Санкт-Петербургский государственный университет
Математико-Механический факультет
Кафедра системного программирования

Статическое разбиение дизассемблированного кода на линейные участки

Курсовая работа студента 444 группы
Анисимова Константина Александровича

Научный руководитель…….....Баклановский М.В.
старший преподаватель
кафедры системного программирования

Санкт-Петербург, 2014

Оглавление.

Введение.

Обратная разработка программ или реверс-инжиниринг уже давно стала не только средством поддержки устаревшего программного обеспечения. Сейчас очень часто применяют обратную разработку в целях взлома чужих алгоритмов и программ, что впоследствии используется либо для создания аналогичной программы с последующей ее продажей либо для выкладывания взломанной версии в открытый доступ с различными целями либо для анализа вредоносного программного обеспечения. В любом случае авторы оригинала теряют свою прибыль. По этому сейчас множество программ запутываются или обфусцируются с целью усложнения обратного анализа.

Здесь будут рассматриваться только программы в виде исполняемых файлов, которые не модифицируют собственный код, так как статический анализ самомодифицирующихся программ практически не возможен. Это связано с тем, что мы в общем случае статически не можем определить, будет ли анализируемый участок кода изменяться перед исполнением или нет.

Линейным участком (или базовым блоком) называется такая непрерывная структурная единица кода, которая всегда исполняется с начала и до конца, и в конце возможна передача управления в другое место.

Для обратной разработки программ необходимо уметь разбивать код на линейные участки, так как это первый шаг в восстановлении структуры программы. Но в данном случае есть возможность использовать динамический анализ кода, хотя в большинстве случаев весь код динамически не проанализировать и приходится неизбежно прибегать к статическому анализу.

Для обфускации это необходимо только в случае модификации графа потока управления (Control Flow Graph). Но обфускация графа потока управления производится обычно либо над исходным кодом программы либо в процессе ее компиляции. В итоге поскольку у нас есть доступ к исходному коду, то составить граф потока управления практически всегда не представляет трудности.

Так же знание всех линейный участков и в частности графа потока управления необходимо для статической бинарной трансляции, так как основной единицей

любой, в том числе и статической, бинарной трансляции, является как раз линейный участок.

Большинство средств для реверс-инжиниринга имеют средства анализа графа потока управления, но для этих целей нет необходимости построения полного графа. Из всех комплексных программ для реверс инжиниринга, статически строит и анализирует линейные участки и граф потока управления только Interactive DisAssembler(IDA) Pro. Но данный продукт является коммерческим.

Описание проблемы

Хотелось бы исследовать возможность статического анализа программы в целях построения графа потока управления программы исключительно по исполняемому коду программы. Будем рассматривать архитектуры Intel x86 и x86_64 как одни из самых популярных. Так как компиляция под разными Операционными Системами отличается в основном только способом системных вызовов, и форматом исполняемого файла, а способы компиляции ЯВУ в исполняемый код практически не меняются, то можно производить исследование на одной ОС, которой была выбрана ОС Windows.

Каким образом можно построить CFG? Очевидно, что мы можем начать с самого начала исполняемого кода, дизассемблировать его линейно с начала и до конца, а потом разбивать на линейные участки, встречая инструкцию передачи управления. Но у этого способа только один плюс - мы найдем все линейные участки. А недостатки очень серьезные: граф скорее всего не будет полным, так как мы не знаем откуда начнется исполнение программы; так же будет много лишних линейных участков. Таким образом поступают в основном разнообразные отладчики, например, OllyDbg.

Значит лучше начать с тех мест, где программа может начать свое исполнение. Таких мест несколько. Если посмотреть на формат исполняемого файла, то можно найти несколько точек начала исполнения программы:

EntryPoint – точка входа, с которой начинается исполнение практически любой программы, ее адрес находится в заголовке файла.

ExportTable – Таблица экспортируемых функций в .DLL библиотеках. Адрес таблицы находится в заголовке файла, или в секции .idata.

