УДК 004.056 **DOI** 10.15622/ia.24.5.9

П.Д. Борисов, Д.В. Варламов, Ю.В. Косолапов

АЛГОРИТМ ВЫЧИСЛЕНИЯ ПОХОЖЕСТИ ГРАФОВ И ЕГО ПРИМЕНЕНИЕ ДЛЯ СРАВНЕНИЯ БИНАРНЫХ ИСПОЛНЯЕМЫХ ФАЙЛОВ

Борисов П.Д., Варламов Д.В., Косолапов Ю.В. **Алгоритм вычисления похожести графов и его применение для сравнения бинарных исполняемых файлов.**

Аннотация. Рассматривается задача статического (без запуска) сравнения бинарных исполняемых файлов. Программа и любая ее процедура могут быть представлены в виде ориентированного графа. Для программы соответствующий граф представляет собой граф вызова функций (процедур), где узлами являются сами функции, а ребро из вершины a в b описывает вызов функции b из функции a. Для процедуры такой граф представляет собой граф потока управления, где вершинами являются базовые блоки, а ребро между узлами a и b означает возможное исполнение команд блока b после исполнения команд блока a. В работе предлагается алгоритм сравнения направленных графов, который далее применяется для сравнения программ. В основе алгоритма сравнения графов лежит применение функции похожести узлов. Для сравнения графов процедур в качестве такой функции похожести применяются нечеткая (fuzzy) хеш-функция и криптографическая хеш-функция. Далее этот способ сравнения графов процедур используется как функция похожести узлов при сравнении графов программ. На базе предложенного алгоритма разработан метод сравнения программ, проведено его исследование в рамках двух экспериментов. В первом эксперименте исследовано поведение метода при сравнении программ, полученных с применением разных опций оптимизации (O0, O1, O2, O3 и Os). Во втором эксперименте исследована возможность выявления эффективных и стойких обфусцирующих преобразований в рамках ранее разработанной модели. В первом эксперименте получены свидетельства в пользу верности гипотезы об уменьшении похожести файлов с ростом оптимизации от O1 до O3. Во втором эксперименте подтверждены некоторые полученные ранее результаты, касающиеся эффективности (неэффективности) и стойкости (нестойкости) обфусцирующих преобразований.

Ключевые слова: сравнение графов, похожесть программ, эффективность и стойкость обфускации.

1. Введение. В области защиты информации *различители* играют важную роль. В общем виде различитель можно определить как функцию dis, принимающую в качестве аргумента два объекта, отличие в которых находится, и возвращающую значение из отрезка действительных чисел $[0, 1]$. Чем ближе значение, возвращаемое различителем, к 1,

тем больше отличие в сравниваемых объектах, и наоборот (нулевое значение соответствует полному совпадению сравниваемых объектов в выбранном пространстве признаков). Например, в криптографии наличие эффективного различителя, позволяющего отличить (с непренебрежимой вероятностью) вариант использования криптографической схемы, когда в ней задействуется «неслучайный» криптографический элемент/примитив (например, помехоустойчивый код), от варианта, когда применяется «случайный» аналог этого примитива, часто свидетельствует о слабости этой криптографической схемы [1]. В стеганографии стойкость стегосистемы (стойкость к обнаружению скрытого канала передачи информации) зависит от того, насколько вычислительно сложно отличить стегообъект (объект, содержащий скрытые данные) от чистого покрывающего объекта (объекта, не содержащего встроенных данных) [2]. Решение задачи распознавания помехоустойчивого кода, используемого в канале связи, также часто основывается на построении различителя [3]. В компьютерной безопасности системы защиты от спама, атак и от вредоносного кода обычно строятся на основе различителей. Именно, при защите от спама используются алгоритмы отличения входящей корреспонденции (электронных писем, мгновенных сообщений), относящейся к нежелательной (к спаму), от остальной [4]. При обнаружении атак различители позволяют отличить «нормальное» состояние (состояние сети или состояние кода, исполняемого в текущий момент на процессоре) от аномального [5–7]. А при проверке файлов на наличие вредоносного кода различители используются как дополнительный механизм выявления ранее неизвестного или замаскированного вредоносного кода, когда проверка по базе сигнатур не дает результата [8, 9]. В [10] предложен способ оценки эффективности и стойкости обфусцирующих преобразований программ, в основе которого лежит применение функции похожести sim, определённой на парах исполняемых файлов и возвращающей значение от 0 до 1. На основе этой функции естественным образом может быть построен различитель dis для произвольной пары программ P_1 и P_2: $\mathsf{dis}(P_1, P_2) = 1 - \mathsf{sim}(P_1, P_2)$. Поэтому естественно считать, что модель из [11] основана на применении различителя, определённого на парах программ.

В настоящей работе предлагается способ построения функции sim, которая может служить, с одной стороны, как дополнительный показатель похожести в рамках функции похожести программ, предложенной в [11] на основе методов машинного обучения, с другой – может применяться самостоятельно при оценке структурной похожести файлов исполняемых программ.

Работа, кроме введения и заключения, содержит четыре раздела. Во втором разделе кратко приводится обзор подходов к сравнению файлов. Третий раздел посвящен построению алгоритма сравнения ориентированных графов. На основе этого алгоритма в четвертом разделе строится метод сравнения исполняемых файлов. Экспериментальному исследованию этого метода посвящен пятый раздел.

2. Обзор литературы. Как отмечено в [11], условно алгоритмы сравнения программ можно разделить на *универсальные* и *специализированные*. В первом случае алгоритмы сравнения не используют информацию о типе сравниваемых файлов: эти файлы рассматриваются как последовательности байтов. К таким относятся, например, алгоритмы нечетких хеш-функций ssdeep [12], mrsh-v2 [13], tlsh и sdhash [14]. Независимость от типа сравниваемых файлов позволяет применять эти функции в различных сценариях обработки информации, в частности, в сценариях защиты от спама, при поиске вредоносного кода, похожего на уже известный код и т.п. В [15] показано, что при сравнении исполняемых программ указанные выше функции показывают низкую точность обнаружения похожих файлов. Например, изменение одной ассемблерной инструкции может привести к падению до нуля значения похожести. Специализированные алгоритмы сравнения, в свою очередь, учитывают структуру исполняемых файлов, поэтому применимы только к файлам заданного типа. Однако, с другой стороны, это позволяет сохранить высокую точность сравнения при незначительных изменениях файлов. К таким алгоритмам можно отнести bindiff [16], bcc [17], machoc [18], machoke [19] и tah [20], которые основаны на сравнении графов процедур. В частности, алгоритмы bindiff и bcc находят похожие процедуры в сравниваемых бинарных файлах путем применения ряда эвристик (при сопоставлении графов процедур), позволяющих судить о семантической эквивалентности сравниваемых процедур. К таким эвристикам, в частности, относится сравнение хеш-значений сравниваемых процедур. Дополнительное использование в bcc графов зависимостей от данных и управления (PDG – program dependence graph) позволяет повысить долю верно найденных семантически эквивалентных процедур, по сравнению с bindiff. В алгоритмах machoc и machoke для сравниваемых процедур во время обхода соответствующих графов строятся последовательности хеш-значений базовых блоков, после чего вычисляются финальные хеш-значения от этих последовательностей и выполняется их сравнение. Стоит отметить, что алгоритмы machoc и machoke отличаются только способами получения графовых представлений процедур: в первом

случае используется дизассемблер IDA Pro, а во втором – Radare2. Алгоритм tah также строит хеш-значение процедуры, основываясь на ее графе, однако, строит это значение, основываясь не на содержимом базовых блоков, а на основе связей между блоками.

В [11] экспериментально показано, что при совместном использовании специализированных и универсальных функций похожести в моделях сравнения файлов на основе методов машинного обучения наибольший вклад в принятие решения дают специализированные функции. Таким образом, пополнение арсенала специализированных функций похожести может позволить, например, выполнять более точную оценку эффективности и стойкости обфусцирующих преобразований в рамках модели, предложенной в [10].

3. Алгоритм сравнения графов. Программы и процедуры (далее функции, из которых состоит программа, будут называться процедурами, чтобы исключить конфликт с термином «функция похожести») часто представляются направленными графами. Под направленным графом далее будем понимать пару $G = \langle V, E\rangle$ с множеством вершин V и множеством ребер E. Естественной представляется разработка алгоритма сравнения абстрактных графов $G^1 = \langle V^1, E^1\rangle$ и $G^2 = \langle V^2, E^2\rangle$ с последующим применением его для сравнения программ. Далее описывается алгоритм sim_G сравнения графов (алгоритм 1), идея которого предложена в [21].

Алгоритм 1. Алгоритм сравнения графов sim_G

Вход: $G^i = \langle V^i, E^i\rangle$, sim_v
Выход: $s \in [0, 1]$

1: Построить список L вида (1); $l = 0$
2: **while** $L \neq \varnothing$ **do**
3: Выбрать первый элемент $(v, \tilde{v}, c)$ из L.
4: $\mathsf{tag}_{G^1}(v) \leftarrow l$, $\mathsf{tag}_{G^2}(\tilde{v}) \leftarrow l$.
5: Удалить из L все тройки $(v', \tilde{v}', c')$, в которых $v' = v$ или $\tilde{v}' = \tilde{v}$.
6: $l = l + 1$
7: **end while**
8: $i = \mathrm{argmax}_{k\in\{1,2\}}|V^k|$, $r = ||V^1| - |V^2||$
9: Назначить узлам графа G^i в произвольном порядке метки с номерами от l до $l + r - 1$.
10: Добавить в граф G^{3-i} *фиктивные* узлы без связей; присвоить этим узлам уникальные значения начальных меток, а новым меткам этих вершин присвоить значения от l до $l + r - 1$.
11: По графу G^i построить матрицу смежности вершин $M^i = M(G^i)$, $i = 1, 2$, в которой строки и столбцы упорядочены по возрастанию значений новых меток от 0 до $l + r - 1$.
12: Найти $\rho(G^1, G^2)$ в соответствии с (4).
return $\rho(G^1, G^2)$.

