Федеральное государственное автономное образовательное учреждение высшего профессионального образования

Национальный исследовательский университет «Высшая школа экономики»

Московский институт электроники и математики Национального исследовательского университета «Высшая школа экономики»

Допущен к защите

Заведующий кафедрой ВСиС

\_\_\_\_\_\_\_\_\_\_\_ А.В. Вишнеков

«\_\_» \_\_\_\_\_\_\_\_\_\_\_\_ 2013 г.

Ишханов Илья Григорьевич

**ИССЛЕДОВАНИЕ МЕТОДОВ ДЕКОМПИЛЯЦИИ ПРОГРАММ ДЛЯ ПРОЦЕССОРОВ X86**

Направление 23.01.00.68 - Информатика и вычислительная техника

Магистерская программа - Сети ЭВМ и телекоммуникации

Магистерская диссертация

|  |  |  |
| --- | --- | --- |
| Выполнил | подпись | магистрант группы СМ-31  И.Г. Ишхановы |
| Научный руководитель | подпись | доцент, к.т.н.  В.Э. Карпов |
| Рецензент | подпись | доцент, к.т.н.  А.А. Рогов |

# СОДЕРЖАНИЕ

[ВВЕДЕНИЕ 1](#_Toc359217269)

[СОДЕРЖАНИЕ 2](#_Toc359217270)

[1 ОБЗОР СУЩЕСТВУЮЩИХ РАБОТ 8](#_Toc359217271)

[1.1 Предыдущие работы 8](#_Toc359217272)

[1.1.1 История декомпиляции (1960-1979) 9](#_Toc359217273)

[1.1.2 История декомпиляции (1980-1999) 19](#_Toc359217274)

[1.1.3 История декомпиляции (2000-н/в) 23](#_Toc359217275)

[1.1.4 Обзор существующих декомпиляторов на язык Си 28](#_Toc359217276)

[1.2 Проблемы декомпиляции 30](#_Toc359217277)

[1.2.1 Отделение кода от данных 34](#_Toc359217278)

[1.2.2 Отделение указателей от констант 35](#_Toc359217279)

[1.2.3 Восстановление указателей, сложенных со смещением 35](#_Toc359217280)

[1.2.4 Виды декомпиляторов 37](#_Toc359217281)

[1.2.5 Теоретические пределы 44](#_Toc359217282)

[2 АРХИТЕКТУРА ДЕКОМПИЛЯТОРА 46](#_Toc359217283)

[2.1 Фазы декомпиляции 46](#_Toc359217284)

[2.1.1 Синтаксический анализ 47](#_Toc359217285)

[2.1.2 Семантический анализ 48](#_Toc359217286)

[2.1.3 Генерация промежуточного кода 50](#_Toc359217287)

[2.1.4 Генерация графа потока управления 50](#_Toc359217288)

[2.1.5 Анализ потока данных 51](#_Toc359217289)

[2.1.6 Анализ графа потока управления 51](#_Toc359217290)

[2.1.7 Генерация кода 52](#_Toc359217291)

[2.2 Группировка фаз 53](#_Toc359217292)

[2.3 Вспомогательный инструментарий декомпилятора 54](#_Toc359217293)

[2.3.1 Загрузчик 55](#_Toc359217294)

[2.3.2 Генератор сигнатур 55](#_Toc359217295)

[2.3.3 Генератор прототипов 56](#_Toc359217296)

[2.3.4 Дизассемблер 56](#_Toc359217297)

[2.3.5 Связывание библиотек 56](#_Toc359217298)

[2.3.6 Постпроцессор 56](#_Toc359217299)

[3 МЕТОДЫ ДЕКОМПИЛЯЦИИ 58](#_Toc359217300)

[3.1 Восстановление функций 58](#_Toc359217301)

[3.1.1 Выделение функций 58](#_Toc359217302)

[3.1.2 Выявление параметров и возвращаемых значений 60](#_Toc359217303)

[3.1.3 Распознавание библиотечных функций 64](#_Toc359217304)

[3.1.4 Обнаружение функции main 67](#_Toc359217305)

[3.2 Восстановление управляющих конструкций 68](#_Toc359217306)

[3.2.1 Сводимые и несводимые графы 69](#_Toc359217307)

[3.2.2 Шаблоны графа потока управления 72](#_Toc359217308)

[3.2.3 Методы анализа управляющих конструкций 80](#_Toc359217309)

[4 ПРОЕКТИРОВАНИЕ И РАЗРАБОТКА ДЕКОМПИЛЯТОРА 86](#_Toc359217310)

[4.1 Введение 86](#_Toc359217311)

[4.2 Проектирование загрузчика 87](#_Toc359217312)

[4.2.1 Загрузчик PE файла 88](#_Toc359217313)

[4.3 Декодер 96](#_Toc359217314)

[4.4 Дизассемблер 103](#_Toc359217315)

[4.4.1 Принципы дизассемблирования 103](#_Toc359217316)

[4.4.2 Проблемы при дизассемблировании 104](#_Toc359217317)

[4.4.3 Модуль дизассемблирования 104](#_Toc359217318)

[4.4.4 Алгоритм дизассемблирования 105](#_Toc359217319)

[4.5 Трассировка программы 110](#_Toc359217320)

[4.6 Декомпилятор 113](#_Toc359217321)

[4.6.1 Генерация промежуточного представления 113](#_Toc359217322)

[4.6.2 Анализ графа потока управления 125](#_Toc359217323)

[5 АПРОБАЦИЯ 127](#_Toc359217324)

[6 ВЫВОДЫ 135](#_Toc359217325)

[7 СПИСОК ЛИТЕРАТУРЫ 137](#_Toc359217326)

# ВВЕДЕНИЕ

**Актуальность.** Создание и разработка сложных программных систем различного назначения часто ведется посредством интеграции отдельных компонент, выполненных как собственными, так и сторонними разработчиками. Это позволяет значительно сократить стоимость и время разработки программного обеспечения. Внешние модули могут поставляться без исходного кода. Наличие таких модулей в системе уменьшает уровень надежности разрабатываемого приложения с точки зрения информационной безопасности. В частности, сторонние модули могут содержать закладки или уязвимости, способствующие утечке информации и успешным атакам на информационную систему. Кроме того, программные модули от внешних разработчиков могут содержать ошибки, исправление которых оказывается затруднительным. Следовательно, весь сторонний код должен подвергаться аудиту с точки зрения безопасности его внедрения и использования.

Программные компоненты, представленные в виде исполняемых файлов или на языке ассемблера, сложны для анализа специалистами в области информационной безопасности. Для более качественного и продуктивного анализа их лучше предоставлять специалистам на более высоком уровне представления, например, на языке высокого уровня, в частности на языке программирования Си. Ассемблерный код и, тем более, исполняемые файлы не позволяют с приемлемыми трудозатратами оценить взаимосвязь элементов программы, а также идентифицировать в программе различные алгоритмические конструкции, в то время как наличие восстановленной программы на языке высокого уровня дает возможность преодолеть указанные выше трудности. В качестве одного из средств для повышения уровня абстракции представления программы может использоваться декомпиляция.

Декомпиляция – это процесс автоматического восстановления программы на языке высокого уровня из программы на языке низкого уровня. Декомпилятор ‑ инструментальное средство, получающее на вход программу на языке ассемблера или другое аналогичное низкоуровневое представление и выдающее на выход эквивалентную ей программу на некотором языке высокого уровня.

Также декомпиляция может использоваться для обеспечения совместимости программных приложений, а именно для анализа протоколов взаимодействия в случае, когда они описаны недостаточно полно или не описаны вообще. Декомпиляция позволяет упростить восстановление состояний и структур данных протокола взаимодействия.

В настоящее время из широко используемых компилируемых языков программирования высокого уровня распространены языки Си и Си++, поскольку именно они наиболее часто используются при разработке прикладного и системного программного обеспечения для платформ Windows, MacOS и Unix. Поэтому декомпиляторы с этих языков имеют наибольшую практическую значимость. Язык Си++ можно считать расширением языка Си, добавляющим в него концепции более высокого уровня относительно языка Си. Поскольку при обратной инженерии в целом и при декомпиляции в частности уровень абстракции представления программы повышается, то можно считать, что программы на языке Си являются промежуточным уровнем при переходе от программы на языке ассемблера к программе на языке Си++. Дальше повысить уровень абстракции представления программы можно посредством широко известных методов рефакторинга, позволяющих, например, выделять объектные иерархии из процедурного кода.

Из-за ряда трудностей задача декомпиляции не решена в полной мере до сих пор, хотя была поставлена еще в 60-е годы прошлого века. С теоретической точки зрения задачи построения полностью автоматического универсального дизассемблера и декомпилятора относят к алгоритмически неразрешимым. Неразрешимость задач следует из того, что задача автоматического разделения кода и данных является алгоритмически неразрешимой.

В силу вышесказанного является актуальной разработка декомпилятора в предположении, что исходная программа была реализована на конкретном языке программирования. Декомпилятор определяется как транслятор с языка низкого уровня в язык высокого уровня, который минимизирует количество артефактов трансляции в результирующей программе и выполняет восстановление высокоуровневых конструкций целевого языка максимально полно в предположении, что исходная программа была написана на этом языке высокого уровня.

**Целью** научно-исследовательской работы является разработка методов и инструментальной среды для декомпиляции программ. Алгоритмы и шаблоны, используемые для декомпиляции, должны быть применимы к широкому спектру целевых низкоуровневых программ и позволять автоматически полно восстановить все высокоуровневые конструкции целевого языка высокого уровня.

Достижение поставленной цели предполагает решение следующих **задач**:

* **разработка** архитектуры обобщенного декомпилятора;
* **разработка** модуля дизассемблирования, генерирующий низкоуровневое промежуточное представление;
* **разработка** алгоритма получения промежуточного кода;
* **разработка** алгоритмов анализа потока данных;
* **создание** макета, на котором можно практически проверить разработанные алгоритмы декомпиляции.

Надо отметить, что в рамках данной работы был реализован прототип, работающий только с исполняемыми файлами семейства ОС Microsoft Windows (PE-файлы), но в будущем планируется его расширить для работы с другими видами исполняемых файлов. Возможность подобного расширения уже заложена в архитектуру прототипа.

**Научная новизна.** В работе предложено улучшить существующую классическую архитектуру декомпилятора добавив в нее новую фазу, позволяющую более качественно произвести разбор исполняемого файла.

В отличие от существующих работ, был разработан и подробно описан алгоритм получения промежуточной формы представления и спроектированы структуры данных для хранения этого промежуточного представление. Описаны алгоритмы, осуществляющие анализ потока данных на основе этого внутреннего представления.

Так же, в ходе выполнения магистерской диссертации был разработан рабочий макет декомпилятора, производящий промежуточное представление. Разработанный инструмент доступен в исходных кодах и может быть использован желающими для собственных исследований алгоритмов декомпиляции.

# ОБЗОР СУЩЕСТВУЮЩИХ РАБОТ

## Предыдущие работы [6]

Самый первый декомпилятор был написан Джоэлом Доннелли в 1960 году на компьютере Remington Rand Univac М-460 Countess в Военно-морской Лаборатории Электроники для декомпиляции машинного кода в язык Neliac. Проектом руководил Профессор Моррис Холстед, занимавшийся декомпиляцией на протяжении 60-х и 70-х годов. Результатом его работы стали методы и способы, определившие фундаментальные основы, по сей день использующиеся при создании декомпиляторов.

На протяжении всего времени с момента создания декомпиляторам находилось множество различных применений. Например, в 60-х годах они использовались для перевода программ с компьютеров второго поколения на компьютеры третьего, что помогло избежать необходимости переписывать всю программную базу с нуля. В 70-80-х годах декомпиляторы использовались для переноса программ, их отладки, воссоздании потерянного исходного кода и внесения изменений в существующие исполняемые файлы. В 90-х декомпиляторы стали серьезным инструментом обратной разработки, позволяющим решать задачи проверки программ на наличие вредоносного кода, верификации кода, сгенерированного компилятором, переноса бинарных программ с одной машины на другую и проверки реализации конкретных функций библиотеки.

Далее следует описание наиболее известных декомпиляторов и исследовательских работ, посвященных проблемам декомпиляции. Большинство из этих описаний впервые появились в кандидатской диссертации Кристины Чифуэнтес "Методы обратной компиляции".

### История декомпиляции (1960-1979)

#### Декомпилятор D-Neliac (1960,1962)

Как сообщает Холстед (Halstead), декомпилятор Donnelly-Neliac (D-Neliac) был разработан Дж. К. Доннели и Х. Энглендером в Военно-морской Лаборатории Электроники (NEL) в 1960 году. Neliac является диалектом Алгола, разработанным в NEL в 1955 году. Декомпилятор D-Neliac преобразовывал машинный код в код на языке Neliac. Различные версии были написаны для компьютеров Remington Rand Univac М-460 Countess и Control Data Corporation 1604.

D-Neliac оказался полезным для перевода программ написанных на других языках на язык Neliac и для обнаружения в них логических ошибок. D-Neliac послужил доказательством того, что возможно создать декомпилятор.

#### Декомпилятор The Lockheed Neliac (1963-1967)

Декомпилятор Lockheed, также известен как декомпилятор из IBM 7094 в Univac 1108 Neliac, который помог в переносе научных приложений с IBM 7094 на новый Univac 1108 от компании Lockheed Missiles and Space. Перфокарты были переведены на Univac 1108 Neliac, а также в код на ассемблере (в тех случаях, когда это было возможно). Двоичный код был получен в результате компиляции приложений на Фортране.

Холстед проанализировал затраты, необходимые для того, чтобы увеличить процент разбираемых инструкций с 50% до 100, и выяснил, что для этого придется еще больше усилий. Декомпиляторы того времени оставляли более сложные случаи на усмотрение программиста. Для автоматизации разбора подобных случаев потребовалось бы затратить времени пропорционально больше тому, которое было затрачено на рассмотрение простых случаев.

#### У. Сассаман (1966)

Сассаман разработал декомпилятор в TRW Inc для упрощения переноса программ с компьютеров второго поколения на третье. Данный декомпилятор получал программы на ассемблере для IBM серии 7000 в качестве входных данных и преобразовывал их в Фортран. Ассемблер был выбран потому, что содержал больше полезной информации по сравнению с двоичным кодом, а Фортран был распространен на компьютерах второго и третьего поколений. Декомпилируемые программы представляли собой инженерные приложения с большим количеством алгебраических алгоритмов. Пользователь приходилось самому определять правила распознавания подпрограмм. Точность декомпиляции была около 90%, но требовала ручного вмешательства.

За счет использования ассемблера, содержащего полезную информацию в виде имен, макросов, данных и команд, недоступных в двоичном коде, была решена проблема отделения инструкций от данных.

#### Прототип К. Р. Холландера (1973)

В своей диссертации Холландера описывает декомпилятор на базе формального синтаксически-ориентированного метаязыка, состоящего из 5 последовательных процессов, каждый из которых реализован как интерпретатор набора метаправил: инициализатора, сканера, синтаксического анализатора, конструктора и генератора.

Инициализатор загружает программу и преобразует её во внутреннее представление. На основе полученного представления сканер находит инструкции, соответствующие шаблонам и отдает их синтаксическому анализатору. Синтаксический анализатор устанавливает соответствие между синтаксисом фраз в исходном языке и их семантическими эквивалентами в целевом языке. Используя результат работы анализатора, конструктор и генератор восстанавливают код программы.

Данный декомпилятор был реализован для трансляции языка System/360 на Алголоподобный язык. Он был написан для компилятора Algol-W, разработанного в Стэндфордском университете. 10 программ, на которых тестировался декомпилятор, были правильно восстановлены.

В этой работе представлен новый подход к декомпиляции с помощью формального синтаксически-ориентированного метаязыка. Главный недостаток данной методологии – необходимость шаблонов перевода ассемблерных команд в высокоуровневые инструкции. Для распознавания инструкции было необходимо, чтобы команды шли в определенном порядке, что ограничивало количество ассемблерных команд, которые могут быть декомпилированы. Распознать промежуточные инструкции, различные шаблоны потока управления, или оптимизированный код было невозможно. Для работы подобного синтаксически-ориентированного декомпилятора необходимо было описать множество всех возможных шаблонов для каждой высокоуровневой инструкции для всех компиляторов. Другим подходом являлась разработка декомпилятора под конкретный компилятор, основываясь на его спецификации. Холландер смог использовать данный подход при разработке декомпилятора, поскольку он имел доступ к спецификации Algol-W, так как это компилятор был написан в университете, где он занимался его исследованием.

#### Диссертация Хаузела (1973)

В кандидатской диссертации Хаузел описывает подход к декомпиляции, используя понятия из теории компиляторов, теории графов и теории оптимизации. Работа его декомпилятор состоит из 3-х шагов: частичной сборки, анализатора и генератора кода.

На этапе частичной сборки инструкций отделяются от данных, строится граф потока управления и создается промежуточное представление программы. Далее анализатор находит в программе циклы и устраняет ненужные промежуточные инструкции. Наконец, генератор кода оптимизирует трансляцию арифметических выражений, после чего генерирует код на целевом языке.

Экспериментальный декомпилятор был написан для декомпиляции ассемблера MIX (MIXAL) на PL/1 для IBM 370. При тестировании 6 программ 88% генерируемого кода были восстановлены правильно, а остальные 12% требовали ручного вмешательства.

Хаузел достиг больших успехов в разработке обобщенного декомпилятора, несмотря на серьезное ограничение по времени (он был написан за 5 человеко-месяцев). Он описывает ряд отображений данных (преобразований) в "Окончательное абстрактное представление" ("Final Abstract Representation") и обратно. Экспериментальный декомпилятор был расширен в работе Фридмана.

Данная разработка доказала, что если использовать методы известные из теории компиляции и теории графов, то можно сгенерировать хороший высокоуровневый код. Использование промежуточного представления позволяет абстрагироваться от конкретной архитектуры компьютера. Основным минусом этой методики является выбор исходного языка – MIX, являющегося упрощенным ассемблером, не используемым на практике.

#### Система Piler (1974)

Система Piler – это попытка разработать общий декомпилятор, позволяющий транслировать множество пар исходных и целевых языков для упрощения переноса компьютерных программ с одной платформы на другую. Система Piler работает в три этапа: интерпретация, анализ и преобразование. В результате несколько интерпретаторов могут быть написаны для разных исходных языков, а несколько конверторов для получения кода на целевых языках высокого уровня. Такой подход позволяет построить простой декомпилятор для пар исходных и целевых языков.

Во время интерпретации исходный машинный код программы загружается в память, анализируется и преобразуется в «микроформу» – 3-х адресную инструкцию. Это означает, что каждая машинная инструкция преобразуется в одну или несколько микроформ. Анализатор определяет логическую структуру программы путем анализа потока данных и транслирует микроформы в промежуточное представление. После этого анализа блок-схема программы доступна для пользователя, и он может изменить блок-схемы в случае обнаружения ошибок. Наконец, конвертер генерирует код на целевом языке программирования высокого уровня.

Несмотря на то, что система Piler была попыткой разработать общий декомпилятор, в итоге был разработан интерпретатор только для машинного языка GE/Honeywell 600, и скелет преобразователей для Univac 1108 в Fortran и Cobol. Основные усилия этого проекта были сосредоточены на анализаторе.

Система Piler была первой попыткой создать общий декомпилятор для широкого класса исходных и целевых языков. Основная проблема заключалась в попытке создать его достаточно общим, используя микроформы, которые были даже более низкоуровневым, чем ассемблер.

#### Ф. Л. Фридман (1974)

Кандидатская диссертация Фридмана описывает декомпилятор трансляции операционных систем мини-компьютеров в рамках одного архитектурного класса. В работе описаны четыре основных этапа: предварительная обработка, декомпилятор, генератор кода, и компилятор.

В начале, препроцессор преобразует код ассемблера в стандартную форму (описательный язык ассемблера). Декомпилятор получает этот описательный язык, анализирует ее и преобразует во внутреннее представление. Потом из него генерируется FRECL-код, после чего FRECL-компилятор компилирует FRECL-программу в машинный код для целевой платформы. FRECL – это язык высокого уровня для разработки и переноса программного обеспечения. Он был разработан Фридманом, также написавшим для него компилятор. Декомпилятор, использовавшийся в этом проекте был адаптацией декомпилятора Хаузела.

Были проведены два эксперимента. Первый предусматривал перенос небольшой, но закрытой части системы IBM 1130 Disk Monitor в Microdata 1600/21, где на вводимые ассемблерные программы потребовалось около 33% ручного вмешательства. В целом, объем работ, необходимых при подготовке кода для ввода в систему переноса был слишком велик, чтобы завершить его в течение разумного промежутка времени, поэтому было решено провести второй эксперимент. В его рамках программы операционной системы Microdata 1621 декомпилировались в FRECL, а затем – обратно в машинный код Microdata 1621. Некоторые из полученных программ вновь включили в операционную систему для проведения тестирования. В среднем лишь 2% от общего числа ассемблерных инструкций потребовали ручного вмешательства, но окончательная программа имела 194% увеличение числа машинных инструкций.

В данной диссертации впервые описываются трудности, с которыми приходится столкнуться при разработке декомпилятора, тесно взаимодействующего с машинно-зависимыми программами. Для представления программ в виде, соответствующем требованиям системы переноса, требовалось большое количество усилий, в то время как размер и время выполнения полученного кода были неоптимальными.

#### Ultrasystems (1974)

Согласно информации, полученной от консультанта по системному проектированию Ultrasystems Inc. Г.Л. Хопвуда, его компания вела разработку декомпилятора для программного обеспечения субмарин Trident. Данный декомпилятор использовался при составлении документации для кода систем ведения огня. В качестве входных данных он принимал ассемблерные программы Trident и декомпилировал их в код на языке высокого уровня Trident (Trident High Level Language), разрабатывавшийся в их компании. Были выделены четыре основных этапа: нормализация, анализ, упрощение выражений и генерация кода.

Входные программы на ассемблере нормировались так, что области данных были отделены от псевдо-инструкций, после чего генерировалось промежуточное представление, и проводился анализ данных. На третьем шаге восстанавливались арифметические и логические выражения и, наконец, результирующий код получалась путем сопоставления управляющих структур структурам, имеющимся в THLL.

Данный проект представлял собой попытку документирования ассемблерных программ, путем их перевода на язык высокого уровня. Основная проблема заключалась в том, что из-за временных ограничений проекта фаза упрощения выражений не была реализована до конца, и как следствие декомпилированный код было трудно читать.

#### В. Шнайдер и Г. Уиниджер (1974)

Шнайдер и Уиниджер опубликовали нотацию с подробным описанием компиляции и декомпиляции языков высокого уровня. Определив контекстно-свободную грамматику для процесса компиляции, статья описывала применение такой грамматики для получения исходного кода программы из объектного модуля. Более того, неоднозначная грамматика компиляции должна была производить оптимальный объектный код и приводить к генерации однозначной грамматики при декомпиляции. Проверка, проведенная позднее, показала, что объектный модуль, полученный из Algol 60, не может быть декомпилирован детерминировано.

В настоящей работе, представлен синтаксически ориентированный декомпилятор, то есть декомпилятор, который использует шаблон по серии инструкций для восстановления оригинальной исходной программы. В данном случае требуется знать грамматику процесса компиляции для того, чтобы, инвертировав ее, получить грамматику декомпиляции. Следует отметить, что проведение оптимизации невозможно в случае, если она не определена как часть грамматики компиляции.

#### Декомпиляция польского кода (1977, 1981, 1988)

В литературе можно найти две работы в области декомпиляции кода из польской формы в Basic код. Проблема возникает при разборе высоко интерактивных систем, в которых требуется быстрая реакция на каждое действие пользователя.

В первой работе программа пользователя находится в промежуточной форме, а затем "декомпилируется" каждый раз, когда действие инициируется пользователем. Подобный алгоритм перевода обратной польской нотации для выражений реализован в системе Piler.

Второй документ разбивает процесс декомпиляции на два этапа: перевод машинного кода в польское представление и преобразования польского кода в исходный код. Данный способ был использован в декомпиляторе, преобразовывающем обратный польский код в таблицу выражений. В этом случае, таблица использовалась, чтобы ускорить сохранение выражений от пользователя в обобщенной форме (в этом случае в обратную польскую запись) и декомпилировать эти выражения всякий раз, когда пользователь хочет их просмотреть или изменить.

Использование термина декомпиляция в данном случае не совсем корректно. Все, что было представлено в этих работах – это методы повторного построения или разбора исходного выражения на основе промежуточного программы в польской нотации. В работе не дано никаких объяснений как получить, упомянутое выше промежуточное представление, основываясь на машинном коде.

#### Кандидатская диссертация Хопвуда (1978)

В кандидатской диссертации Хопвуда описывается 7-ступенчатый ​​декомпилятор, написанный для облегчения процесса переноса программ и реверс инжиниринга при составлении документации. В ней также утверждалось, что в процессе декомпиляции может помочь ручное вмешательство или предоставление дополнительных внешних данных.

Программа, поступающая в декомпилятор, обрабатывается препроцессором, затем загружается в память, после чего строится граф ее потока управления. Узлы этого графа представляют собой одну инструкцию. После построения графа, распознаются шаблоны элементов управления, а инструкции, генерирующие оператор goto, устраняются либо с помощью разделения узлов, либо введением дополнительных переменных. Затем исходная программа преобразуется в промежуточный машинно-независимый код, который используется для определения выражений и удаления ненужных переменных методом прямого замещения. На последнем этапе генерируется конечный код для каждой промежуточной инструкции, создаются функции для операций, не поддерживаемых целевым языком, и добавляются комментарии. Ручное вмешательство требуется для подготовки исходных данных, предоставления декомпилятору дополнительной информацию, необходимой в процессе трансляции, а также для внесения изменений в целевую программу.

Экспериментальный декомпилятор был написан для машины Varian Data 620/i. Он восстанавливал ассемблер в MOL620, машинно-ориентированный язык, разработанный в Калифорнийском университете М.Д. Хопвудом и самим автором работы – Г.Л. Хопвудом. Декомпилятор был проверен на большой программе-отладчике Isadora. Получившийся код требовал дополнительной ручной корректировки перед повторной компиляцией из-за наличия самомодифицирующегося кода, использования дополнительных регистров для вызова подпроцедур и вызовов прерываний обслуживающих программ. Окончательная программа была лучше документирована, чем оригинальная программа на ассемблере.