TLS callbacks – таблица callback’ов Thread local storage. Они вызываются из системы при создании и удалении потоков(threads) в программе. Находятся в секции .tls ,

Также во многих исполняемых файлах есть секция .pdata. Она содержит данные, необходимые для обработки исключений в программе. Одним из полей этой таблицы

является адрес начала функции. С этого адреса также можно начать поиск линейных участков, но он не является точкой входа в прямом смысле.

Теперь можно начиная с этих адресов анализировать линейные участки. Если мы встречаем инструкцию, передающую управление, то нужно попытаться узнать, куда она может передать управление. Это однозначно можно сделать только в случае прямого перехода, то есть когда адрес или смещение указаны в инструкции явно.

Таким образом если в исполняемом коде не встречается косвенных или неявных переходов (за исключением импортируемых функций), то мы можем с уверенностью сказать, что мы построили CFG полностью.

Теперь поймем, что можно сделать, если такие переходы встречаются.

Для начала можно рассмотреть программы, которые были скомпилированы из языков высокого уровня и не подвергавшиеся обфускации. В книге [1] описан исчерпывающий анализ различных структур языков высокого уровня (в основном C++), в какой исполняемый код они преобразуются различными компиляторами, и способ их реверс-инжиниринга, то есть декомпиляции.

Из этой книги можно сделать вывод, что при компиляции косвенные переходы образуются в нескольких случаях:

1. вызов виртуальной функции класса
2. явное (или как результат оптимизации компилятора) составление таблицы или массива функций
3. передача в функцию Callback’а

1) При вызове виртуальных функций какого-нибудь класса, косвенный вызов необходим, так как мы знаем какую перегрузку виртуальной функции вызвать только в момент исполнения. Таким образом, чтобы вызвать виртуальную функцию надо взять указатель на таблицу виртуальных функций в экземпляре класса, загрузить его в регистр и сделать call по смещению, соответствующему нужной функции. Пример реализации таблицы виртуальных функций демонстрирует следующий листинг, где BASE_VTBL адрес таблицы виртуальных функций конкретного наследника класса:

```
        ; выделяем память для класса
    call      ??2@YAPAXI@Z  ; operator new(uint)
        ; загружаем в класс таблицу виртуальных функций, это происходит
        ; в конструкторе
    mov     dword ptr [eax], offset BASE_VTBL
        ; дальше происходит вызов функции
    mov     esi, eax        ; ESI = **BASE_VTBL
    mov     eax, [esi]      ; EAX = *BASE_VTBL == *BASE_FUNC1
    call    dword ptr [eax]       ; CALL BASE_FUNC1
```

Проблема статического обнаружения адресов виртуальных функций заключается в том, что адрес этой таблицы загружается в экземпляр класса в конструкторе. А конструктор наверняка находится в совершенно другой части программы. Таким образом требуется обнаружить конструктор, что является очень сложной задачей.

2) Таблица или массив функций. Косвенные переходы могут встречаться при явном задании таблицы функций в исходном коде, например.

```
// объявление
void (*functions ) (int,int) [4] = [&func1,&func2,&func3,&func4];
// вызов
Functions[1](2,3);
// будет вызвана функция
```

Такая таблица также может быть результатом оптимизаций оператора case. Эту ситуацию можно обнаружить статически, только в том случае, если этот массив является константным.

3) Callback’и. Так называется передача адреса исполняемого кода в качестве одного из параметров другому коду. Например, самый часто используемый – CreateThread. Ему в качестве параметра передается функция, которую вызовет система, создав новый поток. Есть множество других системных callback'ов, их

конечно можно статически определить анализируя таблицу импорта и прототип соответствующей функции. Но в программе или в несистемных библиотеках могут быть свои callback’и, и их все невозможно проанализировать.

