Динамический анализ обфусцированных приложений с диспетчеризацией или виртуализацией кода [1]

М.Г. Бакулин, С.С. Гайсарян, Ш.Ф. Курмангалеев, И.Н. Ледовских, В.А. Падарян, С.М. Щевьева
{bakulinm, ssg, kursh, il, vartan, shchsveta}@ispras.ru

Аннотация. В данной статье приводятся результаты экспериментального исследования по восстановлению графа потока управления, запутанного специализированным компилятором на основе LLVM. Средства запутывания и распутывания бинарного кода независимо разрабатывались двумя коллективами ИСП РАН. Помимо того, для количественной оценки стойкости запутывания были получены метрики сложности кода модельных примеров.

Ключевые слова. Динамический анализ, виртуальная машина, диспетчер, метрики кода.

1. Введение

В настоящее время, специалисты, занимающиеся обратной инженерией бинарного кода, регулярно сталкиваются со сложными системами защиты от анализа. Среди них все чаще используются методы запутывания бинарного кода, способные значительно затормозить процесс восстановления алгоритмов и форматов данных. Такой защитой оснащают не только вредоносное ПО, например, включенный в состав Visual Studio компонент Dotfuscator[2] позволяет рядовому разработчику защитить свои алгоритмы инструментом, интегрированным с базовыми средствами разработки.

Распространение средств запутывания кода делает все более важным техническое оснащение специалистов по обратной инженерии[1]. Ключевую роль в работе играют программные инструменты, способные качественно снизить сложность анализа, позволяющие его провести в разумные временные рамки.

[1] Работа поддержана грантом Президента Российской Федерации для государственной поддержки молодых российских ученых-кандидатов наук МК-1281.2012.9 и грантами РФФИ 12-01-31417 мол_а, 11-07-00450-а.

[2] http://msdn.microsoft.com/en-us/library/ms227240.aspx

Существует два основных подхода к анализу кода – статический и динамический. При динамическом подходе некоторые виды защиты (например, упаковка или шифрование, а также самомодификация кода) влияют на сложность анализа в меньшей степени, чем при статическом. С другой стороны, динамический анализ не может обеспечить полноту покрытия анализируемого приложения. В силу того, что у обоих подходов имеются свои преимущества и недостатки, часто применяется комбинированный метод анализа. В данной статье в качестве инструмента восстановления алгоритма рассматривается система TrEx [2] и метод её практического применения при анализе запутанного кода.

2. Система динамического анализа TrEx

В Институте системного программирования РАН разрабатывается система анализа бинарного кода TrEx. Основным источником данных выступает набор трасс, полученных с общей временной точки при выполнении исследуемой программой типичных сценариев работы. Под трассой понимается последовательность инструкций, выполненных процессором во время выполнения на нём интересующей аналитика программы, а так же значения всех регистров перед выполнением каждой инструкции. Трассы снимаются при помощи полносистемных эмуляторов, таких, например, как эмулятор QEMU[3].

Среда TrEx предоставляет различные возможности для упрощения анализа: разметку шагов трассы в соответствии с тем, какой процесс/нить выполняется на данном шаге, доступ к информации о системных вызовах и вызовах библиотечных функций, динамический прямой и обратный слайсинг трассы[4], построение различных графов анализируемой программы (зависимостей, вызовов, потока управления), восстановление формата данных[5, 6], а также многое другое.

В случае запутанного кода построение, например, графа потока управления может оказаться малополезным для распознавания алгоритма, поскольку различные преобразования запутывания управления не только увеличивают количество вершин и дуг графа, но и существенно меняют его общий вид. Наиболее показательные примеры таких преобразований – диспетчер и виртуализация кода. Для преодоления этих видов запутывания разработан компонент системы TrEx, способный восстанавливать граф потока управления кода, который был защищён наложением виртуальной машины. При таком методе защиты часть кода приложения транслируется в байт-код сгенерированной виртуальной машины, после чего защищаемый фрагмент заменяется интерпретатором этого байт-кода[7]. Многие интерпретаторы виртуальных машин работают по такой схеме: 1) считывание кода операции из памяти; 2) вычисление адреса обработчика (блока интерпретации) этого кода; 3) передача управления (диспетчеризация) и исполнение блока интерпретации данного кода операции; 4) переход к считыванию следующего

кода операции. Как правило, у такого эмулятора имеется счетчик команд виртуальной машины (VPC – Virtual Program Counter), который определяет, из какой области памяти необходимо считывать следующий код операции. Метод восстановления графа основывается на том, какие значения принимает этот счетчик команд виртуальной машины во время исполнения.

Если известны все шаги трассы, соответствующие считыванию кода операции из памяти и для каждого из них известно значение VPC, можно попытаться восстановить граф, используя следующий метод. Если инструкция – считывание кода операции, то в строящийся граф добавляется (если её в нем ещё нет) вершина с номером, равным значению счетчика команд, а также добавляется ребро $a \rightarrow b$, где a – значение счетчика команд при предыдущем считывании кода операции, а b – значение на текущем шаге. После этого граф требуется упростить, заменив длинные цепочки вершин на одну. Делается это так: если у вершины ровно один предшественник и один последователь, при этом у её предшественника только один последователь, вершина и инцидентные ей рёбра удаляются, и добавляется ребро из её предшественника в её последователя. Полученный граф выводится в формате .dot и может быть нарисован при помощи инструмента graphviz[8]

Предложенный способ хорошо себя показал при восстановлении графа потока управления тестовых приложений, защищенных демонстрационными версиями обфускаторов VMProtect[3], CodeVirtualizer[4], Safengine[5], и The Enigma Protector[6] [9].

