Подход к восстановлению потока управления запутанной программы

И.Н.Ледовских, М.Г.Бакулин
{il, bakulinm}@ispras.ru

Аннотация. Запутывание потока управления является одним из наиболее распространенных способов защиты бинарного кода приложений от анализа. Запутывающие преобразования резко увеличивают трудоёмкость выделения и распознавания алгоритмов и структур данных. В статье описывается подход к восстановлению потока управления программы, запутанного комбинацией различных преобразований – от вставки недостижимого кода с помощью непрозрачных предикатов до виртуализации. Приводятся результаты тестирования прототипной реализации данного метода на модельном примере, запутанном различными инструментальными средствами защиты.

Ключевые слова: запутывание потока управления, восстановление потока управления программы

1. Введение

Восстановление алгоритмов при анализе приложений в бинарных кодах осуществляется посредством достаточно большого числа различных методов анализа. Часть этих методов применяется в виде последовательно выполняемых этапов, выполняющих выделение и различные преобразования интересующих аналитика фрагментов приложения. В частности, это относится к методам преодоления механизмов защиты кода от анализа.

В настоящее время применяется множество способов защиты бинарного кода программ как от статических, так и от динамических методов анализа. Примерами таких защитных механизмов могут служить полиморфизм кода, превентивные преобразования, нарушающие работоспособность средств анализа, или средства обнаружения отладчиков и виртуальных сред. Однако наиболее распространенным механизмом защиты кода от анализа остаются запутывающие преобразования. Известные разновидности запутывающих преобразований, а также оценки их эффективности и устойчивости, приводятся в работе [1]. Применительно к анализу программ в бинарных кодах, из всех перечисленных в данной классификации видов запутывания, наиболее актуальными являются преобразования, запутывающие поток управления программы. В этой категории можно выделить преобразования переупорядочивания (переупорядочивание выражений, операторов и циклов), преобразования

агрегации (например, вставки либо выделение функций, клонирование кода, раскрутка циклов) и преобразования вычислений (например, преобразование графа потока управления к несводимому виду, расширение условий циклов, виртуализация кода). В данной работе рассматривается преодоление некоторых запутывающих преобразований, относящихся к группам агрегации и преобразования вычислений, в системе TrEx.

Значительное число преобразований, запутывающих поток управления, основано на использовании непрозрачных переменных и непрозрачных предикатов. Эти понятия вводятся в работах [1] и [2]. Достаточно будет сказать, что это переменные и предикаты, свойства которых априори известны при выполнении запутывающего преобразования, но являются достаточно сложными для распознавания при распутывании. Таким свойством для непрозрачной переменной может являться, например, диапазон ее значений, а для предиката - значение (истина или ложь). Внесение в код непрозрачного предиката P^F (со значением «ложь») или P^T (со значением «истина») приводит к тому, что ветки, соответствующие условию «истина» в первом случае и условию «ложь» во втором, никогда не выполнятся, и соответствующий код становится *недостижимым*. Непрозрачные предикаты используются для запутывания потока управления как сами по себе, так и в комбинации с другими методами запутывания. В частности, *расширение условий циклов* делает более сложным условие завершения цикла за счёт добавления непрозрачного предиката, не влияющего на число итераций цикла. С помощью предикатов вида $P^?$ реализуется клонирование кода, то есть копирование некоторого фрагмента кода, который выполняется наравне с исходным фрагментом в порядке, зависящем от значения непрозрачного предиката. При этом оба фрагмента-клона, как правило, дополнительно запутываются различными способами (например, переупорядочиванием инструкций и вставкой мёртвого или избыточного кода) для затруднения распознавания клонирования.

Широкое применение непрозрачных предикатов обусловлено в значительной степени невысокими накладными расходами в плане влияния данного запутывающего преобразования на быстродействие, и достаточно большим временем их взлома известными методами [4].

Другое рассматриваемое в настоящей статье запутывающее преобразование – значительно более затратное, но при этом более устойчивое к распутыванию – виртуализация кода [5,6]. Защищаемый участок кода приложения транслируется в байт-код некоторой виртуальной машины (ВМ), после чего выполняется замена этого фрагмента приложения кодом соответствующего интерпретатора (рис.1). Исходный фрагмент удаляется безвозвратно, и для восстановления его семантики необходимо провести достаточно сложный анализ использованной виртуальной машины, а затем выполнить обратную трансляцию ее байт-кода в систему команд исходной целевой архитектуры. Задача осложняется тем, что как код самой виртуальной машины, так и ее данные подвергаются дополнительному запутыванию различного уровня

сложности – от вставки непрозрачных предикатов, недостижимого и мёртвого кода до мутации кодов операций виртуальной машины и ее ре-виртуализации.