Высокая степень детализации графа потока управления и использование регистров в сгенерированном коде – основные недостатки данного исследования. Первый был обусловлен тем, что строившиеся графы потока управления ставили в соответствие одному узлу одну команду. Это приводило к их заметному росту для больших программ, и в то же время не давало никакой ощутимой выгоды по сравнению с использованием базовых блоков в качестве узлов (то есть, когда размер узлов зависит от количества изменений потока управления). Второй недостаток был вызван тем, что MOL620 разрешал использование регистров, что подтверждается некоторыми примерами, приведенными в диссертации Хопвуда. Понятие регистра не является концепцией, присущей языкам высокого уровня, и оно не должно использоваться, при генерации кода на языке высокого уровня.

#### Д.А. Уоркман (1978)

Данная работа описывает использование декомпилятора при проектировании языка высокого уровня для систем обучения в реальном времени (в частности – тренировочных самолетов F4). Операционная система F4 была написана на ассемблере, поэтому языком ввода для декомпилятора послужил именно он. Выходной язык не был определен, так как целью этого проекта являлось построение определение необходимых правил его построения.

Работа декомпилятора состояла из двух этапов. Сначала ассемблер преобразовывался в промежуточный язык, а по исходной программе собиралась статистика. Далее генерировался граф потока управления, состоящий из базовых блоков, и проводился анализ потока управления для определения высокоуровневой структуры управления. Полученные результаты говорили о том, что был необходим язык высокого уровня, умеющий обрабатывать битовые строки, поддерживающий циклы и условные структуры управления, но не требующий динамических структур данных или рекурсии.

Данная работа представляет собой новый подход к применению методов декомпиляции. В рамках проекта был проведен простой анализ данных, классифицирующий используемые инструкции. В полном анализе не было никакой необходимости, т.к. задачи сгенерировать высокоуровневый код перед разработчиками не стояло. Анализ потока управления производился полностью и учитывал 8 различных видов циклов и 2 условных оператора.

### История декомпиляции (1980-1999)

#### Zebra (1981)

Прототип Zebra был разработан в Военно-морской Лаборатории Электроники для портирования ассемблерных программ. Zebra принимал на входе подмножество ассемблера ULTRA/32, называемое AN/UYK-7, и генерировал ассемблер для PDP11/70.

Декомпилятор производил работу в 3 прохода. В ходе первого прохода осуществлялся лексический анализ и анализ потока управления для получения графа базовых блоков. В рамках второго прохода программа переводилась в промежуточное представление, которое упрощалось на третьем проходе. Было решено, что разбор семантики программы слишком сложен и что дальнейшее использование декомпиляции экономически нецелесообразно, но может помочь в процессе переноса кода с одной ЭВМ на другую.

Проект использовал известные технологии для разработки декомпилятора. Хотя в исследовании не представлено никаких новых решений, в нем был поднят вопрос об использовании декомпиляции в качестве вспомогательного инструмента при решении определенных задач.

#### Декомпилятор Forth (1982, 1984)

Рекурсивный декомпилятор Forth является программой, которая рекурсивно просматривает словарь-таблицу и возвращаюет примитивы или адреса, связанные с каждым декомпилируемым словом. Данный декомпилятор считается одним из наиболее полезных инструментов Forth. В нем реализован нисходящий синтаксический анализатор, рекурсивно проходящий по декомпилируемым словам.

Данная работа представляет собой в большей степени инструмент обратного синтаксического анализа, а не декомпилятор. Инструмент.

#### Система Переноса Программного Обеспечения (1985)

Ч.В. Йо описывает автоматическую Систему Переноса Программного Обеспечения (Software Transport System, STS), которая портирует ассемблерный код с одного компьютера на другой. Процесс переноса включает в себя декомпиляцию ассемблерной программы в язык высокого уровня на одной машине и последующую компиляцию полученного кода на другой.

Экспериментальный декомпилятор был разработан для архитектуры Intel 8080. В качестве входных данных он принимал ассемблерный код, который декомпилировался в программу на языке PL/M. Перекомпилированная программа PL/M была на 23% эффективнее, чем ее исходная версия на ассемблере. Экспериментальный декомпилятор STS была разработан с целью создания кросс-компилятора языка Си для процессоров Z-80. Проект столкнулся с проблемой отсутствия типов данных в STS.

Как упоминалось выше STS получал в качестве входных данных ассемблерный код для одной машины и ассемблерную грамматику для другой, а на выходе генерировал ассемблерную программу для последней. Полученная на входе грамматика анализировалась для определения таблицы, используемой синтаксическим анализатором для разбора кода и генерации абстрактного синтаксического дерева (АСД) программы. Далее АСД поступало на вход декомпилятору, который затем разбирал поток управления и поток данных методами, описанными Холландером, Фридманом и Барбом, и в конечном итоге генерировал код на языке высокого уровня. Полученный код затем компилировался для целевой машины.

Описанная работа не является исследованием в области декомпиляции, но, тем не менее, в ней был использован по-настоящему новый подход к портированию программ ассемблерного кода с помощью описания грамматики ассемблера целевой архитектуры.

#### FermaT (1989)

Кандидатская диссертация Мартина Уорда является работой, формализующей процесс трансляции программ. Он написал FermaT – движок для преобразования программ, который облегчал прямой и обратный инжиниринг с языка ассемблера в спецификацию и обратно. Была произведена декомпиляция нескольких реальных ассемблерных программ, таких как IBM-370, на Си и COBOL, а впоследствии и с ассемблера 80186 на Си. В настоящее время эта технология распространяется компанией SoftwareMigrations.

#### exe2c (1990)

Компания Austin Code Works спонсировала создание декомпилятора exe2c для семейства PC-совместимых компьютеров под управлением операционной системы DOS. Проект был анонсирован в апреле 1990 года, после того, как его протестировало около 20 человек, было решено, что его необходимо доработать. Через год проект вышел в виде беты, но так никогда и не был закончен.

exe2c является многопроходным декомпилятором, состоящим из 3-х программ: e2a, a2aparse и е2с. e2a – это дизассемблер, преобразующий исполняемые файлы в ассемблер, и снабжающий комментариями ассемблерный листинг. a2aparse является транслятором ассемблера во front-end Си, анализирующим файл, созданный e2a, и генерирующим файлы .cod и .glb. Наконец, е2с транслирует файлы, подготовленные a2aparse, и генерирует код на псевдо-Си языке. Также в рамках проекта предоставлялась интегрированная среда envmnu.

Программы, декомпилированные exe2c, используют заголовочный файл, определяющий регистры, типы и макросы. Полученные в результате декомпиляции программы на языке Си трудно понять, потому что они используют регистры и коды условий (представленные в виде логических переменных). Как правило, одна машинная инструкция декомпилируется в одну или сразу несколько инструкций на Си, которые выполняют необходимые операции на регистрах и устанавливают коды условий, если этого требует инструкция. В окончательной версии кода используется локальный стек, а выражения и аргументы функций остаются неопределенными. Из этого можно сделать вывод, что в exe2c не был реализован анализ потока данных. В то же время анализ потока управления проводился, т.к. циклы и условные конструкции определялись, как правило, верно, но таблица case'ов все же составлялась неправильно. По количеству и типам декомпилированных процедур можно сделать вывод, что разбирался весь идентифицированный код, вплоть до библиотечных процедур, процедур запуска и поддержи времени выполнения.

exe2c является первой попыткой декомпиляции исполняемых файлов. Результаты показывают, что для получения лучшего кода на Си требовался анализ потока данных, дополнительная эвристика, механизмы, позволяющие определять подпрограммы и пропускать весь дополнительный код, произведенный компилятором.

#### Восстановление типов при декомпиляции. А. Майкрофт (1999)

Одной из самых трудных задач, которые приходится решать в процессе декомпиляции – это правильное восстановление высокоуровневых типов данных. ППод такими типами подразумеваются структуры, массивы и пр. В своей работе, Алан Майкрофт (Alan Mycroft) представляет систему для определения высокоуровневых типов в ассемблерном (RTL) коде. Система определения типов Майкрофта во многом основывается на работе Милнера для ML. Майкрофт описывает систему типов в целом, а также ограничения, накладываемые на конкретные типы, такие как структуры и массивы. Экспериментальные результаты работы системы остаются недоступными.

В настоящее время это лучшая система типов, которая описывает восстановление высокоуровневых типов данных и при этом остается машинно-независимой, благодаря тому, что основана на RTL'e и не допускает никаких необоснованных предположений о форме RTL. Реализация результатов необходима, чтобы определить, насколько эта система применима в реальной практике.

### История декомпиляции (2000-н/в)

#### Asm21toc (2000)

Asm21toc ­– это декомпилятор ассемблера для кода цифровой обработки сигнала (ЦОС). Авторы отмечают, что ЦОС является одной из немногих областей, где до сих пор широко используется ассемблер. Как уже отмечалось, декомпилировать ассемблер значительно проще, чем сам исполняемый код; фактически, авторы "сомневаются в полезности" декомпиляции бинарных файлов.

#### Декомпиляция низкоуровневого кода для проверки его правильности (2001)

Катцумата и Охори опубликовали работу по декомпиляции на основе теоретических методов математического доказательства. На вход декомпилятору подавался Jasmin, по сути, ассемблерный язык для Java. На выходе получался ML-подобный упрощенный функциональный язык. В их примере демонстрировалось, как реализация итеративной функции факториала превращается в две функции (эквивалент рекурсивной реализации). Особенность такого подхода состояла в обработке каждой инструкции как математического доказательства, представляющего свое вычисление. Хотя их работу нельзя применить непосредственно при написании общего декомпилятора, она может использоваться в случаях необходимости доказательства валидности (обычно на этапе компиляции).

В своей работе Майкрофт сравнивает свою Теорию декомпиляции типов с этим трудом. Ни в той, ни в другой не осуществляется структурирование циклов и условных операторов. Он приходит к выводу, что обе системы выдают очень похожие результаты в пересекающихся областях, и что их сильные и слабые стороны различаются.

#### Анализ компьютерной безопасности с помощью декомпиляции и высокоуровневой отладки (2001)

Чифунтес и ее коллеги предложили динамическую декомпиляцию как мощный инструмент для проведения работ в сфере обеспечения безопасности. Основная идея состоит в том, что небольшие фрагменты кода, которыми интересуется аналитик по вопросам безопасности, можно было бы декомпилировать в высокоуровневый язык "на лету". Одной из проблем традиционной (статической) декомпиляции является сложность определения диапазонов возможных значений переменных, в то время как динамическому декомпилятору достаточно обеспечить хотя бы одно такое (текущее) значение.

#### Распространение типов в дизассемблере IDA Pro (2001)

И. Гильфанов рассказывает о системе распространения типов (type propagation system) в популярном дизассемблере IDA Pro. В ней типы параметров библиотечных вызовов восстанавливаются из заголовочных фалов, а типы параметров для часто используемых библиотек сохраняются в файлах, называемых библиотеками типов. В местах, где происходит определение параметров, вставляются комментарии с указанием имени и типа параметра. Далее эта информация распространяется на другие части дизассемблируемой программы, в том числе и на все известные вызывающие функции. В настоящее время попытки найти типы переменных, не связанных с параметрами библиотечных вызовов, не предпринимаются.

#### DisC (2001)

DisC – декомпилятор, предназначенный только для чтения программ, написанных на Turbo C версии 2.0 или 2.01, является примером декомпилятора к компилятору. У такого подхода нет существенных преимуществ, так как сложность реализации общих методов декомпиляции не намного выше. Интересно, что поскольку большинство аспектов декомпиляции представляют собой сравнение с каким-либо эталоном, то основная разница между общими и частными декомпиляторами – это только тип шаблонов, которые используются при сопоставлении.

#### ndcc (2002)

Андре Янц (André Janz) изменил dcc для работы с 32-битными PE-файлами (Portable Executable). Цель этого проекта – разработка модифицированного декомпилятора для анализа вредоносного программного обеспечения. По словам автора, для полной реализации подмножества инструкций процессора 80386 необходимо было переписать декомпилятор. Тем не менее, несмотря на серьезные ограничения dcc, были достигнуты неплохие результаты.

#### Анализ косвенных вызовов при трансляции бинарных файлов(2002)

Трегер (Tröger) и Чифунтес (Cifuentes) продемонстрировали метод анализа косвенных вызовов. Если правильно опознать такой вызов как вызов виртуального метода, то из него можно извлечь большое количество информации различного сорта. Представленная техника ограничивается одним базовым случаем, и как результат, не применима для некоторых менее распространенных ситуаций.

#### Boomerang (2002)

Декомпилятор с открытым исходным кодом, с несколькими front-ends (два хорошо развиты) и back-end для языка Си. Он использует внутреннее представление, основанное на Static Single Assignment (SSA) и анализе типов на основе потока данных. Ссылка на сайт проекта: http://boomerang.sourceforge.net.

#### Desquirr (2002)

Плагин IDA Pro, написанный Дэвидом Эрикссоном (David Eriksson) в рамках магистерской диссертации. Хотя плагин не является полноценным декомпилятором, он показывает, что с помощью мощного дизассемблера и приблизительно 5000 строк кода на C++, можно достигнуть неплохих результатов. Так как дизассемблер не содержит семантики машинных команд, то для каждого поддерживаемого процессора требуется написать модуль для декодирования инструкций и способов адресации. Поддерживаются процессоры X86 и ARM. Условия и циклы, интерпретируются как goto, есть простой анализ переходов, некоторые возможности восстановления параметров и возвращаемых значений. http://desquirr.sourceforge.net/desquirr.

#### CodeSurfer (2004)

Балакришнан (Balakrishnan) и Репс (Reps) из Университета Висконсина разработали инфраструктуру для анализа бинарных программ, названную ими Codesurfer/x86. Целью являлось получение промежуточного представления, похожего на то, которое может быть создано для программы на языке высокого уровня. Вначале дизассемблер IDA Pro производит разбор бинарного файла, после чего для получения промежуточного представления применяется алгоритм анализа возможных значений (Value-set Analysis, VSA). Полученный результат представляется в CodeSurfer для последующего анализа. http://www.grammatech.com/research/technologies/codesurfer

#### Анализ типов для Декомпиляторов (2004)

В своей дипломной работе для Технического Университета Дрездена Р. Фальке (R. Falke), реализует адаптацию теории ограничения типов Майкрофта в декомпиляторе под названием YaDeC. Он расширил теорию для обработки массивов. Чтобы иметь возможность легко управлять проектом, он использовал objdump в виде front-end'а, игнорировал числа с плавающей точкой и допускал передачу параметров только через стек.

#### Andromeda (2004-2005)

Андрей Шульга (A. Shulga) написал декомпилятор для языка Си и платформы Windows x86. Конечный вариант декомпилятора так никогда и не был выпущен. В настоящее время написан только front-end для x86 и back-end для C/C++, хотя утверждается, что декомпилятор способен взаимодействовать и с другими front-end и back-end. Демонстрация работы впечатляет, но не ясно, генерируется ли она автоматически или же производится ручное вмешательство. Веб-страница неактивна с мая 2005 года.

#### Hex Rays Decompiler (2007)

И. Гильфанов, автор дизассемблера IDA Pro, выпустил коммерческий плагин для декомпиляции к IDA Pro. Этот плагин добавляет в программу различные инструменты декомпиляции. Просмотреть можно только одну функцию за раз; однако большинство функций декомпилируются в доли секунды, выдавая на выходе Cи-подобный код. Автор подчеркивает, что плагин написан как вспомогательный инструмент, и генерируемый им код не предназначен для повторной компиляции. Вывод включает в себя логические операторы (|| и &&), циклы (for, while, break и.т.д.), параметры функции и возвращаемые ими значения. Существует также API, который позволяет пользователю написать дополнительный функционал для расширения возможностей декомпилятора. На данном этапе, поддерживается только архитектура x86. Декомпилятор зависит от дизассемблера (и возможности ручного вмешательства), отделяющего код от данных и идентифицирует функции.

#### Static Single Assignment (2007)

В октябре 2007 года М. Ван Эммерик (M. Van Emmerik) закончил написание диссертации основной темой, которой было упрощение анализа различных аспектов декомпиляции с помощью SSA-представления. В работе также затрагивались проблемы анализа типов и анализа косвенных переходов и вызовов.

В работе описываются результаты декомпиляции с использованием Boomerang, представлен новый алгоритм для нахождения рекурсий, составлен общий глоссарий терминов, историография проблемы, проведены сравнения с декомпиляторами Java. Ван Эммерик провел исследование подходов компиляции, которые можно было бы использовать при декомпиляции, и описал такие проблемы декомпиляции, как отделение кода от данных, указателей от констант и смещений указателей.

Несмотря на значительные продвижения в области декомпиляции остается довольно много проблем связанных, в частности, с анализом псевдонимов.

### Обзор существующих декомпиляторов на язык Си

Все рассматриваемые декомпиляторы, кроме плагина Hex-Rays, на вход принимают исполняемый файл, и выдают программу на языке Си. В том случае, когда декомпилятор оказывается не в состоянии восстановить некоторый фрагмент исходной программы на языке Си, этот фрагмент сохраняется в виде ассемблерной вставки. Надо заметить, что даже небольшие исходные программы после декомпиляции зачастую содержат очень много ассемблерных вставок, что практически сводит на нет эффект от декомпиляции.

В отличие от этого, плагин Hex-Rays принимает на вход программу, являющуюся результатом работы дизассемблера IDA Pro, то есть схему программы на ассемблеро-подобном языке программирования. В качестве результата Hex-Rays выдает восстановленную программу в виде схемы на Си-подобном языке программирования.

#### Boomerangс

Декомпилятор Boomerang является программным обеспечением с открытым исходным кодом. Разработка этого декомпилятора активно началась в 2002 году, на данный момент есть только альфа версия. Начиная с 2006 года проект не разрабатывается. Изначально задачей проекта была разработка такого декомпилятора, который восстанавливает исходный код из исполняемых файлов, вне зависимости от того, с использованием какого компилятора и с какими опциями исполняемый файл был получен. Для этого в качестве внутреннего представления было решено использовать представление программы со статическими одиночными присваиваниями (SSA). Однако, несмотря на поставленную цель, в результате декомпилятор не сильно адаптирован под различные компиляторы и чувствителен к применению различных опций, в частности, опций оптимизации. Еще одной особенностью, затрудняющей использование декомпилятора Boomerang, является то, что в нем не поддерживается распознавание стандартных функций библиотеки Си.

#### DCC

Проект по разработке этого декомпилятора был открыт в 1991 году и закрыт в 1994 году с получением главным разработчиком степени PhD. В качестве входных данных декомпилятор DCC принимает 16-битные исполняемые файлы в формате DOS EXE. Алгоритмы декомпиляции, реализованные в этом декомпиляторе, основаны на теории графов (анализ потока данных и потока управления). Для распознавания библиотечных функций используется сигнатурный поиск, для которого была разработана библиотека сигнатур. Однако надо заметить, что, несмотря на это, декомпилятор плохо справляется с выявлением функций стандартной библиотеки.

#### REC

Этот проект был открыт в 1997 году компанией BackerStreet Software, но вскоре закрылся из-за ухода ведущего разработчика проекта. Позднее разработка декомпилятора продолжилась его автором в статусе собственного продукта. Сейчас декомпилятор распространяется свободно, а развивается медленно. Одной из особенностей рассматриваемого декомпилятора является то, что он восстанавливает исполняемые файлы в различных форматах, в частности ELF и PE. Также декомпилятор REC можно использовать на различных платформах. В ходе тестирования этого декомпилятора было отмечено, что наиболее успешно декомпилятор восстанавливает исполняемые файлы, полученные при компиляции с включением опций, которые отвечают за отключение оптимизаций и добавление отладочной информации.

#### Hex-Rays

Hex-Rays не является самостоятельным программным продуктом, а распространяется в виде плагина к дизассемблеру IDA Pro. Это самое новое из рассматриваемых средств декомпиляции: плагин появился на рынке в 2007 году. Особенностью данного инструмента является то, что он, как отмечалось, восстанавливает программы, полученные на выходе дизассемблера IDA Pro. Среди алгоритмов, используемых в Hex-Rays, заслуживают внимания алгоритм сигнатурного поиска FLIRT и алгоритм поиска параметров в стеке PIT (Parameter Identification and Tracking).

## Проблемы декомпиляции

В этой главе описываются проблемы, с которыми сталкиваются разработчики при разработке инструментов реверс инжиниринга. Главными источниками этих проблем являются:

1. **Потеря данных.** Исполняемые программы не содержат информации об именах переменных и процедур, комментариях, параметрах и возвращаемых значениях функций и типах данных. Для восстановления этой информации необходимо проводить анализ исходного файла.
2. **Смешение данных.** В бинарных файлах происходит смешение некоторых видов данных, например, программного кода и массивов данных, целых чисел и указателей. На этапе компиляции могут складываться указатели и смещения, в результате чего их тяжело отделить друг от друга. Требуется разделить эти смешанные данных, так как иначе, полученное представление декомпилируемой программы может быть неверным.

Несмотря на эти проблемы, есть ряд задач, которые проще решить, используя инструменты обратной инженерии, а не прямой. Например, в отличие от компиляторов, работающих с изолированными частями программ (модули), декомпиляторы видят программы целиком. В результате чего, можно произвести более качественный анализ потока данных и анализ псевдонимов, но это требует большого количества памяти для хранения внутренних связей всей программы.

Еще одна потенциальная область применения ‑ это анализ безопасности. Реверс инжиниринг работает непосредственно с тем, что будет выполнять процессор. Учитывая, что скомпилированная программа не всегда эквивалентна исходному коду, то при оценке её безопасности чаще используются инструменты, работающие непосредственно с исполняемыми файлами.

В таблице приведены проблемы, с которым приходится сталкиваться декомпиляторам разных типов:

* декомпиляторы байт-кода Java и CLI/CIL;
* декомпиляторы ассемблерного кода и объектных файлов;
* идеальный дизассемблер, преобразующий машинный код ассемблер, который можно заново собрать;
* идеальный декомпилятор машинного кода, который может представить исходную программу на высокоуровневом языке, который можно перекомпилировать;
* проблемы, которые были решены в декомпиляторах dcc и Reverse Engineering Compiler (REC)

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Проблема | Декомпилятор бинарного кода | Декомпилятор dcc | Декомпилятор REC | Декомпилятор объектных модулей | Декомпилятор ассемблерного кода | Идеальный дизассемблер | Декомпилятор байт-кода Java | Декомпилятор CLI |
| Отделение кода от данных | Да | частично | частично | частично | нет | да | нет | нет |
| Отделение указателей от констант | Да | нет | нет | легко | нет | да | нет | нет |
| Восстановление указателей, сложенных со смещением | Да | нет | нет | легко | нет | да | нет | нет |
| Объявление данных | Да | нет | нет | да | легко | да | легко | легко |
| Восстановление параметров и возвращаемых значений функций | Да | да | большая часть | да | да | нет | нет | нет |
| Анализ неявных переходов и вызовов | Да | нет | нет | да | да | да | нет | нет |
| Анализ типов | Да | нет | нет | да | частично | нет | для большинства локальных переменных | нет |
| Слияние инструкций | Да | да | да | да | да | нет | да | да |
| Восстановление управляющих конструкций | Да | да | да | да | да | нет | да | да |
| **Итог** | **9** | **3,5** | **3,25** | **6,5** | **4,5** | **5** | **2,5** | **2** |

### Отделение кода от данных

*Задача отделения кода от данных решается с помощью рекурсивного обходом потока данных и анализом неявных переходов и вызовов.*

Для обеспечения возможности повторной компиляции программы необходимо решить ряд проблем, первичная из которых – отделение кода от данных. Основная причина этой проблемы в принципе однородности памяти Фон Неймановской архитектуры, согласно которому построено большинство современных ЭВМ. Было доказано, что общее решение этой задачи по сложности эквивалентно решению проблемы останова. Названия секций исполняемого файла .text или .data никак не отражают их содержимого, поскольку и компиляторы, и программисты часто вставляют константные данные (например, текстовые строки или таблицы переходов) в секцию кода, или наоборот – программный код в секцию данных. Как результат, проблема отделения кода от данных остается актуальной.

Существует несколько способов решения данной проблемы. Самым мощным инструментом в арсенале статического декомпилятора является рекурсивный обход потока данных. Согласно ему декомпилятор совершает обход всех возможных путей обхода кода. Для реализации этого подхода нужно, чтобы выполнялись следующие условия:

* все пути обхода были валидные (маловероятно для обфусцированного кода);
* возможность нахождения точек входа;
* возможность анализа неявных переходов и вызовов, на основе анализа потока данных.

Одной из задач отделения кода от данных является выявление границ процедур. Такая проблема отсутствует в программах, использующих стандартные инструкции вызова и возврата. Можно выявить случаи оптимизации хвостовой рекурсии, когда пары инструкций call и return заменяются на jmp, который переходит на начало той же функции, что и инструкция call. Стоит учесть, что существуют такие компиляторы, как MLton, не использующие общепринятые нотации при вызове функции и возврате из нее значения. MLton генерирует код, используя Continuation Passing Style (CPS), который можно привести к стандартным вызовам. Описание процесса декомпиляции программ, написанных на функциональных языках программирования, не является основной задачей данной работы, но на первый взгляд выделение в них функций не должно вызывать никаких дополнительных проблем.

Очевидно, что для декомпиляторов байт-кода Java или CIL, проблема разделения кода и данных не актуальна.

### Отделение указателей от констант

Вторая по важности проблема – это отделение указателей от констант. При обнаружении любого текущего операнда, размером, равным размеру указателя, декомпилятору или дизассемблеру нужно решить: является этот операнд константой (целым числом, символом, др.) или указателем на какие-то данные, расположенные в памяти. Это позволит заменить адреса из оригинальной программы при ее перекомпиляции, оставив при этом нетронутыми константные значения.

### Восстановление указателей, сложенных со смещением

Восстановление указателей, сложенных со смещением, производится с помощью граничного анализа.

