Построение обфусцирующего компилятора на основе инфраструктуры LLVM

Курмангалеев Ш. Ф., Корчагин В. П., Савченко В. В., Саргсян С.С.
korchagin@ispras.ru, kursh@ispras.ru sinmipt@ispras.ru
sevaksargsyan@ispras.ru

Аннотация. В статье описываются маскирующие преобразования, реализованные в ходе разработки обфусцирующего компилятора в ИСП РАН, приводится оценка понижения быстродействия и увеличения объема потребляемой приложением памяти, а также оценка возможности восстановления информации об исходном коде. Реализованные маскирующие преобразования могут быть одновременно применены к запутываемому приложению, что увеличивает степень защиты приложения и обеспечивают стойкую защиту от статического анализа.

Ключевые слова: llvm; обфускация.

1. Введение

Оптимальным выбором, позволяющим реализовать стойкие варианты запутывания программ, является создание обфусцирующего компилятора на базе одной из существующих компиляторных инфраструктур. С одной стороны, это позволит производить запутывание программы, имея полную информацию о ней на всех этапах компиляции, с другой – позволит сосредоточиться на разработке алгоритмов защиты, а не на создании требуемой инфраструктуры. Для реализации была выбрана компиляторная инфраструктура LLVM[1], в качестве компилятора переднего плана используется Clang (читается как клэнг).

Целью работы является построение обфусцирующего компилятора для защиты программы от обратной инженерии с помощью инструментов, использующих статический анализ кода. Все разработанные преобразования представляют собой отдельные компиляторные проходы, запускаемые поочередно, после окончания работы оптимизирующих проходов. Преобразования производятся во время обработки промежуточного представления LLVM на машинно-независимом уровне, что, с одной стороны, позволяет получать запутанное промежуточное представление, которое в дальнейшем можно преобразовать в код на языке Си с помощью стандартных

инструментов LLVM. С другой стороны, такой подход обеспечивает поддержку нескольких архитектур при условии совпадения порядка байтов и минимального различия в ABI.

При разработке преобразований учитывались критерии эффективности:

- Маскирующее 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 [2];
- При разработке преобразования нужно учитывать особенности работы средств анализа [3], 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 трудного предиката;
- Запутывание графа потока управления – создание несводимых участков в графе;
- Замена вызовов одной и той же функции на вызовы ее копий.

2. Описание маскирующих преобразований

2.1. Преобразование, маскирующее вызов функций

Для защищаемого вызова создается функция-переходник, внутри которой содержится несколько вызовов функций. Внутри переходника вызов нужной функции диспетчеризуется по значению трудного предиката. Реализовано два варианта преобразования: только для вызовов внешних функций и для вызовов всех функций.

Для каждого вызова функции производится его замена на вызов функции-переходника. Для избегания чрезмерной вложенности вызовов, переходники на переходники не создаются. Для выбора функций, которые будут размещены внутри переходника, была введена мера "близости функций".

Значение меры - коэффициент Жаккарда для множеств типов аргументов двух функций. Половина функций в переходнике выбираются, как самые "похожие" по введенной мере, и другая половина - "непохожие".

Аргументы, передаваемые в функцию-переходник запутываются с помощью битовой операции "исключающее ИЛИ". Между всеми аргументами делается операция "исключающего ИЛИ", обозначим результат операции за S. Затем к каждому аргументу применяется операция "исключающего ИЛИ" с S, и в таком виде аргумент передается в функцию. Также для распутывания передается само значение S. Внутри функции-переходника сначала происходит распутывание аргументов. Производится операция "исключающего ИЛИ" между каждым аргументом и значением S. В результате аргумент получает свое исходное значение.

После распутывания аргументов происходит вычисление непрозрачного предиката, по результату которого происходит диспетчеризация вызова функций. Диспетчеризация вызовов функций производится с помощью большого switch блока. Каждое значение в нем сгенерировано случайным образом и соответствует какой-либо функции. Все вызовы перемешиваются в случайном порядке внутри функции-переходника. Последний аргумент функции-переходника служит для определения функции, которая будет вызвана. Этот аргумент передается в непрозрачный предикат. Так, как значение предиката известно, на этапе компиляции, то можно подобрать такое значение аргумента, которое соответствовало бы нужной функции.

