MODELING AND ANALYSIS OF INFORMATION SYSTEMS, VOL. 28, NO. 1, 2021
journal homepage: www.mais-journal.ru

COMPUTER SYSTEM ORGANIZATION

On Characteristics of Symbolic Execution in the Problem of Assessing the Quality of Obfuscating Transformations

P. D. Borisov[1], Y. V. Kosolapov[1]

DOI: 10.18255/1818-1015-2021-1-38-51

[1]Southern Federal University, 8a Milchakova str., Rostov-on-Don 344090, Russia.

MSC2020: 68N20
Research article
Full text in Russian

Received February 20, 2021
After revision March 10, 2021
Accepted March 12, 2021

Obfuscation is used to protect programs from analysis and reverse engineering. There are theoretically effective and resistant obfuscation methods, but most of them are not implemented in practice yet. The main reasons are large overhead for the execution of obfuscated code and the limitation of application only to a specific class of programs. On the other hand, a large number of obfuscation methods have been developed that are applied in practice. The existing approaches to the assessment of such obfuscation methods are based mainly on the static characteristics of programs. Therefore, the comprehensive (taking into account the dynamic characteristics of programs) justification of their effectiveness and resistance is a relevant task. It seems that such a justification can be made using machine learning methods, based on feature vectors that describe both static and dynamic characteristics of programs. In this paper, it is proposed to build such a vector on the basis of characteristics of two compared programs: the original and obfuscated, original and deobfuscated, obfuscated and deobfuscated. In order to obtain the dynamic characteristics of the program, a scheme based on a symbolic execution is constructed and presented in this paper. The choice of the symbolic execution is justified by the fact that such characteristics can describe the difficulty of comprehension of the program in dynamic analysis. The paper proposes two implementations of the scheme: extended and simplified. The extended scheme is closer to the process of analyzing a program by an analyst, since it includes the steps of disassembly and translation into intermediate code, while in the simplified scheme these steps are excluded. In order to identify the characteristics of symbolic execution that are suitable for assessing the effectiveness and resistance of obfuscation based on machine learning methods, experiments with the developed schemes were carried out. Based on the obtained results, a set of suitable characteristics is determined.

Keywords: obfuscation, symbolic execution, program similarity, program comprehension

INFORMATION ABOUT THE AUTHORS

Petr D. Borisov | orcid.org/0000-0002-8919-8310. E-mail: borisovpetr@mail.ru
postgraduate student.

Yury V. Kosolapov
correspondence author | orcid.org/0000-0002-1491-524X. E-mail: itaim@mail.ru
PhD.

For citation: P. D. Borisov and Y. V. Kosolapov, “On Characteristics of Symbolic Execution in the Problem of Assessing the Quality of Obfuscating Transformations”, *Modeling and analysis of information systems*, vol. 28, no. 1, pp. 38-51, 2021.

МОДЕЛИРОВАНИЕ И АНАЛИЗ ИНФОРМАЦИОННЫХ СИСТЕМ, ТОМ 28, № 1, 2021
сайт журнала: www.mais-journal.ru

COMPUTER SYSTEM ORGANIZATION

О характеристиках символьного исполнения в задаче оценки качества обфусцирующих преобразований

П. Д. Борисов[1], Ю. В. Косолапов[1]

DOI: 10.18255/1818-1015-2021-1-38-51

[1]Южный Федеральный Университет, ул. Мильчакова, 8а, г. Ростов-на-Дону, 344090 Россия.

УДК 517,9
Научная статья
Полный текст на русском языке

Получена 20 февраля 2021 г.
После доработки 10 марта 2021 г.
Принята к публикации 12 марта 2021 г.

Обфускация применяется для защиты программ от анализа и обратного проектирования. Несмотря на то, что в настоящее время существуют теоретически стойкие методы обфускации, эти методы пока не могут применяться на практике. В основном это связано либо с затратностью по ресурсам на исполнение обфусцированного кода, либо с ограничением на применение только к конкретному классу программ. С другой стороны, разработано большое количество методов обфускации, которые применяются на практике. Существующие подходы к оценке таких обфусцирующих преобразований в большей степени основаны на статических характеристиках программ. Однако актуальна задача комплексного (учитывающего и динамические характеристики программ) обоснования их эффективности и стойкости. Представляется, что такое обоснование может быть выполнено с помощью методов машинного обучения на основе векторов признаков, описывающих как статические, так и динамические характеристики программ. В настоящей работе такой вектор предлагается строить на основе характеристик пар сравниваемых программ: исходной и обфусцированной, исходной и деобфусцированной, обфусцированной и деобфусцированной. Для получения динамических характеристик программы в работе построена схема, основанная на символьном исполнении. Выбор символьного исполнения обосновывается тем, что такие характеристики могут описать сложность понимания программы при динамическом анализе. В работе предлагается две реализации схемы: расширенная и упрощенная. Расширенная схема приближена к процессу анализа программы аналитиком, так как включает в себя этапы дизассемблирования и трансляции в промежуточный код, в то время как в упрощенной схеме эти этапы исключены. С разработанными схемами проведены эксперименты с целью выявления характеристик символьного исполнения, подходящих для оценки эффективности и стойкости обфускации на основе методов машинного обучения. На основе полученных результатов определен набор подходящих характеристик.

Ключевые слова: обфускация, символьное исполнение, похожесть программ, понимание программ

ИНФОРМАЦИЯ ОБ АВТОРАХ

Петр Дмитриевич Борисов | orcid.org/0000-0002-8919-8310. E-mail: borisovpetr@mail.ru
аспирант.

Юрий Владимирович Косолапов
автор для корреспонденции | orcid.org/0000-0002-1491-524X. E-mail: itaim@mail.ru
канд. техн. наук.

Для цитирования: P. D. Borisov and Y. V. Kosolapov, “On Characteristics of Symbolic Execution in the Problem of Assessing the Quality of Obfuscating Transformations”, *Modeling and analysis of information systems*, vol. 28, no. 1, pp. 38-51, 2021.

Введение

Обфускация – это модификация программного кода с сохранением его функциональности, затрудняющая анализ, понимание алгоритмов программы и их модификацию. Обфускация широко используется для защиты программ от анализа и обратного проектирования [1]. Несмотря на то, что в настоящее время существуют теоретически стойкие методы обфускации [2], эти методы пока не могут применяться на практике. В основном это связано, либо с затратностью по ресурсам на исполнение обфусцированного кода, либо с ограничением на применение только к конкретному классу программ [3]. Разработано множество методов обфускации, которые на интуитивном уровне затрудняют понимание защищаемых обфускацией алгоритмов, но теоретического обоснования их эффективности нет. Тем не менее предлагается ряд практических способов и метрик для оценки эффективности обфусцирующих преобразований, т.е. стойкости к анализу и пониманию программ [4—8]. Отметим, что понимание исходного кода программы является широко исследуемой областью в программной инженерии [9]. В [10] отмечается, что понимание программы и запутывание кода – это две стороны одной медали, и поэтому метрики для оценки понимания строятся в [10] с использованием знаний из области обфускации. Можно предположить, что методы оценки понимания программы, в свою очередь, также могут быть использованы для оценки эффективности запутывающих преобразований.

