1 Общие положения

Рассмотрим вопросы применения метода автоматического обхода путей выполнения программы на основе символьного исполнения. Данные методы позволяют строить наборы входных данных, используемые для анализа качества целевой программы — поиска критических дефектов, исследования эффективности реализованных в программе алгоритмов и т. д. Автоматическое создание наборов тестов логично противопоставить ручному созданию тестов, которое производится в процессе разработки программы экспертами, обладающими познаниями о структуре кода программы, её функциональности и особенностях обработки входных данных. Эти познания используются для уменьшения временных затрат, путём минимизации тестовых наборов, покрывающих все актуальные фрагменты программы.

Автоматическое создание тестовых наборов не требует знаний экспертов, однако осуществляет обработку путей выполнения согласно фиксированному набору критериев. При наличии правильно построенной модели и алгоритмов вычисления входных данных для программы и используя большую мощность современных вычислительных устройств, автоматические методы могут позволить получить лучшие результаты по покрытию интересующей части программы тестовыми наборами, нежели тестовые наборы, созданные экспертами с учётом знания о внутреннем устройстве программы. Так, например, согласно работе [14] применение инструмента KLEE для анализа набора программ coreutils [49] позволило получить более полное покрытие кода по сравнению с использованием ручного набора тестов, поддерживаемого разработчиками coreutils.

Стоит отметить, что несмотря на подобные результаты, можно высказать предположение об актуальности использования знаний эксперта для настройки инструментов автоматического анализа программ с целью уменьшения вычислительной сложности задачи анализа программ, а, следовательно, и времени

анализа, с целью генерации наборов входных данных, позволяющих проанализировать в первую очередь наиболее интересные с точки зрения эксперта фрагментов программы.

2.2.2 Обзор существующих решений

В рамках некоторых инструментов автоматического обхода путей выполнения были реализованы методы, базирующиеся на рассмотренных выше идеях.

Так, инструмент PEX [23] осуществляет обработку не программы в целом, а отдельных модульных тестов, включающих сценарии работы с объектами и методами классов, реализованных на языке C#. Данные тесты не позволяют исследовать взаимодействие сразу всех компонентов программы, однако нацелены на проверку корректности алгоритмов. Предполагается, что данные тесты составляются вручную или с применением других методов обработки исходного кода (например, генетических алгоритмов, работающих с базой классов). Применение символьного исполнения позволяет автоматически исследовать граничные и экстремальные значения данных, обрабатываемых в рамках созданных тестов.

Инструменты KLEE [14] и S2E [19] предоставляют возможности ручной разметки источников данных в программе, а инструмент EXE — предшественник KLEE — поддерживал только ручную разметку без автоматической обработки файлов, аргументов командной строки и т. д. Подобный подход позволяет добиться в значительной степени точной настройки обхода путей выполнения. Типичный вариант использования этого подхода в рамках инструмента сводится к замене вызовов функций доступа ко внешним источникам обычными переменными, что позволяет абстрагироваться от деталей реализации данных функций средой исполнения и добиться независимости от реального состояния внешней среды. Недостатками данного подхода является требования наличия исходного кода и необходимости интеграции с системами сборки кода. Дополнительно, функции доступа ко внешним источникам могут учитывать

особенности самих внешних источников (например, права доступа, режимы буферизации, внутренняя структура), которые, с свою очередь, могут косвенно влиять на выполнение программы. При подмене результатов вызовов функций произвольными данными соответствующей структуре эти особенности учитываться не будут, что может повлиять на точность анализа в худшую сторону.

В работе [29] предлагается метод обхода путей выполнения, учитывающий внутреннюю логику разбора входных данных программой, и рассматривается реализация данного метода в рамках инструмента SAGE. Вместо обработки инструкций доступа ко внешним источниками данным в рамках метода обрабатываются инструкции вызова функций разбора входных данных на уровне токенов грамматики. Применение подобного метода позволяет не проводить обработку инструкций перевода байтов входных данных в токены с целью создания трассы ограничений. Ограничением данного метода является область его применимости — если в коде программы нельзя выделить отдельное подмножество функций, осуществляющих разбор входных данных на уровне грамматики, использовать метод напрямую нельзя.

В работе [30] рассматривается подход, основанный на использовании грамматики, описывающей формат входных данных программы, для создания множества начальных наборов, корректных с точки грамматики. Эти начальные наборы строятся так, чтобы включать не более N нетерминальных символов грамматики, для которых в грамматике есть только правила раскрытия в терминальные символы (например, нетерминальный символ «целое десятичное число», выводимый в последовательность терминальных символов из множества {-,0,1,2,3,4,5,6,7,8,9}). Построенные наборы данных обрабатываются системой символьного исполнения, совмещённого с обычным, причём при обходе путей выполнения варьируются только конкретные варианты раскрытия нетерминальных символов. Общий эффект применения метода схож с работой, рассмотренной выше — при анализе программы пропускается обработка точек ветвления, относящихся к разбору входных данных, что позволяет улучшить

покрытие программы по путям выполнения и по коду за ограниченное время. Недостатком метода является ограниченная масштабируемость — шаг создания множества начальных наборов не позволяет строить входные данные на основе длинных цепочек применения правил грамматики.

2.3 Предлагаемые методы частичного анализа

В рамках данной работы предлагаются методы, позволяющие проводить обход путей выполнения программы на основе статических спецификаций, ограничивающих области входных данных и функции анализируемой программы, которые будут участвовать в создании трассы ограничений.

Для введения методов рассмотрим некоторые потенциальные варианты организации программ.

2.3.1 Частичный анализ по фрагментам входных данных

Программа, представленная на рисунке 16, осуществляет считывание данных из внешнего источника. Фрагмент данных разбивается на две части; обработку первой осуществляет модуль программы, работа которого начинается при вызове функции parse1; обработку второй части осуществляет модуль программы, работа которого начинается при вызове функции parse2.

```
1 : int main()
2 : {
3 :     char buf1[256], buf2[32];
4 :     int fd = open(...);
5 :     read(fd, buf, 256);
6 :     read(fd, buf, 32);
7 :     parse1(buf1);
8 :     parse2(buf2);
9 :     return 1;
10: }
```

Рисунок 16: Пример программы, обрабатывающей два блока данных независимо

Сделаем следующие предположения касательно данной программы:

- Первый блок данных, считываемый на строке 5, не влияет на работу

функции parse2().

- На некотором наборе входных данных выполнение программы не завершается при выполнении функции parse1().
- Существует потребность проверить корректность реализации функции parse2().

Можно заметить, что при проведении автоматического обхода путей выполнения точки ветвления функции parse1(), зависящие от входных данных, имеют меньшую глубину, чем точки ветвления parse2(). Дополнительно, при анализе программы в трассу ограничений будут попадать формулы, построенные по инструкциям parse1(). В рамках предположений, сделанных выше, точки ветвления и фрагменты трассы ограничений, соответствующие parse1(), не являются актуальными для обработки точек ветвления parse2(), однако будут учитываться при анализе.

Ускорить целенаправленный анализ функции parse2() можно применением фильтра на входные данные. Подобный фильтр зафиксирует конкретные значения первых 256 байтов, считываемых из источника и предотвратит попадание в трассу ограничений всех инструкций, обрабатывающих эти байты. Символьное исполнение программы в таком случае эквивалентно исполнению модифицированной программы (рис. 17).

```
 1 : int main()
 2 : {
 3 :     char buf1[256], buf2[32];
 4 :     int fd = open(...);
 5 :     fseek(fd, 256, SEEK_SET);
 6 :     buf1 = {...};
 7 :     read(fd, buf2, 32);
 8 :     parse1(buf1);
 9 :     parse2(buf2);
10:     return 1;
11: }
```

Рисунок 17: Результат применения фильтрации данных при символьном исполнении

Данная программа пропускает первые 256 байтов внешнего источника, инициализирует первый блок обработки некоторыми константными значениями и считывает значения только во второй блок. При символьном исполнении будут рассматриваться пути выполнения, создаваемые точками ветвления функции parse2(), и имеющие более короткие трассы ограничений.

Отметим важную особенность такого перехода — если бы предположение об отсутствии влияния первого блока данных на работу функции parse2() не выполнялось, это бы могло привести к невозможности построить наборы входных данных для обхода определённых путей выполнения, так как значения блока buf1 зафиксированы. В то же время, применение фильтров не приводит к построению некорректных трасс ограничений, которым не соответствуют никакие реальные пути выполнения.

Для реализации данного метода предлагается следующая модификация инструмента Avalanche:

- Определяется язык спецификаций, позволяющий задавать статические множества интервалов смещений {<StartOffset,EndOffset>}, где $0 \leq StartOffset \leq EndOffset$, или специальные метки Full и None.
- Пользователь задаёт спецификации для каждого источника внешних данных, обрабатываемого целевой программой.
- Добавить дополнительные проверки в шаг 2.1 алгоритма работы инструмента tracegrind, представленного в разделе 1.4:
  - Обновление отображения IsTainted для ячеек памяти и регистров программы и запись ограничений на символьные переменные производится только в том случае, когда источник данных не имеет пометку None и имеет пометку Full или же смещения [Offset,Offset+Size] целиком покрываются интервалами {StartOffset,EndOffset}.

2.3.2 Частичный анализ по блокам кода программы

Фильтрация по данным является сильным ограничением — как было замечено раньше, зависимости по данным между блоками могут вызвать

невозможность построить наборы входных данных из-за фиксированности отдельных их фрагментов. Модифицируем модельный пример, чтобы визуально отобразить данную зависимость (рис. 18).

```
 1 : int main()
 2 : {
 3 :     char buf1[256], buf2[32];
 4 :     int fd = open(...);
 5 :     read(fd, buf1, 256);
 6 :     read(fd, buf2, 32);
 7 :     parse1(buf1);
 8 :     parse2(buf1, buf2);
 9 :     return 1;
 10: }
```

Рисунок 18: Пример программы, обрабатывающей данные отдельными функциональными блоками

Рассмотрим более слабое ограничение на основе следующих предположений для данного примера:

- Первый блок данных влияет на работу функции parse2().
- Функция parse1() вносит ограничения на значения первого блока данных.
- Ограничения, накладываемые функцией parse1() на значения первого блока данных, допускают любые варианты выполнения условных инструкций функции parse2().

Вариантов выполнения этих предположений может быть следующая функциональность parse1() и parse2():

- для каждого байта buf1 B функция parse1() задаёт ограничение: $10 \leq B \leq 20$ (1);
- для каждого байта buf1 B функция parse2() осуществляет ветвление по условию $B \leq 15$ (2);

В этом случае для каждого байта B совместны как система $(10 \leq B \leq 20) \wedge (B \leq 15)$, так и система $(10 \leq B \leq 20) \wedge (B > 15)$. Покажем, что данные предположения действительно задают более слабое ограничение при обходе путей:

1. При использовании фильтра по данным значениями первого блока нельзя манипулировать при символьном исполнении, так как для них не создаются символьные переменные. Если бы при реальном запуске все байты первого блока данных имели бы значение 15, условные инструкции в точках ветвления функции parse2() имели бы один возможный результат (истина), так как результат *ложь* требовал выполнения условия 15>15, что, очевидно, невозможно.
2. Без использования фильтра по данным значения первого блока более не являются фиксированными и оба результата условных инструкций функции parse2() являются возможными.

Отметим практический смысл предположения о допуске любых результатов выполнения инструкций в точках ветвления parse2() в рамках ограничений parse1() - для того, чтобы обойти все пути, порождаемые этими точками ветвления, достаточно единственного пути через parse1(). Это значит, что для анализа parse2() не нужно обходить обрабатывать точки ветвления в parse1(). При использовании такого подхода уменьшается число путей выполнения, которые нужно обработать.

Безусловно, даже рассмотренное выше более слабое предположение может выполняться не всегда. Как и ранее, в случае его невыполнения обход всех путей функции parse2() будет невозможен без обработки точек ветвления функции parse1(). Следующим шагом к ослаблению ограничений является либо фильтрация определённых точек ветвления функции parse1() на основе локальных связей по данным или выполнение анализа в обычном режиме — без использования фильтрации и экспертных знаний.

Для реализации данного метода предлагается следующая модификация инструмента Avalanche:

- Определяется язык спецификаций, позволяющий указывать множества имён функций {<IncName>}, точки ветвления в которых необходимо обрабатывать, и множество имён функций {<ExclName>}, точки ветвления

в которых необходимо пропускать.

- Пользователь задаёт данные спецификации при запуске инструмента.
- Добавить дополнительные проверки в шаг 2.4 алгоритма работы модуля tracegrind, представленного в разделе 1.4:
  - Маркер точки ветвления при выполнении стандартных условий, предполагаемым шагом 2.4, добавляется тогда и только тогда, когда:
    - множество {<IncName>} не пусто и стек вызовов программы на момент выполнения обрабатываемой условной инструкции содержит хотя бы одну функцию, имя которой присутствует во множестве {<IncName>};
    - множество {<IncName>} пусто и стек вызовов программы на момент выполнения обрабатываемой условной инструкции не содержит ни одной функции, имя которой присутствует во множестве {<ExclName>}.

2.4 Экспериментальные результаты применения методов

Для проведения экспериментов на применение частичного анализа были использованы программы, осуществляющие разбор входных данных сложного формата:

- llc — инструмент разбора кода во внутреннем представлении виртуальной машины LLVM [15];
- pbd_dump – инструмент разбора кода во внутреннем представлении виртуальной машины Parrot [50];
- monodis – компонент среды исполнения кода на языке C# mono [51].

Для анализа данных проектов был использован инструмент Avalanche в стандартном режиме и в режиме использования спецификаций по входным данным и функциям. В качестве настроек анализа использовались значения, ранее примененные в практике инструмента Avalanche – время анализа было ограничено двумя часами, максимальная глубина просмотра точек ветвления на каждой

итерации — 100 точками. В качестве начальных данных были использованы файлы простых программ во внутреннем представлении соответствующего формата.

В качестве спецификаций ограничений по входным данным для каждого проекта использовались маски, выделяющие фрагменты, соответствующие структурным элементам внутреннего представления кода, следующим непосредственно за блоком, содержащим заголовочную информацию.

В качестве спецификация ограничений по функциям для каждого проекта использовались фильтры, исключающие ветвления в функциях разбора данных примитивных типов при чтении их из файла. Данные функции использовались практически во всех модулях программ и приводили к появлению излишнего количества повторяющихся точек ветвления.

Результаты запусков приведены в таблице 2.

Таблица 2: Сравнение результатов обычного и частичного режимов анализа

Программа	Проверенные пути выполнения	Среднее время работы STP, с	Количество найденных уникальных дефектов	Время обнаружения первого дефекта, с
Полный анализ				
llc	5153	0,07	5	9
pbc_dump	4238	0.098	3	41
monodis	1122	0.064	-	-
Частичный анализ				
llc	11977	0.034	7	5
pbc_dump	4712	0.078	6	10
monodis	4127	0.048	4	67

В представленной таблице рассмотрены параметры результатов анализа, которые напрямую улучшаются за счёт применения частичного анализа. Уменьшение объёма обрабатываемых данных снижает среднее время обработки трасс ограничений, что увеличивает общую скорость анализа с точки зрения

обхода путей. Повышение количества обойдённых путей позволяет обнаружить большее количество дефектов за ограниченное время анализа. Необходимо отметить так же и параметр в последнем столбце, соответствующий промежутку времени с начала анализа, который понадобился для обнаружения первого дефекта. Уменьшение этого времени является прямым следствием специализации анализа.

