Е. Н. Трошина, А. В. Чернов

Инструментальная среда восстановления исходного кода программы — декомпилятор TyDec

Декомпилятор — это инструмент, позволяющий восстанавливать программы из низкоуровневого представления в высокоуровневое. В данной работе к декомпиляции помимо корректного восстановления программ выдвигается дополнительное требование — «качественное восстановление». Инструментальное средство восстановления программ — декомпилятор TyDec, разработанный авторами данной статьи — восстанавливает программы в низкоуровневом представлении и дизассемблированные трассы программ в программы на языке Си корректно и качественно.

Введение

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. При этом внешние модули могут поставляться без исходного кода. Наличие таких модулей в системе снижает 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 лучше иметь их представление на языке более 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 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 из низкоуровневого представления в высокоуровневое. Под декомпилятором будем понимать инструментальное средство, получающее на вход программу на языке ассемблера или другое аналогичное 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 среди широко используемых компилируемых языков программирования высокого уровня наиболее распространены языки Си и *C++*, 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 *C++* можно считать расширением языка Си, добавляющим в него концепции более высокого уровня. Поскольку при обратной инженерии в целом и при декомпиляции в частности уровень абстракции представления программы повышается, можно считать, что программы на языке Си являются промежуточным уровнем при переходе от программы на языке ассемблера к программе на языке *C++*. Дальнейшее повышение уровня абстракции представления программы возможно 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 алгоритмически неразрешимой является задача автоматического разделения кода и данных.

Помимо функциональной эквивалентности исходной программе, декомпилированная программа должна быть восстановлена наиболее полно, что определяется определяется степенью использования высокоуровневых конструкций целевого языка, т. е. программы, полученные на выходе декомпилятора, должны как можно больше и полнее использовать высокоуровневые конструкции языка.

Для оценки полноты восстановления в работе вводится новое требование к результату декомпиляции программ — *качество*, а также вводится *мера качества*, которая позволяет количественно сравнивать программы, восстановленные различными инструментальными средствами декомпиляции.

Данная работа посвящена разработке алгоритмов и методов автоматической декомпиляции программ на языке ассемблера в программы на языке Си.

Низкоуровневые конструкции языка Си, такие как явные приведения типов, неестественные циклы, организованные с использования операторов *goto* или операторов *continue* или *break*, замена оператора множественного выбора *switch* операторами *if* и другие, а также ассемблерные вставки кода, который не удалось восстановить, в результирующей программе крайне нежелательны. Автоматически восстановленная программа декомпилирована качественно, если она содержит минимальное количество низкоуровневых конструкций и наиболее полно использует высокоуровневые. Например, вместо оператора цикла *while*, если возможно, используется оператор цикла *for*, вместо операций с адресной арифметикой используются операции доступа к элементам массива.

Отображение конструкций языка высокого уровня в конструкции языка низкого уровня — это отображение «многие ко многим». Если зафиксировать компилятор и значения опций, управляющих генерацией кода, то это отображение становится «многие к одному», что несколько упрощает задачу восстановления программ, даже несмотря на то, что распознавание компилятора, которым была скомпилирована высокоуровневая программа — отдельная нетривиальная задача.

Декомпилятор, который восстанавливает низкоуровневую программу, изначально написанную на неизвестном заранее язы-

ке программирования, в качественную программу на фиксированном языке высокого уровня, назовем универсальным.

Разные языки высокого уровня одного типа (например, процедурные, компилируемые) предоставляют примерно одинаковый набор возможностей, однако, некоторые возможности одного языка могут не иметь прямых аналогов в другом языке. Например, указатели языка Си не имеют прямого аналога в языке Фортран. В таких случаях трансляторы программ из одного языка высокого уровня в другой на выходе вынуждены генерировать код, моделирующий особенности исходного языка средствами целевого языка. Сгенерированная на выходе программа содержит *артефакты трансляции*, т. е. фрагменты кода, появившиеся вследствие различия понятийного аппарата исходного и целевого языков. Поэтому, хотя существующие в настоящее время трансляторы между языками высокого уровня и дают на выходе синтаксически и семантически корректный код, он зачастую труден для анализа и модификации человеком. Представляется, что задача построения универсального декомпилятора не проще задачи построения универсального транслятора в фиксированный язык высокого уровня, поэтому в настоящее время нет оснований рассчитывать на возможность успешного построения универсального декомпилятора.

В силу вышесказанного является актуальной разработка методов восстановления программ в предположении, что исходная программа была реализована на конкретном языке программирования. В данной работе в качестве такого языка используется язык Си. *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.

Задача декомпиляции рассматривается как совокупность трех задач:

1. Восстановление функционального интерфейса;
2. Восстановление управляющих конструкций;
3. Восстановление переменных и типов данных.

Настоящая работа продолжает цикл работ, посвященных данной проблеме. В работе [1] представлены методы решения подзадачи декомпиляции — восстановление управляющих конструкций. В работе [2] рассмотрены методы восстановления переменных и типов данных. Она посвящена рассмотрению особенностей реализации всего инструментального средства декомпиляции программ — декомпилятору *TyDec*. Особенностью этого декомпилятора является, помимо требования *корректности* восстановления, также требование *качества* восстановления программы в низкоуровневом представлении.

Обзор существующих мировых разработок в области декомпиляции низкоуровневых программ в язык Си

На настоящий момент неизвестны декомпиляторы, восстанавливающие программы в язык Си, которые позволяют полностью автоматически восстановить любую программу в «читабельном» виде, т. е. достаточно похожем на код, написанный программистом. Для практического использования декомпиляторов требуется хорошо представлять их возможности, и для достижения наилучшего результата, возможно, потребуется использовать набор декомпиляторов в некотором сочетании.

В работе рассматриваются следующие декомпиляторы в язык Си: декомпилятор *Boomerang* [3], декомпилятор *DCC* [4], декомпилятор *REC* [5] и плагин *Hex-Rays* [6] к интерактивному дизассемблеру *IDA Pro* [7]. Рассматриваемые инструменты сравниваются на разработанной системе тестов. Критерием качества разработки системы тестов была полнота покрытия всех составляющих

языка — от управляющих конструкций до типов данных. Кроме того, декомпиляторы сравнивались на чувствительность к различным опциям компиляции исходной программы и возможность восстановления функций стандартной библиотеки, а также обнаружение функции *main*.

Все рассматриваемые декомпиляторы, кроме плагина *Hex-Rays*, на вход принимают исполняемый файл, а на выходе выдают программу на языке Си. Плагин *Hex-Rays* принимает на вход программу, являющуюся результатом работы дизассемблера *IDA Pro*, т. е. схему программы на ассемблероподобном языке программирования. В качестве результата плагин *Hex-Rays* выдает восстановленную программу в виде схемы на Си-подобном языке программирования. Тем не менее, для простоты в дальнейшем будет объединен процесс дизассемблирования с использованием *IDA Pro* с последующей декомпиляцией.

В том случае, когда декомпилятор оказывается не в состоянии восстановить некоторый фрагмент исходной программы в язык Си, этот фрагмент в восстановленной программе 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. Более того, ассемблерные вставки серьезно затрудняют решение задачи повышения уровня абстракции представления программы.

Современные декомпиляторы

Boomerang. Декомпилятор *Boomerang* [3] является программным обеспечением с открытым исходным кодом. Его разработка началась в 2002 году. Изначально задачей проекта было разработать такой декомпилятор, который восстанавливает исходный код из исполняемых файлов вне зависимости от того, каким компилятором и с какими опциями исполняемый файл был получен. Для этого в качестве внутреннего представления используется представление программы со статическими одиночными присваиваниями (*SSA*) [8]. Однако, несмотря на это, декомпилятор не сильно адаптирован под различные компиляторы и чувствителен к применению различных опций, в частности, опций оптимизации. Кроме того, в нем не поддерживается распознавание функций стандартной библиотеки языка Си.

DCC. Проект по разработке этого декомпилятора [4] 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*. В рамках этого проекта было разработано несколько базовых алгоритмов восстановления структурных конструкций программы в язык высокого уровня, которые описаны в публикациях автора. Исходный код декомпилятора доступен на официальном сайте [4]. Предложенные алгоритмы декомпиляции основаны на теории графов (анализ потока данных и потока управления). Для распознавания библиотечных функций используется сигнатурный поиск, для которого была разработана библиотека сигнатур.