В [11] предложена схема оценки стойкости запутывающих преобразований, основанная на сравнении признаков подобия, вычисленных по характеристикам программ. Основным блоком этой схемы является *блок оценки стойкости*, который делает вывод о сходстве программ. Этот блок может быть реализован с помощью методов машинного обучения. Для этого необходимо иметь характеристики программы, описывающие ее с разных сторон анализа: при статическом анализе (структура графа потока управления, полнота дизассемблирования, понятность кода программы и другие) и при динамическом анализе (поведение программы во время выполнения). Для схемы из [11] набор характеристик, предложенных в [4—8], а также характеристик из области понимания программ [9] может быть построен путем статического анализа программы. В то же время собрать характеристики программы с помощью динамического анализа сложнее, так как требуется запустить программу и проанализировать ее поведение, которое может зависеть от среды исполнения и/или входных параметров. В настоящей работе в качестве модели динамического анализа выбрано символьное исполнение [12], которое может характеризовать сложность понимания программы при динамическом анализе, и уже нашло применение в [13] при анализе обфусцированного кода.

Целью настоящей работы является, с одной стороны, получение и оценка характеристик символьного исполнения обфусцированной/деобфусцированной/исходной программ, с другой – оценка схемы получения характеристик символьного исполнения программ, построенной в [11]. Для оценки схемы рассматривается ее упрощенная версия, в которой отсутствуют шаги компиляции в бинарное представление и трансляции обратно в биткод LLVM [14]. Далее, для удобства упрощенную версию будем называть *упрощенной схемой*, а ее полную версию будем называть *расширенной схемой*.

Статья, кроме введения и заключения, содержит три раздела. Первый раздел посвящен обзору работ в области известных методов оценки обфусцирующих преобразований. Во втором разделе предлагаются реализации расширенной и упрощенной схем получения характеристик символьного исполнения программ. Третий раздел посвящен анализу результатов проведенных экспериментов с предложенными схемами.

1. Известные подходы к оценке обфусцирующих преобразований

Один из первых способов комплексной оценки обфусцирующих преобразований предложен К. Колбергом в [4]. Для этого предлагаются четыре индикатора: эффективность (*potency*), стойкость

(*resilience*), стоимость – степень увеличения потребляемых ресурсов обфусцированной программой (*cost*), качество обфускации (*quality*). Эффективность обфускации определяется с использованием метрик качества программ из программной инженерии, таких как длина программы, цикломатическая сложность, сложность потока и структур данных, а также других метрик. Стойкость определяется как функция от времени аналитика на разработку деобфускатора и времени работы самого деобфускатора. Качество обфускации определяется как комбинация трех предыдущих индикаторов: эффективности, стойкости и стоимости. Однако отметим, что в [4] не предлагается способ оценки времени необходимого аналитику для разработки деобфускатора.

В работе [5] предложен метод поиска и выявления зашифрованных данных в программе (в качестве таких данных могут выступать алгоритмы программы). Метод основан на использовании модели N-Gram [15]. С помощью данного метода вычисляется показатель *искусственности*, который используется для выявления участков программы, содержащих данные с большой энтропией. Представляется, что с помощью этого показателя можно оценить эффективность обфусцирующих преобразований, поскольку такие преобразования могут оказывать влияние на энтропию кода программы (как в меньшую, так и в большую сторону).

В работе [6] предложен иной подход: качество обфускации исходного кода оценивается по Колмогоровской сложности. Экспериментально установлено, что чем меньше сходство исходного кода и декомпилированного кода, тем выше Колмогоровская сложность для запутанной программы. Следовательно, чем выше Колмогоровская сложность (оцененная с помощью алгоритмов сжатия), тем лучше обфускация. Этот подход в [7] применяется для оценки запутанности программ, написанных на языке Java. При этом вычисление Колмогоровской сложности выполняется на основе файлов с исходом кодом программ.

Экспериментальный подход для оценки стойкости обфусцирующих преобразований описан в [8]. В этом подходе обфускация рассматривается с точки зрения понимания программного кода аналитиком. Для оценки стойкости была проведена серия контролируемых экспериментов с участием групп аналитиков. Показано, что статический и динамический анализ обфусцированных программ аналитиком занимает значительно больше времени, чем анализ исходной программы. Но этот метод не подходит для автоматического анализа стойкости обфускации.

Проблема автоматической оценки качества обфусцирующих преобразований, как с точки зрения статического анализа, так и с точки зрения динамического анализа, является актуальной. В рассмотренных выше работах, за исключением эксперементального подхода с участием аналитиков, эффективность рассчитывается на основе статического анализа. Для комплексной оценки эффективности необходимо учитывать динамические характеристики программ. Для получения таких характеристик в настоящей работе предлагается использовать символьное исполнение. Для того чтобы определить, какие характеристики символьного исполнения могут быть использованы для оценки эффективности обфускации, строится схема их получения, проводятся эксперименты, выполняется анализ полученных результатов. Отметим, что символьное исполнение находит применение в задачах исследования и анализа обфусцированного кода. В частности, в [13] отмечается, что обфусцирующие преобразования оказывают значительное влияние на эффективность символьного анализа, и предлагается обобщенный подход повышения эффективности такого анализа. Тем не менее в [13] символьное исполнение используется как способ анализа, без рассмотрения применимости в задачах оценки эффективности и стойкости обфускации.

2. Схемы получения характеристик

В [11] предложена схема нахождения характеристик символьного исполнения, которые предполагается использовать для оценки эффективности обфусцирующих преобразований. В этом разделе кратко описывается эта схема, отмечаются некоторые особенности, связанные с шагами трансляции машинного кода в биткод LLVM, а также описывается упрощенная схема.

2.1. Расширенная схема

В соответствии с моделью [11], программа P проходит следующие шаги: 1) компиляция с помощью обфусцирующего компилятора Hikari [16] с различными опциями обфусцирующих преобразований (выбраны 10 различных обфусцирующих преобразований), а также компиляция без применения преобразований; 2) построение графа потока выполнения скомпилированной программы с помощью инструмента mcsema-disass [17]; 3) трансляция полученного на предыдущем шаге представления в биткод LLVM с помощью инструмента mcsema-lift [17]; 4) оптимизация полученного биткода с помощью оптимизатора opt из состава LLVM; 5) символьное исполнение полученных версий биткода с помощью символьного интерпретатора KLEE [18]; 6) обработка полученных характеристик символьного исполнения. Последовательность шагов показана на рис. 1.