Найденные дефекты включают следующие:

- Аварийное завершение программы при попытке выполнить разыменование нулевого или некорректно инициализированного указателя (llc, pbc_dump, mono).
- Аварийное завершение программы по сигналу SIGABRT из-за нарушения условия в программной конструкции assert (llc, pbc_dump, monodis).
- Зацикливание программы из-за попытки считать некорректный объём данных (pbc_dump).

2.5 Выводы к главе 2

Глава 2 содержит подробное описание решения задачи 2, поставленной в вводной части работы. Описание практических экспериментов в разделе 2.4 освещает задачу 5, поставленную в вводной части работы.

Рассмотренные в разделе 2.1 особенности программ, влияющие на сложность обхода путей выполнения, свидетельствуют о том, что в рамках анализа программ выбор того, какие пути должны быть рассмотрены, в значительной степени определяет эффективность инструмента анализа. При реализации механизма данного выбора возможно учитывать имеющуюся информацию о самой программе и формате входных данных, которые она обрабатывает. Тем не менее, существующие решения, основанные на использовании подобной информации, имеют ряд ограничений. Метод, предлагаемый в инструменте SAGE [29], применим к программам, реализующим определённый механизм разбора входных данных; метод, предлагаемый в

инструменте CESE [30], плохо масштабируется с увеличением объёма входных данных. Методы в инструментах KLEE [14], EXE [16] и S2E [19] требуют ручной модификации исходного кода программ.

Предлагаемый метод, описанный в разделе 2.3, может быть применим к любым программам, которые анализируются инструментом Avalanche, и не требует модификации исходного кода. Метод основывается на использовании спецификаций, предоставляемых пользователем, для того, чтобы целенаправленно исследовать пути выполнения программы, осуществляющие обработку отдельных фрагментов входных данных, и исследовать точки ветвления в отдельных функциях программы. Подобный целенаправленный анализ представляется актуальным для тестирования независимых модулей программы в процессе разработки.

Предлагаемый метод, реализованный путём расширения функциональности компонента tracegrind в инструменте Avalanche, решает поставленную цель работы (в части повышения эффективности анализа путей выполнения посредством отсечения некоторых точек ветвления). Приведённые в разделе 2.4 результаты экспериментов по применению метода показали эффективность его применения на практике — для набора анализируемых проектов было достигнуто ускорение анализа и были получены новые критические дефекты, приводящие к аварийному завершению программ.

Глава 3. Эффективное использование вычислительных ресурсов

Рассмотренные ранее ограничения методов анализа программ на основе автоматического обхода путей выполнения, связанные с высокой вычислительной сложностью, являются в некотором смысле фундаментальными. Количество путей в программе можно ограничивать разными методами, однако в общем случае оно будет огромным даже для программ среднего размера. Формально, использование неограниченных потоков входных данных (например, взаимодействие с пользователем в интерактивном режиме) делает количество возможных путей выполнения бесконечным.

В настоящее время одним из возможных решений для проблем подобного рода является использование параллельных и распределённых вычислений. В процессе создания и поддержки программного обеспечения использование многоядерных и многопроцессорных рабочих компьютеров для индивидуальных разработчиков является в настоящее время нормой; дополнительно команды разработчиков часто используют вычислительные кластеры для ежедневного выполнения тестовых наборов или отдельных практических экспериментов.

В связи с этим многие инструменты, использующиеся для автоматизации решения вычислительно сложных задач разработки, предоставляют поддержку режимов параллельного и распределённого выполнения. Актуальность использования подобных режимов определяется тем, насколько эффективно настроено взаимодействие отдельных вычислительных узлов и какой общий прирост производительности может быть достигнут.

Задача автоматического обхода путей выполнения включает отдельные подзадачи, связанные с обработкой больших объёмов данных. Наличие отдельных модулей решения подзадач и внутренняя структура обрабатываемых данных, включающая повторяющиеся элементы (например, отдельные пути выполнения), позволяют предположить существование широких возможностей для распараллеливания и распределения вычислений.

3.1 Распараллеливание анализа путём выделения независимых подзадач

Рассмотрим основные подзадачи, решаемые в процессе анализа программ. Обход осуществляется по дереву путей, причём во время обхода каждому пути соответствует трасса ограничений от точки входа и набор входных данных при условии, что путь является выполнимым. Действия по обработке пути выполнения включают, как минимум, построение трассы ограничений и обработку трассы ограничений решателем для проверки выполнимости. Для различных путей выполнения трассы ограничений могут иметь общие фрагменты. Так, например, путь {IABD...} и путь {IABE...} (рис. 19) имеют:

- общие фрагменты трасс ограничений вплоть до точки ветвления B;
- разные ограничения на выполнение условной операции в точки ветвления B;
- в общем случае независимые фрагменты трасс ограничений после точки ветвления B.

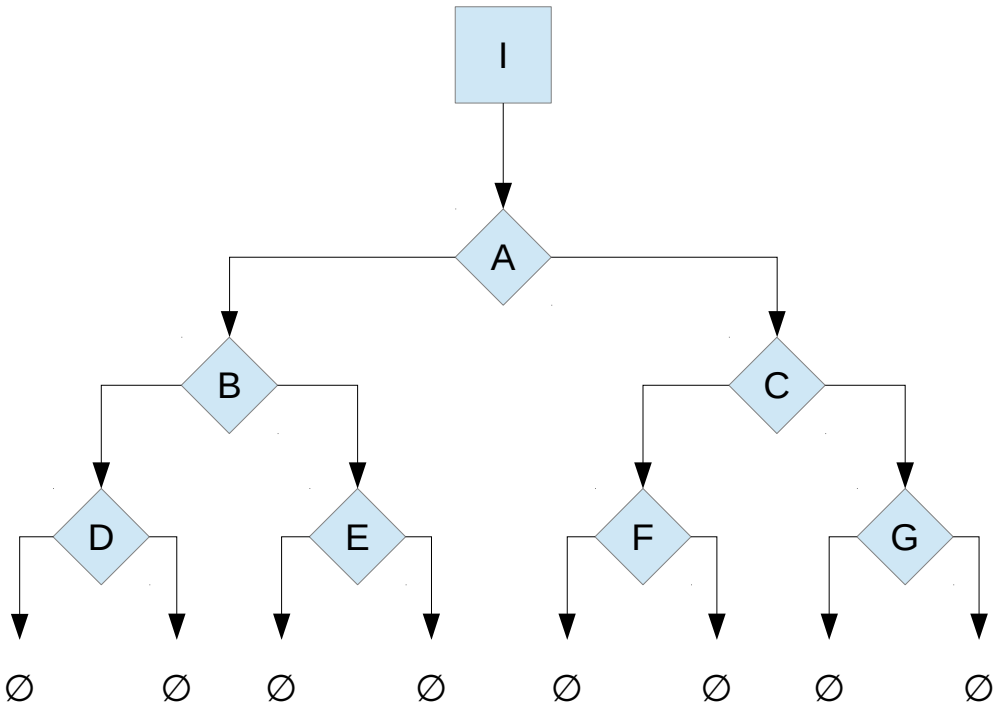


Рисунок 19: Дерево путей выполнения программы

Формально, каждый путь выполнения можно рассматривать независимо.

Возможные варианты параллельной обработки путей, выполняющейся несколькими вычислительными узлами, близки к двум рассмотренным ранее режимам обхода:

1. Каждый вычислительный узел может производить обработку произвольного пути выполнения
2. Каждый вычислительный узел осуществляет обработку путей, входящих в определённое поддерево.

Преимущества первого подхода, эффективно реализуемого на основе режима offline, включают малые затраты на балансировку нагрузки и гибкость организации общего хранилища информации о программе, накапливаемой во время анализа, — количество обращений к данному хранилищу и объём передаваемых данных можно настроить оптимально исходя из затрат на синхронизацию и степени уменьшения объёма вычислений за счёт использования разделяемых данных о работе программы.

Преимущества второго подхода, соответствующего режиму online, включают возможность поддержания локальных версий хранилища информации о программе, позволяющего минимизировать затраты на синхронизацию. В частности, в локальных хранилищах может хранится информация об общем для всего поддерева путей фрагменте трассы ограничений. В то же время, применение подобного подхода требует использования модуля балансировки нагрузки, так как в общем случае обработка поддерева может занимать произвольное время по отношению к другим поддеревьям.

3.2 Внутренняя параллельность независимых подзадач

Подзадачи обработки пути выполнения включают построение трассы ограничений и проверку выполнимости трассы ограничений. В рамках символьного исполнения, сопряжённого с реальным исполнением, возможности использования нескольких вычислительных узлов крайне ограничены. В самом деле, обработка программы, выполняющейся линейно, не позволяет эффективно

использовать все доступные ресурсы, поскольку нарушение последовательности инструкций программы не является приемлимым. Потенциальное ускорение может быть достигнуто за счёт использования двух потоков выполнения: в рамках одного потока проводится выполнение программы, в рамках второго потока — обработка инструкций и создание трассы. При этом обычно действия по обработке инструкций занимают большее время, чем работа самой программы, что требует использования буферов и простоя первого потока при переполнении буферов.

В последнее время было предложено несколько решений [52,53] для организации параллельной обработки наборов ограничений. В то же время трассы ограничений, создаваемые в процессе автоматического обхода путей выполнения, обладают рядом свойств, ограничивающих прирост производительности при увеличении количества вычислительных узлов, выделяемых для решения подзадачи проверки выполнимости. Согласно работе [54], рассматривающей опыт применения системы SAGE, накопленный за десятки машинных лет работы, проверка выполнимости большинства индивидуальных трасс занимает малое время (90% трасс — менее 0.1 секунды, 84% трасс — менее 0.05 секунды). В данных условиях использование большого количества вычислительных узлов для обработки индивидуальных трасс позволит получить ограниченное повышение общей эффективности анализа.

3.3 Обзор существующих решений

Использование распределённых и параллельных вычислений было организовано в рамках нескольких современных систем, осуществляющих автоматический обход путей выполнения в программах. Дополнительно, некоторые исследования предлагают методы повышения эффективности сбора информации о выполнении программы на основе применения параллельных потоков.

Инструмент SAGE [54] предоставляет возможность проводить анализ

программ на вычислительном кластере. В рамках схемы работы на кластере предполагаются отдельные независимые запуски инструмента SAGE, осуществляющие полный анализ программ с различными наборами начальных входных данных и настроек как анализа, так и самой программы. Данное решение не рассматривает возможность повышения скорости анализа в рамках одного запуска программы, но позволяет осуществлять исследование программы в промышленном масштабе. Центральный управляющий модуль контролирует корректность работы отдельных процессов инструмента SAGE, выполняющихся на узлах кластера, контролирует загрузку отдельных узлов и осуществляет сбор подробной статистики по работе инструмента. Подобный подход позволяет повысить шанс обнаружения дефектов и автоматически извлекать большой объём информации, которую можно использовать для уточнения последующих запусков.

Инструмент Cloud9 [55] предоставляет возможность исследования программ на вычислительном кластере с помощью инструмента KLEE [14], рассмотренного в главе 1. Распределение вычислительной нагрузки производится по отдельных фрагментам дерева путей выполнения — на каждом вычислительном узле выполняется независимый процесс KLEE, обрабатывающий выделенный фрагмент дерева. Процессы KLEE проводят анализ в режиме online и поддерживают локальные хранилища информации о программе (трассы ограничений, состояния программы в точках ветвления), и не осуществляют обмен информацией из этого хранилища. Это не позволяет в полной мере использовать оптимизации KLEE, но общая эффективность анализа повышается за счёт доступа к большему объёму вычислительных ресурсов. Процессы KLEE осуществляют обмен данными с динамическим балансировщиком нагрузки, определяющем сложность заданий на индивидуальных процессах по ряду признаков (покрытие кода, количество точек ветвления в поддереве). При обнаружении слишком сложного задания, отдельные процессы сигнализируют об этом балансировщику и под его контролем осуществляют передачу фрагментов задания на менее загруженные узлы. Подобный контроль позволяет добиться

равномерного анализа программы и повышения скорости наращивания покрытия кода и количества проанализированных путей выполнения.

Инструмент KLEE осуществляет обход путей выполнения с использованием предоставленных локальных вычислительных ядер. Работа в режиме online предусматривает синхронизацию параллельных потоков выполнения при работе с множеством состояний программы; выполнение отдельных путей выполнения программы параллельно осуществляется встроенными возможностями виртуальной машины LLVM.

В работе [56] предлагается схема децентрализованного параллельного анализа на основе инструмента Pex [23]. В рамках данной схемы отдельные исполнители заданий выполняются на независимых ядрах в многоядерной системе. Каждое задание заключается в проведении работы с некоторым путём выполнения и разрешением трасс ограничений с помощью решателя. Результатом выполнения задания является не только проверка программы на пути выполнения, но и построение новых заданий. Каждый исполнитель использует детерминированный алгоритм для выбора задания из имеющегося хранилища заданий. Предложенный авторами алгоритм выбора основан на вычислении точки ветвления в построенном дереве ветвлений по фиксированной хэш-функции.

3.4 Параллельные вычисления в рамках Avalanche

Инструмент Avalanche производит анализ программ в режиме offline и использует схемы распределения нагрузки между модулями, обеспечивающую проверку большого количества путей выполнения при минимизации действий, направленных на сбор трассы ограничений. Реализация инструмента Avalanche включает последовательную обработку программ с организацией единого потока данных между модулями под контролем управляющего модуля. В то же время, схема работы инструмента Avalanche не ставит фундаментальных ограничений на использование параллельных потоков вычислений.

Рассмотрим более подробно потоки данных, формируемых отдельными

модулями инструмента Avalanche, согласно схеме, рассмотренной в главе 1. Работа инструмента производится итеративно и на каждой итерации производятся следующие действия:

1. Выполнение программы с целью построения трассы ограничений на текущем наборе входных данных.
2. Разбиение трассы ограничения на N подтрасс по точкам ветвления и обращение последнего ограничения в каждой подтрассе для получения альтернативного пути.
3. Проверка выполнимости каждой из N подтрасс, результатом которой является M новых наборов входных данных (число M произвольно для отдельных итераций, однако M⩽N). Проверка выполнимости включает в себя запуск на выполнение решателя ограничений STP.
4. Для каждого из M наборов входных данных проводится проверка корректности работы программы и оценка прироста покрытия базовых блоков. Наборы данных вносятся в рабочее множество со своими оценками, среди всех наборов в рабочем множестве выбирается наилучший для следующей итерации.

   Поток данных можно рассмотреть на рисунке 20.

