Моделирование и анализ информационных систем. Т. 26, №3 (2019), с. 317–331
Modeling and Analysis of Information Systems. Vol. 26, No 3 (2019), pp. 317–331

Software

DOI: 10.18255/1818-1015-2019-3-317-331

УДК 517.9

Об автоматическом анализе практической стойкости обфусцирующих преобразований

Борисов П. Д., Косолапов Ю. В.

Поступила в редакцию 18 июля 2019
После доработки 9 сентября 2019
Принята к публикации 11 сентября 2019

Аннотация. Разрабатывается способ оценки практической стойкости обфусцирующих преобразований программ, основанный на вычислении показателя похожести для исходной, обфусцированной и деобфусцированной программ. Предлагаются кандидаты для показателей похожести, в основе вычисления которых лежат такие характеристики программ, как граф потока управления, время символьного выполнения и степень покрытия при символьном выполнении. Граф потока управления рассматривается как основа для построения других кандидатов для показателей похожести программ. На его основе предлагается новый кандидат для показателя похожести, при вычислении которого находится расстояние Хэмминга между матрицами смежности графов потока управления сравниваемых программ. Строится схема оценки (анализа) стойкости обфусцирующих преобразований, в соответствии с которой для исходной, обфусцированной и деобфусцированной программ вычисляются или находятся характеристики этих программ, которые сравниваются в соответствии с выбранной моделью сравнения. Разработанная схема, в частности, подходит для сравнения программ на основе показателей похожести. В работе разрабатывается и реализуется один из ключевых блоков построенной схемы – блок получения характеристик программ, скомпилированных для архитектуры x86/x86_64. Разработанный блок позволяет находить граф потока управления, время символьного выполнения и степень покрытия при символьном выполнении. Приводятся некоторые результаты работы построенного блока.

Ключевые слова: обфускация кода, стойкость, символьное исполнение

Для цитирования: Борисов П. Д., Косолапов Ю. В., "Об автоматическом анализе практической стойкости обфусцирующих преобразований", *Моделирование и анализ информационных систем*, **26**:3 (2019), 317–331.

Об авторах:
Борисов Петр Дмитриевич, orcid.org/0000-0002-8919-8310, аспирант
Южный Федеральный Университет,
ул. Мильчакова, 8а, г. Ростов-на-Дону, 344090 Россия, e-mail: borisovpetr@mail.ru

Косолапов Юрий Владимирович, orcid.org/0000-0002-1491-524X, канд. техн. наук
Южный Федеральный Университет,
ул. Мильчакова, 8а, г. Ростов-на-Дону, 344090 Россия, e-mail: itaim@mail.ru

Введение

Обфускация (или запутывание) программного кода используется для защиты программ от анализа типа обратной инженерии (reverse engineering). Обфускация мо-

жет применяться на разных уровнях: на уровне исходного кода, на уровне промежуточного представления, а также на уровне машинных команд. При этом целью обфускации является не только затруднение понимания алгоритмов программы при статическом анализе, но и затруднение понимания при динамическом анализе. В настоящее время разработано множество способов обфускации, которые на интуитивном уровне позволяют затруднить *понимание* защищаемых алгоритмов, однако нет теоретического обоснования стойкости для этих методов. Отметим, что *понимание* (comprehension) исходного кода программного обеспечения является в настоящее время активно исследуемой областью в программной инженерии [1], а в [2] для измерения понятности кода предлагаются метрики, построенные на основе знаний из области обфускации исходного кода (пока отсутствует экспериментальное доказательство пригодности этих метрик).

При разработке обфусцирующего преобразования важной задачей представляется обоснование стойкости этого преобразования к современным методам деобфускации (к распутыванию кода) [3]. Деобфускация может выполняться как с использованием автоматических средств деобфускаци (автоматическая деобфускация, без участия человека), так и с участием человека-эксперта, использующего инструментальные средства анализа. В [4] показано, что для многих способов обфускации имеются способы деобфускации, многие из которых поддаются автоматизации. Поэтому актуальными являются задачи, во-первых, разработки способа анализа стойкости обфусцирующих преобразований, примененных на любом уровне, а во-вторых, актуальна задача разработки практически устойчивых способов обфускации к автоматической деобфускации. В настоящей работе разрабатывается схема оценки практической стойкости обфусцирующих преобразований к автоматическим методам деобфускации.

Статья кроме введения и заключения содержит три раздела. Первый раздел посвящен обзору работ в области современных методов обфускации и автоматической деобфускации. Во втором разделе предлагается способ оценки стойкости обфусцирующих преобразований на основе вычисления показателя похожести программ для исходной, обфусцированной и деобфусцированной программ. Третий раздел посвящен разработке и реализации схемы оценки стойкости обфусцирующих преобразований.

1. Обзор работ

1.1. Обфускация в программной инженерии

В настоящее время исследование в области обфускации ведется по двум направлениям: 1) применение обфускации в криптографии и 2) обфускация в программной инженерии и компьютерной безопасности [5]. В области криптографии интерес к обфускации вызван тем доказанным фактом, что если возможно создать теоретически стойкий алгоритм обфускации программ, то его можно преобразовать в криптосистему с открытым ключом, в которой секретным ключом будет сам алгоритм расшифрования, а открытым ключом будет обфусцированная шифрующая программа [6]. В программной инженерии обфускация применяется для решения следующих задач: для защиты алгоритмов, представляющих собой интеллектуаль-

ную собственность [7] (в частности, для включения «водяных» знаков), для защиты критических участков кода (например, при проверке лицензий) [8], для маскировки критичных данных, например, ключа шифрования и/или расшифрования [6], для маскировки известного программного обеспечения под новое программное обеспечение (например, для обхода сигнатурного анализа антивирусных программ и систем обнаружения вторжений) [9, 10], для защиты программ от эксплуатации возможных уязвимостей (диверсификация) [11, 12], в частности, от атак типа повторно выполнения исполнимого кода [13].

