Е. О. Деревенец, К. Н. Трошина

Структурный анализ в задаче декомпиляции

Декомпиляция — одна из сложнейших задач обратной инженерии. Одной из подзадач декомпиляции является задача восстановления управляющих конструкций. В работе подробно рассматриваются методы восстановления управляющих конструкций языка C и восстановление обработки исключительных ситуаций на примере языка C++ .

Введение

Декомпилятор — это программная система, восстанавливающая программы на языке высокого уровня из программ на языке низкого уровня, из объектного кода или из исполняемых файлов. Декомпиляция требует разработки алгоритмов и методов восстановления информации об исходной программе, которая была утрачена или существенно преобразована в процессе компиляции. Декомпиляция востребована в таких областях информационных технологий, как обеспечение информационной безопасности, поддержка унаследованного кода и т. д.

Обеспечение информационной безопасности требует проверки программы на наличие вредоносного кода, уязвимостей и решения других задач, для которых существуют развитые инструментальные средства, когда доступен исходный код приложений. В частности, для программ на языках высокого уровня, таких как C и C++, существуют развитые инструментальные средства поиска ошибок [1], удобной навигации по исходному коду [2] и другие. Однако на практике часто приходится работать с приложениями, которые доступны только в бинарном виде и получить исходные коды которых не представляется возможным, во-первых, по причине того, что эти приложения были разработаны сторонними разработчиками, во-вторых, по причине утраты исходных кодов. Для программ в бинарном представлении инструментальные средства, позволяющие выполнять анализ их работы, развиты хуже, нежели аналогичные инструменты для программ высокого уровня. В частности, для программ на языке ассемблера разрабатывается инструментальное средство [3], которое позволяет анализировать ассемблерную программу на наличие ошибок и уязвимостей, но даже его наличие не обеспечивает возможность решения всех задач анализа низкоуровневых программ с точки зрения информационной безопасности.

Таким образом, разработка инструментального средства, позволяющего восстанавливать бинарную программу или программу на языке ассемблера в программу на языке высокого уровня, актуальна. В частности, наличие такого средства позволит анализировать низкоуровневые приложения, используя развитые методы и инструменты, предназначенные для анализа высокоуровневых приложений.

Декомпиляция может применяться и при анализе программ, исходный код которых доступен. В частности, при разработке высоконадежных приложений в расчет принимается даже возможность ошибок или некорректных оптимизаций в компиляторе, которые могут привести к неправильной работе программы и появлению уязвимостей [4]. Пример такой оптимизации — устранение компилятором Microsoft Visual C++ .NET вызова функции `memset`. Компилятор может заметить, что после вызова функции `memset` очищенный массив далее нигде не используется, и удалит ее вызов [5]. Однако функция `memset` может использоваться для очистки локального буфера, содержащего секретные данные, например пароль, которые должны быть удалены из памяти, чтобы их нельзя было извлечь посредст-

вом сохранения содержимого памяти или каким-либо иным способом. Восстановленный декомпилятором код может использоваться для поиска несоответствий между сгенерированным объектным кодом и исходной программой.

Данная работа посвящена одной из подзадач декомпиляции — восстановлению управляющих конструкций. В статье подробно рассмотрены методы восстановления структурных конструкций языка высокого уровня: `if-then`, `if-then-else`, `while`, `do-while`, `switch` и `try-catch` — по потоку ассемблерных инструкций. Заметим, что 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 операторов `for`, `while` или даже посредством комбинирования операторов `if` и `goto`. При декомпиляции требуется восстановить наиболее высокоуровневую управляющую конструкцию 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.

В статье описан метод восстановления управляющих конструкций, реализованный в декомпиляторе *TyDec*, который разрабатывается в Институте системного программирования РАН.

Восстановлению стандартных управляющих конструкций `if`, `while`, `for` посвящено достаточное количество работ, и существуют мощные алгоритмы, в то время как восстановлению оператора множественного выбора `switch` в теории и на практике уделено значительно меньше внимания. В статье описан метод восстановления оператора `switch`, который реализован в декомпиляторе *TyDec*. Также надо отметить, что на практике востребованы методы восстановления программ не только в язык C, но и в язык C++, однако наработок как практических, так и теоретических в этой области немного. Множество управляющих конструкций языка C++ расширяется относительно множества управляющих конструкций языка C операторами работы с исключительными ситуациями. В статье представлено описание метода, позволяющего восстанавливать операторы работы с исключительными ситуациями языка C++.

В работе предполагается, что декомпилятору на вход подается программа на языке ассемблера для архитектуры IA-32, полученная в результате работы компилятора или восстановленная из объектного кода с помощью дизассемблера. Также предполагается, что исходная программа была написана на языках C или C++.