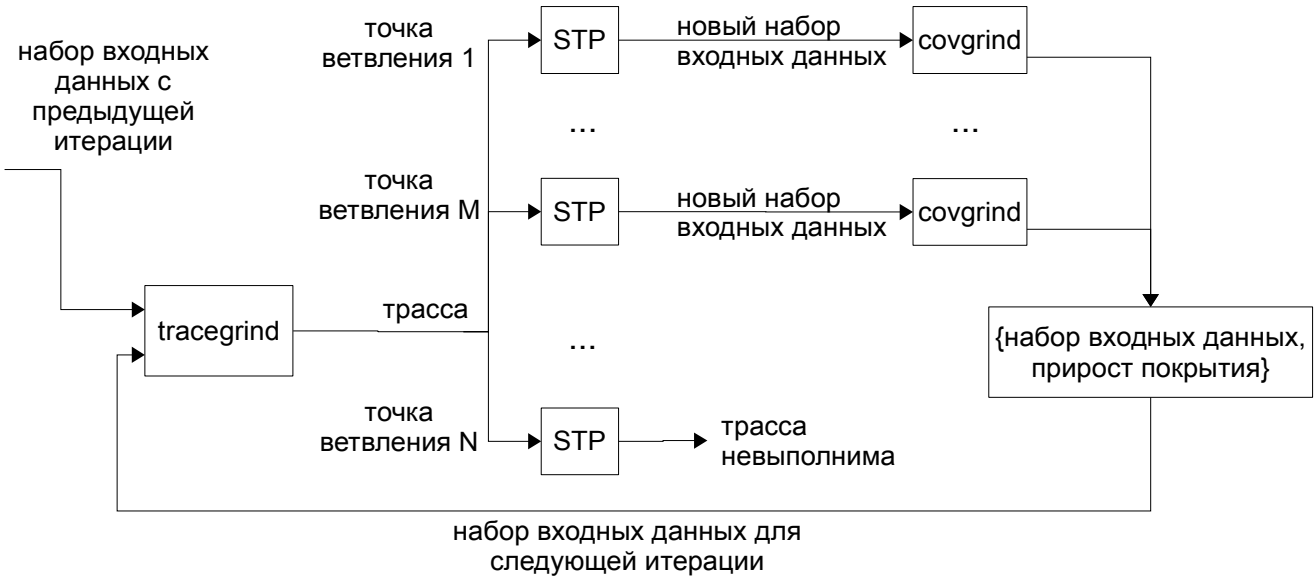


Рисунок 20: Обработка данных в рамках одной итерации анализа инструмента Avalanche

Отметим основные особенности данной схемы обработки:

- Разбиение трассы на подтрассы производится по завершению работы модуля tracegrind и не зависит от результатов обработки подтрасс.
- Работа с подтрассой *A* не зависит от результатов работы с подтрассой *B* для любых *A* и *B*.
- Работа с набором входных данных $Input_A$ зависит от результата обработки подтрассы *A*, но не зависит от обработки трассы *B* или обработки набора входных данных $Input_B$ для любых *A* и *B*.

Как было сказано ранее, внутреннее распараллеливание работы модулей tracegrind и STP потенциально позволит достигнуть ограниченного увеличения производительности, плохо масштабируемого с увеличением количества параллельных узлов. Это подтверждается и практическими результатами экспериментов с помощью инструмента Avalanche, проведёнными ранее. Согласно [25], доля времени анализа, включающая только работу модуля tracegrind, не превышала 10% для 9 из 10 рассмотренных проектов с открытым исходным кодом. В то время как доля времени анализа, включающая только работу модуля STP, для большинства проектов превышала соответствующие значения для модулей tracegrind и covgrind, малые затраты на обработку индивидуальных трасс ограничений не позволяют эффективно наращивать производительность анализа при добавлении вычислительных узлов.

3.5 Предлагаемая модификация схемы работы Avalanche

На основе рассмотренных выше особенностей работы инструмента Avalanche и обзора работ в данной области, был сделан следующий вывод — распараллеливание индивидуальных запусков модулей Avalanche не является эффективным для большинства проектов.

В связи с этим, было принято решение разработать схему проведения параллельных вычислений в рамках отдельных итераций, основываясь на независимости обработки подтрасс и соответствующих им наборов входных

данных, отмеченной выше. В рамках схемы вводятся следующие обозначения:

- **задание обработки пути выполнения** — пакет команд, включающих запуск модуля STP, обработку результата модуля STP и запуск модуля covgrind; входными данными для задания является подтрасса ограничений;
- **банк заданий** — множество заданий обработки пути выполнения;
- **исполнитель** — программный модуль, осуществляющий обработку заданий в рамках одного параллельного потока выполнения;
- **пул исполнителей** — множество исполнителей, зарегистрированных управляющим модулем.

Схема работы инструмента Avalanche, указанная в главе 1, модифицируется следующим образом:

- Внутренняя функциональность модулей tracegrind, STP и covgrind остаётся без изменений.
- Перед началом первой итерации анализа управляющий модуль инициализирует пул исполнителей по количеству запрашиваемых пользователем параллельных потоков выполнения.
- Во время выполнения итерации анализа модуль tracegrind запускается на наборе входных данных *Input*, имеющем глубину *Depth*. Полученная трасса разбивается на упорядочённое множество пар $\{Subtrace_1, Depth+1\}$, $\{Subtrace_2, Depth+2\}, ..., \{Subtrace_N, Depth+N\}$, где N — количество маркеров точек ветвления в трассе, $Subtrace_i$ соответствует фрагменту трассы от начала до i-ого маркера точки ветвления.
- В банк заданий добавляется N заданий. Для i-ого задания входными данными является $Subtrace_i$.
- Пока в банке присутствуют невыполненные задания управляющий модуль проверяет наличие свободных исполнителей в пуле исполнителей.
- При появлении свободного исполнителя, задание с минимальным порядковым номером передаётся данному исполнителю.

- По завершению работы исполнителя над заданием, исполнитель передаёт управляющему модулю пару {*NewInput*,*BBSet*}, где *NewInput* — это новый набор входных данных, *BBSet* — множество базовых блоков, выполненных при запуске программы на наборе *NewInput*, полученное модулем covgrind. Пара может иметь значение {∅,∅}, если подтрасса задания оказалась невыполнимой.
- Для каждого результата {*NewInput*,*BBSet*}, соответствующего заданию *i* и транзитивно паре {$Subtrace_i$,*Depth*+*i*}, управляющий модуль осуществляет базовые действия по формированию нового набора данных {*NewInput*,|*GlobalBBSet**BBSet*|,*Depth*+*i*} и обновлению множеств активных наборов данных Inputs.

  Данная схема обработки представлена на рисунке 21.

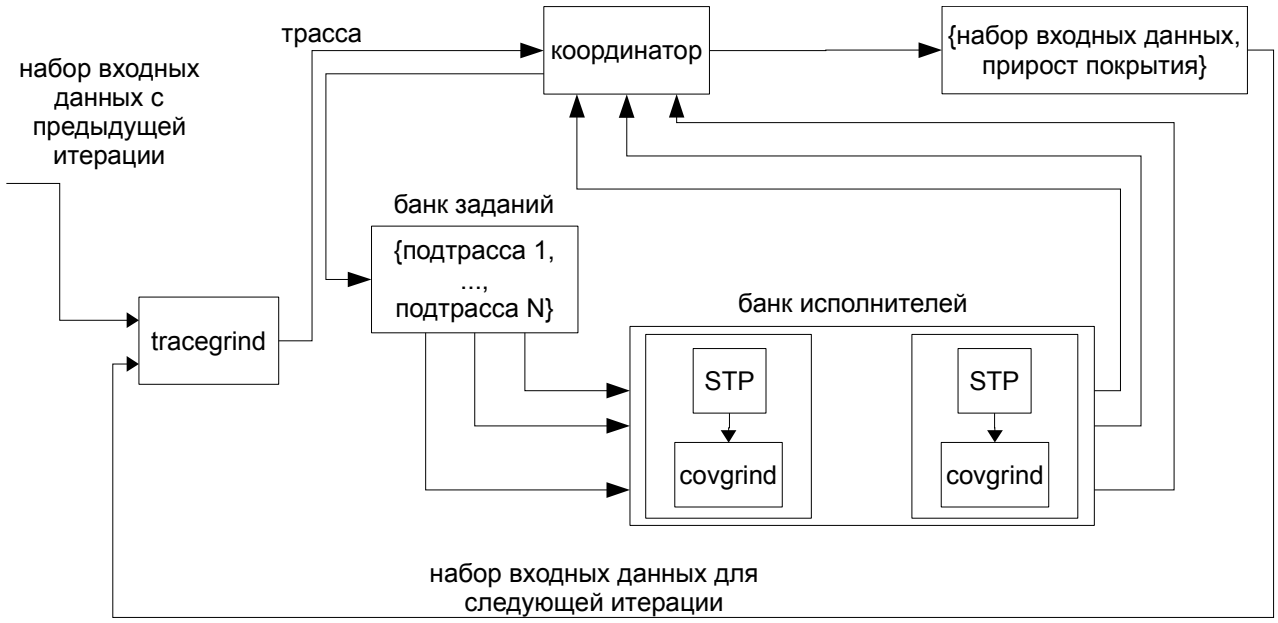


Рисунок 21: Схема параллельной обработки данных в рамках итерации

3.6 Оценки прироста производительности

Рассмотрим возможный прирост производительности, достижимый при использовании параллельной схемы обработки и нескольких параллельных потоков выполнения.

Для выявления практических оценок будем использовать формулировку закона Амдала, рассматривающую применение m параллельных вычислителей при условии, что доля вычислений, которые не могут быть выполнены параллельно, составляет α. В этом случае максимально достижимый прирост производительности составляет:

$$\frac{1}{\alpha + \frac{1-\alpha}{m}}$$

В рамках указанной выше схемы вычисления, проводимые строго с использованием одного потока выполнения, включают:

- Работу модуля tracegrind по извлечению трассы ограничений на пути выполнения программы.
- Работу управляющего модуля по разбиению трассы ограничений на подтрассы, формирование банка заданий и обработку результатов выполнения заданий.
- Выполнение последнего задания из банка заданий в тот момент исполнителем, когда все остальные исполнители уже являются свободными.

К сожалению, чисто теоретическая оценка доли времени, которое требуется для выполнения указанных выше вычислений, не может быть получена. Доля работы компонента tracegrind варьируется в зависимости от исследуемой программы, используемого набора входных данных и среды выполнения. Аналогичными особенностями обладает доля работы управляющего модуля, однако на практике при применении инструмента Avalanche она минимальна. Доля последнего рассмотренного элемента последовательных вычислений варьируется не только для разных программ и входных файлов, но и в рамках отдельных итераций анализа.

Введём следующую формулу для определения величины α, позволяющую описать максимально общий случай. Данная формула рассматривает ситуацию проведения анализа, включающего I итераций.

$$\alpha = \alpha_1 + \alpha_2;$$
$$\alpha_1 = \frac{Tg + Dr}{Tg + Dr + Cv + STP};$$
$$\alpha_2 = \sum_{i=1,I} \frac{T_i * max_{j=1,N_i}(Cv_{i,j} + STP_{i,j})}{T * (Cv_i + STP_i)}$$, где

- Tg — общее время работы tracegrind при проведении анализа,
- Dr — общее время работы управляющего модуля при проведении анализа,
- Cv — общее время работы covgrind при проведении анализа,
- STP — общее время работы STP при проведении анализа,
- Cv_i (STP_i) — общее время работы covgrind (STP) на итерации i,
- $Cv_{i,j}$ $(STP_{i,j})$ — время работы covgrind (STP) для пути выполнения j на итерации i,
- I — число итераций анализа,
- N_i — число подтрасс ограничений, обработанных на итерации i,
- T_i — время выполнения итерации i, T — общее время анализа.

Слагаемое α_1 соответствует доле времени работы модулей tracegrind и управляющего модуля в общем времени анализа. Слагаемое α_2 соответствует указанной выше доле времени простоя исполнителей при обработке последнего задания, причём рассматривается более строгое ограничение, включающее следующие условия:

- все свободные исполнители простаивали на всём протяжении времени обработки последнего задания;
- последнее задание занимало больше всего времени для выполнения среди заданий данной итерации.

Для исследования значений α будем использовать практические результаты работы инструмента Avalanche на наборе проектов с открытым исходным кодом. Приведённые в конце главы результаты применения параллельного анализа будут включать те же самые проекты для оценки эффективности предложенной схемы.

В рамках практических экспериментов была исследована часть проектов, рассмотренные ранее в работе [25], включающая:

- cjpeg — утилита трансформации файлов изображений в формат JPEG (jpeg-7 [57]);

- mpeg2dec — утилита декодирования информации в файлах формата MPEG-2 (libmpeg2-0.5.1 [58]);
- mpeg3dump — утилита декодирования информации в файлах формата MPEG-3 (libmpeg3-1.8 [59]);
- swfdump — утилита декодирования информации в файлах формата SWF (swftools-0.9.0 [60]);
- qtdump — утилита декодирования информации в файлах формата AVI (libquicktime-1.1.3 [61]).

При проведении экспериментов были использованы те же настройки инструмента Avalanche, версии исследуемых проектов и начальные входные данные. Тем не менее полученные результаты незначительно отличаются от указанных в работе [25] из-за невозможности обеспечения полного соответствия среды выполнения и технических характеристик вычислительного устройства. Для получения указанных ниже результатов использовался компьютер Intel Core i5-2500 с 8 гигабайтами оперативной памяти.

В таблице 3 приведены значения составляющих α, полученные по результатам запусков и потенциальное ускорение, определяемое по закону Амдала.

Таблица 3: Оценки повышения производительности при использовании параллельной схемы

Программа	Значение α_1	Значение α_2	Значение α	Максимальное ускорение		
				4 потока	16 потоков	64 потока
cjpeg	0.08	0.0241	0.1041	3.05x	6.24x	8.46x
mpeg2dec	0.03	0.0066	0.0366	3.6x	10.33x	19.36x
mpeg3dump	0.0783	0.0133	0.0916	3.14x	6.74x	9.45x
swfdump	0.031	0.0084	0.0392	3.58x	10.07x	18.45x
qtdump	0.0364	0.0547	0.0911	3.14x	6.76x	9.5x

3.7 Результаты практических экспериментов

В таблице 4 приведены результаты применения инструмента Avalanche для обработки целевых программ при использовании стандартной схемы и при использовании нескольких потоков выполнения.

Таблица 4: Сравнение результатов обычного и параллельного режимов анализа

Программа	Итерации анализа	Запуски STP	Запуски Covgrind	Обнаруженные дефекты	Наборы данных для воспроизведения дефектов
cjpeg (1 поток)	658	26065	4744	3	35
cjpeg (4 потока)	1617	85826	14131	4	191
mpeg2dec (1 поток)	215	21664	21885	0	0
mpeg2dec (4 потока)	669	66871	65490	0	0
mpeg3dump (1 поток)	266	22122	7169	2	27
mpeg3dump (4 потока)	869	67559	20796	2	76
swfdump (1 поток)	273	25638	21883	3	4021
swfdump (4 потока)	891	84947	71429	4	16205
qtdump (1 поток)	217	9578	3059	2	367
qtdump (4 потока)	450	22088	9243	2	1176

На рассмотренных проектах при использовании 4 параллельных потоков выполнения удалось добиться увеличения количества пройденных путей выполнения и обнаружения дополнительного количества критических дефектов. Следует отметить, что проведение анализа программ с помощью инструментов

автоматического обхода путей выполнения крайне тяжело оценивать с позиции производительности. Обработка отдельных путей выполнения может занимать различное время даже для близких путей и порождать различное количество новых путей выполнения. Для сравнения теоретических и практических оценок увеличения производительности анализа будем использовать количество запусков STP и covgrind. Данный выбор может быть обусловлен следующими факторами:

- Схема параллельной работы нацелена на выделение дополнительных вычислительных узлов непосредственно для выполнения запусков STP и covgrind.
- Алгоритм выбора наборов данных для итераций анализа учитывает глубину точки ветвления, соответствующей набору данных (см. раздел 1.4) таким образом, чтобы избежать излишне интенсивного увеличения размеров трасс ограничений и увеличения времени обработки индивидуальных трасс модулем STP.
- Близкие по значениям входных данных пути выполнения приводят к незначительным отклонениям времени работы модуля covgrind при замере покрытия программы.