Можно выделить следующие уровни обфускации программного кода: уровень исходного кода (C/C++/Java/JavaScript и т.п.), уровень промежуточного кода (бит код LLVM) и уровень целевого исполняемого кода (байт-код Java и WebAssembly, машинный код для архитектуры Intel x86/x86_64 и т.п.). Здесь и далее под уровнем целевого исполняемого кода понимается уровень, на котором непосредственно выполняются инструкции кода; в качестве интерпретатора на таком уровне могут выступать, например, процессор архитектуры Intel x86/x86_64 или виртуальная Java-машина. На каждом из этих уровней могут применяться как специфичные для каждого уровня способы обфускации (например, шифрование константных строк, замена имен переменных, добавление избыточных инструкций, добавление непрозрачных предикатов, перестановка базовых блоков), так и общие для всех уровней способы (например, замена набора инструкций эквивалентным по функциональности набором инструкций, вставка мертвого кода).

1.2. Автоматическая деобфускация

В [14] предлагается обфусцирующие преобразования оценивать по трем характеристикам: стоимость обфусцирующего преобразования (cost), устойчивость к автоматической деобфускации (resilience) и устойчивость к деобфускации человеком (potency). Две последние характеристики определяют стойкость преобразований. Автоматическая деобфускация находит применение, например, при исследовании новых образцов вредоносного кода, когда сначала необходимо выяснить, является ли этот образец действительно новым вредоносным кодом, или это обфусцированная версия вредоносного кода с уже известной сигнатурой. Также автоматическая деобфускация применяется при выявлении плагиата программного обеспечения [15]. Деобфускация кода человеком обычно выполняется в случае, если автоматические способы деобфускации не дают результатов. Эмпирически установлено, что понимание программ человеком по своей природе тесно связано с обработкой человеком естественного языка [1]. Учитывая, что привлечение человека к деобфускации обычно существенно повышает стоимость процесса деобфускации, большое внимание уделяется разработке и усовершенствованию автоматических способов деобфускации, в частности, на основе символьного исполнения (см. [16, 17]). Отметим, что это, в свою очередь, приводит к развитию способов обфускации, направленных на противодействие автоматическому распутыванию кода [18].

Автоматическая деобфускация программ может выполняться как статически, так и динамически. Статический анализ не предполагает запуск кода, в то время как при динамическом анализе программа запускается. Совместное использование статического и динамического анализа при оценке стойкости обфусцирующих

преобразований обусловлено тем, что такие преобразования могут быть устойчивы к статическому анализу, но не быть устойчивыми к динамическому, или наоборот. Отметим, что при статическом анализе могут использоваться техники, аналогичные тем, которые используются в компиляторах при оптимизации программ: поиск циклов, условных операторов, выявление избыточных операций, недостижимого и/или мертвого кода и т.п. При динамическом анализе используется, например, сравнение трассы выполненных команд с кодом программы.

Одним из мощных средств, используемом в автоматической деобфускации программ, является символьное исполнение кода [17–19], которое изначально было разработано для автоматического тестирования программного обеспечения, поиска ошибок и выявления уязвимостей [20]. Символьное исполнение позволяет построить наборы входных данных, на которых будет исполняться определенный путь в программе [20]. Эту задачу выполняет решающий модуль (resolver). Таким образом, для каждого пути исполнения формируется набор ограничений на входные параметры, по которым могут быть сгенерированы конкретные наборы значений, используемых для прохождения этого пути. Решение данной задачи обычно получается с помощью SMT-решателя (Satisfiability Modulo Theories). Результатом работы символьного интерпретатора являются наборы входных параметров программы и процент покрытия кода программы символьным исполнением. Для анализа программ символьное исполнение используется в таких средствах как S^2E [16] (на основе символьного интерпретатора KLEE [21]), Triton [17], Angr [22]. Для защиты от методов анализа (деобфускации), применяющих символьное исполнение, в работах [18, 23] предлагается использовать непрозрачные предикаты на основе криптографических хэш-функций. В этом случае у решающего модуля может потребоваться длительное время для нахождения условий, при которых предикат является истинным/ложным. Однако непрозрачные предикаты могут быть выявлены при динамическом символьном исполнении (dynamic symbolic execution), например, путем многоразового запуска программы [24].

В настоящей работе предлагается проводить оценку *практической* стойкости обфусцирующих преобразований по отношению к заданным автоматическим методам деобфускации. Заметим, что здесь не рассматриваются вопросы теоретической вычислительной стойкости (стойкость по отношению ко всем полиномиальным алгоритмам деобфускации).

2. Оценка стойкости

2.1. Информационно-аналитическая модель

Пусть $\mathcal{P}$ — множество всех исполнимых программ для фиксированной архитектуры вычислительного устройства, представленных в заранее известном формате (например, на языке ассемблера), $\mathcal{T}$ — множество таких отображений (преобразований) вида $\tau : \mathcal{P} \to \mathcal{P}$, что для каждой программы $P \in \mathcal{P}$ программа $\tau(P)$ семантически эквивалентна программе P на всех допустимых входных данных. Во множество $\mathcal{T}$, в частности, входят оптимизирующие код преобразования, а также обфусцирующие и деобфусцирующие преобразования. Множество $\mathcal{T}$ задает на множестве $\mathcal{P}$ отношение эквивалентности: две программы P_1 и P_2 эквивалентны, тогда и толь-

ко тогда, когда существует такое отображение τ, что $P_1 = \tau(P_2)$. Во множестве $\mathcal{T}$ выделим подмножество обфусцирующих преобразований $\mathcal{T}^O$, которые направлены на затруднение понимания программ. В $\mathcal{T}^O$ рассмотрим некоторое фиксированное множество преобразований

$$\{\tau_1^o, ..., \tau_r^o\} \subseteq \mathcal{T}^O. \tag{1}$$

и деобфусцирующее преобразование $\tau^d \in \mathcal{T} \setminus \mathcal{T}^O$. Требуется из набора (1) выбрать преобразование, наиболее устойчивое к деобфусцирующему преобразованию τ^d.

