ПРОГНОЗИРОВАНИЕ РАЗМЕРА ИСХОДНОГО КОДА БИНАРНОЙ ПРОГРАММЫ В ИНТЕРЕСАХ ЕЕ ИНТЕЛЛЕКТУАЛЬНОГО РЕВЕРС-ИНЖИНИРИНГА

Израилов К. Е.[1]

DOI: 10.21681/2311-3456-2024-4-13-25

Цель исследования: *повышение эффективности поиска уязвимостей в машинном коде программ путем его реверс-инжиниринга на базе генетических алгоритмов, для чего решается частная задача прогнозирования размера исходного кода на языке программирования С по его скомпилированной версии.*

Методы исследования: *обзор работ, системный анализ, синтез метода, компьютерное моделирование, эксперимент.*

Полученные результаты: *создан метод получения зависимости размера исходного кода (выраженного в токенах языка программирования) от соответствующего ему машинного кода, что позволяет решать частную задачу определения длины хромосомы особи в рамках реверс-инжиниринга на базе генетических алгоритмов; разработан программный прототип, реализующий указанный метод, с помощью которого проведен эксперимент (с использованием датасета ExeBench, содержащего около 200 тысяч функций на языке программирования С), позволивший вывести аналитическую зависимость между размерами исходного и машинного кодов.*

Научная новизна *заключается как в общем развитии нового интеллектуального направления реверс-инжиниринга машинного кода, так и в авторском решении частной задачи прогнозирования размера исходного кода программы по ее бинарному представлению.*

Ключевые слова: *реинжиниринг, обратная разработка, обратный инжиниринг, генетический алгоритм, уязвимость, машинный код, метод, прототип, эксперимент, аналитическая зависимость.*

PREDICTING THE SIZE OF THE SOURCE CODE OF A BINARY PROGRAM IN THE INTERESTS OF ITS INTELLECTUAL REVERSE ENGINEERING

Izrailov K. E.[2]

The goal of the investigation: *increasing the efficiency of searching for vulnerabilities in machine code of programs by reverse engineering it based on genetic algorithms, for which the particular problem of predicting the size of source code in the C programming language from its compiled version is solved.*

Research methods: *works survey, system analysis, synthesis, computer modeling, experiment.*

Result: *a method has been created for obtaining the dependence of the size of the source code (expressed in programming language tokens) on the corresponding machine code, which allows solving the particular problem of determining the length of an individual's chromosome within the framework of reverse engineering based on genetic algorithms; a software prototype was developed that implements the specified method, with the help of which an experiment was carried out (using the ExeBench dataset containing about 200 thousand functions in the C programming language), which made it possible to derive an analytical relationship between the sizes of the source and machine codes.*

The scientific novelty *consists both in the general development of a new intellectual direction of reverse engineering of machine code, and in the author's solution to the particular problem of predicting the size of a program's source code from its binary representation.*

Keywords: *reengineering, reverse engineering, genetic algorithm, vulnerability, machine code, method, prototype, experiment, analytical dependence.*

1 Израилов Константин Евгеньевич, кандидат технических наук, доцент, старший научный сотрудник лаборатории проблем компьютерной безопасности Санкт-Петербургского Федерального исследовательского центра Российской академии наук, Санкт-Петербург, ORCHID: http://orcid.org/0000-0002-9412-5693. Scopus Author ID: 56122749800. E-mail: konstantin.izrailov@mail.ru

2 Konstantin E. Izrailov, Ph.D., Docent, Senior Researcher of Laboratory of Computer Security Problems of St. Petersburg Federal Research Center of the Russian Academy of Sciences, Saint-Petersburg. ORCHID: http://orcid.org/0000-0002-9412-5693. Scopus Author ID: 56123238800. E-mail: konstantin.izrailov@mail.ru