REC. Этот проект [5] 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 продукт распространяется свободно, но без исходного кода. Публикации по данному проекту неизвестны. *REC* восстанавливает исполняемые файлы в различных форматах, в частности *ELF* [9] и *PE* [10]. Также декомпилятор *REC* можно использовать на различных платформах. В ходе его тестирования было выявлено, что наиболее успешно декомпилятор восстанавливает исполняемые файлы, полученные при компиляции с отключенной оптимизацией и добавлении отладочной информации.

Hex-Rays. Как сказано выше, инструмент *Hex-Rays* [6] не является самостоятельным программным продуктом, а распро-

Таблица 1

Сравнительный анализ декомпиляторов

	Boomerang	*DCC*	*REC*	*Hex-Rays*
распознавание библиотечных функций	нет	заявлено	нет	да
активность разработки	заявлено	нет	да	да
переносимость	нет	да	да	да
open source	да	да	нет	нет

страняется как плагин к дизассемблеру *IDA Pro* [7]. Это самое новое из рассматриваемых средств декомпиляции — плагин появился на рынке в 2007 году. Особенностью данного инструмента является то, что он восстанавливает программы, полученные на выходе дизассемблера *IDA Pro*. Из реализованных в нем алгоритмов особого внимания заслуживает алгоритм сигнатурного поиска *FLIRT [FL25]* и алгоритм поиска параметров в стеке *PIT* (*Parameter Identification and Tracking*).

В таблице 1 представлена сводная характеристика всех рассматриваемых декомпиляторов.

Исследование возможностей декомпиляторов

В этом разделе приведены результаты тестирования возможностей рассматриваемых декомпиляторов. Для тестирования был разработан тестовый набор программ на языке Си, покрывающий основные языковые конструкции языка.

Тестирование проводилось по следующей методике. Исходный код программы на языке Си компилировался в исполняемый файл компилятором *gcc* 3.4.5 в системе *Debian Linux* и компилятором *Borland C++ 3.1* в системе *Windows XP*. В первом случае результатом работы компилятора являлся файл формата *ELF [ELF7]* для архитектуры *ia32*, во втором случае — исполняемый файл *DOS* для 16-битного режима процессора. Исполняемый файл формата *ELF* подавался на вход декомпиляторам *Boomerang*, *REC* и *Hex-Rays*, работающим в системе *Windows XP*. Исполняемый файл формата *DOS EXE* подавался на вход декомпилятору *DCC*. Результат декомпиляции сравнивался с исходным текстом.

Таблица 2

Шкала оценки декомпиляторов

Количество баллов за тест	Комментарий
0	Декомпилятор закончил работу с ошибкой выполнения или пустым результатом
1	Декомпилятор выдал ассемблерный код. Программа на Си не была получена
2	Декомпилятор выдал программу на языке Си, которая либо не компилируется, либо работает неверно (неэквивалентна исходной), либо содержит ассемблерные вставки, то есть недекомпилированные фрагменты программы
3	Декомпилятор выдал корректную программу, которая эквивалентна исходной, но использует конструкции, отличные от конструкций в исходной программе
4	Декомпилятор выдал программу, которая эквивалентна исходной и использует те же конструкции, которые использовались в исходной программе

Такая комбинация инструментальных и целевых сред была выбрана по следующим причинам. Во-первых, декомпилятор *DCC* поддерживает только 16-битные исполняемые модули *DOS*, поэтому для оценки качества работы декомпилятора был использован компилятор 16-битного режима. Декомпиляторы *Boomerang* и *REC*, наоборот, не поддерживают 16-битный режим *DOS*. Исполняемый модуль подавался на вход декомпиляторам в формате *ELF*, а не в естественном для Windows формате *PE* [10], т. к. декомпиляторы *Boomerang* и *REC*, как оказалось, некорректно обнаруживали точку начала программы на языке Си в файлах формата *PE*.

Качество работы каждого декомпилятора для каждого теста оценивалось по 4-х бальной экспертной шкале, приведенной в таблице 2. Так, оценка «3» выставлялась в случаях, когда декомпилированная программа использовала адресную арифметику вместо массивов или приведения типов для получения указательных значений вместо корректного объявления типов переменных. Кроме того, оценка «3» выставлялась, если в результате декомпиляции цикл *for* оказывался преобразованным в цикл *while*.

Система тестов

Тестовый набор содержал следующие основные группы.

1. **Типы**. В тестах этой группы проверялась корректность восстановления типов переменных и параметров функций. Язык Си поддерживает богатый набор базовых целочисленных типов — от типа *char* до типа *unsigned long long*. Декомпилятор должен по возможности точно восстановить как размер, так и знаковость переменной.

Также рассматривались типы указателей на базовые типы. Проверялся факт обнаружения того, что переменная является указательным, а не целым типом, и корректность восстановления целевого типа указателя.

Для массивов проверялся факт обнаружения того, что переменная является локальным или глобальным массивом, точность восстановления типа элементов массива, точность восстановления размера массива, как для одномерных, так и для многомерных массивов.

Для структурных типов проверялся факт распознавания использования структурного типа и точность восстановления полей структур.

Кроме того, были рассмотрены разные комбинации типов и оценена корректность их восстановления. В частности, рассматривались массивы структур, указатели на структуры, структуры, содержащие массивы, структуры, содержащие указатели на самих себя.

2. **Языковые конструкции**. В тестах этой группы проверялась корректность восстановления управляющих структур программы. Проверялась корректность восстановления оператора *if* с простым условием, в том числе и с отсутствующей частью *else*, операторов цикла *while* и *do while* с простыми условиями.

В другой группе тестов проверялась корректность восстановления логических операций && (логическое «и»), || (логическое «или») в условиях операторов *if* и циклов. Согласно семантике языка Си, эти операторы транслируются в условные и безусловные переходы, т. е. являются конструкциями, задающими поток управления, а не вычисления значений. Декомпиляторы должны по возможности восстанавливать сложные условия в операторах языка.

Отдельно проверялась корректность восстановления структурных операторов передачи управления, таких как *break*, *continue* и *return*.

Оператор *switch* рассматривался отдельно, т. к. в большинстве компиляторов он транслируется в косвенный безусловный переход, где адрес перехода выбирается из таблицы в соответствии с вычисленным в заголовке оператора значением. Декомпиляторы должны распознавать использование этого оператора в программе.

3. **Функции**. В тестах этой группы проверялась корректность выделения параметров функций и локальных переменных в условиях разных соглашений о вызовах. Кроме того, проверялась корректность обработки рекурсивных функций.

4. **Оптимизации**. В тестах этой группы проверялась корректность работы декомпиляторов при условии, что при компиляции были использованы некоторые оптимизационные преобразования, такие как открытая вставка функций (*inlining*) и оптимизации вызовов функций (*tail call optimization*, *tail recursion optimization*, *sibling call optimization*).

5. **Взаимодействие с окружением**. В данной группе находился тест, проверяющий корректность обнаружения функции *main* в исполняемых файлах формата *PE*. Как известно, выполнение программы на языке Си начинается с функции *main*, которой передается определенный список параметров. Однако в исполняемых файлах вызову функции *main* предшествует выполнение специального кода, задача которого — настроить окружение программы на Си, что заключается, в частности, в создании стандартных потоков ввода-вывода, инициализации служебных структур данных управления динамической памятью и т. п. Этот код частично написан на языке ассемблера, кроме того, он не представляет интереса, т. к. является стандартным для всех программ. Поэтому декомпиляторы должны игнорировать этот инициализационный код и начинать декомпиляцию непосредственно с функции *main*.

Кроме того, в тестах этой группы проверялось распознавание стандартных библиотечных функций языка Си (например, *strlen* и т. д.).

Результаты тестирования

В таблице 3 приведены результаты работы декомпиляторов на выбранном наборе тестов в соответствии с системой оценок, указанной в таблице 2. Каждый столбец таблицы соответствует декомпилятору, а каждая строка — тесту. Общий результат для каждого декомпилятора получен суммированием оценок по всем тестам.

Из всех рассмотренных декомпиляторов только *Boomerang* поддерживает декомпиляцию оператора *switch*. Остальные инструменты генерируют в этом случае некорректный код на языке ассемблера. Только декомпилятор *REC* сумел восстановить цикл *for*, в то время как остальные в этом случае генерируют программу, использующую цикл *while*.

Наиболее развитым в настоящее время является декомпилятор *Hex-Rays*, который, в отличие от других, поддерживает распознавание массивов и распознавание библиотечных функций, хотя даже и *Hex-Rays* имеет много слабых сторон.