Описываемые в статье алгоритмы и методы с незначительными модификациями могут применяться и для восстановления программ для других платформ и на других языках, допускающих компиляцию в исполняемый код на машинном языке.

Статья имеет следующую структуру. В разделе 1 приводится обзор существующих методов структурного анализа. В разделе 2 представлены методы структурного анализа, реализованные в декомпиляторе *TyDec*. Подробное описание метода восстановления структурных конструкций, реализованного в декомпиляторе *TyDec*, представлено в разделе 3. В разделе 4 отражены особенности реализации обработки исключительных ситуаций в языке C++ и предлагается метод восстановления конструкций возбуждения и обработки исключений. В заключении представлены основные результаты работы, а также обозначены направления дальнейших исследований.

Обзор работ по восстановлению структурных конструкций

Одной из наиболее полных монографий, посвященных разработке трансляторов, является монография [6]. Традиционно компилятор имеет следующую структуру: лексический анализатор, синтаксический анализатор, семантический анализатор, оптимизатор и кодогенератор. Оптимизация и кодогенерация — это наиболее сложные этапы работы компилятора. Одной из подзадач оптимизации является структурный анализ или анализ потока управления.

В монографии [6] рассматриваются два метода анализа потока управления. Первый основан на построении доминирующих множеств вершин графа потока управления, а второй — интервальный анализ, который является частным случаем структурного анализа.

Анализ потока управления, основанный на построении доминирующих множеств вершин графа потока управления, позволяет работать только с циклами и без существенных модификаций неприменим для выявления условных операторов. В этом методе сначала вычисляется доминирующее множество вершин, выполняемое либо с помощью алгоритма последовательной итерации, либо с помощью алгоритма Ленгауэра-Тарьяна. Алгоритм последовательной итерации завершается при достижении доминирующими множествами неподвижной точки. Алгоритм Ленгауэра-Тарьяна работает быстрее, его временная сложность $O(n \times \alpha(n))$, однако он более сложный в реализации.

После того как доминирующее множество найдено, выполняется разметка дуг графа потока управления. Дуги графа потока управления классифицируются на прямые, обратные и косые. Прямая дуга — это дуга из доминирующей вершины в доминируемую, обратная дуга — это дуга из доминируемой вершины в доминирующую, все прочие дуги помечаются как косые. Размеченный граф потока управления позволяет выделить управляющие конструкции. Например, обратной дуге $m \rightarrow n$ соответствует цикл, состоящий из вершины n и всех вершин, из которых доступна вершина m по пути, не содержащему вершины n. В самом общем случае циклу в исходной программе соответствует компонента сильной связности графа потока управления.

Недостатком этого метода является небольшой набор обнаруживаемых структурных конструкций (только циклы) и относительная сложность реализации.

Интервальный анализ основан на последовательном выделении регионов различных типов и объединением их в одну новую вершину графа потока управления. *Регион* — это набор базовых блоков, имеющий не более одной входящей и не более одной исходящей дуги. Простейшим случаем региона является базовый блок. На первой итерации работы алгоритма все базовые блоки помечаются как самостоятельные регионы. Для выделения регионов строится дерево обхода графа потока управления в глубину. Вершины исследуются в обратном порядке обхода. Если два региона соединены только одной дугой, такие регионы объединяются. Если вершина является входной точкой циклической или ациклической управляющей конструкции, то регион, соответствующий этой конструкции, выделяется в новую вершину, и соответствующим образом корректируются дуги. Тип конструкции определяется последовательным сравнением подграфов, включающих рассматриваемую вершину, с соответствующими шаблонами. Алгоритм заканчивает работу, когда преобразованный граф не содержит дуг и состоит только из одного региона.

В самой простой версии интервального анализа выполняется выделение только циклов, а условные переходы не анализируются. Интервальный анализ является частным случаем структурного анализа, где помимо циклов выделяются еще и условные операторы, а именно, следующие типы регионов: `block`, `if-then`, `if-then-else`, `self loop`, `while loop`, `natural loop`, `improper interval` (неприводимый подграф, строго говоря, не являющийся регионом в нашем определении).

Диссертация [7] является одной из первых работ, посвященных разработке декомпилятора в язык С. Метод восстановления структурных конструкций в рассматриваемой работе реализован автором в декомпиляторе *DCC*. Структурный анализ основан на выделении в исходном графе потока управления управляющих конструкций посредством некоторого преобразования его в семантически эквивалентный граф.

Так как при декомпиляции в общем случае заранее неизвестно, на каком языке была написана исходная программа, автор выделяет множество управляющих конструкций, общих для большинства языков высокого уровня: `composition`, `conditional`, `pre-tested loop`, `single branching conditional`, `n-way conditional`, `post-tested loop`,

endless loop. В тех случаях, когда исходный граф не может быть структурирован с использованием предопределенного множества структурных конструкций, используется оператор goto.