Графики сравнения фактического ускорения и оценки ускорения исходя из таблицы 4 представлены на рисунках 22-24.

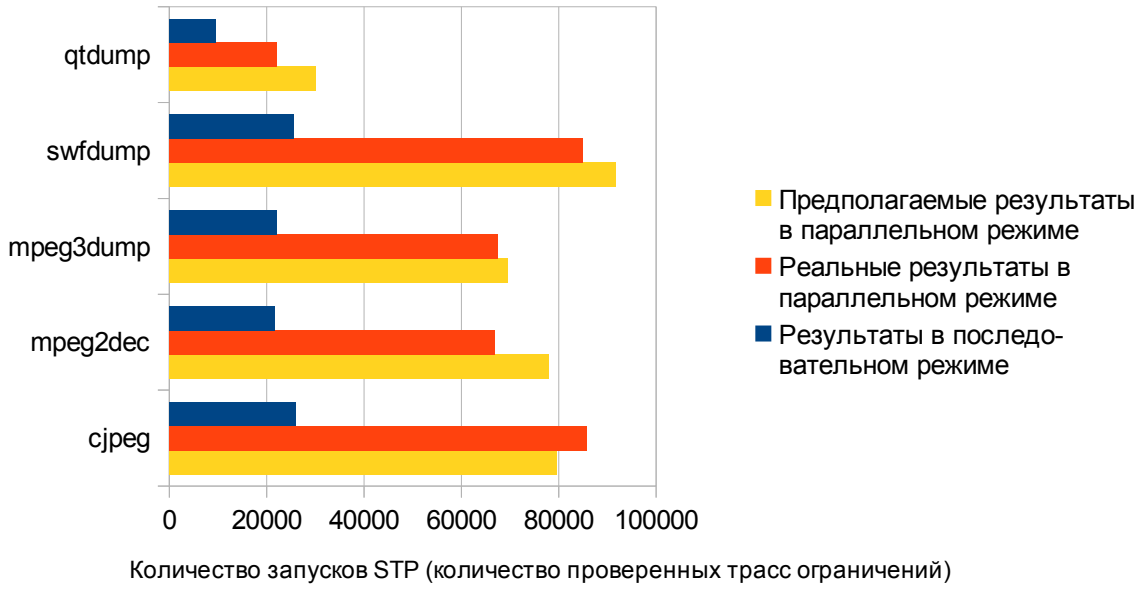


Рисунок 22: Оценка эффективности параллельной схемы по обработке трасс ограничений

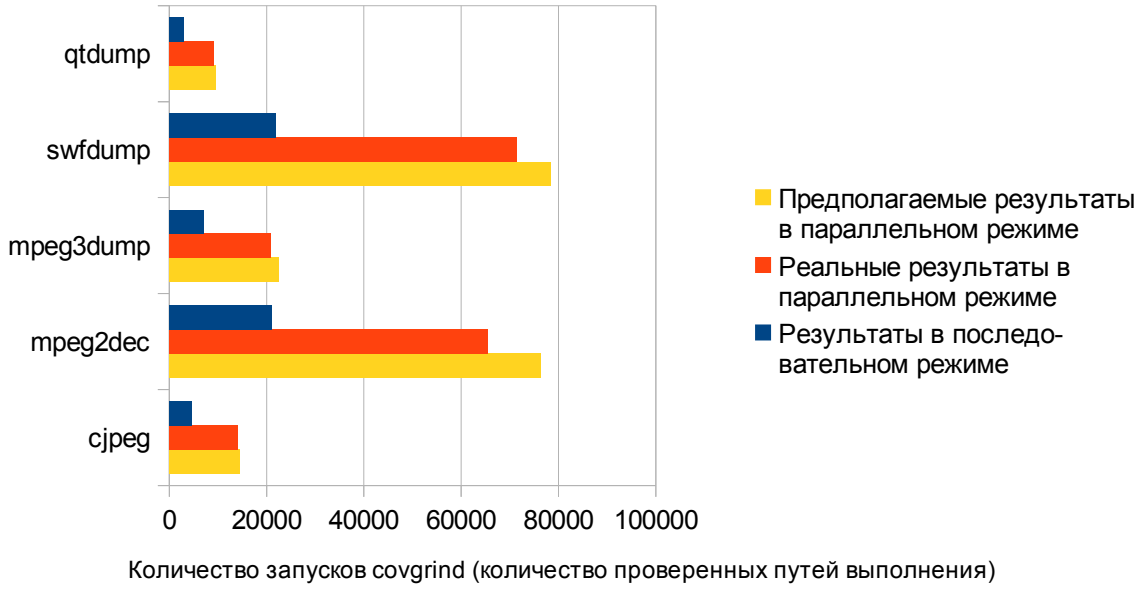


Рисунок 23: Оценка эффективности параллельной схемы по путям выполнения

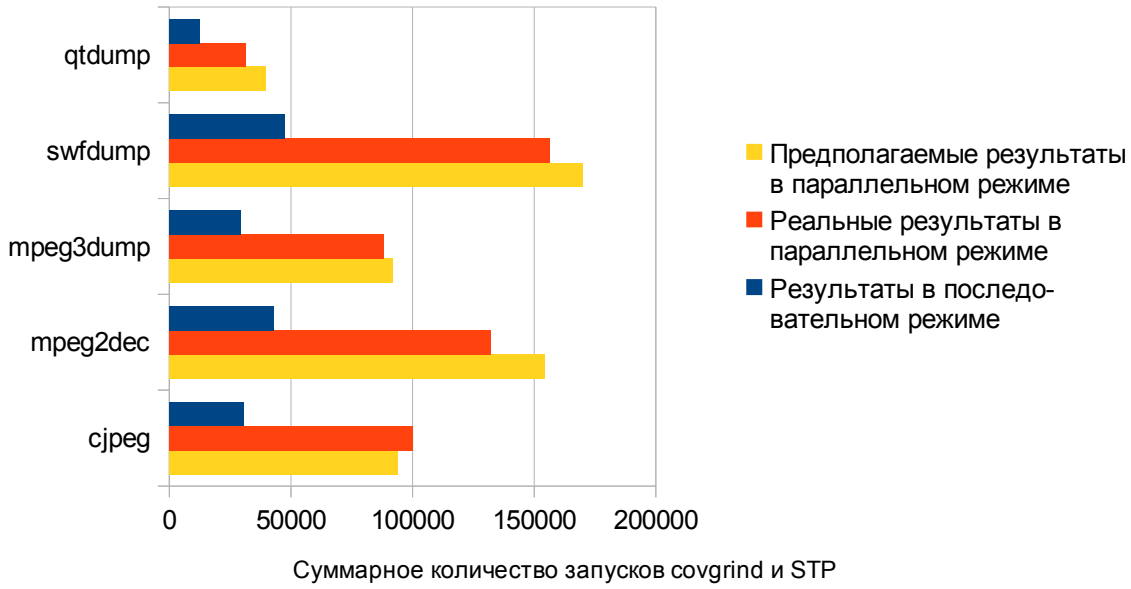


Рисунок 24: Общая оценка эффективности параллельной схемы

Полученное фактическое ускорение близко к ожидаемому.

Дефекты, обнаруженные на проектах, включают следующие:

- Аварийное завершение программы из-за попытки выполнить разыменование нулевого указателя (mpeg3dump, swfdump).
- Зацикливание программы из-за попытки считать некорректный объём данных (cjpeg, qtdump).
- Аварийное завершение программы по сигналу SIGABRT из-за нарушения условия в программной конструкции assert (qtdump).
- Аварийное завершение программы из-за попытки выполнить деление на ноль (cjpeg).

3.8 Выводе к главе 3

Глава 3 содержит подробное описание решения задачи 3, поставленной во вводной части работы. Описание практических экспериментов в разделах 3.6, 3.7 освещает задачу 5, поставленную в вводной части работы.

В главе рассмотрены возможности повышения эффективности анализа программе на основе автоматического обхода путей выполнения за счёт более

оптимального использования вычислительных ресурсов. Исходная реализация инструмента Avalanche, используемого в качестве базового в рамках данной работы, не учитывает особенности доступных вычислительных ресурсов, что делает актуальным задачу расширения функциональности инструмента.

Рассмотренные схемы параллельного и распределенного анализа, реализованные в инструментах SAGE, KLEE, Cloud9 и PEX, учитывают индивидуальные особенности подходов к проведению обхода путей выполнения, лежащих в основе данных инструментов. В связи с этим, прямое внедрение этих схем в инструмент Avalanche позволит получить ограниченный прирост производительности.

Предлагаемая схема параллельной обработки путей выполнения, описанная в разделах 3.4-3.5, учитывает особенности потоков данных и подзадач анализа, проводимого инструментом Avalanche. В рамках данной схемы рассматривается модификация управляющего модуля инструмента Avalanche, позволяющая проводить работу с параллельными вычислителями, выполняющими задания из банка заданий, строящегося на каждой итерации анализа.

Практическая реализация данной схемы включает созданный в рамках работы управляющий модуль инструмента Avalanche расширенной функциональности. Применение разработанной схемы на практике для анализа ряда проектов с открытым исходным кодом, позволило увеличить эффективность анализа на вычислительном устройстве многоядерной архитектуры по сравнению со стандартной версией инструмента Avalanche. Было зафиксировано ускорение обхода путей выполнения и обнаружены дополнительные дефекты, приводящие к аварийному завершению программ.

Глава 4. Использование статической инструментации исполняемого кода

4.1 Обработка кода программ в динамическом анализе

Рассмотрим подробнее особенности методов, используемых для обработки программы с целью извлечения данных о её выполнении, применительно к задаче автоматического обхода путей выполнения. Краткий обзор характеристик данных методов, используемых в инструментах анализа, был дан в главе 1.

Применение систем эмуляции и виртуальных машин, осуществляющих интерпретацию кода программы в некотором внутреннем представлении, позволяет проводить исследование программ с учётом большого количества аспектов времени выполнения. Информация об использовании программой ресурсов, конкретные значения всех элементов внутреннего состояния программы и другие данные доступны среде исполнения напрямую. При анализе программы осуществляется обработка полной трассы выполнения программы, включающей инструкции кода исполняемых файлов программ, динамических библиотек и динамически генерируемого кода. При полносистемной эмуляции существует возможность отслеживать взаимодействие программы с другими процессами.

К сожалению, использование систем эмуляции вносит высокие накладные расходы — помимо затрат на создание трасс и обработку символьных переменных осуществляются базовые действия по обработке инструкций, например динамическая трансляция инструкций анализируемой платформы в некоторое внутреннее представление системы эмуляции.

4.1.1 Инструментация кода программ

Применение динамической инструментации обеспечивает меньшую степень контроля за выполнением программы и вносит побочные эффекты в непосредственное выполнение, т. к. среды инструментации стандартно выполняются в рамках того же процесса, что и целевые программы. В то же время

без потери полноты анализируемого кода снижаются накладные расходы по сравнению с использованием сред эмуляции.

Инструментация исходного кода приводит к минимальным расходам на извлечение данных во время выполнения программы и может производиться независимо от него выполнения (например, на более мощном вычислительном устройстве). Влияние указанных факторов является в значительной степени важным для автоматического обхода путей выполнения. При проведении подобного обхода осуществляется повторное исполнение кода программы. В рамках систем эмуляции разбор инструкций осуществляется постоянно и вносит замедление как в режиме offline, так и в режиме online. Динамическая инструментация исполняемого кода в режиме offline значительно менее эффективна, чем предварительная статическая (целевая программа на каждой итерации запускается, разбирается и модифицируется заново), а в режиме online постоянные переключения состояния будут исчерпывать возможности кэша системы динамической инструментации, что будет приводить к повторной инструментации ранее обработанных блоков инструкций.

В то же время, статическая инструментация имеет крайне важную особенность — только та часть кода, которая была обработана с целью внедрения дополнительного кода, будет осуществлять генерацию трассы и обработку символьных данных. Это означает, что некоторые области кода, который будет реально выполняться, могут быть не учтены в процессе анализа. Данный эффект крайне близок к влиянию метода частичного анализа, рассмотренного в главе 2.

Рассмотрим возможности проведения статической инструментации исполняемого кода. Данный метод по своим свойствам наиболее близок к инструментации исходного кода, однако имеет более широкую область применения — в том случае, когда исходный код не доступен или невозможно повторить процесс компиляции и сборки программы, инструментация исходного кода невозможна. В то же время разбор исполняемого кода является более тяжелой задачей, чем разбор исходного кода; итоговое преобразование

исполняемого кода может быть менее точным, чем при проведении модификации исходного кода — работа с исходным кодом позволяет пользоваться информации о высокоуровневом представлении программы и перекладывать некоторые задачи на систему сборки, в то время как исполняемый файл может быть обработан с целью исключением информации, не нужной во время выполнения, или защищён средствами обфускации кода.

4.1.2 Эффективность применения статической инструментации исполняемого кода

Итак, статическая инструментация исполняемого кода не требует наличия исходного кода и позволяет добиться меньших накладных расходов по сравнению с методами обработки во время выполнения. Актуальность применения её зависит от рассмотренной выше полноты покрываемого кода. Рассмотрим следующие крайние случаи:

- Активное применение динамически подключаемого кода для обработки входных данных при применении статической инструментации исполняемого кода приведёт к потере фрагментов трассы ограничений и не позволит получить новые наборы входных данных.
- Активное применение динамически подключаемого кода, который заведомо не осуществляет обработку входных данных, приведёт к накладным расходам систем динамической инструментации, которые не будут влиять на точность и полноту анализа, тем самым замедляя его.

Из этого следует, что статическая инструментация может оказать как положительное влияние на производительность анализа программы, так и отрицательное влияние на точность анализа. В общем случае, эффективность предварительной инструментации тем выше, чем более точно инструментированный код можно соотнести с исполняемым кодом на подавляющем большинстве путей выполнения.

4.2 Обзор средств инструментации исполняемого кода

Для большинства существующих систем инструментации центральным принципом является предоставление возможностей по проведению настраиваемой обработки целевых программ — блоки исполняемого кода, в которых производится модификация или внедрение дополнительного кода, а также функциональная логика необходимых изменений задаются разработчиком модуля инструментации. Основными факторами, ограничивающими область применимости систем инструментации, являются следующие:

- Поддерживаемые машинные архитектуры и специфические наборы инструкций для данных архитектур;
- Поддерживаемые форматы организации файлов исполняемого кода;
- Предоставляемая разметка точек и блоков инструментации;
- Тип организации инструментационного кода;
- Ограничения, накладываемые на целевые файлы исполняемого кода.

4.2.1 Ранние системы статической инструментации

Ранние системы инструментации создавались для специфических архитектур и технологий и в основном не применимы для использования на данный момент из-за отсутствия поддержки со стороны авторов или устаревания целевых платформ. Тем не менее, данные проекты представляют значительный интерес из-за влияния на развитие области.