2.2. Способ оценки стойкости

В настоящей работе выбор преобразования из набора (1) предлагается выполнять на основе вычисления *показателя похожести* для программ $\tau^d(\tau_i^o(P))$ и $P(\in \mathcal{P})$, $i = 1, ..., r$. Для сравнения двух программ рассмотрим *показатель похожести*

$$\delta : \mathcal{P} \times \mathcal{P} \to [0, 1](\subseteq \mathbb{R}).$$

При этом будем полагать, что $\delta(P_1, P_2) = \delta(P_2, P_1)$ для всех $(P_1, P_2) \in \mathcal{P} \times \mathcal{P}$ и $\delta(P, P) = 0$ для всех P из $\mathcal{P}$. Для показателя δ рассмотрим такую пару $(\epsilon_\delta^+, \epsilon_\delta^-) \in [0, 1] \times [0, 1]$, что при $\delta(P_1, P_2) \le \epsilon_\delta^+$ программы $P_1, P_2(\in \mathcal{P})$ считаются похожими, а при $\delta(P_1, P_2) \ge \epsilon_\delta^+$ – непохожими. Очевидно, что для показателя похожести должно выполняться неравенство

$$\epsilon_\delta^+ < \epsilon_\delta^-. \tag{2}$$

Для заданного показателя δ пара $(\epsilon_\delta^+, \epsilon_\delta^-)$ может быть найдена в результате вычислительного эксперимента в соответствии со следующим алгоритмом 1.

Исходные параметры: $\tilde{\mathcal{P}}(\subseteq \mathcal{P})$ – подмножество различных по функционалу исполнимых программ, $\delta : \mathcal{P} \times \mathcal{P} \to [0, 1]$ — претендент на показатель похожести, $\tilde{\mathcal{T}} \subseteq \mathcal{T} \setminus \mathcal{T}^O$ — подмножество необфусцирующих преобразований

Результат: пара $(\epsilon_\delta^+, \epsilon_\delta^-)$

$\epsilon_\delta^+ := 0, \epsilon_\delta^- := 1$

для каждого $P \in \tilde{\mathcal{P}}$ **выполнять**

  построить $\tilde{\mathcal{T}}(P) = \{\tau(P) : \tau \in \tilde{\mathcal{T}}\}$

  для каждого $(P_1, P_2) \in \tilde{\mathcal{T}}(P) \times \tilde{\mathcal{T}}(P)$ **выполнять**

    $\epsilon_\delta^+ := \max\{\epsilon_\delta^+, \delta(P_1, P_2)\}$

  конец цикла

  для каждого $\tilde{P} \in \tilde{\mathcal{P}}, \tilde{P} \ne P$ **выполнять**

    $\epsilon_\delta^- := \min\{\epsilon_\delta^-, \delta(P, \tilde{P})\}$

  конец цикла

конец цикла

Алгоритм 1: Экспериментальное вычисление пары $(\epsilon_\delta^+, \epsilon_\delta^-)$

Algorithm 1: Experimental calculation of pair $(\epsilon_\delta^+, \epsilon_\delta^-)$

Значения показателя δ естественным образом отображаются в два возможных значения: программы похожи (П) или непохожи (Н). Если $\epsilon_\delta^+ < \delta(P_1, P_2) < \epsilon_\delta^-$, то в случае $\delta(P_1, P_2) \in [0, \epsilon_\delta^+ + (\epsilon_\delta^- - \epsilon_\delta^+)/2]$ полагаем, что программы похожи, а в противном случае считаем, что программы непохожи.

Определение 1. *Будем говорить, что обфусцирующее преобразование* $\tau^o (\in \mathcal{T}^O)$ *эффективно в рамках показателя* δ*, если*

$$\forall P \in \mathcal{P} \;\; \delta(P, \tau^{obf}(P)) \geq \epsilon_\delta^-.$$

Другими словами, преобразование является эффективным, если исходная и обфусцированная программа считаются непохожими в рамках показателя δ.

Определение 2. *Будем говорить, что деобфусцирующее отображение* τ^d *является эффективным преобразованием для преобразования* τ^o *в рамках показателя похожести* δ*, если*

$$\forall P \in \mathcal{P} \;\; \delta(P, \tau^d(\tau^o(P))) \leq \epsilon_\delta^+.$$

Аналогично можно ввести определения $\tilde{\mathcal{P}}$-эффективных обфусцирующих и деобфусцирующих преобразований, если в определениях 1 и 2 рассмотреть вместо множества всех программ $\mathcal{P}$ некоторое фиксированное подмножество $\tilde{\mathcal{P}}$.

Для программы P, обфусцирующего преобразования τ^o и деобфусцирующего преобразования τ^d возможны восемь возможных сочетаний значения показателя δ, которые представлены в таблице 1.

