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

Автореферат
диссертации на соискание ученой степени кандидата технических наук

Москва — 2016

Работа выполнена в Федеральном государственном бюджетном образовательном учреждении высшего образования Московский государственный университет имени М. В. Ломоносова и Федеральном государственном бюджетном учреждении науки Институт системного программирования Российской академии наук.

Научный руководитель:	**Иванников Виктор Петрович**, доктор физико-математических наук, академик РАН
Официальные оппоненты:	**Горбунов-Посадов Михаил Михайлович,** доктор физико-математических наук, заведующий отделом Федерального государственного учреждения «Федеральный исследовательский центр Институт прикладной математики им. М.В. Келдыша Российской академии наук», **Булычев Дмитрий Юрьевич,** кандидат физико-математических наук, доцент кафедры системного программирования математико-механического факультета Федерального государственного бюджетного образовательного учреждения высшего образования «Санкт-Петербургский государственный университет»
Ведущая организация:	Вычислительный центр им. А.А. Дородницына Российской академии наук Федерального исследовательского центра «Информатика и управление» Российской академии наук

Защита состоится “15” декабря 2016 г. в 15 часов на заседании диссертационного совета Д 002.087.01 при Институте системного программирования РАН по адресу: 109004, Москва, ул. А. Солженицына, д. 25.

С диссертацией можно ознакомиться в библиотеке и на сайте Федерального государственного бюджетного учреждения науки Институт системного программирования Российской академии наук.

Автореферат разослан “_____”_____2016 г.

Ученый секретарь
диссертационного совета Д 002.087.01,
кандидат физико-математических наук Зеленов С.В.

Общая характеристика работы

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 ограничения 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 и объем 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.

Краткое содержание работы

Во **введении** представлена краткая характеристика работы, рассматриваются вопросы актуальности работы и научной новизны полученных результатов.

В **первой главе** приводится описание основных понятий и концепций, актуальных для задачи автоматического анализа программ при использовании символьного исполнения. Представлен обзор основных групп подходов и инструментов, реализующих данные подходы, рассмотрены факторы, ограничивающие эффективность анализа, и выделены основные направления исследований методов оптимизации.

Раздел 1.1 описывает основные понятия, относящиеся к выполнению программ: **путь выполнения** — последовательность инструкций программы с точки входа до точки останова, выполненная при запуске программы на наборе входных данных; **точка ветвления** — точка кода

программы, содержащая инструкцию условного перехода или блок условного выполнения и порождающая различные пути выполнения программы; **дерево путей выполнения** — представление множества всех допустимых путей выполнения программы в виде дерева (вершины — точки ветвления, переходы — линейные фрагменты кода на пути между точками ветвления).

При обычном исполнении программы средой осуществляется выделение ресурсов (памяти) и выставление счётчика инструкций на точку входа программы. Осуществляется последовательный разбор инструкций, выполнение требуемых действий — изменение внутреннего состояния программы и, возможно, доступ ко внешним источникам, и выбор следующей инструкции для выполнения, пока не будет достигнута точка останова.

Ключевой особенностью обычного исполнения является работа с конкретными данными — запросы на чтение из внешних источников возвращают непосредственные значения; инструкции в точках ветвления, влияющие на выбор последующих инструкций, имеют однозначно определённый результат. Поэтому один запуск программы порождает один путь выполнения.

Символьное исполнение сводится к абстракции от конкретных значений. Элементы внутреннего состояния программы (например, ячейки памяти, регистры) ставятся в соответствие с **символьными переменными** — переменными фиксированного размера, описывающими все допустимые значения элемента. **Символьное выполнение инструкций** заключается не в манипуляции конкретными значениями, а в составлении функциональных зависимостей между символьными переменными, соответствующими операндам и результату инструкции. Функциональные зависимости, построенные для пути выполнения или его фрагмента, объединяются в **трассы ограничений**.

Ключевой особенностью создаваемой трассы ограничений является тот факт, что свободными переменными в ней являются символьные переменные, соответствующие данным, получаемым из внешних источников. Все остальные данные, обрабатываемые программой, включая результаты выполнения проверок в точках ветвления, определены относительно внешних данных. Нахождение значений, удовлетворяющих трассе ограничений, позволяет составлять реальные значения входных данных для обхода путей выполнения.