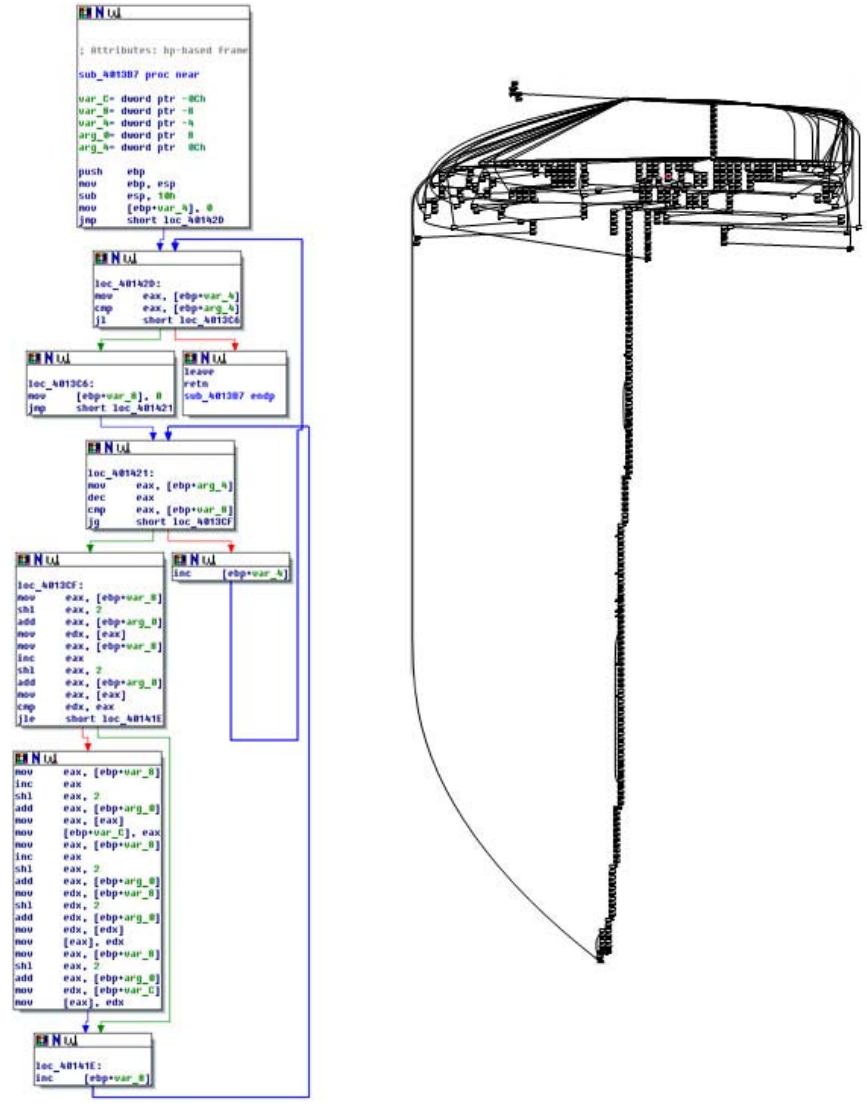
В настоящей статье мы рассмотрим подход к восстановлению потока управления программы, запутанной посредством комбинации преобразований различной сложности. Работа организована следующим образом. Во втором разделе рассматриваются примеры распространенных в настоящее время инструментальных средств защиты программ, использующих виртуализацию совместно с другими запутывающими преобразованиями. В третьем разделе дается краткий обзор известных методов распутывания потока управления, в частности, методов преодоления виртуализации. Далее, в разделах 4 и 5 мы рассматриваем подход, реализуемый системой TrEx, и полученные при апробировании этого подхода результаты. В заключительном шестом разделе полученные результаты оцениваются с точки зрения их практической применимости, а также рассматриваются направления дальнейших исследований.

2. Инструменты запутывания программ

В ходе выполнения данной работы мы провели сравнительный анализ и тестирование шести инструментов запутывания программ, использующих технологию виртуализации. Это три российские разработки – VMProtect [12], Enigma Protector [14], Private exe Protector [16], и три зарубежных – CodeVirtualizer [13], Safengine [15] и Obsidium [17]. Две системы были исключены из дальнейшего анализа – Obsidium, демонстрационная версия которого не поддерживает технологии виртуализации кода, и Private exe Protector, реализующий защиту упаковкой кода; виртуализацией в этой системе защищен только распаковщик, а код защищаемого приложения после распаковки доступен в памяти и может быть легко восстановлен динамическими методами.