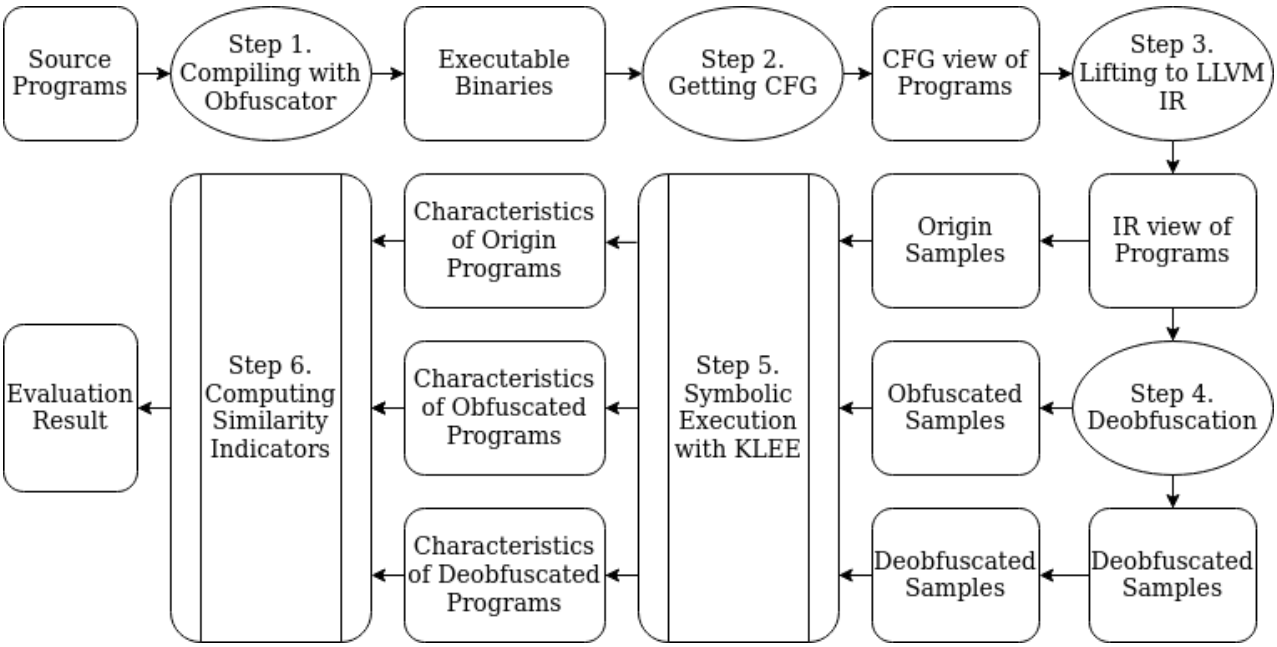

Fig. 1. An extended scheme for finding the characteristics of symbolic execution of programs

Рис. 1. Расширенная схема нахождения характеристик символьного исполнения программ

После первого шага для каждой программы P создается 11 различных *исполнимых* модулей: 10 обфусцированных и один *оригинальный* (без обфускации). В результате выполнения второго и третьего шага получается 11 различных файлов биткода, соответствующих исполнимым модулям. На четвертом шаге для каждого из 10 файлов биткода обфусцированных программ выполняется оптимизация, полученный результат сохраняется в отдельном файле биткода. Заметим, что оптимизатор, используемый на этом шаге, выполняет роль деобфускатора из модели [11], так как оптимизаторы обычно выполняют преобразования обратные к обфусцирующим [19]. Поэтому оптимизированные файлы биткода будем называть деобфусцированными. К началу пятого шага имеется 21 различный файл биткода: оригинальный, 10 обфусцированных, 10 деобфусцированных. Множество полученных файлов биткода, соответствующих программе P, обозначим $B(P)$. На пятом шаге выполняется символьное исполнение каждого файла биткода из $B(P)$.

2.2. Проблемы анализа восстановленного биткода в расширенной модели

Утилиты трансляции McSema, используемые в реализации расширенной схемы, разделяют процесс трансляции кода бинарной программы на два этапа. На первом этапе строится высокоуров-

невое представление программы – граф потока выполнения, содержащий функции, инструкции базовых блоков и другую необходимую информацию. Такая работа выполняется с помощью сторонних утилит, например IDA Pro [20], DynInst [21]. На втором этапе полученное представление транслируется в биткод LLVM внутренней утилитой McSema. Таким образом, обработка программы в представлении машинного кода и подготовка представления удобного для трансляции в большей степени лежит на утилитах бинарного анализа, применяемых на первом этапе. На втором этапе трансляция выполняется практически напрямую: каждая инструкция машинного кода отображается в инструкцию промежуточного представления. Рассмотрим особенности первого и второго этапов трансляции.

При трансляции программ из представления на языке программирования высокого уровня в более низкоуровневое (промежуточное представление компилятора или машинные инструкции, см. шаг 1 на рис. 1) теряется часть информации о программе (например, информация о типах переменных, именах функции, интерфейсах классов). Поэтому при дизассемблировании и декомпиляции, в частности, возникает проблема отличия исполнимого кода от данных [22]. С одной стороны, все исполнимые файлы программы обычно имеют определенный формат [23, 24], в котором, как минимум, прописывается, какие ее участки являются исполнимым кодом, какие данными, где располагается точка входа в программу. Поэтому формат исполнимых файлов частично помогает разрешить проблему определения исполнимого кода. С другой стороны, остаются проблемы с идентификацией адресов переходов при косвенной адресации, определением границ функций. Отметим, что утерянная при компиляции информация важна в задачах обратного проектирования, в частности, при трансляции кода из низкоуровнего представления (машинный код) в высокоуровневое (промежуточное представление компилятора/псевдокод/язык высокого уровня, см. шаги 2,3 рис. 1). Эта информация позволяет более точно и быстро проанализировать код программы [25]. Однако часто бинарные файлы распространяются без нее, поэтому из-за отсутствия полной информации декомпиляция остается трудной задачей.

Применяемая на втором этапе трансляция, на практике реализуется путем интерпретации. Интерпретация подразумевает, что машинные инструкции не напрямую транслируются в инструкции целевого процессора, а транслируются в байт-код так называемой виртуальной машины. При трансляции создается глобальная структура, описывающая целевой процессор (все его регистры, флаги и другая специфичная для целевой архитектуры информация). Машинные инструкции заменяются аналогичными инструкциями уровня промежуточного представления, в которое транслируется программа, но при этом взаимодействие уже выполняется не со структурами настоящего процессора, а со структурой виртуальной машины описывающей процессор [26]. По этой причине оптимизация (деобфускация) биткода такой программы может не принести существенной разницы с исходным кодом, так как оптимизатор с высокой вероятностью не найдет соответствующего преобразования для оптимизации такой структуры программы (другими словами оптимизатор исследует код виртуальной машины, а не код анализируемой программы).