3. Построение обфусцирующего компилятора на основе инфраструктуры LLVM

Для защиты бинарного кода от анализа используется множество различных методов, один из них – запутывающие преобразования. Такие преобразования обычно вносятся при помощи автоматических обфускаторов, которые принимают на вход исходный код программы или бинарный файл, а на выходе предоставляют исполняемый файл с запутанной программой.

Оптимальным выбором, позволяющим реализовать стойкие варианты запутывания программ, является создание обфусцирующего компилятора на базе одной из существующих компиляторных инфраструктур[10]. С одной стороны, это позволяет производить запутывание программы, имея полную информацию о ней на всех этапах компиляции, с другой – позволяет сосредоточиться на разработке алгоритмов защиты, а не на создании требуемой инфраструктуры. Преобразования могут производиться во время

[3] http://vmpsoft.com/products/vmprotect
[4] http://www.oreans.com/codevirtualizer.php
[5] http://enigmaprotector.com/ru/about.html
[6] http://safengine.com/en-us

обработки промежуточного представления компилятора на машинно-независимом уровне, что позволяет обеспечить поддержку нескольких архитектур

В Институте системного программирования был разработан обфусцирующий компилятор на основе инфраструктуры LLVM [11]. При разработке преобразований учитывались критерии эффективности:

- Запутывающее преобразование должно затрагивать и поток управления и поток данных запутываемой программы;
- Стойкость преобразования должна основываться на алгоритмически сложных задачах, например, требовать от атакующего применения анализа указателей для точного восстановления потоков данных защищенной программы;
- При разработке преобразования нужно учитывать особенности работы средств анализа, например, для автоматических декомпиляторов следует насытить граф потока управления несводимыми участками.

Разработанные методы усложнения программного кода:

- Преобразование, перемещающее локальные переменные в глобальную область видимости;
- Защита константных строк, используемых программой;
- Вставка в код фиктивных циклов;
- Приведение графа потока управления к плоскому виду с применением алгоритма диспетчеризации;
- Переплетение нескольких функций в одну с заменой всех вызовов отдельных функций на вызов одной общей;
- Сокрытие вызовов функций. Для защищаемой функции создается функция-переходник, внутри которой содержится несколько вызовов различных функций. Вызов нужной функции определяется с помощью предиката;
- Запутывание графа потока управления – создание несводимых участков в графе;
- Замена вызовов одной и той же функции на вызовы ее копий.

При помощи среды TrEx было проведено исследование разработанных методов защиты кода для оценки эффективности полученного обфусктора.

4. Восстановление графа потока управления

Анализ кода, полученного от запутывающего компилятора, проходил по схеме слепого эксперимента: аналитики использовали обфскатор как «черный ящик», не располагая информацией о том, какие запутывающие преобразования будут проделываться над кодом.

Для изучения особенностей реализации методов запутывания разработанным обфускатором был использовано один модельный алгоритм (Рис. 1), тремя различными способами переведенный в форму исполняемого кода.

```
01 #include <stdio.h>
02 #include <stdlib.h>
03 #define n 10
04 void bubbleSort(int numbers[], int array_size) {
05       int i, j, temp;
06       for (i = 0; i < array_size; i++) {
07             for (j = 0; j < array_size - 1; j++) {
08                   if (numbers[j] > numbers[j + 1])
09 {
10                         temp = numbers[j + 1];
11                         numbers[j + 1] =
12 numbers[j];
13                         numbers[j] = temp;
14                   }
15             }
16       }
17 }
18 int main() {
19       int a[n];
20       printf("Start\n");
21       for (int i = 0; i < n; i++) {
22             a[i] = rand()*rand();
23             printf("%d ", a[i]);
24       }
25       printf("\n");
26       bubbleSort(a, n);
27       for (int i = 0; i < n; i++) {
28             printf("%d ", a[i]);
29       }
30       printf("Finish\n");
         return 0
}
```

Рис. 1. Листинг программы сортировки пузырьком.

Первый тестовый пример был получен прототипной версией запутывающего компилятора, второй – с помощью его же итоговой версии. Третий пример был получен итоговой версией компилятора, но на модифицированной версии кода. Изменения никак не меняли сам алгоритм и его граф потока управления в частности, но препятствовали применению некоторых

общеупотребительных оптимизаций. Для всех случаев программа компилировалась с параметрами, включающими на максимальную защиту.