Порядок выделения управляющих конструкций влияет на конечный вид графа. В предлагаемом автором методе используется следующий порядок выделения управляющих конструкций: n-way conditionals, loops, 2-way conditionals.

Для поиска циклов каждая вершина x проверяется на наличие входящего в нее обратного ребра из какой-либо вершины y, входящей в тот же регион. Цикл состоит из всех вершин, встречающихся в процессе обхода графа в глубину позже заголовка x и раньше замыкающей вершины y, принадлежащих одному и тому же интервалу и доминируемых заголовком цикла. Циклы, у которых заголовок или замыкающая вершина принадлежат другому циклу, считаются неприводимыми, и при их восстановлении используются операторы goto. После выявления циклов определяется их тип. Цикл с предусловием имеет заголовок с двумя исходящими ребрами и замыкающую вершину с одним обратным ребром в заголовок. Цикл с постусловием, наоборот, имеет замыкающую вершину с двумя ребрами: в заголовок и вовне цикла. В остальных случаях цикл считается бесконечным. Условные конструкции и конструкции сокращенного вычисления выражений определяются путем сравнения подграфов с образцами.

На практике встречаются неприводимые графы, содержащие несколько точек входа. Так как алгоритм выделения структурных конструкций не должен изменять семантику графа потока управления, дублирование вершин не применяют, и граф оставляют в исходном виде, а при восстановлении программы используют оператор goto.

На практике широко известны декомпиляторы *DCC*, *Rec32*, *boomerang* и надстройка *Hex-Rays* к интерактивному дизассемблеру *IdaPro*, которая позволяет автоматически восстанавливать дизассемблированный поток инструкций в программу на С-подобном языке. Так как все рассматриваемые инструментальные средства являются декомпиляторами в язык С, обработку исключительных ситуаций языка С++ ни один их них не поддерживает. Декомпиляторы *DCC*, *Rec32*, *boomerang* являются проектами с открытым исходным кодом, а надстройка Hex-Rays — это коммерческий продукт. Все рассматриваемые декомпиляторы успешно восстанавливают оператор if и операторы циклов while и do-while. Оператор for восстанавливает только декомпилятор Rec32, остальные инструментальные средства вместо него восстанавливают оператор while. Поддерживается восстановление оператора множественного выбора switch только в декомпиляторе *boomerang*.

Предложенные выше методы не полностью решают задачу структурного анализа. Методы, описанные в монографии [6], ориентированы на прямую задачу компиляции. Методы, описанные в работе [7], недостаточно полны, в частности, не поддерживается восстановление оператора множественного выбора switch и восстановление обработки исключительных ситуаций языка С++.

Структурный анализ в задаче декомпиляции

Структурный анализ, как в контексте прямой задачи — компиляции, так и в контексте обратной задачи — декомпиляции — 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 гарантированно выполняются последовательно. Границами базовых блоков являются метки, условные и безусловные переходы, вызовы функций, меняющие поток управления: exit, _exit, longjmp, __cxa_throw, __CxxThrowException и дру-

гие. Построенные таким образом базовые блоки являются вершинами графа потока управления. Далее строятся дуги графа потока управления, 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 и базовый блок, в который передается управление. Если базовый блок завершается условным переходом или инструкцией, не меняющей поток управления, то добавляется дуга в следующий базовый блок.

Следует отметить, 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.

Разработанный авторами метод восстановления управляющих конструкций основан на алгоритме, описанном в данной части статьи, однако имеет несколько 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`. Подграфы, соответствующие этим шаблонам, приведены на рис. 1. 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