Декомпиляторы, работающие с машинным кодом, должны уметь определять базовые указатели, хранящие адрес на начало объекта с данными, и указатели, сложенные со смещением (например, хранящие адрес середины массива с данными). Стоит отметить, что наличие таблицы символов никак не поможет в данной ситуации. Проиллюстрируем это на следующем примере. Программа представлена в виде исходного кода на C:

#include <stdio.h>

int a[8] = {1, 2, 3, 4, 5, 6, 7, 8};

float b[8] = {1., 2., 3., 4., 5., 6., 7., 8.};

char\* str = "collision!";

int main()

{

int i;

for (i=-16; i < -8; i++)

printf("%-3d ", (a+16)[i]);

printf("\n");

for (i=-8; i < 0; i++)

printf("%.1f ", (b+8)[i]);

printf("\n");

printf("%s\n", str);

return 0;

}

и результата ее дизассемблирования в машинный код для архитектуры x86 с помощью IDA Pro:

mov eax, offset str

lea eax, [edx+eax]

mov eax, [eax]

push eax

push offset a3d ; "%-3d "

call sub\_8048324 ; printf

mov eax, offset str

lea eax, [edx+eax]

fld dword ptr [eax]

lea esp, [esp-8]

fstp [esp+24h+var\_24]

push offset a\_1f ; "%.1f "

call sub\_8048324 ; printf

mov eax, str

push eax

call sub\_8048304 ; puts

Главное, на что стоит обратить внимание – это использование в машинном коде одной и той же константы str для доступа ко всем трем массивам. В данной программе таблица символов не была разделена, что позволило дизассемблеру получить значения констант, которые используются в инструкциях, работающих с массивом. В то же время дизассемблер ошибочно посчитал указатель str основным для всех трех массивов. Единственный способ определить, что в первых двух случаях str является результатом сложения основного указателя со смещением, это проверить границы возможных значений для переменной i (регистр edx в машинном коде содержит i\*4).

При декомпиляция первых двух массивов в

((int\*)(str-16))[i]

и

((float\*)(str-8))[i]

соответственно, будет получено довольно точное представлением о том, как выглядела исходная программа. В некоторых ситуациях это код можно скомпилировать в работающую программу, однако качество самого полученного кода довольно низкое. Однако, поскольку компилятор может располагать данные в любом порядке и с любым выравниванием, которое будет требоваться в целевой архитектуре, str-16 и str-8могут перестать указывать на начала нужных массивов, если попытаться перекомпилировать программу для другой целевой платформы.

Без выделения основных указателей и указателей, сложенных со смещением, невозможно определить какие именно данные берутся из памяти, и в результате не получить все доступные определения переменных.

На первый взгляд, проблемы отделения указателей от констант и определения указателей, сложенных со смещением, могут быть объединены, так как обе проблемы возникают в результате потери информации после работы линкера, вычисляющего все возможные константные значения на этапе сборки. Несмотря на это, они рассмотрены по отдельности, поскольку пути их решения несколько различаются.

### Виды декомпиляторов

*Чем сильнее отличаются уровни абстракции между исходным кодом программы и кодом, в который она была скомпилирована, тем сложнее будет процесс ее восстановления. Следовательно, декомпиляция ассемблерного кода намного проще, чем восстановление бинарного.*

Из рисунка ниже видно, что в процессе компиляции исходного кода в ассемблер смешения данных не происходит. На данном этапе есть возможность сохранить простые комментарии (даже на уровне циклов, до и после вызовов процедуры и.т.д.). Более детальные комментарии, а так же управляющие конструкции (циклы, условные операторы), будут потеряны. Исходные выражения разбиваются на множество элементарных арифметических операций (сложение, умножение и т.д.). Еще могут оставаться представления некоторых высокоуровневых типов. Например, доступ к полю структуры Customer.address получается с помощью символов Customer (адрес структуры) и address (смещение для поля address). Массив чисел с плавающей точкой начинается с метки, а место под него выделяется ассемблерными командами, таким образом, несмотря на потерю информации о типе данных, которые в нем хранятся, имя массива и его размер все же сохраняются. Итак, можно сделать вывод, что в декомпиляции из ассемблерного кода, возникает при намного меньше проблем, чем при декомпиляцией из машинного представления.

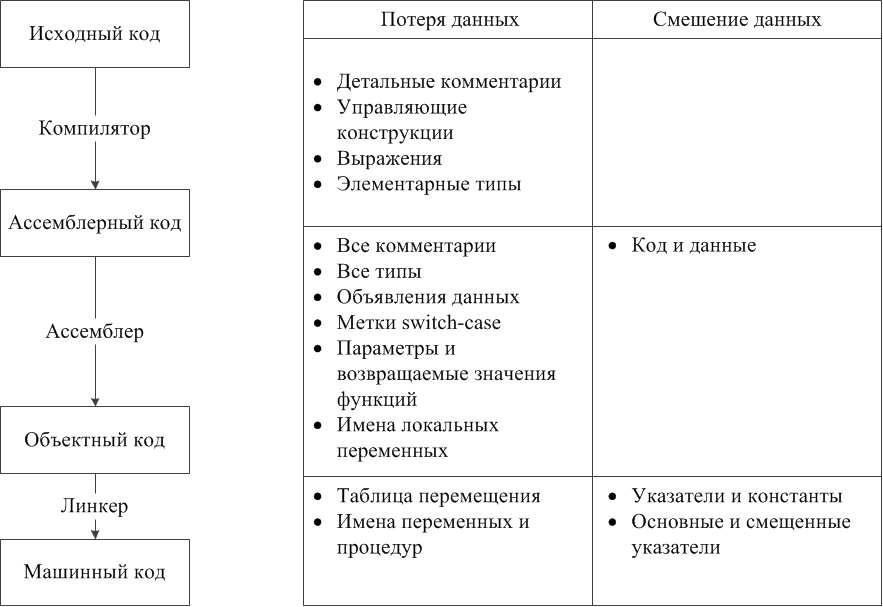


Рис. . Потеря информации в процессе компиляции

После ассемблирования теряются все комментарии, типы, метки переходов для switch-case и объявления данных. Так же на этом этапе смешиваются код и данные, будучи записанными в исполняемые секции объектного файла. Однако на уровне объектного фала еще можно воспользоваться настроечной информацией, которая не была удалена при линковке, для получения полезных сведений о размещении объектов. Например, таблица указателей, будет содержать адреса перемещения, идущие подряд, что явно не похоже на исполняемый код. Конечно, такой подход не поможет определить структуры данных, которые могут и не содержать указатели. По этой причине в таблице сверху для декомпиляторов объектных файлов проблема отделения кода от данных обозначена, как частичная.

После линковки теряются имена переменных и функций (если удаляется символьная и отладочная информация) так же, как полезная таблица перемещений. Из-за того, что процессор одинаково работает с указателями и целыми числами, то для отделения констант от указателей приходится проводить дополнительный анализ. При восстановлении из объектного файла такая проблема решалась бы с помощью таблицы перемещения, но т.к. машинный код ее не содержит, приходится выяснять используется ли результат вычислений для доступа к памяти или нет. После определения указателей так же нужно выяснить, являются ли они базовыми или смещенными. Это более сложные проблемы, которые не могут быть решены для всех возможных случаев.

#### Дизассемблеры

Для создания идеального декомпилятора на базе идеального дизассемблера нужно решить следующие задачи:

* восстановление параметров и возвращаемых значений функций;
* объявление типов с помощью анализа типов;
* слияние ассемблерных инструкций для получения сложных арифметических выражений (уже решено в декомпиляторах Java и CLI с помощью анализа потока данных);
* построение циклов и определение условных переходов (уже решено в декомпиляторах Java и CLI).

Другими словами при наличии идеального дизассемблера его расширение до идеального компилятора – выполнимая задача, т.к. все из вышеперечисленных проблем либо уже решены на практике, либо имеют теоретически обоснованное решение, которое можно реализовать.

#### Декомпиляторы ассемблерного кода

Среди всех видов декомпиляторов с наименьшим количеством проблем сталкиваются те, что принимают на вход только ассемблерный код. Основная причина заключается в том, что не нужно заниматься отделением кода от данных. Более того, здесь присутствуют комментарии и понятные названия переменных, функций, указателей и их смещений. Анализ типов и объявление данных осуществляются довольно просто. Более того, есть имена и размеры объектов, так что выделение основных и смещенных указателей тоже не представляет собой серьезную проблему. В листинге представлен ассемблерный код для программы из листинга выше с явными метками a, b и str.

movl $a+64, %eax

leal (%edx,%eax),%eax

movl (%eax), %eax

pushl %eax

pushl $.LC1 ;"%-3d "

call printf

movl $b+32, %eax

leal (%edx,%eax),%eax

fld (%eax)

leal -8(%esp), %esp

fstpl (%esp)

pushl $.LC2 ; "%.1f "

call printf

movl str, %eax

pushl %eax

call puts

Декомпиляторам ассемблерного кода все же приходится решать самые серьезные задачи обратного инжиниринга: определение параметров и возвращаемых значений функций, слияние ассемблерных инструкций для восстановления сложных арифметических выражений и построение управляющих конструкций. Символы, содержащиеся в ассемблерных программах, позволяют однозначно идентифицировать составные типы вроде массивов и структур, но для определения типов данных, которые в них хранятся, нужно проводить анализ типов.

Все описанное выше справедливо для стандартного языка ассемблера, содержащего символы для каждого элемента данных. В некоторых случаях сложно сгенерировать код, который был бы пригоден для повторной компиляции. Например, когда адреса данных в программе согласованы с какой-либо другой программой. В качестве примера можно привести дизассемблированый код бинарной программы. Декомпилятор ассемблерного кода довольно редко примеряется для таких случаев, и поэтому в дальнейшем не рассматривается.

#### Декомпиляторы объектного кода

Преимущества разбора объектных файлов (o. или .obj), заключается в том, что кроме машинного кода они содержат символьную информацию и таблицу перемещения. Наличие данной информации существенно упрощает отделение указателей от констант и основных указателей от смещенных. В листинге показан результат дизассемблирования объектного файла, сгенерированного для программы из листинга сверху. Как и в прошлом примере, здесь приводятся только части, соответствующие подчеркнутым строчкам в листинге.

mov eax, (offset a+40h)

lea eax, [edx+eax]

mov eax, [eax]

push eax

push offset a3d ; "%-3d "

call printf

mov eax, (offset b+20h)

lea eax, [edx+eax]

fld dword ptr [eax]

lea esp, [esp-8]

fstp [esp+24h+var\_24]

push offset a\_1f ; "%.1f "

call printf

mov eax, str

push eax

call puts

Как видно, здесь все еще содержатся ссылки на символы a, b и str.

#### Декомпиляторы байт-кода Java и CLI

Декомпиляторы данного типа достигли больших успехов, благодаря тому, что они работают с файлами, содержащими подробные метаданные, в отличие от бинарных программ. Например, байт-код Java имеет следующие преимущества по сравнению с машинным кодом:

* данные отделены от кода;
* определение констант и указателей является очень простой задача, благодаря наличию отдельных инструкций для работы со ссылками (getfield, new);
* параметры и возвращаемые значения определяются явно;
* в декодировании какой-либо глобальной информации нет необходимости, потому что вся она содержится в классах, поля-члены классов могут быть получены непосредственно из байт-кода;
* анализ типов нужно проводить только для локальных переменных, т.к. несмотря на наличие отдельных инструкций для целых чисел, ссылок, чисел с плавающей запятой, нельзя получить точную информацию с каким именно подтипом ведется работа (например, bool, short или int).

Таким образом, декомпиляторам подобного вида остается решить лишь две основные проблемы: слияние инструкций и восстановление управляющих конструкций.

Единственная дополнительная задача, которую потребуется решить – это преобразование инструкций Java (или CIL), ориентированных на работу со стеком, в стандартные инструкции процессора по работе с регистрами. Хотя реализация такого алгоритма довольно проста, для этого потребуется создавать временные переменные, хранящие последнее значение стека. Некоторые декомпиляторы игнорируют данную проблему, что приводит к ошибкам при декомпиляции оптимизированного байт-кода.

В настоящее время декомпиляторы данного вида довольно распространены благодаря относительной простоте их реализации и востребованности технологий .NET и Java.

#### Недостатки существующих реализаций декомпиляторов

Все реализации декомпиляторов, имеющиеся на данный момент имеют проблемы при идентификации параметров и возвращаемых значений функций, определении неявных вызовов и переходов. Анализ типов проводится ими в неполном виде, а максимальный размер программ, принимаемых на входе, сильно ограничен. В качестве примера возьмем два известных декомпилятора: REC и dcc. REC – некоммерческий декомпилятор, восстанавливающий получаемые программы в C-подобный код. Способен работать с несколькими видами бинарных файлов. dcc – декомпилятор, написанный в исследовательских целях, принимает на вход только программы DOS для архитектуры 80286 и восстанавливает их код на C.

|  |  |  |
| --- | --- | --- |
| Проблемная область | **dcc** | **REC** |
| Обработка больших программ | Нет | Да, но не проводит серьезного межпроцедурного анализа |
| Определение параметров и возвращаемых значений функций | Да, но проводит межпроцедурный анализ только для регистров, а для параметров на стеке делает предположения | Результаты непостоянны |
| Непрямые переходы | Сравнимает шаблоны на уровне промежуточного представления; сообщает о необходимости определения «дополнительных идиом» | Результаты непостоянны |
| Неявные вызовы | Указатели на функции определяются, как константы; высокоуровневый код не восстанавливается | Высокоуровневый код не восстанавливается |
| Анализ типов | Восстанавливает пары регистров для long int’ов; не восстанавливает числа с плавающей запятой, массивы данных, структуры | Не восстанавливает числа с плавающей запятой, массивы данных, структуры |

### Теоретические пределы

Большинство фундаментальных проблем декомпиляции по сложности сопоставимы с проблемой останова, и как следствие не разрешимы. Согласно теореме Райса, нельзя восстановить все нетривиальные свойства компьютерных программ, что означает существование теоретических пределов для компиляторов и других инструментов, используемых в программной инженерии.

Компилятор всегда может избежать проблем, связанных с такими ограничениями, выбрав консервативный подход. Например, если ему не удается определить, является ли переменная константой, он не будет осуществлять непосредственную подстановку ее значения. Результатом будет работающая программа, которая работает медленнее и потребляет больше памяти, чем могла бы при применении соответствующей оптимизации. Подобные частные случаи теоретических ограничений компилятора могут быть преодолены с помощью дополнительного анализа. Однако наличие теоретического предела подразумевает невозможность создания такого компилятора, который смог бы реализовать полностью оптимизированный результат для всех возможных программ, подаваемых ему на вход.

В отличие от компилятора, из-за таких теоретических ограничений декомпилятор не сможет определить, что текущее значение является адресом процедуры. Консервативным подходом в данном случае было бы считать данное значение целочисленной константой, при этом найти способ убедится в том, что все адреса процедур декомпилированной программе соответствуют тем же адресам в исходной, или что существует инструкция перехода, которая перенаправит поток управления на функцию с таким адресом. В конце концов, даже если значение останется целым числом (при этом, использующимся где-то еще и как указатель), то от такого представления не будет вреда. Подобные решения является более радикальным шагом, и могут привести к более серьезным последствиям, чем небольшая потеря в производительности, как это произошло в случае с компилятором.

Похожая ситуация возникает при анализе параметров, передаваемых в функцию. Конечно, можно воспользоваться верным, но абсолютно нечитабельным вариантом и передавать все возможные переменные в функцию. В худшем случае, декомпилятор может вернуться представлению, которое он получил от дезассемблера.

В случаях, когда принимается решение пренебречь корректностью кода в угоду его читаемости, компилятору необходимо уведомлять об этом пользователя.

# АРХИТЕКТУРА ДЕКОМПИЛЯТОРА

## Фазы декомпиляции

Структура декомпилятора, также как и структура компилятора, состоит из фаз, последовательно преобразующих программу из одного представления в другое. Логические фазы, из которых состоит декомпилятор, приведены на рисунке ниже.

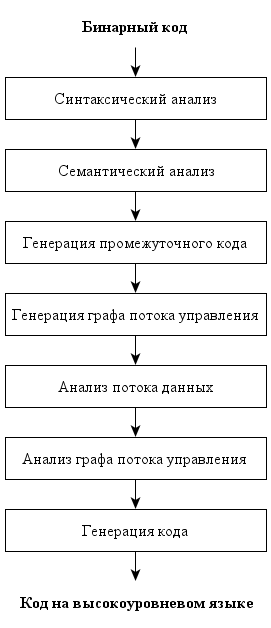


Рис. . Фазы декомпилятора

Можно заметить, что в отличие от компилятора, здесь нет лексического анализа или фазы сканирования. Причина в том, для разбора машинных языков не нужно проводить сложный анализ. Тем не менее, основная проблема заключается в отсутствии точного способа, определяющего начало и конец инструкции. Например, байт 0x50 может быть опкодом для инструкции *push ax*, константой или смещением.

### Синтаксический анализ

Парсер или синтаксический анализатор группирует байты исходной программы в грамматические фразы или предложения исходного машинного языка. Результат работы этой фазы можно представлены в виде дерева разбора. Например, выражение *sub cx, 50* семантически эквивалентно *cx := cx – 50*, и может быть представлено в виде следующего дерева разбора:



Рис. . Дерево разбора выражения sub cx, 50

Выражение разбивается на 2-е фразы: *cx – 50* и *cx := <exp>*.

Самая главная проблема в синтаксическом анализаторе ‑ это отделение данных и инструкций. Например, таблица переходов switсh’а может быть расположена в секции кода, а декомпилятору заранее неизвестно, что это данные, а не инструкции. Следовательно, последовательно разбирать машинный код нельзя, поскольку следующий байт может быть данными, а не инструкцией.

Например, рассмотрим последовательность байт **CE XX E8 61 06**. (второй байт может принимать любое значение). Как показано на рисунке ниже, в зависимости от того, как на нее будет передано управление, могут быть получены разные последовательности инструкций. Поэтому дизассемблеры, которые рассматривают инструкции последовательно, не смогут распознать случай, когда второй байт является байтом данных. Из-за этой ошибки все остальные инструкции будут распознаны неверно.

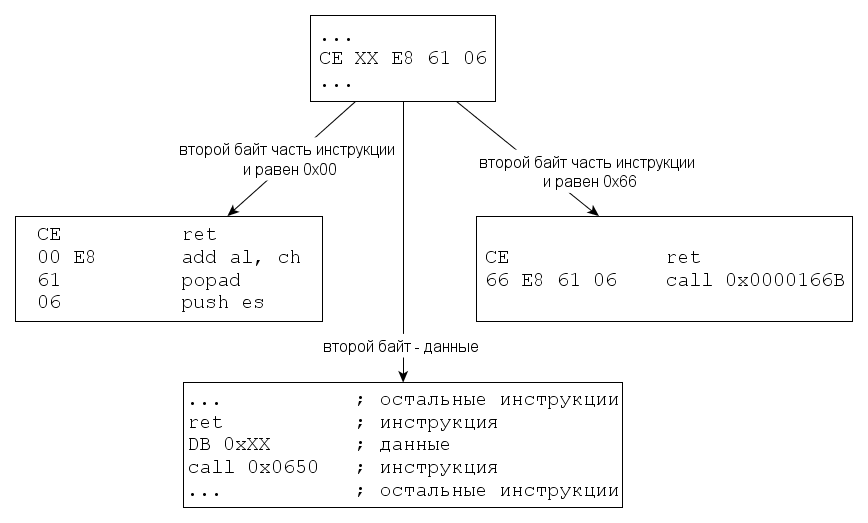


Рис. . Влияние передачи управления на разбор инструкции

### Семантический анализ

В процессе семантического анализа декомпилятор выявляет группы семантически связанных инструкций, собирает информацию о типах и занимается распространением информации о типах в подпрограммы. Очень редко возникают ситуации, когда компилятор неправильно генерирует машинных код, так что можно считать, что программы, сгенерированные компилятором семантически корректны. Следовательно, семантические ошибки могут появиться только из-за неправильного отделения инструкций от данных.

Для проверки семантического значения группы инструкций ищутся идиомы. Например, следующий код:

shl ax, 2

может быть преобразован в семантически эквивалентную инструкцию умножения регистра *ax* на 4. Другой пример – сложение переменных:

add ax, [bp-4]

add ax, [bp-2]

В этом случае, [bp-2]:[bp-4] представляют собой long переменную, которая складываются с регистрами dx:ax и сохраняется в те же регистры. Следует понимать, то, что здесь пара регистров dx:ax совместно используются для хранения переменной типа long, не означает, что в остальной части программы они должны рассматриваться вместе.

Информация о типах, которую удалось получить за счет анализа выражений, распространяется по всему графу. Так, в предыдущем примере две переменные, расположенные на стеке, являются одной переменной типа long, следовательно, все подпрограммы, использующие эти значения из стека, должны преобразовывать их в long переменную. Например, если после рассмотренного кода будут идти следующие инструкции:

asgn[bp-2], 0

asgn[bp-4], 14h

то, за счет распространения long типа [bp-2] и [bp-4] инструкции могут быть представлены в виде:

asgn [bp-2]:[bp-4], 14h

Наконец, несмотря на то, что компилятор не может сгенерировать семантически неверный код, могут возникнуть ошибки при работе программы на архитектурах, отличных от той, под которую программа была собрана изначально.

Рассмотрим программу, собранную под архитектуру i80286. Архитектуры i80386 и i80486 базируются на архитектуре i80286 и их бинарные коды хранятся в том же виде. Однако, более новые архитектуры используют большее число регистров и инструкций в отличие от i80268. Например, инструкция

add ebx, 20

является некорректной для старых процессоров, т.к. на них не существует 32-х битных регистров. Таким образом, хотя инструкция корректна синтаксически, она некорректна семантически.

### Генерация промежуточного кода

Для анализа программы декомпилятору нужно иметь промежуточное представление исходной программы. Оно служит промежуточным звеном между исходной программой и ее декомпилированным представлением. Формат промежуточного представления должен быть достаточно простым, чтобы исходную программу можно было в него легко перевести и, в то же время, получить из него код на целевом языке. Далее будет описано промежуточное представление в виде 3-х адресных инструкций, которое хорошо справляется с поставленной задачей. Все операнды этих инструкций эквивалентны машинным, но их можно представить в подходящем виде, для использования в высокоуровневых выражениях.

### Генерация графа потока управления

Декомпилятор должен уметь выделять высокоуровневые управляющие конструкции. Обычно для решения этой задачи строится граф потока управления, строящийся на основе инструкций перехода. Так же, используя это представление, можно удалить лишние промежуточные инструкции перехода. Например, в следующем коде:

... ; other code

jne x ; x <= maximum offset allowed for jne

... ; other code

x: jmp y ; intermediate jump

... ; other code

y: ... ;final target address

инструкция условного перехода jne x ограничена максимально допустимым смещением, на которое она может перейти, поэтому она не может сразу передать управление на метку y. В результате приходится использовать промежуточный переход jmp y. В графе потока управления этот промежуточный переход можно убрать, заменив jne x прямым переходом на метку y.

### Анализ потока данных

Фаза анализа потока данных нужна для улучшения промежуточного представления и распознания высокоуровневых выражений. Во время исполнения этой фазы удаляются временные регистры и флаги условий, поскольку они концептуально не относятся к высокоуровневым языкам. Рассмотрим следующую последовательность инструкций на промежуточном языке:

asgn ax, [bp-0Eh]

asgn bx, [bp-0Ch]

asgn bx, bx \* 2

asgn ax, ax + bx

asgn [bp-0Eh], ax

Те же самые инструкции, представленные в терминах высокоуровневого представления:

asgn [bp-0Eh], [bp-0Eh] + [bp-0Ch] \* 2

В исходном варианте используются регистры, переменные на стеке и константы. Здесь каждое выражение может быть представлено в виде дерева с максимальной глубиной 2. После анализа полученная инструкция использует переменные на стеке ([bp-0Eh], [bp-0Ch]) и дерево выражения глубиной 3: [bp-0Eh] := [bp-0Eh] + [bp-0Ch] \* 2. Временные регистры ax и bx, используемые в машинном коде, были удалены из высокоуровневого выражения.

### Анализ графа потока управления

Фаза анализа графа потока управления направлена на выделение типичных высокоуровневых управляющих конструкции. Не следует использовать специфичные конструкции, привязанные к конкретному языку. Нужно, чтобы управляющие инструкции, из которых состоят эти типичные управляющие конструкции, могли быть представлены на большинстве высокоуровневых языков. Ниже показаны два простых графа потока управления: while и if-then-else. Далее будут рассмотрены алгоритмы, структурирующие произвольные графы потока управления.

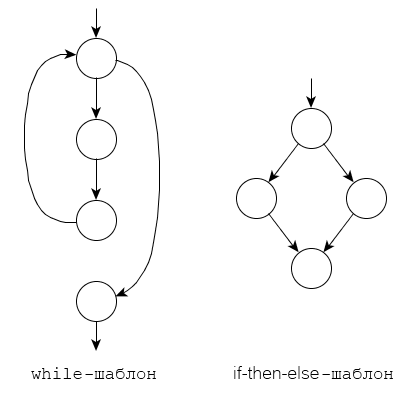


Рис. . Примеры графов потока управления

### Генерация кода

На последней фазе декомпилятор генерирует код на целевом языке, используя граф потока управления и промежуточное представление. Всем локальным переменным на стеке, аргументам и регистрам присваиваются имена. Также присваиваются имена каждой функции, найденной в программе. Управляющие конструкции из промежуточного представления преобразуются в высокоуровневые конструкции. Например, при трансляции на язык Си локальным переменным [bp-0Eh] и [bp-0Ch] из предыдущей главы могут быть даны имена loc1 и loc2. В результате получится следующее выражение:

loc2 = loc2 + (loc1 \* 2)

## Группировка фаз

При реализации декомпилятора, фазы из предыдущей секции, группируются по трем отдельным модулям: front-end, модуль машинно-независимой декомпиляции и back-end.

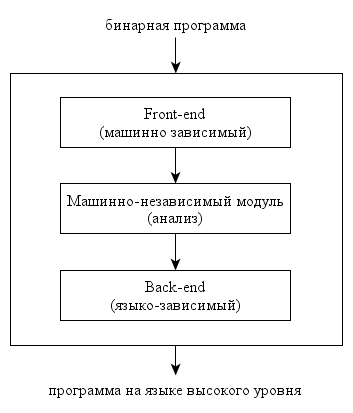


Рис. . Сгруппированные фазы

* модуль front-end содержит фазы, зависящие от конкретной платформы и машинного языка. К этим фазам относятся: семантический анализ, синтаксический анализ, генерация промежуточного представления и генерация графа потока управления. В результате работы этого модуля будет получено промежуточное машинно-независимое представление.
* модуль машинно-независимой декомпиляции работает с промежуточным представлением и является ядром декомпилятора. Содержит в себе фазы анализа потока данных и анализа графа потока управления.
* модуль back-end выдает код на целевом высокоуровневом языке. Содержит фазу генерации кода.

В теории компиляторов, группировка фаз используется для того, чтобы была возможность генерировать программы под разные платформы. Этот механизм позволяет компилятору изменять разные части независимо друг от других. Например, если потребуется добавить новую целевую платформу, то это не затронет front-end, а если нужно добавить новый язык программирования, то не будет затронут back-end. Хотя, есть некоторые ограничения, присущие этому методу.

В теории, группировка фаз упрощает декомпилирование различных платформ и языков путем написание дополнительных модулей front-end и back-end. На практике, результаты декомпиляции будут зависеть от используемого промежуточного представления.