Оставшиеся четыре обфускатора реализуют виртуальную машину с интерпретатором кода типа Decode-Dispatch, поток управления которой имеет характерный вид (рис.1Б) с главным циклом выборки и декодирования инструкций байт-кода и широким ветвлением, реализующим обработку декодированных инструкций. Различия рассматриваемых систем защиты заключаются в основном в модели виртуальной машины (например, стековая RISC-машина в [12] и регистровые RISC и CISC-машины в [13]), в конкретных реализациях байт-кода и в средствах дополнительной защиты. Так, CodeVirtualizer и Safengine предоставляют возможность дополнительного запутывания кода с помощью непрозрачных предикатов (Multi-Branch Technology); первая из этих систем реализует также запутывание данных (обфускация кодов операций и константных операндов), а также ряд дополнительных методов запутывания, недоступных в демо-версии (мутация кодов операций, ре-виртуализация, эмуляция стека). Различаются у рассматриваемых виртуальных машин также реализации обработчиков

инструкций – это отдельные функции для обработки различных инструкций в [15] и блоки обработки в операторе ветвления внутри одной функции в [12,13,14].



А Б

Рис.1.Граф потока управления исходной программы (А) и после запутывания посредством виртуализации (Б)

Все рассматриваемые виртуализаторы работают с программами для архитектуры Intel 64 и способны защищать приложения как для среды Win32, так и для 64-разрядного окружения.

Выше приводится пример запутывания виртуализацией – граф потока управления простой тестовой программы (рис. 1А) и ее защищенной версии (рис.1Б), которая получена с помощью системы CodeVirtualizer с дополнительным запутыванием потока управления непрозрачными предикатами.

3. Подходы к распутыванию потока управления

В настоящее время основой большинства известных подходов к распутывания программ в бинарных кодах являются различные комбинации динамических и статических методов. Это применимо как к преодолению достаточно простых видов запутывания потока управления (как описано в работах [3,4]), так и к распутыванию кода после виртуализации.

Статический анализ и восстановление кода, защищенного с помощью виртуальных машин, представляет собой сложный многоэтапный процесс [5]. В процессе анализа необходим полный анализ виртуальной машины «вручную». Опытный инженер-аналитик должен восстановить систему команд машины, разработать внутреннее представление и реализовать транслятор, отображающий байт-код виртуальной машины в это внутреннее представление. Хотя для каждой отдельной реализации ВМ эта задача имеет вполне приемлемую трудоемкость, ограничения при таком подходе велики. Необходимо учитывать шаблонную структуру инструкций байт-кода и изменчивость кодов операций в рамках шаблонов; далее, появление модифицированных версий существующих ВМ может потребовать повторного анализа.

Вторым этапом распознается точка входа в виртуальную машину; эта задача, сложная для статического анализа, не представляет собой проблемы при динамическом подходе. Затем распознается структура ВМ, и локализуется ее байт-код. Далее, для конкретной реализации байт-кода и конкретной ВМ должен автоматически генерироваться дизассемблер, распознающий шаблоны операций; задача осложняется применением к защищенному виртуализацией коду дополнительных запутывающих преобразований, препятствующих распознаванию и автоматическому анализу блоков обработки байт-кода. Кроме того, рассматриваемые обфускаторы применяют различные виды запутывания данных, затрудняющие распознавание шаблонов инструкций байт-кода – в первую очередь, это запутывание константных операндов инструкций с помощью арифметико-логических преобразований, различающихся для различных экземпляров ВМ.

При успешном построении дизассемблера байт-кода конкретной ВМ, выполняется генерация внутреннего представления для последовательности

инструкций этой машины. Затем к полученному внутреннему представлению применяется ряд компиляторных оптимизаций. В первую очередь, делается замена обращений к стеку простыми присваиваниями, затем – распространение констант и копий. Такие оптимизации способны уменьшить объем внутреннего представления до 20% от изначально сгенерированного [5]. Далее, для виртуальных RISC-машин выполняется обратная замена последовательностей арифметических и логических инструкций – во внутреннем представлении восстанавливаются операции, отсутствующие в системе команд ВМ и реализованные через другие примитивы (например, VMProtect реализует логическую операцию AND посредством трех NOR, а арифметическую SUB – через операции NEG и ADD). Наконец, по оптимизированному внутреннему представлению генерируется код целевой архитектуры – в рассматриваемых случаях, Intel 64.

