# Реализация запутывающих преобразований в компиляторной инфраструктуре LLVM

*Виктор Иванников <ivan@ispras.ru>,*
*Шамиль Курмангалеев <kursh@ispras.ru>,*
*Андрей Белеванцев <abel@ispras.ru>,*
*Алексей Нурмухаметов <oleshka@ispras.ru>,*
*Валерий Савченко <sinmipt@ispras.ru>,*
*Рипсиме Матевосян <hripsime@ispras.ru>,*
*Арутюн Аветисян <arut@ispras.ru>*

**Аннотация.** В статье описываются разработанные в ИСП РАН методы запутывания программ, направленные на противодействие методам статического анализа программ. Рассматриваемые методы запутывания реализованы в обфусцирующем компиляторе на базе LLVM. Приводится оценка замедления и увеличения объема потребляемой памяти.

**Ключевые слова:** обфускация, LLVM, непрозрачный предикат.

## *1. Введение*

В настоящее время актуальна задача защиты программ, как от статического, так и от динамического анализа кода. Доступность качественных средств 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 трудоемкие для анализа преобразования.

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, а также различающийся двоичный образ программы для каждого пользователя, если ввести зависимость заданного набора преобразований от некоторого уникального ключа.

При разработке преобразований необходимо учитывать 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 [9];
- При разработке преобразования нужно учитывать особенности работы средств анализа [10], например, для автоматических декомпиляторов следует насытить граф потока управления несводимыми участками.

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.

В настоящей статье в разделе 2 предлагается обзор существующих решений в области запутывающих компиляторов. Раздел 3 описывает разработанные

методы запутывания. Раздел 4 содержит экспериментальные результаты, а раздел 5 завершает статью.

## 2. Обзор существующих запутывающих компиляторов

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 [1] 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. На всегда 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 проект confuse 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 замен 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 работе Teja 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 и модификации бинарного кода DynInst [8]. В промежуточном представление LLVM 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.

## *3. Разработанные методы усложнения программного кода*

Были разработаны следующие методы усложнения:

- Перенос локальных переменных в глобальную область видимости;
- Шифрование 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;
- Создание несводимых участков в графе потока управления;
- Разбиение констант;
- Клонирование функций;
- Формирование непрозрачных предикатов.



Рассмотрим подробней предложенные методы.

### 3.1. Перенос локальных переменных в глобальную область видимости

Перенос локальных переменных в глобальную область видимости с последующим их использованием в разных функциях производится с целью затруднить точный анализ потоков данных в программе.

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, затем 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.

При восстановлении алгоритма работы программы используется построение слайсов программы. Выполняется отбор тех операторов программы, выполнение которых влияет на выходные данные или на выполнение которых повлияли входные данные. Во время статического анализа для переменных, расположенных в глобальной области памяти, требуется проводить межпроцедурный анализ.

### 3.2. Шифрование строк

Во время статического анализа строковые константы, хранящиеся в открытом виде, могут дать аналитику дополнительную информацию о функционировании программы 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.

Шифрование строк 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, чтобы во время

работы программы все строки не находились в памяти расшифрованными. Шифрование строк производится с помощью операции XOR со случайным ключом.

### 3.3. Вставка фиктивных циклов

Фиктивный цикл – цикл, в котором никогда не происходит более одной итерации. 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 типа фиктивного цикла) вставляется базовый блок с условным переходом в противоположный конец участка. Условный переход содержит в себе непрозрачный предикат, который и маскирует лишь одно исполнение цикла. В качестве подходящего участка рассматривается участок с одним входом и выходом.

### 3.4. Преобразование “диспетчер”

Идея маскирующего преобразования «диспетчер» заключается в преобразовании графа потока управления таким образом, что статический анализ переходов между базовыми блоками становится трудной задачей [11]. При этом базовым блокам присваиваются номера. В начало функции вставляют блок «диспетчер» – аналог switch в языке Си. В конец каждого блока дописывается код, устанавливающий номер следующего блока для выполнения и передающий управление на блок-диспетчер, в котором принимается решение, куда дальше передать управление.