Алгоритм sim_G сравнения графов G^1 и G^2 основан на применении функции похожести $\mathsf{sim}_v : V^1 \times V^2 \to [0,1]$. Значение $\mathsf{sim}_v(v,\tilde{v})$ можно рассматривать как похожесть узла $v \in V^1$ на узел $\tilde{v} \in V^2$: чем ближе значение к 1, тем узлы более похожи, и наоборот. Предполагается, что узлы сравниваемых графов имеют две метки: начальную метку и новую метку. Начальная метка узла v совпадает с v, а новую метку графа G будем обозначать $\mathsf{tag}_G(v)$. В алгоритме 1 узлы двух графов сопоставляются на основе функции sim_v. Сначала по графам G^1 и G^2 строится список троек

$$L = \{(v,\tilde{v},c) : (v,\tilde{v}) \in V^1 \times V^2, c = \mathsf{sim}_v(v,\tilde{v})\}, \tag{1}$$

упорядоченный по убыванию значения c. Таким образом, первый элемент этого списка будет содержать начальные метки v и $\tilde{v}$ двух вершин (одной из графа G^1, другой из G^2), которые имеют наибольшую похожесть в смысле sim_v. Естественно этим узлам присвоить одинаковое значение l новой метки (в начале алгоритма значение новой метки равно $l = 0$). Далее из списка L удаляется первая тройка, а также все тройки, в которых первый элемент равен v или второй равен $\tilde{v}$; значение l увеличивается на единицу, и процедура присваивания новых меток наиболее похожим узлам повторяется до тех пор, пока список L не пуст. Отметим, что сравниваемые графы могут иметь разное число вершин: $r_1 = |V^1| \neq r_2 = |V^2|$ (здесь и далее запись $|A|$ в случае, когда A – конечное множество, означает количество элементов в этом множестве, а в случае, когда A – число, означает модуль этого числа). В графе, имеющем большее число вершин, некоторые вершины не будут иметь новых меток. Новым меткам этих вершин в произвольном порядке могут быть присвоены значения от $\min\{r_1,r_2\}$ до $\max\{r_1,r_2\} - 1$. Граф с большим числом вершин обозначим G^+. В граф с меньшим числом вершин, который обозначим G^-, добавляются $\max\{r_1,r_2\} - \min\{r_1,r_2\}$ *фиктивных* вершин для выравнивания числа вершин в сравниваемых графах. Для каждой из фиктивных вершин значение начальной метки выбирается уникальным (в рамках графа G^-), а новые метки этих вершин принимают значение от $\min\{r_1,r_2\}$ до $\max\{r_1,r_2\}-1$. Множество добавленных в G^- фиктивных вершин обозначим $F(G^-)$. Будем считать, что ни одна вершина из $F(G^-)$ непохожа ни на одну из вершин графа G^+:

$$\forall v \in F(G^-), \forall \tilde{v} \in V^+ : \ \mathsf{sim}_v(v,\tilde{v}) = 0. \tag{2}$$

Пусть $M^1 = (m^1_{i,j})$ и $M^2 = (m^2_{i,j})$ – матрицы смежности вершин графов G^1 и G^2 после процедуры назначения новых меток и добавления

фиктивных вершин (если требуется). Отметим, что $m^k_{i,j} = 1$, если в графе G^k из вершины с новой меткой i выходит ребро в вершину с новой меткой j; в противном случае $m^k_{i,j} = 0$, $k = 1, 2$. Пусть

$$\Delta_{i,j} = |1 - |m^1_{i,j} - m^2_{i,j}||. \tag{3}$$

Таким образом, $\Delta_{i,j} = 1$, если $m^1_{i,j} = m^2_{i,j}$, иначе $\Delta_{i,j} = 0$. Результат сравнения графов основан на вычислении нормированной взвешенной разности между матрицами смежности вершин этих графов, которая с учетом (3), здесь определяется следующим образом:

$$\begin{aligned}\rho(G^1, G^2) &= \frac{\sum_{i,j=0}^{\max\{r_1,r_2\}} \Delta_{i,j} \cdot (\mathsf{sim}_v(v^1_i, v^2_i) + \mathsf{sim}_v(v^1_j, v^2_j))}{2 \cdot (\max\{r_1, r_2\})^2} \\ &= \frac{\sum_{i,j=0}^{\min\{r_1,r_2\}} \Delta_{i,j} \cdot (\mathsf{sim}_v(v^1_i, v^2_i) + \mathsf{sim}_v(v^1_j, v^2_j))}{2 \cdot (\max\{r_1, r_2\})^2},\end{aligned} \tag{4}$$

где v^j_k – начальная метка узла в графе G^j, имеющего новую метку $\mathsf{tag}(v^j_k) = k$, $j = 1, 2$. Равенство (4) вытекает из условия (2). Стоит отметить, что в [21] непохожесть графов предлагалось вычислять путем нахождения нормированного расстояния Хэмминга между матрицами M^1 и M^2. Другими словами, в [21] похожесть имеет вид:

$$\rho'(G^1, G^2) = \frac{\sum_{i,j=0}^{\max\{r_1,r_2\}} \Delta_{i,j}}{(\max\{r_1, r_2\})^2},$$

где похожесть узлов не учитывается напрямую, а только учитывается при назначении новых меток. Представляется, что формула (4) точнее описывается похожесть сравниваемых графов.

На рисунке 1 в качестве примера изображены два сравниваемых графа $G^1 = (V^1, E^1)$ и $G^2 = (V^2, E^2)$: $V^1 = \{a_1, a_2, a_3, a_4\}$, $V^2 = \{b_1, b_2, b_3, b_4, b_5\}$. В таблице 1 приведены значения *некоторой* функции похожести $\mathsf{sim}_v : V^1 \times V^2 \to [0, 1]$. В этой таблице жирным шрифтом выделены значения c, соответствующие тройкам $(v, \tilde{v}, c)$, выбираемым на шаге 3 алгоритма sim_G в цикле (шаги с 2 по 7): на первой итерации первой тройкой в списке будет $(b_3, a_1, 0, 8)$, на второй – тройка $(b_1, a_2, 0, 75)$ и т.д.. В этой же таблице серой заливкой отмечены ячейки для фиктивного узла a_5, который абсолютно непохож ни на один узел графа G^2:

соответствующие значения функции похожести полагаются равными нулю. Матрицы смежности M^1 и M^2 вершин, построенные на основе новых меток, приведены в таблице 2. Серой заливкой в матрице смежности M^1 выделены столбец и строка для фиктивного узла.

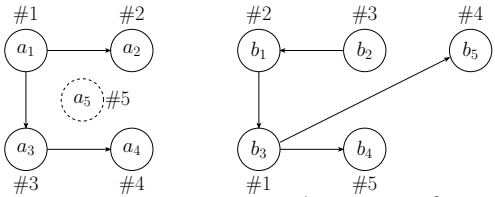

Рис. 1. Пример двух сравниваемых графов G^1 (слева) и G^2 (справа). Начальные метки узлов указаны внутри узлов, а новые метки – рядом с узлами после символа «#». Фиктивный узел, добавленный в граф G^1, обозначен пунктирной линией

Таблица 1. Значения функции похожести $\mathsf{sim}_v : V^1 \times V^2 \to [0, 1]$ (пример)

$\mathsf{sim}_v(\cdot,\cdot)$	a_1	a_2	a_3	a_4	a_5
b_1	0,1	**0,75**	0,23	0,47	0
b_2	0,2	0,1	**0,71**	0,01	0
b_3	**0,8**	0,5	0,1	0,4	0
b_4	0,65	0,74	0,62	**0,31**	0
b_5	0,1	0,6	0,58	0,3	0

Таблица 2. Матрицы смежности вершин графов G^1 и G^2 из примера

M^1	#1	#2	#3	#4	#5
#1	0	1	1	0	0
#2	0	0	0	0	0
#3	0	0	0	1	0
#4	0	0	0	0	0
#5	0	0	0	0	0

M^2	#1	#2	#3	#4	#5
#1	0	0	0	1	1
#2	1	0	0	0	0
#3	0	1	0	0	0
#4	0	0	0	0	0
#5	0	0	0	0	0

Тогда, в соответствии с (4), имеем (в скобках оставлены только ненулевые слагаемые):

$$\begin{aligned}\rho(G^1, G^1) = 50^{-1} \cdot ((0,8 + 0,8)+ \\ + (0,75 + 0,75) + (0,75 + 0,71) + (0,75 + 0,31)+ \\ + (0,71 + 0,8) + (0,71 + 0,71)+ \\ + (0,31 + 0,8) + (0,31 + 0,75) + (0,31 + 0,71) + (0,31 + 0,31)) \\ = 0,2472.\end{aligned}$$

4. Функция похожести для исполняемых файлов. На основе алгоритма 1 предлагается построение функции похожести для программ.