Как можно видеть, такой статический подход сложен, трудоемок и не обладает свойством общности, хотя и способен в случае успеха максимально точно восстановить исходный код запутанной программы. Динамический подход на основе трассировки защищенного приложения является более универсальным и гибким.

Например, в работе [7] описывается подход, основанный на анализе обращений к памяти при выполнении запутанного кода в защищенной среде программного эмулятора QEMU. Описываемый подход основан на распознавании в трассе программы переменных, выделении из их числа подмножества переменных, которые могли служить виртуальными счетчиками команд (VPC), и анализе кластеров памяти, адресуемых посредством этих переменных; в результате такого полностью автоматического анализа удается локализовать интерпретируемый байт-код. Далее, для анализа байт-кода используется знание структуры ВМ в самых общих чертах – достаточно распознавать главный цикл выборки и интерпретации, а также отдельные блоки интерпретации инструкций виртуальной машины (фазы эмуляции). Дальнейшее распознавание синтаксиса и семантики инструкций байт-кода основано на анализе способов доступа фаз эмуляции к кластеру памяти, хранящему интерпретируемую программу, и на восстановлении семантики отработавших в процессе трассировки блоков интерпретации байт-кода. Такой подход позволяет, в частности, относительно легко восстанавливать семантику инструкций передачи управления путем анализа изменений значения VPC. Восстановление вычислений остается достаточно сложной задачей.

В работе [8] предлагается следующий шаг по обобщению подхода к восстановлению виртуализованного кода – восстановление семантики исходной запутанной программы с помощью анализа зависимостей по данным и динамического слайсинга [11]. В трассе запутанной программы выделяется рад событий (например, системные вызовы или просто инструкции, оставленные виртуализатором без изменений); эти события анализируются и

используются как опорные точки для построения динамических обратных слайсов на основе только зависимостей по данным. Инструкции, попавшие в этот набор слайсов, затем объединяются в подтрассу, которая является аппроксимацией семантики запутанного посредством виртуализации кода. Соответствующий поток управления восстанавливается отдельно на основе анализа обращений к памяти при выборке инструкций байт-кода интерпретатором виртуальной машины.

4. Восстановление потока управления в системе TrEx

В системе TrEx в настоящее время исследуется подход, сочетающий достоинства перечисленных в разделе 3 работ. Вначале мы стараемся упростить трассу запутанного приложения посредством обнаружения и удаления непрозрачных предикатов, а также мертвого и избыточного кода. Затем происходит локализация интерпретатора виртуальной машины, его анализ и восстановление программы комбинацией описанных выше динамических методов. В частности, поток управления запутанной программы восстанавливается на основе алгоритма, описанного в работе [7].

4.1. Поиск и распознавание непрозрачных предикатов.

При динамическом анализе бинарного кода по трассе можно легко найти все предикаты, которые всегда передавали управление по одной и той же ветке, то есть собрать и проанализировать статистику переходов. Однако при таком чисто статистическом подходе будет наблюдаться большое число ошибочных распознаваний непрозрачных предикатов. Так, известно, что в среднем даже в незапутанной программе около 60% предикатов, наблюдаемых в трассе, всегда передают управление по одной и той же ветке[3]. Ситуация может быть улучшена посредством анализа большего количества трасс, однако полное покрытие кода всё равно не может быть достигнуто, а затраты времени возрастают. Поэтому в системе TrEx распознавание непрозрачных предикатов выполняется следующим образом.

1. По нескольким трассам одного приложения составляется список всех предикатов, которые передавали управление по одной ветке.
2. Этот список фильтруется, и в нём остаются только те предикаты, которые потенциально являются непрозрачными. При этом удаляются как полезные предикаты, которые могут передавать управление по другой ветке при других входных параметрах, так и предикаты, слишком сложные для автоматического анализа, хотя и являющиеся потенциально непрозрачными.

Для отсеивания ищется первая позиция исследуемого предиката в трассе, после чего выполняется динамический обратный слайсинг. Полученный слайс предиката проверяется на соответствие различным критериям (например, количество инструкций, содержащихся в слайсе; количество базовых блоков, в которые попадают инструкции слайса; модуль, к которому принадлежит

инструкция перехода, и т.д.). Следующий этап фильтрации отсеивает предикаты, которые на самом деле являются проверками значений параметров и результатов функций. Затем при наличии других трасс приложения проверяется, по одним и тем же путям выполняются переходы в предикатах из сокращенного списка, или нет. В случае, если в каком-либо месте трассы слайс предиката получается другим (то есть условие перехода по тому же адресу вычисляется иначе), дальнейшая обработка этого предиката прекращается, и он отвергается.