2.2. Маскирующее преобразование «диспетчер»

Идея маскирующего преобразования «диспетчер» заключается в преобразовании графа потока управления таким образом, что статический анализ переходов между базовыми блоками становится трудной задачей [4].

Данное преобразование состоит в том, что базовым блокам присваиваются номера. В начало функции вставляют блок «диспетчер» - аналог switch в языке C. В конец каждого блока дописывается, код устанавливающий номер следующего блока для выполнения и передающий управление на диспетчер. Диспетчер же на основе переданной ему информации принимает решение куда дальше передать управление.

Терминальные инструкции маскируемых базовых блоков удаляются, а содержащие их базовые блоки модифицируются таким образом, чтобы передать управление диспетчеру с номером базового блока, соответствующего нужной ветви перехода. Для каждого блока делается до 5 копий, которые так же добавляются в диспетчер. Помимо этого, производится усреднение размера базовых блоков, инструкции “call” оцениваются как несколько инструкций, так как передача параметров в промежуточном представлении LLVM производится в той же команде, а на реальных архитектурах по команде на аргумент. Для сокрытия переменной диспетчеризации предпринято следующее:

Значение переменной диспетчеризации вычисляется по формуле I = X1 XOR Z; а следующее значение Z по формуле $Z_{след}$ = X2 XOR $Z_{текущее}$; Z, X1 и X2 выбираются случайным образом для блока предшествующего диспетчеру, и X2 генерируется случайным образом для каждого блока исходной функции во время его обработки.

В каждом блоке выбирается одна переменная подходящего типа, с которой посредством операции "исключающего ИЛИ" (XOR) происходит сцепление переменной диспетчеризации. Такое преобразование не позволит выделить переменную диспетчеризации с помощью алгоритма обратного слайсинга [5], так как в полученную таким образом трассу окажутся, вовлечены живые переменные, вычисляемые в программе.

Вид графа потока управления после преобразования примерно следующий:

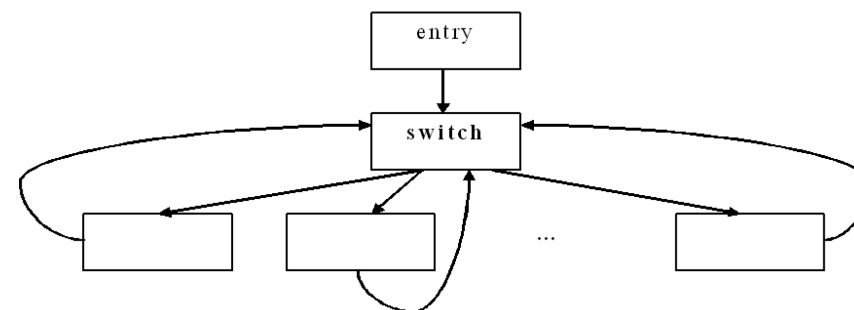

Рис. 1. Вид графа потока управления после запутывания.

2.3. Преобразование, маскирующее строковые константы

Зачастую во время статического анализа, строковые константы, хранящиеся в открытом виде, могут дать аналитику дополнительную информацию о функционировании программы [6]. Или помочь найти интересующий код, по строкам, выводимым во время интересующего его события. Преобразование, маскирующее строковые константы, предназначено для сокрытия информации о строках, во время статического анализа программы.

Защита выполняется следующим образом: вначале все константные строки, кроме тех, что содержатся в агрегатных типах (массивы, контейнеры из стандартной библиотеки), шифруются, в модуль добавляются шифрующая и дешифрующая функции. Перед каждым использованием той или иной строки вставляется вызов функции дешифратора, а после – шифрующей функции. Это справедливо для строк, для которых не выполняются операции с указателями. Если же такие операции имеют место, то для корректной работы запутывающего алгоритма необходим анализ указателей. В таких случаях

обратного шифрования строки не производится. Шифрование строк после использования требуется для того, что бы во время работы программы все строки не находились в памяти расшифрованными, поскольку в этом случае можно сделать снимок памяти процесса, после дешифрования строк. Шифрование строк производится с помощью операции исключающего "или" со случайным ключом.

