Описание подхода к разработке обфусцирующего компилятора

Курмангалеев Ш. Ф., Корчагин В. П., Матевосян Р.А.,
kursh@ispras.ru,korchagin@ispras.ru, hripsime@ispras.ru

Аннотация. В данной статье приводится обзор запутывающих преобразований программ, сформулированы критерии эффективности методов обфускации. Также предлагается подход к реализации обфусцирующего компилятора на основе инфраструктуры LLVM. Особенность подхода заключается в одновременном применении преобразований, маскирующих различные аспекты работы запутываемого приложения, что обеспечивает стойкую защиту от статического анализа.

Ключевые слова: llvm; обфускация.

1. Введение

В настоящее время актуальна задача защиты программ, как от статического, так и от динамического анализа кода. Доступность качественных средств для анализа кода и большой выбор подключаемых модулей, в автоматическом режиме обходящих многие приемы противодействия анализу, понижают планку требований к квалификации аналитика. Что ведет к повышению требований к защите программ. Необходимо использовать либо методы противодействия анализу неизвестные широкому кругу лиц, либо использовать преобразования достаточно трудоемкие для анализа. Оптимальным выбором, позволяющим реализовать максимально стойкие варианты запутывания программ, является создание обфусцирующего компилятора[1] на базе одной из существующих компиляторных инфраструктур. С одной стороны это позволит производить запутывание программы, имея полную информацию о ней на всех этапах компиляции, а с другой позволит сосредоточиться на разработке защиты, а не на создании требуемой инфраструктуры. Для реализации прототипа была выбрана инфраструктура LLVM [2], позволяющая получать для программы промежуточные и бинарные коды, а также из промежуточного представления генерировать код на языке C.

2. Оценка эффектности методов обфускации

Поскольку известные методы защиты от тестирования отладчиками разного типа имеют ряд существенных недостатков, таких как платформозависимость, наличие средств автоматического обхода известных методов, было принято решение отказаться от реализации таких методов**.** В работах [3] и [4] рассматриваются критерии эффективности методов обфускации основанные на метриках сложности программ. Опишем используемые метрики:

Метрикой называется отображение, ставящее в соответствие каждой программе некоторое число.

Метрика LC размера процедуры [5] — простейшая метрика сложности кода. Чем больше процедура, тем выше априорная оценка её сложности. Размер каждой процедуры будет измерен в инструкциях промежуточного представления.

Метрика YC сложности циклической структуры определяется как мощность транзитивного замыкания отношения достижимости в графе потока управления процедуры.

Максимальная мощность транзитивного замыкания равна $(n-1)^2 = n^2 - 2n + 1$, где n — количество базовых блоков в графе потока управления. Из терминального базового блока не достижим ни один базовый блок, из начального базового блока достижимы все базовые блоки, кроме самого *начального базового блока*, а из любого оставшегося базового блока достижимы все базовые блоки, кроме *начального*.

Если в графе потока управления любая вершина достижима из *начального базового блока*, и *конечный базовый блок* достижим из любой вершины, то минимальная сложность циклической структуры графа из n вершин равна $3n - 6$. Эти два соотношения определяют диапазон, в котором может находиться значение этой метрики. Пусть M — мощность транзитивного замыкания для некоторой процедуры. Тогда для удобства

$$YC = \frac{M - (3n - 6)}{(n-1)^2 - (3n - 6)}$$

Следовательно, $0 < YC < 1$.

Метрика DC сложности потока данных процедуры [6] — позволяет оценить сложность зависимостей по данным в процедуре. Значение метрики DC

вычисляется как количество дуг в графе, достигающих определений процедуры.

Метрика MC усложнения программы при запутывании вычисляется по формуле:

$$MC = \frac{YC(\text{послезащиты})}{YC(\text{дозащиты})} + \frac{DC(\text{после})}{DC(\text{дозащиты})}$$

Чем больше метрика усложнения программы, тем сложнее для понимания становится замаскированная программа в силу увеличения числа информационных и управляющих связей. Так же Чернов приводит сводную таблицу метрики *MC* для различных методов обфускации (табл. 1).