Все рассмотренные Си-декомпиляторы по входной программе выдают Си-программу с восстановленными структурными конструкциями и распознанными функциями, но они не восстанавливают полностью типы данных, поэтому получаемая на выходе Си-программа содержит множество операций явного приведения типов. Такая программа сложна для анализа человеком.

Качество декомпиляции

В данной работе декомпиляция рассматривается как задача обратной инженерии. С практической точки зрения она чрезвычайно сложна в силу того, что при компиляции утрачивается много информации о высокоуровневой программе. Часть информации, например, имена неэкспортируемых объектов, утрачивается безвозвратно. Также утрачивается информация о высокоуровневых конструкциях, используемых типах данных и др., однако эту информацию частично или полностью можно восстановить. На практике задача восстановления программы из низкоуровневого представления в программу на языке высокого уровня зачастую решается посредством привлечения специалиста. Такой подход к восста-

Таблица 3

Результаты тестирования декомпиляторов

		Bommerang	*Rec*	*DCC*	*Hex-Rays*
типы данных	*struct*	3	2	3	3
	массивы	4	3	3	4
	unsigned int	4	3	3	3
	unsigned short	3	3	3	3
структурные конструкции языка	логические операции	4	4	4	4
	циклы *for*	3	4	3	3
	циклы *while*	4	3	4	4
	циклы *do while*	4	4	4	4
	оператор *switch*	4	2	2	2
функции	рекурсия	4	4	4	4
оптимизация	*inlining*	2	2	—	2
	tail recursion	2	2	—	2
обнаружение функции *main* в *PE* файлах		1	1	—	4
обнаружение функций стандартной библиотеки		2	2	3	4
сумма баллов		48	43	40	50

новлению программ очень трудозатратный и малоэффективный по времени.

В данной работе восстановление выполняется в язык программирования Си. Программы на языке ассемблера, корректно работающие со стеком и не содержащие данных, интерпретируемых как код, и наоборот, являются *правильно построенными* программами. Так как язык Си сочетает в себе высокоуровневые и низкоуровневые возможности программирования, то можно утверждать, что существует отображение, которое любую *правильно построенную* ассемблерную программу непосредственно переводит в программу на языке Си. Однако такое отображение является трансляцией, но не декомпиляцией. Восстановленная таким образом программа будет содержать много низкоуровневых конструкций языка Си, таких как операторы перехода *goto*, явного приведения типов (*type*), прерывания цикла *break*, прерывания витка цикла *continue*, ассемблерные вставки и др. Такое восстановление является корректным, однако уровень представления программы не повышается.

В таблице 4 представлены штрафы, назначаемые за низкоуровневые конструкции в декомпилированной программе, а также за неполноту восстановления высокоуровневых конструкций языка Си.

Мера качества декомпиляции рассчитывается следующим образом. Пусть исходная программа на языке высокого уровня содержала K штрафных баллов. Пусть восстановленная программа содержит K' штрафных баллов. Качество восстановления оценивается на некотором тестовом наборе программ, который обозначим *TS*. Пусть *prog* — это исходная программа.

Таблица 4

Штрафы за артефакты трансляции и неполноту восстановления

конструкции программы	назначаемые штрафы
операция явного приведения типов (*type*)	2
оператор перехода *goto*	3
оператор выхода из середины цикла *break*	1
оператор прерывания витка цикла *continue*	1
тип данных объединение *union*	3
ассемблерная инструкция	4
невосстановление оператора *switch*	2
невосстановление оператора *for*	1
невосстановление производного типа данных	4
использование адресной арифметики вместо оператора доступа к элементу массива	2

Пусть *KLOC* (*prog*) — это количество тысяч значимых строк кода программы *prog*. Мера качества декомпиляции C_{decomp} вычисляется согласно следующей формуле:

$$C_{decomp} = \sum_{prog \in TS} \frac{\max(0, K' - K)}{KLOC(prog)}. \qquad (1)$$

Чем ближе значение меры к нулю, тем выше качество декомпиляции.

В листинге 1 представлен пример восстановления программы инструментальной надстройкой *Hex-Rays* для дизассемблера *IDA Pro*. Слева представлена исходная программа, справа — восстановленная. Как можно заметить, восстановленная программа содержит множество артефактов трансляции, затрудняющих ее понимание.

В листинге 2 также представлены примеры восстановления программ расширением *Hex-Rays* к интерактивному дизассемблеру *IDA Pro*. Меры качества декомпиляции для *Hex-Rays*, рассчитанные по формуле (1) с учетом назначаемых штрафов, представ-

Исходная программа	Восстановленная программа

int f(int* a, int n){
  int *p;
  int s=0;
  for(p=a; p<a+n; p++) {
  s += *p;
  }
  return s;
}
	int __cdecl sub_401290(int a1, int a2)
	{
	  unsigned int v2; // edx@1
	  int v3; // ecx@1
	  v3 = 0;
	  v2 = a1;
	  while (a1 + 4 * a2 > v2)
	  {
	  v3 += *(_DWORD *)v2;
	  v2 += 4;
	  }

Листинг 1. Пример исходной и восстановленной Си-программ

Пример 1	Результат декомпиляции	Пример 2	Результат декомпиляции

include <stdio.h>
 	
void f
(float a, float b)
{
if (a < b) {
printf("a < b\n");
}
if (a > b) {
printf("a > b\n");
}
return;
}
	__int16 __cdecl
	f(float a1, float a2)
	{
	__int16 result; // ax@3
	__asm
	{
	fld [ebp+arg_0]
	fld [ebp+arg_4]
	fucompp
	fnstsw ax
	sahf
	}
	if (!(_CF | _ZF))
	puts("a < b");
	__asm
	{
	fld [ebp+arg_0]
	fld [ebp+arg_4]
	fxch st(1)
	fucompp
	fnstsw ax
	sahf
	}
	if (!(_CF | _ZF))
	result = puts("a > b");
	return result;
	}
		void f2
		(signed short x,
		unsigned short y)
		{
		unsigned short
		i = 0;
		for (i = 0;
		i < x;
		i++)
		{
		y += 3;
		g(y);
		}
		return;
		}
			int __cdecl
			sub_401314
			(int a1, int a2)
			{
			int result; // eax@2
			int v3;
			// [sp+14h] [bp-4h]@1
			signed __int16 v4;
			// [sp+12h] [bp-6h]@1
			HIWORD(v3) = a1;
			LOWORD(v3) = a2;
			v4 = 0;
			while (1){
			result = SHIWORD(v3);
			if((unsigned __int16)v4>=
			(signed int)SHIWORD(v3))
			break;
			LOWORD(v3)=(_WORD)v3 + 3;
			sub_401290((unsigned __
			int16)v3);
			++v4;
			}
			return result;
			}

Листинг 2. Примеры восстановления программ с невысоким уровнем качества

ленных в таблице 4, для примера 1 и для примера 2, соответственно равны:

$$C_{decomp} = \frac{max(0,11*4-0)}{6*10^{-3}} = \frac{44}{6*10^{-3}} = 7333;$$

и

$$C_{decomp} = \frac{max(0,9*2-0+1)}{7*10^{-3}} = \frac{19}{7*10^{-3}} = 2714.$$

Декомпилятор TyDec

Инструментальное средство декомпиляции программ — декомпилятор *TyDec* — предоставляет возможность автоматического и полуавтоматического восстановления программ, реализованных на языке ассемблера, а также в других формах низкоуровневого представления, в программы на языке Си. Декомпилятор поддерживает полный цикл восстановления программ, включающий восстановление функционального ин-

Таблица 5

Описание программ, для которых выполнялось сравнение качества восстановления

Название	*CLOC*	*ALOC*	Описание
35_wc	241	465	Утилита *wc*, файл «*wc.c*»
36_cat	262	618	Утилита *cat*, файл «*cat.c*»
37_execute	726	1837	Утилита *bc*, файл «*execute.c*»
38_day	503	1383	Утилита *calendar*, файл «*day.c*»
39_deflate	403	669	Утилита *gzip*, файл «*deflate.c*»
59_lalr	674	1664	Утилита *yacc*, файл «*lalr.c*»
84_derive	71	255	Утилита *derive*, файл «*derive.c*»

терфейса, управляющих конструкций и восстановление типов данных. Инструментальная среда декомпиляции *TyDec* позволяет пользователю в процессе работы выполнять переименование объектов декомпилируемой программы.

Декомпилятор *TyDec* не поддерживает выполнение преобразований, обратных некоторым оптимизационным преобразованиям компилятора. Так, встраиваемые функции восстанавливаются как части функций, которые их вызывали, а развернутые компилятором циклы восстанавливаются как линейные последовательности итераций и т. д.