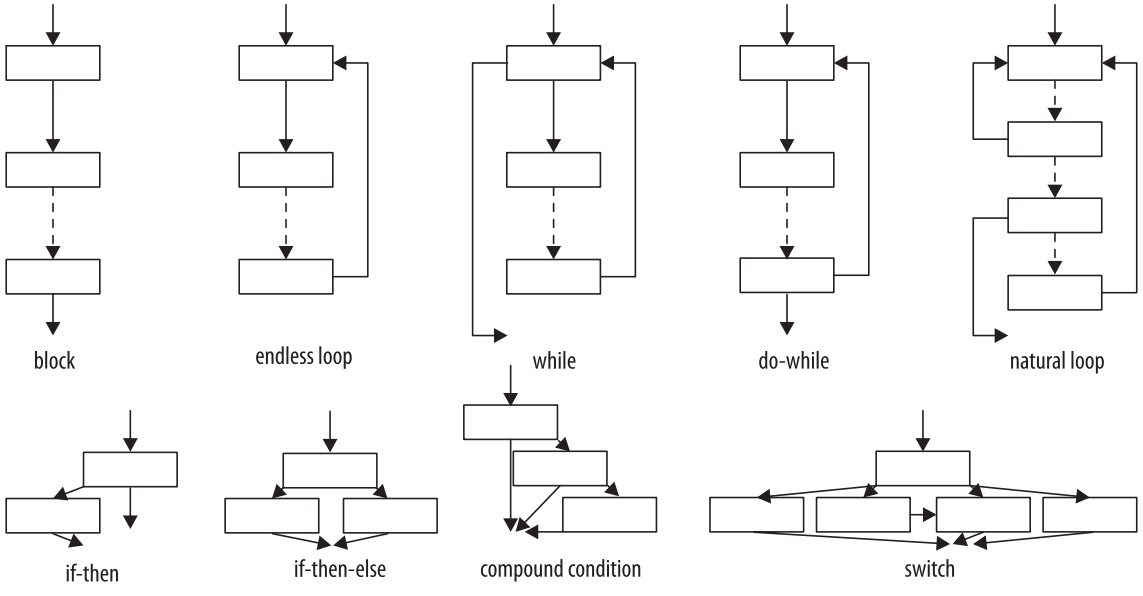

Рис. 1. Типы выделяемых регионов

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, так как 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 особого анализа требует восстановление 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` — по наличию 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:

- если не найдена выходная вершина цикла, то такой цикл считается бесконечным;
- если у вершины `n` две исходящие дуги, то найден цикл с предусловием;
- 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 — перехода `if-then`.

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, так как при такой реализации оператора `switch` он будет восстановлен в последовательность условных операторов `if`. Во втором случае после вычисления выражения, от которого зависит выбор пути исполнения, выполняется переход по неконстантному выражению. В следующем листинге приведен пример ассемблерного кода, соответствующего оператору выбора `switch` языка C.

```
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
```

