УДК 681.3.06

ВОССТАНОВЛЕНИЕ ТИПОВ ДАННЫХ С ИСПОЛЬЗОВАНИЕМ ИНФОРМАЦИИ ВРЕМЕНИ ВЫПОЛНЕНИЯ ПРОГРАММЫ

© 2009 г. В. Ю. Антонов, А. П. Фокин, К. Н. Долгова

avadim@gmail.com, apfokin@gmail.com, katerina@ispras.ru

Кафедра системного программирования

1 Введение

Программные приложения, предоставленные сторонними разработчиками, обычно нуждаются в анализе. Даже те приложения, которые программист разрабатывает сам, в случаях критичности приложения с точки зрения информационной безопасности, требуют дополнительного анализа. Однако чем выше уровень представления программы, тем более развитые инструментальные средства разработаны для их анализа. Так, инструментальные средства для анализа бинарного кода развиты слабо, а инструментальных средств для анализа приложений на языках высокого уровня Си и Си++ достаточно много.

Направление информационных технологий, которое занимается получение неявной информации из программного кода с целью повышения уровня представления программ называется *обратной инженерией*[1]. Типичный тракт повышения уровня программного приложения начинается с бинарного кода, который дизассемблером восстанавливается в ассемблерный листинг, который декомпилятором восстанавливается в программу на языке высокого уровня, а дальше используются различные инструментальные средства, повышающие уровень представления программы до языка спецификаций.

Одним из этапов работы декомпилятора является восстановление типов данных. Качество работы декомпилятора оценивается «понятностью» кода, который он восстанавливает, то есть, чем выше качество декомпилятора, тем меньше восстановленный им код отличается от того, который бы программист написал вручную. Точность и полнота восстановления типов данных без явных операций приведения типов и использования искусственных типов объединений сильно повышает качество декомпиляции.

В декомпиляторе TyDec, который разрабатывается в Институте системного программирования совместно с кафедрой системного программирования ВМК МГУ имеется модуль, который восстанавливает типы данных, основываясь только на статической информации, то есть только по ассемблерному листингу.

Существуют случаи, для которых статический анализ не позволяет восстановить типы данных однозначно. Так, в примере, представленном на рис. 1, при замене типа `int` переменной *s* на тип `char *` ассемблерный код не изменяется. Следовательно, декомпилятор не сможет однозначно восстановить тип переменной *s* при статическом анализе.

```
int f(int *a, int n) {
int s = 0;
int *p = a;
for (; p < a + n; ++p) {
  s += *p;
}
return s;
}
```

Рис. 1: Си программа, для которой неоднозначно восстановление типа переменной s.

Один из возможных вариантов разрешения такого рода неоднозначностей — спросить пользователя, другой — использовать информацию, собранную в процессе работы программы на некотором наборе тестовых данных в процессе восстановления. Собранную информацию назовем профилем программы, а процесс сбора — профилированием.

Профиль программы можно использовать как дополнение к статическим алгоритмам, когда статической информации просто недостаточно.

Данная работа имеет следующую структуру. В разделе 2 предоставляется обзор работ схожей тематики. Раздел 3 посвящен описанию предлагаемого метода сбора и анализа информации времени выполнения программы. В разделе 5 представлено описание системы `Valgrind` [2], используемой для реализации предлагаемого метода. В раздел 6 представлена реализация предлагаемого метода, а в разделе 7 представлены результаты апробации представленного метода. В заключении сформулированы выводы работы и направления дальнейших исследований.

2 Обзор работ схожей тематики

Существует большое количество методов и инструментальных средств, в которых они реализованы, которые позволяют извлечь неявную информацию о типах данных для анализа программы или повышения уровня ее представления. На практике используют методы, извлекающие неявную информацию о типах данных, как в статическом режиме, так и во времени выполнения.

В работе [4] представлен метод динамического восстановления абстрактных типов данных. Цель восстановления абстрактных типов состоит в том, чтобы сгруппировать переменные не только по размеру в памяти и типу операций, которыми они обрабатываются в ассемблерном коде, но и по особенностям их использования в программе. Наличие таких «укрупненных» типов данных позволяет лучше понимать назначение переменных в программе, что существенно упрощает ее поддержку и внесение изменений. Описываемый метод основан на том, что изначально каждому значению во время выполнения программы ставится в соответствие один абстрактный тип. Во время выполнения программы собирается информация о значениях, которые принимали все переменные, после чего в результате пересечения значений переменных они разделяются на укрупненные абстрактные типы данных. Информацию о времени выполнения программы собирается посредством утилиты, которая реализована с помощью системы `Valgrind`.

