DOI: 10.15514/ISPRAS-2022-34(5)-4

Способ оценки похожести программ методами машинного обучения

[1] *П.Д. Борисов, ORCID: 0000-0002-8919-8310 <borisovpetr@mail.ru>*
[2] *Ю.В. Косолапов, ORCID: 0000-0002-1491-524X <itaim@mail.ru>*

[1] *ФГАНУ НИИ «Спецвузавтоматика»*
344003, Россия, г. Ростов-на-Дону, ул. Города Волос, 6
[2] *Южный федеральный университет,*
344006, Россия, г. Ростов-на-Дону, ул. Б. Садовая, д. 105/42

Аннотация. В работе рассматривается задача построения алгоритма сравнения двух исполнимых файлов. В основе алгоритма лежит построение по заданной паре программ вектора показателей похожести и принятие на основе этого вектора решения о похожести или непохожести программ с помощью методов машинного обучения. Показатели похожести строятся с помощью алгоритмов двух типов: с помощью алгоритмов, не учитывающих формат входных данных (значения нечетких хеш-функций, значения коэффициентов сжатия), и с помощью алгоритмов, выполняющих анализ машинного кода (с помощью дизассемблеров). Всего построено 15 показателей: 9 показателей первого типа и 6 второго. На основе построенного обучающего множества пар похожих и непохожих программ (на основе набора программ coreutils) обучены и протестированы 7 разных бинарных классификаторов. Результаты экспериментов показали высокую точность моделей на основе случайного леса и k ближайших соседей. Также выявлено, что совместное применение показателей обоих типов может увеличить точность классификации.

Ключевые слова: обфускация; похожесть программ; машинное обучение

Для цитирования: Борисов П.Д., Косолапов Ю.В. Способ оценки похожести программного кода методами машинного обучения. Труды ИСП РАН, том 34, вып. 5, 2022 г., стр. 63-76. DOI: 10.15514/ISPRAS-2022-34(5)-4

A Method to Evaluate Program Similarity Using Machine Learning Methods

[1] *P.D. Borisov, ORCID: 0000-0002-8919-8310 <borisovpetr@mail.ru>*
[2] *Yu.V. Kosolapov, ORCID: 0000-0002-1491-524X <itaim@mail.ru>*

[1] *FSASE SRI «Specvuzavtomatika»*
6, st. Goroda Volos, Rostov-on-Don, 344003, Russia
[2] *Southern Federal University,*
105/42, Bol'shaya Sadovaya st., Rostov-on-Don, 344006, Russia

Abstract. The problem of constructing an algorithm for comparing two executable files is considered. The algorithm is based on the construction of similarity features vector for a given pair of programs. This vector is then used to decide on the similarity or dissimilarity of programs using machine learning methods. Similarity features are built using algorithms of two types: universal and specialized. Universal algorithms do not take into account the format of the input data (values of fuzzy hash functions, values of compression ratios). Specialized algorithms work with executable files and analyze machine code (using disassemblers). A total of 15 features were built: 9 features of the first type and 6 of the second. Based on the constructed training set of similar and dissimilar program pairs, 7 different binary classifiers were trained and tested. To build the training

set, coreutils programs were used. The results of the experiments showed high accuracy of models based on random forest and k nearest neighbors. It was also found that the combined use of features of both types can improve the accuracy of classification.

Keywords: obfuscation; program similarity; machine learning

For citation: Borisov P.D., Kosolapov Yu.V. A method for evaluating the similarity of program code using machine learning methods. Trudy ISP RAN/Proc. ISP RAS, vol. 34, issue 5, 2022. pp. 63-76 (in Russian). DOI: 10.15514/ISPRAS-2022-34(5)-4

1. Введение и постановка задачи

Преобразования программного кода, сохраняющие его функциональность, можно разделить на оптимизирующие, диверсифицирующие и обфусцирующие [1]. Первые применяются для ускорения работы программы и/или уменьшения размера программы, а вторые – для защиты программ от удаленных атак, так как эксплойт, предназначенный для оригинальной версии программы, может не подходить для диверсифицированного варианта [2]. Целью же обфусцирующих преобразований обычно является сокрытие (маскировка) алгоритмов, используемых в программе, и/или сокрытие критически важных данных, например, криптографических ключей [3]. Основным отличием обфусцирующих преобразований от оптимизирующих и диверсифицирующих является их направленность на затруднение понимания программы аналитиком как при статическом, так и при динамическом анализе. Обычно в результате таких преобразований обфусцированная версия *Obf(P)* программы *P* становится, нестрого выражаясь, «менее похожа» на необфусцированный вариант *P*. Отметим, что термин «похожесть программ» не является четко определенным в программной инженерии, что является следствием неточности самого слова «похожесть». Тем не менее, исследователи сходятся в возможности определения похожести на синтаксическом (структурном) и/или на семантическом (поведенческом) уровне [4]. Именно непохожесть *Obf(P)* и *P* на синтаксическом или семантическом уровнях, например, позволяет маскировать известный код под неизвестный код, а также затрудняет восстановление проприетарного алгоритма или секретных данных из кода. В связи с этим, конкурирующими направлениями исследований являются разработка эффективных методов обнаружения похожести программ и разработка практических методов обфускации, преобразующих программу *P* в непохожую на нее версию *Obf(P)*.