Раздел 1.2 посвящён рассмотрению наиболее важных характеристик существующих инструментов автоматического обхода путей выполнения. В разделе также представлен обзор распространённых инструментов и их особенностей. Обзор включает такие инструменты анализа, как DART, CUTE (jCUTE), PEX, BitBlaze, EXE, KLEE, Sage, Mayhem, S2E, Avalanche, и использует результаты исследовательских групп Стэндфордского университета, Университета штата Иллинойс, Калифорнийского университета в Бэркли, федеральной политехнической школы Лозанны, Университета Карнеги-Меллон, компаний Bell Labs, Microsoft и др.

В первую очередь рассматривается деление инструментов по режиму обработки различных путей выполнения; существующие инструменты функционируют в режимах offline и online.

Режим offline (DART, CUTE, Sage, Mayhem, Avalanche) близок к схеме обычного выполнения программы и подразумевает итеративный процесс, состоящий из следующих шагов:

1. Запуск программы на наборе входных данных с целью сбора трассы ограничений.

2. Построение новых наборов входных данных по трассе ограничений.
3. Выбор нового набора входных данных и переход на шаг 1. Если новых наборов нет или достигнуто внешнее условие останова, анализ завершается.

Режим offline характеризуется низким потреблением ресурсов памяти и независимостью модулей, решающих отдельные подзадачи. Тем не менее, его использование приводит к повышенным накладным расходам на перезапуск программы и повторное выполнение кода.

Режим online (KLEE, S2E, BitBlaze) предполагает полный контроль работы программы в некоторой среде для обхода разных путей выполнения в рамках единственного запуска. Схема этого режима предполагает следующие шаги:

1. Запуск программы и сбор ограничений во время выполнения.
2. Сохранение состояния программы и выбор направления ветвления в каждой точке ветвления.
3. Откат к точке ветвления и восстановление состояния в конце обработки текущего пути.
4. Выбор нового направления ветвления в данной точке и продолжение анализа для нового пути выполнения.

Режим online отличается отсутствием затрат на перезапуск программы и повторное выполнение кода (восстановление состояний производится быстрее), однако требует больше ресурсов виртуальной памяти на поддержание множества состояний программы.

Не менее важным критерием классификации инструментов является используемая среда выполнения программы и метод извлечения необходимой информации для создания трассы ограничений. Среди существующих инструментов можно выделить следующие группы:

1. использование эмуляторов и виртуальных машин (jCUTE, S2E, KLEE, BitBlaze);
2. использование средств инструментации исходного кода (DART, PEX, CUTE);
3. использование средств, проводящих инструментацию исполняемого кода во время работы программы (Mayhem, Avalanche, Sage).

Раздел 1.3 посвящён рассмотрению основных проблем и ограничивающих факторов методов автоматического обхода путей выполнения:

1. экспоненциальный рост количества путей при увеличении объёма кода программ;
2. отсутствие полиномиальных алгоритмов решения задачи выполнимости булевых ограничений;
3. накладные расходы на создание трасс ограничений (процессорное время, виртуальная память);
4. наличие особенностей исследуемых программ, понижающих полноту создаваемых трасс ограничений.

Существующие методы оптимизации анализа, представленные в современных инструментах, нацелены на прямое или косвенное решение данных проблем. В разделе представлен обзор основных групп данных методов. Предлагаемые в рамках данной работы методы оптимизации нацелены на снижение влияния ограничивающих факторов 1 и 3.

В разделе 1.4 приводится аргументация выбора инструмента Avalanche и реализуемого

данным инструментом подхода в качестве базы для исследования методов оптимизации в рамках данной работы. Рассматриваются следующие аргументы:

- инструмент свободно распространяется вместе с исходным кодом;
- инструмент не требует наличия исходного кода или исполняемого кода в некотором специальном представлении;
- инструмент не использует эмуляцию и реализует механизм удалённого анализа, при котором выполнение программы производится на отдельном устройстве целевой процессорной архитектуры (прежде всего, ARM).