Далее происходит анализ предикатов на использование непрозрачных переменных. Строится список адресов используемых в предикатах переменных, и по имеющимся трассам находятся все обращения к этим адресам. В результате получаются списки:

- адресов, к которым обращались из каких-либо иных мест программы, кроме предикатов из сокращенного списка;
- адресов, в которые что-либо записывалось;
- адресов, в которых содержались исполненные инструкции.

Эти списки далее используются как для отсева сомнительных предикатов, так и для автоматического распознавания непрозрачных предикатов, основанных на использовании непрозрачных переменных, а также аналитиком для распознавания таких предикатов в автоматизированном режиме.

Наконец, к сокращенному списку потенциально непрозрачных предикатов применяется поиск по шаблонам – слайсы сравниваются с имеющимися шаблонами ранее распознанных непрозрачных предикатов. Не распознанные и не отвергнутые ранее потенциально непрозрачные предикаты поочередно выносятся на рассмотрение аналитику, который в итоге выносит вердикт о том, принимаются они, или отвергаются.

В результате такого отсеивания остаётся сокращённый список предикатов, на основе которого система упрощает трассу приложения, удаляя распознанные предикаты P^F. Предикаты P^T в настоящее время из трассы не удаляются.

4.2. Поиск простого комбинированного запутывания потока управления

Часто применяется не один метод запутывания, а их комбинация. Например, непрозрачный предикат P^F может задавать переход на инструкцию внутри другого непрозрачного предиката, что усложняет его анализ; также непрозрачные предикаты маскируют участки клонированного кода, когда в разные ветки добавляются разные непрозрачные предикаты. Мёртвый или избыточный код также зачастую вставляется в программу вместе с непрозрачными предикатами. В системе TrEx факт присутствия мёртвого кода в ряде случаев выявляется одновременно с анализом предикатов. При фильтрации списка предикатов ищутся участки мёртвого кода, перемешанного со слайсом предиката. Примером такого перемешивания

может быть добавление операций с регистром, значение которого потом не используется или сразу затирается другим значением. В результате простой предикат может выглядеть, например, так:

```
MOV AX, 1
MOV BX, 10
SUB BX, 6
CMP AX, 0
JZ LABEL
MOV BX, …
{…}
LABEL: {…}
```

Дополнительный анализ основан на проверке наличия между шагами трассы, составляющими слайс предиката, не попавших в слайс инструкций. Если такая инструкция есть, то строится и анализируется последовательность зависящих от нее по данным шагов трассы – динамический прямой слайс.

Избыточный код, комбинированный с непрозрачными предикатами, обнаруживается с помощью шаблонов. Например, вместо этой последовательности команд:

```
PUSH EAX
MOV EAX, ECX
PUSH EDX
MOV EDX, EBX
NOT EAX
NOT EDX
OR EAX, EDX
NOT EAX
MOV ECX, EAX
POP EDX
POP EAX
```

можно использовать инструкцию AND ECX, EBX (если мы не учитываем, что в стеке остались значения регистров EAX и EDX). Поэтому на участках трассы, где локализованы непрозрачные предикаты, производится поиск фрагментов, совпадающих с известными шаблонами запутывания.

4.3. Восстановление потока управления виртуализованного кода

Большая часть эмуляторов виртуальных машин действует по следующей схеме:

1. Считывание кода операции из памяти.
2. Передача управления на соответствующую ветку.
3. Исполнение кода, эмулирующего данный код операции. Переход к 1).

У каждого такого эмулятора есть счетчик команд виртуальной машины (VPC), который определяет, из какой области памяти необходимо считывать следующий код операции.

Основываясь на этих общих свойствах, можно автоматизировать анализ того, какая переменная или регистр машины, на которой запускается приложение, используется как счетчик команд виртуальной машины, а также поиск участка памяти, содержащего интерпретируемый байт-код. Это делается в два этапа:

1. Кластеризация всех считываний по неконстантным адресам из памяти в регистр.
2. Поиск кластеров, считанные из которого данные влияют на передачу управления.

3.3.1. Кластеризация

Будем говорить, что на шаге трассы регистр привязан к участку памяти, если значение регистра на этом шаге является копией или модификацией значения, хранящегося в этом диапазоне памяти. Пусть в трассе имеется считывание из памяти, причем адрес вычисляется динамически на основании значения одного или нескольких регистров. Тогда такое чтение можно связать со всеми участками памяти, к которым привязаны эти регистры. После этого все чтения из памяти можно разбить на кластеры в зависимости от того, к каким участкам памяти привязаны эти считывания.