Полученный исполняемый файл первого примера был запущен при помощи эмулятора QEMU, в результате была снята трасса программы. В процессе трассировки программа вывела на печать следующую информацию:

```
Start
1323961443 1092212534 1392800464 -818133696 1041837016 -
628591679 -1034175621 609956508 1082926972 -1134093940
-1134093940 -1034175621 -818133696 -628591679 609956508
1041837016 1082926972 1092212534 1323961443 1392800464
Finish
```

Первым шагом при исследовании трассы обычно является нахождение инструкций «маяков», которые относятся к изучаемому приложению[12]. Для этого существует несколько подходов. В данном случае был использован поиск в трассе определенного состояния процессора: один из регистров общего назначения содержит числа, выводящиеся на экран приложением. Был найден первый шаг трассы, на котором значение регистра стало равно 1041837016 (одно из чисел, выведенных на печать). Номер этого шага равен `0x98FA1C`[7], номер процесса, соответствующего этому шагу, равен `0x18A8000`, адрес инструкции `0x413B2C`. Также в непосредственной близости (на шаге `0x98F8F4`) был обнаружен вызов функции `rand`, которая, как известно из исходного кода, используется для вычисления значений элементов массива в процессе инициализации. На основании полученной информации был сделан вывод, что данный шаг и процесс соответствует коду исследуемого приложения. После этого была выделена подтрасса, в которой содержатся только те шаги исходной трассы, которые относятся к тому же процессу и тому же адресному пространству, что и шаг `0x98FA1C`. В такой трассе содержатся только те инструкции, которые относятся либо непосредственно к исполняемому файлу изучаемого приложения, либо к различным библиотечным функциям (таким, как `printf`, `rand`), которые были вызваны из исследуемого приложения.

Для дальнейшего уточнения положения функции сортировки пузырьком в трассе были использованы вызовы функции печати на экран. В случае с данным приложением для печати использовалась функция `printf` из библиотеки cygwin1.dll. Адрес начала этой функции равен `0x610C4ADF`. Согласно графу вызовов, в трассе присутствует 23 вызова этой функции,

[7] Для нахождения такого шага использовался модуль поиска состояния, встроенный в систему TrEx, который находит первый шаг, для которого выполняется заданное условие. В данном случае условие было `EAX == 1041837016 || EBX == 1041837016 || ECX == 1041837016 || EDX == 1041837016.`

поэтому можно предположить, что первый вызов относится к печати сообщения “Start” (строка 18 в исходном коде), следующие 10 вызовов соответствуют печати чисел в неотсортированном массиве (строка 21), следующий вызов соответствует печати символа перевода строки (строка 23), ещё 10 вызовов соответствуют печати отсортированного массива (строка 26) и, наконец, последний вызов осуществляет печать сообщения “Finish” (строка 28). Поэтому была создана ещё одна подтрасса, в которую вошли только шаги от двенадцатого до тринадцатого вызова функции `printf`. В полученной трассе содержится примерно 250 тысяч шагов, в то время как в исходной трассе их больше 20 миллионов.

Обычно, аналитик не располагает исходным кодом программы, поэтому для нахождения необходимого вызова системной функции может быть использован и другой способ. Для каждого вызова функции, например, `printf`, восстанавливаются параметры, с которым она была вызвана. Известно, что функция `printf` принимает несколько параметров, и все параметры лежат на стеке. Если был произведён вызов функции `printf(“%d”, a[i])`, то на шаге, соответствующем этому вызову, по адресу `ESP + 4` в памяти будет находиться адрес форматной строки, а по адресу `ESP + 8` будет находиться значение переменной `a[i]`, которое выводится на печать. Пусть необходимо восстановить фактические параметры вызова функции, произошедшего в основной трассе на шаге с номером `0xAA632C` (этот шаг соответствует одиннадцатому вызову `printf` в трассе). Для этого используется имеющийся в системе TrEx компонент Buffer Restore, который на основе обращений к участку памяти может установить его значение. Простейший пример такого обращения: если после инструкции `MOV EAX, DWORD PTR[0x226B2C]` значение регистра `EAX` стало равно `0x42735D`, то, очевидно, по адресу `0x226B2C` в памяти находится число `0x42735D`. Алгоритм восстановления значения, хранящегося в памяти по адресу `ESP + 4`, вернул результат `0x499007`. После этого был запущен алгоритм восстановления буфера по адресу `0x499007`, была получена строка `“%d ”`, то есть данный вызов относится к печати числа. Само число, которое выводится на печать, расположено в памяти по адресу `ESP + 8`. При помощи алгоритма восстановления буфера было установлено, что по этому адресу в памяти находится число `0xBC67198C` (или -1134093940 в десятичной системе). Таким образом, одиннадцатый вызов функции `printf` в трассе соответствует печати числа -1134093940. Аналогичным образом можно восстановить информацию и о других вызовах функции `printf` в трассе.

В результате фильтрации была получена подтрасса, в которой содержится в основном только код функции, которую необходимо проанализировать. После этого был запущен поиск шага выделенной подтрассы, на котором какой-либо из регистров общего назначения принял значение, равное одному из выведенных на печать чисел. Значение 1092212534 встретилось на одном из

регистров на шаге `0xF0D4`. В непосредственной близости на шаге `0xF0D1` находится инструкция перехода по динамически вычисляемому адресу `JMP ECX`. Такие инструкции – характерная черта диспетчеров и эмуляторов виртуальных машин. В программах, запутанных другими виртуализаторами, возможно использование и других подобных инструкций, например `JMP DWORD PTR [EAX*4 + EDI]`, или последовательность инструкций `PUSH ECX, RET` и тому подобные. После нахождения такой инструкции перехода необходимо определить, как был вычислен адрес перехода и определить, что является счётчиком команд виртуальной машины. Участок трассы, соответствующий вычислению адреса перехода, приведён на рисунке 2.

```
00404846    MOV   DWORD PTR SS:[EBP - 0000183Ch], ESI
0040484C    JA    0040480Ch
0040484E    MOV   EAX, DWORD PTR SS:[EBP - 0000183Ch]
00404854    MOV   ECX, DWORD PTR [EAX * 4 + 00499140h]
0040485B    JMP   ECX
```

Рис. 2. Участок трассы первого примера. Вычисление адреса перехода.