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

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 (!((unsigned)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 = (int) 10;
}
```

Рис. 2. Оператор switch в исходной и восстановленной программах

Таблица 1

Примеры для тестирования

Пример	CLOC	ALOC	Описание
35_wc.s	241	465	Утилита wc, файл wc.c
36_cat.s	262	618	Утилита cat, файл cat.c
37_execute.s	788	1837	Утилита bc, файл execute.c
38_day.s	503	1383	Утилита calendar, файл day.c
39_deflate.s	763	669	Утилита gzip, файл deflate.c
59_lalr.s	711	1664	Утилита yacc, файл lalr.c

Таблица 2

Результаты восстановления структурных конструкций

Пример	Операторы `if`		Операторы циклов		Операторы `switch`	
	Src	Rec	Src	Rec	Src	Rec
35_wc.s	28	19	6	4	1	1
36_cat.s	40	11	6	2	1	1
37_execute.s	71	37	15	9	2	2
38_day.s	42	31	13	7	1	1
39_deflate.s	30	34	15	8	0	0
59_lalr.s	29	26	44	42	0	0
Всего	**240**	**158**	**99**	**72**	**5**	**5**

указывающей на таблицу переходов для оператора `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 выше, представлен на рис. 2.

Представленный метод восстановления структурных конструкций высокого уровня реализован в декомпиляторе *TyDec*, разрабатываемом авторами в Институте системного программирования РАН. Реализация была апробирована на наборе программ с открытым исходным кодом.

В табл. 1 представлены количественные характеристики тестовых примеров. Столбец CLOC содержит данные о количестве строк в коде исходной программы. Столбец ALOC содержит информацию о количестве строк в программе на языке ассемблера, полученной в результате компиляции исходной программы.

Результаты тестирования представлены в табл. 2. В столбцах Src приведены данные количестве соответствующих структурных конструкций в исходной программе, Rec — количество конструкций в восстановленной программе. В статистику по восстановлению оператора условного выбора `switch` включены только те операторы, для которых была сгенерирована таблица переходов. Восстановленная программа в тесте `38_deflate.s` содержит больше условных конструкций, чем исходная, из-за того что цикл `while` был оттранслирован в поток ассемблерных инструкций, соответствующих шаблону оператора `if-do-while`.

Обработка исключительных ситуаций языка C++

В контексте структурного анализа расширением языка C++ относительно языка C является возможность обработки исключительных ситуаций. Исключительные ситуации предназначены для обработки ошибок и являются удобным средством языка при разработке программ на основе независимо спроектированных модулей. При возникновении исключительной ситуации управление передается обработчику исключения. После того, как обработчик исключительной ситуации завершился, возможны два варианта передачи управления: либо на инструкцию программы, в которой возникла исключительная ситуация, либо на инструкцию, следующую за блоком обработки исключительных ситуаций. Обработка по первому сценарию называется *обработкой с продолжением.* Это устаревший сценарий обработки исключительных ситуаций, который

в настоящее время практически не поддерживается. Обработка по второму сценарию называется *обработкой без продолжения*. Такой сценарий поддерживается всеми современными языками программирования, поддерживающими обработку исключений, и реализован в языке C++.

Стандарт языка [8] определяет семантику обработки исключительных ситуаций, но не способ ее реализации. Формат реализации зависит от конкретного компилятора и платформы, для которой генерируется код.

Далее представлен обзор наиболее распространенных форматов реализации обработки исключительных ситуаций для архитектуры IA-32.

Формат обработки исключительных ситуаций DWARF2

Формат DWARF2 обработки исключительных ситуаций используется по умолчанию компиляторами GCC и Intel C++ Compiler на GNU/Linux. Общая схема поиска обработчика исключения приведена в [9]. Для раскрутки стека используется Unwind Library [10]. Информация о стековых фреймах, необходимая для раскрутки стека, содержится в секции `.eh_frame` исполняемого файла, формат которой аналогичен формату отладочной информации DWARF (это и дало название формату). Описание формата хранения отладочной информации можно найти в [11].

Информация об обработчиках хранится в секции `.gcc_except_table` в виде нескольких областей LSDA (Language Specific Data Area) [9], по одной на каждую функцию. Область данных LSDA представлена на рис. 3. Она содержит информацию о регионах в исполняемом коде функции Sites и соответствующих им обработчиках исключений, информация о которых записана в таблице Action Record Table. Каждому обработчику в Action Record Table поставлен в соответствие указатель на служебную информацию `typeinfo` типа обрабатываемого исключения. Указатель на соответствующую область данных LSDA содержится в структурах Frame Description Entry, находящихся в секции `.eh_frame`.

Для возбуждения исключительной ситуации используется функция `__cxa_throw`, принимающая в качестве аргументов указатель на область памяти, которая содержит объект исключения (данный участок памяти выделяется с помощью функции `__cxa_allocate_exception`), указатель на структуру `typeinfo` типа исключения и указатель на деструктор класса исключения (может быть равен нулю).

Главным достоинством формата DWARF2 является отсутствие накладных расходов при входе в `try`-блоки и выходе из них. В качестве

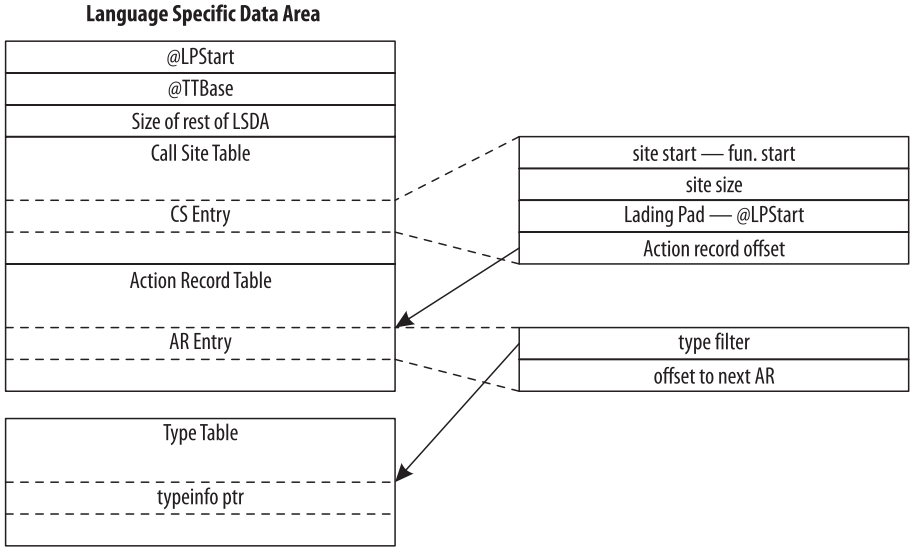

Рис. 3. Структура Language Specific Data Area

```
elf = ElfFile(filename)

eh_frame = elf.sections['.eh_frame']
gcc_except_table = elf.sections['.gcc_except_table']

for fde in eh_frame.frame_description_entries:
   print «region start is», fde.pc_begin
   print «region size is», fde.pc_range

lsda = gcc_except_table.get_lsda_at_offset(fde.lsda)
   for   site in lsda.call_sites_table:
         print «try-catch region is at offset», site.offset
         print «try-catch region size is», site.size