Необходим алгоритм, который бы позволил автоматически связать считывания из памяти с регистрами, а регистры с участками памяти. В статье [7] приводится такой алгоритм, модифицированная версия которого описывается ниже.

Обозначения:

`B(R,i)`	участки памяти, к которым привязан регистр `R` на шаге `i`;
`R<--Val`	в `R` заносится значение, зависящее от `Val`;
`R<<-Val`	в `R` заносится значение, зависящее от `Val` и значения регистра R;
`[M,S]`	участок памяти с адресом `M` и длиной `S`;
`C`	константа;
`OutDep(i)`	множество регистров, значение которых изменилось после выполнения инструкции на шаге `i`.
`InDep(i)`	множество регистров, значения которых повлияли на выходные параметры инструкции.

Связывание происходит в два этапа. На первом этапе мы привязываем считывания из памяти в регистр, и распространяем его вперёд. Алгоритм в виде псевдокода:

```
for i = start to end do
```

```
      foreach R in OutDep(i)
            if R <-- [M,S]
            then
                  B(R,i) = {[M,S]}
            end if
            if R <<- [M,S]
            then
                  B(R,i) = B(R,i-1) UNITE {[M,S]}
            end if
            if R1 <-- [f(R2,R3,...),S] OR R1 <<-
[f(R2,R3,...),S]
                  VarsF(i) = B(R2,i-1) UNITE B(R3,i-1)
UNITE ...
            end if
      end foreach
end for
```

На втором этапе мы привязываем регистр к участку памяти в момент записи его значения в памяти, и распространяем связывание назад. В виде псевдокода это выгляди так:

```
for i = end to start do
      foreach R in InDep(i)
            if [M,S] <-- R
            then
                  B(R,i) = {[M,S]}
            end if
            if R1 <-- R
            then
                  B(R,i) = B(R1,i+1)
            end if
            if R1 <-- [f(R2,R3,...),S] OR R1 <<-
[f(R2,R3,...),S]
                  VarsB(i) = B(R2,i-1) UNITE B(R3,i-1)
UNITE ...
            end if
      end foreach
end for
```

Следует заметить, что привязки B(R,i) обнуляются перед запуском каждого этапа.

После этого для каждого шага трассы вычисляется множество привязанных переменных Vars(i) = VarsF(i) UNITE VarsB(i). Далее происходит разбиение считываний на кластеры, в один кластер попадают все считывания из памяти, для которых пересечение их множества привязанных переменных не пусто.

Предположим, что нам известны все позиции операций считывания кода операции из памяти в трассе, а также известен регистр, использующийся как

счетчик команд виртуальной машины. Тогда можно восстановить граф потока управления исходной программы. Для этого для каждого значения счетчика команд запоминаются все значения, которые счетчик команд принимает при следующем считывании кода операции из памяти. Полученные данные можно представить в виде ориентированного графа, где каждая вершина обозначает возможное значение счетчика команд, а ребро a->b обозначает, что счетчик команд после значения a может принимать значение b. После этого полученный граф упрощается: все вершины, у которых ровно один предшественник и один последователь, и у их предшественника только один последователь, удаляются, и добавляется ребро из их предшественника в их последователя.