## Вспомогательный инструментарий декомпилятора

При создании декомпилятора можно воспользоваться несколькими внешними программами. Например, при загрузке в память исходной программы для нее настраивается таблица переадресации. Декомпилятор для решения этой задачи может воспользоваться загрузчиком. Дальше машинные команды с настроенными относительными или абсолютными адресами декомпилируются и представляется в виде ассемблерного кода. Декомпилятор может воспользоваться сигнатурами компиляторов и библиотек для определения входной точки и определению библиотечных функций. Затем полученная ассемблерная программа подается на вход декомпилятору и он генерирует высокоуровневый ассемблерный код. Дополнительные процедуры над полученной программой, например преобразование while-а в for могут быть сделаны постпроцессором. Этапы работы декомпилятора показаны на рисунке ниже. Следует отметить, что пользователь так же может быть своего рода источником информации, особенно при определении библиотечных функций и отделении кода от данных. Однако, надежнее использовать автоматические инструменты везде, где это возможно.

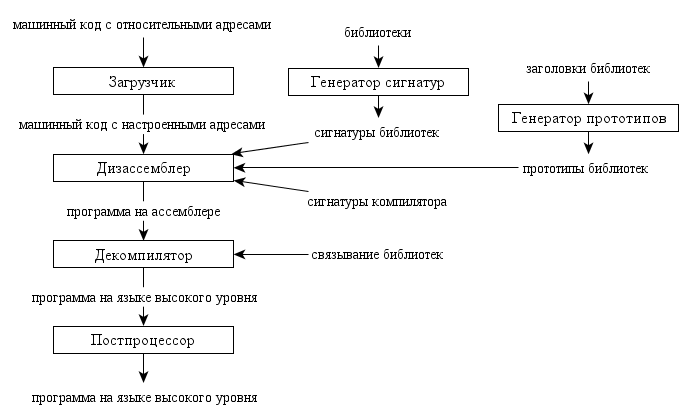


Рис. . Этапы работы декомпилятора

### Загрузчик

Загрузчик загружает программу в память и настраивает адреса машинных команд. В процессе загрузки настраиваются относительные адреса и заносятся обратно в память.

### Генератор сигнатур

Генератор сигнатур – это программа, которая автоматически определяет сигнатуры компилятора и библиотек по шаблонам, однозначно их идентифицирующим. Используя эти сигнатуры, декомпилятор пытается выделить библиотечные функции. Фактически, производится работа, обратная той, которую делает линковщик, собирающий вместе библиотечные функции для последующей компиляции. В результате, будут выделены участи программы, содержащие только пользовательские процедуры, которые и требуется декомпилировать.

Например, программа, выводящая “hello, world” содержит более 25 различных подпрограмм в бинарной программе: 16 из них добавлены для настройки окружения, 9 процедур являются частью printf, и только одна пользовательская функция.

Использование генератора сигнатур не только уменьшает число функций в программе, но и улучшает ее читаемость: за счет подстановки конкретных имен функций вместо абстрактных, автоматически генерируемых декомпилятором.

### Генератор прототипов

Генератор прототипов ­ это программа, которая автоматически определяет тип аргументов библиотечных функций и тип возвращаемого значения. Эти прототипы извлекаются из заголовочных файлов и используются декомпилятором для определения типа и количества аргументов.

### Дизассемблер

Дизассемблер – это программа, которая преобразует машинный язык в ассемблерный.

### Связывание библиотек

Как уже было упомянуто ранее, при генерации кода на целевом языке полезно определять библиотечные функции и подставлять их имена вместо полного разбора. Однако, в тех случаях, когда целевой язык отличается от оригинального языка, на котором написана декомпилируемая программа, сигнатуры библиотечных функций будут отличаться. Чтобы решить эту проблему, можно предопределить какие функции оригинального языка соответствуют аналогичным функциям целевого языка.

### Постпроцессор

Предназначен для преобразования высокоуровневых конструкций в рамках одного языка. Например, если целевой язык – Си, то следующий код:

loc1 = 1;

while (loc1 < 50)

/\* some code in C \*/

loc1 = loc1 + 1;

}

может быть представлен препроцессором в следующем виде:

for (loc1 = 1; loc1 < 50; loc1++) {

/\* some code in C \*/

}

который семантически эквивалентен изначальному варианту, однако использует управляющую конструкцию for, специфичную языку Си, отсутствующую в списке управляющих конструкций общего вида у декомпилятора.

# МЕТОДЫ ДЕКОМПИЛЯЦИИ

## Восстановление функций

### Выделение функций

Одной из основных структурных единиц программ на языке Си являются функции, которые могут принимать параметры и возвращать значения. Откомпилированная программа, однако, состоит из потока инструкций, функции в котором никак структурно не выделяются. Как правило, компиляторы генерируют код с одной точкой входа в функцию и одной точкой выхода из функции. При этом в начало кода, генерируемого для функции, помещается последовательность машинных инструкций, называемая прологом функции, а в конец кода – эпилог функции. Прологи и эпилоги функций, как правило, стандартны для каждой архитектуры и лишь незначительно варьируются. Например, стандартный пролог и эпилог функции для архитектуры i386 показаны ниже:

* пролог:

pushl %ebp

movl %esp, %ebp

* эпилог:

movl %ebp, %esp

popl %ebp

ret

Прологи и эпилоги функций могут быть легко выделены в потоке инструкций. Кроме того, при работе с потоком инструкций можно считать, что инструкции, на которые управление передается с помощью инструкции call, являются точками входа в функции, а инструкции ret завершают функции. Тем не менее, нельзя считать инструкции, расположенные между прологом и эпилогом, или между точками входа и выходом, телом функции, по ряду причин. Во-первых, при компиляции программы могут быть указаны опции, влияющие на форму пролога и эпилога функции. Например, опция компилятора GCC -fomit-frame-pointer подавляет использование регистра %ebp в качестве указателя на текущий стековый кадр, когда это возможно. В этом случае пролог и эпилог функции будут, как таковые, отсутствовать.

movl 4(%esp), %edx

movi 8(%esp), %eax

imull %eax, %eax

imull %edx, %edx

addl %edx, %eax

ret

Во-вторых, отдельные оптимизационные преобразования могут разрушать исходную структуру функций программы. Очевидным примером такого преобразования является встраивание тела функции в точку вызова: встроенная функция не существует как отдельная структурная единица программы, и ее автоматическое выделение представляется затруднительным.

Существуют оптимизирующие преобразования, которые приводят к появлению в машинном коде конструкций, принципиально невозможных в языках высокого уровня. Таким оптимизирующим преобразованием является, например, sibling call optimization. Если список параметров двух функций идентичен, и первая функция вызывает вторую с этими параметрами, то инструкция вызова подпрограммы call может быть преобразована в инструкцию безусловного перехода jmp в середину тела второй функции. Пример такой оптимизации представлен ниже. Функция \_fоо возвращает значение функции \_f, которая вызывается с теми же параметрами, что и функция \_fоо. Компилятор сгенерировал пролог и эпилог для функции \_fоо, а вызов функции \_f заменил безусловным переходом в середину ее тела.

\_f:

pushl %ebp

movl %esp, %еbр

movl 12(%ebp), %eax

addl 8(%ebp), %eax

leave

ret

\_foo:

pushl %ebp

movl %esp, %ebp

leave

jmp \_f

Результатом такого рода «неструктурных» оптимизаций будет появление переходов из одной функции в другую, появление функций с несколькими точками входа или несколькими точками выхода.

Другим источником «неструктурных» конструкций в машинной программе являются операторы обработки исключений в таких языках, как Си++.

Получается, что хотя в типичном случае компилятор генерирует хорошо структурированный, поддающийся разбиению на функции, код, достаточно легко может быть получен и код «неструктурированный».

Следует отметить, что в этом случае влияние программиста, пишущего программу на языке Си, на структуру генерируемого кода ограничено возможностями языка Си, не позволяющего бесконтрольной передачи управления между функциями и не поддерживающего механизм исключений. Принимая это во внимание, можно предположить, что если восстанавливаемая программа получена в результате компиляции кода на языке Си, то она не содержит «неструктурных» особенностей, описанных выше, и может быть разбита на функции.

### Выявление параметров и возвращаемых значений

В языках высокого уровня, в частности Си, поддерживается передача параметров в функции и возврат значений. В языке Си существует только передача параметров по значению, в других языках могут поддерживаться и другие механизмы. Далее рассматриваются только механизмы передачи параметров, отображаемые в генерируемый машинный код. Передача параметров по имени, передача параметров в шаблоны и другие механизмы периода компиляции программы здесь не рассматриваются.

Способы передачи параметров и возврата значений для каждой платформы специфицированы и являются составной частью так называемого ABI (Application Binary Interface). Под платформой понимается тип процессора и тип операционной системы, например, Win32/i386 или Linux/x86\_64. Одной из задач ABI является обеспечение совместимости по вызовам приложений и библиотек, скомпилированных разными компиляторами одного языка или написанных на разных языках.

Так, для платформы win32/i386 используется несколько соглашений о передаче параметров. Соглашение о передаче параметров \_cdecl используется по умолчанию в программах на Си и Си++ и имеет следующие особенности:

1. Параметры передаются в стеке и заносятся в него справа налево (то есть первый в списке параметр заносится в стек последним).
2. Параметры выравниваются в стеке по границе 4 байт, и адреса всех параметров кратны 4. То есть параметры типа char и short передаются как int, но и дополнительное выравнивание для размещения, например, double не производится.
3. Очистку стека производит вызывающая функция.
4. Регистры %eax, %ecx, %edx и %st(0) – %st(7) могут свободно использоваться (не должны сохраняться при входе в функцию и восстанавливаться при выходе из нее).
5. Регистры %ebx, %esi, %edi, %ebp не должны модифицироваться в результате работы функции.
6. Значения целых типов, размер которых не превосходит 32 бит, возвращаются в регистре %eax, 64-битных целых типов – в регистрах %eax и %edx, вещественных типов – в регистре %st(0).
7. Если функция возвращает результат структурного типа, то место под возвращаемое значение должно быть зарезервировано вызывающей функцией. Адрес этой области памяти передается как (скрытый) первый параметр.

Помимо соглашения \_cdecl также используются соглашения о передаче параметров \_stdcall и Pascal. В стандартном соглашении \_stdcall стек после вызова функции очищает сама вызываемая подпрограмма. Параметры в стек заносятся в том же порядке, что и при соглашении \_cdecl — справа налево. Стандартное соглашение о передаче параметров \_stdcall используется при вызове функций Win32 API. Стандартное соглашение о передаче параметров Pascal имеет также ряд отличий от соглашения \_cdecl, в частности, параметры заносятся в стек слева направо. Также как и при соглашении \_stdcall вызываемая функция не очищает стек после вызова.

Особенности стандартных соглашений о передаче параметров приведены в таблице:

|  |  |  |
| --- | --- | --- |
| Название | Порядок занесения параметров на стек | Очистка стека |
| \_cdecl | Заносятся в стек справа налево | Производится вызывающей програм­мой |
| \_stdcall | Заносятся в стек справа налево | Производится самой вызываемой под­программой |
| Pascal | Заносятся в стек слева направо | Производится самой вызываемой под­программой |

Некоторая путаница возникает с порядком занесения на стек параметров callback- функций. Традиционно callback-функции вызываются с соглашением Pascal, то есть параметры заносятся в стек слева направо и стек очищает сама вызываемая функция, однако компиляторы MCVS для callback-функций используют соглашения \_stdcall, по которому стек также очищает вызываемая функция, но параметры заносятся в обратном порядке. В последних версиях компилятора MCVS использование соглашения \_stdcall вместо соглашения Pascal явно написано в заголовочном файле <windef.h>:

#define CALLBACK \_\_stdcall

#define PASCAL \_\_stdcall

Подобные особенности компиляторов должны учитываться при восстановлении программ.

Следует отметить, что данный набор правил – это соглашения, которые «добровольно» выполняются в сгенерированном коде. Пока речь не заходит об интерфейсе с независимо скомпилированными сторонними модулями, программист может в определенной мере модифицировать эти правила, существенно затрудняя задачу автоматического восстановления функций.

Опять же можно предполагать, что если программа декомпилируется из автоматически полученного ассемблерного кода, то в ней используются только соглашения о передаче параметров из некоторого предопределенного множества. Причем в одной программе для разных функций не могут использоваться разные соглашения о передаче параметров, кроме тех случаев, когда программист явно указывает отличный тип соглашения.

На первом этапе решения задачи выявления параметров функций следует определить следующие особенности вызова функций:

1. *Используемое соглашение о передаче параметров.* Требуется определить, какое соглашение из набора предопределенных соглашений используется в программе.
2. *Размер области параметров функции.* Почти все соглашения о передаче параметров могут быть достаточно надежно идентифицированы по используемым инструкциям. Так, соглашение о передаче параметров stdcall требует, чтобы параметры из стека удалялись вызываемой функцией. Для этого может использоваться единственная инструкция системы команд i386 – ret N, где N – размер удаляемых из стека параметров. Таким образом, использование этой инструкции для возврата из функции указывает как на соглашение о передаче параметров, так и на размер параметров функции.

В случае вызова функции по указателю при статическом анализе нам может быть неизвестен адрес вызываемой функции. В этом случае не представляется возможным отследить, как возвращается управление из вызываемой функции. Определение соглашения о вызовах тогда должно быть отложено на фазы последующего анализа.

Итак, на фазе выявления параметров и возвращаемых значений определяется размер передаваемых в функцию параметров и способ возврата значения из функции. В дальнейшем эта информация используется как начальная при восстановлении символических имен и восстановлении типов.

### Распознавание библиотечных функций

При восстановлении библиотечных функций следует различать два случая компоновки программы: динамической и статической.

Если программа скомпонована динамически, то ее взаимодействие с окружением включает в себя обращения к внешним библиотекам. Для этого в специальных разделах исполняемого файла (например, для PE-файла это каталог импорта, таблицы просмотра импорта и адресов импорта) указываются требуемые динамические библиотеки и идентификаторы функций, а также переменных в них. Это может существенно упростить анализ программы, так как интерфейсы динамических библиотек, как правило, стандартны. Например, если некоторая программа, скомпонованная для ОС Linux, использует функцию printf из динамической библиотеки libc.so, то можно предполагать, что вызывается стандартная функция printf языка Си, которой передаются соответствующим образом подготовленные параметры, и проводить дальнейший анализ программы с учетом этой информации.

Если программа скомпонована статически, то анализ взаимодействия программы можно начать с анализа системных вызовов.

Системный вызов— это функция операционной системы, которая выполняется в режиме ядра. Параметры системных вызовов передаются либо в регистрах, либо в стеке. Системные вызовы легко идентифицируется в потоке инструкций. Например, в ОС Linux системный вызов традиционно выполняется с помощью инструкции int 0x80, причем в регистре %еах передается номер системного вызова. Таким образом, посредством анализа значений, загруженных на регистры или на стек, можно восстановить параметры системного вызова.

В случае статически скомпонованной программы тела стандартных библиотечных функций находится непосредственно в исполняемом файле программы и, как правило, информация о названии функции не сохраняется. Декомпиляция функций стандартных библиотек нежелательна по тем же причинам, что и декомпиляция запускающего кода: стандартные функции не изменяются от программы к программе и могут содержать низкоуровневые особенности работы с окружением. Тела стандартных функций в восстанавливаемой программе должны быть идентифицированы и соответствующим образом отмечены.

Следует также заметить, что в статически скомпонованных программах бинарный код одной и той же библиотечной функции в разных программах может незначительно отличаться из-за того, что функции размещены по разным виртуальным адресам и, как следствие, адреса вызываемых функций или глобальных переменных могут изменяться.

Главные трудности при распознавании библиотечных функций заключаются в следующем:

1. Количество таких функций велико, как следствие, объем памяти для библиотек всех версий всех моделей всех популярных производителей компиляторов займет гигабайты памяти, даже если не учитывать периодически выходящих исправлений библиотек типа OWL, MFC, BFC и т.п.
2. Наличие байтов, изменяемых при загрузке программы в память. В основном изменяемые байты происходят из-за наличия ссылок на внешние имена. В таком случае при компиляции программы компилятор еще не знает адреса вызываемой функции и оставляет эти байты равными нулю, записывая в выходной файл так называемую таблицу переадресации (relocation table, также известную, как fixup information или relocation information). При создании выполняемого файла компоновщик (linker) пытается разрешить внешние ссылки, подставляя вместо нулей адреса вызываемых функций, но некоторая часть байтов все равно остается неизвестной, например, ссылки на внешние динамические библиотеки и байты, содержащие абсолютные адреса в программе. Такие ссылки могут быть разрешены только при загрузке программы в память для выполнения. Этим занимается часть операционной системы – системный загрузчик (system loader), который должен разрешить все внешние ссылки. Если даже после загрузки программы в память остаются неразрешенные ссылки (т.е. ссылки на неизвестные имена функций), то такая программа выполняться не может.

Для решения перечисленных проблем при поиске фиксированных или слабо изменяющихся фрагментов кода используется сигнатурный метод. Для всех функций из всех библиотек всех производителей составить базу данных и проверять каждый байт дизассемблируемой программы на возможность начала стандартной функции с этого байта.

Все информация для распознавания функций хранится в сигнатурном файле. Каждая функция представляется шаблоном. Шаблон - это первые 32 байта функции с пометкой всех изменяемых байтов. Например (".." - это изменяемые байты):

558BEC0EFF7604..........59595DC3558BEC0EFF7604..........59595DC3 \_registerbgidriver

558BEC1E078A66048A460E8B5E108B4E0AD1E9D1E980E1C0024E0C8A6E0A8A76 \_biosdisk

558BEC1EB41AC55604CD211F5DC3.................................... \_setdta

558BEC1EB42FCD210653B41A8B5606CD21B44E8B4E088B5604CD219C5993B41A \_findfirst

Как видно, многие функции начинаются с одинаковых байтов. Поэтому строится дерево следующего вида:

558BEC

0EFF7604..........59595DC3558BEC0EFF7604..........59595DC3 \_registerbgidriver

1E

078A66048A460E8B5E108B4E0AD1E9D1E980E1C0024E0C8A6E0A8A76 \_biosdisk

B4

1AC55604CD211F5DC3 \_setdta

2FCD210653B41A8B5606CD21B44E8B4E088B5604CD219C5993B41A \_findfirst

Построение дерева позволяет достичь двух целей:

1. Сэкономить память.
2. Увеличить скорость сопоставления: количество сравнений, необходимых для сопоставления адреса в программе с базой сигнатур, находится в логарифмической зависимости от количества функций.

### Обнаружение функции main

При загрузке программы на выполнение операционная система передает управление инструкции программы, которая размещена по определенному виртуальному адресу. Этот адрес называется точкой входа программы и либо является фиксированным для данной операционной системы, либо доступен в служебной информации, размещенной в соответствующих секциях исполняемого файла. В качестве примера можно привести поле e\_entry в исполняемом файле в формате ELF или поле AddressOfEntryPoint в исполняемом файле в формате РЕ.

Однако во всех существующих реализациях функция main не совпадает с точкой входа, сгенерированного исполняемого модуля. При компоновке программы добавляется специальный запускающий код (start up code), который выполняет инициализацию среды выполнения Си, который при декомпиляции в большинстве случаев не представляет интереса, так как является стандартным для каждой операционной системы. При декомпиляции программы от точки входа запускающий код восстанавливается как часть программы. Запускающий код манипулирует деталями низкоуровневого интерфейса операционной системы, и сам может быть частично написан на ассемблере. С точки зрения практической значимости декомпиляции пользовательского кода запускающий код должен быть опущен, и декомпиляция должна начинаться с функции main.

Для обнаружения начала пользовательского кода, то есть функции main для языка Си, на практике используют два подхода:

1. Непосредственно перед вызовом функции main в стек передаются ее аргументы arge, argv, envp. Таким образом, подпрограмму, вызванную сразу после инструкции засылки аргументов на стек, можно считать началом пользовательского кода.
2. Каждый компилятор генерирует характерный ассемблерный код, который можно найти сопоставлением шаблонов, если есть библиотека шаблонов запускающего кода для компиляторов.

К недостаткам первого подхода можно отнести то, что может оказаться сложно рас­познать передачу параметров, кроме того, те же параметры могут передаваться в другие вспомогательные функции. Второй подход более универсален, но его недостатком является то, что соответствующая библиотека шаблонов должна постоянно пересматриваться каждый раз при выходе новых версий компиляторов или появлении нового компилятора.

## Восстановление управляющих конструкций

В основе восстановления управляющих конструкций лежит построение графа потока управления и его последующий анализ. Теория графов применяется в программировании с самого начала возникновения ЭВМ. Очень удобно выражать задачи обработки информации на теоретико–графовом языке. В заметке Р.Карпа (1960 г., на русском — 1962 г.) была введена в практику теоретико-графовая модель программы в виде управляющего графа. Данная модель стала к настоящему времени классической для решения задач трансляции и конструирования программ.

В графе потока управления каждый узел (вершина) графа соответствует базовому блоку — прямолинейному участку кода, не содержащему в себе ни операций передачи управления, ни точек, на которое управление передается из других частей программы. Имеется лишь 2 исключения: точка, на которую выполняется переход, является первой инструкцией в базовом блоке, и базовый блок завершается инструкцией перехода. Направленные дуги используются в графе для представления инструкций перехода. Также в большинстве реализаций добавлено два специализированных блока: входной блок, через который управление входит в граф, и выходной блок, который завершает все пути в данном графе.

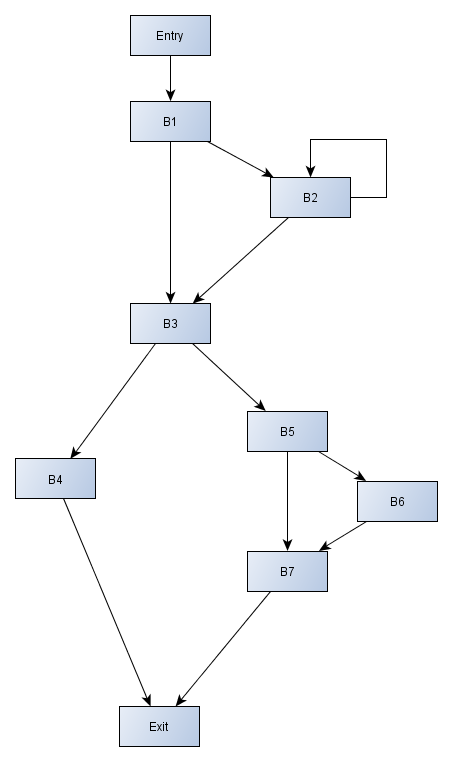


Рис. . Пример графа потока управления

### Сводимые и несводимые графы

Современные языки программирования поддерживают стандартный набор управляющих конструкций (if-then, do-while и пр.), которые сводят необходимость использования оператора goto к минимуму. Данные структурные конструкции порождают специфические графы потока управления. Например, оператор if-then-else порождает фрагмент графа, изображенный на рисунке ниже, где на месте блоков then и else в свою очередь могут находиться другие структурные подграфы. Проанализировав структурные управляющие конструкции языков программирования, можно для каждой конструкции выписать шаблон графа потока управления.

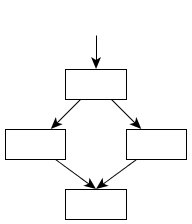


Рис. . Шаблон для структурной конструкции типа if-then-else

Если рассматривать произвольный граф потока управления функции, то имея набор шаблонов, можно попытаться наложить их на данный граф потока управления. Далее каждый наложенный шаблон заменяется новым абстрактным узлом. При проведении таких замен может возникнуть две ситуации:

1. Граф потока управления свернётся в одну вершину. Такие графы потока управления называются *сводимыми*.
2. В противном случае, мы можем обнаружить в графе область, не соответствующую ни одному из шаблонов. Такая область называется *несобственной* (*improper region*), а весь граф – *несводимым*.

Очевидно, что графы потоков управления программ, не использующих оператор goto, всегда сводимы. Тем не менее, с помощью только оператора goto также можно писать программы со сводимым графом потока управления. Заметим также, что одной из распространенных причин возникновения несобственных областей является передача управления в середину цикла.

Синтаксис некоторых языков программирования (таких как, например, Java, Modula-2) позволяет написание только таких программ, графы которых сводимы. Это же верно во многих других языках до тех пор, пока мы не используем оператор goto (а особенно goto внутрь циклов). Таким образом, хоть программы с несводимыми графами встречаются довольно редко, нужно уметь корректно обрабатывать подобные ситуации. Существует два основных способа, применяемых для анализа графов программ, содержащих несводимые области.

• Избавление от несобственных областей. Стандартный способ такого преобразования приведен на рисунке ниже. Недостатком такого способа является существенное увеличение количества вершин в графе.

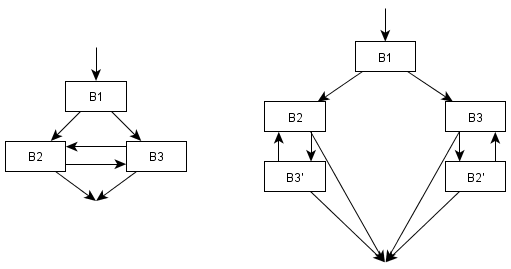


Рис. . Избавление от несобственных областей

• Выделение несводимой области и замена ее соответствующим абстрактным узлом.

### Шаблоны графа потока управления

#### if-then-шаблон

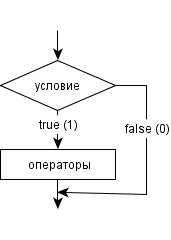


Рис. . if-then-шаблон

В начале проверяется значение внутри блока сравнения (condition), и если оно равно false, то происходит переход в конец.

Пример кода на Си:

if(x == 0)

{

x = 1;

}

x++;

Трансляция кода на ассемблере:

mov eax, $x

cmp eax, 0

jne end

mov eax, 1

end:

inc eax

mov $x, eax

Список команд на ассемблере, используемых для организации перехода по условию:

|  |  |
| --- | --- |
| *Инструкция* | *Значение* |
| JNE | **J**ump if **n**ot **e**qual |
| JE | **J**ump if **e**qual |
| JGE | **J**ump if **g**reater or **e**qual |
| JG | **J**ump if **g**reater |
| JLE | **J**ump if **l**ess than or **e**qual |
| JL | **J**ump if **l**ess than |
| JNGE | **J**ump if **n**ot **g**reater than or **e**qual |
| JGE | **J**ump if **g**reater than or **e**qual |
| JNLE | **J**ump if **n**ot **l**ess than or **e**qual |
| JLE | **J**ump if **l**ess than or **e**qual |

#### if-then-else-шаблон

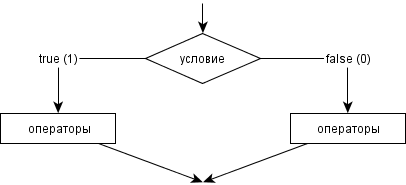


Рис. . if-then-else-шаблон

Как и в случае if-then-шаблона, если условие отрицательно, то происходит переход по условию else. Однако, здесь так же есть безусловный переход в конец после блока "then" (на картинке – action).