Раздел также включает в себя подробное описание общей схемы работы инструмента. Инструмент Avalanche состоит из управляющего модуля и компонентов covgrind, tracegrind и STP. Анализ, проводимый инструментом Avalanche, основывается на схеме offline, описанной выше. Для определения наиболее перспективных наборов данных для текущей итерации анализа используется метрика прироста покрытия базовых блоков (сколько новых базовых блоков кода программы, не зафиксированных ни на одном запуске за предыдущие итерации анализа, было выполнено на наборе входных данных).

Управляющий модуль Avalanche поддерживает множество наборов входных данных. Каждому набору данных соответствует значение метрики прироста покрытия и значение глубины точки ветвления, породившей данный набор. До начала первой итерации анализа во множество вносится первый элемент — набор входных данных, представленный извне, количество базовых блоков, выполняемых при запуске программы на этом наборе, и значение глубины 0. Наиболее перспективным набором в любой момент времени считается набор с наибольшим приростом и наименьшей глубиной.

1. На каждой итерации программа запускается на наиболее перспективном наборе входных данных под управлением компонента tracegrind, осуществляющего сбор трассы ограничений, соответствующей данному пути выполнения.
    1. Модуль tracegrind поддерживает **теневую память**, хранящую метки для ячеек памяти и регистров программы. Наличие метки в момент выполнения программы T означает, что вместо конкретных значений регистра или ячейки памяти необходимо использовать символьные переменные.
    2. При выполнении программы обрабатывается каждый вызов системных функций чтения из внешних источников. В теневую память заносятся метки для ячеек памяти, в которые были записаны прочитанные данные. В трассу ограничений добавляются записи, задающие соответствие между адресами ячеек и смещениями данных во внешнем источнике.
    3. При выполнении каждой инструкции программы осуществляется проверка её аргументов.
    4. Если ни один из аргументов не имеет метки в теневой памяти, то из теневой памяти удаляются метки регистров и ячеек памяти, в которые записывается результат инструкции.

5. Если хотя бы один из аргументов имеет метку, в теневую память добавляются метки для результатов инструкции. В трассу ограничений добавляется запись, фиксирующая зависимость результата и аргументов инструкции; при формировании записи используются символьные переменные и реальные значения согласно пункту 1.
6. Для инструкций ветвления, имеющих помеченные аргументы, в трассу ограничений добавляется маркер ветвления.

2. Трасса ограничений разбивается на подтрассы по маркерам ветвления: первая трасса включает ограничения с начала работы программы до первого маркера, вторая трасса — с начала работы программы до второго маркера и т.д.
3. В каждой подтрассе последнее ограничение, определяющее результат операции ветвления, меняется на противоположное для формирования потенциального альтернативного пути выполнения. Каждая подтрасса имеет значение глубины, соответствующее сумме глубины набора данных, обработанного компонентом tracegrind на итерации, и порядкового номера маркера.
4. Модифицированные подтрассы передаются на компонент STP, производящий поиск значений входных данных, при которых подтрассы выполнимы, составляя тем самым новые наборы входных данных. Глубина каждого набора данных определяется равной глубине породившей его подтрассы.
5. Для каждого успешно полученного нового набора входных данных осуществляется запуск программы под управлением модуля covgrind, отслеживающего множество базовых блоков кода программы, задействованных при выполнении. На основе данных о предыдущих запусках рассчитывается метрика прироста покрытия.
6. Осуществляется переход на следующую итерацию анализа. При отсутствии новых наборов входных данных или по истечении выделенного времени анализа работа инструмента прекращается.

Реализация поставленных в рамках работы задач предполагаются с помощью проведения:

1. модификации модуля tracegrind, позволяющей ограничивать обработку условных операций, оперирующих помеченными данными, на основе спецификаций пользователя (глава 2);
2. модификации управляющего модуля с целью организации параллельной работы компонентов Avalanche (глава 3);
3. замены модулей covgrind и tracegrind (основанных на системе динамической инструментации) на модули статической инструментации (глава 4).