Преобразование	*MC*
Открытая вставка процедур	2.63
Выделение процедур	2.79
Непрозрачные предикаты TTI	14.43
Недостижимый код	3.63
Мёртвый код	3.92
Дублирующий код	2.77
Внесение тождеств	4.85
Внесение несводимости	3.21
Переплетение процедур	1.45
Клонирование базовых блоков	4.89
Развёртка циклов	3.14
«Диспетчер»	6.14
Локализация переменных	2.71
Расширение области действия переменных	2.29
Повышение косвенности	13.00

Табл.1. *Сводная таблица характеристик маскирующих преобразований*

Видно, что наилучшими параметрами обладают преобразования, повышения косвенности, вставки непрозрачных предикатов, и введения диспетчера. Очевидно, что устойчивость к анализу возрастет, при применении нескольких запутывающих преобразований. Но также можно повысить сложность самих преобразований путем связывания потоков данных, вводимых преобразованием, и потоков данных исходной программы, и устранения слабых мест преобразований. Например, для переплетения процедур таким слабым местом является единственная точка диспетчеризации по дополнительному параметру для выбора рабочей процедуры.

3. Сводимость графа потока управления

В современных языках программирования содержится стандартный набор управляющих структур – if-then-else, for, while,do-while,switch и пр. Каждая из этих структур вносит в граф потока управления свой специфический, шаблонный подграф. Примеры таких подграфов для структур do-while и if-then-else показаны на рис. 1:

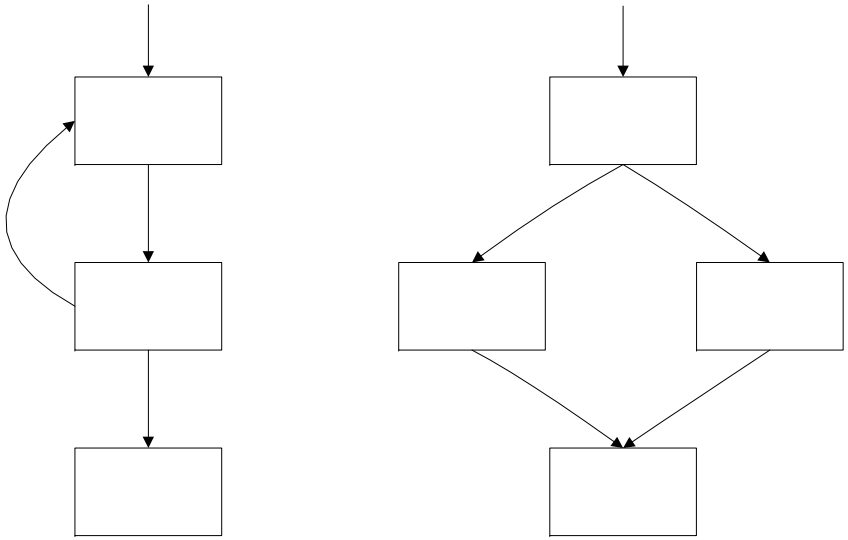

Рис. 1. Слева шаблон do-while, справа шаблон if-then-else.

Рассмотрим произвольный граф потока управления. Имея в наличии шаблоны для управляющих структур, имеется возможность применять их к графу следующим образом: в графе находится тот или иной шаблон, сворачивается в

один узел и вставляется вместо шаблона, к получившемуся графу также применяется свертка. Граф может выродиться в одну вершину, в этом случае будем говорить, что граф сводимый. В противном случае граф несводим. Пример несводимого участка показан на рис. 2 слева. Этот участок можно привести к сводимому, как показано на рис. 2 справа.

Приведение несводимого участка к сводимому увеличивает размер исходного графа потока управления.

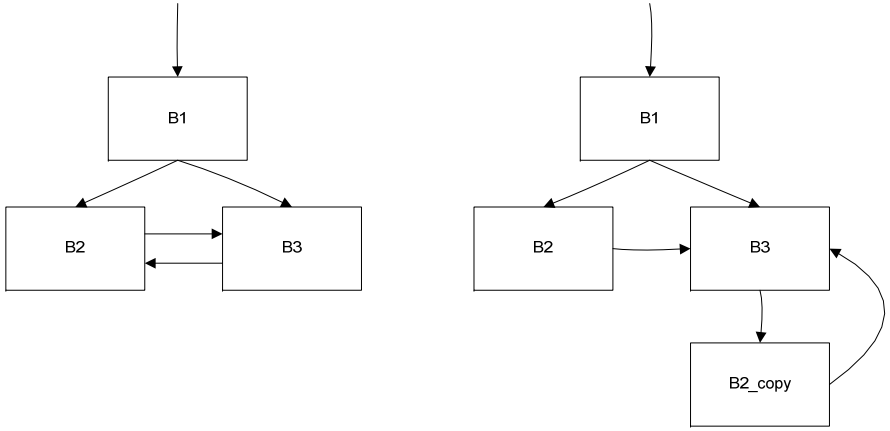

Рис. 2. Слева несводимый граф, справа сводимый вариант.