Таблица 1. Сочетания значений показателя похожести δ для заданного обфусцирующего преобразования τ^o и деобфусцирующего преобразования τ^d
Table 1. Combinations of values of the similarity index δ for a given obfuscating transformation τ^o and deobfussing transformation τ^d

	$\delta(P, \tau^o(P))$	$\delta(\tau^o(P), \tau^d(\tau^o(P)))$	$\delta(P, \tau^d(\tau^o(P)))$
1	П	П	П
2	П	П	Н
3	П	Н	П
4	П	Н	Н
5	Н	П	П
6	Н	П	Н
7	Н	Н	П
8	Н	Н	Н

Преобразования, соответствующие первым четырем сочетаниям таблицы, не являются эффективными обфусцирующими преобразованиями. Сочетания 6 и 8 соответствуют эффективному обфусцирующему преобразованию τ^o, устойчивому в рамках показателя δ к деобфусцирующему преобразованию τ^d, так как, с одной стороны, по показателю δ обфусцированная и исходная программы непохожи, а с другой стороны, исходная программа и деобфусцированная непохожи. Сочетания 5, 7, наоборот, соответствуют обфусцирующему преобразованию τ^o, неустойчивому к деобфусцирующему преобразованию τ^d (также в рамках показателя δ).

Утверждение 1. *Для определения стойкости эффективного обфусцирующего преобразования* τ^o *к деобфусцирующему преобразованию* τ^d *в рамках показателя похожести* δ *достаточно вычислить* $\delta(P, \tau^d(\tau^o(P)))$.

Доказательство. По условию, преобразование τ^o эффективное. Так как значение показателя $\delta(P, \tau^d(\tau^o(P)))$ для сочетаний 6 и 8 таблицы 1 одинаковое (для строк 5 и 7 значение также одинаковое, но отличается от значения в строках 6 и 8), то для определения стойкости этого значения достаточно. □

Таким образом, если из заданного набора (1) обфусцирующих преобразований требуется выбрать наиболее устойчивое к заданному методу деобфускации τ^d, то следует выбирать обфусцирующиее преобразование τ^o с наибольшим числовым значением $\delta(P, \tau^d(\tau^o(P)))$, соответствующим бинарному значению «Н» (программы P и $\tau^d(\tau^o(P))$ непохожи). Если же имеется набор показателей похожести $\{\delta_1, ..., \delta_n\}$, то наиболее стойким следует считать преобразование τ^o с наибольшим числом показателей δ_i, значения которых для программ P и $\tau^d(\tau^o(P))$ соответствуют бинарному значению «Н» и принимают наибольшее значение.

Стоит отметить, что если для пары $(\epsilon_\delta^+, \epsilon_\delta^-)$, найденной в соответствии с алгоритмом 1, не выполняется условие (2), то считается, что показатель δ не подходит для оценки эффективности и стойкости обфусцирующих преобразований в рамках предлагаемого способа. Поиск подходящего показателя похожести, для которого выполняется условие (2), является актуальной и, как можно заключить из обзора [15], пока нерешенной задачей. В частности, остается пока открытым вопрос о составе множества $\tilde{\mathcal{T}}$, используемом в алгоритме 1 для нахождения значений, соответствующих похожим программам. Можно предположить, что во множество $\tilde{\mathcal{T}}$ следует включать преобразования, соответствующие, например, преобразованиям разных компиляторов (из исходного кода в код целевой архитектуры), разным опциям компиляции, использующим различные наборы оптимизирующих преобразований. Однако, в частности, некоторые оптимизирующие преобразования, могут рассматриваться и как обфусцирующие преобразования. Поэтому необходимы исследования для определения состава множества $\tilde{\mathcal{T}}$. Другим способом построения множества $\tilde{\mathcal{T}}(P)$ может быть выбор различных версий/обновлений одной программы P.

Кроме поиска подходящих показателей похожести актуальными являются задачи построения *эффективно вычислимых* показателей похожести и установление корреляции между различными показателями похожести. В следующем разделе предлагаются возможные кандидаты для показателей похожести.

2.3. Показатели похожести программ

2.3.1. Похожесть графов потока управления

Множество всех возможных путей выполнения программы $P(\in \mathcal{P})$, представленное в виде графа $G_P = (V_P, E_P)$ называется *графом потока управления* (control flow graph) программы. Узлы из V_P такого графа обычно соответствуют базовым блокам программы P – последовательностям команд, завершающимися командами перехода на другие базовые блоки. Ребра (из E_P), исходящие из заданного узла, соответствуют условиям, при выполнении которых совершается переход из заданного базового блока к следующему базовому блоку. Как показывают исследования, граф потока управления является важной характеристикой программ, используемой при их сравнении [15]. Многие обфусцирующие преобразования не влияют или слабо

влияют на топологию графа, что используется, например, для выявления обфусцированных версий уже известных вредоносных программ [25–28]. Для сравнения графов часто используется подход, основанный на итеративном поиске и сопоставлении в графах наиболее «похожих» узлов [29, 30]. Под сопоставлением здесь понимается присвоение этим узлам одинаковых меток, уникальных в рамках каждого графа. Некоторые другие подходы к сравнению графов рассматриваются в [31]. Там же предлагается методология для оценки различных способов сравнения графов потока управления.

В настоящей работе для сравнения графов потока управления программ P_1 и P_2 предлагается использовать утилиту *BinDiff*, в которой реализован алгоритм сопоставления вершин из [30]. Результатом работы этой программы является число в диапазоне от 0 (программы полностью отличаются) до 1 (программы полностью совпадают). Такой показатель похожести обозначим δ_{BinDiff}, а его значение определяется формулой:

$$\delta_{\text{BinDiff}}(P_1, P_2) = 1 - \mathit{BinDiff}(P_1, P_2). \tag{3}$$

2.3.2. Похожесть при символьном исполнении

В разделе 1.2. отмечается, что важными характеристиками символьного исполнения являются время исполнения, необходимое для генерации входных параметров, и процент покрытия кода. Время исполнения зависит от времени, необходимого SMT-решателю. Пусть $t_{\text{SE}}(P)$ и $c_{\text{SE}}(P)$ – время, необходимое SMT-решателю, и степень покрытия кода при символьном исполнении программы $P(\in \mathcal{P})$ соответственно. Тогда можно рассмотреть два показателя похожести программ P_1 и P_2 в рамках символьного исполнения:

$$\delta^t_{\text{SE}}(P_1, P_2) = \frac{|t_{\text{SE}}(P_1) - t_{\text{SE}}(P_2)|}{\max\{t_{\text{SE}}(P_1), t_{\text{SE}}(P_2)\}}, \tag{4}$$

$$\delta^c_{\text{SE}}(P_1, P_2) = \frac{|c_{\text{SE}}(P_1) - c_{\text{SE}}(P_2)|}{\max\{c_{\text{SE}}(P_1), c_{\text{SE}}(P_2)\}}. \tag{5}$$