Вторая глава посвящена исследованию методов решения проблемы экспоненциального роста числа путей выполнения и сложности логических зависимостей выполнения программы от входных данных путём автоматической фильтрации потока данных по маскам, накладываемым на источники входных данных и внутренние структурные элементы целевой программы (функции).

В разделе 2.1 рассматривается влияние свойств программы (объём кода, использование циклов и параллельных потоков обработки данных) на сложность обхода путей выполнения.

В разделе 2.2 рассматриваются вопросы актуальности использования экспертных знаний о программах для повышения эффективности их анализа. Приводится обзор существующих решений, базирующихся на таком подходе.

В инструментах Sage и CESE была предложена методика использования формальных грамматик, описывающих множество корректных входных данных. Оба подхода позволяют значительно снизить количество путей выполнения; в то же время, подход в инструменте Sage накладывает дополнительные ограничения на целевые программы, а подход в инструменте CESE плохо масштабируем по размеру входных данных.

В инструментах KLEE, EXE и S2E существует возможность ручной разметки исходного кода программы с целью выделения источников символьных данных. Этот подход является более мощным, чем рассмотренные выше грамматики входных данных, однако требует модификации кода и проведения сборки программы для каждого набора спецификаций.

В разделе 2.3 рассматриваются предлагаемые методы ограничения анализа с учётом структуры данных, получаемых программой из внешних источников, и с учётом структуры кода программы. В отличие от рассмотренных выше аналогов данные методы не накладывают ограничения на формат входных данных целевой программы и не требуют непосредственного доступа к исходному коду программы.

Сложные форматы данных обычно включает в себя элементы, обрабатываемые различными модулями программы. При проведении анализа программы исследование отдельных модулей может иметь больший приоритет; осуществлять направленную обработку точек ветвления в данных модулях можно путём отслеживания зависимостей от отдельных блоков входных данных.

Предлагаемый метод заключается в проведении анализа с учётом статических масок, задаваемых экспертом исходя из знаний о формате входных данных. Маски позволяют задавать отдельные смещения и интервалы смещений в блоках данных, получаемых из внешних источников, которые нужно рассматривать как актуальные при обходе путей. Так, например, данные маски позволяют зафиксировать фрагменты входных файлов и осуществлять перебор значений только в остальных фрагментах входных файлов.

Схема работы модуля tracegrind, указанная ранее, меняется следующим образом:

- (пункт 2) При обработке системных вызовов чтения из внешних источников в теневую память добавляются метки только для тех ячеек памяти, в которые попадают данные по смещениям, проходящим фильтрацию.

Устранение фрагментов входных данных из рассмотрения модулем tracegrind приводит к уменьшению общего размера трассы ограничений и предотвращает обход путей выполнения, отличающихся по точкам ветвления, зависимым только от отфильтрованных входных данных.

Фрагменты программы, объединённые в функции и модули, могут осуществлять обработку входных данных независимо или учитывая малое количество взаимных зависимостей. Для проведения направленного анализа отдельных фрагментов можно проводить фильтрацию точек ветвления.

Предлагаемый метод рассматривает фильтрацию точек ветвления по именам функций,

содержащих данные точки. Эксперт задаёт множество имён функций, точки ветвления в которых нужно игнорировать, или множество имён функций, точки ветвления в которых нужно обрабатывать, и игнорировать при этом любые другие точки ветвления. Поддерживается возможность использовать для задания функций регулярные выражения.

Схема работы модуля tracegrind, рассмотренная ранее, изменяется следующим образом:

- (пункт 6) При обработке условных инструкций маркер ветвления для обозначения подтрассы добавляется только в том случае, когда точка ветвления удовлетворяет включающим фильтрам.

Данные фильтры позволяют уменьшить количество рассматриваемых путей выполнения при наличии знаний о внутренней структуре программы.

Раздел 2.4 содержит описание практических экспериментов по применению схемы частичного анализа.