Пример кода на Си:

if(x == 10)

{

x = 0;

}

else

{

x++;

}

Трансляция кода на ассемблере:

mov eax, $x

cmp eax, 0x0A ;0x0A = 10

jne else

mov eax, 0

jmp end

else:

inc eax

end:

mov $x, eax

Как можно заметить, в отличие от кода в случае if-then-шаблона, здесь присутствует безусловный переход.

#### switch-шаблон

Представление структуры switch-case на ассемблере может быть очень сложным. В Си существует несколько ключевых слов, используемых в операторе switch:

* switch - это ключевое слово используется для указания аргумента, который надо проверить и означающее начало switch'а;
* case - создает метку, по которой надо перейти в случае совпадения параметра со значением аргумента в switch;
* break -используется для перехода в конец блока switch;
* default - метка на которую осуществляется переход, если значение аргумента в switch не совпало ни с одним значением case меток. Рассмотрим пример реализации switch'а. Код на Си:

int main(int argc, char \*\*argv)

{ //10 строка

switch(argc)

{

case 1:

MyFunction(1);

break;

case 2:

MyFunction(2);

break;

case 3:

MyFunction(3);

break;

case 4:

MyFunction(4);

break;

default:

MyFunction(5);

}

return 0;

}

Трансляция кода на ассемблере:

tv64 = -4 ; size = 4

\_argc$ = 8 ; size = 4

\_argv$ = 12 ; size = 4

\_main PROC NEAR

; Line 10

push ebp

mov ebp, esp

push ecx

; Line 11

mov eax, DWORD PTR \_argc$[ebp]

mov DWORD PTR tv64[ebp], eax

mov ecx, DWORD PTR tv64[ebp]

sub ecx, 1

mov DWORD PTR tv64[ebp], ecx

cmp DWORD PTR tv64[ebp], 3

ja SHORT $L810

mov edx, DWORD PTR tv64[ebp]

jmp DWORD PTR $L818[edx\*4]

$L806:

; Line 14

push 1

call \_MyFunction

add esp, 4

; Line 15

jmp SHORT $L803

$L807:

; Line 17

push 2

call \_MyFunction

add esp, 4

; Line 18

jmp SHORT $L803

$L808:

; Line 19

push 3

call \_MyFunction

add esp, 4

; Line 20

jmp SHORT $L803

$L809:

; Line 22

push 4

call \_MyFunction

add esp, 4

; Line 23

jmp SHORT $L803

$L810:

; Line 25

push 5

call \_MyFunction

add esp, 4

$L803:

; Line 27

xor eax, eax

; Line 28

mov esp, ebp

pop ebp

ret 0

$L818:

DD $L806

DD $L807

DD $L808

DD $L809

\_main ENDP

На строке 10 происходит установка стандартного стекового кадра и сохранение ecx. С помощью команды "push ecx" компилятор резервирует место на стеке: создает локальную переменную. В оригинальном коде не Си нет никакой локальной переменной, однако компилятору нужно место для сохранения промежуточных результатов. Компилятор не использует более привычную команду "sub esp, 4" для создания локальной переменной, потому что "push ecx" быстрее. Данное "рабочее пространство" располагается по отрицательному смещению от ebp, а переменная tv64 со значением -4, определенная в начале и используемая в выражении "tv64[ebp]", ссылается на это рабочее пространство.

Метка $L803 является меткой конца switch'а. Таким образом, каждый раз вызов команды "jmp SHORT $L803" означает break. Метка $L818 содержит жестко закодированные адреса, которые являются метками в сегменте кода.

Рассмотрим 11-ую строку:

mov eax, DWORD PTR \_argc$[ebp]

mov DWORD PTR tv64[ebp], eax

mov ecx, DWORD PTR tv64[ebp]

sub ecx, 1

mov DWORD PTR tv64[ebp], ecx

cmp DWORD PTR tv64[ebp], 3

ja SHORT $L810

mov edx, DWORD PTR tv64[ebp]

jmp DWORD PTR $L818[edx\*4]

На псевдо-Си ее можно записать следующим образом:

if( argc - 1 >= 4 )

{

goto $L810;

}

// Определение таблицы с метками, указывающими на case-секции

label \*L818[] = { $L806, $L807, $L808, $L809 };

// Таблица с адресами используется, чтобы перейти на требуемый case

goto L818[argc - 1];

#### do-while-шаблон

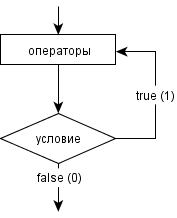


Рис. . do-while-шаблон

Принцип работы: если условие в конце цикла равно true, то происходит переход в начало цикла, иначе - программа выполняется дальше.

Код на языке Си:

do

{

x++;

} while(x != 10);

Трансляция на ассемблер:

mov eax, $x

:beginning

inc eax

cmp eax, 0x0A ;0x0A = 10

jne beginning

mov $x, eax

#### while -шаблон

Цикл while очень похож на цикл do-while, но с небольшими отличиями. Рассмотрим общий пример цикла while:

while(x)

{

//тело цикла

}

Вначале производится проверка равенства значения x true. Если это не так, то не происходит исполнения кода внутри тела цикла. После исполнения тела цикла производится еще одна проверка на равенство x true. Если да, то происходит переход в начало цикла. Дополнительная проверка в конце введена для того, чтобы избежать двух лишних переходов в случае, если x равен false. Таким образом, выполнение цикла while состоит из следующих шагов:

1. проверка условия, если оно равно false, то производится переход в конец;
2. выполнение тела цикла;
3. проверка условия. Если оно равно true, то перейти к шагу 2;
4. если условие не равно true, то идет исполнение кода после цикла.

Код на языке Си:

while(x <= 10)

{

x++;

}

Трансляция на ассемблер:

mov eax, $x

cmp eax, 0x0A

jg end

beginning:

inc eax

cmp eax, 0x0A

jle beginning

end:

Если же теперь произвести повторную трансляцию на язык Си, то получим следующий код:

if(x <= 10)

{

do

{

x++;

} while(x <= 10)

}

#### for-шаблон

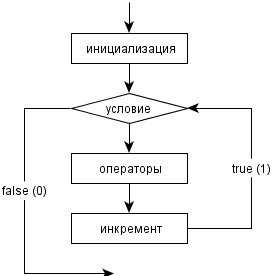


Рис. . for-шаблон

Цикл for – это цикл while с блоком инициализации и итеративной инструкцией. Рассмотрим общий вид цикла:

for(initialization; condition; increment)

{

action;

}

Он транслируется в следующий псевдокод с циклом while:

initialization;

while(condition)

{

action;

increment;

}

Результат в свою очередь транслируется в следующий цикл do-while:

initialization;

if(condition)

{

do

{

action;

increment;

} while(condition);

}

Следует отметить, что очень часто первоначальное значение в блоке инициализации (например, x = 0) меньше значения в блоке условия (например, x < 10). В этом случае компиляторы отмечают, что на момент первой проверки x меньше чем 10 и она на самом деле не нужна. В этом случае они генерируют следующий код:

initialization;

do

{

action

increment;

} while(condition);

### Методы анализа управляющих конструкций

#### Доминирующие множества

Анализ потока управления, основанный на построении доминирующих множеств вершин графа потока управления, позволяет работать только с циклами и без существенных модификаций неприменим для выявления условных операторов.

Перед тем, как рассмотреть этот метод требуется ввести некоторые определения. Дальше под управляющим графом будем понимать ориентированный граф *G =(V,E,r)* с выделенной начальной вершиной *r* (вход), в которую не входит ни одна дуга. Каждая вершина *G* достижима из *r*. Вершина *x* доминирует над вершиной *y* (*x* — обязательный предшественник вершины *y*), если любой путь в *G* из *r* в *y* проходит через *x*. Непосредственным доминатором вершины *w* называется вершина *v* такая, что *v* доминирует над *w* и любой доминатор вершины *w* доминирует над *v*.

В этом методе сначала вычисляется доминирующее множество вершин. Для вычисления доминирующего множества вершин используется либо алгоритм последовательной итерации, либо алгоритм Ленгауэра-Тарьяна. После того как доминирующее множество найдено, выполняется разметка дуг графа потока управления. Дуги графа потока управления классифицируются на прямые, обратные и косые. Прямая дуга — это дуга из доминирующей вершины в доминируемую, обратная дуга — это дуга из доминируемой вершины в доминирующую, все прочие дуги помечаются как косые.

Размеченный граф потока управления позволяет выделить управляющие конструкции, например, обратной дуге т —> п соответствует цикл, состоящий из вершины п и всех вершин, из которых доступна т по пути, не содержащему п. В самом общем случае циклу в исходной программе соответствует компонента сильной связности графа потока управления.

#### Интервальный анализ

Интервальный анализ – одновременно название методов анализа потока данных и потока управления. Применительно к анализу потока управления интервальный анализ означает разбиение графа потока управления на области различного вида, называемые регионами, т.е. набор базовых блоков, имеющий не более одной входящей дуги. Простейшим случаем региона является базовый блок. На первой итерации работы алгоритма все базовые блоки помечаются как самостоятельные регионы. Для выделения регионов строится дерево обхода графа потока управления в глубину. Вершины исследуются в обратном порядке обхода. Если два региона соединены только одной дугой, то они объединяются. Если вершина является входной точкой циклической или ациклической управляющей конструкции, то регион, соответствующий этой конструкции, выделяется в новую вершину, и соответствующим образом корректируются дуги. Тип конструкции определяется последовательным сравнением подграфов, включающих рассматриваемую вершину, с соответствующими шаблонами. Алгоритм заканчивает работу, когда преобразованный граф не содержит дуг и состоит только из одного региона.

Таким образом, в результате применения последовательности сворачивающих преобразований, строится управляющее дерево, определенное следующим образом:

* Корень управляющего дерева – это абстрактный граф, состоящий из одного узла и представляющий исходный граф.
* Листья управляющего дерева – это базовые блоки исходного графа.
* Узлы, не являющиеся корнем или листьями – абстрактные узлы графа потока управления.
* Узлы a1, a2, …, an являются потомками узла a0 тогда и только тогда, когда узел a0 был получен сверткой из узлов a1, … an.

Самой простой формой интервального анализа является Т1-Т2 анализ. Он состоит всего из двух преобразований, показанных на рисунках ниже.

|  |  |
| --- | --- |
| Рис. . Т1-преобразование | Рис. . Т2-преобразование |

Впоследствии был разработан алгоритм структурного анализа, который является наиболее мощным способом интервального анализа.

#### Структурный анализ

В алгоритме структурного анализа выделяются следующие шаблоны сводимых конструкций:

* block шаблон (линейная последовательность узлов графа);
* if-then шаблон;
* if-then-else шаблон;
* switch шаблон;
* conditional шаблон (для свертки сложных логических выражений);
* self loop шаблон (самоцикл – узел, зацикливающийся сам в себя);
* natural loop шаблон;
* while loop шаблон;
* шаблон, соответствующий несобственной области (improper region).

Существенно, что каждый из шаблонов имеет ровно одну входную дугу. Таким образом, например, несводимая область, помеченная данным алгоритмом, будет включать в себя ближайшего общего доминатора всех ее входов.

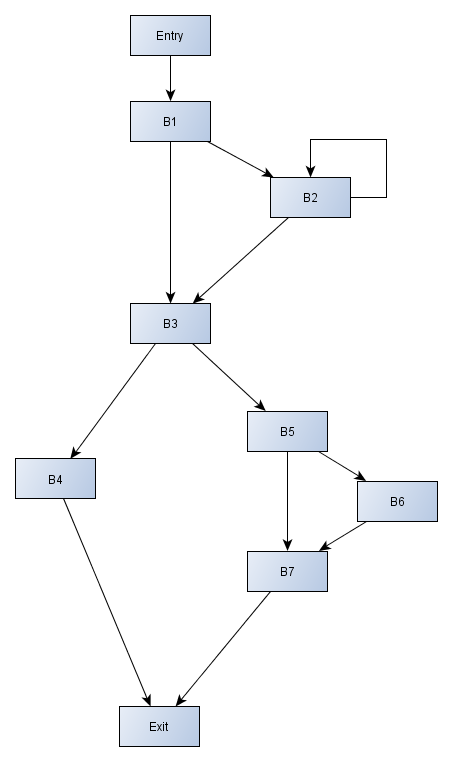
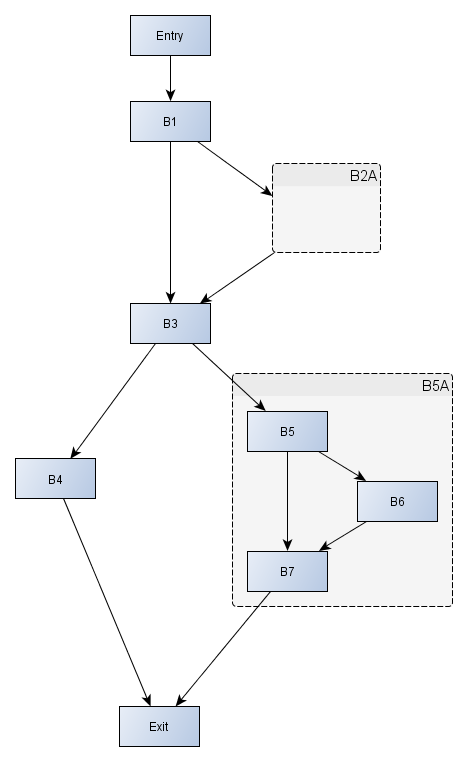
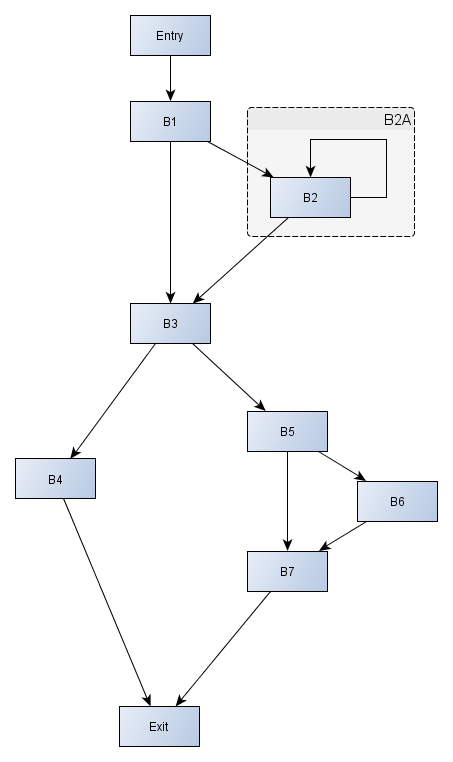
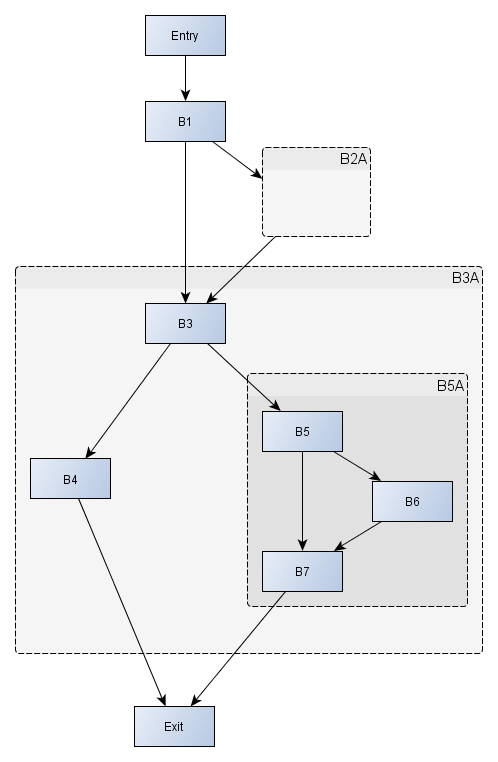
Алгоритм выделяет в графе потока управления структуры одного из этих типов, затем производит «свертку» графа, заменяя выделенную область новым абстрактным узлом, и перенаправляет входящие и исходящие дуги соответствующим образом.

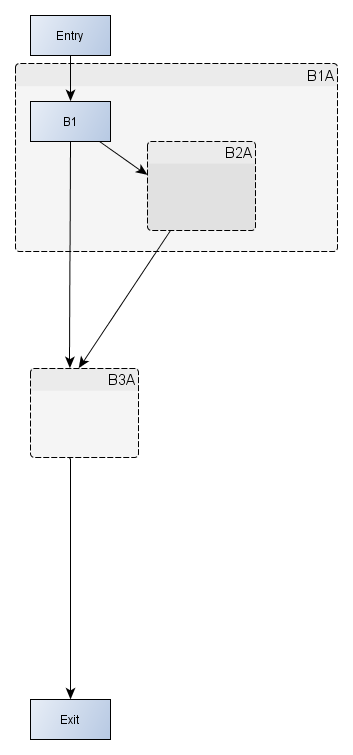
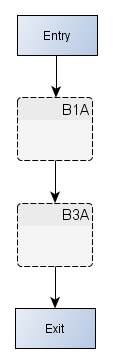
Основные шаги алгоритма структурного анализа таковы:

1. Построение основного дерева поиска в глубину для графа потока управления.
2. Обратный обход вершин графа. При этом обходе алгоритм для каждой вершины пытается наложить на граф шаблон таким образом, чтобы рассматриваемая вершина была входной вершиной шаблона.
3. Если удалось наложить шаблон, то выделенная область сворачивается в абстрактный узел соответствующего типа, при этом входящие и исходящие дуги области перенаправляются соответственным образом.
4. Переход к шагу 2.

Процесс оканчивается тогда, когда количество узлов графа становится равным одному.

Пример процесса свертки графа:

#### Анализ идиом

В анализируемой программе выделяются идиомы посредством ее трансформации и сравнения выделенных идиом с библиотекой идиом. Для каждой идиомы из библиотеки идиом имеется высокоуровневая конструкция, в которую восстанавливается последовательность команд, соответствующая идиоме.

# ПРОЕКТИРОВАНИЕ И РАЗРАБОТКА ДЕКОМПИЛЯТОРА

## Введение

Результатом анализа существующих работ, посвящённых решению проблем декомпиляции, стало проектирование архитектуры и создание прототипа декомпилятора. Было обнаружено, что в них не рассматриваются практические трудности, которые проявляются при разработке декомпилятора.

Одним из таких моментов является необходимость тщательного анализа входных точек программы, что оказывает серьезное влияние на качество отделения кода от данных в исполняемом файле. При анализе формата исполняемого файла не стоит опираться только на главную входную точку программы, т.к. обычно там содержится информация о дополнительных точках входа, которые могут помочь при декомпиляции. Например, в ходе исследования формата PE файла были обнаружены виртуальные адреса, указывающие на обработчики исключений, функции инициализирующие глобальные переменные в многопоточном приложении, список экспортируемых функций и т.п. Анализ дополнительных входных точек очень важнен, поскольку на некоторые из них управление передается раньше, чем на главную входную точку программы.

На практике отдельной проблемой является получение и использование информации времени исполнения декомпилируемой программы. В дополнение к входным точкам программы, обнаруженным в процессе разбора формата исполняемого файла можно использовать информацию от трассировщика. Трассировка — это процесс выполнения программы по шагам, инструкция за инструкцией. Трассировщик может передавать виртуальный адрес на начало блока инструкций, идущих подряд, а декомпилятор может воспользоваться этой информацией, чтобы увеличить число инструкций, отделенных от данных. В работе [41] подробно описано как можно использовать информацию времени выполнения, чтобы улучшить анализ типов данных.

После построения графа машинного кода, декомпилятору необходимо преобразовать полученный код в промежуточную форму. Опять же тут возникает трудность с выбором этой промежуточной формы и способом ее построения. В результате было решено использовать SSA. Single Static Assignment (SSA) – это промежуточное представлением, в котором каждой переменной значение присваивается лишь единожды. Данное свойство позволяет серьезно упростить процесс слияния инструкций, определения и удаления неиспользуемого кода и тупиковых ветвей, а также выделить параметры и возвращаемые значения функций. Более того, как будет описано ниже, сам SSA-граф является потоком управления программы, что позволяет использовать его при восстановлении управляющих конструкций. Таким образом, правильно сформированное SSA-представление позволяет решить множество проблем декомпиляции.

## Проектирование загрузчика

Загрузчик файла осуществляет разбор входного формата исполняемого файла. Загрузчик получает на вход файл, после чего производит анализ формата файла и отдает обнаруженные входные точки, а так же дополнительную символьную информация.

Прежде всего, определить интерфейса загрузчика, который позволил бы разбирать любой формат исполняемого файла вне зависимости от платформы. Это позволит в дальнейшем добавлять загрузчики других форматов исполняемых файлов, при этом, не меняя остальных частей программы.

Интерфейс дизассемблера определяется из расчета информации, требуемой модулю, осуществляющему дизассемблирование, описанному ниже:

* виртуальный адрес входной точки;
* декодировать команду, находящуюся по определенному виртуальному адресу.

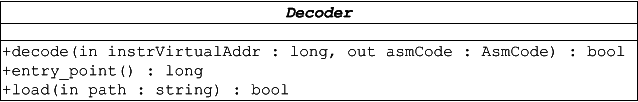


Рис. . Интерфейс Decoder

При разборе формата исполняемого файла загрузчик определяет виртуальный адрес входной точки. Модуль дизассемблирования получает адрес этой входной точки. После этого, в соответствии с внутренним алгоритмом дизассемблер, передает загрузчику виртуальный адрес инструкции, которую нужно разобрать и получает инструкцию во внутреннем низкоуровневом представлении.

Кроме того, как будет показано ниже, загрузчик должен получать различные метаданные из переданного ей исполняемого файла, что может улучшить читаемость кода. Например, это может быть название функции и сборка, из которой эта функция была получена.

### Загрузчик PE файла

Рассмотрим на конкретном основные алгоритмы по анализу PE файла. В соответствии со спецификацией формата исполняемых файлов «Microsoft Portable Executable and Common Object File Format Specification исполняемый модуль состоит из нескольких частей:

* секции для совместимости с MS-DOS;
* PE-заголовка;
* одного или нескольких заголовков секций (называемых обычно таблицей секций);
* размещения - непосредственно секций.

На рисунке ниже приведена схема расположения данных в исполняемом файле, взятая из упомянутой выше спецификации формата исполняемых файлов.



Рис. . Структура PE-файла

Данные здесь показаны как последовательность структур, при этом полностью скрыта организация данных и, соответственно, связь отдельных частей исполняемого файла друг с другом. Рассмотрим составные части исполняемого файла и взаимосвязи их друг с другом более подробно.

Большинство структур, которые используются в исполняемом файле, документированы в файле WinNT.h, входящем в стандартную поставку Microsoft Windows Software Development Kit – набор разработчика программного обеспечения. Ниже представлена таблица, содержащая информацию обратиться к необходимым структурам PE-файла:

|  |  |
| --- | --- |
| Смещение | Описание структуры |
| 00h | MZ – заголовок, описывается структурой IMAGE\_DOS\_HEADER. MZ заголовок служит для обратной совместимости с ОС MS-DOS. Он содержит указатель на заглушку, используемую при попытке запуска программы в MS-DOS. Также этот заголовок содержит поле e\_lfanew, указывающее на начало PE- заголовка. |
| Поле e\_lfanew (IMAGE\_DOS\_HEADER) | PE-заголовок, описывается структурой IMAGE\_NT\_HEADERS. Заголовок состоит из трех частей: сигнатуры “PE00”, обязательного заголовка IMAGE\_FILE\_HEADER и дополнительного заголовка IMAGE\_OPTIONAL\_HEADER. |
| e\_lfanew + sizeof(IMAGE\_NT\_HEADERS) | Таблица описаний секций файла. Представляет собой массив структур типа IMAGE\_SECTION\_HEADER. Количество секций определяется полем NumberOfSections заголовка IMAGE\_FILE\_HEADER. Таблица содержит данные, необходимые для загрузки секций файла в память. |
| Поле PointerToRawData (IMAGE\_SECTION\_HEADER) | Секции исполняемого файла. Смещения на эти данные хранятся в структурах IMAGE\_SECTION\_HEADER. |

#### Поиск входных точек

Найти максимальное число входных точек в файл является важной задачей, поскольку позволяет отделить большее число инструкций от данных в файле. В результате подробного анализа формата PE файла были выявлены следующие входные точки:

|  |  |
| --- | --- |
| Входная точка | Описание |
| Главная точка входа в исполняемый файл | Основная точка входа в программу. Именно эта точка считается стартовой и содержит в себе вызов функции main. Виртуальный адрес для этой входной точки может быть получен путем сложения полей AddressOfEntryPoint с полем ImageBase, расположенных в дополнительном заголовке IMAGE\_OPTIONAL\_HEADER. |
| TLS callback’и | TLS – это thread local storage, которая хранит глобальные переменные, которые объявляются для каждого создаваемого потока, локальны в рамках потока и не требуют синхронизации. Найти информацию обо всех TLS callback’ах можно в секции .tsl. |
| Список экспортируемых функций | Таблица экспортируемых функций (Export Address Table, EAT) содержит виртуальные адреса функций, которыми можно воспользоваться из других исполняемых файлов. Таблицу можно найти в секции .edata. |
| Обработчики исключений | Обработчики исключений хранят в себе информацию о том, какой участок кода может кинуть исключение и как его надо обрабатывать. Директории, хранящие информацию об исключениях можно найти в секции .edata. |

Если рассматривать только главную входную точку, можно упустить важные участки кода. Например, совсем необязательно, чтобы внутри исполняемого файла вызывались экспортируемые функции, и если не просмотреть их отдельно, то они не будут разобраны.

Более того, мало где упоминается информация про TLS callback’и, а ведь эти функции будут запущены еще до главной точки входа. В результате, может быть ситуация, при которой управление на главную точку не передается в принципе, например:

EntryPoint:

XOR EAX, EAX

PUSH EAX

CALL d, ds:[ExitProcess]

...

label:

...