С целью определения влияния промежуточных этапов трансляции на оценку стойкости обфусцирующих преобразований построена упрощенная схема.

2.3. Упрощенная схема

В упрощенной схеме исключены этапы 2 и 3 расширенной схемы, показанной на рис. 1. Таким образом программа транслируется из исходного кода напрямую в биткод LLVM. Обфусцирующие преобразования выполняются корректно, так как работают на уровне промежуточного представления. Далее программа анализируется символьным интерпретатором KLEE. Упрощенная схема анализа изображена на рис. 2.

Благодаря такой организации в упрощенной схеме сохраняется большая часть исходной информации о программе, которая теряется при многочисленных преобразованиях и может быть

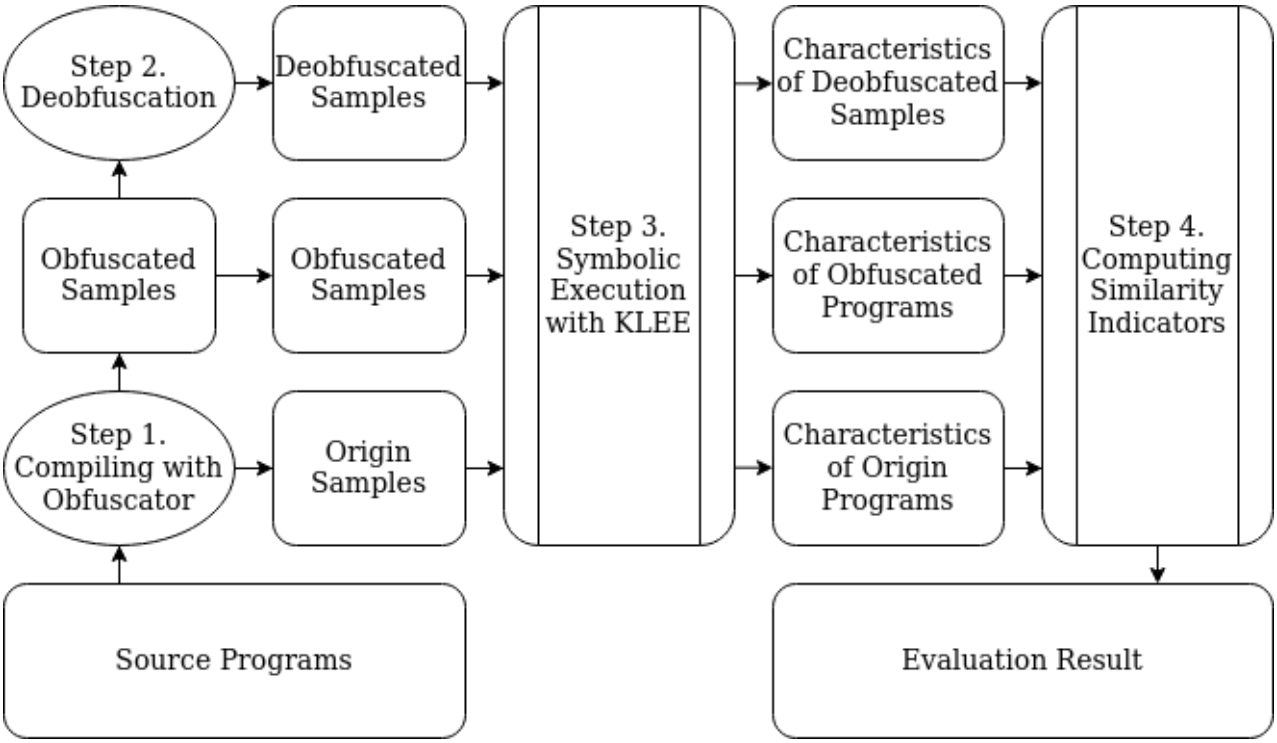

Fig. 2. An simplified scheme for finding the characteristics of symbolic execution of programs

Рис. 2. Упрощенная схема нахождения характеристик символьного исполнения программ

полезна во время ее анализа. Исключение этапов трансляции бинарного представления программы обратно в промежуточное представление LLVM (построение графа потока управления и трансляция полученного представления в биткод LLVM) также устраняет влияние усложненной структуры транслированной программы на работу деобфускатора.

3. Экспериментальное получение характеристик символьного исполнения

Для расширенной и упрощенной схем проведены эксперименты по нахождению характеристик символьного исполнения и вычислению показателей похожести, основанных на таких характеристиках. Эксперименты проводились на компьютере со следующими свойствами: процессор AMD Ryzen 2700U (4/8 ядер/потоков), объем оперативной памяти 16Gb, твердотельный накопитель SSD M.2 PCI-E NVMe 256Gb. Поскольку объем потребляемой оперативной памяти значительно увеличивается во время символьного выполнения, в дополнение к установленной памяти, был увеличен объем файла подкачки (swapfile) до 16Gb. Таким образом, общий объем памяти доступной символьному интерпретатору достиг 32Gb.

В следующих подразделах описываются используемые данные, параметры обфускации, искомые характеристики символьного исполнения, показатели похожести, а также описываются ограничения на эксперименты.

3.1. Описание данных

Для проведения экспериментального исследования была составлена выборка программ $\mathcal{P}$, написанных на языке C. За основу выборки был взят набор программ, использовавшийся в [27] для исследований влияния обфусцирующих преобразований на символьное исполнение. Все программы из $\mathcal{P}$ принимают на вход один параметр командной строки и обрабатывают его. Функционал программ включает: вычисление простых контрольных сумм, сортировку символов входного па-

раметра, поиск символа, проверку свойств заданного числа, преобразование входного параметра, в строку, в число различных систем исчисления, а также выполнение простой операции в зависимости от входного параметра. В случае успешного выполнения программа выводит на экран результат обработки параметра и возвращает 0, иначе возвращает код ошибки.

Большинство программ было изменено, т.к. в них обрабатывался лишь первый символ входного параметра, также добавлено несколько программ, реализующих простой алгоритм. Изначально выборка $\mathcal{P}$ состояла из 50-ти программ. Для ограничения времени, затраченного на символьный анализ $|\mathcal{P}| \cdot |B(P)|$ файлов биткода, было установлено максимальное время символьного исполнения равное 30 минутам. В результате применения схем, изображенных на рис. 1 и рис. 2, для программ из $\mathcal{P}$ было обнаружено, что некоторые программы не могут быть исследованы символьным интерпретатором в течение максимального указанного времени (30 минут). В этом случае время символьного исполнения обфусцированной и исходной программ совпадает и равно максимальному, а другие характеристики символьного исполнения не кажутся объективными, так как символьное исполнение не закончено.