Можно видеть, что адрес перехода равен значению ячейки памяти по адресу `ESI*4 + 0x499140`. Таким образом, можно предположить, что счетчиком команд виртуальной машины служит регистр `ESI`. Для проверки этого предположения был запущен алгоритм восстановления графа потока управления виртуализованного кода. На вход алгоритм принимает условие, которое соответствует считыванию кода операции из памяти (в данном случае, `EIP == 0x404854`[8]) и выражение, которое соответствует значению счётчика команд виртуальной машины при данном считывании (`ESI*4 + 0x499140`). В результате получен граф, изображённый на рисунке 3.

[8] В некоторых случаях требуется чуть более сложное условие. Например, для программ, защищенных при помощи Safengine, условие имеет вид `EIP == a && EBP == b`, где `a` и `b` – константы, определяемые вручную по трассе аналогично тому, как был найден адрес `0x404854`.

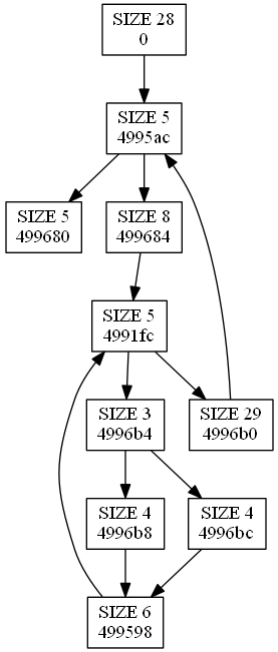

Рис. 3. Граф потока управления, полученный по трассе первого приложения.

Каждой вершине в графе соответствует два числа: размер и адрес. При построении графа длинные цепочки вершин заменялись на одну вершину. Наличие таких цепочек позволяет заключить о разбиении исходных базовых блоков на более мелкие при применении запутывания «диспетчер». Первое число соответствует количеству вершин в цепочке, которую замещает данная вершина, а второе соответствует адресу, из которого считывается адрес первого перехода. Для первого базового блока адрес считается равным нулю. На этом рисунке отчетливо видны два цикла: внешний и внутренний, во внутреннем цикле существует какое-то условие, в зависимости от которого выполняется один или другой базовый блок. Именно такой граф потока управления характерен для сортировки пузырьком, что позволяет предположить об успешности восстановления. Благодаря информации, вынесенной в вершины графа, можно приближённо разметить трассу в соответствии с тем, к какому базовому блоку относится тот или иной участок трассы. В случае данного исследуемого приложения для этого необходимо создать текстовые комментарии к позициям трассы, для которых выполняется условие `EIP == 0x404854 && ESI*4 + 0x499140 == A`, где `A` – адрес, соответствующий какому-либо базовому блоку восстановленного графа. Тогда участок трассы между двумя комментариями будет соответствовать одному базовому блоку. При этом, конечно, на этом участке

будут встречаться не только инструкции, относящиеся к алгоритму, который был защищён запутыванием, но также и управляющие инструкции виртуальной машины или диспетчера.

Во втором примере исходный Си-код был защищен итоговой версией обфускатора. Для нахождения участка трассы, содержащего код сортировки, был использован тот же метод, что и в предыдущем случае. Полученная подтрасса содержит примерно 140 тысяч шагов. Точно так же в подтрассе был найден шаг, на котором один из регистров содержит значение 1092212534, и рядом с этим шагом был найден переход по динамически вычисляемому адресу. Участок трассы, на котором происходит вычисление адреса перехода, изображён на рисунке 4.

```
0041C95C    MOV   EAX, DWORD PTR SS:[EBP -
00002B10h]
0041C962    XOR   EAX, DWORD PTR SS:[EBP -
00002B20h]
0041C968    MOV   EDX, DWORD PTR SS:[EBP -
00002B18h]
0041C96E    XOR   EAX, EDX
0041C970    MOV   ECX, DWORD PTR SS:[EBP -
00002B1Ch]
0041C976    XOR   ECX, EDX
0041C978    XOR   EDX, DWORD PTR SS:[EBP -
00002B14h]
0041C97E    MOV   DWORD PTR SS:[EBP - 00002B18h],
EDX
0041C984    ADD   ECX, EAX
0041C986    MOV   DWORD PTR SS:[EBP - 00002B0Ch],
ECX
0041C98C    CMP   ECX, 000003F1h
0041C992    JA    0041C95Ch
0041C994    JMP   DWORD PTR [ECX * 4 + 005CAF5Ch]
```

Рис. 4. Участок трассы второго приложения. Вычисление адреса перехода.

Видно, что отсутствует четко выраженная конструкция, реализующая виртуальный счётчик команд. Была произведена попытка автоматически восстановить граф со следующими параметрами: адрес инструкции считывания кода операции `0x41C994`, регистр со значением счётчика команд – `ECX`. В результате был получен граф, изображённый на рисунке 5.