При полуавтоматическом восстановлении требуется поддерживать возможность управления специалистом процесса декомпиляции. Для этого инструментальное средство содержит развитый графический интерфейс.

Экспериментальная проверка декомпилятора *TyDec* проводилась на программах с открытым исходным кодом и наборе разработанных тестов. В таблице 5 представлены характеристики программ, для которых выполнялось сравнение качества восстановления, а также описание утилиты *derive*, которая вычисляет значение производной многочлена в точке. Утилита *derive* добавлена в набор программ для экспериментальной проверки восстановления программ, обрабатывающих вещественные значения. Колонка «*CLOC*» содержит количество строк кода в исходной программе, колонка «*ALOC*» содержит количество строк кода в ассемблерной программе.

Декомпилятор *TyDec* поддерживает множество входных форматов: программы на языке ассемблера, реализованные на диалектах *AT&T* и *Intel*, исполняемые модули, а также бинарные трассы выполнения, собранные на симуляторе процессора.

Результатом работы декомпилятора является типизированная структурная Си-программа. Схема архитектуры декомпилятора представлена на рисунке 1.

Многоугольниками представлены компоненты декомпилятора, а стрелками отображается поток данных между ними. Есть множество входных форматов: текстовый файл с ассемблерными инструкциями в нотации *AT&T* или *Intel*, исполняемые файлы, трассы.

«*ASM*-дерево» — это внутреннее представление входной программы в виде вектора инструкций. Модуль «*ASM*-дерево» отвечает за построение и хранение дерева внутреннего представления программы и предоставляет интерфейс для работы с ним другим компонентам системы.

Модуль «Граф потока управления» выполняет построение графа потока управления. Также модуль предоставляет интерфейс для работы с графом другим модулям системы.

Модуль «Восстановление функционального интерфейса» преобразует внутреннее представление программы, выявляя в нем

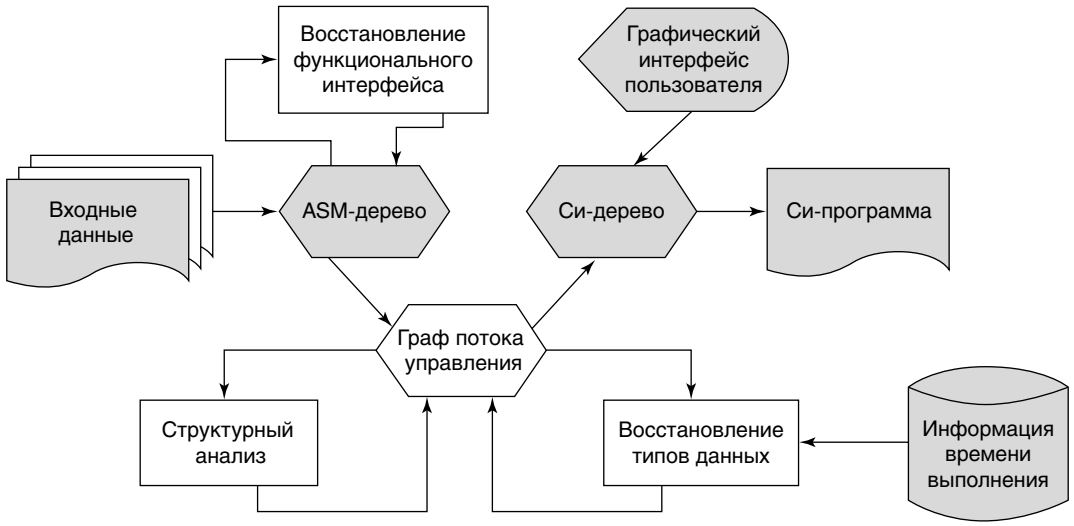


Рис. 1. Инструментальная среда декомпиляции *TyDec*

функции. Модуль «Структурный анализ» преобразует граф потока управления, восстанавливая управляющие конструкции программы.

Модуль «Восстановление типов данных» отвечает за восстановление типов данных декомпилируемой программы. В качестве входных данных модуля используется внутреннее представление программы (граф потока управления). По нему строится дерево зависимостей использования типов данных. Далее применяется алгоритм восстановления типов данных. Информация, полученная при динамическом анализе программы (профиль программы), подгружается непосредственно в модуль восстановления типов данных. Информация профилирования собирается вспомогательными утилитами.

«Си-дерево» — это компонент, обеспечивающий работу с внутренним представлением дерева Си-программы. Этот модуль по внутреннему представлению с использованием результатов работы модулей «Восстановление типов данных» и «Структурный анализ» на выходе строит программу на языке Си.

Модуль «Графический интерфейс пользователя» реализует графический интерфейс декомпилятора и поддержку диалогового режима работы эксперта.

На рисунке 2 представлен пример работы инструментальной среды декомпиляции *TyDec* на ассемблерных программах, которые были получены для Си-программ, листинги которых представлены выше. Эти программы были восстановлены плагином *Hex-Rays* к интерактивному дизассемблеру *IDA Pro* с не очень высоким уровнем качества. Проанализировав результат, можно увидеть,

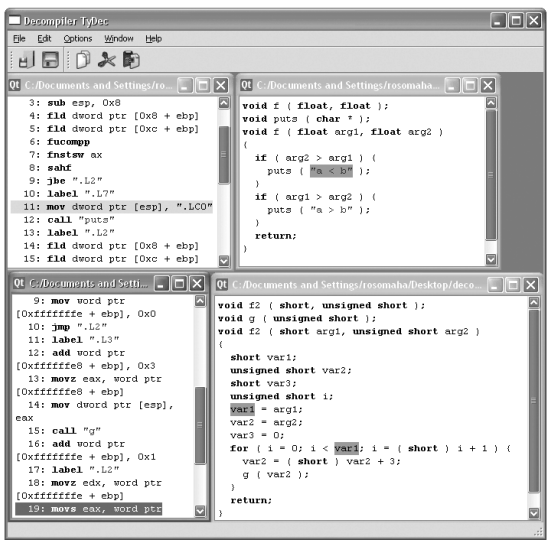


Рис. 2. Пример работы инструментальной среды *TyDec*

что декомпилятор *TyDec* восстановил обе программы с нулевой метрикой качества, что означает высокий уровень качества восстановления.

Компоненты системы

Реализация восстановления функционального интерфейса

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 процессора, функции в котором никак структурно не выделяются. Откомпилированные функции языка Си далее мы будем называть подпрограммами.

Точка входа подпрограммы — это первая инструкция подпрограммы, с которой начинается ее выполнение. *Точка выхода* подпрограммы — это последняя инструкция подпрограммы, после выполнения которой выполняется возврат подпрограммы в точку ее вызова.

Как правило, компиляторы генерируют код с одной точкой входа в функцию и одной точкой выхода из нее. Как уже говорилось, при этом в начало кода для функции помещается последовательность машинных инструкций, называемая прологом функции, а в конец кода помещается эпилог функции. И пролог, и эпилог как правило, стандартны для каждой архитектуры, и претерпевают на ней незначительные вариации.

Прологи и эпилоги функций легко могут быть выделены в потоке инструкций. При работе с трассами можно считать, что инструкции, на которые управление передается с помощью инструкции *call*, являются точками входа в функции, а инструкции *ret* завершают функции.

Если предположить, что все инструкции, расположенные между прологом и эпилогом, или между точкой входа и точкой выхода, составляют тело функции, то можно столкнуться с рядом трудностей.

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 в случаях, когда это возможно. Пример функции без стекового кадра представлен в листинге 3. В этом случае пролог и эпилог функции как таковые отсутствуют.

```
_foo:
        movl    4(%esp), %edx
        movl    8(%esp), %eax
        imull   %eax, %eax
        imull   %edx, %edx
        addl    %edx, %eax
        ret
```

Листинг 3. Пример функции без стекового кадра

Во-вторых, отдельные оптимизационные преобразования могут разрушать исходную структуру функций программы [11]. Очевидным примером такого оптимизационного 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 в листинге 4. Функция *_foo* возвращает значение функции *_f*, которая вызывается с теми же парамет-

рами, что и функция *_foo*. Компилятор сгенерировал пролог и эпилог для функции *_foo*, а вызов функции *_f* заменил безусловным переходом в середину ее тела.

```
_f:
        pushl  %ebp
        movl   %esp, %ebp
        movl   12(%ebp), %eax
        addl   8(%ebp), %eax
        leave
        ret

_foo:
        pushl  %ebp
        movl   %esp, %ebp
        leave
        jmp    _f
```

Листинг 4. Пример *sibling-call* оптимизации

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 таких «неструктурных» конструкций в машинной программе являются операторы обработки исключений таких языков, как *C++*.