Система ATOM [62] была разработана для архитектуры DEC в связке с системой управления и организации исполняемого кода OM [63]. Данная система одной из первых предоставляла пользователю возможности описания логики инструментации, в то время как непосредственно процесс инструментации выполнялся внутренним модулем ATOM. Дополнительно пользователю необходимо было осуществить разметку файлов целевой программы для обозначения точек вставки дополнительного кода.

Авторы системы EEL [64] сфокусировали внимание на возможности

использования структурных элементов исполняемого кода целевой программы (инструкций, блоков инструкций, функций) в качестве точек инструментации. Это позволило значительно упростить процесс разметки целевого кода и спецификации логики модуля инструментации. Дополнительно, подход, используемый в системе EEL, обеспечивал гибкую структуру для поддержки различных архитектур — кодирование и декодирование целевого кода и генерацию инструментационного кода осуществлял специальный модуль, реализующий некоторый общий интерфейс.

Среди систем, разработанных под целевые архитектуры, широко используемые в данный момент, можно выделить BIRD [65] (Windows/x86), BitRaker Anvil [66] (Linux/ARM), LOPI [67] (Linux/x86).

Подход к инструментации в системе BIRD заключался во вставке инструкций вызова внешних функций в исполняемый код целевой программы и сборка инструментационного кода в виде динамической библиотеки, предоставляющей реализацию данных внешних функций. Дополнительно система поддерживала возможность вставки инструкций прерывания, позволяющая осуществлять выполнение инструментационного кода при обработке прерывания во время работы программы.

Система BitRaker Anvil предоставляла пользователю стандартные возможности модификации исполняемого кода программы путём вставки инструкций вызова внешней библиотеки, реализующей инструментационный код. Важным отличием системы являлась нацеленность на проведение анализа программы на основе использования сред эмуляции для выполнения кода архитектуры ARM на устройствах архитектуры x86 в процессе разработки. В рамках данного режима анализа появлялась возможность выполнять инструментационный код не в рамках целевой программы, а в рамках самой среды эмуляции. Применение данного режима позволяло снизить накладные расходы на выполнение инструментационного кода.

В рамках инструмента LOPI был разработан метод инструментации точек

входа и выхода из процедур кода с учётом соглашений о вызове, задаваемых стандартным двоичным интерфейсом приложения, используемым компилятором gcc для целевой архитектуры x86. Инструментатор LOPI использует известную структуру блока инструкций в начале функции для организации эффективного перехода на блок обёртки, внутри которого производится выполнение инструментационного кода напрямую или с помощью вызова функции из внешней библиотеки. Инструментация проводится таким образом, чтобы один блок обёртки мог бы использоваться сразу в нескольких точках инструментации, сокращая тем самым степень увеличения объёма файлов целевой программы и повышая эффективность работы с процессорным кэшем инструкций.

4.2.2 Современные системы статической инструментации

Современные системы инструментации активно используются для разработки отдельных модулей или полноценных инструментов анализа. Существует несколько направлений, разработки в которых интегрируются в системы инструментации для повышения их точности, эффективности и гибкости:

- Развитие интерфейса инструментации, доступного пользователю для описания логики необходимого модуля инструментации.
- Поддержка большого количества целевых платформ и разработка специфических оптимизаций для повышения эффективности работы инструментов на данных платформах.
- Повышение точности алгоритмов декодирования фрагментов данных, управляющих структур и кода целевых программ и разработка алгоритмов извлечения дополнительных данных о связях и особенностях объектов целевых программ для использования инструментом анализа.
- Оптимизация генерируемого инструментационного кода.

Среди наиболее значимых инструментов, предоставляющих возможности проведения статической инструментации исполняемого кода программы можно выделить MAQAO [68,69] (Linux/x86-64), PEBIL [70] (Linux/x86,x86-64), SecondWrite [18] (Linux/x86,ARM) и Dyninst [71] (Linux/x86,x86-64,ppc,ppc64).

Система MAQAO изначально была разработана для разбора исполняемого кода и внедрения модулей регистрации событий для профилирования и оценки эффективности работы компонентов программы. В дальнейшем был разработан язык спецификации точек инструментации MIL [69] и модуль генерации инструментационного кода для поддержки возможности разработки дополнительных инструментов анализа. Важной особенностью системы MAQAO является высокая точность и корректность работы с программами, активно использующими параллельные вычисления — именно для анализа подобных программ было разработано ядро системы.

Система PEBIL производит разбор исполняемых файлов целевой программы, модификацию секций кода и управляющих структур с целью подготовки к инструментации и добавляет инструментационный код в виде дополнительной секции. Блоки инструкций в точках инструментации заменяются на инструкции перехода в добавленные секции. Система PEBIL поддерживает генерацию инструментационного кода в виде внешней динамической библиотеки или коротких блоков инструкций, не требующих подключения дополнительных библиотек. В системе используется двухпроходный декодер инструкций и модуль оптимизации, осуществляющий минимизацию количества инструкций, не относящихся к логике инструмента анализа, но необходимых для корректной передачи управления на инструментационный код.

Инструмент Dyninst предоставляет широкий интерфейс описания инструментационного кода и спецификаций точек инструментации. Дополнительно при обработке исполняемого кода программ отдельные модули Dyninst осуществляют статический анализ с целью построения модели исполняемого кода, включающей информацию о графе потока управления, связях между базовыми блоками кода и т. д. Эта информация может быть использована при создании инструментационного кода и автоматизации выбора точек инструментации. Инструмент Dyninst позволяет описывать инструментационный код на основе интерфейса instructionAPI и в виде исходного кода на языке

спецификаций dynC, являющегося подмножеством языка программирования C.

Инструмент SecondWrite позволяет проводить статическую инструментацию исполняемого кода на основе существующей компиляторной инфраструктуры проекта LLVM. Инструментация файлов программы проводится в три этапа: на первом этапе исполняемый код автоматически трансформируется во внутреннее представление инфраструктуры LLVM, на втором этапе осуществляется непосредственная модификация внутреннего представления с целью добавления нового кода и итоговой оптимизации, на третьем этапе модифицированный код во внутреннем представлении автоматически трансформируется обратно в инструкции целевой процессорной архитектуры. Встроенные возможности LLVM по генерации и оптимизации кода покрывают большое количество потенциальных задач анализа программ; ключевым местом в схеме работы инструмента SecondWrite является высокая точность трансформации кода на первом этапе.

4.2.3 Современные системы динамической инструментации

Среди наиболее популярных и широко используемых в настоящее время систем динамической инструментации исполняемого кода можно отметить Valgrind [72], Pin [27], DynamoRIO [73] и Dyninst.

Инструменты Valgrind, Pin и DynamoRIO базируются на следующей модели обработки программы:

- инструменты образуют промежуточную среду исполнения, осуществляя перехват блоков инструкций перед передачей их на выполнение процессору;
- производится разбор блоков инструкций и их модификация согласно логике инструментации, задаваемой пользователем;
- модифицированные блоки передаются на выполнение процессору; дополнительно осуществляется сохранение их во внутренний кэш, чтобы избежать затрат на повторную инструментацию;

Описание инструментационного кода для данных систем проводится

пользователем на основе программных интерфейсов.

Система Dyninst позволяет осуществлять обработку исполняемого кода во время выполнения программы таким же образом, как и рассмотренную выше статическую инструментацию. В код вставляются дополнительные инструкции перехвата контроля управления в необходимых точках инструментации и непосредственно инструментационный код. Единственным отличием от режима статической инструментации является тот факт, что модифицируется код в виртуальной памяти процесса программы, а не внутри отдельных файлов исполняемого кода.

Системы Pin, DynamoRIO и Dyninst предоставляют возможности подключения к программе в любой момент выполнения и модификации требуемой функциональности инструментации также во время выполнения программы. Это позволяет настраивать инструментационный код на особенности отдельных запусков с целью снижения накладных расходов.

4.2.4 Оценка применимости существующих систем инструментации

Рассмотренные выше системы статической инструментации исполняемого кода в общем случае могут быть использованы в рамках инструмента Avalanche для реализации модулей сбора трассы ограничений и оценки прироста покрытия. Однако ни одна из рассмотренных современных систем инструментации кроме системы SecondWrite не предоставляет поддержку процессорной архитектуры ARM (для платформ Android [74] и Tizen [75]), важность исследования которой была представлена при обсуждении актуальности данной работы. Система SecondWrite является проприетарной, что в значительной степени ограничивает возможности её применения.

На основе результатов обзора существующих инструментов было принято решение о создании нового программного средства, осуществляющего обработку исполняемых файлов формата ARM ELF, используемого в платформах Android/ARM и Tizen/ARM. Для данного средства были сформулированы следующие требования:

- поддержка возможности инструментации на уровне отдельных инструкций с извлечением параметров данных инструкций — данная функциональность необходима для внедрения в программу дополнительных инструкций с целью проведения символьного исполнения;
- поддержка пользовательских спецификаций точек инструментации и инструментационного кода — данная функциональность позволит предоставить возможности создания инструментов анализа, сравнимые с рассмотренными существующими средствами, активно применяемыми на практике.

4.2.5 Особенности формата ARM ELF

Формат ARM ELF задаёт описание общей структуры содержимого файла исполняемого кода. В рамках данного формата файлы составляются из **секций** и заголовка, содержащего информацию о количестве секций, их именах и размерах.

Каждая секция формально может описывать произвольные данные, однако большинство инструментов компоновки программ и динамических загрузчиков осуществляющих размещение программ в виртуальной памяти ориентированы на использование набора секций определённой структуры, содержащих данные, необходимые для выполнения программы.

В рамках формата ELF для данных секций определены относительные виртуальные адреса, по которым динамический загрузчик размещает их в рамках блока памяти, выделенной для программы. Эти адреса могут учитываться внутренними данными секции. Секции объединяются в **сегменты** для более простой обработки их динамическими загрузчиками.

В рамках формата ELF секции можно добавлять, удалять, менять местами, расширять и сужать. В то время как отражение этих изменений в заголовке не составляет труда, полученный файл может оказаться некорректным — при попытке загрузить данный файл в виртуальную память и выполнить его код произойдет сбой, фиксируемый загрузчиком. Для того, чтобы устранить эти сбои, необходимо обновить всю информацию о сегментах и виртуальных адресах. Так,

например, если секция B следовала в сегменте непосредственно за секцией A (то есть имела виртуальный адрес, равный сумме виртуального адреса секции A и размера секции A), расширение секции A приведёт к наложению данных двух секций при загрузке файла в память.

К сожалению, изменение виртуальных адресов секций в свою очередь приведёт к нарушению корректности данных внутри них, если эти данные учитывают значения адресов. Рассмотрим наиболее важные с точки зрения инструментации кода секции и отметим возможности изменения их структуры и виртуальных адресов:

- .text – секция, содержащая непосредственный исполняемый код программы. Внутренние модификации и изменение виртуального адреса данной секции крайне сложны. Данная секция может иметь прямые ссылки в секции .rodata, .got, .plt и дополнительные секции данных.
- .dynsym, .dynstr, .symtab, .strtab – секции, хранящие информацию об именах символов, используемых другими секциями. Перемещение и модификация требует минимальной коррекции.
- .plt, .got, .rel.plt — секции, содержащие записи о внешних зависимостях. Так как на секции .got и .plt существуют ссылки из секции .text, возможности изменения их виртуальных адресов и внутренней модификации крайне ограничены.
- .dynamic – секция, хранящая информацию о виртуальных адресах и размерах других секций, для реализации быстрого доступа для динамического загрузчика. Изменение данных других секций необходимо отразить в этой секции. Дополнительно .dynamic содержит данные о библиотеках, содержащих внешние зависимости. Изменение виртуального адреса и расширение самой секции требует минимальной коррекции.

4.3 Предлагаемый метод инструментации

Предлагаемый метод инструментации состоит из следующих шагов:

1. Разбор файлов исполняемого кода и выделение всех конкретных точек в этих файлах, удовлетворяющих спецификациям пользователя.
2. Параметризация блоков инструментационного кода, описанного пользователем в виде исходного кода на языке C, конкретными значениями, соответствующими выделенным точкам в файлах исполняемого кода.
3. Компиляция параметризованных блоков инструментационного кода в объектные файлы формата ARM ELF по числу файлов целевой программы.
4. Присоединение секций кода и данных каждого объектного файла к соответствующему файлу исполняемого кода целевой программы.
5. Расширение управляющих секций итогового файла для поддержки внешних зависимостей инструментационного кода и корректировка значений в элементах данных секций.
6. Модификация секции кода итогового файла с целью внедрения инструкций передачи управления на инструментационный код.

4.3.1 Язык спецификаций

Язык спецификаций позволяет описывать множество блоков следующего формата, представленном на рисунке 25:

```
  <t><Тип></t>
[ <s><Целевой набор инструкций></s> ]
[ <f+><Имя функции></f> ]*
[ <f-><Имя функции></f> ]*
  <c><Блок кода></c>
[ <dep>f:<Имя внешней функции>:<Имя библиотеки></dep> ]*
[ <dep>d:<Имя внешней переменной>:<Имя библиотеки></dep> ]*
```

Рисунок 25: Язык спецификаций инструментации

Поля блоков имеют следующие значения:

- <Тип> задаёт тип точки инструментации. Поддерживается возможность инструментации инструкций обработки данных (t_data_proc), вызова функций (t_call, t_call_after), инструкций работы с памятью (t_store, t_load, t_push, t_pull), начальных инструкций базового блока и функции (t_bb, t_func_entry), инструкций возврата из функции (t_func_exit), инструкций,

выполняющихся условно (t_cond).

- Поле <Целевой набор инструкций> позволяет задавать специализированную инструментацию отдельно для точек, содержащих инструкции набора ARM или набора Thumb-2.
- Поля <Имя функции> позволяют задавать фильтры на точки инструментации по внутренним функциям кода.
- Поле <Блок кода> позволяет описывать инструментационный код на языке C. Помимо обычных конструкций языка в инструментационном коде можно использовать специальные маркеры, которые заменяются непосредственными константными значениями или блоками инструкций для доступа к параметрам точки инструментации.
- Последние два поля позволяют задавать описание внешних зависимостей инструментационного кода, информацию о которых необходимо добавить при инструментации для устранения ошибок времени выполнения.

Рассмотрим пример спецификаций и их обработки для одной точки инструментации (рис. 26).

```
<t>t_load</t>
<f+>foo</f>
<c>
  int addr;
  ___get_addr___(addr);
  printf("load at %x of ___size___\n", addr);
</c>
<dep>f:printf:libc.so</dep>
```

```
foo:
 ...
 0x1000:
  ldr r1, [r2, #10]
 ...
```

```
//начало блока для инструкции по 0x1000
  int addr;
//встроенный блок ассемблера:
  asm("mov r12, #10\n\t"
      "add r12, r2\n\t"
      "mov %[dst], r12\n\t"
      : [dst] "=r" (addr)
      :: "r12");
  printf("load at %x of 4\n", addr);
```

Рисунок 26: Пример создания параметризованного блока инструментационного кода

Данная спецификация покрывает все инструкции загрузки из памяти, содержащиеся в коде функции с именем foo; инструментационный код печатает сообщение, указывающее размер считанной памяти и эффективный адрес ячейки, откуда считываются данные. Эти два параметра задаются в коде специальными маркерами ___size___ и ___get_addr___(var), обрабатываемыми системой инструментации. Обработка ___size___ заключается в подстановке константного значения, так как это значение известно для инструкции ldr. Обработка ___get_addr___ включает в себя вставку последовательности инструкций для расчёта эффективного адреса, так как это значение может быть посчитано только во время выполнения программы. Последовательность инструкций задаётся встроенным блоком ассемблерного кода, записывающего в переменную addr сумму значения регистра r2 (базы эффективного адреса) и константного смещения 10. Для записи используется промежуточный регистр r12. В процессе инструментации в блок кода будут автоматически добавлены инструкции сохранения и восстановления данного регистра с целью поддержания

корректности выполнения программы.