Для оценки эффективности частичного анализа были выбраны приложения, осуществляющие разбор сложных форматов данных. Были рассмотрены виртуальные машины LLVM, Parrot, и среда выполнения кода C# mono; анализ проводился для компонентов этих систем, осуществляющих разбор кода в промежуточном представлении. Практические эксперименты заключались в проведении анализа в стандартном режиме и проведении частичного анализа на основе фильтрации источников входных данных и функций программы. Задаваемые ограничения частичного анализа предполагали целенаправленный обход кода модулей, осуществляющих обработку фиксированных семантических структур промежуточного представления, исключая обработку содержимого заголовочной информации во входных данных.

Таблица 1: Результаты применения частичного анализа

Программа	Количество уникальных дефектов		Время обнаружения первого дефекта, с	
	Полный анализ	Частичный анализ	Полный анализ	Частичный анализ
llc (LLVM)	5	7	9	5
pbc_dump (Parrot)	3	6	41	10
monodis (mono)	0	4	-	67

Таким образом, использование частичной схемы анализа позволило увеличить число обнаруживаемых дефектов и сократить время, необходимое для обнаружения дефектов. Дополнительно было зафиксировано ускорение обхода путей выполнения и снижение среднего времени обработки трасс ограничений из-за уменьшения среднего размера данных трасс.

Обнаруженные дефекты включают в себя ситуации аварийного завершения программ из-за попытки разыменования нулевого указателя и нарушения условий в программных конструкциях assert.

Третья глава посвящена модификациям инструмента Avalanche, направленным на повышение эффективности обработки данных и использования вычислительных ресурсов.

В разделах 3.1-3.2 приводится оценка возможностей выделения независимых подзадач метода автоматического обхода путей выполнения программы и оценки актуальности распараллеливания их обработки. Рассматривается следующие схемы разбиения:

1. Параллельная обработка отдельных путей выполнения (построение трассы ограничений, построение наборов данных, порождаемых трассой, оценка приоритетности построенных наборов данных).
2. Параллельная обработка множеств путей выполнения.
3. Параллельная обработка программы при извлечении трассы ограничений.
4. Параллельная обработка трассы ограничений.

Первые две схемы являются хорошо масштабируемыми по количеству вычислительных узлов и перспективными с точки зрения прироста производительности (раздел 3.1), в то время как третья и четвёртая схемы позволяют добиться ограниченного прироста производительности (раздел 3.2)

Раздел 3.3 рассматривает существующие разработки для инструментов анализа, нацеленные на более эффективное использование архитектурных особенностей вычислительных ресурсов. Рассмотрены методы, предложенные в инструментах SAGE, Cloud9, KLEE и PEX. Данные методы соответствуют схемам 1 и 2, указанным выше. Применение их непосредственно в инструменте Avalanche представляется возможным, однако не позволит в полной мере учитывать специфику ограничения путей выполнения по глубине точек ветвления и метрике прироста покрытия.

Раздел 3.4 рассматривает особенности взаимодействия компонентов инструмента Avalanche. На основе практических результатов, полученных ранее авторами инструмента, и исследовании потока данных в рамках итераций анализа, проводимого инструментом, предлагается перспективный вариант организации параллельных вычислений. Данный вариант базируется на том, что действия по обработке отдельных точек ветвления в рамках схемы анализа, рассмотренной в разделе 1.4, образуют неделимую последовательность, регулируемую одним потоком данных, но в рамках одной итерации анализа проверки различных точек ветвления являются независимыми (рис. 1).