Таким образом, хотя в обычном случае компилятор генерирует хорошо структурированный код, поддающийся разбиению на функции, достаточно легко может быть получен и «неструктурированный» код. Следует отметить, что в этом случае влияние программиста, пишущего на языке Си, на структуру генерируемого кода ограничено возможностями языка Си, не позволяющего бесконтрольную передачу управления между функциями и не поддерживающего механизм исключений. Поэтому можно предполагать, что если с языка ассемблера восстанавливается программа, полученная в результате компиляции программы на языке Си, то она не содержит «неструктурных» особенностей, описанных выше, и, таким образом, может быть разбита на функции.

Выявление параметров и возвращаемых значений

Языки высокого уровня, в частности, Си, поддерживают передачу параметров в функции и возврат значений. В языке Си поддерживается только передача параметров по значению, в других языках могут поддерживаться и другие механизмы. Заметим, что здесь мы рассматриваем 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 здесь понимается, как обычно, тип процессора и тип операционной системы, например, *Win32/i386* или *Linux x86_64*. Одной из целей *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. Например, соглашение о передаче параметров *_cdecl* используется по умолчанию в программах на Си и *C++* и имеет следующие особенности:

Параметры передаются в стеке и заносятся в него справа налево (т. е. первый в списке параметр заносится в стек последним).

1. Параметры выравниваются в стеке по границе 4 байта, и адреса всех параметров кратны 4. То есть параметры типа *char* и *short* передаются как *int*, но дополнительное выравнивание для размещения, например, *double* не производится.

2. Очистку стека производит вызывающая функция.

3. 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).

4. Регистры *%ebx*, *%esi*, *%edi*, *%ebp* не должны модифицироваться в процессе работы функции, т. е. при необходимости они должны сохраняться при входе и восстанавливаться при выходе из функции.

5. Значения целых типов, размер которых не превышает 32 бита, возвращаются в регистре *%eax*, 64-битных целых типов — в регистрах *%eax* и *%edx*, вещественных типов — в регистре *%st(0)*.

6. Если функция возвращает результат структурного типа, 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*. Если программа транслируется из автоматически полученного ассемблерного кода, который был получен либо компилятором, либо дизассемблером из бинарного файла, который не подвергался обфускирующим преобразованиям (т. е. преобразованиям, сохраняющим его функциональность, но затрудняющим анализ, понимание алгоритмов работы и модификацию при декомпиляции), 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. На первом этапе решения задачи выявления параметров функций следует определить следующие особенности вызова функций:

1. Используемое соглашение о передаче параметров (выбрать одно соглашение из набора предопределенных соглашений). Так как алгоритмы декомпиляции апробируются на программах, полученных компиляцией Си-программ, восстановление функций реализовано для соглашения о передаче параметров *_cdecl*.

2. Размер области параметров функции. Большинство соглашений о передаче параметров могут быть достаточно надежно идентифицированы по используемым инструкциям. Так как 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 размер параметров функции.

В случае вызова функции по указателю при статическом анализе может быть неизвестен адрес вызываемой функции. В этом случае отследить, как возвращается управление из вызываемой функции, не представляется возможным. 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.

Обнаружение функции main

Компиляторы языка Си до вызова функции *main* загружают в стек аргументы командной строки (количество аргументов командной строки передается через параметр *argc*, значения аргументов командной строки передается функции *main* в параметре *argv*). Эвристическим путем на основе анализа исполняемых файлов, сгенерированных компиляторами *MCVS* и *GCC* различных версий, было получено, что такому коду соответствует последовательность байтов:

8B 00 89 44 24 08 A1 00 ** 40 00 89 44 24 04 A1 04 ** 40 00 89 04 24,

где ** обозначает произвольный байт. Посредством сигнатурного поиска последовательность байтов, соответствующая загрузке аргументов командной строки, может быть идентифицирована. Аналогично, весь код, добавленный компилятором к пользовательскому, может быть найден, в результате чего будет получен адрес начала пользовательской функции *main*.

Если сигнатурный метод не находит соответствующую сигнатуру в коде, скорее всего, на вход декомпилятору была дана ассемблерная программа, исходная программа которой не была Си-программой. В этом случае автоматически найти точку начала пользовательского кода не удастся. При полуавтоматическом восстановлении специалист может вручную указать инструментальному средству ассемблерную инструкцию, которую следует считать началом пользовательского кода.

Реализация восстановления управляющих конструкций

Разные высокоуровневые управляющие конструкции программы могут отображаться в одну и ту же последовательность ассемблерных инструкций, например, один и тот же цикл может быть записан с помощью оператора *for*, *while* или даже посредством комбинирования операторов *if* и *goto*. При декомпиляции требуется восстановить управляющую конструкцию наиболее высокого уровня из подходящих, в частности, для данного примера наиболее предпочтительным будет восстановление оператора *for*, потом оператора *while*, а восстановление цикла посредством использования операторов *if* и *goto* вообще не желательно.

Структурный анализ, как в контексте прямой задачи — компиляции, так и в контексте обратной задачи, основан на анализе графа потока управления. В задаче декомпиляции граф потока управления строится по потоку ассемблерных инструкций.

Каждой подпрограмме в исходной программе соответствует свой граф потока управления. Отметим, что разбиение графа потока управления на подпрограммы проводится до структурного анализа.

Построение графа потока управления выполняется следующим образом: сначала вся последовательность инструкций разбивается на базовые блоки. В базовые блоки объединяются все инструкции, которые гарантировано выполняются последовательно. Границами базовых блоков являются метки, условные и безусловные переходы, вызовы функций, меняющих поток управления: *exit*, *_exit* и другие. Построенные таким образом базовые блоки являются вершинами графа потока управления. Далее строятся дуги графа, соответствующие всем возможным передачам управления между базовыми блоками. Если базовый блок завершается инструкцией перехода, то в граф потока управления добавляется дуга, соединяющая этот базовый блок с блоком, в который передается управление. Если базовый блок завершается условным переходом или инструкцией, не меняющей поток управления, то добавляется дуга в следующий базовый блок.

Следует заметить, что не все дуги графа потока управления могут быть построены в процессе статического анализа программы. Например, при косвенных переходах по таблице переходов, как правило, генерируемых при трансляции оператора *switch*, адрес перехода загружается из ячейки памяти. Хотя во многих случаях множество таких адресов переходов может быть построено, в общем случае эта задача алгоритмически неразрешима. Кроме того, в языках, поддерживающих обработку исключительных ситуаций, после инструкции вызова подпрограммы могут выполняться неявные переходы на обработчики исключений. Как следствие, основная сложность восстановления операторов *switch* и *try-catch* заключается в построении дуг графа потока управления, соответствующих неявным переходам.

После того, как граф потока управления построен, выполняется восстановление высокоуровневых управляющих конструкций.

Реализация метода восстановления управляющих конструкций основана на описанном в ранних работах [1, 2] алгоритме, однако имеет ряд модификаций.

Граф потока управления в процессе анализа перестраивается в граф регионов. Сначала выполняется разметка дуг графа на прямые, обратные и косые. Построение регионов выполняется итеративно. Изначально каждый базовый блок помечается как самостоятельный регион. Далее на граф накладываются шаблоны, соответствующие восстанавливаемым структурным конструкциям: *block*, *if-then*, *if-then-else*, *compound condition*, *endless loop*, *while*, *do-while*, *natural loop*, *switch*. Наложение шаблонов представляет собой обход в глубину графа потока управления. Если некоторый путь соответствует шаблону, то все базовые блоки этого пути выделяются в новый регион, после чего наложение выполняется со следующего не пройденного на данной итерации региона. Обход выполняется до тех пор, пока весь граф потока управления не будет представлять собой один регион.

Порядок наложения шаблонов влияет на структуру восстановленной программы. Экспериментально было установлено, что с точки зрения количества восстанавливаемых управляющих конструкций эффективнее сначала выполнять наложение шаблонов, соответствующих циклам, потом шаблонов, соответствующих ациклическим конструкциям.

Признаком цикла является наличие обратной дуги, входящей в регион, с которого начинается обход графа при наложении шаблона. Начиная с региона, в который входит обратная дуга, выполняется продвижение по графу по прямым дугам до тех пор, пока путь не дошел до вершины, из которой начался обход; это обязательно случится, т. к. в нее входит обратная дуга. Пройденные вершины должны быть восстановлены как цикл. Для определения типа цикла на выделенный подграф накладываются шаблоны циклов.