2.3.3. О построении новых показателей похожести

На основе графа потока управления могут быть построены и другие показатели похожести, отличные от (3). Разнообразие показателей, с одной стороны, позволяет выбрать наиболее подходящий показатель в рамках предлагаемого выше способа, а с другой стороны, множество показателей может быть использовано при построении показателя похожести на основе машинного обучения. Отметим, что значения разных показателей похожести могут быть использованы для построения вектора характеристик, например, в алгоритмах машинного обучения, использующих метод опорных векторов SVM (Support Vector Method). Рассмотрим возможный подход к построению показателей похожести на основе графа потока управления.

Пусть P_1 и P_2 – программы, графы потока управления которых требуется сравнить, $[n] = \{1, ..., n\}$. Для удобства отождествим узлы каждого графа с метками, присвоенными этим узлам после выполнения процедуры сопоставления узлов графов G_{P_1} и G_{P_2}, например, в соответствии с алгоритмом из [30]. При этом узлам

графов, которые не были сопоставлены, присваиваются уникальные метки в рамках двух графов. Пусть $k = |V_{P_1} \cup V_{P_2}|$ — число различных меток для двух графов. Для $i = 1, 2$ по графу G_i построим граф $G'_i = (V_{P_1} \cup V_{P_2}, E_{P_i})$, которому сопоставим $(k \times k)$-матрицу смежности A^i, столбцы и строки которой пронумерованы элементами множества $V_{P_1} \cup V_{P_2}$:

$$A^i_{v,\tilde{v}} = \begin{cases} 1, & (v, \tilde{v}) \in E_{P_i} \\ 0, & (v, \tilde{v}) \notin E_{P_i} \end{cases}.$$

Величину $\rho(A^1, A^2) = \{(i, j) \in [k] \times [k] : a^1_{i,j} \neq a^2_{i,j}\}$ будем называть расстоянием Хэмминга между матрицами A^1 и A^2, так как она равна числу пар индексов, где элементы матриц A и B отличаются. Такое расстояние в [32] называется расстоянием редактирования графа. В качестве показателя похожести для P_1 и P_2 предлагается нормированное расстояние Хэмминга между соответствующими матрицами A^1 и A^2:

$$\delta_{\text{Hamm}} = \frac{\rho(A^1, A^2)}{k^2}. \tag{6}$$

Для применения показателя похожести δ_{Hamm} узлам двух сравниваемых программ должны быть присвоены метки, что является непростой задачей. Некоторые подходы сопоставления вершин предлагаются, например, в [29, 30]. Для сопоставления базовых блоков представляется возможным применение нечетких хэш-функций, которые слабо чувствительны к небольшим изменениям [33]. Тогда в сравниваемых программах базовым блокам присваиваются одинаковые метки, если хэш-значения, вычисленные по коду этих базовых блоков, наиболее близки. Затем поиск наиболее похожих базовых блоков повторяется, но при этом не учитываются ранее помеченные блоки. Заметим, что для применения такого подхода необходимо представление кода базовых блоков в виде, подходящем для применения нечетких хэш-функций (или других методов сравнения).

Отметим, что если вместо графа потока управления рассматривать граф вызовов системных функций и/или библиотечных функций при статическом анализе, или трассу системных вызовов при динамическом анализе (такую трассу, например, в операционной системе Linux можно получить с помощью утилит *strace*/*ltrace*), то задача вычисления показателя δ_{Hamm} упрощается, так как узлы уже имеют метки — названия системных и/или библиотечных функций. Однако сравнение на основе графов системных/библиотечных функций имеет ограниченную сферу применения: этот подход неприменим для сравнения программ, в которых не применяются системные/библиотечные функции (программы в рамках такого показателя похожести будут всегда похожими).

3. Схема анализа стойкости

3.1. Схема на основе сравнения характеристик

Так как при обфускации и деобфускации вносятся изменения в программный код, то это может привести к изменению характеристик, от которых зависят показатели

похожести. В [34] проведен обзор характеристик программного обеспечения, предложенных различными исследователями для оценки сложности программного обеспечения. К числу таких характеристик относятся, в частности, плотность условных операторов, число различных путей выполнения программы, размер программы, число различных операторов, степень влияния изменения одного модуля программы на остальные модули программы. О влиянии некоторых из этих характеристик на сложность понимания программ выдвинуты гипотезы в [35].

Схема анализа стойкости обфусцирующих преобразований показана на рис. 1. Исходная программа подвергается обфускации с помощью преобразования τ^o; результат обфускации затем подается на вход блока деобфускации, где применяется выбранное преобразование τ^d; в блоке нахождения характеристик вычисляются характеристики программ, которые сравниваются в блоке оценки стойкости. Для сравнения характеристик программного обеспечения может быть использована модель на основе похожести программ из раздела 2.2.. В качестве характеристик,

Рис. 1. Структурная схема оценки стойкости обфусцирующих преобразований
Fig. 1. Structural scheme for assessing the persistence of obfusting transformations

получаемых в блоке нахождения характеристик, могут быть такие характеристики, как граф потока управления программ, время символьного исполнения, степень покрытия при символьном исполнении. Тогда в блоке оценки стойкости могут использоваться показатели δ_{BinDiff}, δ_{Hamm}, δ^t_{SE}, δ^c_{SE} (см. формулы (3), (6), (4) и (5)).