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 её значение вычисляется по формуле `I = X1 XOR Z`; а следующее значение Z по формуле `Zслед = X2 XOR Zтекущее`; 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 XOR происходит сцепление переменной диспетчеризации. Такое преобразование затруднит автоматическое выделение переменной диспетчеризации, так как в ее вычисление будут вовлечены живые переменные, вычисляемые в программе.

### 3.5. Переплетение функций

Классический подход к переплетению функций обладает малой стойкостью. Он предполагает объединение сигнатур функций и наличие параметра, по которому происходит диспетчеризация [12]. 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. Тогда применение алгоритма обратного слайсинга [13] не позволяет 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 глобальные перемененные, а сама функция возвращает неиспользуемое значение. Если функции возвращают указатели, то тип возвращаемого значения переплетенной функции становится указателем на void.
3) В новой функции, полученной на основе переплетения двух функций, произвольно выбираются по одному блоку из каждой функции, затем над ними производится преобразование зацепления дуг [14]. 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 используется в качестве возвращаемого значения, а также записывается в глобальную переменную, что не позволит исключить добавленные вычисления как мертвый код, результат которого 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.

### 3.6. Сокрытие вызов функций

Преобразование применяется для маскировки вызовов функций, поскольку знание имени вызываемой функции облегчает восстановление алгоритма работы программы. Для маскируемого вызова создается функция-переходник, внутри которой содержится несколько вызовов функций. Аргументы в переходник передаются в измененном виде, после преобразования с помощью битовой операции XOR. 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. Чтобы избежать чрезмерной вложенности вызовов, переходники на переходники не создаются. Затем для каждой функции создается сортированный список ее аргументов. Для выбора функций, которые будут размещены внутри переходника, была введена мера "близости функций" – число от 0 до 1, которое показывает, насколько функции близки друг к другу по сигнатуре. 1 означает, что функции имеют набор аргументов с одинаковыми типами, 0 означает, что таких аргументов у функций нет. Значение меры – коэффициент Жаккара (Jaccard) для множеств типов аргументов двух функций:

$$\Gamma(f_1, f_2) = \frac{|TypeArgs(f_1) \cap TypeArgs(f_2)|}{|TypeArgs(f_1) \cup TypeArgs(f_2)|}$$

Половина функций в переходнике выбираются, как самые "похожие" по введенной мере, другая половина как "непохожие". После того, как функции были отобраны, для каждой из них производится попытка замены вызовов на переходник. Просматриваются все использования функции в пределах обрабатываемого модуля и отбираются те из них, которые являются непосредственным вызовом функции (call), либо вызовом с возможностью обработки исключений (invoke). Другие использования функции, например, её передача в качестве аргумента, пропускаются.

Функция-переходник принимает аргументами объединение аргументов всех функций внутри переходника, первый аргумент используется для получения истинных значений аргументов, а последний для диспетчеризации нужного вызова с помощью непрозрачного предиката. Аргументы, передаваемые в функцию-переходник, запутываются с помощью битовой операции XOR. Все аргументы преобразуются в тип данных длинной 64 бита (если аргумент имеет больший размер, то он передается как есть, без преобразования) и между всеми аргументами применяется операция XOR, обозначим результат операции за S. Затем к каждому аргументу применяется операция XOR с S, и в таком виде аргумент передается в функцию. Также для распутывания передается само значение S. Внутри функции-переходника происходит распутывание аргументов. Затем вычисляется непрозрачный предикат *P*, по результату которого происходит диспетчеризация вызова функций, основанный на китайской теореме об остатках. Вычисление предиката встраивается в функцию-переходник.

Диспетчеризация вызовов функций производится с помощью большого switch блока. Каждое значение в нем сгенерировано случайным образом и соответствует какой-либо функции. Каждой функции передаются те аргументы, которые соответствуют ей по типу. Все вызовы перемешиваются в



случайном порядке внутри функции-переходника. Для диспетчеризации используется последний аргумент функции-переходника, передаваемое в упомянутый выше непрозрачный предикат *P*: можно подобрать такое значение аргумента, которое соответствовало бы нужной функции. Это значение генерируется случайным образом.

Так как переходник включает в себя и похожие, и непохожие друг на друга функции, то часто количество его аргументов превышает количество аргументов, реально необходимых для вызова защищенной функции. Остальные аргументы берутся из глобальной области видимости. Если в глобальной области видимости переменных с таким типом нет, то они создаются. Внутри функции-переходника они могут быть использованы при вычислении непрозрачного предиката, а также для запутывания могут передаваться в другие вызовы, которые не будут использованы.