В настоящей работе похожесть программ определяется на семантическом уровне, причем под программой понимается исполнимый файл программы, полученный по исходному коду с помощью компилятора. Далее в тексте две программы считаются похожими, если на одинаковых входных данных возвращают одинаковый результат, и при этом они реализуют один и тот же алгоритм. В рамках такого определения похожести две программы, например, реализующие разные алгоритмы сортировки и возвращающие на одинаковых входных данных одинаковый результат, считаются непохожими. И хотя такое определение существенно сужает класс пар похожих программ, оно представляется удобным в рамках следующей модели оценки стойкости обфусцирующих преобразований, предложенной в [5]. Пусть $Similarity$ – метод вычисления похожести двух программ, возвращающий действительное число из диапазона [0,1], где 0 – программы похожи, 1 – непохожи. Для программы *P* преобразование Obf_1 считается более стойким, чем Obf_2, если

$$Similarity(P, Obf_1(P)) > Similarity(P, Obf_2(P)).$$

В частном случае, когда $Similarity$ возвращает «жесткое» решение: 0 либо 1 (похожи, либо непохожи), преобразование Obf считается стойким, если $Similarity\big(P, Obf(P)\big) = 1$, и нестойким в противном случае. Здесь и далее предполагается, что метод $Similarity$ сравнения программ строиться без учета известных алгоритмов обфусцирующих преобразований, так как при таком подходе представляется возможным объективное сравнение стойкости разных методов обфускации. В этом случае метод $Similarity$, как с

«мягкими» решениями, так и с «жесткими» решениями, может применяться для итеративного подбора обфусцирующего преобразования (из заданного множества преобразований), обеспечивающего требуемый уровень стойкости. Целью подбора является выбор такого преобразования Obf для программы P, при котором метод $Similarity$ будет считать программы $Obf(P)$ и P наименее похожими – для алгоритма с «мягкими» решениями, и непохожими – для алгоритма с «жесткими» решениями. Отметим, что аналогичный подход к оценке стойкости обфусцирующих преобразований реализуется в [6], где стойкость оценивается не с помощью похожести файлов, а с помощью времени символьного исполнения программ.

В методе $Similarity$ при сравнении P и Q могут учитываться как статические характеристики, получаемые без запуска программ, так и динамические, вычисляемые в результате запуска или симуляции запуска. В [7,8] исследуется возможность применения в $Similarity$ в качестве динамических характеристик таких, которые могут быть построены с помощью разработанной в [5] схемы получения характеристик символьного исполнения. Настоящее исследование направлено на применение в $Similarity$ только статических характеристик. Стоит отметить, что совместное применение как статических, так и динамических характеристик, может позволить повысить точность $Similarity$, однако эта задача является направлением дальнейшего исследования и в этой работе не решается.

Способы статической оценки степени похожести условно можно разделить на два типа: универсальные и специализированные. К универсальным отнесем способы, не учитывающие то, что сравниваемыми объектами являются файлы/фрагменты исполнимого кода. В то время как способы, учитывающие этот аспект, отнесем к специализированным. В работе с помощью методов машинного обучения строится и исследуется алгоритм, который по двум исполнимым файлам P и Q на основании вычисленного значения из диапазона действительных чисел [0,1] возвращает «жесткое» решение: 0 – программы похожи, 1 – программы непохожи. Вектор признаков строиться на основе значений, вычисленных с помощью универсальных и специализированных способов сравнения программ P и Q. Целью исследования является оценка точности некоторых известных моделей классификации и оценка эффекта от совместного применения универсальных и специализированных подходов сравнения.

2. Существующие подходы к оценке похожести

Как отмечалось выше, способы статической оценки степени похожести исполнимых файлов можно разделить на универсальные и специализированные. К универсальным способам относятся способы на основе вычисления значений нечетких хеш-функций (fuzzy hash), которые, в отличие от криптографических хеш-функций, слабо чувствительны к малым изменениям аргумента. Обычно алгоритмы нечетких хеш-функций не учитывают формат данных: используется один алгоритм вычисления значения для машинного кода, текстовых данных, аудиоданных и видеоданных, в чем и заключается их универсальность. Такие функции, например, успешно применяются в системах фильтрации электронных писем, при криминалистическом анализе цифровых документов, при выявлении версий вредоносных файлов (в таких сервисах, как VirusTotal, VirusShare).

Однако результаты исследования [9] показали, что не все нечеткие хеш-функции одинаково эффективны при оценке похожести бинарных файлов программ. В частности, хеш-функция ssdeep реже других, рассмотренных в [9] функций, обнаруживает похожие программы. Там же отмечено, что обфускация может существенно менять бинарный код, что может снизить качество обнаружения похожих программ для любой хеш-функции (не только ssdeep). Это связано с тем, что нечеткие хеш-функции устойчивы только к малым изменениям. Вопрос совместного применения нескольких хеш-функций для повышения качества обнаружения похожих программ в [9] не исследовался.

В универсальности способов на основе нечетких хеш-функций заключается и низкое качество результатов сравнения похожих исполнимых файлов. За счет учета формата данных может повыситься качество сравнения похожих программ, но при этом универсальность хеш-функции будет утрачена. В [10, 11] фактически предлагаются специализированные подходы построения нечетких хеш-функций для исполнимых файлов. Именно, в [10, 11] решается следующая задача: для заданного объекта сравнения (функции/подпрограммы) найти наиболее близкий объект в заданном списке (*архивном множестве* объектов). Для решения этой задачи в [10, 11] на основе дизассемблированного кода строится описание объекта сравнения. Это описание, представляющее собой вектор фиксированной длины, можно рассматривать как значение нечеткой хеш-функций от объекта сравнения, представляющего собой машинный код. В [10] для построения описания используется нейросеть, а в [11] вектор описания объекта сравнения предлагается строить как набор значений мер схожести с каждым элементом *опорного множества* объектов (используются пять мер схожести). Для вычисления меры похожести двух векторов описаний в [10] применяется косинусный коэффициент, а в [11] – обобщенный коэффициент Жаккара. В обоих случаях для применения методов сравнения необходимо иметь архивное множество объектов, с помощью которого может быть построено описание объектов сравнения.

Другие специализированные подходы к оценке похожести программ предлагаются в [12] и [13], где решается задача поиска клонов исходного кода на основе семантического анализа программ. Именно, в [12] и [13] для повышения точности обнаружения клонов предлагается сравнивать графы зависимостей программ. Недостатком такого подхода, как отмечается в [12], является его относительно низкая скорость, хотя в [13] и удалось кроме повышения точности сравнения добиться достаточно высокой скорости сравнения. Заметим, что в свободном доступе не удалось найти программных реализаций подходов из [10,11], что затрудняет их сравнение как между собой, так и с другими подходами. Способы из [12] и [13] хотя и реализованы авторами, также не находятся в свободном доступе.