TLS\_Callback1:

JMP label

В результате при запуске программы, на EntryPoint управление так никогда передано не будет. Поэтому при разборе любого формата исполняемого файла необходимо глубоко исследовать, как именно операционная система его загружает и работает с ним, потому что можно дизассемблировать и, следовательно, декомпилировать совсем не то, что делает программа.

#### Преобразование адресов

Функция декодирования использует декодер, который описан ниже. Модуль, осуществляющий декодирование принимает на вход указатель на массив в памяти, содержащий инструкцию, а выдает разобранную инструкцию.

Поскольку на уровне ассемблерных команд используются виртуальные адреса, а декодер принимает на вход смещение относительно начала файла, то следует описать алгоритм, по которому загрузчик осуществляет сопоставление виртуального адреса и смещения в файле. Сначала определим различные понятия адреса, которые будет применяться, чтобы описать местоположение инструкции:

* смещение (offset) ­ смещение инструкции относительно начала файла;
* виртуальный адрес (Virtual Address, VA) ­ это адресов инструкции внутри адресного пространства, выделенного системой процессу;
* относительный виртуальный адрес (Relative Virtual Address, RVA) ­ это смещение инструкции относительно базового адреса, по которому загружается исполняемый файл в памяти. Более точным будет сказать, что это смещение относительно адреса, по которому загружается в память конкретная секция.

Можно вычислить относительный виртуальный адрес по виртуальному адресу следующим образом: RVA = VA – адрес загрузки исполняемого файла (ImageBase). Соответственно, если нужно по RVA получить VA, то имеем: VA = RVA + ImageBase.

**Алгоритм вычисления смещения инструкции в файле по виртуальному адресу:**

1. По полученному виртуальному адресу вычисляется относительный виртуальный адрес инструкции: RVA = VA - ImageBase.
2. В цикле идет обход всех структур IMAGE\_SECTION\_HEADER, описанных выше.
   1. Вычисляется конец секции путем сложения значения полей структуры IMAGE\_SECTION\_HEADER VirtualAddress и SizeOfRawData.
   2. Если виртуальный адрес инструкции, вычисленный на шаге 1 входит в границы текущей секции, то происходит возврат из функции виртуального адреса инструкции относительно поля PointerToRawData структуры IMAGE\_SECTION\_HEADER.
3. Раз в цикле не была найдена секция, содержащая инструкцию возвращается -1.

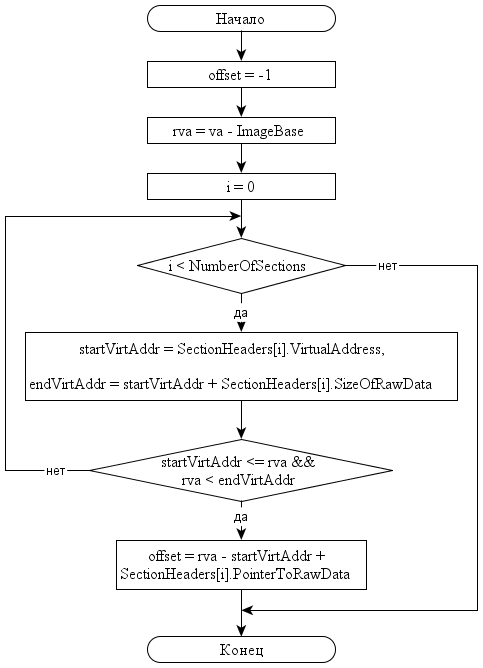


Рис. . Блок-схема алгоритма вычисления смещения по виртуальному адресу

В дальнейшем эта функция будет называться *va\_to\_offset.* Если нужно вычислить смещение по RVA, то можно воспользоваться равенством указанным выше: VA = RVA + ImageBase. Отсюда получаем функцию *rva\_to\_offset*:

int rva\_to\_offset(int rva)

{

va\_to\_offset(rva + ImageBase)

}

#### Подстановка имен импортируемых функций

При просмотре листинга кода необходимо максимально облегчить его чтение, поэтому было решено заменить адреса импортируемых функций на их названия. Чтобы это сделать, нужно просмотреть таблицу импорта (Import Address Table, IAT) и найти название библиотеки и функции, соответствующую адресу вызываемой функции. Вместо того, чтобы каждый раз просматривать таблицу импорта было решено при загрузке исполняемого файла генерировать таблицу соответствий виртуальных адресов импортированных функций и их имен. Имея такую таблицу, загрузчик при декомпиляции инструкции может заменить строковое представление инструкции call, подставив название соответствующей функции.

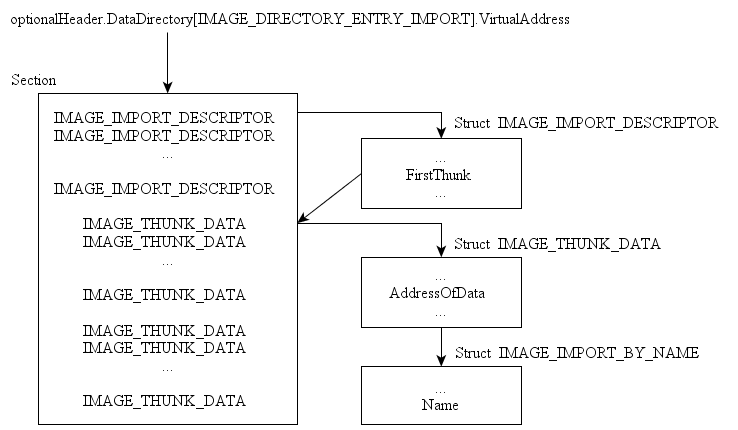


Рис. . Структуры, содержащие IA

**Алгоритм поиска имен и адресов импортируемых функций:**

1. Получить ImportDataDirectory из поля DataDirectory структуры IMAGE\_OPTIONAL\_HEADER. Нужная директория находится по индексу IMAGE\_DIRECTORY\_ENTRY\_IMPORT. В директории импорта содержится поле VirtualAddress, по которому можно определить секцию, со структурой IMAGE\_IMPORT\_DESCRIPTOR, в которой содержится информация об импортируемой библиотеке.
2. В цикле идет обход всех структур IMAGE\_SECTION\_HEADER, описанных выше.
   1. Вычисляется конец секции путем сложения значения полей структуры IMAGE\_SECTION\_HEADER VirtualAddress и SizeOfRawData.
   2. Если виртуальный адрес дескриптора импортируемой библиотеки, вычисленный на шаге 1 входит в границы текущей секции, то секция запоминается.
3. Если секция не была найдена, то происходит выход из функции.
4. Если секция найдена, то просматриваются все дескрипторы импортируемых функции, которые расположены по смещению PointerToRawData структуры IMAGE\_SECTION\_HEADER. По смещению PointerToRawData располагаются структуры IMAGE\_IMPORT\_DESCRIPTOR идущие подряд.
   1. Проверяется, если структура IMAGE\_IMPORT\_DESCRIPTOR заполнена нулями, то выход из цикла. Иначе переходим к следующему шагу.
   2. Находим имя найденной импортируемой функции по полю Name, которое содержит RVA. Используя алгоритм, описанный выше преобразуем RVA в смещение в файле, содержащее строку.
   3. В поле FirstThunk содержится VA на структуру IMAGE\_THUNK\_DATA, которая содержит информацию о конкретной импортируемой функции из библиотеки. По VA находим смещение в файле и начинаем просматривать список идущих подряд структур IMAGE\_THUNK\_DATA.
      1. Проверяется, если структура IMAGE\_THUNK\_DATA заполнена нулями, то выход из цикла. Иначе переходим к следующему шагу.
      2. В полю u1.AddressOfData структуры IMAGE\_THUNK\_DATA располагается RVA на структуру IMAGE\_IMPORT\_BY\_NAME. По RVA получается смещение в файле на структуру содержащую имя.
      3. В бинарное дерево поиска заносится имя функции, содержащиеся в поле Name структуры IMAGE\_IMPORT\_BY\_NAME и виртуальный адрес импортируемой функции, который вычисляется на основе виртуального адреса, полученного на шаге 4.3

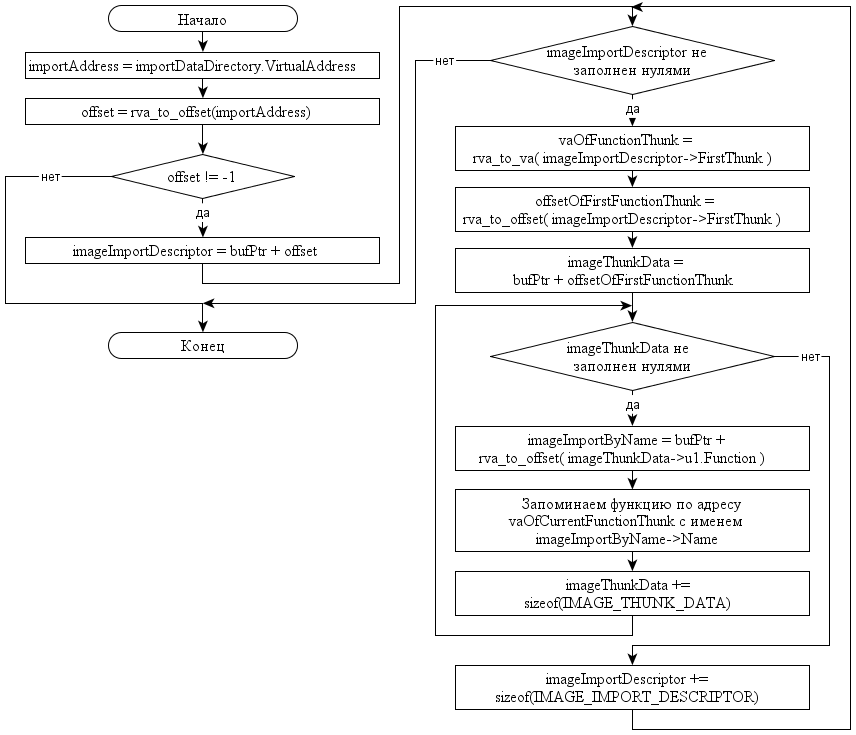


Рис. . Блок-схема алгоритма поиска имен и адресов импортируемых функций.

## Декодер

Перед тем, как приступить к решению задачи по отделению кода от данных необходимо транслировать массив байт в ассемблерную команду. Для этого нужно разобрать формат команды, точно так же как это делает процессор. Машинная команда представляет собой закодированное по определенным правилам указание процессору на выполнение некоторой операции. Команда имеет следующий формат:

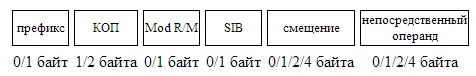


Рис. . Формат машинной команды

Рассмотрим каждое поле отдельно:

* *Префикс*. Необязательная часть инструкции, позволяет изменить некоторые особенности ее выполнения. В команде может быть использовано сразу несколько префиксов разного типа.
* *Код операции* (КОП) – это элемент, сообщающий процессору вид действия, которое ему необходимо совершить. Значение в поле кода операции некоторым образом определяет в блоке микропрограммного управления подпрограмму, реализующую действия для данной команды.
* *Байт "Mod R/M"* определяет режим адресации, а также иногда дополнительный код операции. Необходимость байта "Mod R/M" зависит от типа инструкции.
* *Байт SIB* (Scale-Index-Base) определяет способ адресации при обращении к памяти в 32-битном режиме. Необходимость байта SIB зависит от режима адресации, задаваемого полем "Mod R/M".
* *Операнд и/или смещение операнда* в сегменте данных, с которыми нужно что-то делать. Операнды в команде могут и не задаваться, а подразумеваться по умолчанию.

Таким образом, структура, представляющая ассемблерную команду должна содержать обобщенную информацию о декодированной команде, необходимую для ее идентификации и представления пользователю, а также детальные сведения об операции и операндах, которые в нее входят. Данные сведения потребуются на дальнейших этапах анализа кода программы. В первую очередь при разбиении ассемблерного кода на базовые блоки и проведении SSA.

Нужно также помнить, что количество ассемблерных команд в исполняемом файле может быть довольно большим, что налагает определенные ограничения на объем памяти, занимаемый такой структурой.

Исходя из этого, а так же учитывая данные, которые потребуются на последующих этапах анализа кода, структура, представляющая машинную команду должна содержать следующую информацию:

* строковое представление инструкции;
* строковое представление мнемоники команды;
* начальный виртуальный адрес команды;
* размер команды в байтах;
* виртуальные адреса ссылающихся инструкций;
* сведения об операции;
* сведения об операндах.

***Строковое представление инструкции*** необходимо для вывода команды на экран в виде, удобном для пользователя.

***Строковое представление мнемоники команды*** необходимо для ее вывода в виде, удобном для пользователя.

***Начальный виртуальный адрес*** команды необходим для определения местоположения команды в программе, для организации унифицированного способа идентификации команд на большинстве этапов компиляции.

***Виртуальные адреса ссылающихся инструкций***, с которых осуществляется переход на текущую команду, потребуются на этапе построения SSA-формы.

***Размер команды*** в байтах необходим на этапе дизассемблирования для определения начального виртуального адреса следующей команды, путем сложения начального виртуального адреса текущей команды и ее размера.

***Сведения об операции***, которые потребуются при анализе инструкции:

* код операции;
* является ли инструкция безусловным или условным переходом, и если является, то:
  + вид перехода (безусловный\вид безусловного\вызов функции\возрват из функции);
  + адрес перехода;
* строковое представление операции (например, add, sub, jmp).

Информация о том, является ли текущая операция каким-либо видом условного или безусловного перехода обязательно нужна для правильного разбиения кода на базовые блоки в процессе дизассемблирования и при построении SSA-формы.

*Тип инструкции перехода*, также как и *адрес* перехода, понадобится на этапе анализа SSA-формы, при выделении функций и при генерации и анализа потока управления.

***Сведения об операндах*** должны содержать:

* тип операнда (регистр\константа\адрес в памяти\константа\другое);
* размер операнда в байтах;
* если операнд является обращением к области памяти, то:
  + базовый регистр;
  + регистр со смещением (если есть);
  + константное значение смещения (если есть);
* модифицируется ли данный операнд в результате операции;
* используется ли сегментный регистр (es, ds, fs, gs, cs, ss);
* строковое представление операнда (например, eax\0х0401400\ebp[28]).

*Тип операнда,* информация о том,является ли операнд обращением к области памяти и дополнительные сведения, позволяют определить откуда команда получает данные для обработки. Необходимо для выделения *определений* составления *выражений* на этапе генерации SSA-формы и при анализе потока данных.

Флаг модификациипозволяет определить операнды, модифицирующиеся в результате операции, что также помогает при генерации SSA-формы.

Строковое представление операнда необходимо для его вывода в виде, удобном для пользователя.

В результате анализа спроектирована следующая иерархия классов, содержащая всю необходимую информацию о машинной инструкции:

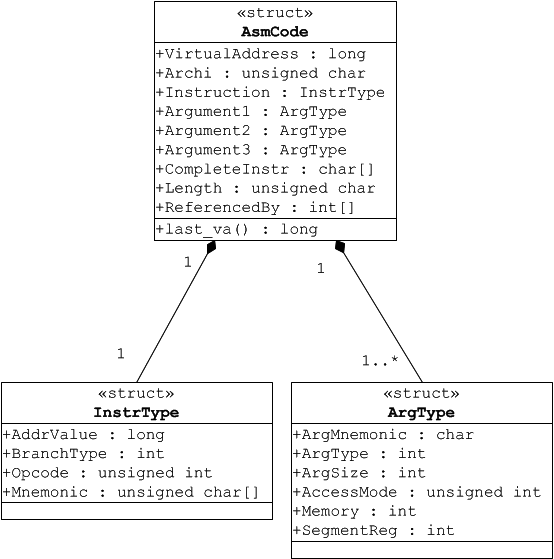


Рис. . Структуры , для хранения ассемблерной команды

Для того, чтобы заполнить эту структуру была использована библиотека BeaEngine. Она может декодировать инструкции архитектур 16/32/64, включая такие наборы инструкций, как FPU, MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, VMX, CLMUL, AES. Информация, которую можно получить в результате декодирования с помощью данной библиотеки, полностью соответствует требованиям к структурам данных, описанным выше, поэтому было решено использовать ее при декодировании.

Для получения разобранной команды используется функция Disasm, имеющая следующую сигнатуру:

int Disasm (LPDISASM pDisAsm);

Функция возвращает количество байт, занимаемых инструкцией. В качестве аргумента функция принимает указатель на структуру DISASM. Перед тем, как вызвать функцию обязательно нужно установить 2-а поля: EIP и VirtualAddr. Рассмотрим подробнее структуру DISASM:

typedef struct \_Disasm {

UIntPtr EIP;

UInt64 VirtualAddr;

UInt32 SecurityBlock;

char CompleteInstr[INSTRUCT\_LENGTH];

UInt32 Archi;

UInt64 Options;

INSTRTYPE Instruction;

ARGTYPE Argument1;

ARGTYPE Argument2;

ARGTYPE Argument3;

PREFIXINFO Prefix;

InternalDatas Reserved\_;

} DISASM, \*PDISASM, \*LPDISASM;

Поскольку в рамках данной работы не требуется проводить полный разбор библиотеки BeaEngine, рассмотрим только поля структуры DISASM, которые используются для заполнения информации об команде:

|  |  |  |
| --- | --- | --- |
| Тип | Название | Описание |
| UInt64 | VirtualAddr | Опциональное поле. Используется при разборе инструкций call, безусловного и условного переходов, чтобы можно было определить виртуальные адреса команд, на которые осуществляется переход. |
| char | CompleteInstr | Строка, в которой содержится полное строковое представление инструкции. |
| INSTRTYPE | Instruction | Структура, содержащая информацию по анализируемой инструкции. |
| ARGTYPE | Argument1 | Структура, содержащая первый аргумент инструкции, если он есть. |
| ARGTYPE | Argument2 | Структура, содержащая второй аргумент инструкции, если он есть. |
| ARGTYPE | Argument3 | Структура, содержащая третий аргумент инструкции, если он есть. |

Структура INSTRTYPE содержит сведения об операции:

typedef struct {

Int32 Category;

Int32 Opcode;

char Mnemonic[16];

Int32 BranchType;

EFLStruct Flags;

UInt64 AddrValue;

Int64 Immediat;

UInt32 ImplicitModifiedRegs;

} INSTRTYPE;

|  |  |  |
| --- | --- | --- |
| Тип | Название | Описание |
| Int32 | Category | Определяет набор инструкций (MMX, FPU, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, VMX). |
| Int32 | Opcode | Содержит код операции. |
| char | Mnemonic | Предоставляет имя команды в строковом виде. |
| Int32 | BranchType | Определяет тип перехода (call, ret, безусловный переход, условный переход) |
| EFLStruct | Flags | Содержит используемые флаги. |
| UInt64 | AddrValue | Если данная инструкция ‑ это инструкция перехода, то в этом поле содержится адрес целевой команды |
| Int64 | Immediat | Если инструкция содержит константу, то она содержится здесь. |
| UInt32 | ImplicitModifiedRegs | Если инструкция модифицирует регистр явным образом, то здесь будет содержаться об этом информация. |

Последняя структура ARGTYPE, рассмотренная в данном разделе содержит информацию об аргументе инструкции:

typedef struct {

char ArgMnemonic[64];

Int32 ArgType;

Int32 ArgSize;

Int32 ArgPosition;

UInt32 AccessMode;

MEMORYTYPE Memory;

UInt32 SegmentReg;

} ARGTYPE;

|  |  |  |
| --- | --- | --- |
| Тип | Название | Описание |
| Char | ArgMnemonic | Содержит аргумент в строковом виде. |
| Int32 | ArgType | Тип аргумента. |
| Int32 | ArgSize | Размер поля в байтах. |
| Int32 | ArgPosition | Равен HighPosition (1), если аргумент – регистр ah, ch, bh или dh, LowPosition (0) – в ином случае. |
| UInt32 | AccessMode | Содержит информацию о том, модифицируется ли данный регистр (WRITE=0x2) или нет (READ=0x1) |
| MEMORYTYPE | Memory | Если аргумент располагается в памяти, то данное поле содержит дополнительную информацию. |
| UInt32 | SegmentReg | Содержит информацию об используемом сегментном регистре. |

## Дизассемблер

### Принципы дизассемблирования

Для подробного анализа исполняемого кода необходимы специальные инструменты, такие как дизассемблеры. При этом исполняемый код переводится в вид, пригодный для анализа, выделяются составные части кода, такие как подпрограммы, строятся дерево вызовов, таблицы перекрёстных ссылок и т. д. Таким образом, может быть построена модель исполняемого кода, находящегося в анализируемом файле.

После первичной обработки кода можно производить дополнительную обработку, направленную на получение более подробной информации о коде.

Смещение кода, находящегося в исполняемом файле, может быть получено при помощи анализа метаданных исполняемого файла. Например, смещение точки входа в исполняемый файл типа .exe находится в заголовках PE файла. Смещения экспортируемых функций находятся в директории экспорта, при этом смещение директории экспорта в свою очередь может быть получено из таблицы директорий PE-заголовка. Кроме того, исполняемый код может находиться в таблице обработчиков исключений, доступ к которой можно получить через директорию LOAD\_CONFIG, а также в функциях, вызываемых при обработке TLS, адреса которых могут быть получены через директорию TLS. Таким образом, модель метаданных, модель данных и модель кода могут рассматриваться как самостоятельно, так и объединяться в единую модель, которую можно назвать древовидной моделью файла.

### Проблемы при дизассемблировании

Дизассемблер ­ – это транслятор, преобразующий машинный код, объектный файл или библиотечные модули в текст программы на языке ассемблера. Чтобы понять проблемы, которые приходится решать при дизассемблировании бинарных файлов, нужно определить, как эти файлы компилировались.

Компиляция производится компилятором, который генерирует объектные файлы из исходных кодов, а поскольку это однонаправленный процесс, приводящий к потере данных, то полностью восстановить исходный текст не представляется возможным без вмешательства человека.

Основные проблемы, с которыми пришлось столкнуться на этапе дизассемблирования:

* Отсутствие информации об именах переменных и процедур, комментариях, параметрах и возвращаемых значениях функций, и типах данных;
* Смешение кода и данных.
* Выявление границ процедур.

### Модуль дизассемблирования

Общую структуру программного модуля дизассемблирования, можно представить в следующем виде:

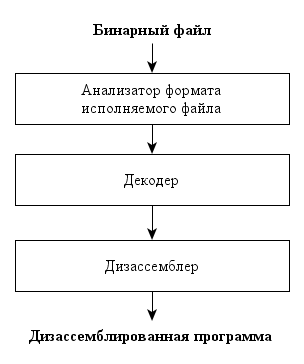


Рис. . Структура дизассемблера

На вход дизассемблеру подаются исполняемые файлы. После этого дизассемблер разбирает формат входного файла и определяет входные точки, с которых можно начать анализ исполняемого файла. Далее, используя декодер, он начинает обход найденных входных точек и анализирует декодированные инструкции для определения мест передачи потока управления. На выходе генерируется ассемблерный код.

Остановимся подробнее на реализации модуля для дизассемблирования. Он был разработан на Си++. Анализатор формата файла был описан выше, поэтому здесь он подробно рассматриваться не будет.

### Алгоритм дизассемблирования

Перейдем непосредственно к дизассемблеру. Как уже было сказано выше, существует проблема отделения кода от данных. В связи с этим нельзя последовательно разбирать машинный код, поскольку каждый следующий байт может быть данными, а не инструкцией.

Например, рассмотрим последовательность байт **CE XX E8 61 06**. (второй байт может принимать любое значение). Как показано на рисунки ниже, в зависимости от того, как на нее будет передано управление, могут быть получены разные последовательности инструкций. Поэтому дизассемблеры, которые рассматривают инструкции последовательно, не смогут распознать случай, когда второй байт является байтом данных. Из-за этой ошибки все остальные инструкции будут распознаны неверно.



Рис. . Варианты разбора инструкций

Рассмотрим существующие подходы к дизассемблированию:

1. *Линейный проход* (*Linear sweep, LS*). Эта техника дизассемблирования, при которой производится линейный проход по секции кода. Последовательно разбирается инструкция за инструкцией.
2. *Расширенный линейный проход* (*Extended linear sweep, ELS*). Эта техника расширяет предыдущий подход, распознавая таблицы переходов основываясь на перемещаемых адресах.
3. *Рекурсивный обход* (*Recursive traversal, RT*). В этом случае дизассемблер проходит по всем ветвям потока управления в коде, то есть вызовы функций, условные и безусловные переходы.
4. *RT с анализом потока данных* (*Data-flow guided RT, DRT*). Эта техника расширяет RT за счет анализа потока данных, определяя косвенные переходы, подставляя значения, взятые выше по графу потока управления. После производится граничная проверка найденного адреса и если адрес выходит за границы, то разбор не производится.
5. *Гибрид ELS/RT.* В ходе линейного прохода производится рекурсивный обход. В результате это позволяет определить ошибки в случае несовпадения результатов работы.
6. *Спекулятивное дизассемблирование.* Этот подход ортогонален LS и RT. Он запоминает, какие части были продизассемблированы и пытается заполнить пробелы за счет спекулятивного дизассемблирования кода. При этом он помечает разобранные участи как спекулятивные и может их отметить, если обнаружена неверная инструкция. Он зависит от того, насколько сложно определить, что инструкция неверна.
7. *Интерактивные дизассемблеры.* Некоторые дизассемблеры включают человека в цикл работы для определения того, какие бинарные данные должны быть разобраны. Самый главный вопрос, в каких случаях лучше справляется человек, а в каких автоматизированные инструменты.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Корректная обработка данных в сегменте кода | Требует наличия отладочной информации | Определяет косвенные переходы | Генерирует "мусорные" инструкции | Требует участие человека |
| Линейный  проход (LS) | нет | нет | не всегда | да | нет |
| Расширенный  линейный проход (ELS) | не всегда | да | не всегда | да | нет |
| Рекурсивный обход (RT) | да | нет | нет | нет | нет |
| Рекурсивный обход с анализом потока данных (DRT) | да | нет | не всегда | нет | нет |
| Спекулятивное  дизассемблирование | не всегда | нет | не всегда | да | нет |
| Интерактивные дизассемблеры | да | нет | Да | нет | да |

Из таблицы видно, что если требуется провести дизассемблирование максимально полно, без порождения мусорных инструкций, то следует использовать DRT. Однако, учитывая, что в процессе декомпилирования обязательно будет проводиться анализ потока данных, то можно остановиться на RT, и определить косвенные переходы позднее.