## *3.7.* 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.

Реализована вставка предиката двумя различными способами: после указанного базового блока или добавление инструкций по вычислению предиката в указанный базовый блок. После вставки предиката модифицируются φ-функции в базовых блоках, которые используются в терминальном условном переходе. φ -функция должна быть определена для каждого базового блока, являющегося предшественником данного. Для новых предшественников (базовых блоков предиката) φ-функция доопределяется нулевым значением.

Также реализованы интерфейсы для автоматической генерации предикатов. В запутывающих преобразованиях используются три типа предикатов:

1. Выражения, которые могут быть как истинны, так и ложны в зависимости от выбранных параметров, например, проверка истинности 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.
2. Выражения, которые всегда истинны, например, уравнение $(x^3 - x)\ mod\ 3 = 0$. 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 $7 * y^2 - 1 = x^2$. Значения переменных x и y выбираются так же, как и ранее.

### 3.8. Другие преобразования

В данном разделе описываются преобразования генерации несводимых участков графа потока управления, разбиение констант, клонирование функций и внесение зависимости от ключа пользователя.

Генерация несводимых участков в графе потока управления применяется для затруднения работы автоматических декомпиляторов. Колберг [12] описывает алгоритм, который приводит граф потока управления к несводимому. Для каждого цикла добавляется «фиктивное» ребро из заголовка цикла в его тело. Добавление такого ребра осуществляется с помощью вставки непрозрачных предикатов.

Предложена модификация упомянутого алгоритма: для всех циклов функции добавляются «фиктивные» ребра из одного цикла в другой. Недостаток такой трансформации состоит в том, что она эффективно запутывает только код функций, содержащих несколько циклов. Поэтому дополнительно производится следующая трансформация: для множества блоков функции выбирается N блоков и между ними случайно добавляются ребра. Фиктивные переходы защищаются непрозрачными предикатами.

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, бóльшие единицы. Для разбиения случайным образом выбирается число меньшее исходного, которое будет выступать в качестве первого слагаемого, второе слагаемое получается автоматически.

При клонировании функций для каждого использования функции внутри программного модуля производится создание своего экземпляра вызываемой функции. Для каждого вызова будет сгенерирована копия тела функции, затем этот вызов будет исправлен на вызов соответствующей копии. Такое преобразование увеличивает размер кода программы и время, требуемое для его автоматического анализа. После применения клонирования совместно с другими запутывающими преобразованиями, зависящими от генератора случайных чисел, функции утратят полную идентичность, что повысит сложность анализа, так как потребуется проанализировать каждую копию.



Чтобы обеспечить одинаковую работу на всех поддерживаемых платформах без зависимости от библиотечной реализации генератора случайных чисел, была реализована поддержка собственной версии генератора. Реализация использует линейный конгруэнтный генератор: при заданном стартовом числе $X_0$ следующее определяется по формуле: $X_{n+1} = (A * X_n + C) \bmod M$. В качестве параметров генератора выбраны значения, используемые в библиотеке `glibc`. Вместо реализации по умолчанию возможно использование любого другого алгоритма.

Ключ пользователя выступает в роли «затравки» для генератора, позволяя управлять недетерминизмом в преобразованиях, накладывая на программу уникальный для каждого пользователя характер изменения программы.

## *4. Экспериментальные результаты*

Произведем оценку увеличения объема и уменьшения быстродействия запутанной подпрограммы. Во время обфускации на уровне промежуточного представления кода генерируется количество дополнительных инструкций, например, инструкция `call` внутреннего представления LLVM разворачивается в несколько инструкций, одна из которых – это непосредственно вызов функции, а остальные – это инструкции передачи аргументов и обработки возвращаемого значения. Помимо этого, оптимизатор LLVM в зависимости от кода программы и набора оптимизаций может генерировать различный бинарный код для одних и тех же инструкций промежуточного представления. Кроме того, количество примененных преобразований зависит от входной программы (например, если в программе нет циклов, то и переплетать нечего). Поэтому представляется затруднительным дать точную оценку замедления для произвольной программы. В целях практического использования используются коэффициенты, полученные опытным путем с использованием достаточно большой базы программ.