Отметим, что существует также ограничение по объему оперативной памяти, установленной на вычислительном устройстве и доступной символьному интерпретатору. Несмотря на то, что этот объем оперативной памяти был увеличен за счет увеличения объема файла подкачки, остались программы, символьное исполнение которых досрочно завершалось операционной системой из-за того, что процесс символьного интерпретатора занимал всю доступную оперативную память. Такие программы также были исключены из выборки $\mathcal{P}$. Таким образом, в результате для оценки результатов эксперимента было отобрано 35 программ ($|\mathcal{P}| = 35$).

3.2. Параметры обфускации

Используемый обфусцирующий компилятор Hikari является дальнейшим развитием компилятора Obfuscator-LLVM, подробно описанного в [28], но предоставляет расширенный набор возможных преобразований кода программ на уровне промежуточного представления. Для выполнения эксперимента выбраны следующие обфусцирующие преобразования: O_{acd} – встраивание кода, препятствующего анализу структур классов (Anti-Class Dump), O_{sub} – замена инструкций эквивалентными (Substituion), O_{cff} – реализует обфусцирующее преобразование сглаживания графа потока управления программы [29] (Control Flow Flattening), O_{bcf} – встраивание непрозрачных предикатов с целью добавления ложных ветвлений и усложнения графа потока управления программы (Bogus Control Flow), O_{ind} – замена инструкций ветвления косвенными переходами (Indirect Branching), O_{fcw} – создание фиктивных функций-прокси, усложняющих анализ зависимостей между функциями (Function Call Wrapper), O_{fco} – обфускация инструкций вызова функций (Function Call Obfuscation), O_{sbb} – разбиение базовых блоков на семантически эквивалентную последовательность базовых блоков (Split Basic Block), O_{enc} – кодирование статических строк (String Encoding), O_{all} – примение всех обфусцирущих преобразований вместе (All Obfuscation Options). Множество всех обфусцирующих преобразований обозначим $\mathcal{O}$:

$$\mathcal{O} = \{O_{\text{acd}}, O_{\text{all}}, O_{\text{bcf}}, O_{\text{cff}}, O_{\text{enc}}, O_{\text{fco}}, O_{\text{fcw}}, O_{\text{ind}}, O_{\text{sbb}}, O_{\text{sub}}\}.$$

Некоторые преобразования предусматривают дополнительную параметризацию, например, указание вероятности применения к каждому базовому блоку. Для всех таких преобразований использовались параметры по умолчанию.

3.3. Характеристики символьного исполнения

Необходимым условием выбора характеристики является ее чувствительность к изменениям программы. Так как очевидно, что если характеристика не меняется при изменении программы (например, с помощью обфусцирующих преобразований), то по этой характеристике трудно

оценить влияние таких изменений на понимание программы. В качестве набора анализируемых характеристик символьного исполнения выбрано множество

$$\mathcal{F} = \{F_{\text{time}}, F_{\text{iexe}}, F_{\text{icov}}, F_{\text{bcov}}, F_{\text{ilen}}, F_{\text{tsmt}}, F_{\text{qsmt}}, F_{\text{blen}}, F_{\text{qall}}, F_{\text{mem}}\},$$

где F_{iexe} – количество исполненных инструкций в ходе анализа, F_{time} – время символьного исполнения, F_{icov} – процент покрытия инструкций в биткоде, F_{bcov} – процент покрытия инструкций перехода, F_{ilen} – общее количество инструкций в файле биткода, F_{tsmt} – общее время, затраченное решающим модулем SMT (Satisfiability Modulo Theories), F_{qsmt} – количество запросов к решающему модулю в среднем за одно исполнение, F_{blen} – количество операторов перехода в коде программы, F_{qall} – количество запросов к решающему модулю всего в ходе символьного анализа программы, F_{mem} – средний объем потребленной памяти в ходе символьного исполнения.

Для устранения влияния процессов операционной системы на результаты выполнения экспериментов, для каждой программы из $\mathcal{P}$ эксперимент выполняется t раз (на характеристики символьного исполнения могут повлиять такие процессы, как обновление компонент системы, фоновое исполнение служб, плановое выполнение задач). В настоящей работе параметр t равен 5. Для каждой характеристики $F(\in \mathcal{F})$ символом F^i обозначим значение этой характеристики после i-ой итерации алгоритма, а через $\overline{F}$ обозначим усредненное значение характеристики по всем t итерациям: $\overline{F} = (\sum_{i=1}^{t} F^i)t^{-1}$.

3.4. Показатели похожести программ

В [11] предложены показатели похожести программ для заданных характеристик. В настоящей работе для каждой характеристики $F(\in \mathcal{F})$, программы $P(\in \mathcal{P})$, обфусцирующего преобразования $O(\in \mathcal{O})$ вычисляются три меры похожести:

$$\delta^F(P, O(P)) = \frac{|\overline{F}(P) - \overline{F}(O(P))|}{\max\{\overline{F}(P), \overline{F}(O(P))\}}, \quad \delta^F(P, D(O(P))) = \frac{|\overline{F}(P) - \overline{F}(D(O(P)))|}{\max\{\overline{F}(P), \overline{F}(D(O(P)))\}},$$

$$\delta^F(O(P), D(O(P))) = \frac{|\overline{F}(O(P)) - \overline{F}(D(O(P)))|}{\max\{\overline{F}(O(P)), \overline{F}(D(O(P)))\}},$$

которые характеризуют качество обфусцирующего преобразования O в рамках характеристики F. Эти значения усредняются по множеству $\mathcal{P}$ для каждого $O(\in \mathcal{O})$ по каждому $F(\in \mathcal{F})$:

$$\overline{\delta}^F(\mathcal{P}, O(\mathcal{P})) = \frac{\sum_{P(\in\mathcal{P})} \delta^F(P, O(P))}{|\mathcal{P}|}, \quad \overline{\delta}^F(\mathcal{P}, D(O(\mathcal{P}))) = \frac{\sum_{P(\in\mathcal{P})} \delta^F(P, D(O(P)))}{|\mathcal{P}|},$$

$$\overline{\delta}^F(O(\mathcal{P}), D(O(\mathcal{P}))) = \frac{\sum_{P(\in\mathcal{P})} \delta^F(O(P), D(O(P)))}{|\mathcal{P}|}.$$