В результате в модуле был реализован алгоритм RT, позволяющий максимально полно и точно произвести статический анализ бинарного файла и получить дизассемблированный листинг программы. При этом, проблема с определением косвенных переходов можно частично решить за счет использования трассировщика, что и было сделано (см. далее в главе). В этом алгоритме не производится попытка дизассемблирования данных, идущих после безусловного перехода и инструкции возврата из функции. Такой подход позволяет избавиться от проблем, связанных со случайной трансляцией данных как инструкции, которые могут привести к сбою на этом этапе. Его работа происходит по следующему принципу:

1. В стек адресов инструкций заносятся адреса входных точек исполняемого файла.
2. Делается проверка, есть ли в стеке адресов инструкций хоть одна инструкция. Если нет, то программа полностью проанализирована, иначе переходим к следующему пункту.
3. Из стека получается инструкция и производится проверка, была ли уже проанализирована функция с таким адресом или нет.
4. Если инструкция уже была разобрана, то в информацию о команде добавляется виртуальный адрес инструкции, с которой осуществлялся переход и переходим к пункту 2, иначе к следующему пункту.
5. Производится анализ полученной инструкции:
   1. Если инструкция является вызовом функции или условным переходом, то в стек инструкций заносятся 2-а адреса: адрес инструкции, следующей за текущей, и адрес, на который передается управление текущей инструкцией (если этот прямой переход, а не косвенный).
   2. Если инструкция является безусловным переходом, то в стек адресов инструкций заносится только адрес инструкции, на которую передается управление.
   3. Если это функция ret или же инструкция была разобрана с ошибками, то в стек ничего не заносится.
6. В первую инструкцию, с которой начался разбор, добавляется виртуальный адрес инструкции, с которой осуществлялся переход.
7. Переход на 2 шаг.

Ниже приведен листинг функций, осуществляющих дизассемблирование в упрощенном виде. Для простоты восприятия из данного листинга был исключен код, добавляющий в команды адреса, с которых производятся переходы на разбираемую инструкцию.

bool disassemble(const Decoder& decoder)

{

queue<va\_t> jumpInstructionQueue;

jumpInstructionQueue.push(decoder.entry\_point());

while( false == jumpInstructionQueue.empty() )

{

disassemble\_next\_jump( decoder, jumpInstructionQueue );

}

fill\_code\_collection\_using\_instruction\_map();

return true;

}

Функция disassemble() получает на вход декодер и запрашивает у него адрес точки входа. Далее этот адрес заносится в очередь jumpInstructionQueue и функция переходит к выполнению цикла while, соответствующего шагам 3-6, приведенного выше алгоритма. Функция, в упрощенном виде:

bool disassemble\_next\_jump(const Decoder& decoder, queue<va\_t>& jumpInstructionQueue)

{

AsmCode currentAsmCode;

va\_t instrVirtAddr = jumpInstructionQueue.front();

jumpInstructionQueue.pop();

if(is\_instruct\_decoded(instrVirtAddr))

{

return;

}

while(decoder.decode(instrVirtAddr, &currentAsmCode))

{

\_instructionMap.insert(currentAsmCode.VirtualAddr, new AsmCode(currentAsmCode));

if(currentAsmCode.Instruction.BranchType == RetType)

{

break;

}

if(currentAsmCode.Instruction.BranchType == JmpType)

{

if( 0 != currentAsmCode.Instruction.AddrValue )

{

jumpInstructionQueue.push(currentAsmCode.Instruction.AddrValue);

}

break;

}

if(0 != currentAsmCode.Instruction.AddrValue)

{

jumpInstructionQueue.push(currentAsmCode.Instruction.AddrValue);

}

instrVirtAddr += currentAsmCode.length;

}}

Функция disassemble\_next\_jump() представляет собой реализацию шагов 3-6. Она получает на вход декодер и очередь с адресами перехода, которые надо дизассемблировать. Из очереди достается адрес и происходит проверка на повторное дизассемблирование. Если инструкции по данному адресу уже дизассемблировались, то происходит выход из функции. Если нет, то функция начинает последовательно декодировать инструкции, пока ей не встретится инструкция безусловного перехода или декодер не вернет инструкцию NOP. Все целевые адреса условных и безусловных переходов сохраняются в структуру данных jumpInstructionQueue. Все декодированные инструкции сохраняются в \_instructionMap. Результатом работы этих функций является заполненное поле \_instructionMap, которая содержит виртуальные инструкции и соответствующие им структуры AsmCode, которые были описаны выше.

## Трассировка программы

При дизассемблировании не возникает проблем с прямыми переходами по виртуальному адресу. Когда происходит вызов функции по виртуальному адресу, то сразу можно отследить, куда передается управление. Однако, если вызов функции происходит по косвенному адресу, в таких случаях как вызов виртуальной функции в C++, то сразу отследить конечный адрес не представляется возможным. Похожую ситуацию можно наблюдать при работе с callback функциями. Например, рассмотрим следующий код:

#include <stdio.h>

void callback()

{

printf("callback\n");

}

void run\_callback(void (\*handler)())

{

handler();

}

int main()

{

run\_callback(callback);

exit(0);

}

здесь вызов функции handler в функции run\_callback будет скомпилирован в следующем виде:

...

push ebp

mov ebp, esp

call d ss:[ebp+08h]

pop ebp

...

При дизассемблировании данного кода нельзя получить адрес перехода. Используя статический анализ адрес перехода можно определить, только если провести анализ потока данных, переданный на этот участок. Однако этот адрес перехода можно получить и раньше. Для этого, было предложено доработать классическую структуру декомпилятора и добавить фазу трассировки.

Трассировка это процесс пошагового выполнения программы. Чаще всего трассировка используется при отладке программы. В данном случае, после проведения трассировки анализируемой программы гарантирует, что продизассемблированы все участи бинарного файла, которые были исполнены при работе программы.

Рассмотрим подробнее, что происходит на фазе трассировки исполняемой программы. Самая часто используемая возможность - третье прерывание. Один байт отлаживаемой программы заменяется на однобайтовую инструкцию int 3, которая при срабатывании передает управление отладчику. Этот способ позволяет производить отладку в защищенном режиме и размещать отладчик там, куда не доберется отлаживаемая программа, как бы она этого не захотела (в нулевом кольце защиты). Достоинство этого метода в том, что он позволяет ставить неограниченное число точек останова, а недостаток - в непосредственной модификации кода программы. Многие защиты проверяют целостность своего кода, а значит, такой метод на них срабатывать не будет.

Чтобы не писать с нуля трассировщик, было решено воспользоваться инструментом Intel Pin Tool, который позволяет производить трассировку нужных участков программы. Pin Tool представляет из себя программу, которая в качестве входных параметров принимает путь к трассируемой программе и путь к модулю, в котором содержится код, срабатывающий при исполнении инструкций.

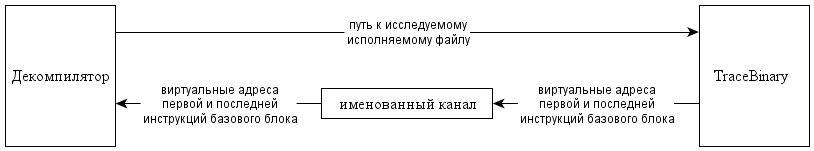


Рис. . Схема взаимодействие декомпилятора и трассировщика

Модуль TraceBinary к Intel Pin Tool, который трассирует программу и передает результаты декомпилятору. Для каждого трассируемого базового блока (basic block, BBL ­ это последовательность инструкций, содержащая один вход и один выход) модуль производит проверку, принадлежит ли базовый блок непосредственно исполняемому файлу, а не одной из библиотечных функций. Если этот блок инструкций является частью трассируемой программы, то по именованным каналам виртуальный адрес первой и последней инструкции в базовом блоке передается декомпилятору.

## Декомпилятор

### Генерация промежуточного представления

Одним из самых мощных приемов статического анализа в декомпиляции является SSA-форма. Правильно сгенерированное SSA-представление программы позволяет разрешить проблемы подстановки выражений, удаления неиспользуемого кода и тупиковых ветвей кода (частично решает проблему обфусцированного кода) и определения параметров и возвращаемых значений функций.

SSA-граф состоит из набора *базовых блоков*, содержащих список *утверждений* в формате: *определение\_индекс* *:=* *выражение. Базовым блоком* является последовательность команд, имеющих только одну точку входа и только одну точку выхода, и не содержащую инструкций передачи управления. *Определение* представляет собой регистр или область памяти, в который записывается результат *выражения* (если таковой имеется). Выражение может содержать другие, *ранее объявленные*, определения и константы. Согласно правилам построения SSA каждое определение может быть назначено только один раз, поэтому для дифференциации ранее использованных регистров и адресов используется *индекс*. В качестве индекса может служить любая, гарантированно уникальная в рамках одного и того же регистра или области памяти, константа. На практике выходит, что на эту роль лучше всего подходят начальные виртуальные адреса команд, которые уже были определены и назначены на этапе дизассемблирования. Такой подход избавляет от необходимости заботиться об определении, хранении, назначении и своевременном инкрементировании отдельного счетчика.

Рассмотрим пример преобразования следующего набора ассемблерных команд в SSA-форму:

mov eax, 12

and eax, 24

В SSA представлении будет выглядеть как:

eax\_0 := 12

eax\_1 := eax\_0 & 24

zf\_1 := 0

sf\_1 := 0

Из примера видно, что при генерации первичной SSA-формы необходимо учитывать неявное выставление флагов процессором в результате выполнения каждой команды.

Рассмотрим другой случай, когда второй операнд команды будет не константой, а переменной, хранящейся в памяти:

mov eax, 12

and eax, esp[-8]

Тогда SSA-представление примет следующий вид:

eax\_0 := 12

m[esp-8]\_0 := \*(esp-8)

eax\_1 := eax\_0 & m[esp-8]\_0

zf\_1 := eax\_1 == 0

sf\_1 := eax\_1 >= 2147483648

В данном случае осуществляется доступ к ранее неопределенной области памяти. Поскольку использование ранее неопределенных переменных в SSA-форме невозможно, то необходимо ввести дополнительное определение для операции доступа к переменной по адресу esp[-8]. Еще одна особенность примера, приведенного выше – указание в определении смещения -8. Такое дополнение дает возможность просто и наглядно создавать SSA-определения для отдельных членов массивов или переменных, хранящихся в памяти. Нужно также отметить, что, так как в данном случае выражения zf\_1 и sf\_1 представляют собой неравенства из-за того, что определить значение, хранящееся по адресу esp[-8] может быть невозможно на этапе статического анализе.

В следующем примере демонстрируется SSA-форма для простейшей управляющей конструкции if:

mov eax, 12

and eax, esp[-8]

jn skip1

xor eax, eax

skip1:

ret

Сгенерированное для этого примера SSA представление:

eax\_0 := 12

zf\_0 := 0

sf\_0 := 0

m[esp-8]\_0 := \*(esp-8)

eax\_1 := eax\_0 & m[esp-8]\_0

zf\_1 := eax\_1 == 0

sf\_1 := eax\_1 >= 2147483648

goto skip1 if sf\_1 == 1

eax\_2 := eax\_1 xor eax\_1

sf\_2 := 0

zf\_2 := 0

skip1:

eax\_3 := φ(eax\_1, eax\_2)

zf\_3 := φ(zf\_1, zf\_2)

sf\_3 := φ(sf\_1, sf\_2)

return(eax\_3, zf\_3, sf\_3)

В листинге сверху продемонстрировано использование φ-функции. На примере выражения для eax\_3 принцип ее работы можно описать фразой «либо eax\_1, либо eax\_2». Несмотря на то, что подобное описание звучит довольно размыто, φ-функция с успехом справляется со своей основной задачей: однозначное определение точки схождения двух ветвей графа и представление значений, которые может принимать определение в этой точке программы.

Далее на примере программы бесконечным циклом будет продемонстрировано «правильное» разбиение программы на базовые блоки с последующей генерацией SSA-формы и вставки φ-функций:

01 mov eax, 12

endlessloop1:

02 and eax, esp[-8]

03 jn skip1

04 xor eax, esp[-4]

05 jmp endlessloop1

skip1:

06 ret

Нулевой блок будет содержать только команду из строки 01, т.к. на строку 02 осуществляется переход инструкцией jmp endlessloop1. Обнаружение команды, на которую осуществляется переход из какой-либо другой части программы, должен приводить к завершению заполнения текущего блока и формированию нового, в начало которого и помещается эта команда. Таким образом, первый блок будет содержать строки 02 и 03. Заполнение этого блока должно быть завершено после обнаружения команды условного перехода jn, т.к. любая инструкция перехода является точкой выхода. По той же причине заполнение следующего блока будет завершено после добавления туда команды jmp endlessloop1. Последняя команда будет в блоке №3. В итоге получится следующее разбиение:

Block#0

mov eax, 12

Block#1

and eax, esp[-8]

jn Block#3

Block#2

xor eax, esp[-4]

jmp Block#1

Block#3

ret

SSA-граф тогда примет следующий вид:

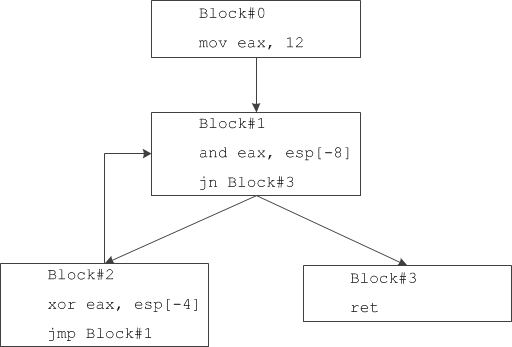


Рис. . SSA-граф

Определив базовые блоки, на входе которых соединяется более одной ветви графа, можно увидеть, что единственной точкой вставки φ-функции является начало блока №1. Таким образом, SSA-форма для программы из листинга сверху примет следующий вид:

Block#0

eax\_0 := 12

zf\_0 := 0

sf\_0 := 0

Block#1

eax\_1 := φ(eax\_0, eax\_3)

zf\_1 := φ(zf\_0, zf\_3)

sf\_1 := φ(sf\_0, sf\_3)

m[esp-8]\_0 := \*(esp-8)

eax\_2 := eax\_1 & m[esp-8]\_0

zf\_2 := eax\_2 == 0

sf\_2 := eax\_2 >= 2147483648

goto Block#3 if sf\_2 == 1

Block#2

m[esp-4]\_0 := esp[-4]

eax\_3 := eax\_2 xor m[esp-4]\_0

sf\_3 := eax\_2 == 0

zf\_3 := eax\_2 >= 2147483648

goto Block#1

Block#3

return(eax\_2, zf\_2, sf\_2)

Полученное представление возможно не очень удобно для восприятия человеком, но зато отлично поддается автоматическому разбору. Например, для обнаружения неиспользуемых переменных необходимо лишь найти определения, которые не присутствуют ни в одном выражении.

#### Алгоритм генерации SSA-представления

Основываясь на базовых операциях генерации SSA-графа, описанных в предыдущем разделе, можно привести требования к классам, формирующим программный блок генератора SSA.

Класс, описывающий базовый блок SSA должен предоставлять:

* список входящих в него SSA-утверждений;
* список неинициализированных определений;
* список текущих определений;
* список блоков, на которые ссылается текущий блок;
* список блоков, которые ссылаются на текущий блок.

Список неинициализированных определений – это регистры или области памяти, для которых в рамках текущего блока было найдено хотя бы одно использование, но не было обнаружено ни одного (явного или неявного) определения. На этапе первичного формирования SSA-графа невозможно определить множество всех входящих значений для каждого из блоков, поскольку еще не были определены все связи между ними. Поэтому, список неинициализированных определений необходим на этапе связывания базовых блоков, чтобы узнать какие из определений должны быть проинициализированы значениями из родительских блоков.

Список текущих определений – это все выходные определения текущего базового блока, включая все определения всех блоков, ссылающихся на него, которые не были переопределены в текущем. Так как одновременное хранение всех таких определений в каждом блоке не является целесообразным, правильнее будет реализовать рекурсивную функцию, которая будет искать запрошенное определение в текущем блоке по базовому имени. В случае если оно не будет найдено, та же функция должна последовательно вызываться для всех блоков, ссылающихся на данный. В случае, если данный базовый блок содержит инструкцию call, функция должна возвращать пустой результат. Данный прием позволит выделить базовые блоки, входящие в процедуры а также определить аргументы, которые они принимают на вход. Случаи нахождения более, чем одного текущего определения должны обрабатываться с помощью φ-функций.

На этапе связывания список текущих определений всех ссылающихся блоков будет сопоставляться со списком неинициализированных определений текущего блока. В результате все неинициализированные определения текущего базового блока должны быть связаны с текущими определениями родительских блоков. Исключением может послужить ситуация с вызовом функции внешней библиотеки в одном из ссылающихся блоков. В данном случае необходимо знать соглашение вызова процедур, использующееся в декомпилируемой программе, чтобы точно определить, какие из определений будут восстановлены после ее вызова. Цель данной работы – описание базового рабочего алгоритма, и поэтому рассмотрение этой проблемы в ней опускается.

Класс, описывающий SSA-утверждение должен предоставлять:

* сведения о SSA-определении;
* сведения о SSA-выражении;
* строковое представление утверждения;
* тип утверждения.

Утверждение должно инкапсулировать работу с определением и выражением. Тип утверждения должен однозначно идентифицировать, является ли утверждение классическим присвоением или каким-либо видом перехода. Во втором случае SSA-определение должно оставаться не должно быть заполнено.

Класс, описывающий SSA-определение должен предоставлять:

* базовое имя;
* виртуальный адрес;
* смещение;
* массив индексов использующих утверждения;
* строковое представление определения.

Базовое имя – это имя регистра или адрес в памяти, которое используется в определении. Виртуальный адрес используется в качестве индекса определения и соответствует виртуальному адресу команды, для которой генерируется текущее определение. Смещение должно использоваться при создании определения для элемента массива или области памяти. Массив индексов использующих утверждения предназначен для облегчения обнаружения неиспользуемых выражений и распространения инструкций на этапе анализа SSA-формы. Данный массив позволяет, как легко идентифицировать все неиспользуемые выражения, так и выявить случаи единичного использования для последующего распространения. Т.к. обе эти операции проводятся для утверждений, находящихся в одном и том же базовом блоке, то будет достаточным хранить только индексы утверждений.

Класс, описывающий SSA-выражение должен предоставлять:

* тип операции;
* сведения об операндах;
* строковое представление выражения.

Поле типа операции необходимо для явного описания вида выражения (например, логическое «И», целочисленное сложение, деление чисел с плавающей запятой и т.д.). Сведения об операндах будут представлять собой ссылки на определения или константы. Эти ссылки потребуется изменять, например, в процессе связывания базовых блоков или на этапе распространения выражений.

Основываясь на описанных выше требованиях были спроектированы классы, представленные на рисунке ниже:

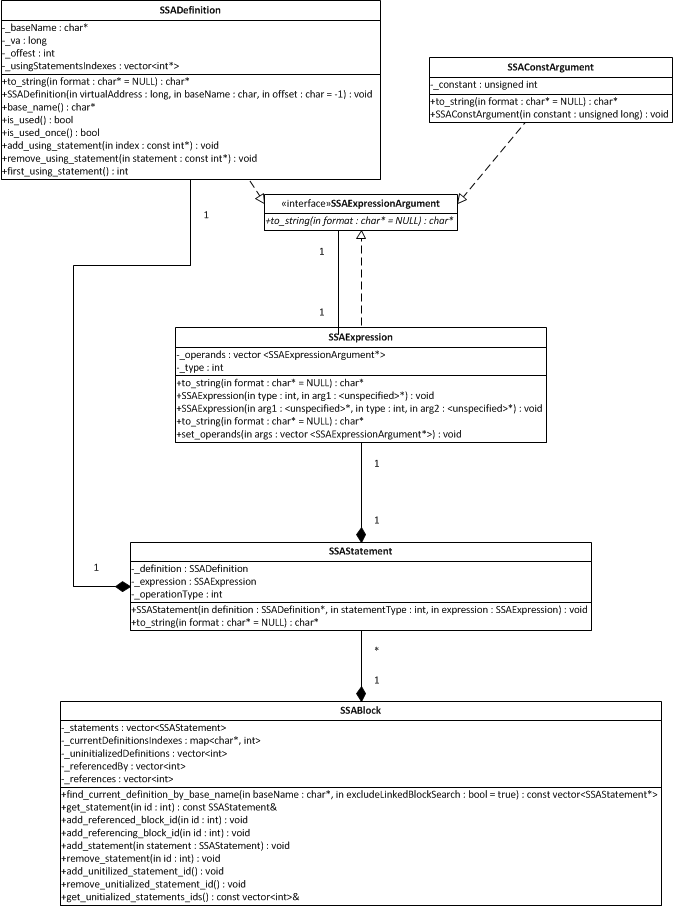


Рис. . UML-диаграмма базовых классов SSA-генератора

Первая фаза генерации SSA формы заключается в том, чтобы получить внутреннее низкоуровневое представление команд и преобразовать его в выражения в терминологии SSA. В конце работы алгоритма будет получен граф связанных базовых блоков. Связи между определениями и выражениями разных блоков пока отсутствуют и будут устанавливаться в рамках следующего прохода. Для простоты восприятия, такие подробности, как заполнение массива индексов использующих инструкций, в этом алгоритме опущены.

1. На вход поступают: ссылка на массив всех инструкций; виртуальный адрес текущей инструкции, первой в формируемом базовом блоке; идентификатор блока, который ссылается на переданную инструкцию.
2. Проверка, если виртуальный адрес переданной инструкции уже входит в один из базовых блоков, то добавить идентификатор блока в массив ссылаемых блоков и вернуть управление.
3. Создается пустой текущий базовый блок и добавляется в граф SSA.
4. Добавить идентификатор блока, который ссылается на данный блок, в массив ссылаемых блоков.
5. Последовательный обход всех команд, начиная с команды, переданной в функцию.
   1. Если на данную команду ссылается какая-нибудь другая инструкция и текущий базовый блок непустой, то:
      1. вызвать текущую функцию. В нее передать адрес текущей инструкции и идентификатор текущего базового блока.
      2. После возврата управления добавить в массив ссылаемых инструкций текущего базового блока идентификатор блока, возвращенный функцией.
      3. Вернуть управление из текущей функции, передать идентификатор текущего блока.
   2. Получить ассемблерную команду из списка и начать ее преобразование в SSA-утверждение.
   3. Если в выражении используются определения:
      1. Если в списке текущих определений блока есть записи об определениях с соответствующими базовыми именами, то добавить в выражение ссылку на них.
      2. Если какое-либо из определений не удалось найти на предыдущем шаге, то совершить следующую последовательность действий:
         1. создать для него пустое SSA-утверждение, указав 0 в качестве индекса, и добавить его в базовый блок;
         2. в массив идентификаторов неинициализированных определений данного блока добавить запись об этом утверждении;
         3. в массив текущих определений данного блока *попытаться* добавить запись об этом утверждении;
         4. перейти к пункту 5.3.1.
   4. Добавить полученное утверждение в текущий блок;
   5. Обновить или добавить запись в массив текущих утверждений;
   6. Если текущая инструкция является инструкцией перехода, то выбрать адрес, на который осуществляется переход, и пункту 5.1.1;
   7. Завершить выполнение, передать идентификатор текущего блока.

В результате выполнения данного алгоритма будет получен SSA-граф, с выстроенными между базовыми блоками связями. Для каждого блока также осуществлена подстановка всех доступных в рамках этого блока определений. Идентификаторы всех ненайденных определений вынесены в отдельный массив.

Следующим шагом в генерации SSA-представления является связывание определений, расположенных в разных блоках. В результате будет получена полноценная SSA-форма, пригодная для проведения дальнейшего анализа. Алгоритм связывания определений представляет собой следующую последовательность действий: Для простоты восприятия, такие подробности, как управление заполнением массива индексов использующих инструкций, в этом алгоритме опущены.

1. Создать множество определений, находящихся в процессе разбора;
2. Последовательно для каждого имеющегося блока, проверить имеет ли он неинициализированные определения, если да, то:
3. Вызвать функцию связывания определений и передать в нее идентификатор блока, для которого нужно провести связывание неинициализированных определений и ссылку на множество из шага 1;
   1. Если идентификатор текущего блока находится во множестве блоков, которые находятся в процессе разбора, то перейти к пункту 3.7;
   2. Если массив идентификаторов неинициализированных определений пуст, то перейти к пункту 3.6;
   3. Добавить идентификатор текущего блока в множество разбираемых блоков;
   4. Пройти по массиву идентификаторов ссылающихся блоков. Если он не пуст, то для каждого блока из этого списка:
      1. Проверить, имеет ли он неинициализированные определения. Если да, то вызвать текущую функцию и передать в нее идентификатор этого блока и ссылку на множество из пункта 1;
      2. Вызвать функцию поиска текущего определения по базовому имени для каждого неинициализированного определения текущего блока;
      3. Результат выполнения добавить во временный массив;
   5. Для каждого неинициализированного определения текущего блока:
      1. Найти определение в массиве из шага 3.4.3;
      2. Если найдено более одного соответствия, то:
         1. Сформировать SSA-утверждение, содержащее φ-функцию от этих определений, и вставить ее в начало блока.
         2. Заменить все использования текущего определения, на использование определения, сформированного на предыдущем шаге;
         3. Перейти к пункту 3.5.5;
      3. Если найдено одно соответствие, то:
         1. Заменить все использования текущего определения этим соответствием;
         2. Перейти к пункту 3.5.5;
      4. Если не найдено ни одного соответствия, перейти к пункту 3.5.6;
      5. Удалить старое утверждение, сформированное для данного определения на предыдущем этапе генерации графа.
      6. Удалить индекс определения из массива неинициализированных;
      7. Если данное определение является последним в базовом блоке для его базового имени, то обновить массив текущих определений;
   6. Удалить идентификатор блока из множества блоков, находящихся в процессе разбора.
   7. Вернуть управление.
4. Завершить цикл.

Алгоритм последовательно проходит по массиву базовых блоков, находящихся в графе. При обнаружении базового блока с непустым массивом неинициализированных значений для него вызывается рекурсивная функция, которая пытается проинициализировать все неинициализированные определения, проводя их поиск среди блоков, которые ссылаются на него, и окончательно выставить все текущие значения. Если функция обнаруживает ссылающийся блок, который содержит неинициализированные значения, то она вызывает себя, передавая идентификатор данного предка. Множество значений, находящихся в процессе разбора необходимо во избежание зацикливания алгоритма в случаях получения блоком некоторых определений из себя самого.

В результате применения данного алгоритма имеется SSA-граф с выстроенными межблочными связями и упорядоченными связями между определениями. Полученный результат можно использовать для дальнейшего анализа путем проведения необходимых SSA-трансформаций.