При восстановлении циклов особый анализ требуется для восстановления оператора продолжения выполнения *continue* и оператора выхода из середины цикла *break*. Оператор *continue* распознается по наличию дуги из тела цикла в заголовок, а оператор *break* — наличием дуги из тела цикла в выходную вершину цикла. Пусть признаком цикла было наличие обратной дуги из вершины *m* в вершину *n*. То есть вершина *n* доминирует над вершиной *m*. Тип цикла определяется по следующим признакам:

1. Если не найдена выходная вершина цикла, то такой цикл считается бесконечным.
2. Если у вершины *n* две исходящие дуги, то найден цикл с предусловием.
3. Если у вершины *m* две исходящие дуги, то найден цикл с постусловием.

После того, как выполнена проверка на наличие циклов, выполняется поиск ациклических управляющих конструкций. Сначала накладывается шаблон, соответствующий оператору множественного выбора *switch*. Потом накладывается шаблон *block*, а затем шаблоны, соответствующие условным переходам, причем сначала выполняется наложение шаблона условного перехода *if-then-else*, а потом — *if-then*.

Восстановление оператора switch

Восстановление оператора *switch* выполняется с учетом того, что, как правило, он переводится компилятором либо в последовательность сравнений, либо в переход по таблице. Первый способ реализации используется достаточно редко, а реализация через переход по таблице встречается почти всегда, особенно если значения, соответствующие меткам *case*, сгруппированы близко на оси целых чисел.

Первый случай не требует дополнительного рассмотрения, т. к. при такой реализации оператора *switch* он будет восстановлен в последовательность условных операторов *if*. Во втором случае в сегменте кода при дизассемблировании должны быть выделены таблицы переходов. Частично эта подзадача решается при разделении кода и данных.

Далее, после вычисления выражения, от которого зависит выбор пути исполнения, выполняется переход по неконстантному выражению. В листинге 5 приведен пример ассемблерного кода, соответствующий оператору выбора *switch* языка Си.

```
      cmpl    $6, -8(%ebp)
      ja      L10
      movl    -8(%ebp), %edx
      movl    L11(,%edx,4), %eax
      jmp     *%eax
      .section .rdata,"dr"
      .align 4
L11:
      .long   L3
      .long   L4
      .long   L5
      .long   L6
      .long   L7
      .long   L8
      .long   L9
      .text
```

Листинг 5. Результат трансляции оператора *switch*

Восстановление оператора *switch* выполняется по нескольким шаблонам. Один из шаблонов включает в себя инструкцию перехода по регистру или обращению к памяти. При нахождении такой инструкции отслеживается значение выражения, по которому происходит переход. Если оно имеет вид *L(,%reg, 4)*, где *L* — метка, то она считается указывающей на таблицу переходов для *switch*. В таком случае в граф потока управления добавляются дуги из вершины, содержащей переход по регистру, в *case*-вершины, метки которых записаны в таблице переходов. В процессе структурного анализа вершина-заголовок *switch* и *case*-вершины выделяются в регион типа *switch*.

Пример восстановления декомпилятором оператора *switch* по ассемблерному коду, приведенному в листинге 5, представлен в листинге 6.

Восстановление оператора *for*

Восстановление оператора *for* выполняется посредством наложения шаблона на восстановленный оператор *while*. Оператор цикла *for* характеризуется набором переменных (обычно их не больше двух), которые могут инициализироваться непосредственно перед телом цикла, в теле цикла изменяются не более одного раза и являются частью условия выполнения цикла. В листинге 7 представлен пример записи цикла *for* в виде цикла *while*.

Шаблон оператора *for* можно описать следующим образом. Если имеется множество переменных, для которых выполнены следующие условия, причем два последних условия выполнены обязательно, а первое может и не выполняться:

1. Каждая переменная встречается в левой части операторов присваивания, находящихся перед циклом.
2. Все переменные изменяются в теле цикла не более одного раза.
3. Условие перехода на очередную итерацию цикла или выхода из него зависит от всех переменных множества и только от них.

Тогда оператор *while* можно заменить оператором *for*, а управляющими переменными цикла *for* являются все переменные этого множества.

В операторе *for* могут отсутствовать все три управляющие конструкции оператора: инициализация, условие окончания и операция изменения управляющих переменных цикла. Если опущена инициализация, т. е. не выполнено первое условие для множества переменных, то оператор *while* все равно можно заменить оператором *for*, однако если не выполнены последние два условия, оператор *while* не заменяется оператором *for*.

Апробация модуля восстановления структурных конструкций

Результаты апробации реализации модуля «Структурный анализ» представлены в таблице 6. Колонки «*Src*» содержат количество соответствующих структурных кон-

Исходная программа	Восстановленная программа

switch (getch()) {
  case 'a':
  result = 1;
  break;
  case 'b':
  result = 2;
  break;
  case 'c':
  result = 3;
  break;
  case 'd':
  result = 4;
  break;
  case 'e':
  result = 5;
  break;
  case 'f':
  result = 6;
  break;
  case 'g':
  result = 7;
  break;
  default:
  result = 10;
  break;
	eax6 = getch ( );
	var8 = eax6 - 97;
	if (var8 <= 6){
	  switch (var8) {
	  case 0:
	  var9 = 1;
	  break;
	  case 1:
	  var9 = 2;
	  break;
	  case 2:
	  var9 = 3;
	  break;
	  case 3:
	  var9 = 4;
	  break;
	  case 4:
	  var9 = 5;
	  break;
	  case 5:
	  var9 = 6;
	  break;
	  case 6:
	  var9 = 7;
	  break;
	  }
	} else {
	  var9 = 10;
	}

Листинг 6. Оператор *switch* в исходной и восстановленной программах

струкций в исходной программе, «*Rec*» — количество конструкций в восстановленной программе. В статистику по восстановлению оператора выбора *switch* включены только те операторы, для которых была сгенерирована таблица переходов. Восстановленная программа в тесте *38_deflate.c* содержит больше условных конструкций, чем исходная, из-за того, что цикл *while* был оттранслирован в поток ассемблерных инструкций, соответствующих шаблону оператора *if-do-while*. Из статистики восстановления оператора *for* были исключены бесконечные циклы, записанные в исходных программах с использованием оператора *for*, т. к. ассемблерное представление бесконечного цикла, записанного с использование оператора *for*, идентично ассемблерному представлению бесконечного цикла, записанного с использование оператора *while*.

оператор *for*	оператор *while*

for(i=0, j=N; i < M && j; i++, j--) {
/*
тело цикла
*/
}
	i=0;
	j=N;
	while(i < M && j) {
	/*
	тело цикла
	*/
	i++;
	j--;
	}

Листинг 7. Запись цикла *for* в виде цикла *while*

Восстановление переменных и типов данных

Методы восстановления типов данных подробно описаны в статье [2]. Ниже представлено краткое описание моделей восстановления и построенных на их основе алгоритмов.

Модели восстановления типов данных оперируют объектами. Объекты строятся из конструкций ассемблерной программы, в которые отображаются переменные исходной программы, а именно:

1. Регистры общего назначения центрального процессора.

2. Непосредственно адресуемые области памяти:

- ячейки памяти по абсолютным адресам, которые соответствуют глобальным переменным в исходной программе;
- ячейки памяти по фиксированным смещениям относительно стекового кадра, соответствующим локальным параметрам;
- ячейки памяти по фиксированному смещению относительно вершины стека, а также занесенных в стек соответствующими инструкциями. Они соответствуют фактическим параметрам в вызываемых функциях.

3. Косвенно-адресуемые области памяти, что соответствует разыменованию других объектов, которые возникают в результате выполнения операций доступа к памяти в ассемблерном коде.

Для каждого регистра общего назначения центрального процессора, локальной переменной и параметра предварительно строится двудольный граф «определения-использования», где в левой доле находятся определения (т. е. присваивание значений)

Таблица 6

Результаты восстановления структурных конструкций

Пример	*if (Src)*	*if (Rec)*	*while (Src)*	*while (Rec)*	*switch (Src)*	*switch (Rec)*	*for (Src)*	*for (Rec)*
35_wc.c	25	25	4	4	1	1	3	2
36_cat.c	37	20	4	4	1	1	2	1
37_execute.c	70	94	13	16	2	2	1	0
38_day.c	46	33	0	5	1	1	1	8
39_deflate.c	50	34	11	5	0	0	37	5
59_lalr.c	29	38	8	9	0	0	36	34
Всего	257	244	40	43	5	5	93	50

соответствующих регистров, локальных переменных или параметров, а в правой — использования. Для каждой компоненты связности графа строится один объект, таким образом, одному регистру, локальной переменной или параметру могут соответствовать несколько объектов.