Именно, программу P представим как направленный граф $G^P = \langle V^P, E^P \rangle$, где $V^P = \{f_1, ..., f_r\}$ – множество процедур, $E \subseteq V^P \times V^P$ – множество связей между процедурами, причем $(f_i, f_j) \in E$, если из процедуры f_i выполняется обращение к f_j. Такой граф обычно называется графом вызова функций (call function graph). Начальными метками вершин в G^P могут быть, например, названия процедур. Каждую из процедур $f \in V^P$ также можно представить в виде графа $G^f = \langle V^f, E^f \rangle$, где $V^f = \{b_1^f, ..., b_{n_f}^f\}$ – множество базовых блоков процедуры f, то есть таких последовательностей команд, среди которых команда передачи управления находится только в конце последовательности, $E^f \subseteq V^f \times V^f$ – множество связей между базовыми блоками процедуры f. Ребро (b_i^f, b_j^f) принадлежит E^f, если последняя команда блока b_i^f передает управление на блок b_j^f. Граф G^f называется графом потока управления (control flow graph). Начальной меткой каждого узла в G^f может быть, например, адрес первой команды соответствующего базового блока.

Пусть для программ P_1 и P_2 стоит задача оценить их статическую похожесть (без запуска программ). Так как P_1 и P_2 представимы своими графами G^{P_1} и G^{P_2}, сравнение может заключаться в построении подходящей функции похожести sim_f для процедур программ и применении алгоритма 1: $\mathsf{sim}_G(G^{P_1}, G^{P_2}, \mathsf{sim}_f)$. В свою очередь, процедуры программ также представимы графами, узлами в которых являются базовые блоки. Поэтому для сравнения процедур также может быть применён алгоритм 1, если построить подходящую функцию для сравнения базовых блоков. В настоящей работе сравнение базовых блоков предлагается выполнять с помощью хеш-функций $h_i : \{0,1\}^* \to \{0,1\}^{m_i}$, $i = 1, 2$. При этом будем предполагать, что h_1 – криптографическая хеш-функция, а h_2 – нечеткая хеш-функция с алгоритмом сравнения $\mathsf{score} : \{0,1\}^* \times \{0,1\}^* \to [0,1]$. С каждым базовым блоком $b \in V^f$ процедуры $f \in V^P$ свяжем пару $(h_1(b), h_2(b))$, где $h_i(b)$ – хеш-значение от последовательности ассемблерных команд этого базового блока, $i = 1, 2$. Использование криптографической хеш-функции h_1 позволяет находить идентичные блоки, так как в этом случае хеш-значения совпадают. В случае, когда блоки неидентичны, использовать для сравнения значения криптографической функции не имеет смысла, так как разница аргументов в одном бите приведет в среднем к разнице половины битов значений. Именно поэтому для сравнения неидентичных блоков предлагается использовать нечеткую хеш-функцию h_2. Функция сравнения sim_{bb} базовых блоков b_1 и b_2, функция сравнения процедур f_1

и f_2, функция сравнения программ P_1 и P_2 соответственно имеют вид:

$$\mathsf{sim}_{bb}(b_1, b_2) = \begin{cases} 1, \text{если } h_1(b_1) = h_1(b_2), \\ \mathsf{score}(h_2(b_1), h_2(b_2)), \text{иначе}, \end{cases} \tag{5}$$

$$\mathsf{sim}_f(f_1, f_2) = \mathsf{sim}_G(G^{f_1}, G^{f_2}, \mathsf{sim}_{bb}), \tag{6}$$

$$\mathsf{sim}_P(P_1, P_2) = \mathsf{sim}_G(G^{P_1}, G^{P_2}, \mathsf{sim}_f). \tag{7}$$

Далее функцию sim_P будем называть simgraph, подчеркивая, что похожесть вычисляется на основе сравнения графов программ.

В настоящей работе предложенный алгоритм сравнения программ реализован[1] на языке Python с использованием фреймворка дизассемблирования Radare2. В качестве криптографической хеш-функции h_1 и нечеткой хеш-функции h_2 используются соответственно md5 и ssdeep. Выбор в пользу функции md5 с длиной хеш-значения 128 битов обосновывается тем, что число базовых блоков в сравниваемых процедурах существенно меньше $2^{128/2}$, чем обеспечивается пренебрежимая вероятность коллизии (когда два разных базовых блока дают одно и то же хеш-значение). Естественно, что в качестве h_1 может применяться практически любая криптографическая хеш-функция. Более того, от функции h_1 в решаемой задаче требуется только пренебрежимая вероятность коллизии, поэтому h_1 не обязана быть криптографической. Но так как криптографические хеш-функции имеют пренебрежимую вероятность коллизии, то для простоты в функции sim_{bb} (см. формулу (5)) предполагается, что h_1 именно такая. Выбор же функции ssdeep в качестве h_2 обосновывается тем, что эта функция позволяет сравнивать строковые данные произвольной длины, в то время как функция tlsh на вход принимает данные не менее 50 байтов [14], функция mrsh-v2 – не менее 160 байтов, а функция sdhasdh – не менее 512 байтов [14] [13]. Так как не все базовые блоки имеют длину более 50 байтов, то в качестве h_2 выбрана именно функция ssdeep. Одним из дальнейших направлений исследования является замена функции ssdeep на функцию, вычисляющую расстояние редактирования строк (расстояние Левенштейна), которая применима при сравнении строк любой длины.

5. Результаты экспериментов. В работе проведено два эксперимента (далее – эксперимент 1 и эксперимент 2). В первом эксперименте исследовалось поведение значений построенной функции

[1]Реализация размещена на сайте GitHub по адресу https://github.com/Tempiress/executableFilesComparison

похожести на множестве пар функционально одинаковых программ, полученных с помощью различных компиляторов и различных опций оптимизации. Ожидается, что для более агрессивных опций оптимизации значения похожести будут меньше, чем для менее агрессивных. Во втором эксперименте исследовалось применение построенной функции похожести в модели оценки эффективности и стойкости обфусцирующих преобразований, предложенной в [10]. Во всех экспериментах для сравнения значений, полученных на основе функции simgraph, вычисляется похожесть программ на основе нечетких хеш-функций ssdeep и tlsh, а также разработанной в [22] функции похожести исполняемых файлов, представленных в виде изображений (далее эта функция называется simpic).

5.1. Эксперимент 1. Формирование множества пар функционально одинаковых программ реализовано на основе построенного в [10] множества исполняемых файлов. Именно, множество пар формируется на основе наборов программ CoreUtils, PolyBench и HashCat (всего 164 программы), собранных девятью компиляторами – GCC (версий 7.5.0, 8.4.0, 9.4.0, 10.3.0), Clang (версий 7.0.1, 8.0.1, 9.0.1, 10.0.0) и AOCC (версии 3.0.0) – с пятью опциями оптимизации: O0, O1, O2, O3 и Os. Таким образом, множество исполняемых файлов, построенное в [10], состоит из 7380 файлов. Так как сложность алгоритма sim_G – $O(|V^1| \times |V^2|)$ (все узлы одного графа сравниваются со всеми узлами другого графа), то для сокращения времени эксперимента из 164-х программ были случайно выбраны 82 программы. Каждая выбранная программа, скомпилированная фиксированным компилятором с опцией O0, сравнивалась с версией той же программы, скомпилированной тем же компилятором, но с опциями O1, O2, O3 и Os. Таким образом, было выполнено $82 \cdot 9 \cdot 4 = 2952$ сравнения с помощью функций simgraph, ssdeep, tlsh и simpic. Значения, возвращаемые функциями ssdeep и tlsh, нормализовались в соответствии с правилами из [15]. Результаты представлены в таблице 3.

Ожидается, что с ростом оптимизации от O1 до O3 значение функции похожести должно падать, так как в рассматриваемых компиляторах опция O2 включает преобразования, подключаемые опцией O1, а опция O3 включает преобразования, задаваемые опцией O2. Сразу отметим, что ни для одной функции эта тенденция не наблюдается на всех рассматриваемых компиляторах. В то же время, для функций simgraph и tlsh на всех компиляторах программы, полученные с опциями O2 и O3 всегда менее похожи на неоптимизированную программу (O0), чем программа полученная с помощью опции O1. При этом функция simgraph

более чувствительна к действию оптимизирующих преобразований, так как отношение похожести в случае (O0,O1) к похожести в случае (O0,O2) или к похожести в случае (O0,O3) выше, чем соответствующие отношения для функции tlsh. Для функций ssdeep и simpic утверждение о том, что «на всех компиляторах программы, полученные с опциями O2 и O3 всегда менее похожи на неоптимизированную программу», не является верным. Именно, для функции ssdeep это не выполняется для четырех компиляторов: GCC версий 8.4, 9.4, 10.3 и Clang версии 8.0.1. Для функции simpic это не выполняется на компиляторе AOCC и частично на компиляторе Clang версии 9.0.1.