Казалось бы все плохо, но можно заметить, что адреса функций в этих случаях когда-нибудь должны загружаться с помощью инструкций mov или lea, у которых второй операнд – адрес в памяти. Таким образом, если этот адрес находится в исполняемой секции файла, то это скорее всего адрес функции, а если в секции данных – то адрес указателя на функцию.

Этим способом также будет обработана и та часть переходов по регистру, где адрес перехода сначала загружается из константной памяти и не изменяется в дальнейшем. Данная проверка может существенно замедлить работу программы, но зато мы практически полностью решим проблему Callback’ов, и решим первые две проблемы в случае, если таблица функций задана константной, то есть находится в секции не доступной для записи.

Таким образом можно попробовать проанализировать исполняемые файлы на возможность построения полного графа потока управления. Также нужно собрать статистику о различных проблемных ситуациях возникающих в процессе построения CFG и другую информацию, такую как средняя длина инструкции или линейного участка.

Решение

Для этого была написана программа на языке C++ с использованием нескольких библиотек:

- BeaEngine – библиотека для дизассемблирования исполняемого кода[3].
- Boost::accumulators – для сбора статистики
- Boost::log - для логгирования

Эта программа была запущена на большой выборке файлов ОС windows(порядка 1500 файлов). В итоге было проанализировано 350 Мбайт исполняемого кода за 16 часов или 5 Кбайт в секунду.

Конечно в данной выборке есть значительное количество обфусцированных файлов, по этому не получится ограничиться только скомпилированными файлами. Но тем не менее все равно можно составить некоторую статистику по различным сложнообрабатываемым случаям.

В основной целью было определить количество файлов, в которых с уверенностью CFG был построен на 100%, то есть не встретилось неявных передач управления. Таких файлов оказалось 1-2 % , что кажется странным, если считать написанное выше верным. Но среди всех файлов было несколько простых тестовых файлов. Одним из них была программа “Hello World”, написанная на C++ и скомпилированная в Microsoft Visual C++ 2013 со стандартными настройками, но убранной оптимизацией, на которой программа показала, что CFG был построен с не 100% достоверностью. В ходе анализа построенного CFG, было выяснено, что в случае наличия floating point операций, если не выполнились некоторые проверки исполняемого образа, то должен вызываться некоторый callback до вызова main. Вызов производился инструкцией call rax. Конкретно в этой программе адрес этого callback’а отсутствовал, по этому он не мог вызваться. То есть CFG в этом случае был построен полностью, но этого нельзя гарантировать в общем случае.

Также было замечено, что значительная часть файла – это недостижимый код. Он берется из стандартной библиотеки C++, с которой файл слинкован, так как компоновщик в процессе своей работы присоединяет эту библиотеку целиком. То же самое обычно происходит и с другими библиотеками.

Таким образом скорее всего полностью разобрано достаточно большое количество файлов, но к сожалению очень часто мы не можем определить эти файлы. То есть полностью обойтись без информации об исходном коде или о процессе компиляции не представляется возможным.

Также были произведены замеры и проведено сравнение результатов в следующих трех вариантов программы :