В работе [3] представлено инструментальное средство *hobbes*. Этот инструмент разработан для идентификации ошибок в программах на языке Си, которые могут возникнуть в результате неправильного использования особенностей языка. Утилита интерпретирует бинарный код, используя информацию о значениях, расположенных по адресам памяти, которые встречаются в регистрах и переменных. В качестве результата утилита выдает предупреждения, когда вероятность ошибки во время выполнения программы превышает установленную пользователем границу. Для анализа используется только динамический анализ, даже для тех случаев, когда статического анализа может быть достаточно. Однако в инструментальном средстве *hobbes* слабо развита обработка указателей, а именно, утилита не различает указатели на разные типы данных, вследствие чего ошибки времени выполнения, которые могут возникнуть из-за неправильной работы с указателями, не выявляются. Сбор информации о значениях, которые принимают переменные во время выполнения, выполняется системой динамически. Инструментальное средство состоит из двух частей: интерпретатора кода `x86` и модуля проверки типов. Интерпретатор кода работает как инструментирующий компилятор, который встраивает в код вызовы функций модуля проверки типов. Архитектура интерпретатора аналогична инструментирующему компилятору, на основе которого построена система `Valgrind`. Само инструментальное средство `Valgrind` в реализации не использовано, так как на момент начала разработки еще не существовало его работающей версии.

3 Профилирование значений ячеек памяти

В декомпиляторе *TyDec* реализованы мощные методы статического восстановления типов данных. В случаях, когда статических методов недостаточно для повышения точности восстановления типов данных можно использовать информацию времени выполнения программ, а именно, множество значений, которые хранились в ячейках памяти и регистрах процессора во время выполнения программы. В частности, можно с достаточной степенью достоверности определить, что некоторый регистр в некоторой точке программы хранит 32-х битное значения, которые являются адресами, что соответствует переменной указательного типа в исходной программе на языке Си, даже если она не разыменовывалась. Подобным образом можно сделать обоснованное предположение о знаковости значений в регистре и, соотвественно, о знаковости типа соотвествующей этому регистру переменной в исходной программе на языке Си.

Схема виртуального адресного пространства процесса для ОС Linux на 32-х битной архитектуре IA32 представлена на рис. 2. Каждому процессу выделяется диапазон виртуальных адресов от `0x00000000` до `0xFFFFFFFF` Операционная система выделяет процессу 3Гб, которые расположены непрерывно, начиная с ячейки с адресом `0x00000000` и до ячейки с адресом по `0xBFFFFFFF`. Оставшаяся память зарезервирована системой. Начиная с адреса `0x08048000` загружается копия исполняемого файла, сегмент данных и BSS сегмент. Кроме этого, в памяти расположены стек, куча и библиотеки. Следовательно, переменная указательного типа во время работы программы может принимать не любые значения, а только те, которые соответствуют адресам ячеек памяти виртуального адресного пространства процесса. Обозначим $ValidAddr$ — множество всех используемых адресов памяти программы. Очевидно, что оно включает в себя все перечисленные сегменты, используемые процессом.

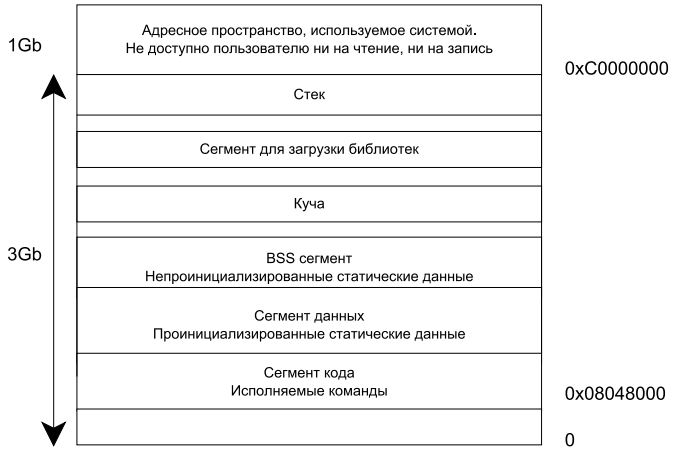

Рис. 2: Виртуальное адресное пространство процесса.