Введение

Безопасность программного обеспечения (далее – ПО) является актуальнейшей проблемой современного IT-мира [1]. Одним из путей ее разрешения считается анализ конечного продукта (т.е. программы) на предмет наличия уязвимостей, за которым следует их нейтрализация. При этом, если обнаружение достаточно тривиальных уязвимостей и возможно с помощью автоматических средств, то в случае сложных алгоритмических или архитектурных уязвимостей требуется привлечение экспертов по безопасности программного кода. И если при разработке программ на интерпретируемых языках программирования и/или компилируемых в байт-код (например, на Python или Java) эксперту может потребоваться изучение хорошо понятного исходного кода (далее – ИК), то для ряда популярных языков программирования (например, C и C++) программа имеет бинарный вид, содержащий машинный код (далее – МК), ручной анализ которого будет иметь сверхвысокую трудоемкость. Ситуация усугубляется тем, что «заложение» уязвимостей в код злоумышленником производится сознательно с применением механизмов усложнения их обнаружения – в том числе, через запутывание алгоритмов и ослабление архитектуры программного продукта.

Вышесказанное указывает на наличие в предметной области проблемного вопроса, как противопоставления следующих потребностей и возможностей. Во-первых, существующие методы и средства имеют высокую эффективность лишь для тривиальных уязвимостей или при наличии псевдо-исходного кода (приставка «псевдо» отражает тот факт, что код не будет в точности соответствовать исходному, он обязан лишь быть синтаксически корректным и компилироваться в анализируемый машинный). С другой стороны, для огромного количества областей IT-мира программы представляют собой машинный код, поиск нетривиальных уязвимостей в котором с применением экспертных методов имеют недопустимо высокую трудоемкость.

Одним из существующих подходов к поиску уязвимостей (в особенности обладающих средним и высоким уровнем внедрения) в машинном коде является его предварительное преобразование в человеко-ориентированную форму путем декомпиляции [2, 3] – т.е. получения соответствующего псевдо-исходного кода (а потенциально – программных алгоритмов и архитектуры). Однако, существующие средства декомпиляции (например, продукт IDA Pro [4] с плагином Hex-Rays и Ghidra [5]) используют встроенные сложные алгоритмы преобразования конструкций МК и их комбинаций в подобные элементы в ИК. Богатый авторский опыт работы в области реверс-инжиниринга [6–8] позволяет утверждать, что такие средства зачастую вызывают программные исключения, восстанавливают не компилируемый или неполный код, или же конечный результат слабо подходит для ручной обработки. Альтернативным подходом, развиваемым автором, является «генетический реверс-инжиниринг машинного кода» (далее – ГРИМК), основанный на применении искусственного интеллекта в части использования генетических алгоритмов для осуществления декомпиляции. Суть ГРИМК заключается в решении оптимизационной задачи подбора вариантов ИК, наиболее близких к МК после компиляции, декомпиляцию которого необходимо произвести; выбор архитектуры CPU и компилятора считается частично решенной задачей [9]. Результатом применения ГРИМК является не просто псевдо-исходный код, лишь отражающий логику ПО, а код реального языка программирования, гарантированно компилируемый в исследуемый МК. Такой ИК может быть проанализирован экспертом на предмет наличия в нем уязвимостей, модифицирован и скомпилирован в безопасный МК.

Принцип работы авторского подхода ГРИМК состоит в итеративном подборе такого варианта ИК, который бы компилировался в заданный МК. Для этого применяется «умный» перебор элементов языка программирования, из которых составляется ИК, сравниваемый после компиляции с исследуемым МК; с этой позиции, ГРИМК может быть отнесет к области генетического программирования, предназначенного для генерации или изменения программ, решающих некоторую вычислительную задачу [10]. Как следствие, одной из задач (возможно косвенной, но тем не менее, принципиально важной) в данном подходе является определение размера ИК. В ином случае пришлось бы подбирать не только элементы последовательности, но и ее размер, что не только качественно усложнит реализацию подхода, но и теоретически сделает задачу нерешаемой (поскольку, размер исходного кода может увеличиваться практически до бесконечности). Естественно, целесообразно составлять ИК не из отдельных символов, а из конструкций его языка программирования (для языка C/C++ это будут заголовки функций [11] и переменные [12], элементы границ блоков «{» и «}», операторы «+» или «–», операторы «if» или «else» и т.п.).