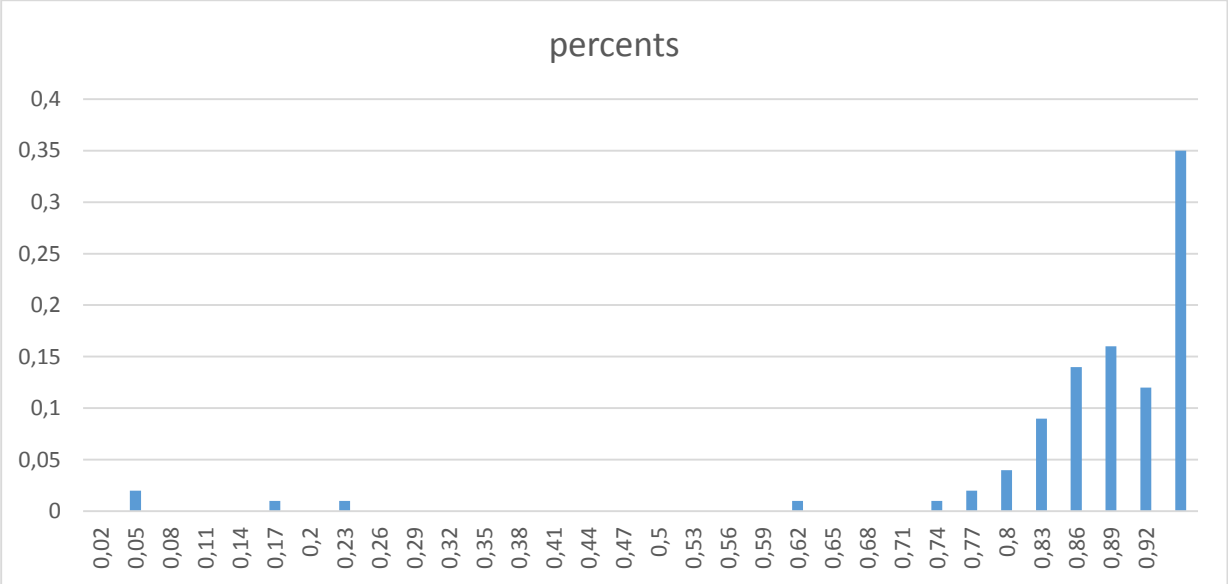
- с анализом всех операнда памяти всех и

Статистика

В ходе работы программы собиралась следующая статистика, также были посчитаны минимумы и максимумы, при убранных 2% пиковых данных, для того, чтобы убрать некоторые уникальные файлы. Для некоторых наборов данных была построена гистограмма.