Заметим, что граф потока управления может быть найден с помощью множества автоматизированных средств. Такой граф может быть получен, в частности, для исполнимого файла, скомпилированного для целевой архитектуры, например, Intel

x86/x86_64. Для нахождения характеристик символьного исполнения таких файлов разработан и реализован подход, описанный в следующем разделе.

3.2. Нахождение характеристик символьного исполнения

Для оценки стойкости обфусцирующих преобразований разработан алгоритм нахождения характеристик исполнимых файлов, скомпилированных для архитектуры Intel x86/x86_64 [36]. Результатом выполнения этого алгоритма являются характеристики программ, которые могут быть использованы для оценки стойкости обфусцирующих преобразований в рамках схемы оценки стойкости обфусцирующих преобразований, показанной на рис. 1. Модель нахождения характеристик включает следующие компоненты: модуль дизассемблирования, модуль трансляции кода программы на языке ассемблера в код на языке промежуточного представления, модуль оптимизации, модуль символьного исполнения, модуль сбора анализируемых характеристик на уровне промежуточного представления.

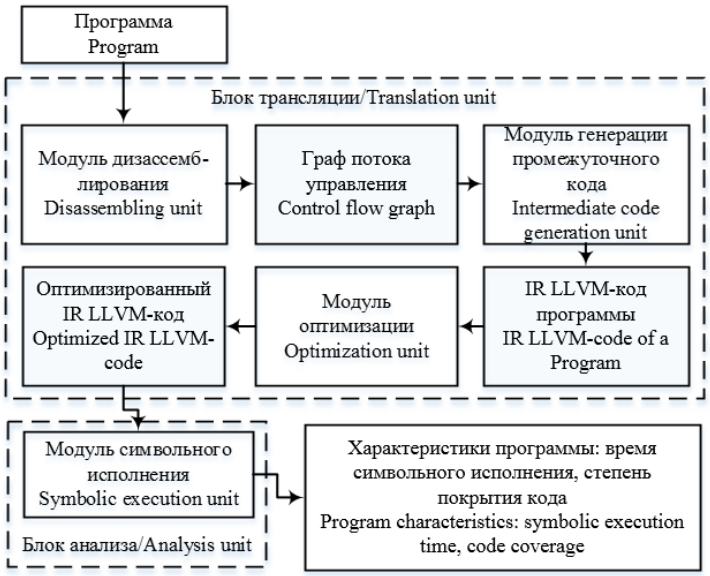

Рис. 2. Реализация блока нахождения характеристик на основе символьного исполнения
Fig. 2. Implementing a block for finding characteristics based on symbolic execution

Модуль дизассемблирования реализуется с помощью средства *IDA Pro 7.0* [37] и утилиты *mcsema-disass*, входящей в фреймворк *McSema* [39], предназначенной для транслирования исполнимых программ, скомпилированных для архитектур x86/x86_64/arm, в программы на языке промежуточного представления *LLVM IR* (Low Level Virtual Machine Intermediate Representation) [38]. Средство *IDA Pro 7.0* дизассемблирует программу, а утилита *mcsema-disass*, используя API-интерфейс *IDA Pro 7.0*, на основе полученного представления формирует граф потока управления. В качестве промежуточного представления используется биткод *LLVM IR*. Модуль трансляции в промежуточное представление *LLVM IR* реализуется с помощью утилиты *mcsema_lift* из состава фреймворка *McSema*. Утилита *mcsema_lift* генерирует *LLVM IR* по полученному графу потока управления программы. Таким

образом, исполнимая программа транслируется в единый модуль в промежуточном представлении *LLVM IR*. Модуль оптимизации представлен утилитой *opt* из состава *LLVM*. Оптимизация программы производится на уровне промежуточного представления. Символьное исполнение выполняется с помощью фреймворка KLEE [21]. Напомним, что суть символьного исполнения заключается в подмене входных параметров программы символьными значениями. Источниками входных параметров для консольных программ часто являются аргументы командной строки и стандартный поток ввода. Символьный интерпретатор пытается исследовать всю программу, определить все возможные пути исполнения и соответствующие этим путям входные параметры. Каждое ветвление в программе воспринимается как ограничение на входные параметры, которым они должны удовлетворять, чтобы продолжить выполнение программы по каждому из возможных путей.

В качестве примера работы реализованного алгоритма нахождения характеристик символьного выполнения проведен ряд экспериментов с программой Maze, реализованной на языке C, исходный код которой доступен в сети. Программа представляет собой простую игру, для победы в которой необходимо через стандартный поток ввода (stdin) правильно задать путь до выхода из представленного на консоли лабиринта. Для компиляции программы использовался компилятор Obfuscator-LLVM [40], в котором имеется возможность добавить обфусцирующие преобразования на стадии представления программы на языке LLVM IR. В ходе эксперимента программа Maze была скомпилирована в четырех разных режимах: без применения обфусцирующих преобразований (maze-org), с применением обфусцирующего преобразования Instruction Substitution (maze-sub), с применением преобразования Control Flow Flattening (maze-fla), а также с применением обфусцирующего преобразования Bogus Control Flow (maze-bcf). Каждый из полученных четырех исполнимых файлов был подан на вход схемы (см. рис. 2) и получены соответствующие характеристики: время символьного исполнения и степень покрытия кода при символьном исполнении. Результаты анализа показаны в таблице 2.

Таблица 2. Пример вычисления характеристик для программы Maze
Table 2. An example of calculating characteristics for the program Maze

Исполнимый файл	**Время символьного исполнения, с.**	**Степень покрытия кода, %**
maze-org	43,48	31,53
maze-sub	43,56	31,58
maze-fla	244,85	34,61
maze-bcf	40,41	35,43

Как видно из таблицы, применение обфусцирующих преобразований изменяет время символьного исполнения программы. В то же время степень покрытия кода показала неоднозначные значения. Обфусцирующее преобразование Control Flow Flattening значительно увеличило время символьного исполнения, в то время как остальные обфусцирующие преобразования не оказали заметного эффекта на время символьного исполнения.