Таким образом, задачу текущего исследования можно сформулировать следующим образом:

«Создание метода и программного средства прогнозирования размера исходного кода по скомпилированному из него машинному коду».

Детали постановки и решения задачи будут раскрыты в статье далее, ее же актуальность обосновывается отсутствуем каких-либо подходящих методов или средства из числа существующих. Необходимо отметить, что данное исследование является продолжением предыдущего (а в частности – развитием, исправлением недочетов и адаптацией для российского научного сегмента), результаты которого были апробированы в 2024 году на международной научно-практической конференции «Индустрия 4.0» (https://smartindustrycon.ru/), публикующей труды в цифровой библиотеке «IEEE Xplore».

Проведем обзор работ, близких к задаче исследования – определению зависимости между размерами ИК и МК; в случае отсутствия подходящих публикаций рассмотрим более общие, затрагивающие вопрос определения длины хромосомы в генетических алгоритмах, а также ее представления.

В исследовании [13] приводится метод преобразования МК в алгоритмы без необходимости восстановления ИК. Приложением метода указывается обнаружение вредоносного кода, в особенности, ИК которого был сознательно «запутан». Метод в своей работе использует графы потоков управления для представления программы в виде алгоритмов.

Авторы работы [14] описывают средство BinDeep, предназначенное для сопоставления фрагментов ИК с аналогичными по функционалу фрагментами в МК. В качестве практической значимости решения указывается поиск клонов кода, и, в частности, содержащего вредоносный код.

В работе [15] для декомпиляции фрагментов МК применяется рекуррентная нейронная сеть. В качестве особенности предложенного подхода указывается его независимость от языка программирования. В основе модели нейронной сети заложено ее обучение на шаблонах ИК. Применяться решение может для ручного анализа ПО в случае, когда его ИК отсутствует.

Исследование [16] также посвящено декомпляции, но в части восстановления встроенных в МК функций, оптимизированных в процессе компиляции из ИК. Авторский метод построен на основе машинного обучения с учителем и может быть объединен с продуктом Ghidra. Применением метода является анализ ПО, соответствующего МК, для обнаружения вредоносного кода и определения факта хищения интеллектуальной собственности.

Работа [17] посвящена прогнозированию размера ИК кода, однако не по МК, а согласно конкретному процессу программной инженерии. Для этого предлагается метод из 6 шагов, учитывающих следующие особенности ПО: нескорректированные веса субъектов и вариантов использования, нескорректированные варианты использования, техническая сложность и факторы окружающей среды, скорректированные варианты использования, трудозатраты в человеко-часах. Таким образом, имеется возможность формального предсказания размера ИК крупных проектов.

В работе [18] представлена модель COCOMO II, предназначенная для оценки размера исходного кода программных средств. При этом расчет размера производится как с учетом нового разработанного кода, так и повторно используемого, а также модифицированного для адаптации. Аналитическая модель основана на 8 компонентах, для каждого из которых приводятся методики расчета. В качестве назначения модели указана оценка трудоемкости и длительность разработки программных продуктов.

В исследовании [19] производится общее сравнение генетических алгоритмов и генетического программирования, указывая, что в отличие от первых, последние имеют переменную длину хромосом, структура которых, как правило, является не строкой, а деревом.

Авторы в [20] описывают применение кода Грея для кодирования целочисленных признаков в результате чего хромосома в генетическом алгоритме имеет представление последовательности бит; длина же хромосомы, соответственно, определяется максимальным кодируемым числом. Одним из результатов такого способа кодирования является то, что изменение одного бита в результате мутации приведет к замене закодированного с помощью хромосомы числа на смежное – т.е. к небольшому изменению особи, что является достаточно важной особенностью работы генетического алгоритма.

В работе [21] исследуется влияние различных параметров генетических алгоритмов на его работу при решении задачи о рюкзаке (укладывание ценных вещей при ограничении его вместимости), а именно следующих: число вычислений целевой функции, тип селекции и скрещивания, а также вероятность мутации и опциональное добавление в новое поколение лучшего индивидуума предыдущего. При этом длина и представление хромосомы никак не рассматривались.