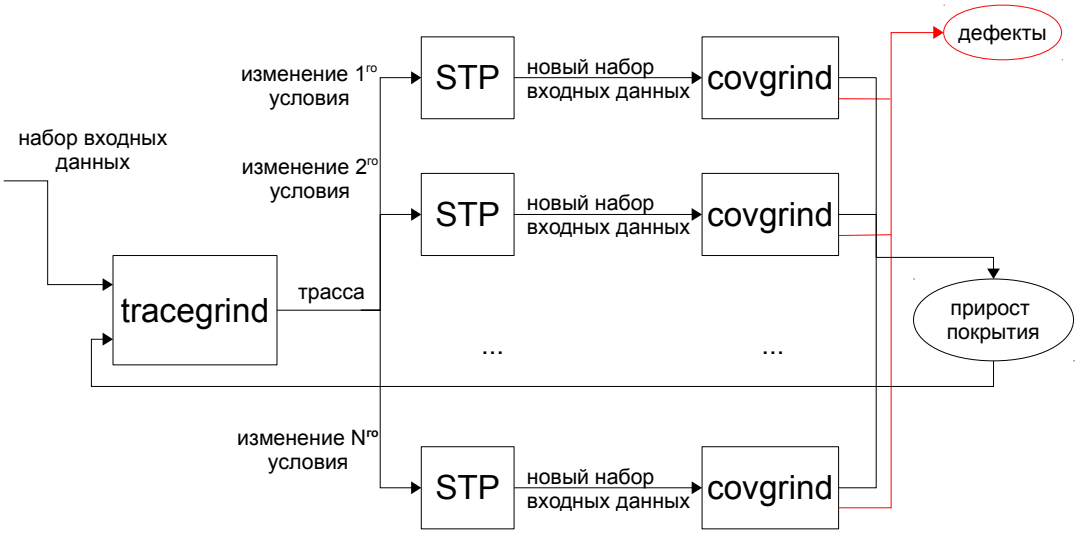

Рисунок 1: Обработка путей выполнения в рамках итерации анализа Avalanche

Раздел 3.5 описывает предлагаемую схему параллельной обработки отдельных путей выполнения в рамках итераций анализа. Предлагаемая схема параллельной обработки вводит следующие принципы:

- Управляющий модуль инструмента Avalanche настраивается на поддержку M параллельных потоков. На каждом потоке включается активный исполнитель.
- В рамках итерации анализа управляющий модуль разбивает трассу, полученную от модуля tracegrind на N подтрасс. Для каждой подтрассы управляющий модуль оформляет задание, включающее вызов модуля STP и вызов модуля covgrind, если модуль STP успешно вычисляет набор входных данных по подтрассе.
- Управляющий модуль раздаёт задания активным исполнителям в порядке появления заданий и в порядке освобождения исполнителей. Исполнители бездействуют только в случае нехватки заданий для обработки.

Раздел 3.6 рассматривает практические оценки возможности ускорения анализа при использовании параллельной схемы.

В рамках предлагаемой схемы доля времени анализа, соответствующая активной работе только одного узла, ограничивается работой управляющего модуля и модуля tracegrind, и обработкой последнего задания итерации. Эффективность применения параллельной схемы будет максимальна, когда эта доля времени минимальна.

Соотношение работы компонентов Avalanche и особенности подтрасс, получаемых на итерациях, зависят от анализируемых программ. В общем случае оценку доли вычислений, выполняющихся в один поток при простое остальных, можно определить следующим образом:

$$\alpha = \frac{Tg + Dr}{Tg + Dr + Cv + STP} + \sum_{i=1,I} \frac{T_i * max_{j=1,N_i}(Cv_{i,j} + STP_{i,j})}{T * (Cv_i + STP_i)}$$, 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 путей выполнения на итерации *i*
- T_i — время выполнения итерации *i*, *T* — общее время анализа.

В рамках экспериментов были рассмотрены проекты с открытым исходным кодом, ранее исследованные инструментом Avalanche. Для рассмотренных проектов при сохранении параметров анализа Avalanche, предложенных авторами инструмента Avalanche, значение параметра α не превышало 0.11.

В разделе 3.7 приводятся результаты практических экспериментов по применению реализации параллельной схемы обработки путей выполнения. В таблице 2 рассматриваются результаты применения параллельной схемы анализа для проектов с открытым исходным кодом при использовании 4 потоков по сравнению с результатами применения стандартной схемы анализа.

Таблица 2: Сравнение результатов последовательного и параллельного анализа

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

Таким образом, применение параллельной схемы обработки позволило добиться повышения общей скорости работы инструмента — за ограниченное время удалось добиться обхода большего

числа путей выполнения программ и, как следствие, обнаружить большее количество дефектов.

Зафиксированные дефекты включали в себя аварийное завершение программы из-за попытки разыменования нулевого указателя, попытки произвести операцию деления на ноль, и зацикливание программы из-за попытки считать некорректный объём данных.