4. Декомпиляция. Проблемы декомпиляции

Декомпилятор – программа, которая воссоздает код на языке программирования высокого уровня, исходя из исполняемого модуля, который в свою очередь был сгенерирован компилятором. Таким образом, можно сказать, что декомпилятор является противоположностью компилятора. Как во время компиляции, так и во время декомпиляции код проходит несколько этапов анализа. Одним из таких этапов является генерация графа потока управления (CFG) и анализ потока управления на его основе. Для воссоздания управляющих структур высшего уровня декомпилятор ищет в CFG знакомые ему шаблоны (напр. рис. 1). При нахождении несводимых участков в графе, декомпилятор будет пытаться привести их к сводимым, как это сделалось с графом из рис. 2. В работе [7] указаны два сложно поддающихся анализу случая с циклами (рис. 3).

Даже если декомпилятору удастся сделать такое преобразование, воссоздание начальной версии кода станет невозможным.

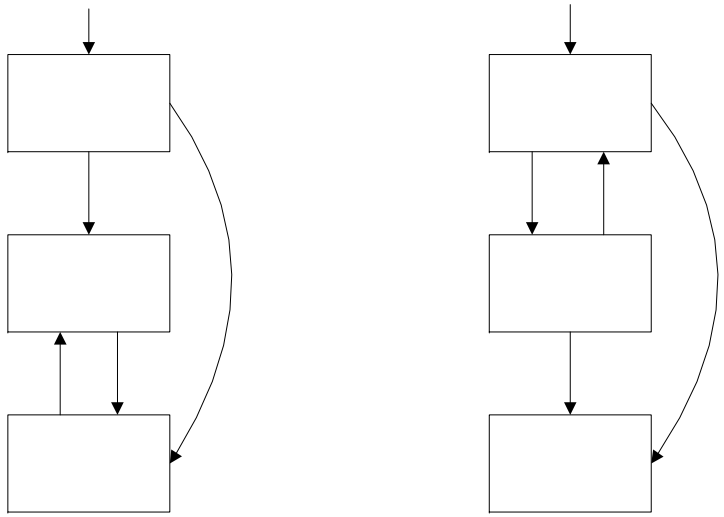

Рис. 3. Слева случай с несводимым графом, справа случай с ребром, выходящим из цикла.

Первый случай на рис. 3 идентичен треугольнику на рис. 2 слева. А на рис. 3 справа дело обстоит наоборот – ребро не входит в цикл, а выходит из него.

Таким образом, можно заключить, что генерация трудных для декомпилятора случаев, повысит стойкость защиты.

5. Определение критериев эффективности методов обфускации

В работе Щелкунова [5] предлагается ряд неформальных правил, которым должны удовлетворять запутывающие преобразования:

- Маскировать граф потока управления подпрограммы;
- Граф потока управления подпрограммы должен быть неприводимым;
- Определения фальшивых переменных не должны уничтожаться в конце подпрограммы;
- Определения фальшивых переменных не должны уничтожаться до того, пока эти переменные не были хоть раз использованы;
- «Мертвый» код не должен сильно отличаться от реально выполняемого кода. Все правила, приведенные выше, должны относиться, в том числе, и к нему.

Использование вышеприведенных правил позволит существенно усложнить процесс автоматической деобфускации подпрограммы. Более того, благодаря

использованию глобального контекста, достаточно сложно будет понять, что именно делает подпрограмма без детального анализа подпрограмм, которые с ней взаимодействуют.