Исследование [22] посвящено решению задачи покрытия территории группой беспилотных летательных аппаратов (далее – БПЛА) с помощью генетического алгоритма. Для этого, в качестве возможных представлений хромосомы указывается как последовательность точек, которые необходимо посетить одному БПЛА, так и аналогичное объединение траекторий для нескольких БПЛА. Таким образом, длина хромосомы должна определяться минимально-

необходимым количеством точек траектории для покрытия всей заданной территории (чему в работе внимания не уделяется).

Краткий обзор работ показал практически полное отсутствие решений (под которым здесь понимаются методы и их программные реализации), применимых к задаче текущего исследования.

Генетический реверс-инжиниринг

Опишем далее общую идею авторского генетического реверс-инжиниринга, уделив особое внимание используемой терминологии и постановке задачи исследования.

Идея

Классический подход к декомпиляции может быть назван обратным (или реверс) инжинирингом, поскольку он согласно жизненному циклу ПО [23] осуществляет преобразование от текущего представления (т.е. МК или аналогичного ему ассемблерного кода) к предыдущему (т.е. ИК). При этом, как указывалось, могут применяться как автоматические средства, так и ручной труд эксперта. Предлагаемый же автором подход ГРИМК с этой точки зрения может быть назван «псевдо-прямым», поскольку он стремится подобрать такое представление ИК, которое бы при компиляции преобразовывалось в исследуемый МК. Таким образом, ГРИМК решает оптимизационную задачу [24], в которой параметром является последовательность конструкций ИК (символов, токенов, узлов абстрактного синтаксиса, шаблонов или иных сущностей), которые бы при компиляции получали МК, наиболее близкий к необходимому (в идеале – к оригиналу). Получение идентичных МК означает решение оптимизационной задачи и обнаружение ИК – т.е. достижение глобального экстремума. В этом аспекте, ГРИМК схож с ручной декомпиляцией, поскольку эксперт точно также подбирает такие конструкции ИК, последовательность которых бы в точности соответствовала анализируемому машинному. Данный процесс (для эксперта) усложняется тем, что неверный выбор конструкций в начале ИК может повлиять на невозможность подбора корректных конструкций в дальнейшем. Для обоснования этого приведем пример процесса реверс-инжиниринга тривиального МК (использованы следующие условные инструкции: MOV – операция сохранения значения второго аргумента в первом, CALL – вызов функции с аргументом в регистре AX и возвратом результата в том же регистре; здесь и далее префикс из числа и двоеточия соответствует порядковому номеру строки)

```
1: MOV AX, x
2: CALL funct
3: MOV y, AX
```

полученного из ИК:

```
y = funct(x);
```

В случае прямого преобразования инструкций МК (т.е. без возвратов и изменений в уже сформированном коде) в конструкции ИК строки 1 и 2 позволят получить следующий код:

```
funct(x)
```

Однако, строка 3 интерпретируется, как сохранение результата «funct()» в переменной «y», что потребует добавления соответствующего оператора присваивания перед функцией:

```
y = funct(x)
```

В случае последовательного преобразования такое изменение будет невозможно, поскольку ИК для вызова функций уже сформирован.

Прямой перебор

Отметим, что теоретически, задача получения ИК по соответствующему ему МК могла бы быть решена полным перебором всех конструкций ИК; тем не менее, на практике такой способ не применим, поскольку данный процесс будет недопустимо длительным. Приведем очень грубые примеры оценки такого процесса для ИК на языке программирования С, учитывая следующие ограничения и условия:

- символы текста ИК состоят из комбинации букв английского алфавита, цифр, знаков и пробелов;
- каждый символ может принимать одно из 50 значений;
- в случае рассмотрения ИК, как последовательности лексем синтаксиса языка программирования (идентификаторов, ключевых слов, операторов, цифр и специальных символов), их количество считается равным примерно 100;
- используется понятное эксперту деление текста ИК на строки (в конце логических конструкций);
- размер текста является условно минимальным за счет применения коротких имен переменных и функций, а также оптимизации количества пробелов, используемых конструкций, операторов и т.п.;
- приблизительное время компиляции одного ИК (с учетом длительности запуска процесса, загрузки исходного файла и сохранения ассемблерного) занимает 1 секунду;
- при переборе вариантов ИК не учитывается синтаксис языка программирования и, следовательно, даже изначально некорректная комбинация символов считается потенциально компилируемой в исследуемый МК (а не отбрасывается, что было бы более логичным).