         offset = site.action_record_offset — 1

         while offset > = 0:
            action_record =
lsda.action_table.get_record_at_offset(offset)
            typeid =
lsda.type_table.get_id_at_offset(action_record.type_filter)
            typeinfo = elf.get_type_info(typeid)
            print "this region contains catch-block for type",
typeinfo.name
            if action_record.next_offset == 0:
              break
            offset = offset + action_record.next_offset
```

недостатка формата DWARF2 можно отметить сложность генерации исключительных ситуаций из обработчиков сигналов и других участков кода, стековый фрейм которых не формируется напрямую компилятором.

Так как вся информация, необходимая для обработки исключительных ситуаций, содержится в статических таблицах известного формата, то восстановление `try-catch` блоков заключается в разборе структур данных в секциях `.eh_frame` и `.gcc_except_table`.

В декомпиляторе *TyDec* реализован метод восстановления обработки исключительных ситуаций в формате DWARF2. Такое расширение декомпилятора позволяет выделять `try`-регионы вместе с соответствующими им \verb|`catch`|-блоками.

Восстановление конструкции возбуждения исключений `throw` сведено к поиску вызова функции __cxa_throw и определению ее фактических параметров.

Общая схема работы метода восстановления блоков `try-catch` представлена в листинге (см. выше).

Формат обработки исключительных ситуаций SjLj

Формат SjLj используется по умолчанию компилятором GCC на платформе Win32. Формат SjLj обработки исключительных ситуаций использует функции `setjmp()` и `longjmp()` или их альтернативные реализации для сохранения и восстановления контекста исполнения.

Во время работы процесса или потока на его стеке поддерживается односвязный список структур `SjLj_Function_Context`. При входе в функцию, содержащую `try-catch`-блок, на стеке формируется данная структура, содержащая указатель на соответствующую область данных LSDA (Language Specific Data Area), и помещается в конец списка вызо-

вом функции _Unwind_SjLj_Register. При выходе из функции последний элемент списка удаляется вызовом функции _Unwind_SjLj_Unregister.

Определения этих структур и функций можно найти в исходном файле компилятора GCC unwind-sjlj.c.

Как и в формате обработки исключительных ситуаций DWARF2, формат SjLj требует размещение области данных LSDA в секции исполняемого файла gcc_except_table. Структура области данных LSDA одинаковая как для формата DWARF2, так и для формата SjLj. Для возбуждения исключительной ситуации используется функция __cxa_throw.

Формат обработки исключительных ситуаций SjLj проще для реализации, чем формат DWARF2, так как в нем решена проблема с генерацией исключений из обработчиков сигналов. Однако этот формат требует накладных расходов при входе в функции, содержащие try-блоки, даже если код внутри этих блоков никогда не выбрасывает исключений.

Восстановление try-catch-блоков в формате SjLj выполнить сложнее, чем в формате DWARF2 из-за необходимости определения значений полей структуры SjLj_Function_Context, которая формируется динамически во время работы программы.

Исключения в компиляторе Microsoft Visual C++

В компиляторе Microsoft Visual C++ исключения реализованы поверх структурной обработки исключений (Structured Exception Handling). Структурная обработка исключений SEH поддерживает во время работы программы стек обработчиков исключений, который реализован в виде односвязного списка структур EXCEPTION_REGISTRATION и расположен в стеке процесса или потока [12]. Формат структуры EXCEPTION_REGISTRATION представлен в следующем листинге:

```
struct EXCEPTION_REGISTRATION
{
EXCEPTION_REGISTRATION *prev;
DWORD handler;
int id;
DWORD ebp;
};
```

Вершина стека процесса находится по адресу FS:[0]. Для реализации исключений языка C++ структура EXCEPTION_REGISTRATION расширяется дополнительными полями id и ebp. Расширенная структура представлена на рис. 4.

Обработка одного исключения в формате SEH, как правило, выполняется в два прохода: сначала поиск подходящего обработчика исключения, начиная с вершины стека, а потом — раскрутка стека до нужного состояния. Компилятор для каждой функции создает и регистрирует отдельный обработчик, код которого завершается вызовом функции __CxxFrameHandler. Код обработки исключения в формате SEH представлен в следующем листинге:

```
__ehhandler$?g@@YAXXZ:
mov edx, DWORD PTR [esp + 8]
lea eax, DWORD PTR [edx + 12]
mov ecx, DWORD PTR [edx-24]
xor ecx, eax
call @__security_check_cookie@4
mov eax, OFFSET
__ehfuncinfo$?g@@YAXXZ
jmp ___CxxFrameHandler3
```

Функции _CxxFrameHandler через регистр eax передается указатель на структуру funcinfo, содержащую указатель на таблицу try-блоков, каждая запись в которой содержит указатель на таблицу соответствующих catch-блоков [13]. Поля этой и связанных с ней структур представлены на рис. 5.

Формат обработки исключительных ситуаций, используемый в компиляторе Microsoft Visual C++, имеет наибольшую сложность восстановления, так как используемые при обработке исключений структуры данных формируются динамически в процессе работы программы.

Методы восстановления содержимого этих структур данных являются предметом дальнейших исследований.

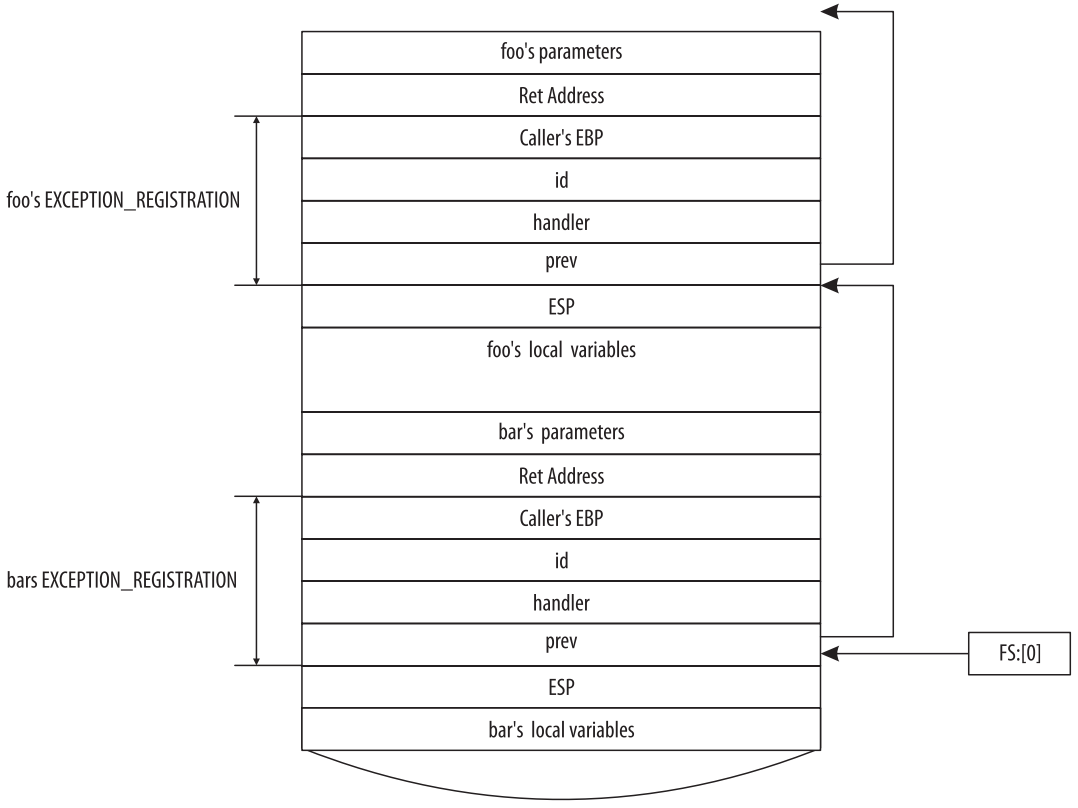

Рис. 4. Структура стека в программе, скомпилированной компилятором MSVC

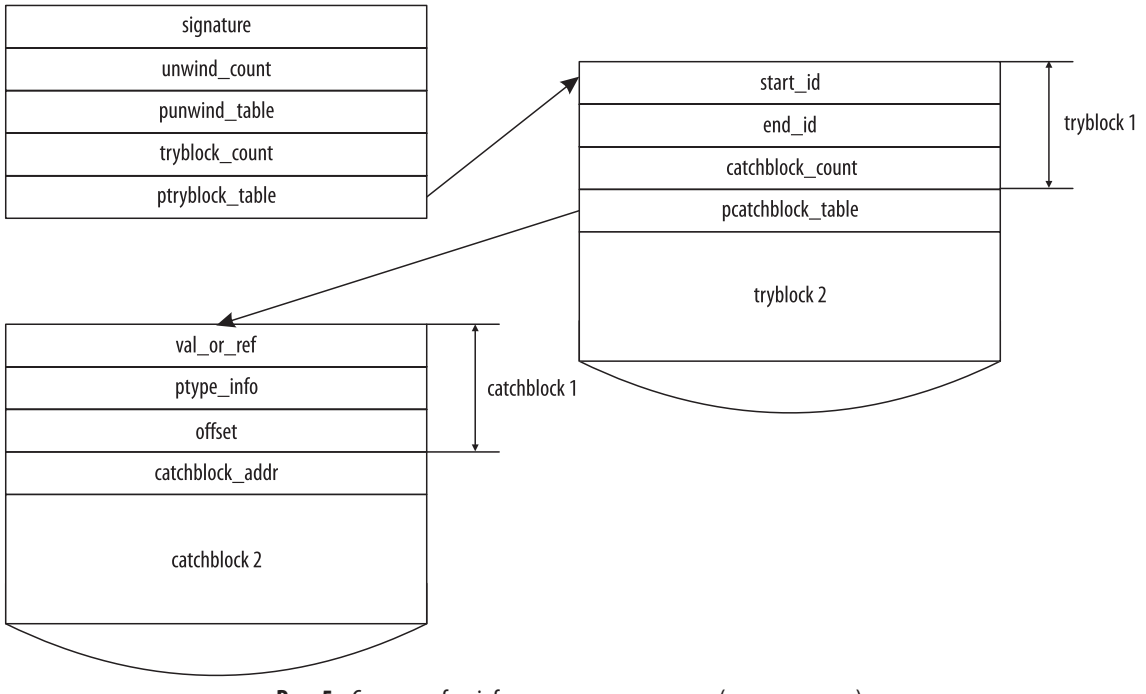

Рис. 5. Структура funcinfo и связанные структуры (основные поля)

В декомпиляторе *TyDec* реализован метод восстановления исключительных ситуаций в формате DWARF2. Ассемблерный листинг размечается на участки кода, которые заключены в исходной программе в try-блоки, а также выделяются участки кода, отвечающие за catch-блоки. Если в восстанавливаемой программе присутствуют фрагменты, соответствующие обработке исключительных ситуаций, то программа восстанавливается в программу на языке C++, в противном случае программа восстанавливается в программу на языке C.

Заключение

В работе представлен метод восстановления управляющих конструкций языков высокого уровня по ассемблерной программе. Помимо восстановления стандартных управляющих конструкций, таких как циклы и операторы условного перехода, описанный метод позволяет восстанавливать оператор множественного выбора. Предложенный метод реализован в разрабатываемом авторами декомпиляторе *TyDec* и апробирован на ряде программ с открытым исходным кодом. Результаты тестирования показали состоятельность представленного подхода.

Помимо этого в работе рассмотрены способы реализации обработки исключений в языке C++ различными компиляторами на архитектуре IA-32. В статье предложен метод восстановления информации об исключениях в формате DWARF2, используемом в ОС Linux и других операционных системах семейства Unix, так как этот формат позволяет полностью восстановить информацию об обработке исключений с помощью только статического анализа исполняемого файла. Представленный метод также реализован в декомпиляторе *TyDec*. Расширение декомпилятора возможностями восстановления структур `try-catch` блоков и типов обрабатываемых исключений позволяет декомпилировать программы, которые изначально были написаны не только на языке C, но и на C++.