2.4. Маскирующее преобразование, приводящее граф потока управления к несводимому

Колберг [7] описывает алгоритм, который приводит граф потока управления к несводимому. Алгоритм заключается в том, что для каждого цикла добавляется «фиктивное» ребро из заголовка цикла в его тело. Добавление такого ребра осуществляется с помощью вставки непрозрачных предикатов.

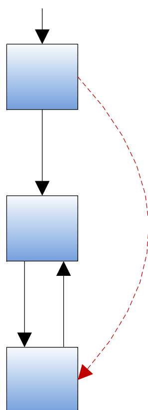

Рис.2. Добавление «фиктивного» ребра в цикл.

Таким образом, трансформируя граф потока управления, мы создаем сложный для анализа случай, препятствующий качественной декомпиляции.

2.5. Переплетение циклов

Алгоритм запутывания для всех циклов функции добавляет «фиктивные» ребра из одного цикла в другой. Как видно из рис. 2, метод создает ребра как входящие в цикл, так и выходящие из него.

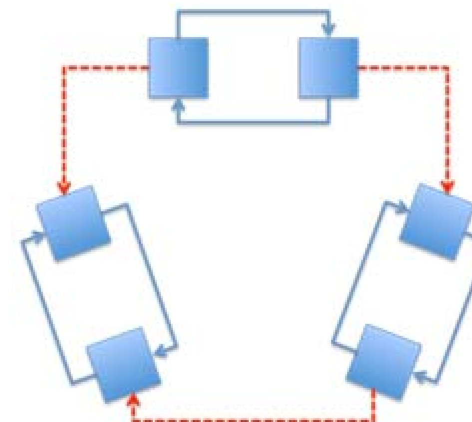

Рис.3. Переплетение 3 циклов, пунктиром показаны «фиктивные» ребра.

Минус такой трансформации состоит в том, что она эффективно запутывает только код больших функций, которые имеют сложные структуры, содержащие несколько циклов. Помимо этого применяется следующий подход для запутывания всего графа потока управления: из множества блоков функции выбирается N блоков и между ними случайно добавляются ребра. Фиктивные переходы защищаются непрозрачными предикатами.

2.6. Маскирующее преобразование, осуществляющее переплетение функций

Классический подход к переплетению функций, обладает малой стойкостью. Он предполагает объединение сигнатур функций и наличие параметра, по которому происходит диспетчеризация[8]. Восстановить исходный код переплетенных таким образом функций не составляет особого труда. Предложена модификация упомянутого алгоритма таким образом, чтобы помимо диспетчеризующего условия переплетаемые функции имели точки пересечения потоков управления и потоков данных, чтобы применение алгоритма обратного слайсинга не позволяло найти единственную точку, в которой производится выбор рабочей функции.

Переплетение происходит следующим образом:

1. Объединяются сигнатуры двух функций, генерируется дополнительный параметр, по которому в процессе выполнения будет производиться выбор функции.

2. Если функции возвращают целочисленное значение, то для реального возврата значения из переплетенной функции используются 2 глобальные перемененные, а сама функция возвращает неиспользуемое значение. Если оригинальные функции возвращают указатели, то тип возвращаемого

значения переплетенной функции становится указателем на void. Преобразование проиллюстрировано на рисунках

3. В новой функции, полученной на основе переплетения двух функций, произвольно выбираются два базовых блока (один из первой функции второй из второй). Для которых производится преобразование зацепления дуг [9] рис. 4. В генерируемом общем базовом блоке производятся вычисления с глобальными переменными. Для затруднения анализа потоков данных эти переменные используются для вычислений в и других функциях модуля. Таким образом, у двух переплетенных функций всегда будут общие вычисления. Результат вычислений, используются в качестве возвращаемого значения и переплетенной функции и в глобальную переменную, что не позволит исключить добавленные вычисления как мертвый код, результат которых нигде не используется.

4. После генерации переплетенной функции, все места вызова оригинальных функций, заменяются на вызов переплетенной функции с генерацией дополнительных параметров и изменением обработки возвращаемого значения.

