Обфускация потока управления программ с использованием LLVM в целях защиты интеллектуальной собственности.

# Введение

В последнее десятилетие все большую долю рынка программного обчеспечения стали занимать программные продукты, рапространяемые в промежуточном архитектурно-независимом коде или иначе в байт-коде. Это спровоцировало увеличение рисков получения и несанкционированного использования исходного кода из исполняемых модулей. Подбная проблема беспокоит большое количество комапний-разработчиков программного обеспечения, они заинтересованы в защите своей интелектуальной собственности и своих продуктов от несанкционированного реверс-инжиниринга. Хотя закон об авторском праве и препятствует нелицензионному использованию программного продукта, разработчики обеспокоены возможностью воровства проприетарных структур данных и алгоритмов. Несмотря на наличие некоторых методов защиты программных продуктов, таких как шифрование, выполнение на стороне сервера и использование машинного кода, обфускация является пожалуй самым дешевым и простым способом решения данной проблемы

**Обфускация** — приведение исходного текста или исполняемого кода программы к виду, сохраняющему ее функциональность, но затрудняющему анализ, понимание алгоритмов работы и модификацию при декомпиляции.

«Запутывание» кода может осуществляться на уровне алгоритма, исходного текста и/или ассемблерного текста. Для создания запутанного ассемблерного текста могут использоваться специализированные компиляторы, использующие неочевидные или недокументированные возможности среды исполнения программы. Существуют также специальные программы, производящие обфускацию, называемые **обфускаторами.**

* Затруднение декомпиляции проприетарных программ с целью предотвращения обратной разработки или обхода DRM и систем проверки лицензий.
* Нарушение авторских прав программистов и скрытие авторства.
* Генерация различных исполняемых модулей одной программы

Исходный текст:

int COUNT = 100;

float TAX\_RATE = 0.2;

for (int i=0; i<COUNT; i++)

{

tax[i] = orig\_price[i] \* TAX\_RATE;

price[i] = orig\_price[i] + tax[i];

}

1. Код после обфускации:
2. *for(int a=0;a<100;a++){b[a]=c[a]\*0.2;d[a]=c[a]+b[a];}*

# На уровне машинного кода

Как правило, обфускация на уровне машинного кода уменьшает скорость выполнения и соответственно увеличивает время выполнения программы. Поэтому она применяется в критичных к безопасности, но не критичных к скорости местах программы, таких как проверка регистрационного кода.

Простейший способ обфускации машинного кода — вставка в него недействующих конструкций (таких как or ax, ax)

На уровне промежуточного кода

В отличие от обычных языков, таких как C++ и Паскаль, компилирующих в машинный код, язык Java, NetP и языки платформы .NET компилируют исходный код в промежуточный код (байт-код), который содержит достаточно информации для адекватного восстановления исходного кода. По этой причине, для этих языков применяется обфускация промежуточного кода.

Усложнение исследования кода

Как было сказано выше, декомпиляция программ Java и .NET достаточно проста. В этом случае обфускатор оказывает неоценимую помощь тем, кто хочет скрыть свой код от посторонних глаз. Зачастую после обфускации декомпилированный код повторно не компилируется.

Обфускация HTML помогает спамерам: на почтовом клиенте, который способен отображать HTML, текст читается, но антиспам-фильтр, который имеет дело с исходным HTML-файлом, пропускает нежелательное сообщение, не распознавая в нём запретной строки.

Простейший пример обфусцированного HTML:

<b>Маш</b><b>ина</b>

При просмотре пользователь увидит слово «**Машина**», в то время как в исходном коде оно расчленено и воспринимается как два раздельных слова.

Оптимизация

В интерпретируемых языках обфусцированный код занимает меньше места, чем исходный, и зачастую выполняется быстрее, чем исходный. Современные обфускаторы также заменяют константы числами, оптимизируют код инициализации массивов, и выполняют другую оптимизацию, которую на уровне исходного текста провести проблематично или невозможно.

Проблема уменьшения размера важна, например, при программировании для сотовых телефонов на J2ME, где размер программы серьёзно ограничен. Обфускация JavaScript уменьшает размер HTML-файлов и, соответственно, ускоряет загрузку.

Недостатки

Потеря гибкости кода

Код после обфускации может стать более зависимым от платформы или компилятора.