Дополнительно в разделе рассмотрены потенциальные верхние оценки прироста производительности по закону Амдала и графики сравнения предполагаемого и реального прироста по результатам практических экспериментов.

В **четвёртой главе** рассматривается вопрос выбора механизма сбора информации о выполнении программы.

В разделе 4.1 приводится оценка особенностей различных подходов к извлечению данных о программе во время выполнения. Рассматривается возможность использования **предварительной** инструментации **исполняемого** кода программы в рамках метода автоматического обхода путей выполнения и оценка эффективности такого подхода по сравнению с другими подходами.

Инструментация исходного кода и предварительная инструментация исполняемого кода:

- инструментация исходного кода требует наличия исходного кода и является чувствительной к настройкам систем компиляции и сборки; в то же время, этот метод не требует специальной поддержки различных процессорных архитектур и форматов исполняемого кода и позволяет обращаться к высокоуровневым конструкциям кода программы;
- предварительная инструментация исполняемого кода требует поддержки различных процессорных архитектур и форматов исполняемого кода и чувствительна к обфускации кода; в то же время, этот метод не требует наличия исходного кода и позволяет работать с низкоуровневыми элементами кода.

Динамические методы (эмуляция, инструментация кода во время выполнения) и предварительная инструментация исполняемого кода:

- динамические методы вносят высокий уровень накладных расходов при многократном выполнении блоков кода программы; в то же время, эти методы обеспечивают полноту покрытия программного кода;
- предварительная инструментация исполняемого кода не позволяет достичь полноты покрытия кода при загрузке динамических библиотек, недоступных на этапе инструментации, и использования программой самомодифицирующегося кода; в то же время, этот метод вносит накладные расходы на разбор и изменения кода однократно.

Раздел 4.2 содержит обзор средств, позволяющих проводить инструментацию исполняемого кода. По результатам проведённого обзора не было обнаружено инструментов, полностью предоставляющих функциональность, необходимую в рамках данной работы (прежде всего, из-за отсутствия поддержки процессорной архитектуры ARM последних версий). Было принято решение о необходимости разработки новой системы, осуществляющей работу с исполняемым кодом формата ARM ELF, для использования в рамках инструмента Avalanche. Для предлагаемой системы был выделен ряд основных требований:

- (базовая функциональность статической инструментации) разрабатываемая система должна позволять внедрять в файлы исполняемого кода дополнительные сегменты кода; данные изменения не должны приводить к нарушению исходной функциональности;
- (соответствие уровню существующих систем) разрабатываемая система должна позволять настраивать процесс инструментации под целевую задачу анализа;
- (возможность реализации модулей Avalanche) разрабатываемая система должна позволять проводить инструментацию на уровне отдельных инструкций и блоков инструкций исполняемого кода.

В разделе также приводится краткий обзор основных особенностей формата ARM ELF.

Предлагаемый метод статической инструментации исполняемого кода представлен в разделе 4.3. Основные шаги при обработке файлов исполняемого включают следующие:

- Разбор пользовательских спецификаций, описывающих функциональность, которую необходимо внедрить в файлы целевой программы.
- Последовательный разбор файлов программы с целью выделения точек инструментации и контекста данных точек.
- Для каждой точки инструментации создаётся блок кода, соответствующий требуемому инструментационному коду и учитывающий параметры контекста точки.
- Блоки инструментационного кода переводятся в блоки исполняемого кода и объединяются в объектный файл.
- Содержимое объектного файла присоединяется к файлу исполняемого кода целевой программы.
- Итоговый файл исполняемого кода модифицируется таким образом, чтобы при выполнении в точках инструментации осуществилась передача управления на внедрённый код.

Язык спецификаций инструментации позволяет описывать тройки {T, F, C}, где T – тип точки инструментации, F – фильтр по именам функций целевого исполняемого файла, C – исходный код на языке C, реализующий требуемую функциональность.

Поддерживаются следующие типы точек инструментации:

- позиции в исполняемом коде, на которых находятся инструкции функциональных групп (обработка данных, доступ к памяти и т.д.);
- позиции в исполняемом коде, обладающие специальными особенностями по потоку управления (точки входа и выхода из функций, базовых блоков и т.д.).