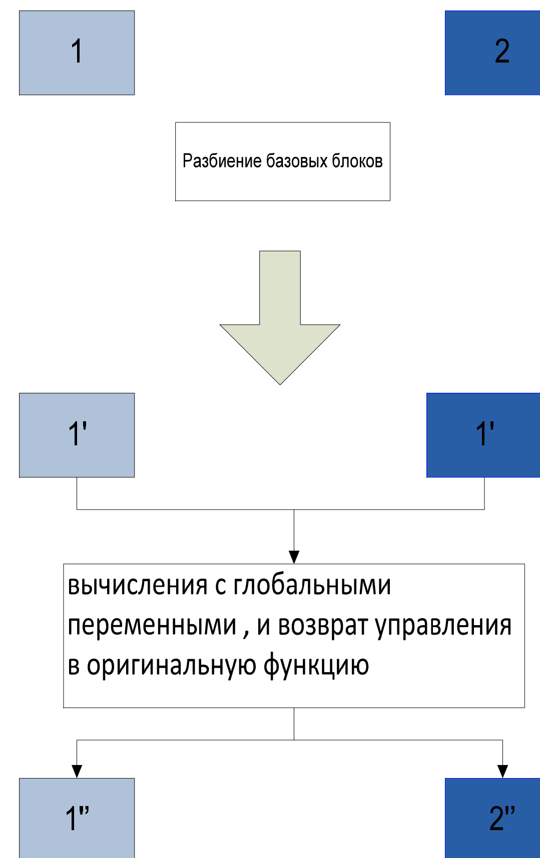

Рис. 4. Преобразование зацепления дуг.