Пример 1. ИК состоит из базового сложения двух переменных (без завершающего «;») и является следующим:

```
1: x+y
```

и содержит 3 символа (без учета перевода строки). Таким образом, для подбора такого ИК даже с учетом знаний о его длине максимально потребуется перебрать $50^3 = 125000 \approx 10^5$ вариантов комбинаций символов, компиляция которых суммарно займет ~35 часов.

Если рассматривать ИК, как последовательность лексем длиной 3, количество вариантов будет равно $100^3 = 10^6$, компиляция которых займет ~350 часов.

Пример 2. ИК состоит из «функции-заглушки», возвращающей число 0, является следующим:

```
1: int f()
2: {
3:  return 0;
4: }
```

и содержит 23 символа (с учетом одного на каждый перевод строки). Таким образом, поиск ИК максимально потребует перебрать $50^{23} \approx 10^{39}$ вариантов комбинаций символов, время компиляции которых уже будет сверхвысоким (отметим, что как считается, время жизни Вселенной не превосходит 10^{18} секунд).

Если рассматривать ИК, как последовательность лексем длиной 9 («int», «f», «(», «)», «{» и т.п.), количество вариантов будет равно $100^9 = 10^{18}$, компиляция которых также займет сверхвысокое время (хотя и на 20 порядков меньшее, чем при представлении ИК, как последовательности символов).

Пример 3. ИК состоит из реальной функции определения максимального из двух чисел:

```
1: int f(int x, int y)
2: {
3:  if(x > y)
4:   return x;
5:  else
6:   return y;
7: }
```

и содержит 66 символов (с учетом переводов строк). Таким образом, поиск ИК максимально потребует перебрать $50^{66} \approx 10^{112}$ вариантов комбинаций символов, что скорее всего недостижимо даже теоретически.

Если рассматривать ИК, как последовательность лексем длиной 24 («int», «f», «(», «int», «x» , «,», «int» , «y», «)», «{» и т.п.), количество вариантов будет равно $100^{24} = 10^{48}$, что хотя на 64 порядка и меньшее, чем при представлении ИК в виде последовательности символов, однако также недостижимо высоко.

Таким образом, решение задачи реверс-инжиниринга путем прямого перебора символов или лексем для поиска ИК, компилируемого в заданный МК, является нецелесообразным даже для небольших выражений.

Однако, решение такого рода задач может оказаться возможным применением различных методов оптимизаций, например, с помощью генетических алгоритмов [25]. Основная предпосылка такого выбора заключается в схожести принципов его работы и процесса ручного восстановления ИК экспертом.

Суть генетических алгоритмов заключается в создании популяции особей, особенности которых (структура, параметры, свойства и т.п.) задаются хромосомой, состоящей их набора генов.

На первом этапе генетического алгоритма создается начальная популяция особей, гены которых могут быть заданы случайным образом. Таким образом, после этого этапа будет сгенерировано множество случайных особей.

На втором этапе происходит селекция (т.е. отбор) особей, наиболее адаптированных к окружающей среде с применением так называемой Функции приспособленности. Данная функция в численном виде определяет «живучесть» каждой особи, что позволяет отобрать наиболее удачных из них. Таким образом, после этого этапа популяция состоит из ее «лучших» (с позиции Функции приспособленности) представителей. При этом очевидно, что приспособленность особей определяется именно их генами. Если для какой-либо особи Функция приспособленности оказывается равной заданному значению (например, максимально возможному) то задача считается решенной, а алгоритм завершается.

На третьем этапе происходит скрещивание особей, заключающееся в перемешивании их ген и получении других особей (для пополнения популяции, уменьшенной на втором этапе). Таким образом, после этого этапа новые особи обладают генами от лучших представителей популяции.