Для каждой характеристики F из $\mathcal{F}$ набор значений $\overline{\delta}^F(\mathcal{P}, O(\mathcal{P})), \overline{\delta}^F(\mathcal{P}, D(O(\mathcal{P}))), \overline{\delta}^F(O(\mathcal{P}), D(O(\mathcal{P})))$, где $O(\in \mathcal{O})$ позволяет выявить обфусцирующее преобразование, максимально изменяющее исходную программу в рамках рассмотренных показателей похожести. Для фиксированного O трудно сделать предположение о стойкости обфусцирующего преобразования, так как полученные нормированные показатели похожести для каждой F могут сильно различаться друг от друга. Требуется введение весов для характеристик из $\mathcal{F}$ для нахождения интегрального показателя похожести. С другой стороны, к решению этой задачи можно подойти с использованием методов машинного обучения. Для этого необходимо выявить наиболее изменчивые характеристики. С этой целью, значения $\overline{\delta}^F(\mathcal{P}, O(\mathcal{P})), \overline{\delta}^F(\mathcal{P}, D(O(\mathcal{P}))), \overline{\delta}^F(O(\mathcal{P}), D(O(\mathcal{P})))$, усредненные по множеству $\mathcal{O}$, обозначим

$$\overline{\delta}^F(\mathcal{P}, \mathcal{O}(\mathcal{P})), \overline{\delta}^F(\mathcal{P}, D(\mathcal{O}(\mathcal{P}))), \overline{\delta}^F(\mathcal{O}(\mathcal{P}), D(\mathcal{O}(\mathcal{P}))). \tag{1}$$

Показатель $\overline{\delta}^F(\mathcal{P}, \mathcal{O}(\mathcal{P}))$ характеризует среднее влияние обфусцирующего преобразования на значение характеристики F (эффективность). Показатель $\overline{\delta}^F(\mathcal{P}, D(\mathcal{O}(\mathcal{P})))$ характеризует способность деобфускатора приблизить обфусцированную программу к ее исходной версии (стойкость). А показатель $\overline{\delta}^F(\mathcal{O}(\mathcal{P}), D(\mathcal{O}(\mathcal{P})))$ характеризует способность деобфускатора нивелировать обфусцирующие преобразования (контрольное значение). На основании этих значений можно сделать выбор характеристик, которые могут применяться в модели оценки стойкости на основе машинного обучения.

Усредненные по множеству $\mathcal{F}$ значения

$$\overline{\delta}^{\mathcal{F}}(\mathcal{P}, O(\mathcal{P})), \overline{\delta}^{\mathcal{F}}(\mathcal{P}, D(O(\mathcal{P}))), \overline{\delta}^{\mathcal{F}}(O(\mathcal{P}), D(O(\mathcal{P}))) \tag{2}$$

позволяют выявить обфусцирующие преобразования, оказывающие наибольшее влияние на характеристики символьного исполнения программ.

4. Результаты экспериментов

Для рассматриваемых наборов $\mathcal{F}$, $\mathcal{O}$ и $\mathcal{P}$ результаты расчета усредненных значений (1) и (2) для расширенной и упрощенной схем представлены в таблицах 1, 2 и 3, 4 соответственно. Первый столбец в таблицах 1 и 3 показывает насколько обфускация в среднем изменяет программу от исходной в рамках конкретного показателя похожести, т.е. первый столбец показывает эффективность обфускации. Второй столбец показывает насколько деобфусцированная программа отличается от исходной в рамках того же показателя, т.е. этот столбец характеризует стойкость обфусцирующих преобразований в рамках этого показателя похожести. Экспериментально подтверждается, что третий столбец может определяться по значениям первых двух столбцов: если значения первого и второго столбцов близки, то ожидается, что значение третьего столбца будет близко к нулю.

В [30] предлагается значения показателей похожести, которые менее 0,05, считать незначительными. В настоящей работе эта оценка уточняется, основываясь на среднеквадратичном отклонении значений от среднего. Обозначим $M_{\mathcal{F}}$ и $\sigma_{\mathcal{F}}$ соотвественно среднее значение и среднеквадратичное отклонение показателей похожести усредненных по $\mathcal{O}$, а $M_{\mathcal{O}}$ и $\sigma_{\mathcal{O}}$ – соотвественно среднее значение и среднеквадратичное отклонение показателей похожести усредненных по $\mathcal{F}$. Вычисленные значения

$$\Delta_{\mathcal{O}} = \max\{M_{\mathcal{O}} - \sigma_{\mathcal{O}}, 0\}, \Delta_{\mathcal{F}} = \max\{M_{\mathcal{F}} - \sigma_{\mathcal{F}}, 0\}$$

предлагается рассматривать как границы, по которым можно определить значимость показателей. Если для выбранного показателя, его значение меньше соответствующей границы, показатель считается незначимым. Для таблицы 1 и 3 это означает, что значения символьной характеристики практически не изменяются (в среднем по $\mathcal{O}$). Аналогично, для таблицы 2 и 4 это означает, что преобразование O практически не влияет на все символьные характеристики выполнения (в среднем по $\mathcal{F}$).

Анализ таблиц 1 и 3 показывает, что наиболее изменчивыми как при обфускации, так и при деобфускации являются характеристики: F_{time}, F_{iexe}, F_{tsmt}, F_{qsmt}, F_{qall}, F_{mem}. Отметим, что эти характеристики являются наиболее изменчивыми, как в рамках расширенной схемы, так и в рамках упрощенной схемы. Так как общее время символьного исполнения включает в себя время, затраченное модулем SMT, а также общее количество запросов включает в себя запросы к модулю SMT, то из перечисленных выше характеристик оставлены только четыре характеристики: F_{time}, F_{iexe}, F_{qall}, F_{mem}.

Сравнение таблиц для расширенной схемы с таблицами для упрощенной схемы показывает, что значения показателей для расширенной схемы выросли как минимум в два раза по сравнению с упрощенной. Представляется, что это связано с проблемами описанными ранее (см. подраздел 2.2), в частности, с дополнительными этапами дизассемблирования и трансляции, которые исключены

Table 1. Similarity features averaged by $\mathcal{O}$ for extended scheme

Таблица 1. Усредненные по $\mathcal{O}$ показатели похожести расширенной схемы

F	$\overline{\delta}^F(\mathcal{P}, \mathcal{O}(\mathcal{P}))$	$\overline{\delta}^F(\mathcal{P}, D(\mathcal{O}(\mathcal{P})))$	$\overline{\delta}^F(\mathcal{O}(\mathcal{P}), D(\mathcal{O}(\mathcal{P})))$
F_{iexe}	**0.168**	0.165	0.019
F_{time}	**0.224**	0.222	0.045
F_{icov}	**0.108**	0.108	0.001
F_{bcov}	0.057	0.057	0.001
F_{ilen}	**0.109**	0.109	0.001
F_{tsmt}	**0.144**	0.147	0.02
F_{qsmt}	**0.182**	0.179	0.02
F_{blen}	0.039	0.039	0
F_{qall}	**0.203**	0.195	0.025
F_{mem}	**0.123**	0.122	0.012
$\boldsymbol{\Delta_{\mathcal{F}}}$	**0.076**	**0.076**	**0**