Рис. 5. Граф потока управления, полученный по трассе второго примера

Такой граф явно не соответствует ожидаемому результату. Были проведены дополнительные исследования, с целью выяснить, что же в действительности используется в качестве счётчика команд виртуальной машины в данном случае. Однако, несмотря на достаточно долгое изучение, выявить дополнительный код, реализующий виртуальный счетчик команд, никак не удавалось. Тогда было сделано предположение, что оптимизирующий компилятор обнаружил в коде константу, которая задавала размер массива, выполнил распространение констант по коду и развернул внутренний цикл сортировки[9]. Действительно, можно видеть, что в графе во внутреннем цикле имеется 9 ветвлений. Для проверки предположения, что некорректный граф получается из-за оптимизаций компилятора, исходный код был изменён таким образом, чтобы размер массива задавался пользователем с клавиатуры. Полученный код был скомпилирован итоговой версией запутывающего компилятора, после чего снята трасса третьего примера. В результате выделения участка трассы, соответствующего функции сортировки, была получена подтрасса размером примерно 217 тысяч шагов.

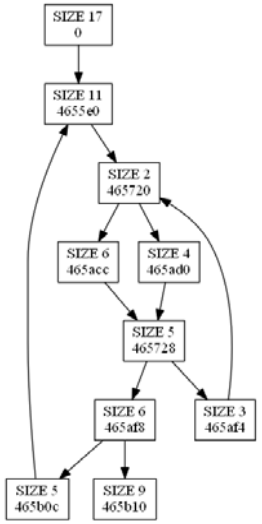

Рис. 6. Граф потока управления, полученный по трассе третьего примера

[9] Уже после снятия трассы третьего примера был скомпилирован код исходной программы без применения запутывающих преобразований, и в графе потока управления полученной функции точно так же присутствовали 9 ветвлений. Отсутствие развёрнутого цикла в графе первого примера связано, по-видимому, с другими настройками компиляции в прототипной версии запутывающего компилятора.

Код вычисления адреса перехода в третьем приложении получился такой же, как во второй версии с точностью до констант. Был применён алгоритм восстановления графа потока управления, результат изображён на рисунке 5. Как можно видеть, граф восстановлен достаточно точно: видно два цикла (внешний и внутренний), и во внутреннем цикле исполняется один из двух базовых блоков в зависимости от выполнения или не выполнения условия, но при этом цикл с предусловием был, по-видимому, заменён на цикл с постусловием.

5. Оценка сложности сгенерированного кода на основе метрик

Вычисление метрик программного кода позволяет не только оценивать различные свойства программ, прогнозировать объем работ, характеризовать проектные решения, сложность и надежность программного обеспечения в процессе разработки и сопровождения. Использование метрик целесообразно и при решении задач анализа приложений – в частности, применительно к бинарному коду. В процессе исследования бинарного кода требуется уметь классифицировать приложения по сложности, оценивать трудоемкость анализа как всей программы, так и ее отдельных частей (модулей, функций), а также находить в коде участки, представляющие первоочередной интерес для аналитика и потому, как правило, защищенные различными методами запутывания. Эти задачи предъявляют несколько иные требования к выбору метрик кода, чем процесс разработки и модификации приложений. В настоящее время в рамках среды TrEx создаётся компонент для вычисления различных метрических характеристик анализируемого кода. В ходе выполнения описываемой работы были выполнены первые эксперименты по получению и использованию метрик сложности бинарного кода в среде TrEx.

Выделяют несколько групп метрик сложности кода: количественные метрики, метрики сложности потока управления, метрики сложности данных, комбинированные, гибридные и объектно-ориентированные метрики. В анализе двоичного кода можно использовать лишь часть известных метрик из числа количественных, метрик сложности потоков управления и данных, а также получаемых на их основе гибридных мер. Количественные метрики, изначально используемые для оценки трудозатрат по проектам, основаны на подсчёте различных конструкций исходного или бинарного кода. Они легко могут быть адаптированы для анализа трасс программ. Количественные характеристики программ обычно рассматриваются в первую очередь, ввиду простоты их получения – для вычисления этих метрик не требуется предварительный анализ кода или трасс, построение графов потока управления, потока данных, вызовов и т.п.

В первых экспериментах использовались количественные метрики, позволяющие оценить насыщенность программы управляющими конструкциями – метрика Джилба и BCD-метрика. Полученные оценки

корректировались посредством аналога метрики LOC (Lines Of Code) – для запутанного бинарного кода в качестве такого аналога LOC был взят размер исполняемого файла. Метрики Джилба [13] показывают сложность программного обеспечения на основе насыщенности программы условными операторами или операторами цикла. Вычисляются два значения: абсолютная сложность CL (количество управляющих операторов) и относительная сложность по Джилбу cl=CL/n, где n – общее число операторов программы. BCD-метрика является модификацией ABC-метрики [14], которая основана на подсчете присваиваний (**A**ssignment), передач управления за пределы области видимости, т.е. вызовов функций (**B**ranch), и логических проверок (**C**ondition). Мера записывается тройкой значений, например, ABC = <7, 4, 2>, но для оценки сложности программы вычисляется одно число, как квадратный корень из суммы квадратов A, B, C. Модификация этой метрики, названная BCD (**B**ranch and **C**all **D**ensity) сводилась к вычислению лишь двух значений – B (количество инструкций условных и безусловных переходов) и C (количество инструкций вызова). Затем вычисляются плотности этих инструкций BD=B/n и CD=C/n. Результирующее значение для характеристики сложности программы получается как квадратный корень из суммы квадратов BD и CD.