Кроме того, представляется, что эти подходы в меньшей степени (чем подходы из [10,11]) соответствуют задаче оценки стойкости обфусцирующего преобразования по следующим причинам. Во-первых, методы поиска клонов, в частности из [12] и [13], нацелены на выявление трех типов клонов (см. [14]), которые возникают именно в контексте клонирования кода, когда в первую очередь перед разработчиком стоит задача адаптации готового кода под собственные нужды, а задача запутывания кода при этом не является первоочередной. Во-вторых, важными характеристиками алгоритмов поиска клонов являются точность и скорость работы, в то время как при оценке стойкости обфусцирующих преобразований наибольшую роль играет точность, при этом скорость существенна в меньшей степени. В-третьих, для выявления клонов кода алгоритму поиска могут потребоваться исходные тексты программ, а при выявлении похожести исполнимых файлов сравниваются скомпилированные версии программ, что позволяет оценивать любые обфусцирующие преобразования и абстрагироваться от уровня их применения (уровня исходного кода, уровня промежуточного представления и уровня машинного кода).

3. Способ оценки похожести исполнимых файлов

2.1 Начальное описание программ

В настоящем исследовании похожесть программ оценивается на основе ряда статических характеристик с привлечением методов машинного обучения. Для каждой программы P ее *начальное описание* задается как множество статических характеристик двух типов: множество универсальных характеристик состоит из тех, которые получены с помощью средств, не учитывающих формат данных (значения нечетких хеш-функций, значения коэффициентов сжатия); во множество специализированных характеристик отнесены те,

которые получены с помощью средств анализа машинного кода (в частности, с помощью дизассемблеров). Саму программу *P* можно отнести как к универсальным характеристикам, так и к специализированным характеристикам. Это связано с тем, что многие утилиты сравнения файлов, в качестве входных данных принимают сами файлы. При этом одни утилиты рассматривают файлы как набор бинарных данных (утилиты с универсальным способом сравнения), а другие – как исполнимый код (утилиты со специализированным способом сравнения).

Использование значений нечетких хеш-функций для начального описания обосновано простотой их использования и тем, что они уже применяются для сравнения файлов с исполнимым кодом. В частности, эти функции находят применение в криминалистическом анализе вредоносного кода [15], в онлайн-сервисах обнаружения вредоносного кода, таких как VirusTotal [16]. Применение алгоритмов сжатия обосновано, с одной стороны простотой их применения, а с другой стороны, тем, что они могут использоваться как аппроксимация нормализованного расстояния сжатия [4]. В [17], например, Колмогоровская сложность программы используется как оценка качества обфусцирующих преобразований, а для аппроксимации этой сложности применяются алгоритмы сжатия. Использование коэффициентов сжатия в начальном описании программы, таким образом, основано на предположении, что похожие по функциональности необфусцированные программы, имеют близкую сложность по Колмогорову.

Для получения специализированных характеристик в настоящей работе применяются дизассемблеры, позволяющие по файлу с исполнимым кодом построить его представление в формате BinExport [18], пригодном для сравнения с помощью утилиты BinDiff [19]. Формат BinExport позволяет хранить граф потока управления и граф вызова функций, которые используются утилитой BinDiff для вычисления расстояния между исполнимыми файлами. Выбор BinDiff обоснован, с одной стороны, тем, что, как показывают результаты исследования [20], подход, используемый в BinDiff, дает высокую точность сравнения; с другой стороны, в настоящий момент формат BinExport поддерживается популярными дизассемблерами IDA Pro [21], Ghydra [22] и Binary Ninja [23]. В частности, соответствующие плагины для IDA Pro и Ghydra позволяют автоматизировать процесс выгрузки данных в формате BinExport. Утилита Binary Ninja таких средств не предлагает, однако ее прикладной программный интерфейс позволяет автоматизировать этот процесс на языке Python.

Таким образом, для каждой программы *P* в базе данных сохраняется ее начальное описание, включающее характеристики, перечисленные в табл. 1.

Табл. 1. Начальное описание программы P, 0 – универсальная характеристика, 1 – специализированная характеристика
Table 1. Initial description of program P, 0 - universal characteristic, 1 - specialized characteristic

Характеристика		**Тип значения**	**Описание**
Обозначение	**Тип**		
P	0/1	Строка	Путь к файлу программы *P*
ssdeepHash	0	Строка	Значение нечеткой хеш-функции ssdeep (см. [24])
sdhashHash		Строка	Значение нечеткой хеш-функции sdhash (см. [25])
tlshHash		Строка	Значение нечеткой хеш-функции tlsh (см. [25])
lzmaVal		Число (float)	Степень сжатия программы *P* алгоритмом lzma (см. [26])
bz2Val		Число (float)	Степень сжатия программы *P* алгоритмом bz2 (см. [27])
deflateVal		Число (float)	Степень сжатия программы *P* алгоритмом deflate (см. [28])

idaBinExp	1	Строка	Путь к каталогу, содержащему представление программы в формате BinExport, полученное с помощью IDA Pro
ghydraBinExp		Строка	Путь к каталогу, содержащему представление программы в формате BinExport, полученное с помощью Ghydra
ninjaBinExp		Строка	Путь к каталогу, содержащему представление программы в формате BinExport, полученное с помощью Binary Ninja

2.2 Описание похожести программ

Для пары программ *P* и *Q* по их начальным описаниям строится вектор *sim*(*P*,*Q*) из пятнадцати показателей похожести, каждый из которых принимает действительное значение от нуля до единицы: значение 0 соответствует похожим программам, а значение 1 – непохожим. Вектор *sim*(*P*,*Q*) включает показатели из табл. 2, где приведены условные наименования показателей и способ их вычисления. Показатели, вычисленные по универсальным (соответственно специализированным) характеристикам, в табл. 2 называются универсальными (соответственно специализированными) показателями. На основании вектора *sim*(*P*,*Q*) в дальнейшем с помощью методов машинного обучения будет приниматься решение о похожести или непохожести *P* и *Q*.