Table 2. Similarity features averaged by $\mathcal{F}$ for extended scheme

Таблица 2. Усредненные по $\mathcal{F}$ показатели похожести расширенной схемы

O	$\overline{\delta}^{\mathcal{F}}(\mathcal{P}, O(\mathcal{P}))$	$\overline{\delta}^{\mathcal{F}}(\mathcal{P}, D(O(\mathcal{P})))$	$\overline{\delta}^{\mathcal{F}}(O(\mathcal{P}), D(O(\mathcal{P})))$
O_{acd}	0.015	0.012	0.018
O_{all}	0.715	0.709	0.01
O_{bcf}	0.105	0.103	0.012
O_{cff}	0.019	0.022	0.016
O_{enc}	0.072	0.07	0.015
O_{fco}	0.016	0.011	0.014
O_{fcw}	0.026	0.025	0.021
O_{ind}	0.298	0.3	0.007
O_{sbb}	0.077	0.075	0.013
O_{sub}	0.014	0.019	0.02
$\boldsymbol{\Delta_{\mathcal{O}}}$	**0**	**0**	**0.01**

Table 3. Similarity features averaged by $\mathcal{O}$ for simplified scheme

Таблица 3. Усредненные по $\mathcal{O}$ показатели похожести упрощенной схемы

F	$\overline{\delta}^F(\mathcal{P}, \mathcal{O}(\mathcal{P}))$	$\overline{\delta}^F(\mathcal{P}, D(\mathcal{O}(\mathcal{P})))$	$\overline{\delta}^F(\mathcal{O}(\mathcal{P}), D(\mathcal{O}(\mathcal{P})))$
F_{iexe}	**0.06**	0.059	0.004
F_{time}	**0.128**	0.15	0.066
F_{icov}	**0.015**	0.015	0
F_{bcov}	**0.009**	0.009	0
F_{ilen}	**0.03**	0.03	0
F_{tsmt}	**0.021**	0.021	0.01
F_{qsmt}	**0.071**	0.07	0.001
F_{blen}	0.005	0.005	0
F_{qall}	**0.104**	0.103	0.001
F_{mem}	**0.025**	0.025	0.002
$\boldsymbol{\Delta_{\mathcal{F}}}$	**0.005**	**0.002**	**0**

из упрощенной схемы. Тем не менее, обе схемы могут использоваться для выбора характеристик, в виду того, что соответствующие наборы наиболее изменчивых показателей почти совпадают. Исключение – показатель F_{bcov}, который является изменчивым в рамках упрощенной схемы, в отличие

Table 4. Similarity features averaged by $\mathcal{F}$ for simplified scheme

Таблица 4. Усредненные по $\mathcal{F}$ показатели похожести упрощенной схемы

O	$\overline{\delta}^{\mathcal{F}}(\mathcal{P}, O(\mathcal{P}))$	$\overline{\delta}^{\mathcal{F}}(\mathcal{P}, D(O(\mathcal{P})))$	$\overline{\delta}^{\mathcal{F}}(O(\mathcal{P}), D(O(\mathcal{P})))$
O_{acd}	0.005	0.008	0.007
O_{all}	0.223	0.222	0.009
O_{bcf}	0.015	0.016	0.01
O_{cff}	0.016	0.019	0.009
O_{enc}	0.026	0.028	0.011
O_{fco}	0.006	0.01	0.007
O_{fcw}	0.007	0.01	0.007
O_{ind}	0.143	0.144	0.008
O_{sbb}	0.014	0.017	0.007
O_{sub}	0.014	0.015	0.008
$\Delta_{\mathcal{O}}$	**0**	**0**	**0.007**

от расширенной, но его значение близко к границе. Однако так как расширенная схема в большей степени соответствует процессу динамического анализа программы аналитиком, а расхождение с результатами упрощенной проявляется только в одном показателе, то в качестве показателей выбраны те, которые определены как изменчивые в рамках расширенной схемы.

Анализ таблиц 2 и 4 показывает, как обфусцирующие преобразования влияют на изменение показателей похожести программ. Применение всех обфускаций одновременно ожидаемым образом оказывает максимальный эффект. Из таблиц видно, что практически все обфусцирующие преобразования оказывают различной степени эффект на характеристики символьного исполнения в рамках рассматриваемых показателей похожести. Несмотря на то, что по значениям из таблиц 2 и 4 можно выделить обфусцирующие преобразования, оказывающие наибольший эффект, значения из этих таблиц характеризуют качество каждого обфусцирующего преобразования исключительно в рамках символьного исполнения – без учета характеристик этих преобразований, полученных при статическом анализе.

Заключение

Показатели похожести, полученные в результате статического анализа программы, можно использовать для оценки эффективности обфусцирующих преобразований, а также для определения их устойчивости к статическим методам деобфускации. Такие показатели могут быть построены на основе различных метрик сложности, рассчитанных, например, от исполнимого файла программы. В настоящей работе для моделирования динамического анализа предложено использовать символьное выполнение. Разработаны и построены две схемы получения характеристик символьного исполнения (расширенная и упрощенная). Для полученных характеристик предложены показатели похожести. С построенными схемами проведены эксперименты с целью определить изменчивые характеристики символьного исполнения с одной стороны, а с другой – сравнить результаты выполнения построенных схем между собой. Сравнение результатов экспериментов обеих схем показывает незначительное отличие в выборе набора изменчивых характеристик символьного исполнения (в упрощенной схеме характеристика F_{bcov} определяется как изменчивая в отличие от расширенной схемы). Также сравнение показывает, что значения показателей упрощенной схемы в среднем меньше значений таких же показателей расширенной схемы, так как символьное исполнение в упрощенной схеме требует меньше ресурсов (времени и памяти) для анализа программы. Тем не менее, характеристики (F_{time}, F_{iexe}, F_{qall}, F_{mem}) являются наиболее чувствительными к изменениям программы в рамках обеих схем. Эти четыре характеристики предлагается использовать

в векторе характеристик для оценки эффективности и стойкости обфускации на основе методов машинного обучения.

Отметим, что расширенная схема нахождения характеристик символьного исполнения является более универсальной по отношению к обфусцирующим преобразованиям, поскольку допускается применение обфускации на уровне машинных инструкций, а также такая схема в большей степени соответствует процессу динамического анализа программы аналитиком.

