О некоторых задачах обратной инженерии

К.Н. Долгова, А.В. Чернов
{katerina, cher}@ispras.ru

Аннотация. В статье даётся краткое введение в проблематику задачи декомпиляции программ как одной из задач обратной инженерии. Рассматриваются возможности и недостатки существующих инструментальных средств декомпиляции программ.

1. Введение

В настоящее время в комплексном программном обеспечении широко применяются программные приложения, разработанные сторонними производителями. В ряде случаев такие приложения предоставляются без исходного кода на языке высокого уровня, необходимого для их аудита с точки зрения информационной безопасности их использования. Несмотря на это, такие приложения обязательно должны быть исследованы для оценки рисков их использования. Ни бинарный код, ни ассемблерный листинг, полученный в результате дизассемблирования, не позволяют с приемлемыми трудозатратами оценить взаимосвязь элементов программы, а также идентифицировать в программе стандартные алгоритмические конструкции. Восстановление программы на языке высокого уровня дает возможность преодолеть указанные выше трудности. Программные приложения, представленные в виде исполняемых файлов или на языке ассемблера, сложны для анализа их специалистами в области информационной безопасности, криптографии и т.д. и должны быть предоставлены им для анализа на более высоком уровне представления. В качестве одного из инструментальных средств повышения уровня абстракции представления программы может использоваться декомпилятор.

Под *декомпилятором* мы будем понимать инструментальное средство, получающее на вход программу на языке ассемблера и выдающее на выход эквивалентную ей программу на некотором языке высокого уровня.

Задача декомпиляции была поставлена в 60-е годы XX века сразу же, когда стали широко применяться компиляторы с языков высокого уровня, но не утратила своей актуальности и по сей день [2]. Эта задача не решена в полной мере из-за наличия ряда трудностей принципиального характера. В частности,

при компиляции программы из языка высокого уровня в язык ассемблера характерно отображение «многие к одному» концепций языка высокого уровня в концепции языка ассемблера, и, как следствие, однозначное восстановление программы на языке высокого уровня становится зачастую невозможным.

В силу указанных выше причин полностью автоматический декомпилятор реализовать принципиально невозможно. Поэтому системы декомпиляции программ должны работать во взаимодействии с аналитиком, который (зачастую методом проб и ошибок) управляет процессом декомпиляции. В ходе декомпиляции программы решаются следующие задачи: выделение структурных единиц программы, в частности, подпрограмм в однородном ассемблерном листинге, выявление параметров подпрограмм и возвращаемых ими значений, структурный анализ, то есть восстановление операторов циклов, ветвлений и т. п., восстановление типов данных, как базовых, так и производных и другие. Поскольку все эти задачи достаточно трудоемки и алгоритмически неразрешимы, на сегодняшний день нет известных декомпиляторов, восстанавливающих программы в какой-либо язык высокого уровня, которые качественно справлялись бы со всеми перечисленными выше задачами. Для решения задач посредством использования декомпиляторов требуется хорошо представлять возможности используемого инструмента, и для достижения наилучшего результата, возможно, потребуется использовать набор декомпиляторов в некоторой композиции. В данной работе предлагается обзор наиболее известных декомпиляторов в язык Си из бинарных файлов, рассматривается набор тестов, на основе которого можно сделать сравнительный анализ работоспособности декомпиляторов, и выполняется этот анализ.

В данной работе в качестве процессорной архитектуры, с которой ведётся декомпиляция, выбрана архитектура Intel i386, наиболее распространённая в настоящее время. В листингах фрагментов программ на языке ассемблера используется синтаксис AT&T [3].

Предлагаемая работа имеет следующую структуру. Поскольку и в литературе, и на практике зачастую смешиваются понятия дизассемблирования программы и декомпиляции программы, уместно рассмотреть различия этих задач. Этому посвящен второй раздел статьи. В третьем разделе статьи дается описание основных подзадач декомпиляции с описанием возникающих трудностей при их решении. В четвертом разделе приводится обзор языка Си с точки зрения обратной инженерии. Пятый раздел посвящен описанию существующих декомпиляторов для языка Си. В пятом разделе представлены результаты сравнительного тестирования декомпиляторов на разработанном наборе тестовых примеров. В заключении сформулированы выводы работы и направления дальнейших исследований.

2. Декомпиляция и дизассемблирование

Рассмотрим независимо друг от друга задачу дизассемблирования и задачу декомпиляции программ. Под декомпиляцией понимается построение программы на языке высокого уровня, эквивалентной исходной программе на языке низкого уровня (языке ассемблера). Под дизассемблированием понимается построение программы на языке ассемблера, эквивалентной исходной программе в машинном коде. Программа в машинном коде представляется либо в виде исполняемого модуля в стандартном для целевой операционной системы формате (например, для Win32 в формате PE [16], а для Linux – в формате ELF [15]), либо в виде дампа содержимого памяти, либо в виде трассы исполнения программы.