Отметим, что утилита BinDiff, сравнивающая графы потока управления программ *P* и *Q*, возвращает два значения: значение похожести программ и степень уверенности (confidence). Поэтому в число показателей в табл. 2 для каждого из трех рассмотренных дизассемблеров, поддерживающих формат BinExport, входят как значение похожести, возвращенное BinDiff (строки 10, 12, 14 в табл. 2), так и произведение значения похожести на степень уверенности (строки 11, 13, 15 в табл. 2). Пакет анализа исполнимого кода Radare2 [29] не поддерживает формат BinExport, что не позволяет применить утилиту BinDiff. Однако Radare2 содержит утилиту radiff2, позволяющую находить расстояние Майерса [30] и расстояние Левенштейна [31]. Так как при сравнении дизассемблирование не выполняется, то показатели *myersdiff* (строка 8 в табл. 2) и *levenshteindiff* (строка 9 в табл. 2) являются универсальными.

Табл. 2. Показатели похожести программ P и Q, тип 0 – универсальный показатель, тип 1 – специализированный показатель
Table 2. Program similarity features P and Q, type 0 - universal feature, type 1 - specialized feature

№	**Показатель**	**Тип**	**Описание**
1	*ssdeep*	0	Результат сравнения *ssdeepHash(P)* и *ssdeepHash(Q)* утилитой сравнения хеш-значений функции ssdeep
2	*sdhash*		Результат сравнения *sdhashHash(P)* и *sdhashHash(Q)* утилитой сравнения хеш-значений функции sdhash
3	*tlsh*		Результат сравнения *tlshHash(P)* и *tlshHash(Q)* утилитой сравнения хеш-значений функции *tlsh* с учетом длины файлов *P* и *Q*
4	*tlshxlen*		Результат сравнения *tlshlenHash(P)* и *tlshlenHash(Q)* утилитой сравнения хеш-значений функции *tlsh* без учета длины файлов *P* и *Q*
5	*lzma*		$1 - \frac{\lvert lzmaVal(P) - lzmaVal(Q)\rvert}{\max\{lzmaVal(P), lzmaVal(Q)\}}$
6	*bz2*		$1 - \frac{\lvert bz2Val(P) - bz2Val(Q)\rvert}{\max\{bz2Val(P), bz2Val(Q)\}}$
7	*deflate*		$1 - \frac{\lvert deflateVal(P) - deflateVal(Q)\rvert}{\max\{deflateVal(P), deflateVal(Q)\}}$
8	*myersdiff*		Результат сравнения бинарных файлов *P* и *Q* с помощью утилиты radiff с опцией -s (расстояние Майерса) из пакета анализа программ Radare2
9	*levenshteindiff*		Результат сравнения бинарных файлов *P* и *Q* с помощью утилиты radiff с опцией -ss (расстояние Левенштейна)

10	*idadiff*	1	Результат сравнения представлений *idaBinExp(P)* и *idaBinExp(Q)* с помощью утилиты BinDiff (см. [19])
11	*idadiffc*		$idadiff \cdot confidence_{ida}$
12	*ghydradiff*		Результат сравнения представлений *ghydraBinExp(P)* и *ghydraBinExp(Q)* с помощью утилиты BinDiff
13	*ghydradiffc*		$ghydradiff \cdot confidence_{ghydra}$
14	*ninjadiff*		Результат сравнения представлений *ninjaBinExp(P)* и *ninjaBinExp(Q)* с помощью утилиты BinDiff
15	*ninjadiffc*		$ninjadiff \cdot confidence_{ninja}$

2.3 Алгоритм сравнения программ

Алгоритм сравнения программ P и Q предлагается строить следующим образом: 1) на основе входных файлов P и Q строится вектор показателей похожести $sim(P,Q)$; 2) вектор $sim(P,Q)$ передается обученному бинарному классификатору, который принимает решение о похожести (значение 0) или непохожести программ (значение 1). Стоит отметить, что «жесткое» решение – 0 или 1 – классификатором обычно принимается на основе «мягкого» решения из диапазона [0,1] и порога: при превышении порога «мягкое» решение преобразуется в «жесткое» решение 1, а при недостижении порога возвращается «жесткое» решение 0. Для реализации алгоритма сравнения программ необходимо решить две задачи: построить обучающее множество и выбрать лучший классификатор.

В то время, как выбор наилучшего классификатора может быть осуществлен путем сравнения результатов экспериментов, построение обучающего множества является отдельной задачей, в рамках которой следует определить достаточный для обучения объем и состав обучающей выборки. Обучающее множество должно состоять из двух непересекающихся подмножеств: подмножества пар похожих программ и подмножества пар разных, непохожих программ. Каждая пара похожих программ P и Q используется для вычисления соответствующего вектора $sim(P,Q)$; аналогично вектор $sim(P,Q)$ вычисляется и для каждой пары непохожих программ P и Q. Таким образом, по множествам похожих и непохожих пар программ могут быть построены соответствующие множества векторов показателей похожести, которые далее используются для обучения классификаторов.

Отметим, что, как множество похожих, так и множество непохожих пар программ, как представляется, должно строиться на основе программ, реализующих алгоритмы из разных предметных областей: программы обработки строк, утилиты по работе с файловой системой, криптографические утилиты, алгоритмы обработки сигналов, алгоритмы линейной алгебры и т.п. Такой подход к построению обучающего множества может позволить достичь универсальности метода $Similarity$ выявления похожести программ, без привязки к предметной области алгоритмов, реализуемых сравниваемыми программами.