Таблица 3. Усредненные значения функций похожести по 50% программ набора, состоящего из программ пакетов CoreUtils, PolyBench и HashCat

		AOCC	CGG				Clang			
		3.0.0	7.5	8.4	9.4	10.3	7.0.1	8.0.1	9.0.1	10.0.0
simgraph	(O0,O1)	0,087	0,089	0,088	0,088	0,097	0,082	0,082	0,083	0,099
	(O0,O2)	0,067	0,061	0,066	0,066	0,068	0,075	0,075	0,076	0,076
	(O0,O3)	0,067	0,065	0,070	0,071	0,071	0,075	0,075	0,075	0,075
	(O0,Os)	0,065	0,068	0,063	0,062	0,064	0,074	0,074	0,074	0,074
ssdeep	(O0,O1)	0,162	0,038	0,266	0,271	0,324	0,123	0,115	0,116	0,116
	(O0,O2)	0,135	0,016	0,329	0,375	0,449	0,115	0,136	0,113	0,115
	(O0,O3)	0,127	0,016	0,324	0,403	0,410	0,115	0,133	0,113	0,113
	(O0,Os)	0,150	0,016	0,459	0,430	0,444	0,116	0,135	0,116	0,116
tlsh	(O0,O1)	0,243	0,55	0,603	0,617	0,613	0,253	0,257	0,257	0,257
	(O0,O2)	0,19	0,517	0,533	0,543	0,527	0,247	0,243	0,243	0,223
	(O0,O3)	0,187	0,4	0,453	0,393	0,42	0,25	0,253	0,253	0,227
	(O0,Os)	0,137	0,393	0,523	0,537	0,54	0,227	0,203	0,203	0,223
simpic	(O0,O1)	0,634	0,356	0,356	0,355	0,352	0,567	0,555	0,559	0,581
	(O0,O2)	0,640	0,337	0,342	0,339	0,344	0,574	0,544	0,561	0,563
	(O0,O3)	0,675	0,344	0,347	0,350	0,350	0,444	0,463	0,406	0,420
	(O0,Os)	0,567	0,331	0,331	0,328	0,332	0,580	0,552	0,585	0,600

В целом, разработанная функция simgraph демонстрирует наибольшую чувствительность к изменениям, вызванным оптимизацией, по сравнению с функциями ssdeep, tlsh и simpic. Это делает её более подходящей для задач, где требуется более точное сравнение программ на уровне структуры. Рост же похожести в случае (O0,O3) по сравнению со случаем (O0,O2) представляется особенностями компиляторов AOCC и GCC (для всех рассмотренных версий компилятора Clang такой рост не наблюдается), и может являться предметом дальнейших исследований.

5.2. Эксперимент 2. В [10] предложена новая модель оценки эффективности и стойкости обфусцирующих преобразований. Кратко опишем ее. Пусть $\mathcal{O}$ – множество базовых обфусцирующих преобразований. На основе базовых преобразований могут быть

построены последовательности обфусцирующих преобразований. Множество всех (технически реализуемых) последовательностей, которые могут быть построены на основе преобразований из $\mathcal{O}$, в соответствии с [10], обозначим $\mathcal{O}_0^*$. Эффективность e и стойкость r обфусцирующего преобразования $\mathsf{Obf} \in \mathcal{O}_0^*$, примененного к программе P, определены в [10] следующим образом:

$$e(\mathsf{Obf}, P) = 1 - \mathsf{sim}(\mathsf{Obf}(P)) = \mathsf{dis}(\mathsf{Obf}(P), \mathsf{A}(P_0)),$$
$$r(\mathsf{D}, \mathsf{Obf}, P) = 1 - \mathsf{sim}(\mathsf{D}(\mathsf{Obf}(P)), \mathsf{A}(P_0)) = \mathsf{dis}(\mathsf{D}(\mathsf{Obf}(P)), \mathsf{A}(P_0)),$$

где sim – некоторая функция похожести, D – деобфускатор, по отношению к которому оценивается стойкость Obf, A – аппроксимация «самой понятной» версии P_0 программы P. Набор $\mathcal{M} = (\mathcal{O}, \mathcal{Q} \subset \mathcal{O}_0^*, \mathsf{sim}, \mathsf{D}, \mathsf{A})$ в [10] назван *моделью оценки эффективности и стойкости обфусцирующих преобразований* из множества Q.

Для реализации этой модели необходимо зафиксировать множество $\mathcal{O}$, выбрать функцию похожести sim исполняемых файлов программ, деобфускатор D и способ аппроксимирования A самой понятной программы P_0 для каждого P. В качестве множества $\mathcal{O}$ в эксперименте 2 выбрано множество преобразований из таблицы 4 (множество преобразований, предоставляемых обфускатором Hikari [23]).

Таблица 4. Набор $\mathcal{O}$ преобразований, предоставляемых компилятором Hikari для программ на языке C [10]

	o	**Описание преобразования**	**Тип**
$\mathcal{O}$	bcf	встраивание непрозрачных предикатов	C
	cff	сглаживания графа потока управления	C
	enc	кодирование статических строк	D
	fcw	создание фиктивных функций-прокси	A
	ind	замена инструкций ветвления косвенными переходами	C
	sbb	разбиение базовых блоков	A
	sub	замена инструкций эквивалентными	D
	all	применение всех обфусцирущих преобразований	–

В качестве деобфускатора D использована модель на основе оптимизатора, предложенная и реализованная в [10], а в качестве аппроксимации $\mathsf{A}(P_0)$ – предложенная там же самая короткая версия программы, как правило, получаемая с помощью оптимизации с опцией Os. В качестве функции sim использованы simgraph, ssdeep, tlsh и simpic. В этом эксперименте использованы два набора программ: 20 программ из набора [25] (набор $\mathcal{F}_1$) и 20 случайно выбранных программ из набора, состоящего из пакетов CoreUtils, PolyBench и HashCat (набор $\mathcal{F}_2$).

Заметим, что в последнем столбце таблицы 4 указан тип преобразования, который в [10] определен в соответствии с типами, введенными в [24] для обфускатора Tigress: A – абстрактные преобразования, C – преобразования графа потока управления, D – кодирование данных и арифметических операций. Для каждой последовательности преобразований естественным образом определяется и ее тип. Например, типом последовательности $\mathsf{Obf} = (\mathsf{ind}, \mathsf{fcw}, \mathsf{sbb})$ будет CCA.

В таблицах 5, 6, 7 и 8 представлены типы последовательностей преобразований в упорядоченном по убыванию значений эффективности и стойкости виде.

Таблица 5. Оценка эффективности (e) и стойкости (r) типов обфусцирующих преобразований при использовании функции simgraph

e				r			
$\mathcal{F}_1$		$\mathcal{F}_2$		$\mathcal{F}_1$		$\mathcal{F}_2$	
DAC	0,865	ACA	0,967	all	0,723	CAA	0,902
DAA	0,837	DAA	0,966	CAA	0,71	DAC	0,889
CAD	0,8215	ACD	0,963	ADA	0,706	DAD	0,885
AC	0,821	ADA	0,963	CCC	0,698	ADA	0,884
ACA	0,813	DAD	0,963	AC	0,697	DAA	0,883
AA	0,811	CAA	0,961	AA	0,691	AC	0,881
CAC	0,809	CAD	0,961	AD	0,691	CAC	0,881
ADA	0,806	AC	0,96	DAA	0,691	ACD	0,874
ACD	0,785	AA	0,958	CCA	0,689	AA	0,872
AD	0,785	AD	0,958	ACA	0,688	CAD	0,871
DAD	0,781	DAC	0,958	ACD	0,688	ACA	0,867
CAA	0,781	CAC	0,951	CAC	0,684	AD	0,856
all	0,724	A	0,865	CAD	0,683	DCA	0,831
A	0,72	DCA	0,861	DAD	0,681	A	0,83
CA	0,719	CA	0,856	DAC	0,676	CA	0,818
DCA	0,717	all	0,843	CA	0,665	DA	0,811
DA	0,693	DA	0,834	DCA	0,665	CCA	0,81
CCA	0,693	CCA	0,821	CDA	0,652	CCC	0,81
CCC	0,673	CC	0,787	DCC	0,65	all	0,801
CDA	0,651	CDA	0,786	CC	0,650	CCD	0,792
CCD	0,638	CCC	0,784	A	0,641	CC	0,789
DCC	0,637	CCD	0,781	DA	0,640	CDA	0,788
CC	0,637	C	0,777	CCD	0,633	C	0,777
DC	0,619	CD	0,776	DC	0,620	CD	0,776
DCD	0,617	DCC	0,775	DCD	0,619	DCC	0,776
CD	0,615	DC	0,770	DD	0,615	DC	0,770
C	0,614	DCD	0,768	DDA	0,615	DCD	0,770
DD	0,614	DDA	0,768	CD	0,614	D	0,769
D	0,614	D	0,768	C	0,613	DDA	0,768
DDA	0,611	DD	0,766	D	0,6125	DD	0,767

Из таблицы 6 видно, что при использовании функции ssdeep последовательности мало отличаются, особенно на наборе $\mathcal{F}_2$. Отсюда можно заключить, что эта функция малопригодна в задаче оценки эффективности и стойкости обфусцирующих преобразований.