Отметим, что каждый блок инструментационного кода обрамляется блоками инструкций, не имеющих эффекта. Эти блоки непосредственно используются для размещения инструкций сохранения и восстановления регистров и стека.

Печать сообщения производится с помощью стандартной функции printf, которая помечается в спецификации как внешняя. При обработке зависимостей предполагается добавление записи, указывающей загрузчику операционной системы провести поиск реализации данной функции в библиотеке libc.so.

4.3.2 Разбор файлов целевой программы и генерация кода

Предлагаемый метод не требует проведения специализированного разбора кода целевой программы и совместим с существующими алгоритмами дизассемблирования. Определение параметров отдельных инструкций производится согласно стандартным спецификациям процессорной архитектуры ARM.

Аналогично данный метод может использовать любые инструменты компиляции кода, поддерживающие создание объектных файлов в формате ARM ELF на основе кода на языке C с использованием ассемблерных вставок.

4.3.3 Модификация структуры файлов целевой программы

Перед непосредственным проведением инструментации сгенерированный инструментационный код необходимо добавить в файлы целевой программы. Рассмотрим два различных случая:

1. Все внешние зависимости, используемые инструментационным кодом, уже описаны в файле, который подвергается инструментации.
2. Хотя бы одна внешняя зависимость, используемая инструментационным кодом, не описана в инструментируемом файле.

Для первого случая модификация структуры инструментируемого файла является тривиальной. Так как формат ARM ELF не задаёт ограничений на количество секций, извлечём секции .text и .rodata из объектного файла, сгенерированного по инструментационному коду и добавим их как независимые

секции (рис. 27). Здесь и далее будем опускать на рисунках фрагменты ELF, относящиеся к заголовочной информации.

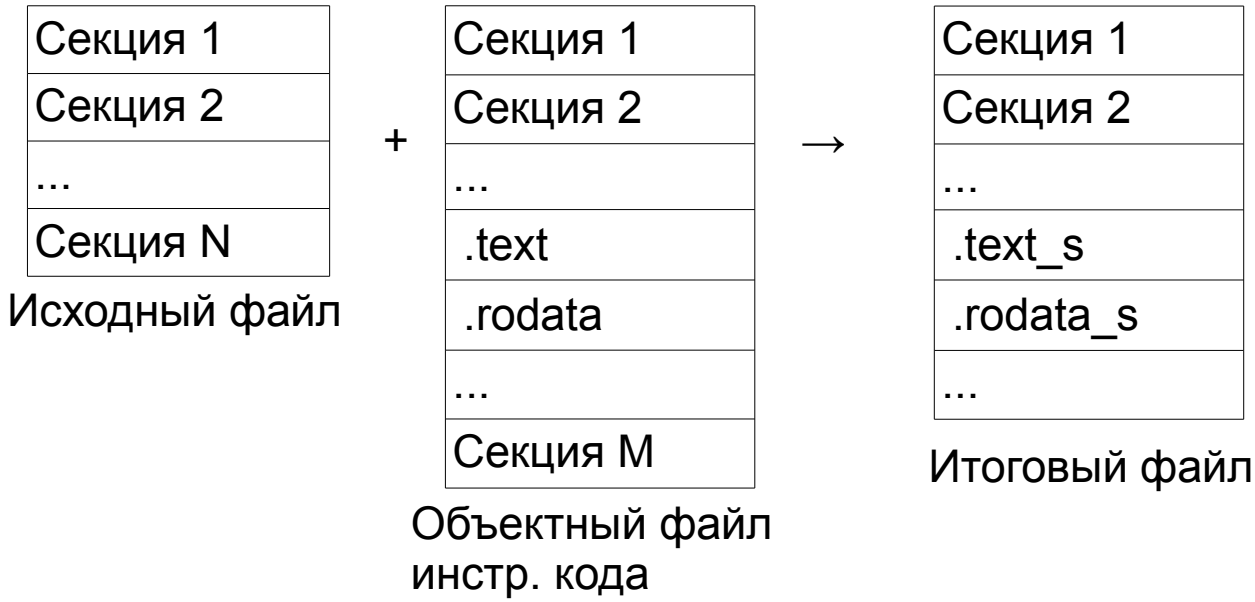


Рисунок 27: Модификация файлов ARM ELF

При обработке второго случая требуется намного больше изменений. Как и в первом случае, секции кода и данных инструментационного объектного файла добавляются как независимые секции. Каждая внешняя зависимость должна иметь записи в следующих рассмотренных ранее секциях: .dynsym, .dynstr, .symtab, .strtab, .plt, .got, .rel.plt. Для обеспечения корректности работы зависимостей инструментационного кода каждую из этих секций необходимо расширить. В зависимости от распределения секций по сегментам необходимо осуществить перестановку секций и последующее обновление записей в секции .dynamic. Модификация виртуальных адресов секций .plt и .got строго ограничена и привязана к изменению их размера, чтобы сохранить корректность ссылок на элементы этих секций из .text.

Алгоритм перестановки учитывает исходную структуру секций файла, определяет подмножество секций *Immovable*, виртуальные адреса которых необходимо сохранить, и определяет подмножество секций *Movable*, виртуальные адреса которых можно свободно менять.

Размещение секций по сегментам можно представить как множество пар

{<$vaddr_i$,$vaddr_i$+$size_i$>}, где $vaddr_i$ – виртуальный адрес секции, $size_i$ – размер секции. Трансформация переводит данное множество в {<$vaddr_i$',$vaddr_i$'+$size_i$'>}, причём для секции *i* из множества Immovable $vaddr_i$'=$vaddr_i$. В случае невозможности такой модификации для некоторых секций из множества *Immovable* создаются добавочные секции, располагающиеся независимо по виртуальным адресам, но содержащие данные того же рода с учётом поправки на разницу виртуальных адресов.

На рисунке 28 представлен пример модификации файла ELF, показывающий масштаб требуемых изменений для отдельных секций.

[Nr]	Name	Addr	Size
[0]	NULL	0	0
[1]	.interp	10154	13
[2]	.note.ABI-tag	10168	20
[3]	.note.gnu.build-id	10188	24
[4]	.hash	101ac	30
[5]	.gnu.hash	101dc	34
[6]	**.dynsym**	10210	70
[7]	**.dynstr**	10280	4d
[8]	**.gnu.version**	102ce	e
[9]	**.gnu.version_r**	102dc	20
[10]	.rel.dyn	102fc	8
[11]	**.rel.plt**	10304	30
[12]	**.init**	10334	c
[13]	**.plt**	10340	5c
[14]	.text	1039c	1a0
[15]	.fini	1053c	8
[16]	.rodata	10544	4
[17]	.ARM.exidx	10548	8
[18]	.eh_frame	10550	4
[19]	.init_array	20f04	4
[20]	.fini_array	20f08	4
[21]	.jcr	20f0c	4
[22]	**.dynamic**	20f10	f0
[23]	.got	21000	28
[24]	.data	21028	8
[25]	.bss	21030	8
[36]	.shstrtab	0	170
[37]	**.symtab**	0	7a0
[38]	**.strtab**	0	3ae

[Nr]	Name	Addr	Size
[0]	NULL	0	0
[1]	.interp	10154	13
[2]	.note.ABI-tag	10168	20
[3]	.note.gnu.build-id	10188	24
[4]	.hash	101ac	30
[5]	.gnu.hash	101dc	34
[6]	**.dynsym**	10210	**80**
[7]	**.dynstr**	**1064c**	**65**
[8]	**.gnu.version**	**10290**	**10**
[9]	**.gnu.version_r**	**102ac**	**40**
[10]	.rel.dyn	102fc	8
[11]	**.init**	**102a0**	c
[12]	**.plt**	**10334**	**68**
[13]	.text	1039c	1a0
[14]	.fini	1053c	8
[15]	.rodata	10544	4
[16]	.ARM.exidx	10548	8
[17]	.eh_frame	10550	4
[18]	.init_array	20f04	4
[19]	.fini_array	20f08	4
[20]	.jcr	20f0c	4
[21]	**.dynamic**	**10554**	**f8**
[22]	.got	21000	28
[23]	.data	21028	8
[24]	.bss	21030	8
[35]	**.rodata_s**	**21590**	**1000**
[36]	**.text_s**	**2103c**	**554**
[37]	**.got2**	**21038**	**4**
[38]	**.rel.plt**	**23590**	**60**
[39]	.shstrtab	0	188
[40]	**.symtab**	0	**7d0**
[41]	**.strtab**	0	**3ae**

Рисунок 28: Пример модификации структуры секций файла

Секции .dynsym, .dynstr, .gnu.version (информация о версиях библиотек, предоставляющих внешние зависимости), .gnu.version_r (информация о распределении символов внешних зависимостей по библиотекам), .plt, .rel.plt,

.dynamic, .symtab, .strtab были расширены. Для секций .dynstr, .gnu.version, .gnu.version_r, .init (короткий блок кода для передачи управления на точку входа), .dynamic, .rel.plt были изменены виртуальные адреса. Секции .rodata_s и .text_s были добавлены из объектного файла инструментационного кода. Требуемые расширения секции .got были помещены во вспомогательную секцию .got2.

4.3.4 Инструментация кода файлов целевой программы

Результатом предыдущего шага является файл исполняемого кода, загрузка и использование которого не приведёт к ошибкам, фиксируемым динамическим загрузчиком, и будет полностью соответствовать выполнению оригинальной версии исполняемого файла.

Предыдущие изменения тем не менее позволяют подготовить файл к инструментации на основе простого алгоритма.

На вход данного шага инструментации подаётся файл целевой программы модифицированной структуры и объектный файл инструментационного кода, содержащий информацию о соответствии точек инструментации и отдельных параметризованных блоков. Для каждой точки инструментации, которая находится по смещению offset в секции .text (виртуальный адрес — *vaddr*) файла и которой соответствует блок кода по смещению *instr_offset* в секции .text_s (виртуальный адрес — *instr_vaddr*), проводятся следующие действия:

1. Инструкция *insn*, занимающая позицию *offset*, заменяется на инструкцию безусловного перехода на адрес i*nstr_offset*+*instr_vaddr*.
2. На позицию *instr_offset* в секции .text_s добавляются инструкции, записывающие регистры, изменяемые непосредственно инструментационным кодом, в стек. Это изменение не приводит к нарушению функциональности инструментационного кода, т. к. согласно пункту 4.3.4 туда были помещены инструкции, не имеющие эффектов.
3. В конец инструментационного блока вместо инструкций, не имеющих эффект, вставляются инструкции восстановления регистров, затем блок

инструкций *insn_mod*, затем инструкция безусловного перехода на адрес *offset*+*vaddr*+*size*, где *size* – размер инструкции *insn*.

4. Если выполнение исходной инструкции *insn* не зависит от значения регистра-счётчика инструкций, блок инструкций *insn_mod* включает только *insn*. В противном случае блок *insn_mod* включает несколько инструкций, выполнение которых эквивалентно выполнению *insn* (осуществляется компенсация различий в значении регистра-счётчика инструкций).

   Рисунок 29 иллюстрирует проведение шагов для одной точки инструментации.

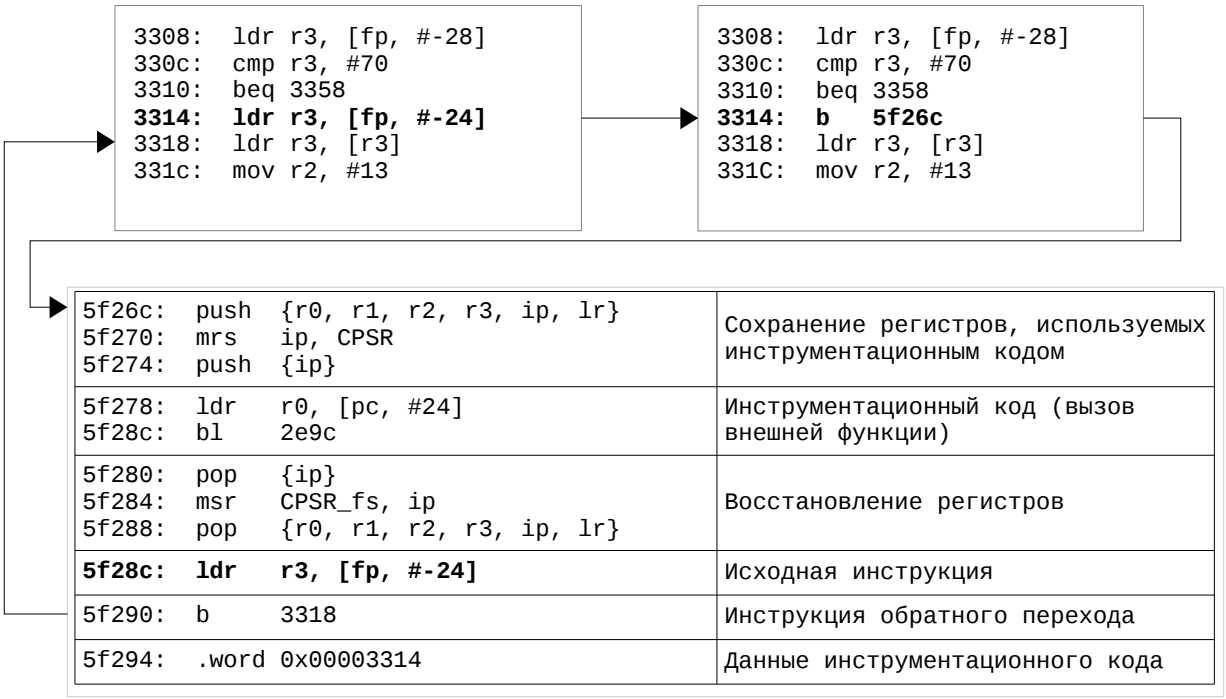


Рисунок 29: Пример проведения инструментации для одной точки

Дополнительные преобразования проводятся для следующих ситуаций:

- При замене инструкций в точке инструментации используются 32-битные инструкции перехода как для блоков ARM, так и для блоков Thumb-2. Если в точке инструментации находилась короткая 16-битная инструкция Thumb-2, производится сцепка - замена и перенос сразу двух инструкций согласно шагам 3 и 4 алгоритма, рассмотренного выше. Данное правило не

применяется, если замена пары инструкций приведёт к некорректной ситуации — например, если из другого фрагмента кода программы существует переход на смещение второй короткой инструкции (в этом случае при замене переход будет переводить управление на середину инструкции, а не начало).

- Если для смещения в коде существуют две и более точек инструментации или если точки инструментации покрывают несколько смежных инструкций, входящих в один базовый блок, соответствующие им блоки инструментации помечаются, как сцепленные. Инструкция перехода в секцию .text_s заменяет только инструкции первой точки инструментации, инструкция перехода из секции .text_s в секцию .text помещается только в последний блок, а все остальные переходы осуществляются напрямую между блоками.