В настоящей работе для построения обучающего множества предлагается использовать известный набор разных (по предметным областям реализуемых алгоритмов) программ с исходным кодом. Наличие исходного кода позволяет, используя разные компиляторы и опции компиляции (например, опции оптимизации) строить пары похожих программ: любая пара программ, полученных из одного исходного кода, но с помощью разных компиляторов и/или разных опций компиляций, считается парой похожих программ (в смысле определения похожести, данного во введении). При этом любая пара программ, полученных из разных исходных кодов, независимо от того, какие компиляторы и опции компиляции использовались, образует пару непохожих программ.

4. Результаты экспериментов

В работе для составления обучающего множества был выбран набор программ coreutils [32], включающий 106 программ, написанных на языке Си. Для компиляции использовалось 9 вариантов компиляторов (GCC [33] версий 7.5.0, 8.4.0, 9.4.0, 10.3.0, Clang [34] версий 7.0.1, 8.0.1, 9.0.1, 10.0.0, AOCC [35] версии 3.0.0) для операционной системы Linux, в каждом из которых применялись 5 опций оптимизации (O0, O1, O2, O3, Os). Таким образом, для каждой программы P создается 45 ее вариантов: $P_1, \dots, P_{45}$, – а всего для набора coreutils создается $45 \cdot 106 = 4770$ программ. Для каждой программы P можно построить $C_{45}^2 = 990$ пар похожих программ: $\{(P_i, P_j): i, j = 1, \dots, 45, i \neq j\}$. Поэтому множество векторов показателей похожести, соответствующих всем парам похожих программ, имеет мощность $990 \cdot 106 = 104940$.

Множество пар непохожих программ состоит из $C_{106}^2 \cdot 45^2 = 11269125$ пар, так как каждая программа в паре непохожих программ может быть выбрана 45-ю способами, причем по набору coreutils из 106 программ можно построить C_{106}^2 пар. Таким образом, число непохожих пар почти в 108 раз больше множества пар похожих программ. Для уменьшения дисбаланса в соотношении похожих/непохожих пар программ множество векторов показателей непохожих программ строится на основе пар, полученных в рамках фиксированного компилятора и фиксированной опции оптимизации.

Таким образом, можно построить $C_{106}^2 \cdot 45 = 250425$ векторов показателей. Для построения обучающих и тестовых выборок из числа непохожих случайным образом выбираются 104940 пары, а из числа похожих пар используются все 104940. Построение обучающих и тестовых выборок, а также обучение выполнено с помощью пакета sklearn [36] для языка Python. Для экспериментального выбора лучшего классификатора исследуются классификаторы *GaussianNB* (*GNB*), *LogisticRegression* (*LR*), *SVM KNeighborsClassifier* (*KNC*), *LinearDiscriminantAnalysis* (*LDA*), *RandomForestClassifier* (*RFC*), *DecisionTreeClassifier* (*DTC*) из пакета sklearn.

Табл. 3. Результаты обучения классификаторов
Table 3. Learning outcomes of classifiers

Класси-фикатор	Номер эксп-та	Precision (Похожие)	Precision (Непохожие)	Recall (Похожие)	Recall (Непохожие)	Accuracy
SVM	1	0,89	0,87	0,87	0,90	0,88
	2	0,99	0,82	0,78	0,99	0,89
	3	0,98	0,90	0,89	0,98	0,93
DTC	1	0,91	0,91	0,91	0,91	0,91
	2	0,94	0,93	0,93	0,94	0,93
	3	0,97	0,97	0,97	0,97	0,97
RFC	1	0,95	0,94	0,94	0,95	0,94
	2	0,98	0,93	0,93	0,98	0,95
	3	**0,99**	**0,98**	**0,98**	**0,99**	**0,98**
LDA	1	0,90	0,85	0,84	0,90	0,87
	2	0,98	0,82	0,78	0,98	0,88
	3	0,96	0,87	0,86	0,96	0,91
KNC	1	0,92	0,94	0,94	0,92	0,93
	2	0,96	0,92	0,92	0,96	0,94
	3	**0,99**	**0,98**	**0,98**	**0,99**	**0,98**
LR	1	0,88	0,86	0,86	0,88	0,87
	2	0,95	0,83	0,81	0,96	0,89
	3	0,94	0,89	0,88	0,94	0,91
GNB	1	0,71	0,62	0,53	0,78	0,65
	2	0,66	0,75	0,81	0,58	0,70
	3	0,68	0,73	0,76	0,63	0,7

В работе проведено пять экспериментов: эксперимент 1 – обучение классификаторов только на множестве универсальных показателей, эксперимент 2 – обучение только на множестве специализированных показателей, эксперимент 3 – обучение на всех показателях, эксперимент 4 – обучение классификаторов только на множестве универсальных показателей с исключением ряда показателей, эксперимент 5 – обучение на множестве всех показателей с исключением ряда универсальных. Целью первых трех экспериментов является исследование эффекта от совместного использования универсальных и специализированных показателей, а цель четвертого и пятого – исследовать вклад отдельных универсальных показателей в точность обучения классификаторов.

Результаты первых трех экспериментов приведены в табл. 3, где каждому классификатору соответствуют три строки значений: значения *i*-ой строки получены в рамках эксперимента *i*, *i*=1,2,3. Из табл. 3 видно, что классификаторы на основе специализированных показателей (вторая строка для каждого классификатора) обладают ожидаемо большей точностью (accuracy), чем классификаторы на основе универсальных показателей (первая строка). Тем не менее, совместное применение этих показателей увеличивает точность для всех рассмотренных классификаторов (третья строка каждого классификатора). Это свидетельствует о целесообразности совместного применения всех показателей из табл. 2. Из табл. 3 также видно, что классификаторы *RFC* и *KNC* демонстрируют наибольшую точность при сравнении похожих и непохожих программ; наименьшую точность продемонстрировал классификатор *GNB*.