Дальнейшим направлением исследования является объединение показателей похожести, полученных с помощью статического анализа, с показателями похожести на основе выбранных характеристик символьного исполнения в вектор признаков для проведения экспериментов по оценке эффективности и стойкости обфусцирующих преобразований с помощью методов машинного обучения.

References

[1] C. Collberg and C. Thomborson, "Watermarking, Tamper-Proofing, and Obfuscation - Tools for Software Protection", *IEEE Transactions on Software Engineering*, vol. 28, pp. 735–746, Aug. 2002. DOI: 10.1109/TSE.2002.1027797.

[2] S. Garg, C. Gentry, S. Halevi, M. Raykova, A. Sahai, and B.Waters, "Candidate Indistinguishability Obfuscation and Functional Encryption for all Circuits", in *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, 2013, pp. 40–49. DOI: 10.1109/FOCS.2013.13.

[3] H. Xu, Y. Zhou, J. Ming, and M. Lyu, "Layered obfuscation: a taxonomy of software obfuscation techniques for layered security", *Cybersecurity*, vol. 3, p. 9, Apr. 2020. DOI: 10.1186/s42400-020-00049-3.

[4] C. Collberg, C. Thomborson, and D. Low, "A Taxonomy of Obfuscating Transformations", *Tech. Report, N 148, Dept. of Computer Science, Univ. of Auckland*, Jul. 1997.

[5] Y. Kanzaki, A. Monden, and C. Collberg, "Code Artificiality: A Metric for the Code Stealth Based on an N-Gram Model", in *2015 IEEE/ACM 1st International Workshop on Software Protection*, 2015, pp. 31–37. DOI: 10.1109/SPRO.2015.14.

[6] R. Mohsen and A. Pinto, "Algorithmic Information Theory for Obfuscation Security", in *Proceedings of the 12th International Conference on Security and Cryptography - Volume 1: SECRYPT, (ICETE 2015)*, 2015, pp. 76–87. DOI: 10.5220/0005548200760087.

[7] R. Mohsen and A. Pinto, "Evaluating Obfuscation Security: A Quantitative Approach", in *International Symposium on Foundations and Practice of Security*, Springer, Oct. 2015, pp. 174–192, ISBN: 978-3-319-30302-4. DOI: 10.1007/978-3-319-30303-1_11.

[8] M. Ceccato, M. Di Penta, J. Nagra, P. Falcarin, F. Ricca, M. Torchiano, and P. Tonella, "The Effectiveness of Source Code Obfuscation: an Experimental Assessment", in *2009 IEEE 17th International Conference on Program Comprehension*, May 2009, pp. 178–187. DOI: 10.1109/ICPC.2009.5090041.

[9] J. Siegmund, "Program Comprehension: Past, Present, and Future", in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 5, Mar. 2016, pp. 13–20. DOI: 10.1109/SANER.2016.35.

[10] E. Avidan and D. Feitelson, "From Obfuscation to Comprehension", in *2015 IEEE 23rd International Conference on Program Comprehension*, May 2015, pp. 178–181. DOI: 10.1109/ICPC.2015.27.

[11] P. Borisov and Y. Kosolapov, "On the Automatic Analysis of the Practical Resistance of Obfusting Transformations", *Modeling and Analysis of Information Systems*, vol. 26, no. 3, pp. 317–331, Sep. 2019. DOI: 10.18255/1818-1015-2019-3-317-331.

[12] J. King, "Symbolic Execution and Program Testing", *Commun. ACM*, vol. 19, no. 7, pp. 385–394, Jul. 1976. DOI: 10.1145/360248.360252.

[13] B. Yadegari and S. Debray, "Symbolic Execution of Obfuscated Code", in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, Oct. 2015, pp. 732–744. DOI: 10.1145/2810103.2813663.

[14] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation", in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, ser. CGO '04, USA: IEEE Computer Society, 2004, pp. 75–86, ISBN: 0769521029.

[15] P. F. Brown, P. V. deSouza, R. L. Mercer, V. J. D. Pietra, and J. C. Lai, "Class-Based n-Gram Models of Natural Language", *Comput. Linguist.*, vol. 18, no. 4, pp. 467–479, Dec. 1992, ISSN: 0891-2017.

[16] N. Zhang, *Hikari – an improvement over Obfuscator-LLVM*, 2017.

[17] A. Dinaburg and A. Ruef, "Mcsema: Static translation of x86 instructions to llvm", in *ReCon 2014 Conference, Montreal, Canada*, 2014.

[18] C. Cadar and M. Nowack, "KLEE symbolic execution engine in 2019", *International Journal on Software Tools for Technology Transfer*, Jun. 2020. DOI: 10.1007/s10009-020-00570-3.

[19] S. Muchnick, *Advanced Compiler Design Implementation*. 1997, ISBN: 9781558603202.

[20] C. Eagle, *The IDA pro book: the unofficial guide to the world's most popular disassembler*, 2nd ed. No Starch Press, ISBN: 1593273959.

[21] G. Ravipati, A. R. Bernat, N. Rosenblum, B. P. Miller, and J. K. Hollingsworth, "Towards the Deconstruction of Dyninst", UW Madison, Tech. Rep., Jul. 2007, pp. 1–9.

[22] R. N. Horspool and N. Marovac, "An approach to the problem of detranslation of computer programs", *The Computer Journal*, vol. 23, no. 3, pp. 223–229, 1980.

[23] C. Visual and B. Unit, *Microsoft portable executable and common object file format specification*, 1999.

[24] H. Lu, *Elf: From the programmer's perspective*, 1995.

[25] J. Křoustek, P. Matula, J. Končický, and D. Kolář, "Accurate Retargetable Decompilation Using Additional Debugging Information", Jan. 2012.

[26] S. Dasgupta, S. Dinesh, D. Venkatesh, V. S. Adve, and C. W. Fletcher, "Scalable validation of binary lifters", *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 655–671, 2020.

[27] S. Banescu, C. Collberg, V. Ganesh, Z. Newsham, and A. Pretschner, "Code obfuscation against symbolic execution attacks", Dec. 2016, pp. 189–200. DOI: 10.1145/2991079.2991114.

[28] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin, "Obfuscator-LLVM – Software Protection for the Masses", May 2015, pp. 3–9. DOI: 10.1109/SPRO.2015.10.

[29] T. László and Á. Kiss, "Obfuscating C++ Programs via Control Flow Flattening", *Annales Universitatis Scientiarum Budapestinensis de Rolando Eötvös Nominatae. Sectio Computatorica*, vol. 30, no. 1, pp. 3–19, 2009.

[30] Y. Kosolapov and P. Borisov, "Similarity Features For The Evaluation Of Obfuscation Effectiveness", in *2020 International Conference on Decision Aid Sciences and Application (DASA)*, 2020, pp. 898–902. DOI: 10.1109/DASA51403.2020.9317301.