Обозначим через *obj* конструкции ассемблерной программы, соответствующие переменным на языке высокого уровня. То есть это:

- регистры общего назначения центрального процессора,
- ячейки памяти в абсолютной адресации, которые соответствуют глобальным переменным в исходной программе,
- ячейки памяти по фиксированным смещениям относительно стекового кадра, соответствующие локальным параметрам,
- ячейки памяти по фиксированному смещению относительно стекового кадра по регистру %esp, а также положенные на стек соответствующими инструкциями. Они соответствуют фактическим параметрам в вызываемых функциях.

В корректной программе на языке Си указатели, как правило, хранят адреса из множества отображенных виртуальных адресов (стек, куча, статические данные, код), либо специальное значение *null*. Значения, которые принимают знаковые целочисленные переменные, вне зависимости положительные они или отрицательные, сгруппированы около нуля, то есть значение -1 переменная принимает значительно чаще, нежели, например, $-2{\cdot}10^9$. Беззнаковые целочисленные переменные редко принимают значения, которые соответствуют -2 или -3 а, например, значение $3 \cdot 10^9$ переменные принимают чаще для 32-х битной архитектуры, нежели беззнаковое значение, соответствующее -2. Значение -1 необходимо рассматривать специальным образом, так как оно соответствует максимальному беззнаковому целому значению и часто выступает в роли индикатора ошибки.

Для 32-х битной архитектуры нужно профилировать значения только для 32-х битных ячеек памяти, так как указатели имеют размер 32-бита. Несмотря на то, что в идеале хорошо бы профилировать значения напрямую ячеек памяти, это на практике реализовать затруднительно, так как:

- значения разных переменных могут храниться в одних по одним и тем же адресам. Стек, как и куча, постоянно переиспользуются,
- одни и те же локальные переменные могут иметь различный адрес в разные моменты времени.

Однако для того чтобы использовать значение ячейки памяти ее необходимо загрузить на регистр. Следовательно, вместо профилирования непосредственно значений ячеек памяти, будем собирать значения, которые загружаются на регистр и выгружаются из регистров соответствующими командами.

Пусть VP — это отображение из множества $Addr$ виртуальных адресов сегмента кода программы во множество 32-х битных значений Val, которые записывались или считывались инструкцией по этому адресу. Множество адресов, которые не принадлежат сегменту кода программы, исключены из рассмотрения.

Виртуальные адреса должны быть отображены на инструкции дизассемблированной программы. Следовательно, отображение DM — это отображение из множества $Addr$ виртуальных адресов памяти в множество объектов Loc, соответствующих этим адресам памяти в дизассемблированной программе.

Отображение IVP из множества ассемблерных инструкций во множество 32-х битных значений, которые загружаются на регистр или выгружаются в память соответствующими инструкциями, строится посредством комбинирования отображений VP и DM

$$VP : Addr \to 2^{Val}$$
$$DM : Addr \to Loc$$
$$IVP : Loc \to 2^{Val}$$
$$IVP(x) = VP(DM^{-1}(x))$$

В качестве результата получаем, что каждой инструкции дизассемблированной программы ставится в соответствие множество 32-х битных значений, которое она загружает или выгружает в память. Если значение, которое загружается или выгружается не 32-х битное, или оно не было покрыто во время выполнения программы, оно считается не аннотированным.

Множество $VP(obj)$ — это множество значений, принимаемых объектом obj за время выполнения программы. Другими словами, это — множество значений, записываемых в ячейку памяти, выделенную для переменной, соответствующей объекту obj.

$null$ — это нулевой указатель.

$\sigma_{obj}(x)$ — функция, возвращающая 1, если $x \in VP(obj)$, и 0 в противном случае.

Рассмотрим две метрики:

$PC(obj)$ — мера уверенности «указатель». Принимает значения от 0 до 1. Чем ближе значение к 1, тем больше уверенность в том, что объект obj является указателем.

$$PC(obj) = \frac{|VP(obj) \cap ValidAddr|}{|VP(obj) - null|}$$

Так как указатель может принимать значение *null*, которое не включено в $ValidAddr$, *null* исключается из множества значений, принимаемых объектом *obj*.