По трассам в среде TrEx были получены значения метрик для незащищенной программы сортировки пузырьком и трёх вариантов обфускации на основе инфраструктуры LLVM. Результаты сведены в таблицу 1; для сравнения приводятся также характеристики этого же приложения, защищенного демо-версией обфускатора Safengine.

Можно видеть, что относительная сложность анализируемых тестовых примеров как по Джилбу, так и на основе BCD-метрики, после наложения запутывающих преобразований не только не увеличилась, но даже стала заметно меньше (за исключением значения BCD-меры в случае Safengine). Данный факт объясняется тем, что все примененные обфускаторы применяют к программе ряд запутывающих преобразований помимо запутывания потока управления посредством диспетчера. В частности, добавляется значительное количество непрозрачных предикатов, недостижимого и мёртвого кода; применяются также различные виды запутывания данных. Эти преобразования увеличивают количество инструкций в коде программы, что приводит к уменьшению относительных оценок сложности потока управления на основе количественных метрик.

Таблица 1. Относительная сложность кода

Программа	**cl**	**BCD**
Без защиты	0,213607	0,190239
Защищена Safengine	0,206598	0,193034

Пример 1	0,141131	0,131036
Пример 2	0,18671	0,176082
Пример 3	0,171154	0,159457

Для получения адекватной оценки сложности восстановления потока управления было принято решение использовать комбинированные количественные метрики, представляющие собой произведение размера исполняемого файла в килобайтах (аналог метрики LOC для бинарного кода) на значения метрик Джилба и BCD.

Такая оценка, как можно видеть из таблицы 2, более точно соответствует реальным затратам времени на анализ и восстановление потока управления использованного нами запутанной программы. Особенно наглядно это проявляется для оценок примеров № 1 и №2; пример №3, для которого исходный текст программы был изменен с целью исключения ряда оптимизирующих преобразований, закономерно получил более низкую оценку сложности, несмотря на то, что значения относительных мер для него ближе к их значениям для примера №2. Код с защитой Safengine имеет меньшие комбинированные оценки сложности при больших относительных; это объясняется тем, что демонстрационная версия данного протектора имеет ряд ограничений, в частности, по запутыванию потока данных. В целом, результаты данного эксперимента свидетельствуют о возможности предварительной оценки сложности анализа защищенного кода посредством простых количественных метрик и их комбинаций.

Таблица 2. Сложность кода с поправкой на размер

Программа	**Размер, Кб**	**LOC*cl**	**LOC*BCD**
Без защиты	134	28.6233	25.492
Защищена Safengine	194	40.08	37.4486
Пример 1	630	88.9125	82.55
Пример 2	1893	353.442	305.6
Пример 3	414	70.858	66.015

6. Заключение

Предложенный метод восстановления графа потока управления виртуализованного кода показал свою эффективность, однако имеет и недостатки. Один из них напрямую связан с природой динамического анализа: если при запуске программы одна из веток условного перехода ни разу не

выполнялась, то в восстановленном графе не будет соответствующего ветвления. Чтобы нивелировать этот недостаток предполагается дополнить метод следующим образом: если какой-либо базовый блок имеет больше одного последователя, то код операции (не счётчик команд!), соответствующий последнему исполненному переходу по вычислимому адресу в этом базовом блоке, помечается как код условного перехода, и все другие блоки, в которых был считан такой же код, тоже будут помечаться как блоки, имеющие возможный условный переход. Однако для определения цели этого невыполнившегося перехода будет необходимо полностью восстановить алгоритм работы данной виртуальной машины.

Следующий недостаток связан с тем, что для корректного восстановления графа необходимо вручную найти нужные инструкции перехода и счётчик команд виртуальной машины. В трассе может быть несколько таких инструкций с различными адресами. Например, у каждой функции приложения может быть свой собственный интерпретатор команд виртуальной машины, который использует свой собственный счётчик команд, и если неправильно определить, какая из этих инструкций перехода по вычисляемому адресу относится к изучаемой функции, можно получить некорректный граф. Возможна и другая ситуация: для передачи управления во всех функциях используется одна общая инструкция перехода, и в результате будет получен граф, представляющий смесь графа вызовов и потока управления. Дополнительную сложность при этом создаёт то, что для определения ошибочности восстановленного графа аналитик может ориентироваться только на своё чутьё и ожидания того, какой граф должен быть получен. В будущем планируется добавить модуль, собирающий статистику обо всех переходах по вычисляемым адресам и предоставляющий её пользователю для облегчения поиска нужного перехода.

Отдельно стоит заметить, что можно восстановить только граф потока управления той программы, которая была подана на вход запутывателю. Таким образом, если были проведены предварительные дополнительные запутывания потока управления (независимо от того, намеренные это усложнения или просто оптимизирующие преобразования компилятора), получить исходный граф представляется невозможным. Для борьбы с некоторыми из запутываний (в частности, с разворачиванием циклов и клонированием кода) необходимо уметь составлять модели поведения полученных базовых блоков и сравнивать полученные модели между собой, но это достаточно сложная задача для автоматического анализа.