Таблица 6. Оценка эффективности (e) и стойкости (r) типов обфусцирующих преобразований при использовании функции ssdeep

e				r			
$\mathcal{F}_1$		$\mathcal{F}_2$		$\mathcal{F}_1$		$\mathcal{F}_2$	
all	1,0	A	1,0	all	1,0	A	1,0
CAA	1,0	AA	1,0	DAA	1,0	AA	1,0
DDA	1,0	DDA	1,0	CCC	1,0	DDA	1,0
DCA	1,0	DD	1,0	DCC	0,994	DD	1,0
CCC	1,0	DCD	1,0	CCA	0,993	DCD	1,0
CCA	1,0	DCC	1,0	DC	0,991	DCC	1,0
DCC	0,994	DCA	1,0	DCA	0,988	DCA	1,0
DCD	0,992	DC	1,0	CAA	0,986	DC	1,0
CDA	0,991	DAD	1,0	DCD	0,984	DAD	1,0
CCD	0,99	DAC	1,0	CC	0,983	DAC	1,0
CAC	0,987	DAA	1,0	CDA	0,983	DAA	1,0
CAD	0,986	DA	1,0	DDA	0,98	DA	1,0
CA	0,986	D	1,0	CCD	0,98	D	1,0
DC	0,983	CDA	1,0	CA	0,979	CDA	1,0
CC	0,983	CD	1,0	ADA	0,978	CD	1,0
CD	0,981	CCD	1,0	DD	0,977	CCD	1,0
DD	0,977	CCC	1,0	A	0,976	CCC	1,0
AD	0,976	CCA	1,0	DAD	0,976	CCA	1,0
DAD	0,976	CC	1,0	AA	0,976	CC	1,0
A	0,976	CAD	1,0	AD	0,975	CAD	1,0
ADA	0,976	CAC	1,0	D	0,974	CAC	1,0
DA	0,975	CAA	1,0	CAD	0,974	CAA	1,0
AA	0,975	CA	1,0	C	0,973	CA	1,0
DAA	0,975	C	1,0	DA	0,973	C	1,0
DAC	0,974	ADA	1,0	ACD	0,973	ADA	1,0
ACA	0,974	AD	1,0	DAC	0,973	AD	1,0
AC	0,974	ACD	1,0	ACA	0,972	ACD	1,0
D	0,974	ACA	1,0	CD	0,972	ACA	1,0
C	0,973	AC	1,0	CAC	0,972	AC	1,0
ACD	0,972	all	1,0	AC	0,966	all	1,0

Эти результаты коррелируют с результатами эксперимента 1 и результатами работы [11], где в рамках другого эксперимента выявлен незначительный вес признака, вычисленного с использованием функции ssdeep. Стоит отметить, что низкая точность результатов функции ssdeep впервые была продемонстрирована в [15]. Поэтому далее из рассмотрения функция ssdeep исключена. Здесь стоит отметить, что этот вывод не

противоречит использованию функции ssdeep в реализации функции sim_{bb}: в sim_{bb} функция ssdeep используется только для сравнения базовых блоков, а сделанный вывод относится к случаю, когда ssdeep используется для сравнения файлов полностью.

Таблица 7. Оценка эффективности (e) и стойкости (r) типов обфусцирующих преобразований при использовании функции tlsh

e				r			
$\mathcal{F}_1$		$\mathcal{F}_2$		$\mathcal{F}_1$		$\mathcal{F}_2$	
all	0,998	all	1,0	all	0,983	all	1,0
CCC	0,963	CCC	1,0	DCC	0,894	CCC	0,989
DCC	0,939	DDA	0,993	DCA	0,877	DCA	0,946
CCA	0,923	DAD	0,991	DAC	0,861	DDA	0,946
DCD	0,917	DCD	0,991	CCA	0,86	DAD	0,941
DCA	0,912	DCC	0,988	DAA	0,858	DCC	0,94
CCD	0,897	CCA	0,979	DCD	0,855	DCD	0,938
DC	0,89	DCA	0,971	DC	0,853	DAA	0,935
DDA	0,889	DD	0,968	DAD	0,841	CCA	0,934
DAD	0,888	CAA	0,966	CCC	0,834	DD	0,909
CAA	0,888	DC	0,931	DD	0,825	CAA	0,906
DAC	0,872	DAA	0,93	CAA	0,818	DC	0,895
CC	0,871	CC	0,916	DDA	0,809	DAC	0,883
DD	0,864	CCD	0,913	AC	0,808	DA	0,854
DAA	0,863	CDA	0,895	ACD	0,794	CDA	0,846
CAD	0,846	CAC	0,89	D	0,786	CC	0,817
CAC	0,842	CA	0,887	DA	0,781	CCD	0,814
CA	0,826	DAC	0,873	CC	0,775	AA	0,81
CDA	0,814	CAD	0,872	CAC	0,772	ADA	0,795
AC	0,804	DA	0,852	ACA	0,769	CA	0,793
CD	0,8	C	0,805	CDA	0,768	ACA	0,787
DA	0,797	CD	0,804	AD	0,768	D	0,771
ACD	0,79	ADA	0,798	CA	0,766	CAD	0,752
AD	0,79	AD	0,783	A	0,754	CAC	0,752
C	0,781	AA	0,773	AA	0,749	A	0,748
D	0,776	ACA	0,769	CCD	0,745	AC	0,719
A	0,758	D	0,75	ADA	0,737	CD	0,716
AA	0,741	A	0,724	CAD	0,736	C	0,7
ADA	0,735	ACD	0,722	C	0,725	ACD	0,69
ACA	0,732	AC	0,704	CD	0,72	AD	0,679

В таблице 9 (а, б, в) результаты из таблиц 5, 7 и 8 сгруппированы по последовательностям длины 3, 2 и 1 соответственно. Из таблиц видно, что для simgraph последовательности обфусцирующих преобразований, содержащие базовые преобразования типа A, обладают более высокой эффективностью и стойкостью (эти последовательности сосредоточены ближе к верху таблицы).

Таблица 8. Оценка эффективности (e) и стойкости (r) типов обфусцирующих преобразований при использовании функции simpic

e				r			
$\mathcal{F}_1$		$\mathcal{F}_2$		$\mathcal{F}_1$		$\mathcal{F}_2$	
all	0,406	all	0,791	all	0,365	all	0,784
DCC	0,247	CCA	0,756	DAC	0,257	CCA	0,734
CDA	0,241	AA	0,745	CCD	0,231	DAC	0,729
CCA	0,239	CCC	0,739	CAA	0,220	DCC	0,726
CAD	0,235	ADA	0,727	CAC	0,215	CCC	0,716
CCD	0,225	CAA	0,726	CDA	0,215	CCD	0,714
DCD	0,225	CDA	0,718	D	0,212	AC	0,710
DAA	0,223	DAD	0,713	A	0,211	CC	0,707
ACD	0,222	ACA	0,690	CCA	0,210	ACD	0,707
DA	0,221	CCD	0,684	ADA	0,206	AD	0,707
A	0,220	CA	0,683	DD	0,204	DCD	0,706
DCA	0,217	DD	0,672	DAA	0,202	DCA	0,705
DDA	0,216	CD	0,668	AA	0,200	C	0,703
CAC	0,216	CC	0,664	CAD	0,189	ADA	0,703
ACA	0,210	A	0,661	ACA	0,188	CAA	0,702
AA	0,208	C	0,661	ACD	0,187	ACA	0,701
AC	0,203	CAC	0,661	DCA	0,186	A	0,700
CA	0,202	ACD	0,649	C	0,185	CDA	0,699
ADA	0,202	AC	0,644	AD	0,182	CAD	0,696
DD	0,199	DCD	0,634	DDA	0,179	CD	0,693
CAA	0,196	CAD	0,632	CA	0,177	CA	0,692
CCC	0,195	AD	0,619	CD	0,177	D	0,692
CC	0,191	D	0,590	DCC	0,175	DA	0,692
DAC	0,185	DDA	0,571	DCD	0,174	AA	0,690
DC	0,185	DA	0,541	CCC	0,171	DAD	0,687
C	0,181	DAC	0,523	CC	0,169	DD	0,685
AD	0,176	DCC	0,516	DA	0,166	DC	0,680
DAD	0,175	DCA	0,489	DC	0,162	DAA	0,679
CD	0,161	DAA	0,471	AC	0,162	DDA	0,678
D	0,140	DC	0,454	DAD	0,127	CAC	0,666

Такой результат объясняется тем, что преобразования типа A, согласно таблице 4, меняют как граф вызова функций/процедур (добавляются фиктивные функции), так и граф потока управления (разбиваются базовые блоки). Для последовательностей преобразований длины один на разных наборах функция simgraph одинаково ранжировала типы как в рамках оценки эффективности, так и в рамках оценки стойкости. При этом как наиболее эффективными, так и наиболее стойкими являются одиночные преобразования типа A. Примечательно, что преобразование типа DDA на обоих наборах демонстрирует наименьшую эффективность и стойкость (таблица 9(а)).