$UC(obj)$ — мера уверенности «беззнаковый» для 32-х битной платформы. Принимает значения от $\frac{1}{2^{31}}$ до 1. Чем ближе значение меры к 0, тем больше уверенности, что переменная *obj* является знаковой.

$$UC(obj) = 1 - \sum_{x=-2^{31}+1}^{-2} \frac{\sigma_{obj}(x)}{2^{2*\lfloor log_2|x+1|\rfloor+1}}$$

Если переменная принимала значение -2, то мера уверенности «беззнаковый» уменьшается на $1/2$. Для значений -3 и -4 мера уверенности «беззнаковый» уменьшается на $1/4$. Для остальных отрицательных значений производятся аналогичные действия. Из этого следует метрика и её минимальное значение:

$$1 - \sum_{x=-2^{31}+1}^{-2} \frac{1}{2^{2*\lfloor log_2|x+1|\rfloor+1}} = \frac{1}{2^{31}}$$

Данные метрики положены в основу метода сбора и анализа информации времени выполнения программы для восстановления знаковости целочисленной переменной и отличия целого типа данных от указательного по бинарному коду исполняемой программы. Метод основан на построении профиля значений для каждой ячейки памяти в программе, соответствующей переменной, и вычисления для них метрик. На основе полученных значений, можно предположить, является ли переменная знаковой или беззнаковой, целочисленного типа или указательного.

4 Распознование инструкций memset и memcpy

Стандарт языка Си позволяет выполнять побайтовый доступ к объектам. Так, например, возможно скопировать значение одного структурного типа в другой структурный тип побайтно, используя функцию `memcpy`. Обычно компилятор вставляет целиком (*inline*) функции побайтового копирования `memcpy`, побайтового обнуления `memset` и тому подобные, вместо их вызова для оптимизации. Тела таких функций должны быть идентифицированы в декомпилируемом коде и не рассматриваться при восстановлении типов, так как они нарушают правила строгой типизации и правила распротстанения информации о типах по программе.

Распознавание инструкций `memset` и `memcpy` можно проводить либо сигнатурным анализом, либо анализом поведения во время выполнения программы. Сигнатурный анализ требует постоянно обновляющуюся базу сигнатур для всех поддерживаемых компиляторов. Анализ поведения во время выполнения программы проще при реализации и использовании.

Инструкции `memset` и `memcpy` в результате оптимизации компиляторов могут быть заменены набором идущих подряд считываний и присваиваний (см. Таблицу 2).

Например, функция `memset(p, 0, 2 * sizeof(*p))` транслируется в ассемблер в виде кода, состоящего из двух инструкций:

```
movl   $0, (%eax)
movl   $0, 4(%eax)
```

Функция `memcpy(q, p, 5 * sizeof(*p))` транслируется как

```
movl  (%edx), %eax
movl  %eax, (%ecx)
movl  4(%edx), %eax
movl  %eax, 4(%ecx)
movl  8(%edx), %eax
movl  %eax, 8(%ecx)
movl  12(%edx), %eax
movl  %eax, 12(%ecx)
movl  16(%edx), %eax
movl  %eax, 16(%ecx)
```

Таким образом, если во время анализа выполнения программы были найдены идущие подряд инструкции присваивания `movl` $x, addr_i$, для которых:

- Используется фиксированное значение x.
- Адреса ячеек памяти $addr_i$ расположены в возрастающем порядке или в убывающем и $|addr_i - addr_{i-1}|$ =размер ячейки памяти.

Тогда этот набор инструкций можно заменить на инструкцию `memset`, которая присваивает значение x ячейкам памяти, расположенным по адресам $addr_i$.

Если во время анализа выполнения программы были найдены чередующиеся инструкции считывания `movl` $addr1_i, eax$ и присваивания `movl` $eax, addr2_i$, для которых:

- Адреса ячеек памяти $addr1_i$ расположены в возрастающем порядке или в убывающем и $|addr1_i - addr1_{i-1}|$ =размер ячейки памяти.
- Адреса ячеек памяти $addr2_i$ расположены в порядке, соответствующем порядку ячеек памяти $addr1_i$, и $|addr2_i - addr2_{i-1}|$ =размер ячейки памяти.

Тогда этот набор инструкций можно заменить на инструкцию `memcpy`, которая копирует значения из ячеек памяти $addr1_i$ в ячейки памяти $addr2_i$.