Традиционно декомпиляция рассматривается в более широком смысле, а именно, как построение программы на языке высокого уровня по программе в машинном коде. Очевидно, что в такой постановке задача декомпиляции поглощает задачу дизассемблирования. Такое «широкое» понимание декомпиляции излишне, поскольку дизассемблирование и декомпиляция решают разные по сути задачи, хотя и используют схожие методы (в частности, построение графа потока управления и исполняемого покрытия программы). Так, при дизассемблировании выполняется трансляция исполняемого файла, представляемого в виде набора машинных команд, в программу на языке ассемблера. При декомпиляции программа с представления низкого уровня транслируется в представление высокого уровня. Дальнейшим этапом повышения уровня абстракции программы может быть рефакторинг, посредством которого из программы на языке Си можно, например, получить программу на языке Си++.

Рассмотрим разбиение задач декомпиляции и дизассемблирования на подзадачи. Так, при дизассемблировании требуется решать следующие основные задачи:

- *Разделение кода и данных*. Для каждой ячейки программы (или ячейки памяти дампа) должно быть установлено, хранит ли ячейка исполняемые инструкции или данные. Задача эта сама по себе алгоритмически неразрешима [5] и не всегда может быть решена однозначно (например, в случае самомодифицирующегося кода, динамически подгружаемого кода и т. п.).
- *Замена абсолютных адресов на символические.*

При декомпиляции должны быть решены следующие основные задачи:

- Выделение функций в потоке инструкций.
- Выявление параметров и возвращаемых значений.
- Восстановление структурных конструкций языка высокого уровня.
- Замена всех обращений к памяти на конструкции языка высокого уровня (в частности, сюда входит идентификация обращения к локальным переменным и параметрам и их замена на символические имена, идентификация обращений к массивам и их замена на операции с массивами и т. д.).
- Восстановление типов объектов языка высокого уровня, выявленных на предыдущем шаге.

В дальнейшем мы будем рассматривать задачу декомпиляции в узкой постановке, то есть как задачу трансляции программы, представленной на языке низкого уровня, в частности, на языке ассемблера, в программу на языке высокого уровня, в частности, на Си.

3. Обзор основных подзадач декомпиляции

Рассмотрим основные задачи декомпиляции и подходы к их решению.

3.1. 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. И 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 трассами можно считать, что инструкции, на которые управление передается с помощью инструкции **call**, являются точками входа в функции, а инструкции **ret** завершают функции. Возникает соблазн считать инструкции, расположенные между прологом и эпилогом, или между точками входа и выходом, телом функции, однако в этом случае можно натолкнуться на ряд сложностей. 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. Во-вторых, отдельные оптимизационные преобразования могут разрушать исходную структуру функций программы. 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. 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++.

Таким образом, хотя в типичном 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 программу на языке Си, на структуру генерируемого кода ограничено возможностями языка Си, не позволяющего бесконтрольной передачи управления между функциями и не поддерживающего механизм исключений. Поэтому можно предполагать, что если восстанавливается программа с языка ассемблера, полученная в результате компиляции программы на языке Си, то она не содержит «неструктурных» особенностей, описанных выше, и может быть разбита на функции.

3.2. 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. 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 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 [9]:

1. Параметры передаются в стеке и заносятся в стек 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.

Отметим, что этот набор правил – это именно 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 (либо компилятором, либо дизассемблером), 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. 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.

3.3. Структурный анализ

Одним из результатов предыдущих фаз анализа ассемблерного листинга программы является разбиение потока инструкций ассемблерного листинга на отдельные функции и выявление точек входа в функции и возврата из функций.

Инструкции ассемблерной программы в функции могут рассматриваться как представление нижнего уровня (Low-level intermediate representation) [12]. В частности, представление низкого уровня отличается от представления высокого уровня (программы на языке Си) отсутствием структурных управляющих конструкций (**if**, **for** и т. п.).

Для восстановления управляющих конструкций сначала строится граф потока управления программы. По графу потока управления строится дерево доминаторов, затем дуги графа потока управления классифицируются на «прямые», «обратные» и «косые».

На основании этой информации уже можно выполнять непосредственно структурный анализ, то есть восстановление высокоуровневых управляющих конструкций [6]. Поиском в глубину в графе выделяются шаблоны основных структурных конструкций, которые затем организуются в иерархическую структуру.