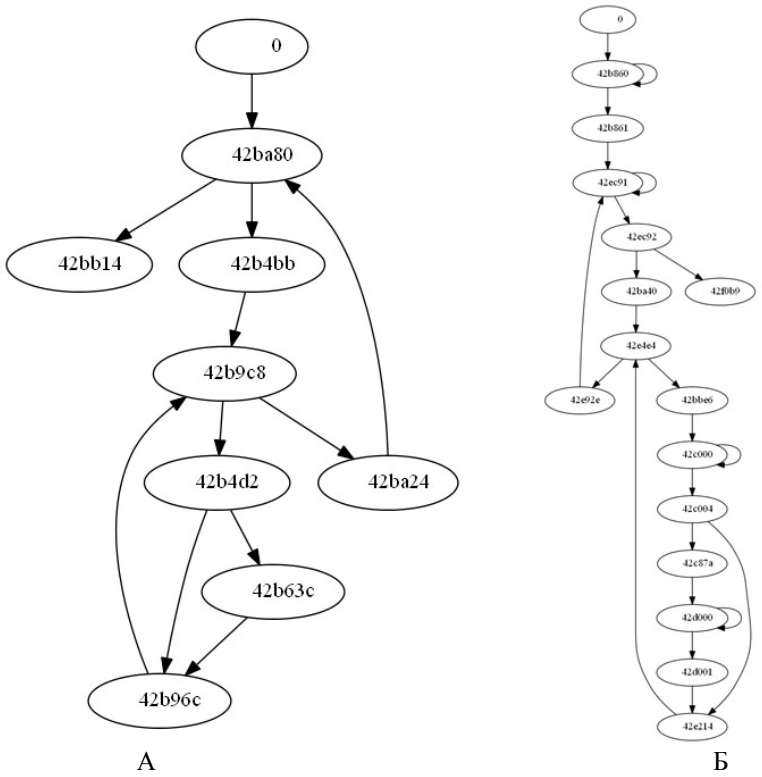
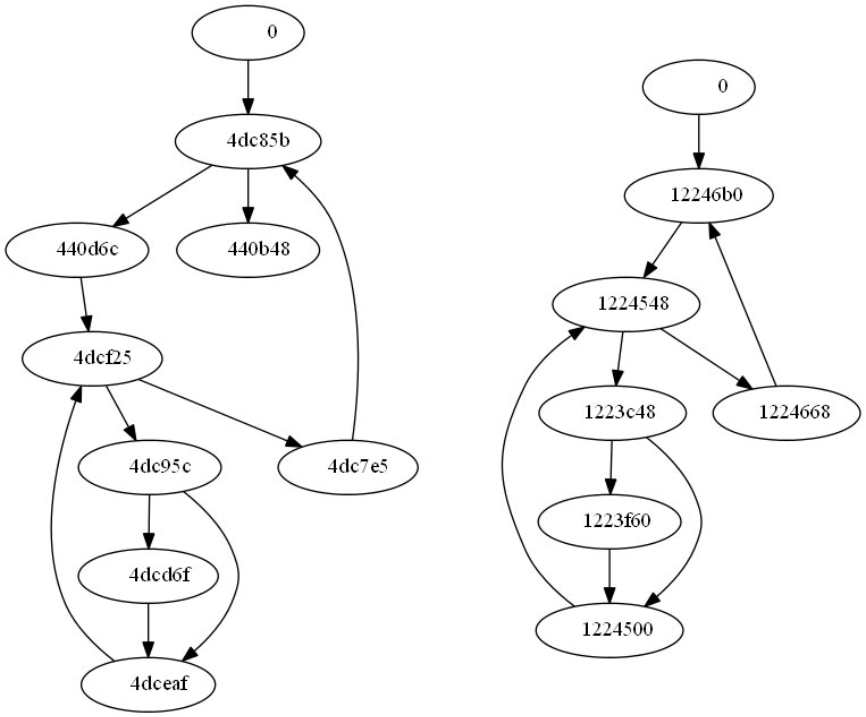


Рис.2 Восстановленный граф потока управления после запутывания с помощью CodeVirtualizer (А) и CodeVirtualizer с Multi Branch Technology (Б)

5. Результаты экспериментов

Описанный подход к восстановлению потока управления запутанной программы был опробован на модельных примерах, код которых был запутан демонстрационными версиями обфускаторов VMProtect, CodeVirtualizer, Enigma Protector и Safengine. В качестве основного примера, используемого в данной статье, использовалась программа, реализующая алгоритм пузырьковой сортировки. Исходный граф потока управления, полученный с помощью IDA Pro, приведен выше на рисунке 1А.



А Б

Рис. 3.Восстановленный граф потока управления после запутывания посредством VMProtect (А) и Enigma Protector (Б)

Запутывание посредством CodeVirtualizer и Safengine выполнялось в двух вариантах: 1)только с использованием виртуальной машины и 2)виртуализацией с добавлением Multi Branch Technology. На рисунке 1 приводятся результаты восстановления потока управления программы, запутанной с помощью CodeVirtualizer. Можно видеть, что для программы без дополнительного запутывания (рис. 1А) поток управления восстановлен

точно. Восстановленный граф потока управления после виртуализации с добавлением Multi Branch Technology (рис. 2Б) содержит больше базовых блоков и переходов, то есть непрозрачные предикаты не были удалены из анализируемой трассы. Это связано с тем, что для экспериментов использовалась версия системы TrEx 2.0, в которой формат трассы не поддерживает перемещения фрагментов трассы и, соответственно, удаления непрозрачных предикатов вида $\boldsymbol{P^T}$. Однако эти предикаты распознаются соответствующим модулем расширения системы и помечаются в трассе для удаления в будущих версиях. Что касается восстановления потока управления исходной программы, то и в этом варианте оно выполнено полностью.