Блоки кода для точек инструментации формируются в виде исходного кода на языке C, добавляются в единый файл исходного кода и переводятся в объектный файл с исполняемым кодом с использованием средств компиляции кода.

Получение итогового модифицированного варианта исполняемого файла включает следующие этапы:

- Присоединение исполняемого кода инструментации к целевому файлу в виде независимого фрагмента.

- Для каждой точки инструментации непосредственно в исполняемом коде целевого файла производятся следующие изменения:
  - Замена инструкции или блока инструкций на инструкцию безусловного перехода на соответствующий блок инструментационного кода в добавленном фрагменте.
  - Перенос заменённой инструкции в конец соответствующего блока инструментационного кода для сохранения исходной функциональности программы.
- Производится корректировка управляющей информации целевого файла исполняемого кода (изначально сформированной в процессе сборки) для внесения записей о глобальных символах и внешних зависимостях, используемых инструментационным кодом.
- Производится коррекция смещений констант и вызовов функций из внешних библиотек для инструментационного кода.

Раздел 4.4 содержит описание деталей программной реализации разработанной системы инструментации. Приведён обзор комплекса средств и библиотек binutils, используемых для работы с файлами формата ELF и разбора исполняемого кода. Для автоматической генерации объектных файлов инструментационного кода используется комплекс компиляции и сборки gcc.

В разделе 4.5 рассмотрены инструменты анализа программы, созданные на основе данной системы для использования в инструменте Avalanche. Модуль инструментации программы tracegrind_static осуществляет модификацию программы с целью внедрения функциональности по созданию трассы ограничений. Данная модификация покрывает точки кода, содержащие инструкции обработки данных, доступа к памяти и условные инструкции; для каждой точки инструментационный код извлекает параметры инструкций и осуществляет вызов специализированной библиотеки, поддерживающей теневую память аналогично исходной версии модуля tracegrind. Модуль инструментации covgrind_static осуществляет внедрение дополнительного кода в начало каждого базового блока. Инструментационный код поддерживает множество базовых блоков, в которое добавляются уникальные идентификаторы базовых блоков.

Раздел 4.6 содержит обзор результатов экспериментов анализа с целью выявления практической эффективности применения статической инструментации кода вместо динамической инструментации. Для оценки анализа были рассмотрены инструменты разбора файлов мультимедиа, ранее исследованные с помощью инструмента Avalanche. Статическая инструментация была проведена для исполняемых файлов программ, динамических библиотек данных проектов и отдельных функций обработки данных в системных библиотеках.

Таблица 3: Сравнение эффективности статической и динамической инструментации

Программа	Итерации анализа	Пути выполнения	Уникальные дефекты	Наборы данных	Среднее время сбора трассы на пути, с	Среднее время проверки пути, 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

Так, для рассмотренных проектов с открытым исходным кодом при проведении анализа с использованием пары устройств (гостевое устройство процессорной архитектуры ARM Tizen PD-RQ; хост-устройство Intel Core i5-2500) было зафиксировано снижение затрат на сбор трассы ограничений и оценку приоритетности пути выполнения по приросту покрытия базовых блоков. Для некоторых проектов это позволило обнаружить больше уникальных дефектов за отведённое время работы инструмента Avalanche. При этом все дефекты, обнаруженные при анализе с использованием динамической инструментации, были обнаружены и при статической инструментации. Найденные дефекты включают следующие ошибки: аварийное завершение программы из-за попытки разыменовать нулевой или неинициализированный указатель, деление на ноль, зацикливание программы из-за попытки считать некорректный объём входных данных.

В **заключении** представлены основные результаты работы:

- 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.

Внедрение результатов работы:

Реализованные в рамках работы методы внедрены в инструмент динамического анализа программ Avalanche, разрабатываемый в Институте системного программирования РАН. Инструмент активно развивается в рамках научно-исследовательских работ, проводимых в Институте, позволил найти критические дефекты более, чем в 30 проектах с открытым исходным кодом, и более 100 критических дефектов в программах, входящих в состав операционной системы Astra Linux Orel.

Список работ, опубликованных автором по теме диссертации:

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.