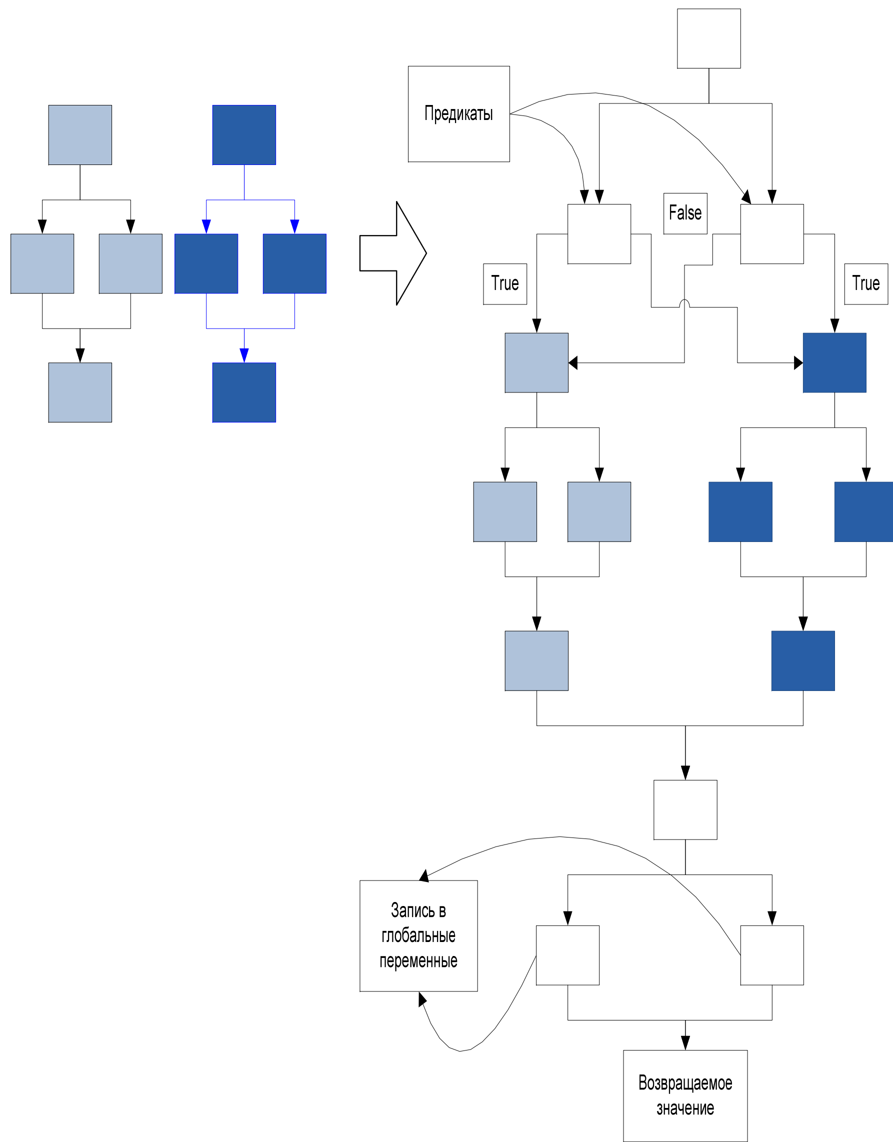

Рис. 5. Переплетение функций.

2.7. Маскирующее преобразование, направленное на увеличение сложности анализа потока данных в программе

Идея преобразования состоит в следующем:

1. Перенос локальных переменных в глобальную область видимости;
2. Использование и изменение перенесенных переменных в разных функциях.

В общем случае нельзя изменять значения переменных, вынесенных из других функций в произвольном месте программы, так как это может привести к неправильному выполнению компилируемой программы. Поэтому строится граф вызовов для всех функций в модуле (рис. 5), анализируя который для каждой функции вычисляется множество переменных модификация которых, не нарушит работоспособность программы. Такими переменными будут переменные, вынесенные из функций, расположенных на разных путях в дереве вызовов. После формирования множеств подходящих переменных, осуществляется добавление мусорного кода, использующего для вычислений «безопасные» переменные. Также найденные переменные используются в предикатах. Функции, передаваемые по адресу в другие функции, не обрабатываются, так как они могут использоваться в многопоточном коде и обращение к одной глобальной переменной может вызвать сбой в работе программы.

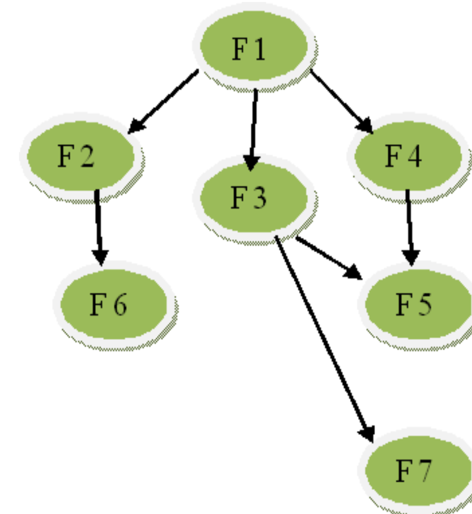

Рис.6. Пример дерева вызовов.

2.8. Маскирующее преобразование, осуществляющее разбиение целочисленных констант

Часто в коде в явном виде встречаются константы характерные для определенных алгоритмов, например константа 0x67452301 для MD5. Поиск констант позволяет определить используемый алгоритм, что упрощает анализ программы. Для противодействия предложен алгоритм разбиения констант. Разбиваются только константы больше единицы. Для разбиения случайным образом выбирается число меньшее исходного, которое будет выступать в качестве первого слагаемого, второе слагаемое получается автоматически.

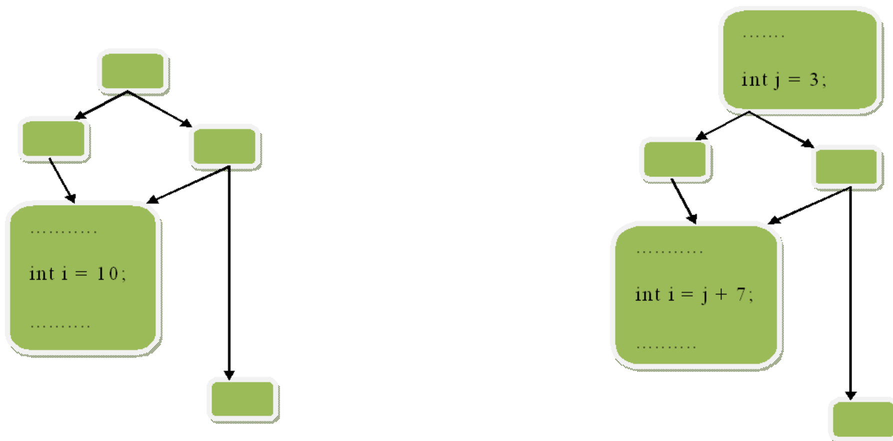

Рис.7. Пример работы.