Трудности отладки

Обфускатор не даёт постороннему выяснить, что делает код, но и не даёт разработчику отлаживать его. При отладке приходится отключать обфускатор.

Недостаточная безопасность

Хотя обфускация помогает сделать распределённую систему более безопасной, не стоит ограничиваться *только* ею. Обфускация — это безопасность через неясность. Ни один из существующих обфускаторов не гарантирует сложности декомпиляции и не обеспечивает безопасности на уровне современных криптографических схем. Вполне вероятно, что эффективная защита невозможна (по крайней мере в некотором конкретном классе решаемых задач).

Ошибки в обфускаторах

Современный обфускатор — сложный программный комплекс. Зачастую в обфускаторы, несмотря на тщательное проектирование и [тестирование](https://ru.wikipedia.org/wiki/Тестирование), вкрадываются ошибки. Так что есть ненулевая вероятность, что прошедший через обфускатор код вообще не будет работать. И чем сложнее разрабатываемая программа, тем больше эта вероятность.

Вызов класса по имени

Большинство языков с промежуточным кодом могут создавать или вызывать объекты по именам их классов. Современные обфускаторы позволяют сохранить указанные классы от переименования, однако подобные ограничения сокращают гибкость программ.

Существующие подходы к обфускации

# CFG flattering

Преобразование графа исполнения из направленной структуры в зацикленную.

# Opaque predicates (Формирование непрозрачных предикатов)

# 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. Переходы могу быть реализованы следующим образом.

# 1. Выражения, которые могут быть как истинны, так и ложны в зависимости от выбранных параметров, например, проверка истинности диофантова уравнения . 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.

# 2. Выражения, которые всегда истинны, например, уравнение . 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. Выражения, которые всегда ложны, например, целочисленное уравнение . Значения переменных x и y выбираются так же, как и ранее.

# Junk code

“Замусоривание” кода инструкциями не влияющими на логику программы с целью увелечения объема кода.

# Constant unfolding

Вычисление известных констант во время исполнения программы с целью затруднения статического анализа.

# Pattern-based obfuscation

Замена инструкций на их аналоги/последовательность инструкций выполняющих аналогичные действия.

# Uncommon instructions

Использование специализированных(и редко используемых) инструкций для замены инструкций защищаемого кода.

# Virtual Machine

Преобразование защищаемого кода в код абстрактного процессора и выполнение данного кода на эмуляторе такого процессора.

# Похожие работы

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, а с другой позволит сосредоточиться на разработке защиты, а не на создании требуемой инфраструктуры. Кроме того, такой подход обеспечивает поддержку нескольких архитектур при условии совпадения порядка байтов и минимального различия в ABI.

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 генерации кода;
* возможность получить различную информацию об обрабатываемой программе на любой стадии компиляции, требуемую для реализации алгоритмов запутывания кода.

LLVM [1] – компиляторная инфраструктура с открытыми исходными кодами, удовлетворяющая перечисленным требованиям и поддерживающая множество целевых архитектур (x86, ARM, MIPS, PowerPC и др.). Промежуточное представление (LLVM IR) машинно-независимого уровня играет центральную роль в процессе компиляции. Все оптимизации реализованы как компиляторные проходы преобразования LLVM IR. Анализ кода может быть реализован как отдельный проход, а его результаты могут разделять несколько проходов, трансформирующих код. Все машинно-зависимые оптимизации происходят отдельно для каждой машины на собственном внутреннем представлении низкого уровня.

Среди существующих проектов запутывающих компиляторов можно выделить два основных типа. Первые из них предоставляют возможность запутывания программ с применением широкого набора разных взаимодополняющих методов, а другие разработаны для реализации и проверки одного конкретного метода запутывания. Опишем самые известные примеры компиляторов обоих типов.

Совместный исследовательский проект HES-SO/RCSO под руководством Pascal Junod с названием “Obfuscator” [2]. Одна из его частей основана на LLVM и преобразует промежуточное представление LLVM, а другая производит преобразование бинарной программы для архитектуры ARM. На уровне промежуточного представления реализованы некоторые классические методы запутывания. Арифметические и логические инструкции заменяются на эквивалентные им выражения, состоящие из последовательности нескольких инструкций.