5 Система Valgrind

При разработке утилит динамического анализа исполняемых файлов можно использовать для упрощения разработки системы построения инструментальных систем для динамического анализа. Особенностью таких систем является то, что для интеграции их с анализируемым приложением не требуется исходного кода приложения, а достаточно только исполняемого модуля. Одной из таких систем является система построения динамических инструментов для исполняемых файлов `Valgrind`

Система `Valgrind` позволяет писать утилиты на основе своего ядра, которые позволяют анализировать бинарные приложения. Особенностью системы являются развитая функциональность устойчивость приложений к ошибкам времени выполнения, высокая производительность за счет минимизации количества операций.

Инструментальное средство представляет собой надстройку, реализованную на языке Си, к ядру системы `Valgrind` В общем виде инструментальное средство можно представить следующим образом: *Valgrind core + tool plug-in = Valgrind tool*. Инструментальное средство статически линкуется к исполняемому модулю, в который добавляется дополнительный код для вызова методов ядра системы `Valgrind`.

На основе системы `Valgrind` реализовано много инструментальных систем, позволяющих выполнять динамический анализ приложений.

1. Система `Memcheck`[5] позволяет проводить анализ всех операций чтения и записи в памяти, выделение и освобождение блоков памяти. В результате, система позволяет выявлять следующие ошибки в приложении:
   (a) использование непроинициализированной памяти,

(b) чтение/запись памяти после её освобождения,

(c) чтение/запись несоответствующих адресов в стеке,

(d) утечки памяти,

(e) неправильное использование операторов `malloc`/`new`/`new []` и `free`/`delete`/`delete []`,

(f) перекрывающиеся области памяти в `src` и `dst` в `memcpy()` и других аналогичных функциях.

2. Система `Cachegrind`[2] позволяет находить места в программе, где имеется не оптимальное взаимодействие с типичным суперскалярными процессорами, в следствие чего время работы программы сильно увеличивается. Система симулирует программу и добавляет комментарии в исходный код об отсутствии нужных данных в кэше или неправильном прогнозировании ветвления.

3. Система `Callgrind`[2] позволяет построить граф вызовов для программного запуска. По умолчанию, система собирает информацию о количестве выполнившихся инструкций, количество вызовов между функциями, а так же считает отношение долю выполнившихся инструкций по отношению к инструкциям программы. Помимо этого симулятор кэша представляет информацию о поведении операций доступа к памяти.

4. Система `Helgrind` [2] позволяет выполнять анализ с целью выявления ошибок синхронизации потоков в программах, реализованных на языках Си, Си++ и Fortran, использующих примитивы стандарта POSIX.

Архитектура системы `Valgrind` основана на методе «Дизассемблирование и повторный синтез», который позволяет встраивать код анализа в исполняемый код программы. Реализация этого метода системе состоит из 8 этапов:

1. Дизассемблирование. На этом этапе исполняемый код программы преобразуется во внутреннее представление системы `Valgrind` в виде древовидной структуры.

2. Оптимизация, этап 1. Внутреннее представление оптимизируется и преобразуется в список.

3. Инструментация. На этом этапе выполняется добавление инструментированного кода во внутреннее представление программы.

4. Оптимизация, этап 2. На этом этапе выполняется дополнительное оптимизационное преобразование внутреннего представления программы.

5. Построение дерева. После того как выполнены оптимизационные преобразования внутреннее представление программы из списка перестраивается обратно в древовидную структуру. В течение этого шага ВП преобразуется в древовидную структуру.

6. Выделение инструкций. На этом этапе по внутреннему представлению в виде древовидной структуры выделяются инструкции, которые преобразуется в список.

7. Выделение регистров. Из списка инструкций перестраивается политика использования регистров, что необходимо в результате добавления инструментированного кода.

8. Сборка. На последнем этапе список инструкций преобразуется в машинный код.

Так как система `Valgrind` для инструментации не требует исходного кода и является продуктом, который распространяется свободно, она была выбрана для реализации утилиты, позволяющей собирать и анализировать информацию о типах данных, доступную только во время выполнения приложения.

Рис. 3: Общая схема работы модуля восстановления типов данных декомпилятора TyDec.