Заключение

В настоящей работе разработана схема автоматической оценки стойкости обфусцирующих преобразований, основанная на вычислении характеристик трех версий программы: исходной программы, обфусцированной программы и деобфусцированной программы. Предложены характеристики, которые могут быть использованы в рамках построенной модели на основе похожести программ. Построена реализация алгоритма вычисления характеристик на основе символьного исполнения. Эксперименты показали, что обфусцирующие преобразования влияют на характеристики, на основе которых вычисляются предложенные показатели похожести.

Дальнейшим направлением исследования является проведение экспериментов для оценки применимости предложенных показателей похожести (3), (6), (4) и (5), разработка новых показателей, в частности, на основе методов машинного обучения, а также исследование корреляции различных показателей похожести.

Список литературы / References

[1] Siegmund J., “Program Comprehension: Past, Present, and Future”, *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, **5** (2016), 13–20.

[2] Avidan E., Feitelson D. G., “From Obfuscation to Comprehension”, *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension*, 2015, 178–181.

[3] Поздеев А. Г., Кривопалов В. Н., Ромашкин Е. В., Радченко Е. Д., “Математические и программные средства обфускации программ”, *ПДМ*, **1** (2009), 52–53; [Pozdeev A. G., Krivopalov V. N., Romashkin E. V., Radchenko E. D., “Matematicheskie i programmnye sredstva obfuskacii programm”, *PDM*, **1** (2009), 52–53, (in Russian).]

[4] Чернов А. В., “Анализ запутывающих преобразований программ”, 2002, http://www.citforum.ru/security/articles/analysis/; [Chernov A. V., “Analiz zaputyvayushchih preobrazovanij programm”, 2002, http://www.citforum.ru/security/articles/analysis/, (in Russian).]

[5] Kuzurin N., Shokurov A., Varnovsky N., Zakharov V., “On the Concept of Software Obfuscation in Computer Security”, *International Conference on Information Security. – Springer, Berlin, Heidelberg*, 2007, 281–298.

[6] Diffie W., Hellman M., “New directions in cryptography”, *IEEE Transactions on Information Theory*, **22**:6 (1976), 644–654.

[7] Collberg C. S., Thomborson C., “Watermarking, Tamper-Proofing, and Obfuscation Tools for Software Protection”, *IEEE transactions on software engineering*, **28**:8 (2002), 735–746.

[8] Lee B., Kim Y., Kim J., “binOb+: a Framework for Potent and Stealthy Binary Obfuscation”, *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security, ASIACCS 2010*, 2010, 271–281.

[9] Borello J.M., Me L., “Code Obfuscation Techniques for Metamorphic Viruses”, *Journal in Computer Virology*, **4**:3 (2008), 211–220.

[10] Moser A., Kruegel C., Kirda E., “Limits of Static Analysis for Malware Detection”, *Proceedings of Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, 2007, 421–430.

[11] Baiardi F., Sgandurra D., “An obfuscation-based approach against injection attacks”, *Proceedings of the Sixth International Conference on Availability, Reliability and Security (ARES)*, 2011, 51–58.

[12] Нурмухаметов А. Р., "Применение диверсифицирующих и обфусцирующих преобразований для изменения сигнатуры программного кода", *Труды ИСП РАН*, **28**:5 (2016), 93–104; [Nurmukhametov A. R., "Primenenie diversificiruyushchih i obfusciruyushchih preobrazovanij dlya izmeneniya signatury programmnogo koda", *Trudy ISP RAN*, **28**:5 (2016), 93–104, (in Russian).]

[13] Косолапов Ю. В., "Об обнаружении атак типа повторного использования исполнимого кода", *Моделирование и анализ информационных систем*, **26**:2 (2019), 213–228; [Kosolapov Y. V., "About Detection of Code Reuse Attacks", *Modeling and Analysis of Information Systems*, **26**:2 (2019), 213–228, (in Russian).]

[14] Collberg C., Thomborson C., Low D., "A taxonomy of obfuscating transformations", *Technical Report 148, The University of Auckland, New Zealand*, 1997.

[15] Walenstein A., El-Ramly M., Cordy J. R., Evans W. S., Mahdavi K., Pizka M., Ramalingam G., von Gudenberg J. W., "Similarity in Programs", *Duplication, Redundancy, and Similarity in Software*, 2007, 1–8.

[16] Chipounov V., Kuznetsov V., Candea G., "The S2E Platform: Design, Implementation, and Applications", *ACM Transactions on Computer Systems*, **30**:1 (2012), 1–49.

[17] Saudel F., Salwan J., "Triton: A Dynamic Symbolic Execution Framework", *Symposium Sur La Security Des Technologies de L'information et Des Communications, SSTIC*, 2015, 31–54.

[18] Wang Z, Ming J., Jia C., Gao D., "Linear Obfuscation to Combat Symbolic Execution", *Proceedings of Computer Security - ESORICS 2011*, **6879** (2011), 210–226.

[19] Brumley D., Hartwig C., Liang Z., Newsome J., Song D., Yin H., "Automatically Identifying Trigger-based Behavior in Malware", *Botnet Detection. Advances in Information Security*, **36** (2008), 65–88.

[20] King J. C., "Symbolic execution and program testing", *Communications of the ACM*, **19**:7 (1976), 385–394.

[21] Cadar C., Dunbar D., Engler D. R., "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs", *8th USENIX Symposium on Operating Systems Design and Implementation*, **8** (2008), 209–224.