Граф потока управления усложняется путем вставки ложных ветвлений, закрытых непрозрачными предикатами. При этом базовый блок разбивается на две части и после первой части вставляется непрозрачный предикат. Навсегда выполняющуюся ветвь этого условия помещается вторая часть базового блока, а на ту, которая не выполняется никогда, помещается некоторый мусорный код. С помощью такой конструкции в программу вставляются несводимые участки графа потока управления. Такой подход отличается от обычного метода получения несводимых графов путем добавления дополнительного входа внутрь тела цикла, например, прикрытого непрозрачным предикатом.

Кроме того, реализовано преобразование диспетчера, трансформирующее граф потока управления программы. Эта реализация в целом соответствует подходу, описанному в [12], но имеет небольшие улучшения, касающиеся диспетчеризации управляющих структур (if-then-else, for-loops, switch). Они диспетчеризуются не как единое целое, а разделяются на свои составные части (например, ветви условного оператора), каждой из которых соответствует свое собственное значение переменной диспетчеризации. Оценки производительности приводятся на примере бенчмарка libtomcrypt.

Преобразование диспетчеризации увеличивает на 15% размер кода и на треть замедляет производительность.

Другой пример запутывающего компилятора общего назначения – это проектconfuse под руководством Chih-Fan Chen et al [3]. В нем каждый метод запутывания реализован как отдельный компиляторный проход, преобразующий промежуточное представление. Всего представлены три метода: обфускация строк, вставка излишнего кода, запутывание графа потока управления. Обфускация строк происходит в условных операторах, в которых сравниваются значение строковой переменной с константной строкой.

Константная строка заменяется на значение ее хэша и больше не содержится в программе, а вместо переменной подставляется вызов хэш-функции от этой переменной. Вставка излишнего кода основана на математических тождествах, позволяющих заменить какую-нибудь простую арифметическую операцию на длинную цепочку излишних вычислений. Комбинирование различной последовательности подобных тождественных замена арифметических выражений на эквивалентные используются для видоизменения кода.

Подход к запутыванию графа потока управления базируется на стандартном методе, описанным еще Коллбергом в [12]. Берется базовый блок и рассекается на две части, а между ними вставляется предикат. Если ставится непрозрачный предикат, то на его всегда исполнимую ветвь ставится вторая часть базового блока либо ее запутанная некоторой последовательностью предыдущих преобразований версия. Ветвь, которая никогда не выполняется, можно оставить пустой или же поместить на нее любой код. Если же ставится прозрачный предикат, то на обе его ветви помещается код, семантически эквивалентный коду из второй части рассеченного базового блока. Это код видоизменяется посредством применения к нему некоторого набора предыдущих оптимизаций. Кроме того, непрозрачные предикаты соединяются с условиями выхода из цикла для их запутывания.

Последним запутывающим компилятором общего назначения рассмотрим коммерческий morpher[4]. По заявлениям разработчиков, в нем реализовано самое большое количество запутывающих преобразований: зацепление дуг графа потока вызовов (CFG arches meshing), клонирование базовых блоков и функций, вставка непрозрачных предикатов, зацепление функций, вставка псевдоциклов длиной 1 в линейную последовательность инструкций, расшифровка и зашифровка констант до и после использования в программе.

Из второй группы запутывающих компиляторов наибольший интерес представляют те, в которых применяются нестандартные методы запутывания.

В статье [5] предлагается метод запутывания потока управления программы путем превращения ее в многопоточное приложение. Для передачи управления между потоками используется диспетчер, гарантирующий сохранение исходной семантики последовательной программы.

Предложенный метод реализован в рамках инфраструктуры LLVM. Он не зависит он других методов запутывания потока управления, поэтому его можно комбинировать с другими метода для увеличения сложности.

Интересный подход запутывания вредоносных программ реализован в работеTeja Tamboli [6]. На вход компилятору подается исходный код вредоносной программы вместе с исходным кодом какого-нибудь обычного приложения. Полученные биткоды обоих файлов смешиваются на этапе компоновки путем вставки функций из биткода обычной программы в биткод вредоносного приложения. На выходе получается бинарный код вредоносного приложения, похожий на код обычной программы.