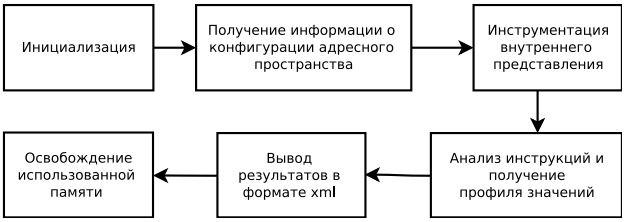

Рис. 4: Схема работы инструмента TDTrace

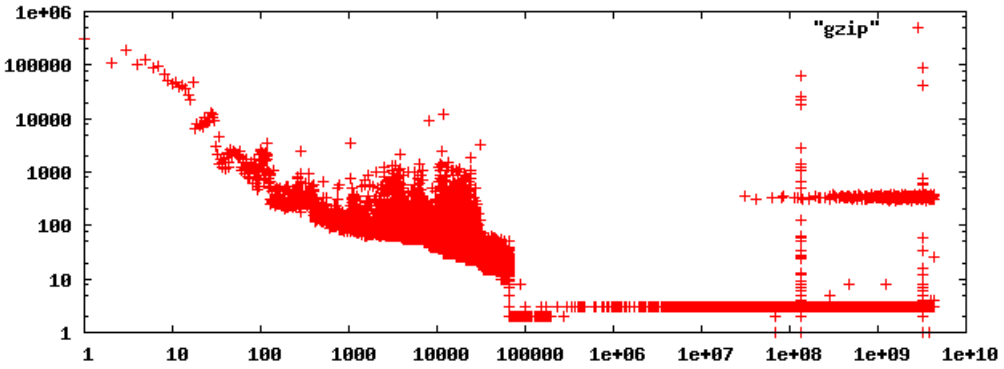

Рис. 5: Набор полученных значений ячеек памяти для программы gzip

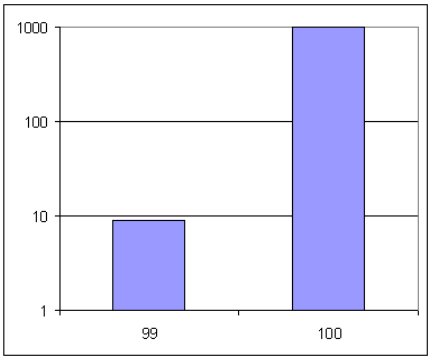

Рис. 6: Статистика полученных значений метрики $UC(obj)$ для программы gzip

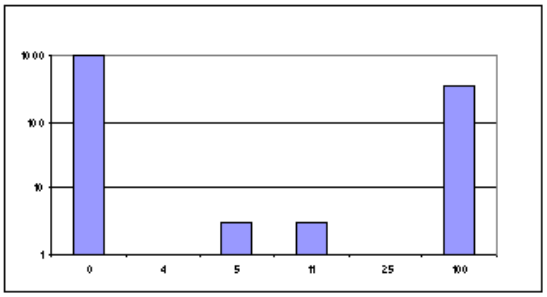

Рис. 7: Статистика полученных значений метрики $PC(obj)$ для программы gzip

```
...
<memoryAddress type="variable" addr="0x0804A008" pointer="0" />
<memoryAddress type="variable" addr="0x0804A00C" pointer="0" />
<memoryAddress type="variable" addr="0x0804A010" pointer="0" />
<memoryAddress type="variable" addr="0x0804A014" pointer="0" />
<memoryAddress type="variable" addr="0x0804A018" pointer="0" />
<memoryAddress type="variable" addr="0x0804A01C" pointer="0" />
<memoryAddress type="variable" addr="0x0804A020" pointer="0" />
<memoryAddress type="variable" addr="0x0804A024" pointer="0" />
<memoryAddress type="variable" addr="0x0804A028" pointer="0" />
<memoryAddress type="variable" addr="0x0804A02C" pointer="0" />
<memoryAddress type="variable" addr="0xBE9754A0" pointer="100" />
<memoryAddress type="variable" addr="0xBE9754A4" pointer="0" />
<memoryAddress type="variable" addr="0xBE9754A8" pointer="100" />
<memoryAddress type="variable" addr="0xBE9754B0" pointer="100" />
<memoryAddress type="variable" addr="0xBE9754B4" pointer="0" />
<memoryAddress type="variable" addr="0xBE9754D0" pointer="100" />
...
```

Таблица 1: Пример вывода утилиты TDTrace

6 Утилита TDTrace

На рис. 3 представлена общая схема работы модуля, восстанавливающего типы данных декомпилятора *TyDec*.