- доля исполняемого кода разобранного на линейные участки

  1) включены все эвристики

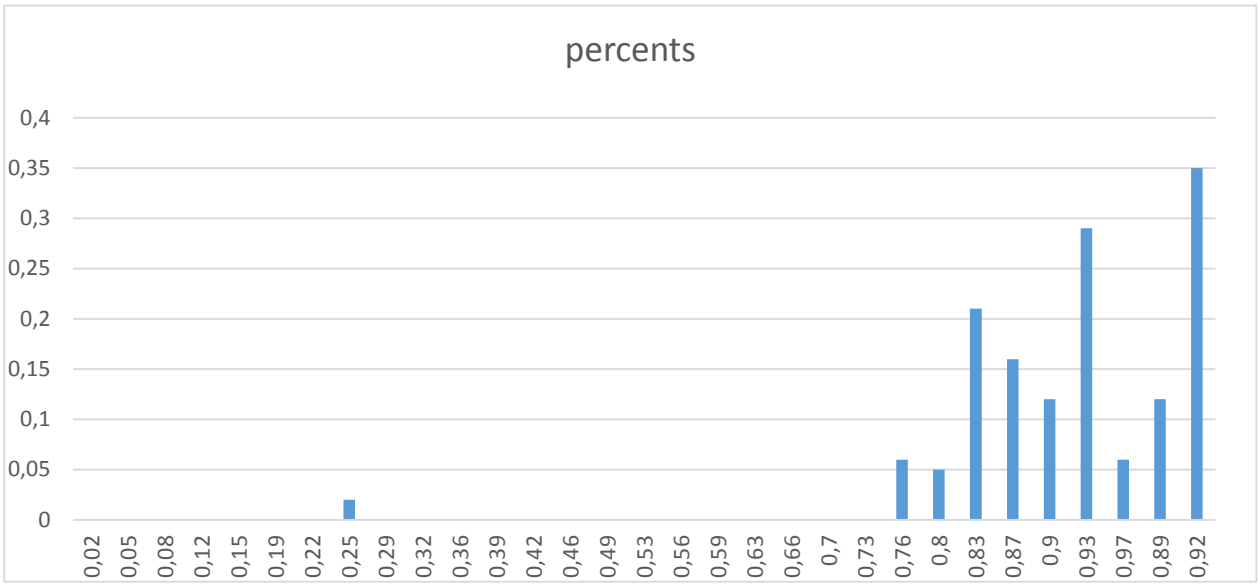


average : 0.87

2% minimum : 0.22

2% maximum : 0.95

2) используются только данные для обработки исключений

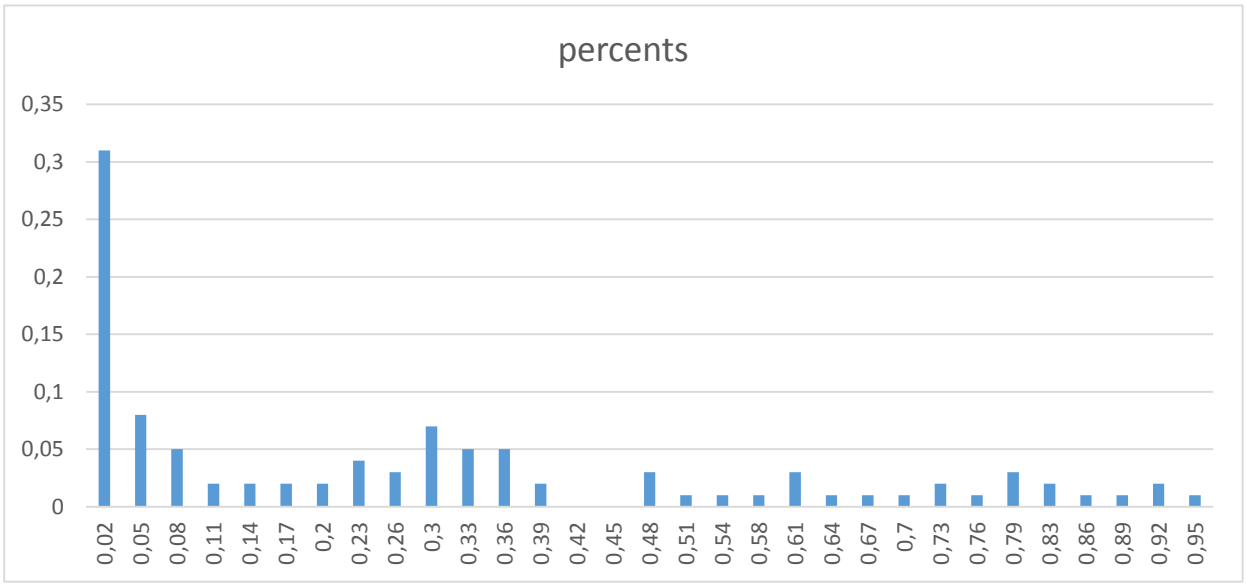


average : 0.828488

2% minimum : 0.700023

2% maximum : 0.924060

3) используется все кроме данных для обработки исключений



average : 0.36

2% minimum : 0.25

2% maximum : 0.78

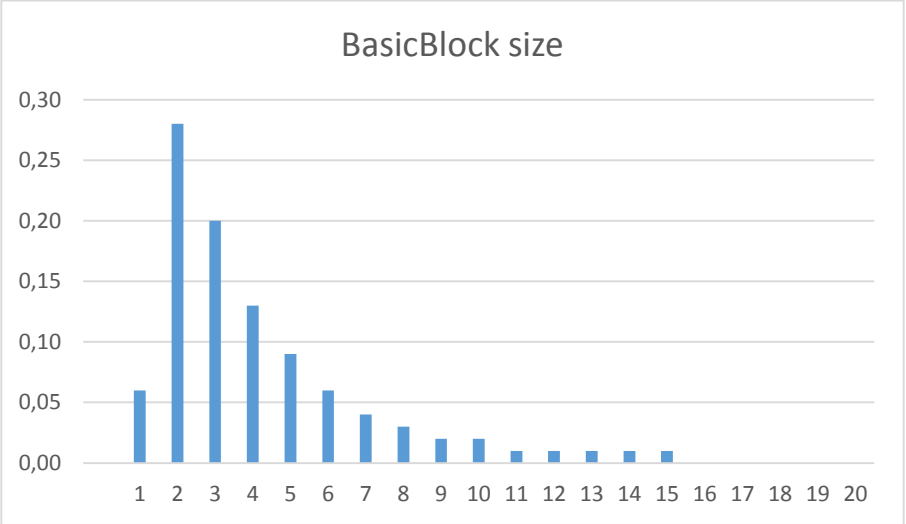
Из этого сравнения можно видеть, что информация об исключениях дает огромный прирост линейных участков, но в то же время часть этих участков является