В конце обратимся к методу, описанному в статье [7]. Описывается подход к автоматической защите тригерных вредоносных приложений от средств современного антивирусного анализа, так называемый метод обфускации условного кода. Запутыванию подвергаются условные ветвления с условием, удовлетворяющим некоторым условиям. Пример – сравнение строки, полученной приложением во время исполнения, с некоторым указанным в условии значением. Сравнение самих строк заменяется на сравнение их хэшей. Базовый блок находящийся внутри условия шифруется. В качестве ключа шифрования используется значение из условия. Кроме того, перед ним вставляется функция дешифрования, которая будет во время исполнения расшифровывать зашифрованный базовый блок. В качестве ключа будет использоваться значение, полученное во время исполнения. При реализации использовалась компиляторная инфрастуктура LLVM и утилита анализа модификации бинарного кода DynInst [8]. В промежуточном представлениеLLVM находились пригодные для запутывания участки программы. В логическом выражении условия производилась замена значений на их хэши, а внутрь условия вставлялась функция дешифрования и ключ шифрования со специальным маркером. После кодогенерации над бинарным кодом с помощью утилиты DynInst производилась шифровка и удаления из тела условия ключа и специального маркера.

**Еще работы. Проанализировать и дать краткое описание.**

1) Obfuscator-llvm (<https://github.com/obfuscator-llvm/obfuscator/wiki>)

2) Kryptonite ([http://download.tuxfamily.org/overclokblog/Obfuscation%20of%20steel%3a%20meet%20my%20Kryptonite/0vercl0k\_Obfuscation\_of\_steel\_meet\_kryptonite.pdf](http://download.tuxfamily.org/overclokblog/Obfuscation%20of%20steel%3A%20meet%20my%20Kryptonite/0vercl0k_Obfuscation_of_steel_meet_kryptonite.pdf))

3) CONFUSE (<http://www.cs.columbia.edu/~aho/cs4115_Spring-2013/lectures/13-05-16_Team11_Confuse_Paper.pdf>)

4) <http://www.ruscrypto.ru/resource/summary/rc2014/05_kurmangaleev.pdf>

5) [Deobfuscation: recovering an OLLVM-protected program](http://blog.quarkslab.com/deobfuscation-recovering-an-ollvm-protected-program.html)

(http://blog.quarkslab.com/deobfuscation-recovering-an-ollvm-protected-program.html)

# Руководство пользователя

# Пока изложу здесь мысли о том как пользоваться обфускатором:

# 1. Так как не имеет смысла тратить время на графический интерфейс, приложение будет консольным.

# 2. Исходный код (проект), который нужно обфусцировать нужно передавать в качестве параметра. В связи с этим, думаю стоит подумать, о рекурсивном обходе папки проекта и интерактивным подтверждением у пользователя стоит ли обфусцировать данный файл (опционально).

# 3. Так же можно указать несколько атрибутов, среди которых виды обфускации, которые нужно применить (если ничего не указано, то применять все). И для какой ОС компилировать бинарник.

4. С помощью –help можно посмотреть пример использования и все атрибуты.

5. Было бы неплохо выводить информацию о различиях с исполняемым файлом без обфускации (на сколько увеличился размер, время исполнения и т.д)

6. Нужно также написать установщик для обфускатора.

# Литература

1. The LLVM Compiler Infrastructure. <http://LLVM.org/>
2. Obfuscator reloaded, Application Security Forum – Western Switzerland, November 7th, 2012, Yverdon-les-Bains, Switzerland. <http://crypto.junod.info/obfuscatorwf12_talk.pdf>
3. Chih-Fan Chen, Theofilos Petsios, Marios Pomonis, Adrian Tang. Confuse: LLVMbased Code Obfuscation. <http://www.cs.columbia.edu/~aho/cs4115_Spring-2013/lectures/13-05-16_Team11_Confuse_Paper.pdf>
4. Обфускатор Morpher. <http://morpher.com/>
5. Rasha Salah Omar, Ahmed El-Mahdy, Erven Rohou, Thread-Based Obfuscation through Control-Flow Mangling, arXiv:1311.0044
6. Practical Reverse Engineering: x86, x64, ARM, Windows Kernel, Reversing Tools, and Obfuscation Bruce Dang, Alexandre Gazet, Elias Bachaalany, Sebastien Josse
7. Compilers: Principles, Techniques, and Tools. Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman
8. Tarek Ziadé. Expert Python Programming. - Published by Packt Publishing Ltd, 32 Lincoln Road, Birmingham. 2008. -372с