2.9. Маскирующее преобразование, осуществляющее размножение тел функций

Для каждого использования функции внутри программного модуля, производится создание своего экземпляра вызываемой функции.

Например:

```
#include<stdio.h>
int sum (int a, int b) {
return a+b;
}
int main () {
printf(“Sum 2 + 2 = %d \n“, sum (2, 2) );
printf(“Sum 2 + 3 = %d \n“, sum (2, 3) );
return 0;
}
```


Для каждого вызова функции **sum** будет сгенерирована копия тела функции, затем каждый вызов оригинальной функции будет исправлен на вызов соответствующей копии. Такое преобразование увеличивает размер кода программы и время, требуемое для его автоматического анализа. Также, будучи применено совместно с другими запутывающими преобразованиями, зависящими от генератора случайных чисел, они утратят полную идентичность, что повысит сложность анализа, так как потребуется проанализировать каждую копию.

2.10. Маскирующее преобразование, осуществляющее вставку ложных циклов

В коде запутываемой программы происходит поиск участков кода, по структуре напоминающих одну итерацию цикла. Далее в начало участка или в его конец (в зависимости от разновидности вставляемого цикла) вставляется базовый блок с условным переходом в противоположный конец выделенного участка кода. Условный переход содержит в себе непрозрачный предикат, который и маскирует лишь одно исполнение цикла. В качестве подходящего участка кода рассматривается участок с одним входом и выходом.

2.11. Формирование непрозрачных предикатов

Разработано API для использования непрозрачных предикатов в запутывающих преобразованиях. Предикатом является базовый блок или несколько базовых блоков, имеющих один общий терминальный базовый блок. Терминальный базовый блок предиката заканчивается инструкцией условного перехода, которая всегда передает управление только по одной ветке. Причем, основываясь на информации, доступной на этапе компиляции, известно, по какому пути произойдет переход.

Реализовано три типа непрозрачных предикатов:

1. Истинность диофантова уравнения $x ^ 2 - n * y ^ 2 = 1$. Если параметр n не является точным квадратом, то это уравнение Пелля. При вставке этого предиката, случайным образом выбирается, будет ли он всегда иметь истинное значение, либо ложное. Так, как на этапе компиляции известно, некоторое количество решений данного уравнения, а также для константных значений переменных можно легко проверить истинность уравнения, то можно подобрать значения переменных для выбранного значения предиката.

2. Предикат, основанный на модульной арифметике: $(x ^ 3 - x) \bmod 3 = 0$. Это выражение всегда истинно. Значение переменной x для вычисления значения предиката выбирается случайным образом среди целочисленных глобальных переменных. Если таких глобальных переменных нет, то для вычисления предиката используется случайная целочисленная константа.

3. Целочисленное уравнение: $7 * y ^ 2 - 1 = x ^ 2$. Это выражение всегда ложно. По аналогии с предыдущим предикатом значения переменных x и y

случайным образом выбираются среди глобальных целочисленных переменных.

При необходимости использовать непрозрачный предикат при запутывании программы, из предложенных предикатов выбирается один случайный.

3. Оценка понижения быстродействия и потребляемой памяти

Произведена оценка увеличения объема и уменьшения быстродействия запутанной подпрограммы. Количество примененных преобразований зависит от входной программы, а замедление программы сильно зависит от быстродействия машины, на которой она будет исполняться, то представляется невозможным для произвольной программы, дать точную оценку замедления. Поэтому в целях практического использования используются коэффициенты, полученные опытным путем, с использованием достаточно большой кодовой базы.

В практических целях было произведен замер замедления на тестах из пакета OpenSSL 1.0.1. и посчитаны метрики на разных уровнях оптимизаций.

Замедление для одного запутывающего преобразования составляет 1,2-5,5 раз, потребление памяти увеличивается в 1,05-2,5 раз.