недостижимым кодом, в чем можно убедиться на примере теста ‘Hello World’. Но в тоже время эта информация не является исчерпывающей, так как добавление анализа адресов инструкций добавляет значительные 5%.

Следующие результаты были получены в ходе запуска варианта с включенными всеми вариантами эвристик:

- средний размер линейного участка в инструкциях

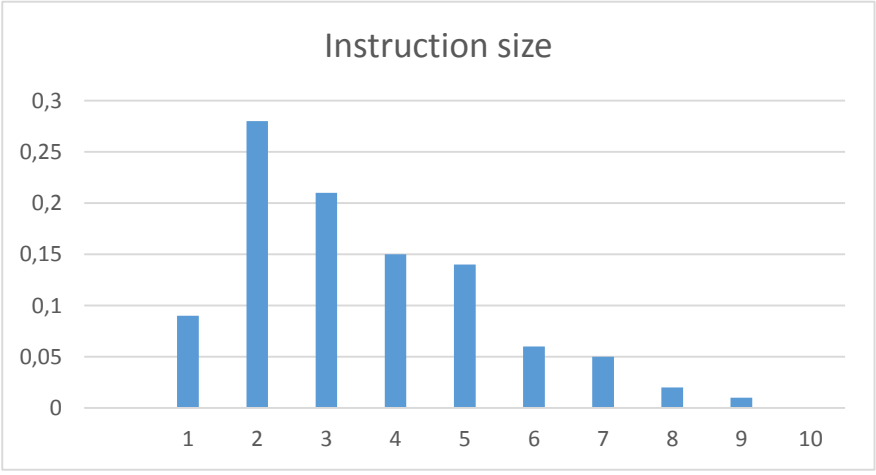
average : 4.294107
2% minimum : 1
2% maximum : 18



Отсюда можно видеть, что средняя длина линейного участка, полученная статически, меньше, чем широко известная длина 5-6 инструкций, полученная динамически. Эта разница возникает из-за того, что при статическом анализе, если мы попадаем в середину линейного участка, то мы его разбиваем на два навсегда. Тогда как при динамическом, если один участок передает управление в середину другого, то мы все равно их считаем двумя разными.

- средняя длинна инструкции

average : 3.908607
2% minimum : 1
2% maximum : 8



- доля различных видов передачи управления относительно всех инструкций, передающих управление.

Memory calls

average : 0.03

2% minimum : 0

2% maximum : 0.13

Register calls

average : 0.007

2% minimum : 0

2% maximum : 0.05

Вывод

Была написана программа для разбиения файлов на линейные участки, которую в последствии можно будет использовать при написании обфускатора. В итоге можно сказать, что задача построения CFG исключительно по исполняемому файлу в некоторых случаях разрешима, но мы не сможем достоверно узнать, что CFG построен полностью без дополнительной информации, коей может служить, например PDB файл.

Также были собраны различные статистические данные, которые позволяют судить о количестве неявных переходов в исполняемых файлах.

Используемая литература

[1] Искусство дизассемблирования, Крис Касперски, Ева Рокко, 2008

[2] Microsoft PE and COFF file specification, MSDN, msdn.microsoft.com/en-us/gg463119.aspx

[3] Disassembler library x86, x86-64,beaengine.org

[4] BOOST C++ libraries, boost.org