С целью оценки влияния отдельных показателей на решение классификатора, штатными средствами пакета sklearn для классификаторов *RFC* и *DTC* получены веса показателей во всех первых трех экспериментах (см. табл. 4). Анализ весов показателей в табл. 4 для эксперимента 3 показывает, что средний вес специализированных показателей больше среднего веса универсальных показателей. В частности, для *DTC* суммарный вес первых равен 0.711, а для *RFC* – 0.632. Откуда для *DTC* средний вес специализированных показателей равен 0.118, а средний вес универсальных характеристик – 0.032; для *RFC* средний вес специализированных характеристик равен 0.105, а средний вес универсальных характеристик – 0.041. Таким образом, показатели на основе специализированных характеристик вносят больший вклад в принятие решения о похожести программ. При этом стоит отметить, что среди показателей на основе таких характеристик наибольшим весом, как в случае *DTC*, так и в случае *RFC*, обладает ninjadiff. Это может быть связано с тем, что перед выгрузкой данных в формате BinExport, исполнимый файл анализировался трижды (используя прикладной программный интерфейс пакета Binary Ninja), в то время как для дизассемблеров IDA Pro и Ghydra, анализ выполнялся только один раз.

Табл. 4. Веса показателей для классификаторов DecisionTreeClassifier и RandomForestClassifier
Table 4. Feature weights for the DecisionTreeClassifier and RandomForestClassifier classifiers

Показатель	**Тип показателя**	**Эксперимент 1**		**Эксперимент 2**		**Эксперимент 3**	
		DTC	*RFC*	*DTC*	*RFC*	*DTC*	*RFC*
ssdeep	0	**0,019**	**0,016**	-	-	**0,003**	**0,004**
sdhash		0,117	0,140	-	-	0,086	0,079
tlsh		0,181	0,151	-	-	**0,012**	0,036
tlshxlen		0,300	0,256	-	-	0,025	0,078
lzma		**0,046**	**0,061**	-	-	**0,006**	**0,015**
bz2		**0,062**	**0,063**	-	-	0,040	**0,023**
deflate		**0,051**	**0,068**	-	-	**0,012**	**0,018**
myersdiff		0,135	0,128	-	-	0,022	0,056
levenshteindiff		0,088	0,118	-	-	0,089	0,073
idadiff	1	-	-	0,061	0,119	0,025	0,061
idadiffc		-	-	0,055	0,115	0,013	0,064
ghydradiff		-	-	0,063	0,113	0,025	0,059
ghydradiffc		-	-	0,305	0,216	0,220	0,119
ninjadiff		-	-	0,429	0,292	0,391	0,209
ninjadiffc		-	-	0,087	0,145	0,030	0,103

Веса в табл. 4 дополнительно подтверждают выводы исследования [9]: алгоритмы tlsh и sdhash показывают лучший результат (согласно табл. 4, дают больший вклад в принятое классификатором решение) при нахождении похожих файлов с исполнимым кодом, чем алгоритм ssdeep. Возможная причина этого заключается в том, что алгоритм, лежащий в основе ssdeep, зависит от длины входных данных: чем больше данных, тем длиннее хеш-значение. А так как применение разных компиляторов и разных опций компиляции к одному исходному коду приводит к созданию файлов разной длины, то и хеш-значения будут разной длины. И поэтому соответствующий алгоритм сравнения хеш-значений одинаковых по функциональности, но разных по размеру программ дает низкий уровень похожести.

Табл. 5. Влияние показателей на точность классификаторов
Table 5. Influence of features on the accuracy of classifiers

Класси-фикатор	**Номер эксп-та**	**Исключенные показатели**				
		–	*ssdeep*	*ssdeep, bz2*	*ssdeep, bz2, lzma*	*ssdeep, bz2, lzma, deflate*
SVM	4	0,88	0,88	0,88	0,88	0,88
	5	0,93	0,94	0,94	0,94	0,94
DTC	4	0,91	0,91	0,91	0,91	0,9
	5	0,97	0,97	0,97	0,97	0,97
RFC	4	0,94	0,94	0,94	0,94	0,92
	5	0,98	0,98	0,98	0,98	0,98
LDA	4	0,87	0,87	0,86	0,86	0,86
	5	0,91	0,91	0,91	0,91	0,9
KNC	4	0,93	0,93	0,93	0,93	0,91
	5	0,98	0,98	0,98	0,98	0,98
LR	4	0,87	0,87	0,87	0,87	0,86
	5	0,91	0,91	0,91	0,91	0,9
GNB	4	0,65	0,65	0,65	0,67	0,67
	5	0,7	0,69	0,69	0,69	0,69

В табл. 5 приведены результаты четвертого и пятого экспериментов, которые отличаются от экспериментов 1 и 3 только тем, что исследовалось влияние показателей *ssdeep*, *bz2*, *lzma*, *deflate* на точность принимаемого классификатором решения, путем последовательного исключения этих показателей при обучении классификаторов. Перечисленные показатели выбраны в связи с тем, что в эксперименте 1 они обладают наименьшими весами, согласно табл. 4. В эксперименте 3 наименьшими весами обладают *ssdeep*, *lzma*, *deflate, tlsh*, однако показатель *tlsh* в четвертом и пятом экспериментах не исключался при обучении, так как, согласно первому эксперименту, он обладает высоким весом. Несмотря на то, что вес показателя *bz2* в третьем эксперименте больше веса *tlsh*, показатель *bz2* в четвертом и пятом экспериментах исключался. Это связано с тем, что другие два показателя похожести на основе коэффициента сжатия файлов (*lzma* и *deflate*) обладают малым весом, поэтому представляется обоснованным исследование влияния на решение классификатора всех показателей, построенных на основе коэффициентов сжатия файлов.

Серым цветом в табл. 5 выделены те ячейки, в которых наблюдается снижение уровня точности по сравнению со значением в ячейке из той же строки и предыдущего столбца: исключение из обучения показателя приводит к снижению уровня точности. Заметим, что для всех классификаторов, за исключением классификатора *GNB*, с наименьшим уровнем точности, исключение из обучения показателя *ssdeep* не уменьшает точности классификации, а для классификатора *SVM* даже увеличивает точность. Поэтому представляется, что значение хеш-функции ssdeep можно исключить из числа статических характеристик. К

нерекомендуемым к применению хеш-функциям, вероятно, стоит отнести и dcfldd [37], длина хеш-значения которой по аналогии с ssdeep, зависит от длины входного файла. Заметим, что дополнительное исключение всех показателей, основанных на степени сжатия файлов, приводит к уменьшению точности для всех классификаторов (хотя бы в одном из экспериментов 4 или 5), кроме *SVM* и *GNB*. Это может свидетельствовать в пользу применения таких показателей при обнаружении похожести файлов.