Таблица 9. Оценка эффективности (e) и стойкости (r) типов обфусцирующих преобразований длины: а) 3; б) 2; в) 1

а)

simgraph				tlsh				simpic			
e		r		e		r		e		r	
$\mathcal{F}_1$	$\mathcal{F}_2$	$\mathcal{F}_1$	$\mathcal{F}_2$	$\mathcal{F}_1$	$\mathcal{F}_2$	$\mathcal{F}_1$	$\mathcal{F}_2$	$\mathcal{F}_1$	$\mathcal{F}_2$	$\mathcal{F}_1$	$\mathcal{F}_2$
DAC	ACA	CAA	CAA	CCC	CCC	DCC	CCC	DCC	CCA	DAC	CCA
DAA	DAA	ADA	DAC	DCC	DDA	DCA	DCA	CDA	CCC	CCD	DAC
CAD	ACD	CCC	DAD	CCA	DAD	DAC	DDA	CCA	ADA	CAA	DCC
ACA	ADA	DAA	ADA	DCD	DCD	CCA	DAD	CAD	CAA	CAC	CCC
CAC	DAD	CCA	DAA	DCA	DCC	DAA	DCC	CCD	CDA	CDA	CCD
ADA	CAA	ACA	CAC	CCD	CCA	DCD	DCD	DCD	DAD	CCA	ACD
ACD	CAD	ACD	ACD	DDA	DCA	DAD	DAA	DAA	ACA	ADA	DCD
DAD	DAC	CAC	CAD	DAD	CAA	CCC	CCA	ACD	CCD	DAA	DCA
CAA	CAC	CAD	ACA	CAA	DAA	CAA	CAA	DCA	CAC	CAD	ADA
DCA	DCA	DAD	DCA	DAC	CCD	DDA	DAC	DDA	ACD	ACA	CAA
CCA	CCA	DAC	CCA	DAA	CDA	ACD	CDA	CAC	DCD	ACD	ACA
CCC	CDA	DCA	CCC	CAD	CAC	CAC	CCD	ACA	CAD	DCA	CDA
CDA	CCC	CDA	CCD	CAC	DAC	ACA	ADA	ADA	DDA	DDA	CAD
CCD	CCD	DCC	CDA	CDA	CAD	CDA	ACA	CAA	DAC	DCC	DAD
DCC	DCC	CCD	DCC	ACD	ADA	CCD	CAD	CCC	DCC	DCD	DAA
DCD	DCD	DCD	DCD	ADA	ACA	ADA	CAC	DAC	DCA	CCC	DDA
DDA	DDA	DDA	DDA	ACA	ACD	CAD	ACD	DAD	DAA	DAD	CAC

б)

simgraph				tlsh				simpic			
e		r		e		r		e		r	
$\mathcal{F}_1$	$\mathcal{F}_2$	$\mathcal{F}_1$	$\mathcal{F}_2$	$\mathcal{F}_1$	$\mathcal{F}_2$	$\mathcal{F}_1$	$\mathcal{F}_2$	$\mathcal{F}_1$	$\mathcal{F}_2$	$\mathcal{F}_1$	$\mathcal{F}_2$
AC	AC	AC	AC	DC	DD	DC	DD	DA	AA	DD	AC
AA	AA	AA	AA	CC	DC	DD	DC	AA	CA	AA	CC
AD	AD	AD	AD	DD	CC	AC	DA	AC	DD	AD	AD
CA	CA	CA	CA	CA	CA	DA	CC	CA	CD	CA	CD
DA	DA	CC	DA	AC	DA	CC	AA	DD	CC	CD	CA
CC	CC	DA	CC	CD	CD	AD	CA	CC	AC	CC	DA
DC	CD	DC	CD	DA	AD	CA	AC	DC	AD	DA	AA
CD	DC	DD	DC	AD	AA	AA	CD	AD	DA	DC	DD
DD	DD	CD	DD	AA	AC	CD	AD	CD	DC	AC	DC

в)

simgraph				tlsh				simpic			
e		r		e		r		e		r	
$\mathcal{F}_1$	$\mathcal{F}_2$	$\mathcal{F}_1$	$\mathcal{F}_2$	$\mathcal{F}_1$	$\mathcal{F}_2$	$\mathcal{F}_1$	$\mathcal{F}_2$	$\mathcal{F}_1$	$\mathcal{F}_2$	$\mathcal{F}_1$	$\mathcal{F}_2$
A	A	A	A	C	C	D	D	A	A	D	C
C	C	C	C	D	D	A	A	C	C	A	A
D	D	D	D	A	A	C	C	D	D	C	D

Для функций tlsh большей эффективностью и стойкостью обладают последовательности, содержащие преобразования типа D, C и их комбинации. При этом преобразования типа A располагаются

ближе к низу таблицы. Для одиночных преобразований наиболее эффективными являются преобразования типа C, а наиболее стойкими – преобразования типа D. Наблюдаются одинаковые результаты для разных наборов программ, однако для эффективности и стойкости результаты оценки не совпадают. Для функции simpic сложнее выделить базовые преобразования, сочетания которых приводят к наиболее эффективным и стойким преобразованиям. И для одиночных преобразований эта функция показывает разные результаты относительно стойкости для разных наборов программ, при этом для эффективности результаты одинаковые и повторяют результат для функции simgraph.

Таким образом, функции simgraph и tlsh ведут себя стабильно на разных наборах программ. Однако для tlsh преобразования типа C считаются наименее стойкими, что не соответствует эмпирическим результатам и оценкам, полученным с помощью других функций [10].

В таблице 10 указаны типы последовательностей, которые для обоих наборов $\mathcal{F}_1$ и $\mathcal{F}_2$ попали в группу из лучших (худших) типов последовательностей, имеющих наибольшее (соответственно наименьшее) значение эффективности, стойкости или и того, и другого одновременно. Из этой таблицы видно, что для всех трех функций: 1) преобразование типа DAA входит в 15 наиболее эффективных преобразований, 2) преобразования типа CAA и DAC входят в 15 наиболее стойких преобразований, 3) преобразование типа D входит в 15 наименее эффективных, 4) преобразование типа CD входит в 15 наименее стойких. Стоит отметить, что эти результаты частично коррелируют с результатами работы [10], где эффективность и стойкость оценивалась с помощью функции похожести, построенной в [11] методами машинного обучения на базе набора специализированных и универсальных характеристик (среди которых имеются характеристики на основе нечетких хеш-функций). Построенная в [11] функция позволила в [10] выделить наиболее стойкие и наиболее слабые типы преобразований, которые таковыми являются, исходя из результатов других независимых работ. Результаты настоящей работы коррелируют с результатами работы [10] в том, что одиночные преобразования типа D считаются наименее эффективными, а преобразования типа CAA относятся к числу наиболее стойких.

Таблица 10. Типы обфусцирующих последовательностей, входящие в N лучших ($\mathrm{T}\langle N\rangle$) и худших ($\mathrm{B}\langle N\rangle$) типов последовательностей, как для $\mathcal{F}_1$, так и для $\mathcal{F}_2$

		e	r	e, r
simgraph	T3	DAA	ADA, CAA	—
	B3	D	—	—
	T7	CAD, DAA, ACA, AC, ADA	ADA, CAA, AC, DAA	DAA,AC,ADA
	B7	DCD, DDA, DD, D	DCD, DDA, DD, D	DCD, DDA, DD, D
	T15	all, CAD, DAA, ACD, A, DCA, CAC, ACA, AA, AC, CA, ADA, CAA, DAC, AD, DAD	CAC, ACA, AA, AC, ADA, DAA, ACD, DCA, CAD, CAA, DAC, AD, DAD	CAD, DAA, ACD, DCA, CAC, ACA, AA, AC, ADA, CAA, DAC, AD, DAD
	B15	CD, D, DC, C, DD, DDA, DCC, CDA, CCA, CCD, DCD, DA, CCC, CC	DCC, CDA, CCD, DCD, DDA, DD, CC, DC, C, CD, D	DCC, CDA, CCD, DCD, DDA, DD, CC, DC, C, CD, D
tlsh	T3	all, CCC	all, DCA	all
	B3	—	—	—
	T7	all, DCC, CCA, DCD, DCA, CCC	all, DCA, DCC, DCD, DAA	all, DCC, DCD, DCA, DCC
	B7	A, ACA, AA, D	C, CD	—
	T15	all, DCC, CCA, CCD, DCD, DAA, DCA, DDA, DD, CAA, CCC, CC, DC, DAD	all, DCC, CCA, DCD, DAA, DCA, DDA, DD, CAA, CCC, DAC, DC, DAD	all, DCC, CCA, DCD, DAA, DCA, DDA, DD, CAA, CCC, DC, DAD
	B15	ACD, DA, A, ACA, AA, AC, CA, ADA, C, AD, CD, D	CAD, CCD, A, CAC, ACA, AA, CA, ADA, C, AD, CD	A, ACA, AA, CA, ADA, C, AD, CD
simpic	T3	all, CCA	all, DAC	all
	B3	—	—	—
	T7	all, CDA, CCA	all, CCD, DAC	all
	B7	DC	DC, DAD	DC
	T15	all, CDA, CCA, CCD, A, DAA, ACA, AA	all, CCA, CCD, ACD, ACA, ADA, CAA, DAC	all, CCA, CCD, ACA
	B15	AC, DAC, DC, AD, D	DA, DDA, CA, DC, DAD, CD	DC

6. Заключение. Разработка новых алгоритмов сравнения бинарных исполняемых файлов является актуальной задачей, так как новые функции похожести находят применение в разных областях программной инженерии. Полученные в настоящей работе результаты можно разделить на две части.