3.4. Восстановление типов

Задача автоматического восстановления типов данных на настоящее время – одна из задач в области декомпиляции, наименее проработанных с теоретической точки зрения. Ее можно условно разделить на подзадачу

восстановления базовых типов данных языка, таких как *char*, *unsigned long* и т. п., и на подзадачу восстановления производных типов, таких как типы структур, массивов и указателей. В работе [13] рассматривается восстановление как базовых, так и производных типов при декомпиляции, однако этот подход имеет ряд существенных недостатков, и отсутствует его практическая реализация. В работе [4] описан подход к автоматическому восстановлению производных типов языка по исполняемому файлу. Такой подход используется для анализа на уязвимость программ в виде исполняемых файлов и поэтому не применим напрямую к задаче восстановления типов при декомпиляции.

На практике же все декомпиляторы, кроме Hex-Rays, вообще не восстанавливают даже базовые типы переменных, а в выражениях используют явное приведение типов, что делает восстановленные выражения сложными для понимания и модификации.

4. Языки высокого уровня с точки зрения обратной инженерии

Языки высокого уровня позволяют повысить уровень абстракции представления реализуемого алгоритма, избавляя программиста от необходимости заботиться о низкоуровневых деталях. Эти языки соперничают друг с другом по простоте использования и гибкости, а разработчики компиляторов соперничают по производительности сгенерированного ими кода. Следовательно, имеется большое количество разнообразных языков высокого уровня, и для каждого из них существует множество компиляторов.

При восстановлении программ по программе на языке низкого уровня, имея широкое представление о языке высокого уровня, нужно с достаточной точностью восстановить то, что было написано на языке высокого уровня в исходном тексте программы. Точность и трудозатраты восстановления программы сильно зависят от языка высокого уровня, на котором была написана исходная программа.

Язык Си формально считается языком высокого уровня, однако в нем присутствует много черт языка низкого уровня. В частности, в языке Си поддерживается прямой доступ к памяти и работа с указателями. При обращении к элементам массива не контролируется выход за его пределы, то есть возможен доступ к областям памяти, не имеющим никакого отношения к массиву. С другой стороны, в языке Си поддерживаются такие высокоуровневые конструкции, как производные типы данных: массивы, структуры, объединения, а также условные операторы, циклы и т. д.

На практике особую значимость имеют декомпиляторы, транслирующие ассемблерный листинг в язык Си. Во-первых, восстанавливать программы, написанные изначально на языке Си, удобно, потому что это процедурный язык и у него много низкоуровневых особенностей. Во-вторых, язык Си

широко применяется в промышленном программировании, и большое количество системных приложений написано именно на языке Си. С другой стороны, восстанавливать программу из ассемблера в объектно-ориентированный язык принципиально сложнее, да и к тому же программа, реализованная на основе процедурной парадигмы программирования, может быть переведена в объектно-ориентированную программу посредством рефакторинга ее кода. Следовательно, в данной работе ограничим множество рассматриваемых декомпиляторов теми, которые восстанавливают на языке Си программы, представленные либо на языке ассемблера, либо в виде исполняемых файлов.

5. Декомпиляторы в язык Си

В данном разделе дается краткое описание существующих на сегодняшний момент декомпилятров в язык Си. Это – декомпиляторы Boomerang [5], DCC [8], REC [14] и плагин Hex-Rays [10] к дизассемблеру IdaPro [11]. 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. Тем не менее, для простоты мы в дальнейшем объединим процесс дизассемблирования с использованием Ida Pro и последующей декомпиляции.

5.1. Boomerang

Декомпилятор Boomerang [5] является программным обеспечением с открытым исходным кодом (open source). Разработка этого декомпилятора активно началась в 2002 году, но сейчас проект развивается достаточно вяло. 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.

5.2. DCC

Проект по разработке этого декомпилятора [8] 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.

5.3. REC

Этот проект [14] 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 достаточно вяло. 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.

5.4. Hex-Rays

Как уже говорилось, инструмент Hex-Rays [10] не является самостоятельным программным продуктом, а распространяется в виде плагина к дизассемблеру IdaPro [11]. 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 [1] и алгоритм поиска параметров в стеке PIT (Parameter Identification and Tracking).

В таблице 1 представлена сводная характеристика всех рассматриваемых декомпилятров.

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

Таблица 1. Сравнительный анализ декомпиляторов

6. Исследование возможностей декомпиляторов

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

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

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, поскольку, как оказалось, декомпиляторы Boomerang и REC некорректно обнаруживают точку начала программы на Си в файлах формата PE.

Качество работы каждого декомпилятора для каждого теста оценивалось по четырехбальной экспертной шкале, приведенной в таблице 2.