На вход модулю, выполняющему восстановление типов данных, подается ассемблерная программа в виде внутреннего представления. По внутреннему представлению строится начальное множество объектов, которое подается на вход алгоритму восстановления базовых типов данных. Результатом работы алгоритма является три группы объектов:

1. объекты, для которых базовые типы восстановлены,
2. объекты, для которых восстановлена информация о том, что они имеют некоторый указательный тип,
3. Объекты, для которых на данной итерации восстановить тип не удалось.

Объекты указательного типа передаются на вход алгоритму восстановления производных типов данных. Результатом работы этого алгоритма является новое множество объектов, построенное для каждого поля скелета производного типа и для первого элемента массива. Это новое множество объектов объединяется с множеством объектов, типы которых не были восстановлены алгоритмом восстановления базовых типов данных и передается опять на вход алгоритму восстановления базовых типов. Если у структуры все типы полей восстановлены, то такая структура считается восстановленной иотправляется во множество восстановленных типов. Массив считается восстановленным, если восстановлен тип первого элемента и размер массива.

Информация, собранная на основе профильного анализа, хранится отдельно и подгружается по запросу. Для профилирования разработана утилита `TDTrace`

Утилита `TDTrace` позволяет анализировать адреса и значения, используемые во время выполнения программы, а так же представляет полученную информацию в удобном для последующего использования виде. Утилита получает информацию о конфигурации адресного пространства, через ядро системы `Vagrind` анализирует все инструкции программы, составляя для каждого адреса набор записываемых туда значений, считает для каждого адреса характеристики $UC(obj)$ и $PC(obj)$ и выдаёт полученный результат в формате xml-документа.

Схема работы утилиты представлена на рис. 4.

Для удобства работы с утилитой были введены параметры запуска и реализован модуль, преобразующий полученный после работы инструмента xml-документ в объектную модель.

Параметры командной строки утилиты приведены ниже:

- Опции командной строки запуска. Каждая опция может принимать значение `yes` или `no`. Значение `yes` соответствует работе функции, значение `no` означает отключение функции.
  - Опция `print-all-readwrite-instructions` отвечает за вывод адресов всех инструкций считывания и записи в память.
  - Опция `print-all-ips` отвечает за вывод адресов всех инструкций программы и списка значений, принимаемых параметрами, для каждой из них.

7 Апробация

Утилита `TDTrace` была апробирована на нескольких программах с открытыми исходными кодами.

Ниже на рис. 5, рис. 6, рис. 7 приведены результаты работы утилиты `TDTrace` для программы gzip. На рис. 5 по оси x расположены значения, принимаемые ячейками памяти, по оси y расположена величина, отражающая частоту, с которой ячейки памяти принимали это значение. На рис. 6 и рис. 7 по оси x расположены значения соответствующих метрик, а по оси y расположены относительные частоты.

Проанализировав код программы gzip было установлено, что утилита `TDTrace` верно определила все переменные указательного и беззнакового типов.

Эксперименты показали хорошие результаты, что позволяет говорить о состоятельности подхода.

8 Заключение

В данной работе рассмотрены подходы к восстановлению типов данных в задаче декомпиляции и предложен метод сбора и анализа информации времени выполнения программы для восстановления типов данных по бинарному коду исполняемой программы. Дано описание системы `Valgrind`, используемой для реализации предлагаемого метода. В работе представлено описание инструментального средства `TDTrace`, реализующего предлагаемый метод, и результаты его тестирования. Результаты тестирования показали состоятельность предложенных метрик.

Представляется актуальным интеграция разработанного инструмента в инструментальную среду декомпиляции программ. Это позволит улучшить процесс восстановления типов данных. Улучшение полученных методик и интеграция в инструментальную среду декомпиляции программ является направлением дальнейших исследований авторов.

Список литературы

[1] Chikofsky, E.J.; J.H. Cross II (January 1990) *Reverse Engineering and Design Recovery: A Taxonomy in IEEE Software.* IEEE Computer Society

[2] Nicholas Nethercote and Julian Seward. (2007) *Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation.* Proceedings of ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007), San Diego, California, USA, June 2007.

[3] M. Burroes, S. Fruend, J. Wiener (2003) *Run-time type checking for binary programs* Proceedings of CC, p.90-105.

[4] P. Cuo, J. Perkins,S. McCamant and M. Ernst (2006) *Dynamic inference of abstruct types.* Proceedings of ISSTA, p.749-754.

[5] Описание системы *Memcheck*: http://hald.dnsalias.net/projects/memcheck/

[6] Описание систем, реализованных на основе системы Valgrind: http://valgrind.org/info/tools.html