К первой части результатов, с одной стороны, относится новый способ сравнения бинарных исполняемых файлов, описываемый формулами (5)–(7), а с другой стороны, потенциальные возможности построения новых функций похожести на базе этого способа. В предлагаемом в настоящей работе способе используется общий для

многих специализированных алгоритмов подход [16–20], включающий сравнение графов процедур и сравнение графов программ. Новым в предлагаемом способе является использование нового алгоритма сравнения графов sim_G как для сравнения графов процедур в функции sim_f, так и для сравнения графов программ в функции sim_P. Алгоритм sim_G может быть основой и для построения других алгоритмов сравнения. Например, такие алгоритмы можно получить, заменив в функции sim_P ((7)) функцию сравнения процедур sim_f, например, алгоритмами сравнения процедур из [16–20]. И наоборот, в известных алгоритмах сравнения графов программ алгоритм сравнения процедур может быть заменен на sim_f ((6)). Кроме того, в самой функции sim_f возможно замена sim_{bb} другой функцией сравнения базовых блоков. Перспективным представляется использование функции сравнения базовых блоков вида

$$\widetilde{\mathsf{sim}}_{bb}(b_1, b_2, \mathsf{T}) = \begin{cases} 1, \text{если } h_1(b_1) = h_1(b_2), \\ \mathsf{score}(h_2(\mathsf{T}(b_1)), h_2(\mathsf{T}(b_2))), \text{иначе}, \end{cases}$$

где алгоритм T выполняет преобразование инструкций базового блока перед вычислением хеш-значения. Например, алгоритм T может заменять ассемблерные команды типом команды (арифметическая операция, пересылка данных и т.п.) и/или заменять конкретные аргументы команд их кодами (регистр, ячейка памяти, константа и т.п.). Другие функции сравнения могут быть также получены даже в рамках способа сравнения, описанного формулами (5)–(7), например, путем использования другого фреймворка дизассемблирования или использования других нечетких хеш-функций при сравнении базовых блоков процедур.

Среди обозначенного потенциального разнообразия вариантов сравнения бинарных исполняемых файлов в работе реализован способ, описанный формулами (5)–(7), с использованием фреймворка дизассемблирования Radare2, криптографической функции md5 и нечеткой хеш-функции ssdeeep. Реализация этого способа и проведение экспериментов составляют вторую часть результатов настоящей работы. Проведенные эксперименты показали, что, благодаря повышенной чувствительности к оптимизационным изменениям, функция simgraph превосходит ssdeep, tlsh и simpic в точности сравнения структурного содержания программ, что определяет её преимущество для соответствующих задач. Более высокая чувствительность проявляется в том, что для функции simgraph отношение похожести в случае

(O0,O1) к похожести в случае (O0,O2) или к похожести в случае (O0,O3) всегда выше, чем соответствующие отношения у других функций (см. таблицу 3). Результаты использования функции simgraph в модели оценки эффективности и стойкости обфусцирующих преобразований показали, что построенная функция вряд ли может самостоятельно использоваться для такой оценки, так как, в силу устройства алгоритма sim_G, наиболее эффективными и стойкими считаются обфусцирующие последовательности, включающие преобразования типа A. Это расходится с эмпирическими результатами независимых исследований, в которых наиболее эффективными и стойкими считаются обфусцирующие последовательности с преобразованиями типа C. Тем не менее, функция simgraph может применяться для оценки степени запутанности графа выполнения программ, так как сравнение выполняется на основе графа. Ее достоинством представляется возможность интерпретации значений этой функции: меньшие значения соответствуют меньшей похожести на уровне графов программ и базовых блоков, и наоборот. Представляется, что simgraph может использоваться для построения более точных функций похожести, например, на основе методов машинного обучения, как это сделано в [11].

Стоит отметить высокую сложность разработанного алгоритма сравнения графов, так как на этапе сопоставления вершин графов все вершины одного графа сравниваются со всеми вершинами другого графа. Одним из дальнейших направлений исследования является поиск путей уменьшения сложности сравнения.

Авторы благодарят рецензентов за замечания и соответствующие предложения, позволившие лучше представить полученные результаты.

Литература

1. Van Tilborg H., Jajodia S. Encyclopedia of cryptography and security. Springer Science & Business Media. 2014. 1416 p.
2. Li B., He J., Huang J., Shi Y.Q. A survey on image steganography and steganalysis // Journal of Information Hiding and Multimedia Signal Processing. 2011. vol. 2. no. 2. pp. 142–172.
3. Chabot C. Recognition of a code in a noisy environment // IEEE International Symposium on Information Theory. IEEE, 2007. pp. 2211–2215. DOI: 10.1109/ISIT.2007.4557548.
4. Crawford M., Khoshgoftaar T.M., Prusa J.D, Richter A.N., Al Najada H. Survey of review spam detection using machine learning techniques // Journal of Big Data. 2015. vol. 2. DOI: 10.1186/s40537-015-0029-9.
5. Forrest S., Hofmeyr S., Somayaji A. The evolution of system-call monitoring // Proceedings of the 2008 Annual Computer Security Applications Conference (ACSAC). 2008. pp. 418–430. DOI: 10.1109/ACSAC.2008.5.
6. Khraisat A., Gondal I., Vamplew P., Kamruzzaman J. Survey of intrusion detection systems: techniques, datasets and challenges // Cybersecurity. 2019. vol. 2. DOI: 10.1186/s42400-019-0038-7.

7. Kosolapov Y.V. On detecting code reuse attacks // Automatic Control and Computer Sciences. 2020. vol. 54. no. 7. pp. 573–583. DOI: 10.3103/S0146411620070111.
8. Kiger J., Ho S.-S., Heydari V. Malware binary image classification using convolutional neural networks // Proceedings of the 17th International Conference on Cyber Warfare and Security (ICCWS). 2022. vol. 17. pp. 469–478. DOI: 10.34190/iccws.17.1.59.
9. Polsani H., Jiang H., Liu Y. DeepGray: Malware Classification Using Grayscale Images with Deep Learning // The 37th International FLAIRS Conference. 2024. pp. 1–5.
10. Борисов П.Д., Косолапов Ю.В. Способ количественного сравнения обфусцирующих преобразований // Информатика и автоматизация. 2024. Т. 23. № 3. С. 684–726. DOI: 10.15622/ia.23.3.3.
11. Борисов П.Д., Косолапов Ю.В. Способ оценки похожести программ методами машинного обучения // Труды Института системного программирования РАН. 2022. Т. 34. № 5. С. 63–76. DOI: 10.15514/ISPRAS-2022-34(5)-4.
12. Kornblum J. Identifying almost identical files using context triggered piecewise hashing // Digital investigation. 2006. vol. 3. pp. 91–97. DOI: 10.1016/j.diin.2006.06.015.
13. Breitinger F., Baier H. Similarity preserving hashing: Eligible properties and a new algorithm mrsh-v2 // Digital Forensics and Cyber Crime: 4th International Conference (ICDF2C 2012). 2013. pp. 167–182. DOI: 10.1007/978-3-642-39891-9_11.
14. Roussev V. An evaluation of forensic similarity hashes // Digital investigation. 2011. vol. 8. pp. S34–S41. DOI: 10.1016/j.diin.2011.05.005.
15. Pagani F., Dell'Amico M., Balzarotti D. Beyond precision and recall: understanding uses (and misuses) of similarity hashes in binary analysis // Proc. of the Eighth ACM Conference on Data and Application Security and Privacy. 2018. pp. 354–365. DOI: 10.1145/3176258.3176306.
16. BinDiff. URL: https://www.zynamics.com/ (дата обращения: 23.06.2025).
17. Aslanyan H., Avetisyan A., Arutunian M., Keropyan G., Kurmangaleev S., Vardanyan V. Scalable Framework for Accurate Binary Code Comparison // Ivannikov ISPRAS Open Conference (ISPRAS). 2017. pp. 34–38. DOI: 10.1109/ISPRAS.2017.00013.
18. Machoc hash. URL: https://github.com/ANSSI-FR/polichombr/blob/dev/ (дата обращения: 23.06.2025).
19. Machoke. URL: https://github.com/conix-security/machoke (дата обращения: 23.06.2025).
20. Li Y., Jang J., Ou X. Topology-aware hashing for effective control flow graph similarity analysis // Security and Privacy in Communication Networks: 15th EAI International Conference (SecureComm). 2019. pp. 278–298.
21. Borisov P.D., Kosolapov Y.V. On the Automatic Analysis of the Practical Resistance of Obfuscating Transformations // Aut. Control Comp. Sci. 2020. vol. 54. pp. 619–629. DOI: 10.3103/S0146411620070044.
22. Борисов П.Д., Косолапов Ю.В. О функции похожести графических представлений исполняемых файлов в модели оценки обфусцирующих преобразований // Известия ЮФУ. 2025. № 3(245). С. 264–273.
23. Naville Z. Hikari–an improvement over Obfuscator-LLVM. 2017. URL: https://github.com/HikariObfuscator/Hikari (дата обращения: 26.11.2024).
24. Holder W., McDonald J.T., Andel T.R. Evaluating optimal phase ordering in obfuscation executives // Proceedings of the 7th Software Security, Protection, and Reverse Engineering/Software Security and Protection Workshop. 2017. pp. 1–12. DOI: 10.1145/3151137.3151140.
25. small-programs. A set of small programs for experiments with obfuscations. URL: https://github.com/Boriskin61/small-programs (дата обращения: 22.06.2025).

Борисов Петр Дмитриевич — заведующий лабораторией, ФГАНУ НИИ «Спецвузавтоматика». Область научных интересов: исследование и анализ программного кода, методы обфускации и деобфускации, способы оценки качества обфусцирующих преобразований. Число научных публикаций — 10. borisovpetr@mail.ru; улица Города Волос, 6, 344011, Ростов-на-Дону, Россия; р.т.: +7(863)297-5111.