5. Заключение

Отличительной особенностью предлагаемого в работе способа сравнения файлов с исполнимым кодом является использование двух типов характеристик: универсальных и специализированных. Результаты показали, что наибольший вклад в принятие решения классификатором вносят специализированные характеристики, средний вес которых, как минимум в два раза, больше среднего веса универсальных характеристик. Тем не менее совместное применение характеристик разных типов повышает точность классификации, а наилучшей точностью классификации обладают классификаторы *KNeighborsClassifier* и *RandomForestClassifier*.

Начальное описание программы может быть расширено значениями других средств анализа, в том числе значениями других нечетких хеш-функций, не учитывающих формат данных, например, mvhash-b [38], а также, например, значениями хеш-функций, которые вычисляются по графу потока управления программы, таких, как machoce [39], machoke [40] (однако не для всех рассмотренных дизассемблеров имеются реализации этих алгоритмов). Разные алгоритмы сжатия также могут использоваться для увеличения размерности вектора показателей, однако, как показывают результаты для одного из лучших классификаторов – для *RFC* (см. табл. 4) – соответствующие показатели вносят наименьший вес в принятие решения (если не учитывать показатель ssdeep, вес которого минимален). Вектор показателей также может быть расширен за счет использования других средств сравнения файлов исполнимого кода, например, в дополнение к BinDiff может быть использовано средство BCC, предложенное в [41] и показавшее, согласно [20], наибольшую точность сравнения бинарного кода.

Дальнейшим направлением исследования является оптимизация набора характеристик, применение характеристик, получаемых средствами динамического анализа программ, например, средствами символьного анализа, исследование качества предлагаемого метода для оценки похожести программ из набора, отличного от coreutils, а также оценка качества обфусцирующих преобразований с помощью построенных классификаторов.

Авторы благодарят рецензентов за замечания, позволившие глубже взглянуть на проблему сравнения бинарных файлов, в частности, на проблему выбора обучающего множества пар похожих и непохожих программ.

Список литературы / References

[1] Нурмухаметов А.Р. Применение диверсифицирующих и обфусцирующих преобразований для изменения сигнатуры программного кода. Труды ИСП РАН, том 28, вып. 5, 2016 г., стр. 93-104 / Nurmukhametov A. R. The application of compiler-based obfuscation and diversification for program signature modification. Trudy ISP RAN/Proc. ISP RAS, 2016, vol. 28, issue 5, pp. 93-104 (in Russian). DOI: 10.15514/ISPRAS-2016-28(5)-5.

[2] Терешкин А.В. Разработка и исследование метода защиты от удаленных атак на основе диверсификации программного обеспечения. Диссертация на соискание степени кандидата технических наук. Южный федеральный университет, 2007 г., 157 стр. / Tereshkin A.V. Development and research of a method of protection against remote attacks based on software diversification. Dissertation for the degree of candidate of technical sciences. South Federal University, 2007, 157 p. (in Russian).

[3] Collberg C., Thomborson C. Watermarking, Tamper-Proofng, and Obfuscation – Tools for Software Protection, IEEE Transactions on Software Engineering, vol. 28, issue 8, 2002, pp. 735-746.

[4] Walenstein A., El-Ramly M. et al. Similarity in programs. In Dagstuhl Seminar Proceedings. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, vol. 6301, 2007, 8 p.

[5] Borisov P.D., Kosolapov Yu.V. On the automatic analysis of the practical resistance of obfuscating transformations. Automatic Control and Computer Sciences, vol. 54, issue 7, 2020, pp. 619-629.

[6] Holder W., McDonald J.T., Andel T.R. Evaluating optimal phase ordering in obfuscation executives. In Proc. of the 7th Software Security, Protection, and Reverse Engineering/Software Security and Protection Workshop, 2017, pp. 1-12.

[7] Borisov P.D., Kosolapov Yu.V. Similarity Features for The Evaluation Of Obfuscation Effectiveness. In Proc. of the International Conference on Decision Aid Sciences and Application (DASA), 2020, pp. 898-902.

[8] Борисов П.Д., Косолапов Ю.В. О характеристиках символьного исполнения в задаче оценки качества обфусцирующих преобразований. Моделирование и анализ информационных систем, том. 28, вып. 1, 2021 г., стр. 38-51. / Borisov P.D., Kosolapov Yu.V. On characteristics of symbolic execution in the problem of assessing the quality of obfuscating transformations. Modeling and Analysis of Information Systems, vol. 28, issue 1, 2021, pp. 38-51 (in Russian).

[9] Pagani F., Dell'Amico M., Balzarotti D. Beyond precision and recall: understanding uses (and misuses) of similarity hashes in binary analysis. In Proc. of the Eighth ACM Conference on Data and Application Security and Privacy, 2018, pp. 354-365.

[10] Ding S.H., Fung, B.C., Charland P. Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In Proc. of the IEEE Symposium on Security and Privacy (SP), 2019, pp. 472-489.

[11] Юмаганов А.С. Поиск похожих символьных последовательностей и графов на основе методов машинного обучения. Диссертация на соискание степени кандидата технических наук. Самарский национальный исследовательский университет имени академика С.П. Королева, 2020 г., 130 стр. / Yumaganov A.S. Search for similar character sequences and graphs based on machine learning methods. Dissertation for the degree of candidate of technical sciences. Samara National Research University named after Academician S.P. Korolev, 2020, 130 p. (in Russian).