Разработка методов, которые позволят восстанавливать конструкции обработки исключений в формате `SjLj` и в формате, поддерживаемом компилятором MSVC, является направлением дальнейших исследований.

СПИСОК ЛИТЕРАТУРЫ

1. Инструментальное средство CodeSonar. URL: http://www.grammatech.com/products/codesonar/

2. Инструментальное средство CodeSurfer. URL: http://www.grammatech.com/products/codesurfer/

3. *Balakrishnan and Ganai M.* PED: Proof-guided Error Diagnosis by Triangulation of Program Error Causes // Proc. of Software Engineering and Formal Methods (SEFM). 2008. November.

4. *Balakrishnan G., Reps T., Melski D., Teitelbaum T.* WYSINWYX: What You See Is Not What You eXecute // Proc. of VSTTE. 2005.

5. *Michael Howard.* Some Bad News and Some Good News. MSDN. 2002. October. URL: http://msdn.microsoft.com/en-us/library/ms972 826.aspx

6. *Steven S. Muchnick.* Advanced Compiler Design and Implementation. Chapter 7. Morgan Kaufmann, 1997.

7. *Cifuentes Cristina.* Reverse Compilation Techniques. PhD thesis. Queensland University of Technology. 1994. July.

8. Стандарт языка C++. ISO/IEC 14 882 : 1998. Standard for the C++ Programming Language.

9. Описание бинарного интерфейса компилятора языка C++ для архитектуры Itanium. Itanium C++ ABI, Section 7. Exception Handling Tables. URL: http://www.codesourcery.com/public/cxx-abi/exceptions.pdf

10. Описание бинарного интерфейса архитектуры AMD64. System V Application Binary Interface AMD64 Architecture Processor Supplement. Section 6.2. Unwind Library Interface. 2007. December. URL: http://www.x86–64.org/documentation/abi.pdf

11. Спецификация ядра ОС Linux. Linux Standard Base Core Specification 3.0RC1. Chapter 8. Exception Frames. URL: http://refspecs.freestandards.org/ LSB_3.0.0/LSB-Core-generic/LSB-Core-generic/ehframechpt.html

12. *Matt Pietrek.* A Crash Course on the Depths of Win32 Structured Exception Handling. Microsoft Systems. 1997. January. URL: http://www.microsoft.com/msj/0197/exception/exception.aspx

13. *Vishal Kochhar.* How a C++ compiler implements exception handling. 2002. April. URL: http://www.codeproject.com/KB/cpp/exceptionhandler.aspx