Варламов Данил Викторович — студент, кафедра алгебры и дискретной математики института математики, механики и компьютерных наук им. и.и. воровича, Южный федеральный университет (ЮФУ). Область научных интересов: исследование и анализ программного кода. varlamov@sfedu.ru; улица Мильчакова, 8а, 344090, Ростов-на-Дону, Россия; р.т.: +7(863)297-5111.

Косолапов Юрий Владимирович — канд. техн. наук, доцент кафедры, кафедра алгебры и дискретной математики института математики, механики и компьютерных наук им. и.и. воровича, Южный федеральный университет (ЮФУ). Область научных интересов: помехоустойчивые коды в криптографии и стеганографии, теоретическая и практическая обфускация кода. Число научных публикаций — 110. yvkosolapov@sfedu.ru; улица Мильчакова, 8а, 344090, Ростов-на-Дону, Россия; р.т.: +7(863)297-5111.

DOI 10.15622/ia.24.5.9

P. BORISOV, D. VARLAMOV, Yu. KOSOLAPOV

A GRAPH SIMILARITY CALCULATION ALGORITHM AND ITS APPLICATION FOR COMPARING BINARY EXECUTABLE FILES

Borisov P., Varlamov D., Kosolapov Yu. **A Graph Similarity Calculation Algorithm and Its Application for Comparing Binary Executable Files.**

Abstract. The paper addresses the task of static (without execution) comparison of binary executable files. A program and any of its procedures can be represented as a directed graph. For a program, the corresponding graph is a function (procedure) call graph, where the nodes are the functions themselves, and an edge from vertex a to b describes a call to function b from function a. For a procedure, such a graph is a control flow graph, where the vertices are basic blocks, and an edge between nodes a and b means that the commands of block b can be executed after the commands of block a. The study proposes an algorithm for comparing directed graphs, which is then applied to program comparison. The graph comparison algorithm is based on applying a node similarity function. For comparing procedure graphs, a fuzzy hash function and a cryptographic hash function are used as this similarity function. Subsequently, this method of comparing procedure graphs is used as the node similarity function for comparing program graphs. Based on the proposed algorithm, a method for program comparison has been developed, and its investigation was conducted through two experiments. In the first experiment, the method's behavior was studied when comparing programs compiled with different optimization options (O0, O1, O2, O3, and Os). In the second experiment, the possibility of identifying effective and resilient obfuscating transformations within a previously developed model was investigated. Within the first experiment, evidence was obtained supporting the hypothesis that similarity decreases as optimization increases from O1 to O3. Within the second experiment, some previously obtained results concerning the effectiveness (ineffectiveness) and resilience (non-resilience) of obfuscating transformations were confirmed.

Keywords: graph comparison, program similarity, effectiveness and resilience of obfuscation.

References

1. Van Tilborg H., Jajodia S. Encyclopedia of cryptography and security. Springer Science & Business Media. 2014. 1416 p.
2. Li B., He J., Huang J., Shi Y.Q. A survey on image steganography and steganalysis. Journal of Information Hiding and Multimedia Signal Processing. 2011. vol. 2. no. 2. pp. 142–172.
3. Chabot C. Recognition of a code in a noisy environment. IEEE International Symposium on Information Theory. IEEE, 2007. pp. 2211–2215. DOI: 10.1109/ISIT.2007.4557548.
4. Crawford M., Khoshgoftaar T.M., Prusa J.D, Richter A.N., Al Najada H. Survey of review spam detection using machine learning techniques. Journal of Big Data. 2015. vol. 2. DOI: 10.1186/s40537-015-0029-9.
5. Forrest S., Hofmeyr S., Somayaji A. The evolution of system-call monitoring. Proceedings of the 2008 Annual Computer Security Applications Conference (ACSAC). 2008. pp. 418–430. DOI: 10.1109/ACSAC.2008.5.
6. Khraisat A., Gondal I., Vamplew P., Kamruzzaman J. Survey of intrusion detection systems: techniques, datasets and challenges. Cybersecurity. 2019. vol. 2. DOI: 10.1186/s42400-019-0038-7.

7. Kosolapov Y.V. On detecting code reuse attacks. Automatic Control and Computer Sciences. 2020. vol. 54. no. 7. pp. 573–583. DOI: 10.3103/S0146411620070111.
8. Kiger J., Ho S.-S., Heydari V. Malware binary image classification using convolutional neural networks. Proceedings of the 17th International Conference on Cyber Warfare and Security (ICCWS). 2022. vol. 17. pp. 469–478. DOI: 10.34190/iccws.17.1.59.
9. Polsani H., Jiang H., Liu Y. DeepGray: Malware Classification Using Grayscale Images with Deep Learning. The 37th International FLAIRS Conference. 2024. pp. 1–5.
10. Borisov P.D., Kosolapov Yu.V. [A method to quantitative compare obfuscating ttransformations]. Informatics and Automation. 2024. vol. 23. no. 3. pp. 684–726. DOI: 10.15622/ia.23.3.3. (In Russ.).
11. Borisov P.D., Kosolapov Yu.V. [Method to Evaluate Program Similarity Using Machine Learning Methods]. Trudy Instituta sistemnogo programmirovanija RAN – Proceedings of the Institute for System Programming of the RAS (Proceedings of ISP RAS). 2022. vol. 34. no. 5. pp. 63–76. DOI: 10.15514/ISPRAS-2022-34(5)-4. (In Russ.).
12. Kornblum J. Identifying almost identical files using context triggered piecewise hashing. Digital investigation. 2006. vol. 3. pp. 91–97. DOI: 10.1016/j.diin.2006.06.015.
13. Breitinger F., Baier H. Similarity preserving hashing: Eligible properties and a new algorithm mrsh-v2. Digital Forensics and Cyber Crime: 4th International Conference (ICDF2C 2012). 2013. pp. 167–182. DOI: 10.1007/978-3-642-39891-9_11.
14. Roussev V. An evaluation of forensic similarity hashes. Digital investigation. 2011. vol. 8. pp. S34–S41. DOI: 10.1016/j.diin.2011.05.005.
15. Pagani F., Dell'Amico M., Balzarotti D. Beyond precision and recall: understanding uses (and misuses) of similarity hashes in binary analysis. Proc. of the Eighth ACM Conference on Data and Application Security and Privacy. 2018. pp. 354–365. DOI: 10.1145/3176258.3176306.
16. BinDiff. Available at: https://www.zynamics.com/ (accessed 23.06.2025).
17. Aslanyan H., Avetisyan A., Arutunian M., Keropyan G., Kurmangaleev S., Vardanyan V. Scalable Framework for Accurate Binary Code Comparison. Ivannikov ISPRAS Open Conference (ISPRAS). 2017. pp. 34–38. DOI: 10.1109/ISPRAS.2017.00013.
18. Machoc hash. Available at: https://github.com/ANSSI-FR/polichombr/blob/dev/ (accessed 23.06.2025).
19. Machoke. Available at: https://github.com/conix-security/machoke (accessed 23.06.2025).
20. Li Y., Jang J., Ou X. Topology-aware hashing for effective control flow graph similarity analysis. Security and Privacy in Communication Networks: 15th EAI International Conference (SecureComm). 2019. pp. 278–298.
21. Borisov P.D., Kosolapov Y.V. On the Automatic Analysis of the Practical Resistance of Obfuscating Transformations. Aut. Control Comp. Sci. 2020. vol. 54. pp. 619–629. DOI: 10.3103/S0146411620070044.
22. Borisov P.D., Kosolapov Yu.V. [On the Similarity Function of Graphical Representations of Executable Files in the Obfuscation Transformation Evaluation Model]. Izvestija JuFU – Bulletin of the Southern Federal University. 2025. no. 3(245). pp. 264–273. (In Russ.).
23. Naville Z. Hikari–an improvement over Obfuscator-LLVM. 2017. Available at: https://github.com/HikariObfuscator/Hikari (accessed 26.11.2024).
24. Holder W., McDonald J.T., Andel T.R. Evaluating optimal phase ordering in obfuscation executives. Proceedings of the 7th Software Security, Protection, and Reverse Engineering/Software Security and Protection Workshop. 2017. pp. 1–12. DOI: 10.1145/3151137.3151140.
25. small-programs. A set of small programs for experiments with obfuscations. Available at: https://github.com/Boriskin61/small-programs (accessed 22.06.2025).

Borisov Petr — Head of the laboratory, FSASE SRI «Specvuzavtomatika». Research interests: program code analysis, obfuscation and deobfuscation, methods for assessing the quality of obfuscating transformations. The number of publications — 10. borisovpetr@mail.ru; 6, Goroda Volos St., 344011, Rostov-on-Don, Russia; office phone: +7(863)297-5111.

Varlamov Danil — Student, Department of algebra and discrete mathematics, i.i. vorovich institute of mathematics, mechanics, and computer science, Southern Federal University (SFedU). Research interests: software code analysis and investigation. varlamov@sfedu.ru; 8a, Milchakov St., 344090, Rostov-on-Don, Russia; office phone: +7(863)297-5111.

Kosolapov Yury — Ph.D., Associate professor of the department, Department of algebra and discrete mathematics, i.i. vorovich institute of mathematics, mechanics, and computer science, Southern Federal University (SFedU). Research interests: error-correcting codes in cryptography and steganography, theoretical and practical code obfuscation. The number of publications — 110. yvkosolapov@sfedu.ru; 8a, Milchakov St., 344090, Rostov-on-Don, Russia; office phone: +7(863)297-5111.