4. Оценка возможности восстановления информации об исходном коде

Компиляция программы - это процесс с потерями, поэтому точно восстановить исходный код программы на языке высокого уровня в общем случае невозможно. Обфускация программы накладывает дополнительные ограничения на процесс восстановления, так как перед восстановлением самого исходного кода, аналитику следует восстановить граф потока управления программы, расшифровать зашифрованные константы и данные, избавиться от мертвого кода, и провести другие преобразования, обратные запутывающим.

Стоит отметить, что отладка обфусцированного кода представляет собой сложную задачу даже для автора запутывающего преобразования, при наличии исходного кода защищаемого приложения и полной информации о трансформации, произошедшей с программой.

Совместное применение нескольких опций позволит увеличить сложность пропорционально произведению увеличения сложностей каждого преобразования в отдельности. Для примерной оценки сложности анализа, был проведен эксперимент. К программе Sqlite были применены следующие преобразования переплетения функций, перевода переменных в глобальную область видимости, преобразования диспетчеризации и сокрытия вызовов функций. Размер кода приложения увеличился с 2.9 МБ до 15 Мб.

Потребление памяти дизассемблером Ida Pro возросло в ~10 раз. Во время обработки защищенного файла возникло исключение в одной из библиотек Ida Pro, время анализа по сравнению с оригинальным кодом возросло примерно в 10 раз, затем произошло исключение в библиотеках дизассемблера. Также было произведено исследование с помощью инструмента комбинированного анализа, полученные результаты [10] свидетельствуют о том, что обеспечиваемый уровень защиты сравним с уровнем, обеспечиваемым коммерческими разработками.

5. Заключение

В данной статье было приведено описание маскирующих преобразований, реализованных во время разработки обфусцирующего компилятора в ИСП РАН, приведена оценка понижения быстродействия и увеличения объема потребляемой приложением памяти, а также оценка возможности восстановления информации об исходном коде.

Учитывая увеличение сложности при применении маскирующих преобразований, можно сделать вывод, что задача декомпиляции и полного статического анализа потребует намного больше ресурсов, чем декомпиляция и анализ незапутанной программы. Это позволяет говорить о том, что разработанные преобразования обеспечивают хороший уровень защиты.

Список литературы

[1]. Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization.— Master’s thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, 61 pages

[2]. Д. А. Щелкунов. Применение запутывающих преобразований и полиморфных технологий для автоматической защиты исполняемых файлов от исследования и модификации. Труды международной конференции РусКрипто. Апрель 2008 г.

[3]. А.В. Чернов. Анализ запутывающих преобразований программ. Труды ИСП РАН, том 3, 2002, стр. 7-38.

[4]. Chenxi Wang, Jonathan Hill, John Knight, and Jack Davidson. 2000. Software Tamper Resistance: Obstructing Static Analysis of Programs. Technical Report. University of Virginia, Charlottesville, VA, USA., 18 pages

[5]. И.Н. Ледовских, М.Г. Бакулин. Подход к восстановлению потока управления запутанной программы. Труды ИСП РАН, том 22, 2012, стр. 153-168. DOI: 10.15514/ISPRAS-2012-22-10.

[6]. Н.П. Варновский, А.В. Шокуров. Гомоморфное шифрование. Труды ИСП РАН, том 12, 2006, стр. 27-36.

[7]. Christian Collberg. Jasvir Nagra Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection. Addison-Wesley Professional. Pub. Date: July 24, 2009. Print ISBN-10: 0-321-54925-2.

[8]. Н.П. Варновский, В.А. Захаров, Н.Н. Кузюрин, А.В. Чернов, А.В. Шокуров. Об особенностях применения методов обфускации программ для информационной защиты микроэлектронных схем. Труды ИСП РАН, том 11, 2006, стр. 27-60.

[9]. А. В. Чернов. Об одном методе маскировки программ. Труды ИСП РАН, том 4, 2003, стр. 85-119.
[10]. М.Г. Бакулин, С.С. Гайсарян, Ш.Ф. Курмангалеев, И.Н. Ледовских, В.А. Падарян, С.М. Щевьева Динамический анализ обфусцированных приложений с диспетчеризацией или виртуализацией кода. Сдано в печать: Труды ИСП РАН, том 23, 2012, 17 стр. DOI: 10.15514/ISPRAS-2012-23-3.