Для восстановления типов данных в данной работе разработано две модели: модель восстановления базовых типов данных (*char*, *unsigned char*, *short int*, *int*, …, *float*) и модель восстановления производных типов данных (массивы, структуры, указатели произвольного уровня косвенности).

Основная задача, которая должна быть решена при восстановлении *базовых типов данных*, это определить:

1. Имеет ли объект указательный, вещественный или целый тип.
2. Для объектов целого или вещественного типа определить размер типа в байтах.
3. Для объектов целого типа определить, знаковый ли это тип данных или беззнаковый.

Все *базовые типы* языка Си отображаются в модельные типы, которые представляются в виде тройки *<core, size, sign>*. *Ядро (core)* может принимать значения, указательный (*pointer*), целый (*int*) или вещественный (*float*). *Размер (size)* может принимать значения: 1, 2 или 4 (для 32-битной архитектуры). *Знак (sign)* может принимать значения: «знаковый» (*signed*) или «беззнаковый» (*unsigned*). Множество модельных типов — это множество троек, полученных отображением базовых типов языка Си, следовательно, оно не содержит троек, которые не соответствуют ни одному базовому типу языка. В процессе восстановления типов данных для каждого объекта строится модельный тип. Для этого вводится понятие *обобщенного модельного типа* данных. О*бобщенный модельный тип* данных — это тройка множеств *<CORE, SIZE, SIGN>*, где

$$CORE \subseteq \{pointer, int, float\},$$
$$SIZE \subseteq \{1, 2, 4\} \text{ и}$$
$$SIGN \subseteq \{signed, unsigned\}.$$

Изначально каждый объект характеризуется обобщенным модельным типом, каждый атрибут которого — полное множество, т. е.

$$< \{pointer, int, float\}, \{1,2,4\}, \{signed, unsigned\} >.$$

Далее по ассемблерной программе строятся три системы уравнений типов для каждого из атрибутов модельного типа. Например, по ассемблерной инструкции *add* r_1, r_2, r_3, которая складывает регистры r_1 и r_2 и помещает результат в регистр r_3, для атрибута *sign* строится уравнение $obj_3 : <sign_3> = obj_1<sign_1> + obj_2<sign_2>$, где объекты obj_1, obj_2, obj_3 построены по регистрам r_1, r_2, r_3 соответственно, а $<sign_1>$, $<sign_2>$, $<sign_3>$ — это атрибуты обобщенных восстанавливаемых типов объектов obj_1, obj_2, obj_3 соответственно. Каждая система решается итеративным алгоритмом, который основан на продвижении значений. Сбор информации по всем атрибутам выполняется независимо от значений остальных атрибутов тройки.

Начальные значения для каждого атрибута объекта уточняются ограничениями, накладываемыми регистрами процессора, ассемблерными инструкциями и т. д.

Определены пять типов ограничений:

1. *Регистровое ограничение*, оно влияет на атрибут *core* и атрибут *size* типа объекта.
2. *Командное ограничение*, оно влияет на все атрибуты типа объекта.
3. *Флаговое ограничение*, оно влияет на атрибут *sign* соответствующего типа объекта.
4. *Ограничение окружения*, оно влияет на все три атрибута типа объекта. Это ограничение накладывается, если в исходной программе использовались стандартные функции. Типы параметров и возвращаемых значений стандартных библиотечных функций предполагаются известными.
5. *Ограничение профиля* является дополнительным. Это ограничение предоставляет информацию о том, что тип объекта не

является указательным или не является знаковым. Ограничения профиля строятся на основе результатов динамического анализа программы.

Продвижение значений атрибутов выполняется на основании правил работы с типами данных, зафиксированных в стандарте языка Си. Для вычисления атрибутов используется решетка свойств с монотонной невозрастающей функцией слияния (пересечение множеств).

Приведем пример системы уравнений типов для атрибута *sign*, состоящую из двух уравнений.

$$\begin{gathered}\langle sign_p\rangle\ \{unsigned\}\ + \\ \langle sign_q\rangle\ \{signed, unsigned\} = \\ \langle sign_r\rangle\ \{signed, unsigned\}\end{gathered} \quad (1)$$

$$\begin{gathered}\langle sign_v\rangle\ \{signed, unsigned\}\ + \\ \langle sign_q\rangle\ \{signed\} = \\ \langle sign_r\rangle\ \{signed, unsigned\}\end{gathered} \quad (2)$$

В уравнении (1) тип объекта obj_p беззнаковый, о знаковости типа объекта obj_q нет информации, он может быть как знаковый, так и беззнаковый, но, согласно стандарту языка Си, тип объекта obj_r должен быть беззнаковый. В уравнении (2) из уравнения (1) уже известно, что тип объекта obj_r беззнаковый, следовательно, согласно стандарту, тип объекта obj_v тоже должен быть беззнаковый, т. к. тип объекта obj_q знаковый согласно уравнению (2).

Нахождение решения всех систем уравнения выполняется до достижения неподвижной точки. Далее по найденному обобщенному модельному типу выполняется построение модельного типа, в свою очередь по которому находится соответствующий базовый тип языка Си.

Модель восстановления *производных типов данных* основывается на представлении адресов обращения к памяти в канонической форме:

$$(base + offset + \sum_{j=1}^{n} C_j x_j),$$

где *base* — это база, т. е. базовый адрес, который является объектом при восстановлении базовых типов, *offset* — это константное смещение, значение которого известно при статическом анализе, $\sum_{j=1}^{n} C_j x_j$ — мультипликативная составляющая. При условии, что исходная Си-программа строго удовлетворяет стандарту, *base* имеет указательный тип. Для каждой инструкции обращения к памяти во входной программе на языке ассемблера, за исключением инструкций обращения к локальным переменным, глобальным переменным и параметрам функций, строится локальное адресное выражение адреса обращения к памяти в данной инструкции. Например, инструкции *movl 12 (%ebx), %ecx* соответствует локальное адресное выражение *%ebx* + 12. Для аргументов локального адресного выражения выполняется прослеживание значений в обратном направлении либо до загрузки из памяти значения, либо до вычисления, отличного от сложения или умножения, либо до границы базового блока. Найденные выражения для аргументов подставляются в локальное адресное выражение, и после алгебраических преобразований строится полное адресное выражение, которое отображается в терм адресного выражения. Таким образом, полным адресным выражениям доступа к памяти во входной программе соответствуют термы адресных выражений в модели восстановления производных типов.

Все множество термов адресных выражений разделяется на множества термов с эквивалентными базами. Каждое такое множество соответствует одному производному типу данных. Для разделения множеств термов на множества с эквивалентными базами используется алгоритм прямого продвижения. Сначала каждый объект, соответствующий базе, помечается меткой. Далее выполняется прямое продвижение меток по входной программе в соответствии с правилами переписывания термов. Все термы с одинаковыми метками объединяются в одно множество, соответствую-

щее классу эквивалентности. В результате может оказаться, что количество полученных классов эквивалентности больше количества производных типов в исходной Си-программе. Это означает, что для статического анализа было недостаточно информации для точного определения классов эквивалентности баз. Однако количество различных множеств эквивалентных баз производных типов данных не меньше количества производных типов в исходной программе. Неточности восстановления статического анализа частично устраняются с помощью дальнейшего динамического анализа.