Проведем сравнительный анализ преимуществ и недостатков методов обфускации по группам, поскольку все преобразования одной группы имеют сходные достоинства и недостатки:

- Маскировка текста. Преобразования такого типа не оказывают влияния на стойкость к анализу бинарного кода программы, поэтому в данной работе они не рассматриваются;
- Маскировка управляющей структуры. Поскольку преобразования затрагивают только граф потока управления, то при анализе потока данных зачастую появляется возможность обратить такое преобразование;
- Маскировка данных. Позволяет существенно усложнить анализ программы, поскольку усложняют анализ, как с помощью автоматических средств анализа, так и непосредственно аналитик не может сделать заключение о функционале, не обладая информацией о потоке данных касающегося анализируемого фрагмента. Но зачастую к себе привлекают внимание, повторяющиеся действия по шифрованию/дешифрованию или нехарактерная работа с данными. После восстановления схемы работы, анализ не представляет большой проблемы;
- Преобразования, затрагивающие и граф потока управления, и поток данных, компенсируют недостатки обоих подходов и позволяют добиться максимальной устойчивости. Но более трудоемки в разработке и реализации.

Можно сформулировать следующие 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;
- При разработке преобразования нужно учитывать особенности работы средств анализа, например для автоматических декомпиляторов следует насытить граф потока управления несводимыми участками.

6. Предлагаемые методы запутывания

На базе проведенных исследований разработаны методы усложнения программного кода с оценкой роста используемой памяти и оценкой падения быстродействия обфусцированного программного кода.

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.

Для реализации алгоритмов запутывания был выбран подход создания обфусцирующего компилятора на базе имеющихся решений с открытым кодом. Такой выбор обусловлен следующими причинами:

- Многие алгоритмы обфускации требуют наличия информации характерной для компиляторов;
- Встраивание защиты во время компиляции позволяет увеличить ее стойкость и скорость разработки защиты;
- Во время компиляции в расположении имеется максимальная информация о программе, в отличие от случая защиты исполняемого файла.

В настоящий момент параллельно проводятся работы по тестированию стойкости предложенных преобразований [8].

7. Требования к инфраструктуре

Сформулируем основные требования к компиляторной инфраструктуре. Компиляторная инфраструктура, на базе которой будет разрабатываться запутывающий компилятор должна удовлетворять следующему набору требований:

- Обеспечивать компиляцию исходных кодов на C/C++ под Windows и Linux;
- Иметь открытые исходные коды;
- Иметь документацию и поддержку сообщества;
- Расширяемость;
- Возможность влиять на генерируемый код на любом этапе компиляции, от препроцессора до кодогенерации;
- Возможность получить различную информацию об обрабатываемой программе на любой стадии компиляции, требуемую для реализации алгоритмов запутывания кода;
- Возможность, получить код на языке "C"из внутреннего представления компилятора.

- LLVM - компиляторная инфраструктура с открытыми исходными кодами. Перечислим ее основные особенности:
- Реализована на C++;
- Модульная и расширяемая архитектура;
- Имеет как статический, так и динамический компилятор.
- Поддерживает несколько фронтэндов:
- C, C++, Objective-C (Clang, GCC/dragonegg);
- Ruby (Rubinius, MacRuby);
- Python (unladen swallow);
- Поддерживает множество целевых архитектур: ARM, Alpha, Intel x86, Microblaze, MIPS, PowerPC, SPARC и т. д.

Промежуточное представление (LLVM IR) играет центральную роль в процессе компиляции. Все оптимизации реализованы как компиляторные проходы преобразования“LLVM IR to LLVM IR”.

Анализ кода, может быть реализован как отдельный проход, а его результаты могут разделять несколько проходов трансформирующих код.

Все машинно-зависимые оптимизации происходят в отдельном генераторе кода для каждой машины.

8. Заключение

В данной статье был проведен обзор запутывающих преобразований программ, были приведены критерии эффективности методов обфускации. Был предложен подход к реализации обфусцирующего компилятора.

Описанный в работе подход к построению обфусцирующего компилятора применяется при разработке обфусцирующего компилятора в ИСП РАН. Реализация представляет собой набор компиляторных проходов, запускаемых после работы стандартных оптимизирующих проходов компилятора LLVM. Особенности реализации разработанных алгоритмов будут раскрыты в последующих публикациях.

Список литературы

[1] Christian Collberg; JasvirNagra Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection Addison-Wesley Professional Pub. Date: July 24, 2009 Print ISBN-10: 0-321-54925-2, 748 стр.

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

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

[4] 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.

[5] M. H. Halstead. Elements of Software Science. Elsevier North-Holland, 1977.

[6] E. I. Oviedo. Control flow, data flow, and program complexity. In Proceedings of IEEE COMPSAC, 1980, pp. 146-152.

[7] Simon Moll. Decompilation of LLVM IR. B.Sc. Thesis, Saarland University, 2011.

[8] 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.