4.4 Описание практической реализации предлагаемого метода

Реализация предлагаемого метода основывается на наборе существующих инструментальных систем для разбора и генерации кода.

Действия по обработке файлов формата ARM ELF, дизассемблированию кода и модификации структуры секций и сегментов осуществляются с помощью набора инструментов из пакета binutils [76], распространяемого свободно вместе с исходным кодом:

- Компонент binutils readelf осуществляет извлечение информации о имеющихся секциях, символах и зависимостях в файле ELF.
- Компонент binutils objdump осуществляет дизассемблирование кода на основе алгоритма прямого прохода; в реализацию инструмента были внесены изменения, позволяющие автоматически извлекать параметры инструкций, использовать которые можно при описании инструментационного кода. Дополнительно эти параметры используются

для оптимизации инструкций сохранения и восстановления регистров, внедряемых в инструментационный код, и для выделения инструкций, для которых необходимо провести трансформацию согласно пункту 4 алгоритма, рассмотренном в предыдущем разделе.

- Компонент binutils objcopy используется для модификации параметров секций при внедрении внешних зависимостей инструментационного кода и пересчёте таблицы сегментов.
- Компонент rewriter, реализованный в рамках данной работы на основе базовых библиотек работы с файлами ELF в составе binutils, используется для проведения непосредственной инструментации, описанной выше.
- Компонент addsymbol, реализованный в рамках данной работы, используется для автоматической модификации всех секций при внедрении дополнительных внешних зависимостей.

Для автоматической генерации объектных файлов по исходному инструментационному коду используется система компиляции и сборки кода gcc [77], что обеспечивает возможность применения широко используемых оптимизаций.

Схема взаимодействия указанных инструментов в рамках общего процесса инструментации представлена на рисунке 30.

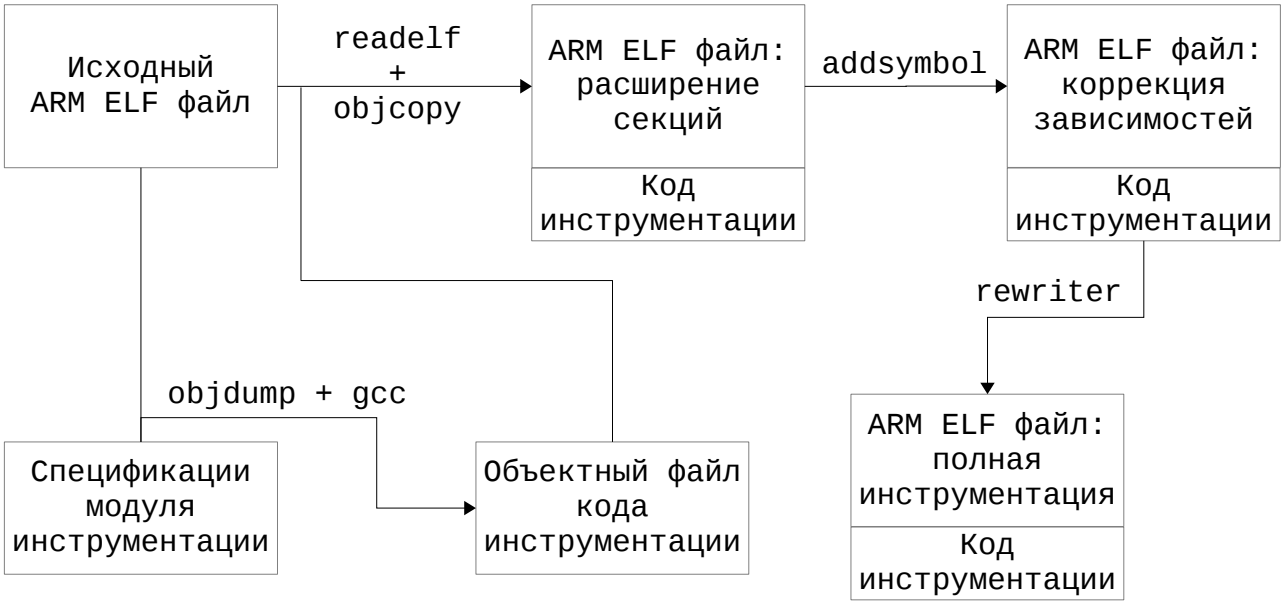


Рисунок 30: Схема взаимодействия компонентов системы инструментации

4.5 Применение статической инструментации в Avalanche

Стандартно модули инструмента Avalanche tracegrind и covgrind используют среду динамической инструментации Valgrind.

Реализация модуля tracegrind осуществляет обработку инструкций внутреннего представления VEX, используемую в Valgrind. В рамках данного внутреннего представления осуществляется активная работа с промежуточными переменными, в которые помещаются результаты отдельных унарных и бинарных операций обработки данных и чтения из памяти. Данное обстоятельство упрощает работу с символьными переменными, так как сложные инструкции, имеющие сразу несколько эффектов, разбиваются на простые операции внутреннего представления.

Реализация модуля tracegrind на основе статической инструментации предполагает работу непосредственно с инструкциями целевой архитектуры, что позволяет сократить количество записей в создаваемой трассе ограничений при увеличении средней длины записей. Дополнительно в модуле статической инструментации осуществляется запись трассы в компактном формате и применение отдельного модуля, трансформирующего данную трассу в формат,

обрабатываемый модулем STP. При проведении анализа с использованием удалённого устройства, более подробно рассмотренного в следующем разделе, появляется возможность получить малый выигрыш по скорости.

Реализация инструмента covgrind с помощью статической инструментации очень близка к исходной реализации на основе среды Valgrind. Данный модуль осуществляет отслеживание событий входа в базовые блоки и запись идентификаторов данных блоков в поддерживаемое им множество. В рамках динамической инструментации добавление блоков во множество осуществляется непосредственно при самой инструментации, а сам исполняемый код программы остаётся прежним. Это позволяет в значительной мере сократить количество операций добавления элементов во множество, выполняемых во время работы программы.

Для достижения такого же уровня оптимизации в реализации модуля covgrind на основе статической инструментации используется самомодифицирующийся инструментационный код, который выполняется только один раз для каждого базового блока. Данный код включает добавление идентификатора базового блока в необходимое множество, а затем восстанавливает инструкции в точках инструментации, замененные на инструкции перехода. При последующем выполнении базового блока передача управления на инструментационный код не производится.

4.6 Результаты практических экспериментов

В рамках практических экспериментов было исследовано подмножество проектов, рассмотренных в работе [25] с целью анализа с применением инструмента Avalanche.

Анализ проектов проводился в удалённом режиме с использованием компьютера Intel Core i5-2500 с 8 гигабайтами оперативной памяти и устройства Tizen PD-RQ целевой архитектуры ARM. Распределение работы компонентов инструмента Avalanche в удалённом режиме представлено на рисунке 31.

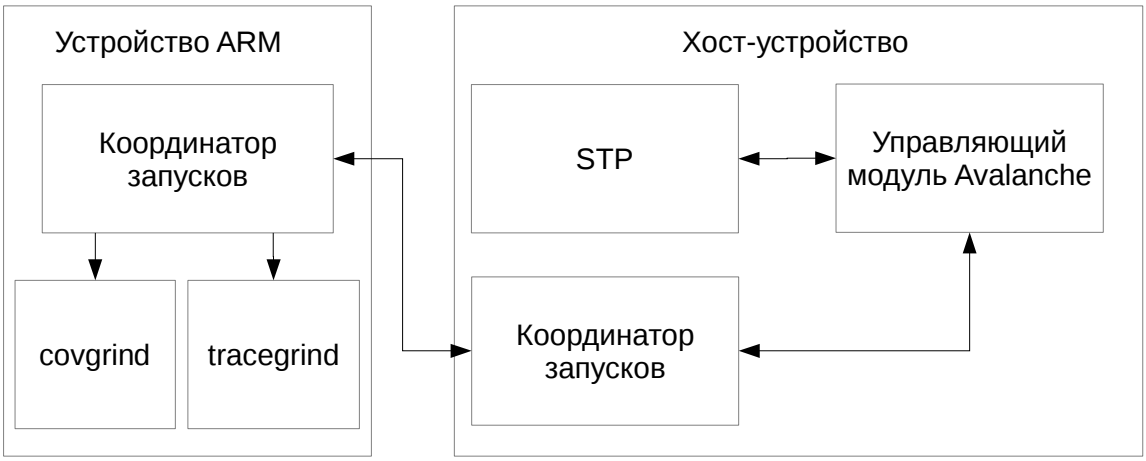


Рисунок 31: Взаимодействие компонентов Avalanche в распределенном режиме

Описание программ, для которых осуществлялись тестовые запуски, представлено в главе 3. Дополнительно была исследована программа djpeg из пакета jpeg-7, позволяющая проводить автоматическую трансформацию файлов формата JPEG в другие форматы графических данных.

Эксперименты проводились с использованием стандартных параметров инструмента Avalanche, рассмотренных в работе [25], - ограничением времени анализа двумя часами и ограничением глубины просмотра на каждой итерации в 100 точек ветвления. Результаты экспериментов приведены в таблице 5.

Таблица 5: Сравнение результатов анализа при использовании разных режимов инструментации

Программа	Итерации анализа	Пути выполнения	Уникальные дефекты	Наборы данных	Сбор трассы, с	Оценка покрытия, с
cjpeg (дин.)	208	4163	1	2	4.12	1.24
cjpeg (ст.)	2709	4009	1	64	0.33	0.32
djpeg (дин.)	217	3455	0	0	3.49	1.38
djpeg (ст.)	1451	36443	0	0	0.16	0.09
mpeg2dec (дин.)	57	4051	1	1	3.73	1.28
mpeg2dec (ст.)	2091	13459	1	92	0.21	0.08
mpeg3dump (дин.)	58	2637	2	4	10.23	1.58

mpeg3dump (ст.)	511	37528	2	10	0.60	0.10
swfdump (дин.)	174	3528	1	785	4.61	1.14
swfdump (ст.)	575	14759	4	3459	0.48	0.33
qtdump (дин.)	301	1592	2	7	6.01	2.75
qtdump (ст.)	4574	35402	3	24	0.35	0.11

Применение статической инструментации позволило сократить среднее время работы модулей tracegrind и covgrind (столбцы «Сбор трассы» и «Оценка покрытия» соответственно) и за ограниченное время осуществить обход большего числа путей выполнения и для отдельных проектов обнаружить дополнительные дефекты. Исключение составляет программа cjpeg, для которой отличия в работе реализаций модулей привели к альтернативному порядку обхода точек ветвления и попаданию в область программы большей глубины. Это значительно увеличило время обработки трасс модулем STP, что негативно сказалось на общей статистике анализа. Несмотря на это, при использовании статической инструментации был успешно обнаружен дефект, зафиксированный при анализе с динамической инструментацией.

Отметим, что указанные выше результаты для статической инструментации учитывают время на проведение непосредственной инструментации – среднее время работы модулей в последних двух столбцах рассчитывается как $\frac{ModuleTotal + StInstr}{Runs}$, где ModuleTotal – общее время работы модуля, StInstr – время проведения статической инструментации на основе спецификаций для модуля, Runs — количество отдельных запусков модуля.

Дефекты, обнаруженные на проектах, включают следующие:

- Аварийное завершение программы из-за попытки выполнить разыменование нулевого указателя (mpeg3dump, qtdump, swfdump).

- Зацикливание программы из-за попытки считать некорректный объём данных (cjpeg, qtdump, mpeg3dump).
- Аварийное завершение программы из-за попытки выполнить деление на ноль (mpeg2dec).

4.7 Выводы к главе 4

Глава 4 содержит подробное описание решения задачи 4, поставленной в вводной части работы. Описание практических экспериментов в разделе 4.6 освещает задачу 5, поставленную в вводной части работы.

В существующих инструментах, позволяющих проводить автоматический обход путей выполнения с использованием символьного исполнения, активно используются методы статической инструментации исходного кода и динамической инструментации исполняемого кода, а также средства эмуляции и виртуальные машины.

В главе рассмотрена возможность использования в рамках метода автоматического обхода путей выполнения статической инструментации исполняемого кода. Согласно положениям, изложенным в разделе 4.1, применение статической инструментации исполняемого кода позволит расширить область применимости метода по сравнению со статической инструментацией исходного кода и снизить объем накладных расходов по сравнению с динамической инструментацией исполняемого кода и применением эмуляторов и виртуальных машин.

Для исследования эффективности использования статической инструментации исполняемого кода был предложен и разработан метод проведения подобной инструментации (раздел 4.3).

Практическая реализация предлагаемого метода, рассмотренная в разделе 4.4, использует существующие программные средства из пакетов binutils и gcc для генерации исполняемого кода и базовой обработки файлов целевых программ, и новые программные средства, созданные в рамках данной работы, для

модификации кода целевых программ и обеспечения корректности инструментации.

На основе разработанного программного средства статической инструментации были созданы модули, замещающие компоненты covgrind и tracegrind инструмента Avalanche, проводящего анализ программ посредством автоматического обхода путей выполнения. Применение данных модулей на практике позволило повысить скорость анализа для ряда проектов с открытым исходным кодом. Выигрыш по скорости был достигнут без потери точности анализа (были обнаружены все дефекты, обнаруживаемые исходной версией инструмента) и позволил обнаружить дополнительные критические дефекты программ, приводящие к аварийному завершению.

Заключение

В диссертации:

- Проведён обзор методов анализа программ на основе автоматического обхода путей выполнения, существующих ограничений данных методов, и выбраны целевые направления оптимизации методов анализа. В рамках исследования одного из выбранных направлений проведён обзор подходов к проведению статической инструментации исполняемого кода.
- Предложен метод частичного анализа программ, позволяющий осуществлять автоматический обход подмножества путей выполнения программы, удовлетворяющего правилам, задаваемым пользователем. Программная реализация метода встроена в систему динамического анализа Avalanche.
- Предложена и реализована модифицированная схема взаимодействия компонентов инструмента Avalanche, позволяющая проводить параллельную обработку независимых путей выполнения анализируемых программ.
- Предложен метод статической инструментации исполняемого кода для извлечения трасс выполнения программы. Разработана программная система, предоставляющая возможности проведения настраиваемой статической инструментации кода для платформы ARM/Linux, и на основе разработанной системы созданы подключаемые модули для инструмента Avalanche, выполняющие обработку программ во время выполнения.
- Для предложенных и разработанных в рамках работы методов проведены практические исследования, свидетельствующие об эффективности их применения при проведении динамического анализа программ.