### Анализ графа потока управления

После получения графа становится возможным проведение SSA-трансформаций, которые позволят упростить сгенерированное нами представление. Стоит отметить, что непосредственное удаление утверждения из блока может привести к расхождению информации, хранящейся в массивах с индексами, поэтому под удалением утверждения пока будем понимать переопределение утверждения в null.

#### Удаление неиспользуемого кода

Выполнение этой задачи довольно просто благодаря тому, что каждое определение содержит массив индексов использующих утверждений, заполнявшийся на предыдущих шагах генерации SSA-формы. Все, что необходимо сделать – это:

* удалить все переменные, массив индексов использующих утверждений которых пуст;
* удалить индекс этого определения из соответствующих массивов всех определений, которые использовались в этом утверждении;
* повторять эту операцию необходимо до тех пор, пока не останется ни одного определения, счетчик которого будет равен 0.

Стоит отметить, что при выставлении данного счетчика должен учитываться факт вызовов функций, как внешних, так и внутренних, поэтому все текущие определения блоков, предшествующих таким вызовам имеют счетчик равный -1.

Данная операция позволит избавиться от неиспользуемых определений флагов и временных переменных, что существенно облегчит имеющееся SSA-представление.

#### Распространение инструкций

Следующим шагом является слияние инструкций. Как говорилось выше, данную операцию целесообразно проводить для переменных, имеющих единичный случай использования. Такие переменные чаще всего являются временными и в ассемблерном коде предназначаются для хранения промежуточных результатов получения из памяти и хранения результатов выполнения инструкций для последующей записи в память. Основываясь на данной информации для корректного распространения выражения необходимо выполнить следующую последовательность действий:

1. Пройти по всем утверждения в графе и для каждого проверить:
   1. Если массив индексов использующих утверждений которого хранит 1 значение и не является φ-функцией и утверждение находящееся по указанному в массиве индексу тоже не является φ-функцией, то:
      1. перейти к утверждению, в котором происходит использование данного определения, и заменить его использование выражением, хранящимся в исходном утверждении;
      2. удалить исходное утверждение.

Таким образом, решается одна из основных проблем декомпиляции, описанных выше – проблема восстановления высокоуровневых выражений.

# АПРОБАЦИЯ

Демонстрация работы созданных и описанных алгоритмов декомпиляции будет проводиться на функции подсчета числа Фибоначчи. Функция является рекурсивной и содержит одну управляющую конструкцию:

unsigned int fibonacci(int x)

{

if (x > 2)

return (fibonacci(x - 1) + fibonacci(x - 2));

else

return (1);

}

После компиляции и дизассемблирования данного кода выглядит следующим образом:

0x00401020 push ebp

0x00401021 mov ebp, esp

0x00401023 push esi

0x00401024 cmp d ss:[ebp+08h], 02h

0x00401028 jle 00401050h

0x0040102a mov eax, d ss:[ebp+08h]

0x0040102d sub eax, 01h

0x00401030 push eax

0x00401031 call 0040100Ah

0x00401036 add esp, 04h

0x00401039 mov esi, eax

0x0040103b mov ecx, d ss:[ebp+08h]

0x0040103e sub ecx, 02h

0x00401041 push ecx

0x00401042 call 0040100Ah

0x00401047 add esp, 04h

0x0040104a add eax, esi

0x0040104c jmp 00401055h

0x00401050 mov eax, 00000001h

0x00401055 pop esi

0x00401056 cmp ebp, esp

0x00401058 call 00401390h

0x0040105d pop ebp

0x0040105e ret

После первого прохода SSA-генератора код будет разбит на базовые блоки и приведен к стандартному представлению в SSA. Результат выполнения каждой инструкции будет сохраняться в отдельную временную переменную, отличающуюся от других суффиксом – текущим виртуальным адресом. Неявные назначения и использования флагов будут сделаны явными. Результат выглядит следующим образом:

Basic Block #1

jump\_2:

esp\_0x00401020 := esp\_0 - 04h

m[esp]\_0x00401020 := ebp\_0

ebp\_0x00401021 := esp\_0

esp\_0x00401023 := esp\_0x00401020 - 04h

m[esp]\_0x00401023 := esi\_0

AF\_0x00401024 :=

CF\_0x00401024 :=

OF\_0x00401024 := m[ebp+08h]\_0 > 0xFFFFFFFFh - 02h

PF\_0x00401024 :=

SF\_0x00401024 := m[ebp+08h]\_0 - 02h < 0

ZF\_0x00401024 := m[ebp+08h]\_0 - 02h == 0

goto jump\_1 if ZF\_0x00401024 == 1 or SF\_0x00401024 != OF\_0x00401024

Basic Block #2

eax\_0x0040102a := m[ebp+08h]\_0

eax\_0x0040102d := eax\_0x0040102a - 01h

AF\_0x0040102d :=

CF\_0x0040102d :=

OF\_0x0040102d := eax\_0x0040102a > 0xFFFFFFFFh - 01h

PF\_0x0040102d :=

SF\_0x0040102d := eax\_0x0040102d < 0

ZF\_0x0040102d := eax\_0x0040102d == 0

esp\_0x00401030 := esp\_0 - 04h

m[esp]\_0x00401030 := eax\_0x0040102a

Basic Block #3

call jump\_2

Basic Block #4

esp\_0x00401036 := esp\_0 + 04h

AF\_0x00401036 :=

CF\_0x00401036 :=

OF\_0x00401036 := esp\_0 > 0xFFFFFFFFh - 04h

PF\_0x00401036 :=

SF\_0x00401036 := esp\_0x00401036 < 0

ZF\_0x00401036 := esp\_0x00401036 == 0

esi\_0x00401039 := eax\_0

ecx\_0x0040103b := m[ebp+08h]\_0

ecx\_0x0040103e := ecx\_0x0040103b - 02h

AF\_0x0040103e :=

CF\_0x0040103e :=

OF\_0x0040103e := esp\_0 > 0xFFFFFFFFh - 04h

PF\_0x0040103e :=

SF\_0x0040103e := eax\_0x0040103e < 0

ZF\_0x0040103e := eax\_0x0040103e == 0

esp\_0x00401041 := esp\_0x00401036 - 04h

m[esp]\_0x00401041 := ecx\_0x0040103e

Basic Block #5

call jump\_2

Basic Block #6

esp\_0x00401047 := esp\_0 + 04h

AF\_0x00401047 :=

CF\_0x00401047 :=

OF\_0x00401047 := esp\_0 > 0xFFFFFFFFh - 04h

PF\_0x00401047 :=

SF\_0x00401047 := esp\_0x00401047 < 0

ZF\_0x00401047 := esp\_0x00401047 == 0

eax\_0x0040104a := eax\_0 + esi\_0

AF\_0x0040104a :=

CF\_0x0040104a :=

OF\_0x0040104a := eax\_0 > 0xFFFFFFFFh - esi\_0

PF\_0x0040104a :=

SF\_0x0040104a := eax\_0x0040104a < 0

ZF\_0x0040104a := eax\_0x0040104a == 0

goto jump\_3

Basic Block #7

jump\_1:

eax\_0x00401050 := 00000001h

Basic Block #8

jump\_3:

esi\_0x00401055 := m[esp]\_0

esp\_0x00401055 := esp\_0 + 04h

ebp\_0x0040105d := m[esp]\_0x00401055

esp\_0x0040105d := esp\_0x00401055 + 04h

goto m[esp]

Как можно видеть, дизассемблированный код был разбит на базовые блоки, а ассемблерные инструкции приведены к виду, удобному для последующего анализа декомпилятором. Следующий проход генератора SSA-формы свяжет определения в базовых блоках и разместит пси-функции:

Basic Block #1

jump\_2:

esp\_0x00401020 := esp\_0 - 04h

m[esp]\_0x00401020 := ebp\_0

ebp\_0x00401021 := esp\_0

esp\_0x00401023 := esp\_0x00401020 - 04h

m[esp]\_0x00401023 := esi\_0

m[ebp+08h]\_0x00401024 := m[ebp+08h]\_0

AF\_0x00401024 :=

CF\_0x00401024 :=

OF\_0x00401024 := m[ebp+08h]\_0x00401024 > 0xFFFFFFFFh - 02h

PF\_0x00401024 :=

SF\_0x00401024 := m[ebp+08h]\_0x00401024 - 02h < 0

ZF\_0x00401024 := m[ebp+08h]\_0x00401024 - 02h == 0

goto jump\_1 if ZF\_0x00401024 == 1 or SF\_0x00401024 != OF\_0x00401024

Basic Block #2

eax\_0x0040102a := m[ebp+08h]\_0x00401024

eax\_0x0040102d := eax\_0x0040102a - 01h

AF\_0x0040102d :=

CF\_0x0040102d :=

OF\_0x0040102d := eax\_0x0040102a > 0xFFFFFFFFh - 01h

PF\_0x0040102d :=

SF\_0x0040102d := eax\_0x0040102d < 0

ZF\_0x0040102d := eax\_0x0040102d == 0

esp\_0x00401030 := esp\_0x00401023 - 04h

m[esp]\_0x00401030 := eax\_0x0040102a

Basic Block #3

call jump\_2

Basic Block #4

esp\_0x00401036 := esp\_0 + 04h

AF\_0x00401036 :=

CF\_0x00401036 :=

OF\_0x00401036 := esp\_0 > 0xFFFFFFFFh - 04h

PF\_0x00401036 :=

SF\_0x00401036 := esp\_0x00401036 < 0

ZF\_0x00401036 := esp\_0x00401036 == 0

esi\_0x00401039 := eax\_0

ecx\_0x0040103b := m[ebp+08h]\_0

ecx\_0x0040103e := ecx\_0x0040103b - 02h

AF\_0x0040103e :=

CF\_0x0040103e :=

OF\_0x0040103e := esp\_0 > 0xFFFFFFFFh - 04h

PF\_0x0040103e :=

SF\_0x0040103e := eax\_0x0040103e < 0

ZF\_0x0040103e := eax\_0x0040103e == 0

esp\_0x00401041 := esp\_0x00401036 - 04h

m[esp]\_0x00401041 := ecx\_0x0040103e

Basic Block #5

call jump\_2

Basic Block #6

esp\_0x00401047 := esp\_0 + 04h

AF\_0x00401047 :=

CF\_0x00401047 :=

OF\_0x00401047 := esp\_0 > 0xFFFFFFFFh - 04h

PF\_0x00401047 :=

SF\_0x00401047 := esp\_0x00401047 < 0

ZF\_0x00401047 := esp\_0x00401047 == 0

eax\_0x0040104a := eax\_0 + esi\_0

AF\_0x0040104a :=

CF\_0x0040104a :=

OF\_0x0040104a := eax\_0 > 0xFFFFFFFFh - esi\_0

PF\_0x0040104a :=

SF\_0x0040104a := eax\_0x0040104a < 0

ZF\_0x0040104a := eax\_0x0040104a == 0

goto jump\_3

Basic Block #7

jump\_1:

eax\_0x00401050 := 00000001h

Basic Block #8

jump\_3:

esp\_0x00401055\_1 := psi(m[esp]\_0x00401023, m[esp]\_0x00401047)

esi\_0x00401055 := m[esp]\_0x00401055\_1

esp\_0x00401055 := esp\_0x00401055\_1 + 04h

ebp\_0x0040105d := m[esp]\_0x00401055

esp\_0x0040105d := esp\_0x00401055 + 04h

goto m[esp]

В результате некоторые определения из разных блоков были связаны, а для блоков с несколькими точками вхождения были добавлены пси-функции. Удаление неиспользуемых инструкций сильно упростит имеющееся представление, позволив избавиться от неиспользуемых флагов:

Basic Block #1

jump\_2:

esp\_0x00401020 := esp\_0 - 04h

m[esp]\_0x00401020 := ebp\_0

ebp\_0x00401021 := esp\_0

esp\_0x00401023 := esp\_0x00401020 - 04h

m[esp]\_0x00401023 := esi\_0

m[ebp+08h]\_0x00401024 := m[ebp+08h]\_0

OF\_0x00401024 := m[ebp+08h]\_0x00401024 > 0xFFFFFFFFh - 02h

SF\_0x00401024 := m[ebp+08h]\_0x00401024 - 02h < 0

ZF\_0x00401024 := m[ebp+08h]\_0x00401024 - 02h == 0

goto jump\_1 if ZF\_0x00401024 == 1 or SF\_0x00401024 != OF\_0x00401024

Basic Block #2

eax\_0x0040102a := m[ebp+08h]\_0x00401024

eax\_0x0040102d := eax\_0x0040102a - 01h

esp\_0x00401030 := esp\_0x00401023 - 04h

m[esp]\_0x00401030 := eax\_0x0040102a

Basic Block #3

call jump\_2

Basic Block #4

esp\_0x00401036 := esp\_0 + 04h

esi\_0x00401039 := eax\_0

ecx\_0x0040103b := m[ebp+08h]\_0

ecx\_0x0040103e := ecx\_0x0040103b - 02h

esp\_0x00401041 := esp\_0x00401036 - 04h

m[esp]\_0x00401041 := ecx\_0x0040103e

Basic Block #5

call jump\_2

Basic Block #6

esp\_0x00401047 := esp\_0 + 04h

eax\_0x0040104a := eax\_0 + esi\_0

goto jump\_3

Basic Block #7

jump\_1:

eax\_0x00401050 := 00000001h

Basic Block #8

jump\_3:

esp\_0x00401055\_1 := psi(m[esp]\_0x00401023, m[esp]\_0x00401047)

esi\_0x00401055 := m[esp]\_0x00401055\_1

esp\_0x00401055 := esp\_0x00401055\_1 + 04h

ebp\_0x0040105d := m[esp]\_0x00401055

esp\_0x0040105d := esp\_0x00401055 + 04h

goto m[esp]

В результате с представлением стало проще работать. Следующим шагом, на пути восстановления код является распространение выражений. Для всех определений, использующихся один раз, декомпилятор попробует подставить их выражения в местах использований.

jump\_2:

m[esp-04h]\_0x00401020 := ebp\_0

m[esp-08h]\_0x00401023 := esi\_0

goto jump\_1 if m[esp+04h]\_0x00401020 - 02h == 0 or ( m[esp+04h]\_0x00401020 - 02h < 0 ) != ( m[esp+04h]\_0x00401020 > 0xFFFFFFFFh - 02h )

Basic Block #2

m[esp-12h]\_0x00401030 := m[esp+04h]\_0x00401020 - 01h

Basic Block #3

call jump\_2

Basic Block #4

esi\_0x00401039 := eax\_0

m[esp]\_0x00401041 := m[ebp+08h]\_0 - 02h

Basic Block #5

call jump\_2

Basic Block #6

esp\_0x00401047 := esp\_0 + 04h

eax\_0x0040104a := eax\_0 + esi\_0

goto jump\_3

Basic Block #7

jump\_1:

eax\_0x00401050 := 00000001h

Basic Block #8

jump\_3:

esi\_0x00401055 := psi(m[esp-08h]\_0x00401023, m[esp+04h]\_0x00401047)

esp\_0x00401055 := psi(esp-08h, esp+04h) + 04h

ebp\_0x0040105d := m[esp\_0x00401055]

esp\_0x0040105d := esp\_0x00401055 + 04h

goto m[esp]

Следующим шагом является восстановление из SSA-формы. Данный этап представляет собой удаление суффиксов определений и пси-функций. Полученное после этого этапа представление можно будет подвергнуть восстановлению потока управления.

jump\_2:

m[esp-04h] := ebp\_0

m[esp-08h] := esi\_0

esp\_1 := esp-08h;

goto jump\_1 if m[esp+04h] - 02h == 0 or ( m[esp+04h] - 02h < 0 ) != ( m[esp+04h] > 0xFFFFFFFFh - 02h )

m[esp-12h] := m[esp+04h] - 01h

call jump\_2

esi := eax\_0

m[esp] := m[ebp+08h]\_0 - 02h

call jump\_2

esp\_1 := esp\_0 + 04h

eax := eax\_0 + esi\_0

goto jump\_3

jump\_1:

eax := 00000001h

jump\_3:

esi := m[esp\_1]

esp := esp\_1 + 04h

ebp := m[esp]

goto m[esp]

На этапе восстановления потока управления декомпилятор должен заменить использования оператора goto высокоуровневыми управляющими конструкции согласно определенным выше шаблонам и упростить условия перехода:

jump\_2:

m[esp-04h] := ebp\_0

m[esp-08h] := esi\_0

if(m[esp+04h] >= 02h)

{

m[esp-12h] := m[esp+04h] - 01h

call jump\_2

esi := eax

m[esp\_0] := m[ebp+08h] - 02h

call jump\_2

esp\_1 := esp + 04h

eax := eax + esi

}

else

{

eax := 00000001h

}

esi := m[esp\_1]

esp := esp\_1 + 04h

ebp := m[esp]

goto m[esp]

В полученном представлении явно виден двойной рекурсивный вызов функции. В первом вызове в нее передается значение m[esp+04h] - 01h, после чего результат сохраняется во временной переменной esi. Далее функция вызывается второй раз и в нее передается значение m[ebp+08h] - 02h, которое позже складывается со значением, сохраненным в esi. При этом если значение, полученное функцией – меньше 2-х то возвращается единица.

Таким образом, декомпилятор может разобрать имеющуюся функцию на приемлемом уровне благодаря использованным и разработанным выше приемам реверс инжиниринга. Были восстановлены рекурсивный вызов, управляющие конструкции и арифметические выражения. Следующим шагом на пути улучшения покрытия кода и представления его в более удобном для чтения виде, будет реализация этапа определения параметров и возвращаемых значений функций в рамках анализа SSA-формы.

# ВЫВОДЫ

В магистерской диссертации ставилась задача исследования и разработки методов декомпиляции программ, позволяющих восстанавливать программы из низкоуровневого представления, в частности из исполняемого файла, корректно и наиболее качественно в язык программирования высокого уровня. Разработать и апробировать прототип инструментальной среды декомпиляции программ на языке ассемблера в программы на языке.

В результате работы получены следующие результаты, характеризующиеся научной новизной:

1. Проанализированы существующие методы дизассемблирования и PE формат файла и на основе них разработан метод анализа исполняемого файла, позволяющий производить разбор файла, используя все возможные входные точки, а не только главную.
2. Предложен новый этап декомпиляции, позволяющий решить проблему распознавания косвенных переходов без порождения лишних инструкций ­ фаза трансляции.
3. Предложен алгоритм для преобразования низкоуровневого представления в промежуточную форму. Хотя существуют работы ( [27], [24], [4]), в которых предлагается использовать в качестве промежуточной формы SSA, в них нет подробного алгоритма его образования.
4. Разработан макет декомпилятора, в котором была реализована часть разработанных методов декомпиляции. На основе исполняемого файла уже генерируется промежуточное представление.

Полученный макет можно использовать для дальнейшего анализа и улучшения полученной промежуточной формы. Проект располагается на github’е по адресу <https://github.com/razyoba/kraken>. Таким образом, любой желающий может использовать существующий проект и производить необходимый ему анализ.

Как выяснилось в ходе анализа предметной области, сейчас нет готового инструмента, который можно было взять в готовом виде и написать свой алгоритм, улучшающий качество разобранного кода. В дальнейшем планируется доработать этот проект, чтобы любой желающий мог его использовать.

Уже сейчас все используемые библиотеки являются кросс-платформенными, хотя некоторые части системы написаны с привязкой к Windows. Во-первых, будет добавлена возможность собирать проекты и под Linux. Во-вторых, будет доработано API для работы с промежуточном представлением, чтобы его было удобно использовать. В-третьих, необходимо расширить набор ассемблерных инструкций, которые можно перевести в промежуточную форму.

# СПИСОК ЛИТЕРАТУРЫ

x

|  |  |
| --- | --- |
| 1 | Balakrishnan G., Reps T. Analyzing Memory Accesses in x86 Executables. 2004. |
| 2 | BeaEngine Home [Электронный ресурс] // BeaEngine: [сайт]. URL: http:/​/​www.beaengine.org/ |
| 3 | Boomerang documentation [Электронный ресурс] // Boomerang: [сайт]. URL: http:/​/​boomerang.sourceforge.net/​doxy/​index.html |
| 4 | Chang B.Y.E., Harren M., and Necula G.C. The 13th International Static Analysis Symposium (SAS) // Analysis of Low-Level Code Using Cooperating Decompilers. 2006. pp. 318-335. |
| 5 | Cifuentes C., Gough K.J. XIX Conferencia Latinoamericana de Informatica // A methodology for decompilation. Buenos Aires. 1993. pp. 257-266. |
| 6 | Cifuentes C. Reverse compilation techniques. 1994. 342 pp. |
| 7 | Decompiler Design - Introduction [Электронный ресурс] // Decompiler Design: [сайт]. URL: http:/​/​www.backerstreet.com/​decompiler/​introduction.htm |
| 8 | Durfina L., Kroustek J., Zemek P., Kolar D., Hruska T., Masarik K., and Meduna A. Advanced Static Analysis for Decompilation Using Scattered Context Grammars. Angers, France. 2011. pp. 164-170. |
| 9 | Hariri M. A Decompiler Project. 2005. |
| 10 | IDA F.L.I.R.T. Technology: In-Depth [Электронный ресурс] // hex-rays: [сайт]. URL: https:/​/​www.hex-rays.com/​products/​ida/​tech/​flirt/​in\_depth.shtml |
| 11 | IDA Multilevel IR // Department of Computer and Information Science. URL: www.ida.liu.se/​~chrke/​courses/​ACC/​PDF-2012/​F1-Multilevel-IR.pdf (дата обращения: 12.04.2012). |
| 12 | Intel Instruction Set pages [Электронный ресурс] // Intel Instruction: [сайт]. URL: http:/​/​web.itu.edu.tr/​kesgin/​mul06/​intel/​index.html |
| 13 | Intel. 64 ia 32 architectures software developer instruction set reference manual. 2012. |
| 14 | Irvine K. Assembly Language for x86 Processors. 6th ed. Prentice-Hall, 2010. 768 pp. |
| 15 | Lim J., Reps T. A System for Generating Static Analyzers for Machine Instructions. 2008. |
| 16 | Microsoft. Microsoft Portable Executable and Common Object File Format Specification. 2013. |
| 17 | Muchnick S.S. Advanced Compiler Design and Implementation. Morgan Kaufmann, 1997. 856 pp. |
| 18 | Pin User Guide [Электронный ресурс] // Intel Developer Zone: [сайт]. URL: http:/​/​software.intel.com/​sites/​landingpage/​pintool/​docs/​58423/​Pin/​html/ |
| 19 | Reps T., Balakrishnan G., and Lim J. Representation Recovery from Low-Level Code. 2006. |
| 20 | Reps T., Balakrishnan G., Lim J., and T eitelbaum T. A Next-Generation Platform for Analyzing Executables. 2005. |
| 21 | Rosen B.K., Wegman M.N., and Zadeck F.K. POPL '88 Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages // Global value numbers and raduntal computations. 88. pp. 12-27. |
| 22 | Rugaber S., Clayton R. Proceedings of the 1st Working Conference on Reverse Engineering // The representation problem in reverse engineering. 1993. |
| 23 | The dcc Decompiler [Электронный ресурс] // DCC: [сайт]. [2012]. URL: http:/​/​itee.uq.edu.au/​~cristina/​dcc.html |
| 24 | The Decompilation Wiki [Электронный ресурс] // Program-Transformation: [сайт]. URL: http:/​/​www.program-transformation.org/​Transform/​DeCompilation |
| 25 | Torri S.A. Generic Reverse Engineering Architecture with Compiler and Compression Classification Components. 2009. |
| 26 | Van Emmerik M. Signatures for library functions in executable files. 1993. |
| 27 | Van Emmerik M. Static Single Assignment for Decompilation. 2007. 334 pp. |
| 28 | Vinciguerra L., Wills L., Kejriwal N., Martino P., and Vinciguerra R. An Experimentation Framework for Evaluating Disassembly and Decompilation Tools for C++ and Java. 2003. P. 14. |
| 29 | Wartell R., Zhou Y., Hamlen K.W., Kantarcioglu M., and Thuraisingham B. Differentiating code from data in x86 binaries // ECML PKDD'11 Proceedings of the 2011 European conference on Machine learning and knowledge discovery in databases - Volume Part III. 2011. pp. 522-536. |
| 30 | x86 Disassembly/Disassemblers and Decompilers [Электронный ресурс] // wikibooks: [сайт]. URL: http:/​/​en.wikibooks.org/​wiki/​X86\_Disassembly (дата обращения: 27.12.2011). |
| 31 | Аветисян А.И., Иванников В.П., Гайсарян С.С. Анализ и трансформация программ. 2008. 78 с. |
| 32 | Антонов В.Ю., Долгова Е.Н., "Восстановление типов данных с использованием информации времени выполнения программы," *Сборник статей молодых ученых факультета ВМиК МГУ*, № 6, 2009. С. 6-16. |
| 33 | Антонов В.Ю., Фокин А.П., Долгова К.Н. Восстановление объектных структур данных при декомпиляции // СБОРНИК СТАТЕЙ МОЛОДЫХ УЧЕНЫХ факультета ВМК МГУ. 2009. № 6. |
| 34 | Ахо А., Лам М., Сети Р., Ульман Д.Д. Компиляторы: принципы, технологии и инструменты. 2-е изд. Вильямс, 2008. 1184 с. |
| 35 | Деревенец Е.О., Трошина Е.Н., "Структурный анализ в задаче декомпиляции," *Прикладная информатика*, 2009. С. 87-99. |
| 36 | Долгова К.Н., Чернов А.В., Деревенец Е.О., "Методы и алгоритмы восстановления программ на языке ассемблера в программы на языке высокого уровня," *Проблемы информационной безопасности. Компьютерные системы*, 2008. С. 54-68. |
| 37 | Лакийчук О.А., "Алгоритмы поиска доминаторов в управляющем графе," *Конструирование и оптимизация программ*, 2003. |
| 38 | Об упаковщиках в последний раз: Часть первая - теоретическая [Электронный ресурс] // Underground Information Center: [сайт]. URL: http:/​/​uinc.ru/​articles/​41/ |
| 39 | "Обзор алгоритмов декомпиляции," *Электронный журнал "Исследовано в России"*, Т. 4, 2001. С. 1143-1158. |
| 40 | Трошина Е.Н., Чернов А.В. Восстановление типов данных в задаче декомпилирования в язык C // Прикладная информатика. Декабрь 2009. № 6. С. 99-117. |
| 41 | Трошина Е.Н. Исследование и разработка методов декомпиляции программ. 2009. 134 с. |
| 42 | Юров В.И. Assembler. Питер, 2003. 640 с. |

x