Так, оценка «3» выставлялась в случаях, когда в декомпилированной программе использовались адресная арифметика вместо массивов или приведение типов для получения указательных значений вместо корректного объявления типов переменных. Кроме того, оценка «3» выставлялась, если в

результате декомпиляции цикл **for** оказывался преобразовыванным в цикл **while**.

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 в ней используются конструкции, отличные от конструкций исходной программы.
4	Декомпилятор выдал программу, которая эквивалентна исходной, и в которой используются те же конструкции, которые использовались в исходной программе.

Таблица 2. Шкала оценки декомпиляторов

6.1. Система тестов

Тестовый набор содержал следующие основные группы.

1) **Типы.** В тестах этой группы проверялась корректность восстановления типов переменных и параметров функций. В языке Си поддерживается 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. Проверялись факт обнаружения того, что переменная обладает указательным, а не целым типом, а также 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 указательных, массивовых и структурных типов и оценена корректность восстановления таких составных типов. 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, то есть 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, так как 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 в ситуации, когда 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, так как 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. п.). Реализация сигнатурного поиска присутствует в декомпиляторе DCC, но наиболее развита эта технология в декомпиляторе Hex-Rays.

6.2. Результаты тестирования

В таблице 3 приводятся результаты работы декомпиляторов на выбранном наборе тестов в соответствии с системой оценок, приведенной 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 декомпиляторы генерируют в этом случае некорректный код на языке ассемблера. Только декомпилятор REC сумел восстановить цикл **for**, в то время как остальные декомпиляторы в этом случае генерируют программу, использующую цикл **while**.

		Bom-merang	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

Таблица 3. Результаты тестирования декомпиляторов

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

7. Заключение.

В данной работе рассмотрена задача декомпиляции как восстановления программы на языке высокого уровня по программе на языке ассемблера или в машинных кодах. Дано краткое описание основных шагов процесса декомпиляции программы. В работе представлено сравнительное тестирование существующих декомпиляторов и указаны их сильные и слабые стороны. Так, ни один существующий на данный момент декомпилятор не поддерживает в достаточной мере восстановление типов данных, как базовых, так и производных. Существующие декомпиляторы испытывают сложности с восстановлением цикла **for** и оператора **switch**. Наиболее развитым из всех декомпиляторов является Hex-Rays, однако он является коммерческим и с закрытыми исходными кодами, поэтому его доработка невозможна.

Поэтому представляются актуальными разработка и реализация алгоритмов, позволяющих восстанавливать базовые и производные типы данных в процессе декомпиляции. Алгоритмы должны быть апробированы в рамках экспериментальной инструментальной среды декомпиляции программ. Разработка таких алгоритмов и инструментальной среды декомпиляции программ является направлением дальнейших исследований авторов.

Литература

[1] Гуильфанов И. FLIRT – Fast Library Identification and Recognition Technology. http://www.idapro.ru/description/flirt/
[2] Щеглов К.Е. Обзор алгоритмов декомпиляции//Электронный журнал «Исследовано в России». http://zhurnal.ape.relarn.ru/articles/2001/116.pdf
[3] AT\&T Assembly Syntax. http://sig9.com/articles/att-syntax
[4] G. Balakrishnan, T. Reps. Analyzing Memory Accesses in x86 Executables. Compiler Construction vol. 2985/2004, Springer Berlin / Heidelberg, 2004, стр. 5-23.
[5] Boomerang Decompiler Home Page. http://boomerang.sourceforge.net/
[6] C. Cifuentes, D. Simon, A. Fraboulet. Assembly to High-Level Language Translation. Technical Report 439. Department of Computer Science and Electrical Engineering. The University of Queensland. 1998.
[7] M. Davis. Computability and Unsolvability, New York: McGraw-Hill, 1958.
[8] DCC Decompiler Home Page. http://www.itee.uq.edu.au/~cristina/dcc.html
[9] Agner Fog. Calling conventions for different C++ compilers and operating systems.
[10] Hex-Rays Decompiler SDK. http://www.hex-rays.com/
[11] Интерактивный дизассемблер Ida Pro. http://www.idapro.ru/
[12] Steven S. Muchnik. Advanced Compiler Design And Implementation.
[13] A. Mycroft. Type-based decompilation. In European Symp. on Programming, 1999.
[14] REC Decompiler Home Page. http://www.backerstreet.com/rec/

[15] Tool Interface Standards (TIS). Executable and Linkable Format (ELF). http://www.x86.org/intel.doc/tools.htm
[16] Tool Interface Standards (TIS). Portable Executable Formats (PE). http://www.x86.org/intel.doc/tools.htm