Список литературы

1. 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. — 2013. — т. 25. — с. 29-38.
2. Ермаков М.К., Вартанов С.П. Применение статической бинарной инструментации с целью проведения динамического анализа программ для платформы ARM // Труды Института системного программирования РАН. — 2015. — т. 27 (1). — с. 5-24.
3. Ермаков М.К. Проведение динамического анализа исполняемого кода формата ARM ELF на основе статического бинарного инструментирования // Научно-технические ведомости СПбГПУ. Информатика. Телекоммуникации. Управление — 2016 — Выпуск 1(236). — с. 108-117.
4. Ермаков М.К. Динамический анализ исполняемого кода в формате ELF на основе статической бинарной инструментации // Материалы межд. науч.-прак. конференции «Инструменты и методы анализа программ – 2015» — Санкт-Петербург, 14-16 ноября 2015. — СПб.:Изд-во Политехн. ун-та, 2015. — с. 14-21.
5. King, J. C. Symbolic execution and program testing / James C. King // Communications of the ACM. — 1976. — Vol. 19, no. 7. — P. 385– 394.
6. Boyer, R. S. SELECT—a formal system for testing and debugging programs by symbolic execution / Robert S. Boyer, Bernard Elspas, Karl N. Levitt // ACM SIGPLAN Notices. — 1975. — Vol. 10, no. 6. — P. 234–245.
7. Pasareanu, C.S. A survey of new trends in symbolic execution for software testing and analysis / Corina S. Pasareanu, Willem Visser // International Journal on Software Tools for Technology Transfer. — 2009. — Vol. 11, no. 4. — P. 339–353.

8. Cadar, C. Symbolic execution for software testing / Cristian Cadar, Koushik Sen // Communications of the ACM. — 2013. — Vol. 56, no. 2. — P. 82.
9. Unleashing Mayhem on binary code / Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, David Brumley // Proceedings - IEEE Symposium on Security and Privacy. — 2012. — P. 380–394.
10. Een, N. Temporal Induction by Incremental SAT Solving / Niklas Een, Niklas Sorensson // Electronic Notes in Theoretical Computer Science. — 2003. — Vol. 89, no. 4. — P. 543–560.
11. Godefroid, P. DART : Directed Automated Random Testing / Patrice Godefroid // Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation. — 2005. — P. 213-223.
12. Sen, K. CUTE : A Concolic Unit Testing Engine for C / Koushik Sen, Darko Marinov, Gul Agha // Program. — 2005. — Vol. 30. — P. 263–272.
13. Sen, K. {CUTE} and {jCUTE}: Concolic Unit Testing and Explicit Path {Model-Checking} Tools / Koushik Sen, Gul Agha // Computer Aided Verification. — 2006. — Vol. 4144. — P. 419–423.
14. Cadar, C. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs / Cristian Cadar, Daniel Dunbar, Dawson R. Engler // Proceedings of the 8th USENIX conference on Operating systems design and implementation. — 2008. — P. 209–224.
15. The LLVM Compiler Infrastructure Project. [Электронный ресурс] — Электрон. дан. — Режим доступа: http://llvm.org/. — Загл. с экрана — Англ.
16. EXE / Cristian Cadar, Vijay Ganesh, Peter M Pawlowski et al. // Proceedings of the 13th ACM conference on Computer and communications security - CCS ’06. — 2006. — Vol. 12. — P. 322.
17. Ganesh, V. A decision procedure for bit-vectors and arrays / Vijay Ganesh, David L. Dill // Computer Aided Verification. — 2007. — P. 524–536.
18. A compiler-level intermediate representation based binary analysis and rewriting system / Kapil Anand, Matthew Smithson, Khaled Elwazeer et al. // Proceedings

of the 8th ACM European Conference on Computer Systems - EuroSys ’13. — 2013. — P. 295.

19. Chipounov, V. S2E : A Platform for In-Vivo Multi-Path Analysis of Software Systems / Vitaly Chipounov, Volodymyr Kuznetsov, George Candea // Security. — 2011. — Vol. 46. — P. 1–14.
20. Godefroid, P. Automated Whitebox Fuzz Testing / Patrice Godefroid, Michael Y. Levin, David a. Molnar // Search. — 2008. — Vol. 9, no. July.
21. Framework for instruction-level tracing and analysis of program executions / Sanjay Bhansali, Wen-Ke Chen, Stuart de Jong et al. // Proceedings of the 2nd international conference on Virtual execution environments - VEE ’06. — 2006. — P. 154.
22. Godefroid, P. SAGE: Whitebox Fuzzing for Security Testing / Patrice Godefroid, Michael Y. Levin, David Molnar // Queue. — 2012. — Vol. 10, no. 1. — P. 20.
23. Tillmann, N. Pex: White Box Test Generation for .NET / Nikolai Tillmann, Jonathan de Halleux // Proc. TAP. — 2008. — P. 134–153.
24. BitBlaze: A new approach to computer security via binary analysis / Dawn Song, David Brumley, Heng Yin et al. // Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). — 2008. — Vol. 5352 LNCS. — P. 1–25.
25. Исаев, И. К. Применение динамического анализа для генерации входных данных, демонстрирующих критические ошибки и уязвимости в программах / И. К. Исаев, Д. В. Сидоров // Программирование. — 2010. — т. 4. — С. 5167.
26. QEMU. [Электронный ресурс] — Электрон. дан. — Режим доступа: http://wiki.qemu.org/Main_Page. — Загл. с экрана — Англ.
27. Pin: building customized program analysis tools with dynamic instrumentation / Chi-Keung Luk, Robert Cohn, Robert Muth et al. // Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation - PLDI ’05. — 2005. — Vol. 40, no. 6. — P. 190.

28. Cook, S. A. The complexity of theorem-proving procedures / Stephen A. Cook // Proceedings of the third annual ACM symposium on Theory of computing - STOC ’71. — 1971. — P. 151–158.
29. Godefroid, P. Grammar-based whitebox fuzzing / Patrice Godefroid, Adam Kiezun, Michael Y. Levin // ACM SIGPLAN Notices. — 2008. — Vol. 43, no. 6. — P. 206.
30. Majumdar, R. Directed test generation using symbolic grammars / Rupak Majumdar, Ru-Gang Xu // Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering - ASE ’07. — 2007. — P. 134.
31. Efficient state merging in symbolic execution / Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, George Candea // Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation - PLDI ’12. — 2012. — P. 193.
32. Anand, S. Demand-driven compositional symbolic execution / Saswat Anand, Patrice Godefroid, Nikolai Tillmann // Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). — 2008. — Vol. 4963 LNCS. — P. 367–381.
33. Enhancing Symbolic Execution with Veritesting / Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, David Brumley // Proceedings of the 36th International Conference on Software Engineering - ICSE 2014. — 2014. — P. 1083–1094.
34. Automated Coverage-Driven Test Data Generation Using Dynamic Symbolic Execution / Ting Su, Geguang Pu, Bin Fang et al. // 2014 Eighth International Conference on Software Security and Reliability. — 2014. — P. 98–107.
35. Burnim, J. Heuristics for scalable dynamic test generation / Jacob Burnim, Koushik Sen // ASE 2008 - 23rd IEEE/ACM International Conference on Automated Software Engineering, Proceedings. — 2008. — P. 443–446.
36. Augmented dynamic symbolic execution / Konrad Jamrozik, Gordon Fraser, Nikolai Tillmann, Jonathan De Halleux // Proceedings of the 27th IEEE/ACM

International Conference on Automated Software Engineering - ASE 2012. — 2012. — P. 254.

37. White, L. A Domain Strategy for Computer Program Testing / L.J. White, E.I. Cohen // IEEE Transactions on Software Engineering. — 1980. — Vol. SE-6, no. 3. — P. 247–257.
38. Inkumsah, K. Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution / Kobi Inkumsah, Tao Xie // ASE 2008 - 23rd IEEE/ACM International Conference on Automated Software Engineering, Proceedings. — 2008. — P. 297–306.
39. Galeotti, J. P. S. U. Improving search-based test suite generation with dynamic symbolic execution / Juan Pablo (Saarland University) Galeotti, Gordon (University of Sheffiel) Fraser, Andrea (Simula Research Laboratory) Arcuri // International Symposium on Software Reliability Engineering (ISSRE). — 2013. — P. 360–369.
40. Loop-extended symbolic execution on binary programs / Prateek Saxena, Pongsin Poosankam, Stephen McCamant, Dawn Song // Proceedings of the eighteenth international symposium on Software testing and analysis ISSTA 09. — 2009. — P. 225.
41. Obdrzalek, J. Efficient Loop Navigation for Symbolic Execution / Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). — Vol. 6996 LNCS. — 2011. — P. 453–462.
42. Marek, T. Symbolic Execution and Program Loops / Marek Trtik // Phd thesis, Masaryk University. — 2013.
43. Botella, B. Symbolic execution of floating-point / Bernard Botella, Arnaud Gotlieb, Claude Michel. — 2006. — Vol. 2, no. June 2005. — P. 97– 121.
44. FloPSy - Search-Based Floating Point Constraint Solving for Symbolic Execution / Kiran Lakhotia, Nikolai Tillmann, Mark Harman, Jonathan de Halleux // Testing Software and Systems. — 2010. — Vol. 6435. — P. 142–157.

45. Bergan, T. Symbolic execution of multithreaded programs from arbitrary program contexts / Tom Bergan, Dan Grossman, Luis Ceze // ACM SIGPLAN Notices. — 2014.
46. Ganai, M. DTAM: dynamic taint analysis of multi-threaded programs for relevancy / M. Ganai, Dongyoon Lee, A. Gupta // Symposium on the Foundations of Software — 2012. — P. 1–11.
47. Kahkonen, K. Testing Multithreaded Programs with Contextual Unfoldings and Dynamic Symbolic Execution / Kari Kahkonen, Keijo Heljanko // 2014 14th International Conference on Application of Concurrency to System Design. — IEEE, 2014. — jun. — P. 142–151.
48. Lamport, L. Time, Clocks, and the Ordering of Events in a Distributed System / Leslie Lamport // Communications of the ACM. — 1978. — Vol. 21, no. 7. — P. 558–565.
49. Coreutils — GNU core utilities. [Электронный ресурс] Электрон. дан. — Режим доступа: http://www.gnu.org/software/coreutils/coreutils.html — Загл. с экрана — Англ.
50. Parrot VM. [Электронный ресурс] Электрон. дан. — Режим доступа: http://parrot.org/ — Загл. с экрана — Англ.
51. C# Compiler | Mono. [Электронный ресурс] Электрон. дан. — Режим доступа: http://www.mono-project.com/docs/about-mono/languages/csharp/ — Загл. с экрана — Англ.
52. Cubicle: A parallel SMT-based model checker for parameterized systems: Sylvain Conchon, Amit Goel, Sav a Krstic et al. // Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). — 2012. — Vol. 7358 LNCS, no. 1. — P. 718–724.
53. Reisenberger, C. PBoolector: A Parallel SMT Solver for QF_BV by Combining Bit-Blasting with Look-Ahead / Christian Reisenberger // Master's thesis, JKU Linz — 2014.
54. Bounimova, E. Billions and billions of constraints: Whitebox fuzz testing in

production / Ella Bounimova, Patrice Godefroid, David Molnar // Proceedings - International Conference on Software Engineering. — 2013. — P. 122–131.

55. Parallel symbolic execution for automated real-world software testing / Stefan Bucur, Vlad Ureche, Cristian Zamfir, George Candea // Proceedings of the sixth conference on Computer systems - EuroSys ’11. — 2011. — P. 183.

56. A Parallel Approach to Concolic Testing with Low-cost Synchronization / Xiao Yu, Shuai Sun, Geguang Pu et al. // Electronic Notes in Theoretical Computer Science. — 2011. — Vol. 274. — P. 83–96.

57. Independent JPEG Group. [Электронный ресурс] — Электрон. дан. — Режим доступа: http://www.ijg.org/ — Загл. с экрана — Англ.

58. libmpeg2 — a free MPEG-2 video stream decoder. [Электронный ресурс] Электрон. дан. — Режим доступа: http://libmpeg2.sourceforge.net/ — Загл. с экрана — Англ.

59. Heroine Virtual: Libmpeg3. [Электронный ресурс] Электрон. дан. — Режим доступа: http://www.heroinewarrior.com/libmpeg3.php/ — Загл. с экрана — Англ.

60. SWFTOOLS. [Электронный ресурс] Электрон. дан. — Режим доступа: http://www.swftools.org/ — Загл. с экрана — Англ.

61. Libquicktime homepage. [Электронный ресурс] Электрон. дан. — Режим доступа: http://libquicktime.sourceforge.net/ — Загл. с экрана — Англ.

62. Srivastava, A. ATOM / Amitabh Srivastava, Alan Eustace // ACM SIGPLAN Notices. — 1994. — jun. — Vol. 29, no. 6. — P. 196–205.

63. Srivastava, A. A Practical System for Intermodule Code Optimization at Link-Time / Amitabh Srivastava, David W. Wall // Journal of Programming Languages. — 1993. — Vol. 1, no. 1. — P. 1–18.

64. Larus, J. R. EEL: Machine-Independent Executable Editing / James R Larus, Eric Schnarr // Proceedings of the SIGPLAN 95 Conference on Programming Language Design and Implementation. — Vol. 30. — 1995. — P. 291–300.

65. Bird: Binary interpretation using runtime disassembly / Susanta Nanda, Wei Li,

Lap Chung Lam, Tzi Cker Chiueh // Proceedings of the CGO 2006 - The 4th International Symposium on Code Generation and Optimization. — 2006. — P. 358–369.

66. Calder, B. Bitraker Anvil: Binary instrumentation for rapid creation of simulation and workload analysis tools / Brad Calder, Todd Austin, Don Yang, Timothy Sherwood, Suleyman Sair, et al..// In Proceedings of Global Signal Processing (GSPx) Conference, — 2004.
67. Kagstrom, S. Automatic Low Overhead Program Instrumentation with the LOPI Framework / S. Kagstrom, Hakan Grahn, Lars Lundberg // 9th Annual Workshop on Interaction between Compilers and Computer Architectures (INTERACT'05). — IEEE. — P. 82–93.
68. Djoudi, L. Maqao: Modular assembler quality analyzer and optimizer for itanium 2 / Lamia Djoudi, Denis Barthou // Workshop on EPIC architectures and compiler technology. — 2005. — P. 1–20.
69. MIL: A language to build program analysis tools through static binary instrumentation / Andres S. Charif-Rubial, Denis Barthou, Cedric Valensi et al. // 20th Annual International Conference on High Performance Computing, HiPC 2013. — 2013. — P. 206–215.
70. PEBIL: Efficient static binary instrumentation for linux / Michael a. Laurenzano, Mustafa M. Tikir, Laura Carrington, Allan Snavely // ISPASS 2010 - IEEE International Symposium on Performance Analysis of Systems and Software. — 2010. — P. 175–183.
71. Bernat, A. R. Anywhere, any-time binary instrumentation / Andrew R. Bernat, Barton P. Miller // Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools - PASTE '11. — 2011. — P. 9.
72. Nethercote, N. Valgrind / Nicholas Nethercote, Julian Seward // Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation - PLDI '07. — 2007. — P. 89.
73. Bruening, D. Efficient, transparent, and comprehensive runtime code

manipulation / Derek Bruening // Electrical Engineering. — 2004. — P. 306.

74. Android. [Электронный ресурс] Электрон. дан. — Режим доступа: https://www.android.com/ — Загл. с экрана — Англ.
75. Tizen | An open source, standards-based software platform for multiple device categories. [Электронный ресурс] Электрон. дан. — Режим доступа: https://www.tizen.org/ — Загл. с экрана — Англ.
76. GNU Binutils. [Электронный ресурс] Электрон. дан. — Режим доступа: https://www.gnu.org/software/binutils/ — Загл. с экрана — Англ.
77. GCC, the GNU Compiler Collection — GNU Project — Free Software Foundation (FSF) [Электронный ресурс] Электрон. дан. — Режим доступа: https://gcc.gnu.org/ — Загл. с экрана — Англ.