В практических целях был произведен замер замедления на тестах из пакета OpenSSL 1.0.1 (таблица 1).

*Таблица 1. Параметры замедления и увеличения потребления памяти*

| **Метод** | **Замедление программы** | **Увеличение потребления памяти** |
|---|---|---|
| Клонирование функций | 1,20 | 1,10 |
| Переплетение функций | 1,20 | 1,10 |
| Шифрование строк | 5,00 | 1,05 |
| Вставка фиктивных циклов | 1,20 | 1,10 |
| Разбиение констант | 1,20 | 1,05 |
| Сокрытие вывозов внешних функций | 5,00 | 1,50 |
| Сокрытие вывозов всех функций | 8,00 | 1,70 |
| Диспетчер | 5,50 | 2,50 |
| Генерация несводимых участков в графе потока управления | 1,20 | 1,05 |
| Перенос локальных переменных в глобальную область видимости | 1,20 | 1,05 |

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 преобразования: переплетение функций, перенос локальных переменных в глобальную область видимости, преобразование «Диспетчер», сокрытие вызовов функций. Размер кода приложения увеличился с 2.9 МБ до 15 Мб. Потребление памяти дизассемблером Ida Pro возросло в ~10 раз, время анализа по сравнению с оригинальным кодом возросло примерно в 10 раз. Кроме того, некоторые версии Ida Pro оказались неспособны закончить анализ, поскольку во время работы возникает исключение в одной из библиотек, и программа аварийно завершает работу. Следует отметить, что декомпилятор Hex-Rays с задачей также не справился.

Также было произведено исследование с помощью инструмента комбинированного анализа TrEx [15]. Полученные результаты демонстрируют, что обеспечиваемый уровень защиты сравним с уровнем, обеспечиваемым коммерческими разработками.



## 5. Заключение

В статье описана реализация запутывающего компилятора на базе инфраструктуры LLVM. Проведен обзор аналогов созданного компилятора. Предложен и реализован набор как новых, так и известных методов запутывания. Приведены результаты тестирования средствами статического и динамического анализа.

**Список литературы**


[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: LLVM-based 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] Tamboli, Teja, "Metamorphic Code Generation from LLVM IR Bytecode" (2013). Master's Projects. http://scholarworks.sjsu.edu/etd_projects/301/

[7] Monirul Sharif, Andrea Lanzi, Jonathon Giffin, and Wenke Lee. Impeding malware analysis using conditional code obfuscation. Informatica, 2008.

[8] Инструмент Dyninst. http://www.dyninst.org/dyninst

[9] 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.

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

[11] 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

[12] C. Collberg, C. Thomborson, D. Low. A Taxonomy of Obfuscating Transformations. Departament of Computer Science, the University of Auckland, 1997. URL: http://www.cs.arizona.edu/~collberg/Research/Publications/CollbergThomborsonLow97a

[13] Frank Tip. "A survey of program slicing techniques". Journal of Programming Languages, Volume 3, Issue 3, pages 121–189, September 1995.

[14] А. В. Чернов. Об одном методе маскировки программ. Труды Института системного программирования РАН, том 4, 2003, стр. 85-119.

[15] 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. Труды Института системного программирования РАН, том 23, 2012, стр. 49-66.2008.

# Implementing Obfuscating Transformations in the LLVM Compiler Infrastructure

*Victor Ivannikov <ivan@ispras.ru>*
*Shamil Kurmangaleev <kursh@ispras.ru>*
*Andrey Belevantsev <abel@ispras.ru>*
*Alexey Nurmukhametov <oleshka@ispras.ru>*
*Valery Savchenko <sinmipt@ispras.ru>*
*Hripsime Matevosyan <hripsime@ispras.ru>*
*Arutyun Avetisyan <arut@ispras.ru>*

**Abstract.** The paper describes the methods for obfuscating C/C++ programs to prevent applying static analyzers to them. The methods are implemented within the well-known LLVM compiler infrastructure. Experimental results presenting resulting program slowdown and used memory growth are given.

**Keywords:** Obfuscation, LLVM, opaque predicates.