Building an obfuscation compiler based on LLVM infrastructure

Kurmangaleev S.F. kursh@ispras.ru
ISP RAS, Moscow, Russia
Korchagin V. P. korchagin@ispras.ru
ISP RAS, Moscow, Russia
Savchenko V.V. sinmipt@ispras.ru
ISP RAS, Moscow, Russia
Sargsyan S.S. sevaksargsyan@ispras.ru
ISP RAS, Moscow, Russia

Annotation. The paper describes the obfuscating transformations, which were implemented while developing an LLVM-based obfuscating compiler in ISP RAS. The proposed transformations are based on well-known obfuscation algorithms and are specifically improved to resist better to static analysis deobfuscation techniques. The application performance decrease estimation and the increase of application memory consumption estimation are presented. Also, the possibility of source code information recovery is estimated. The implemented obfuscating transformations can be applied together to the given application to provide the strong protection from the static analysis deobfuscation attacks.

Keywords: LLVM, obfuscation

References

[1]. Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization.— Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, 61 pages
[2]. D. A. Shhelkunov. Primenenie zaputyvayushhikh preobrazovanij i polimorfnykh tekhnologij dlya avtomaticheskoj zashhity ispolnyaemykh fajlov ot issledovaniya i modifikatsii. [Applying obfuscation transformations and polymorphic technologies for automatic protection executable files from analysis and modification]. Trudy mezhdunarodnoj konferentsii RusKripto. [Proceedings of international conference RusCrypto]. April 2008 (in Russian).
[3]. A.V. Chernov. Analiz zaputyvayushhikh preobrazovanij programm. [Analysis obfuscating program transformations] Trudy ISP RAN [The Proceedings of ISP RAS], 2002, vol.3, pp. 7-38 (in Russian).
[4]. Chenxi Wang, Jonathan Hill, John Knight, and Jack Davidson. 2000. Software Tamper Resistance: Obstructing Static Analysis of Programs. Technical Report. University of Virginia, Charlottesville, VA, USA., 18 pages
[5]. Ilya N. Ledovskikh, Maxim G. Bakulin. Podkhod k vosstanovleniyu potoka upravleniya zaputannoj programmy. [An Approach to Reconstruction Control Flow of the 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.

[6]. N.P. Varnovskij, A.V. Shokurov. Gomomorfnoe shifrovanie. [Homomorphic encryption]. Trudy ISP RAN [The Proceedings of ISP RAS], 2007. Vol. 12, pp. 27-36. (in Russian).

[7]. Christian Collberg. Jasvir Nagra Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection. Addison-Wesley Professional. Pub. Date: July 24, 2009. Print ISBN-10: 0-321-54925-2.

[8]. N.P. Varnovskij, V.A. Zakharov, N.N. Kuzyurin, A.V. Chernov, A.V. Shokurov. Ob osobennostyakh primeneniya metodov obfuskatsii programm dlya informatsionnoj zashhity mikroehlektronnykh skhem. [About usage features of the program obfuscation techniques applying to informational microelectronic circuits security]. Trudy ISP RAN [The Proceedings of ISP RAS], 2006, vol. 11, pp. 27-60 (in Russian).

[9]. A.V. Chernov Ob odnom metode maskirovki programm [About one method program masking], Trudy ISP RAN [The Proceedings of ISP RAS], 2003, vol.4, pp. 85-119 (in Russian).

[10]. M.G. Bakulin, S.S. Gaissaryan, Sh.F. Kurmangaleev, I.N. Ledovskikh, V.A. Padaryan, S.M. Shchevyeva. Dinamicheskij analiz obfustsirovannykh prilozhenij s dispetcherizatsiej ili virtualizatsiej koda. [Dynamic analysis of virtualization- or dispatching-obfuscated applications]. Trudy ISP RAN [The Proceedings of ISP RAS], 2012,vol. 23, pp. 49-66. (in Russian). DOI: 10.15514/ISPRAS-2012-23-3.