После того, как выполнено разделение множества термов адресных выражений на классы эквивалентных баз, для каждого класса эквивалентности собирается множество смещений. Обратное отображение модельного представления производного типа в тип языка Си выполняется посредством построения скелета производного типа для каждого класса эквивалентности. Смещения по всем базам отображаются в поля скелета типа. По множеству смещений в одном классе эквивалентности строится скелет структурного типа, в котором типы полей вычисляются с помощью алгоритма восстановления базовых типов.

```
[1] struct t {
[2] int of1;
[3] short of2;
[4] double of3;
[5] }
[6]
[7] void f(void) {
[8] struct t * tmp1, *tmp2, *tmp3;
[9]    tmp1=tmp2;
[10]  tmp1->of1;
[11]  tmp2->of2;
[12]  tmp3=tmp2;
[13]  tmp3->of3;
[14] }
```

Листинг 8. Пример работы алгоритма восстановления производных типов данных

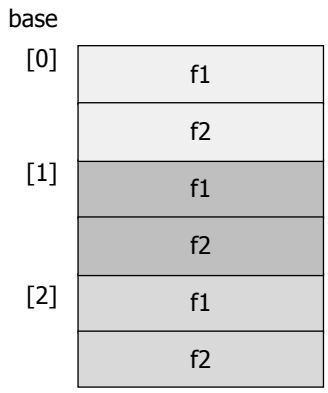
Объекты, соответствующие переменным *struct t *tmp1*, **tmp2* и **tmp3*, помечаются метками *L*1, *L*2 и *L*3 соответственно. Все три метки оказываются в одном классе эквивалентности баз после применения прямого продвижения меток. Операция присваивания в строке 9 определяет эквивалентность объектов, помеченных метками *L*1 и *L*2. Операция присваивания в строке 12 определяет эквивалентность объектов, помеченных метками *L*2 и *L*3. Далее строится множество смещений для нового структурного типа *S*. В соответствии со строками 10, 11 и 13 множество смещений для структуры *S* следующее: {*of*1, *of*2, *of*3}.

```
struct S{
    type1 of1;
    type2 of2;
    type3 of3;
}
```

Листинг 9. Пример скелета восстанавливаемой структуры *S*

Массивы восстанавливаются аналогичным способом. Если все смещения по эквивалентным базам имеют одну и ту же мультипликативную составляющую, восстанавливаемый производный тип объявляется массивом. Для дальнейшего восстановления строится объект для первого элемента массива. Размер элемента массива вычисляется как НОД всех мультипликативных составляющих. В некоторых случаях оказывается возможным восстановить размер самого массива. На рисунке 3 представлен пример восстановления массива структур.

Массив *arr* состоит из элементов, тип которых — структура, состоящая из полей *f*1 и *f*2. В исходной программе были инструкции обращения к полю *f*2 *i*-го элемента массива и полю *f*1 *j*-го элемента массива. На рисунке 3 представлены выражения в канонической форме, соответствующие этим инструкциям. Размер элемента массива составляет 8, однако разность между смещениями *f*2 элемента *arr*[*i*] и *f*1 элемента *arr*[*j*]




```
arr[i].f2 => arr + 4 + 8 * i;
arr[j].f1 => arr + 0 + 8 * j;
```

Рис. 3. Пример восстановления массива структур

составляет |4–0| = 4 < 8, что меньше размера элемента массива, и, следовательно, тип элементов массива *arr* — структура.

Экспериментальные результаты

Тестирование работы инструментальной среды *TyDec* было проведено более чем на 100 примерах, пятая часть которых — это программы с открытым исходным кодом. На всех примерах инструментальная среда показала высокое качество восстановления низкоуровневых программ.

Экспериментальное сравнение качества восстановления программ декомпилятором *TyDec* и декомпилятором *Hex-Rays* проводилось на 7 программах, 6 из которых — это утилиты с открытым исходным кодом, а последняя программа — это утилита, которая вычисляет значение производной многочлена в точке.

Таблица 7 показывает процент восстановления исходного кода для декомпилятора *TyDec* и декомпилятора *Hex-Rays* соответственно. Следует отметить, что программа *36_cat* и программа *37_execute* не были полностью восстановлены декомпилятором *Hex-Rays* из-за наличия оператора *switch*, восстановление которого он не поддерживает в полном объеме. В таблице 8

Таблица 7

Процент восстановления программы декомпиляторами

Название	*TyDec*	*Hex-Rays*
35_wc	100%	100%
36_cat	100%	**84%**
37_execute	100%	**0%**
38_day	100%	100%
39_deflate	100%	100%
59_lalr	100%	100%
84_derive	100%	100%

представлен подсчет количества штрафных баллов за использование низкоуровневых конструкций при восстановлении, а также за невосстановление высокоуровневых конструкций языка Си в соответствии с таблицей 4. Колонки «*OR*» содержат количество штрафных балов у исходных программ, колонки «*TD*» — количество штрафных балов у программ, восстановленных декомпилятором *TyDec*, и колонки «*HR*» — количество штрафных балов у программ, восстановленных декомпилятором *Hex-Rays*.

Вычисление качества восстановления программ обоими декомпиляторами в соответствии с формулой (1) представлено в таблице 9. Как можно заметить, на всех примерах декомпилятор *TyDec* показал существенно лучшее качество восстановления программ, чем декомпилятор *Hex-Rays*.

Заключение

Статья посвящена обзору нового инструментального средства декомпиляции программ — декомпилятору *TyDec*. Помимо корректности декомпиляции, при разработке декомпилятора *TyDec* большое внимание уделялось качеству восстанавливаемого кода. В статье вводится понятие качества декомпиляции, а также представлена мера качества декомпиляции программ. Декомпилятор *TyDec* восстанавливает все конструкции целевого языка программировании Си.

В настоящее время в институте системного программирования РАН ведутся рабо-

Таблица 8

Вычисление количества штрафных баллов

	35_wc			*36_cat*			*38_day*			*39_deflate*			*59_lalr*			*84_derive*		
	OR	*TD*	*HR*	*OR*	*TD*	*HR*	*OR*	*TD*	*HR*	*OR*	*TD*	*HR*	*OR*	*TD*	*HR*	*OR*	*TD*	*HR*
(type)	1	1	32	0	0	9	7	6	102	23	6	66	5	0	332	0	0	0
goto	1	4	2	0	3	7	0	5	0	0	0	1	0	1	0	0	0	2
break	0	0	5	8	8	10	0	0	0	1	3	0	5	3	9	0	1	1
continue	0	0	0	5	0	0	0	0	0	1	0	0	0	1	0	0	1	0
union	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
asm	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	48
switch	-	0	1	-	0	1	-	0	1	-	0	0	-	0	0	-	0	0
for	-	1	2	-	1	2	-	6	14	-	32	37	-	2	36	-	1	5
struct	-	0	1	-	0	0	-	0	1	-	0	0	-	0	1	-	0	0
[]	-	0	0	-	0	3	-	0	3	-	0	16	-	16	79	-	0	0
total	**5**	**15**	**52**	**13**	**18**	**59**	**14**	**33**	**230**	**48**	**44**	**201**	**15**	**41**	**539**	**0**	**3**	**204**

Таблица 9

Вычисление меры качества

Название	TyDec	HexRays
35_wc	41	195
36_cat	19	156
38_day	37	429
39_deflate	0	389
59_lalr	38	777
84_derive	42	2873

ты по реализации на основе декомпилятора *TyDec* надстройки к инструментальному средству *TrEx* [12], которое предоставляет широкий набор интерфейсов для анализа и обработки трасс программы.

Список литературы

1. *Е. О. Деревенец, К. Н. Трошина.* // Структурный анализ в задаче декомпиляции. Прикладная информатика №4 2009 г., Москва МаркетДС, стр. 87–99.
2. *Е. Н. Трошина, А. В. Чернов.* // Восстановление типов данных в задаче декомпилирования в язык Си. Прикладная информатика №6. 2009 г., Москва МаркетДС, стр. 99–117.
3. Boomerang Decompiler Home Page. http://boomerang.sourceforge.net/.
4. DCC Decompiler Home Page. http://www.itee.uq.edu.au/~cristina/dcc.html.
5. REC Decompiler Home Page. http://www.backerstreet.com/rec/.
6. Hex-Rays Decompiler SDK. http://www.hex-rays.com/.
7. Интерактивный дизассемблер и отладчик IDA Pro http://www.idapro.ru/.
8. *R. Cytron, J. Ferrante, B. Rosen, M. Wegman, F. Zadeck.* // Efficiently computing static single assignment form and the control dependence graph. ACM Transactions on Programming Languages and Systems. October 1991, 450–490 pp.
9. Tool Interface Standards (TIS). Executable and Linkable Format (ELF). http://www.x86.org/intel.doc/tools.htm.
10. Tool Interface Standards (TIS). Portable Executable Formats (PE). http://www.x86.org/intel.doc/tools.htm.
11. *J. Cavazos, G. Fursin F. Agakov.* Rapidly selecting good compiler optimizations using performance counters. // Proceeding of the international symposium on the Code Generation and Optimization, 2007 г.
12. *А. Ю. Тихонов, А. И. Аветисян, В. А. Падарян.* Методика извлечения алгоритма из бинарного кода на основе динамического анализа. // Проблемы информационной безопасности. Компьютерные системы. №3, 2008. стр. 66–71.