Не стоит забывать, что восстановленный граф потока управления может только помочь аналитику в восстановлении алгоритма исследуемой программы. Аналитику требуется ещё установить, что делает каждый из базовых блоков, а в случае с кодом, запутанным посредством комбинации различных методов, это может оказаться достаточно сложной задачей. Исследование этих задач и разработка соответствующего инструментария

является направлением дальнейшей работы в рамках развития среды динамического анализа TrEx.

Список литературы

[1] А.Ю.Тихонов, А.И. Аветисян. Развитие taint-анализа для решения задачи поиска программных закладок // Труды Института Системного Программирования. — 2011. — Т. 20. — С. 9–24.

[2] Падарян В. А., Гетьман А. И., Соловьёв М. А. Программная среда для динамического анализа бинарного кода // Труды Института Системного Программирования. — 2009. — Т. 16. — С. 51–72.

[3] К. Батузов, П. Довгалюк, В. Кошелев, В. Падарян. Два способа организации механизма полносистемного детерминированного воспроизведения в симуляторе QEMU // Труды Института Системного Программирования. — 2012. — Т. 22. — С. 77–94. DOI: 10.15514/ISPRAS-2012-22-6.

[4] Korel B., Laski J. Dynamic program slicing // Information Processing Letters, Vol. 29, Issue 3. — 1988. — P. 155–163.

[5] А.И. Гетьман, Ю.В. Маркин, В.А. Падарян, Е.И. Щетинин. Восстановление формата данных // Труды Института Системного Программирования. — 2010. — Т. 19. — С. 195–214.

[6] А.И. Аветисян, А.И. Гетьман. Восстановление структуры бинарных данных по трассам программ // Труды Института Системного Программирования. — 2012. — Т. 22. — С. 95–118. DOI: 10.15514/ISPRAS-2012-22-7.

[7] Monirul Sharif, Andrea Lanzi, Jonathon Giffin, Wenke Lee. Automatic reverse engineering of malware emulators // Proceedings of the 2009 30th IEEE Symposium on Security and Privacy. — SP ’09. — Washington, DC, USA : IEEE Computer Society, 2009. — P. 94–109.

[8] J. Ellson, E.R. Gansner, E. Koutsofios et al. Graphviz and dynagraph – static and dynamic graph drawing tools // Graph Drawing Software / Ed. by M. Junger, P. Mutzel. — Berlin/Heidelberg : Springer-Verlag, 2004. — Mathematics and Visualization. — P. 127–148.

[9] И.Н.Ледовских, М.Г.Бакулин. Подход к восстановлению потока управления запутанной программы // Труды Института Системного Программирования. — 2012. — Т. 22. — С. 155–167. DOI: 10.15514/ISPRAS-2012-22-10.

[10] Курмангалеев Ш.Ф., Корчагин В.П., Матевосян Р.А. Описание подхода к разработке обфусцирующего компилятора // Труды Института Системного Программирования. — 2012. — Т. 23. 10 страниц. Принято к печати. DOI: 10.15514/ISPRAS-2012-23-4.

[11] Курмангалеев Ш.Ф., Корчагин В.П., Савченко В.В., Саргсян С.С. Построение обфусцирующего компилятора на основе инфраструктуры LLVM // Труды Института Системного Программирования. — 2012. — Т. 23. 15 страниц. Принято к печати. DOI: 10.15514/ISPRAS-2012-23-5.

[12] Тихонов А. Ю., Аветисян А. И., Падарян В. А. Методика извлечения алгоритма из бинарного кода на основе динамического анализа // Проблемы информационной безопасности. Компьютерные системы. — 2008. — Т. 3. — С. 66–71.

[13] Милютин А., Метрики кода программного обеспечения http://www.viva64.com/ru/a/0045/ . Дата публикации: 20.07.2009.

[14] Hassan Raza Bhatti, Automatic Measurement of Source Code Complexity // Master’s Thesis, Lulea University of Technology, Lulea, Sweden, 2011

Dynamic analysis of virtualization- and dispatching-obfuscated applications

M.G. Bakulin, S.S. Gaissaryan, Sh.F. Kurmangaleev,
I.N. Ledovskikh, V.A. Padaryan, S.M. Shchevyeva
ISP RAS, Moscow, Russia
{bakulinm, ssg, kursh, il, vartan, shchsveta}@ispras.ru

Abstract. Obfuscation algorithms are now widely used to prevent software reverse engineering. Binary code virtualization is one of the most powerful obfuscations technics. Another obfuscation method known as “dispatching” can be used to transform application control flow similarly to virtual machine insertion. Our research was aimed at reconstruction of control flow graph in case of both code virtualization and dispatching. To achieve this goal, we implemented de-obfuscation tool which keeps track of virtual program counter used by virtual machine emulator and reconstructs the application control flow. This paper describes experimental results of test application de-obfuscation via dynamic analysis. Both obfuscating and de-obfuscating tools were independently developed by two different teams of ISP RAS – the LLVM-based obfuscating compiler and the software environment for dynamic analysis of binary code. The paper briefly introduces both software tools and then describes results of experimental research on recovering of control flow graph of obfuscated application. Application was initially protected by specialized obfuscating LLVM-based compiler. Next, TrEx environment was used to analyze program execution trace, to find the dispatcher-protected part of application and to recover its control flow. Additionally, some software code complexity metrics for test applications were calculated to estimate obfuscation resilience provided by different versions of obfuscating compiler.