[12] Саргсян С., Курмангалеев Ш. и др. Масштабируемый инструмент поиска клонов кода на основе семантического анализа программ. Труды ИСП РАН, том 27, вып. 1, 2015 г., стр. 39-50. / Sargsyan S., Kurmangaleev S. et al. Scalable code clone detection tool based on semantic analysis, Proceedings of ISP RAS, vol. 27, issue 1, 2015, pp. 39–50 (in Russian). DOI: 10.15514/ISPRAS-2015-27(1)-3.

[13] Осадчая А.О., Исаев И.В. Метод поиска клонов в программном коде. Научно-технический вестник информационных технологий, механики и оптики, том 20, вып. 5, 2020 г., стр. 714-721. / Osadchaya A.O., Isaev I.V. Search of clones in program code. Scientific and Technical Journal of Information Technologies, Mechanics and Optics, vol. 20, issue 5, 2020, pp. 714-721 (in Russian).

[14] Bellon S., Koschke R. et al. Comparison and evaluation of clone detection tools. IEEE Transactions on Software Engineering, vol. 33, issue 9, 2007, pp. 577-591.

[15] Sarantinos N., Benzaid C. et al. Forensic malware analysis: The value of fuzzy hashing algorithms in identifying similarities. In Proc. of the IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom), 2016, pp. 1782-1787.

[16] Oliver J., Cheng C., Chen Y. TLSH – a loclity sensitive hash. In Proc. of the Fourth Cybercrime and Trustworthy Computing Workshop, 2013, pp. 7-13.

[17] Mohsen R. Quantitative measures for code obfuscation security. PhD Thesis, Imperial College London, 2016, 216 p.

[18] BinExport. Available at: https://github.com/google/binexport, accessed 10.10.2022.

[19] BinDiff. Available at: https://www.zynamics.com/, accessed 10.10.2022.

[20] Arutunian M., Hovhannisyan H. et al. A Method to Evaluate Binary Code Comparison Tools. In Proc. of the 2021 Ivannikov Memorial Workshop (IVMEM), 2021, pp. 3-5.

[21] A powerful disassembler and a versatile debugger. Available at: https://hex-rays.com/ida-pro/, accessed 10.10.2022.

[22] Ghidra Software Reverse Engineering Framework. Available at: https://github.com/NationalSecurityAgency/ghidra, accessed 10.10.2022.

[23] Binary Ninja. Available at: https://binary.ninja/, accessed 10.10.2022.

[24] Kornblum J. Identifying almost identical files using context triggered piecewise hashing. Digital investigation, vol. 3, 2006, pp. 91-97.

[25] Roussev V. An evaluation of forensic similarity hashes. Digital investigation, vol. 8, 2011, pp. S34-S41.
[26] Pavlov I. Lzma SDK (software development kit). Available at: https://7-zip.org/sdk.html, accessed 10.10.2022.
[27] Seward, Julian. bzip2 and libbzip2. Avaliable at http://www. bzip. org, accessed 10.10.2022.
[28] Oswal S., Singh A., Kumari K. Deflate compression algorithm. International Journal of Engineering Research and General Science, vol. 4, issue 1, 2016, pp. 430-436.
[29] Radare2. Available at: https://rada.re/, accessed 10.10.2022.
[30] Myers E. W. An O(ND) difference algorithm and its variations. Algorithmica, vol. 1, issue 1, 1986, pp. 251-266.
[31] В.И. Левенштейн. Двоичные коды, способные исправлять удаления, вставки и обращения. Доклады Академии Наук СССР, том 163, no. 4, 1966 г., стр. 845-848. / V.I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. Soviet physics doklady, vol. 10, no. 8, 1966, pp. 707-710.
[32] Coreutils – GNU core utilities. Available at: https://www.gnu.org/software/coreutils/, accessed 10.10.2022.
[33] GCC, the GNU Compiler Collection. Available at: https://gcc.gnu.org/, accessed 10.10.2022.
[34] Clang: a C language family frontend for LLVM. Available at: https://clang.llvm.org/, accessed 10.10.2022.
[35] AMD Optimizing C/C++ and Fortran Compilers (AOCC). Available at: https://developer.amd.com/amd-aocc/, accessed 10.10.2022.
[36] Scikit-learn. Machine learning in Python. Available at: https://scikit-learn.org/, accessed 10.10.2022.
[37] Dcfldd. Available at: https://dcfldd.sourceforge.net/, accessed 10.10.2022.
[38] Breitinger F., Astebol K.P. et al. mvhash-b – A new approach for similarity preserving hashing. In Proc. of the Seventh International Conference on IT Security Incident Management and IT Forensics, 2013, pp. 33-44.
[39] Machoc hash. Available at: https://github.com/ANSSI-FR/polichombr/blob/dev/docs/MACHOC_HASH.md, accessed 10.10.2022.
[40] Machoke. Available at: https://github.com/conix-security/machoke, accessed 10.10.2022.
[41] Aslanyan H., Avetisyan A. et al. Scalable Framework for Accurate Binary Code Comparison. In Proc. of the 2017 Ivannikov ISPRAS Open Conference (ISPRAS), 2017, pp. 34-38.

Информация об авторах / Information about authors

Петр Дмитриевич БОРИСОВ является заведующим лабораторией в ФГАНУ НИИ «Спецвузавтоматика». Сфера научных интересов: исследование программного кода, теоретическая и практическая обфускация кода.

Petr Dmitrievich BORISOV is the head of the laboratory of the FGANU Research Institute "Spetsvuzavtomatika". Research interests: code analisys, theoretical and practical code obfuscation.

Юрий Владимирович КОСОЛАПОВ – кандидат технических наук, доцент кафедры алгебры и дискретной математики института математики, механики и компьютерных наук им. И.И. Воровича. Сфера научных интересов: помехоустойчивые коды в криптографии, теоретическая и практическая обфускация кода.

Yuri Vladimirovich KOSOLAPOV – Candidate of Technical Sciences, Associate Professor, Department of Algebra and Discrete Mathematics, Institute of Mathematics, Mechanics and Computer Science named after I.I. Vorovich. Research interests: error-correcting codes in cryptography, theoretical and practical code obfuscation.