[22] Shoshitaishvili Y.et al., "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis", *IEEE Symposium on Security and Privacy*, 2016, 138–157.

[23] Sharif M. I., Lanzi A., Giffin J. T., Lee W., "Impeding Malware Analysis Using Conditional Code Obfuscation", *Proceedings of NDSS*, 2008, 1–13.

[24] Udupa S. K., Debray S. K., Madou M., "Deobfuscation: Reverse Engineering Obfuscated Code", *Proceedings of the 12th Working Conference on Reverse Engineering (WCRE'05)*, 2005, 44–53.

[25] Nagarajan V., Gupta R., Zhang X., Madou M., De Sutter B., "Matching Control Flow of Program Versions", *IEEE International Conference on Software Maintenance*, 2007, 84–93.

[26] Bonfante G., Kaczmarek M., Marion J.Y., "Control Flow Graphs as Malware Signatures", *International Workshop on the Theory of Computer Viruses*, 2007, 1–6.

[27] Park Y., Reeves D., Mulukutla V., Sundaravel B., "Fast Malware Classification by Automated Behavioral Graph Matching", *Proceedings of the Sixth Annual Workshop on Cyber Security and Information Intelligence Research*, 2010, 1–4.

[28] Kinable J., Kostakis O., "Malware classification based on call graph clustering", *Journal in Computer Virology*, **7**:4 (2011), 233–245.

[29] Lim H.I., "Comparing Control Flow Graphs of Binary Programs through Match Propagation", *IEEE 38th Annual Computer Software and Applications Conference*, 2014, 598–599.

[30] Dullien T., Rolles R., "Graph-based comparison of executable objects", **5**:1 (2005), 1–8.

[31] Chan P.P.F., Collberg C., "A Method to Evaluate CFG Comparison Algorithms", *14th International Conference on Quality Software*, 2014, 95–104.

[32] Axenovich M., Kezdy A., Martin R., "On the editing distance of graphs", *J. Graph Theory*, **58**:2 (2008), 123–138.

[33] Борисов П. Д., Косолапов Ю. В., "О выборе характеристик для оценки стойкости обфусцирующих преобразований", *Современные информационные технологии: тенденции и перспективы развития. Труды XXVI научной конференции СИТО-2019.*, 2019, 42–44; [Borisov P. D., Kosolapov Y. V., "O vybore harakteristik dlja ocenki stojkosti obfuscirujushhih preobrazovanij", *Sovremennye informacionnye tekhnologii: tendencii i perspektivy razvitiya. Trudy XXV nauchnoj konferencii SITO-2019.*, 2019, 42–44, (in Russian).]

[34] Lehman M. M., Belady L. A. Program Evolution. Processes of Software Change., *Academic press*, 1985.

[35] Schnappinger M., Osman M. H., Pretschner A., Pizka M., Fietzke A., "Software Quality Assessment in Practice: a Hypothesis-Driven Framework", *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2018, 1–6.

[36] Борисов П. Д., Косолапов Ю. В., "Модель экспериментального анализа стойкости алгоритмов обфускации", *Современные информационные технологии: тенденции и перспективы развития. Труды XXV научной конференции СИТО-2018.*, 2018, 37–39; [Borisov P. D., Kosolapov Y. V., "Model' eksperimental'nogo analiza stojkosti algoritmov obfuskacii", *Sovremennye informacionnye tekhnologii: tendencii i perspektivy razvitiya. Trudy XXV nauchnoj konferencii SITO-2018.*, 2018, 37–39, (in Russian).]

[37] "IDA Pro", https://www.hex-rays.com/products/ida/.

[38] "The LLVM Compiler Infrastructure", https://llvm.org/.

[39] "McSema", https://github.com/trailofbits/mcsema.

[40] Junod P., Rinaldini J., Wehrli J., Michieliny J., "Obfuscator-LLVM – Software Protection for the Masses", *Conference: 2015 IEEE/ACM 1st International Workshop on Software Protection (SPRO)*, 2015, 3–9.

Borisov P. D., Kosolapov Y. V., "On the Automatic Analysis of the Practical Resistance of Obfusting Transformations", *Modeling and Analysis of Information Systems*, **26**:3 (2019), 317–331.

DOI: 10.18255/1818-1015-2019-3-317-331

Abstract. A method is developed for assessing the practical persistence of obfuscating transformations of programs based on the calculation of the similarity index for the original, obfuscated and deobfuscated programs. Candidates are proposed for similarity indices, which are based on such program characteristics as the control flow graph, symbolic execution time and degree of coverage for symbolic execution. The control flow graph is considered as the basis for building other candidates for program similarity indicators. On its basis, a new candidate is proposed for the similarity index, which, when calculated, finds the Hamming distance between the adjacency matrices of control flow graphs of compared programs. A scheme for estimating (analyzing) the persistence of obfuscating transformations is constructed, according to which for the original, obfuscated and deobfuscated programs, the characteristics of these programs are calculated and compared in accordance with the chosen comparison model. The developed scheme, in particular, is suitable for comparing programs based on similarity indices. This paper develops and implements one of the key units of the constructed scheme - a block for obtaining program characteristics compiled for the x86/x86_64 architecture. The developed unit allow to find the control flow graph, the time for symbolic execution and the degree of coverage for symbolic execution. Some results of work of the constructed block are given.

Keywords: code obfuscation, resistance, symbolic execution

On the authors:
Petr D. Borisov, orcid.org/0000-0002-8919-8310, graduate student,
Southern Federal University,
8a Milchakova str., Rostov-on-Don 344090, Russia, e-mail: borisovpetr@mail.ru

Yury V. Kosolapov, orcid.org/0000-0002-1491-524X, PhD,
Southern Federal University,
8a Milchakova str., Rostov-on-Don 344090, Russia, e-mail: itaim@mail.ru