Восстановление потока управления программы, запутанной с помощью VMProtect и Enigma Protector, также было выполнено успешно (рис. 3 А,Б).

В то же время, демонстрационная версия обфускатора Safengine при первых попытках восстановления потока управления вызвала ряд сложностей как в базовом варианте (только виртуализация), так и с использованием технологий дополнительного запутывания. Восстановленный граф потока управления в обоих случаях содержит избыточные передачи управления, анализ которых нами пока не завершен. Поэтому в данной статье мы воздерживаемся от оценки результатов, полученных применительно к данной системе защиты приложений.

6. Заключение

Описанный в разделе 4 подход к восстановлению потока управления был реализован прототипом комплекса программных инструментов в рамках среды динамического анализа защищенного бинарного кода TrEx. Реализация представляет собой набор модулей расширения системы, для корректной работы которых требуется построение графа потока управления для каждой функции анализируемого приложения. Для ускорения анализа с использованием алгоритмов слайсинга желательно построение графа зависимостей.

При тестировании прототипа были выявлены ограничения и неточности реализованного подхода, в частности, невозможность удаления из трассы части распознанных непрозрачных предикатов, а также неточность восстановления потока управления программы, запутанной с помощью обфускатора Safengine. Ограничения, связанные с форматом трассы и невозможность удаления непрозрачных предикатов $\boldsymbol{P^T}$, планируется преодолеть в одной из будущих версий. Восстановление потока управления после запутывания с помощью Safengine требует дальнейшего исследования; эти работы продолжаются в настоящее время наряду с работами по исследованию задачи восстановлению потока данных запутанной посредством виртуализации программы. В целом мы считаем, что первые эксперименты демонстрируют возможность практического применения предложенного

подхода для преодоления комплексного запутывания потока управления приложений в бинарных кодах.

В дальнейшем предполагается продолжить работы по улучшению и расширению применимости предложенной методики и ее программной реализации. Также планируется реализовать восстановление потока данных виртуализованного кода и объединение подтрасс, полученных в результате раздельного восстановления потока управления и потока данных. Также за рамками данной статьи осталось описание методики, с помощью которой в трассе локализуются участки с различными видами запутывания, что позволяет существенно ускорить работу описанных в статье алгоритмов за счет их применения к предварительно выбранным частям трасс приложения.

Литература

[1] Christian Collberg, Clark Thomborson, Douglas Low. A Taxonomy of Obfuscating Transformations. / Technical Report #148. Department of Computer Science, The University of Auckland. July, 1997

[2] А.В. Чернов. Анализ запутывающих преобразований программ. // Труды Института системного программирования РАН, том 3, 2002 г. Стр. 7-38.

[3] S.K. Udupta, S.K. Debray, and M. Madou. Deobfuscation: Reverse engineering obfuscated code. In Proc. 12th IEEE Working Conference on Reverse Engineering, pp.45-54, November 2005

[4] Mila Dalla Preda, Matias Madou, Koen De Bosschere, Roberto Giacobazzi. Opaque Predicates Detection by Abstract Interpretation. // In Proc. Intern. Conf on Algebraic Methodology and Software Technology. 2006 pp. 35-50.

[5] R. Rolles. Unpacking virtualization obfuscators. In Proc. 3rd USENIX Workshop on Offensive Technologies (WOOT'09), August 2009

[6] B. Lau. Dealing with virtualization packer. In Second CARO Workshop on Packers, Decryptors, and Obfuscators, May 2008.

[7] M. Sharif, A. Lanzi, J. Griffin, and W. Lee. Automatic reverse engineering of malware emulators. In Proc. 2009 IEEE Symposium on Security and Privacy, May 2009.

[8] Kevin Coogan, Gen Lu, Saumya Debray. Deobfuscation of Virtualization-Obfuscated Software: A Semantic-Based Approach. In Proc. 18-th ACM Conference on Computer and Communication Security, October 2011.

[9] 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.

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

[11] B. Korel, J. Laski. Dynamic program slicing. // Information Processing Letters, vol. 29, issue 3, pp. 155-163. — 1988.

[12] VMPSoft. VMProtect. http://www.vmpsoft.com

[13] Oreans Technologies. CodeVirtualizer. http://www.oreans.com

[14] The Enigma Protector. http://enigmaprotector.com

[15] Safengine. Safengine Protector. http://www.safengine.com

[16] Setisoft Technology. Private Exe Protector. http://www.setisoft.com

[17] Obsidium Software. Obsidium. http://www.obsidium.de