Keywords: Dynamic analysis, virtual machine, dispatcher, code metrics

References

[1]. Tikhonov A.YU., Avetisyan A.I. Razvitie taint-analiza dlya resheniya zadachi poiska programmnykh zakladok [Development of taint-analysis methods to solve the problem of searching of undeclared fatures]. Trudy ISP RAN [The Proceedings of ISP RAS], 2011, vol. 20, pp. 9–24 (in Russian).

[2]. Padaryan V. A., Get'man A. I., Solov'yov M. A. Programmnaya sreda dlya dinamicheskogo analiza binarnogo koda [Software environment for dynamic analysis of binary code]. Trudy ISP RAN [The Proceedings of ISP RAS], 2009, vol. 16, pp. 51–72 (in Russian).

[3]. Batuzov K., Dovgalyuk P., Koshelev V., Padaryan V. Dva sposoba organizatsii mekhanizma polnosistemnogo determinirovannogo vosproizvedeniya v simulyatore QEMU [Two approaches to full-system deterministic replay in QEMU]. Trudy ISP RAN [The Proceedings of ISP RAS], 2012, vol. 22, pp. 77–94 (in Russian). DOI: 10.15514/ISPRAS-2012-22-6.

[4]. Korel B., Laski J. Dynamic program slicing. Information Processing Letters, 1988, vol. 29, Issue 3, pp. 155–163. doi: 10.1016/0020-0190(88)90054-3

[5]. Get'man A.I., Markin YU.V., Padaryan V.A., SHHetinin E.I. Vosstanovlenie formata dannykh [Format recovery] Trudy ISP RAN [The Proceedings of ISP RAS], 2010, vol. 19, pp. 195–214 (in Russian).

[6]. Avetisyan A.I., Get'man A.I. Vosstanovlenie struktury binarnykh dannykh po trassam program [Recovery of binary data structures from program traces] Trudy ISP RAN [The Proceedings of ISP RAS], 2012, vol. 22, pp. 95–118 (in Russian). DOI: 10.15514/ISPRAS-2012-22-7.

[7]. Sharif M., Lanzi A., Giffin J., Lee W. Automatic reverse engineering of malware emulators Proceedings of the 2009 30th IEEE Symposium on Security and Privacy — SP ’09. — Washington, DC, USA : IEEE Computer Society, 2009, pp. 94–109. doi: 10.1109/SP.2009.27

[8]. Ellson J., Gansner E.R., Koutsofios E., et al. Graphviz and dynagraph – static and dynamic graph drawing tools. Graph Drawing Software (ed. by M. Junger, P. Mutzel), Berlin/Heidelberg: Springer-Verlag, 2004. — Mathematics and Visualization. pp. 127–148. doi: 10.1007/978-3-642-18638-7_6

[9]. Ledovskikh I.N., Bakulin M.G.. Podkhod k vosstanovleniyu potoka upravleniya zaputannoj programmy [An approach to reconstruction of control flow of an obfuscated program] Trudy ISP RAN [The Proceedings of ISP RAS], 2012, vol. 22, pp. 153–168 (in Russian). DOI: 10.15514/ISPRAS-2012-22-10.

[10]. Kurmangaleev SH.F., Korchagin V.P., Matevosyan R.A. Opisanie podkhoda k razrabotke obfustsiruyushhego kompilyatora [Description of the Approach to Development of the Obfuscating Compiler] Trudy ISP RAN [The Proceedings of ISP RAS], 2012, vol. 23, pp. 67–76 (in Russian). DOI: 10.15514/ISPRAS-2012-23-4.

[11]. Kurmangaleev SH.F., Korchagin V.P., Savchenko V.V., Sargsyan S.S. Postroenie obfustsiruyushhego kompilyatora na osnove infrastruktury LLVM [Building Obfuscation Compiler Based on LLVM Infrastructure] Trudy ISP RAN [The Proceedings of ISP RAS], 2012, vol. 23, pp. 77–92 (in Russian). DOI: 10.15514/ISPRAS-2012-23-5.

[12]. Tikhonov A. YU., Avetisyan A. I., Padaryan V. A. Metodika izvlecheniya algoritma iz binarnogo koda na osnove dinamicheskogo analiza [Methodology of Exploring of an Algorithm from Binary Code by Dynamic Analysis] Problemy informatsionnoj bezopasnosti. Komp'yuternye sistemy [Problems of Computer Security. Computer Systems], 2008, vol. 3, pp. 66–71 (in Russian).

[13]. Milyutin A., Metriki koda programmnogo obespecheniya [Software code metrics] http://www.viva64.com/ru/a/0045/ [http://www.viva64.com/en/a/0045/], 20.07.2009.

[14]. Hassan R. B. Automatic Measurement of Source Code Complexity. Master’s Thesis, Lulea University of Technology, Lulea, Sweden, 2011.