На четвертом этапе происходит мутация отдельных генов, что вносит некоторый «шум» в хромосомы особи и, как следствие, их в приспособленность. Этот этап необходим для выхода из локальных экстремумов при решении оптимизационной задачи [26].

Затем, выполнение повторяется со второго этапа.

Терминология

Приведем соответствие терминов генетических алгоритмов и ассоциированных с ним понятий ГРИМК:

1) Особь – некоторый вариант текста ИК на языке программирования С, который подвергается компиляции в МК (например, «x + y;», поскольку для сокращения записи далее в примерах заголовок функции будем опускать);
2) Популяция (особей) – множество вариантов ИК, сформированных в процессе работы генетического алгоритма (например, «x + y;», «x + z;» и «y + z;»);
3) Хромосома (особи) – последовательность конструкций ИК, а именно, токенов языка программирования (например, «x», «+», «y» и «;»);

4) Ген (хромосомы) – конструкция ИК в определенной позиции, составляющая всю хромосому (например, «x» в позиции 1, «+» в позиции 2, «y» в позиции 3 и «;» в позиции 4);
5) Селекция – отбор экземпляров ИК, компилируемых в МК, наиболее близкий к исследуемому (например, если МК получен из ИК – «x+y;», то из экземпляров «x+z» и «1*2» будет селектирован первый);
6) Скрещивание – перемешивание конструкций двух экземпляров ИК с получением нового экземпляра ИК (например, в результате скрещивания родителей «x+b» и «a+y» может быть получен потомок «x+y»);
7) Мутация – случайное изменение конструкции экземпляра ИК в виде замены одного токена на другой (например, экземпляр «x-y» с некоторой вероятностью может быть мутирован в «x+y»);
8) Функция приспособленности – функция, вычисляющая близость двух МК – исследуемого и полученного компиляцией из некоторой особи или экземпляра ИК (например, если искомым ИК является «x+y», то МК для «x+z» будет считаться лучше приспособленным или близким к анализируемому, чем МК для «1*2»);
9) Размер хромосомы – длина ИК в выбранных конструкциях, т.е. количество составляющих его токенов (например, для «x + y;» это будет 4).

В качестве конструкций ИК (т.е. генов хромосомы особи) выбраны токены языка программирования C, поскольку, следуя примерам ИК разбиение его на символы является крайне нецелесообразным, а использование более сложных конструкций, таких, как деревья абстрактного синтаксиса и/или ссылки на элементы формального синтаксиса языка программирования требует дополнительных исследований.

В указанных терминах алгоритм работы ГРИМК заключается в следующем. Во-первых, создается множество случайных экземпляров ИК. Во-вторых, все экземпляры ИК компилируются в некоторые МК. В-третьих, с помощью Функции приспособленности оценивается близость их МК и исследуемого, а затем отбираются наиболее близкие к искомому ИК. Если получен ИК, в точности компилируемый в нужный МК, то задача считается решенной. В-пятых, создаются новые экземпляры ИК из токенов старых. В-шестых, некоторые токены в ИК меняются на случайные. И, в-седьмых, процесс повторяется с момента компиляции множества ИК.

Также под токенами понимаются отдельные элементы языка, такие, как ключевые слова, переменные и пр. Так, функция нахождения суммы двух чисел со следующим ИК (из 43 символов):

```
int sum (int x, int y)
{
  return x + y;
}
```

состоит из последовательности 16 токенов – «int», «sum», «(«, «int», «x», «,», «int», «y», «)», «{«, «return», «x», «+», «y», «;», «}».

Задача исследования

Исходя из идеи ГРИМК, важным вопросом остается выбор длины хромосомы – т.е. длины экземпляра ИК. Несмотря на существование генетических алгоритмов с хромосомами переменной длины [27], одним из решений данного вопроса может быть предсказание размера ИК в токенах на основании размера экземпляра МК. Измерение длины в токенах более предпочтительно, поскольку оно не учитывает длину имен переменных и функций, которые, по сути, никак не влияют на логику работы программы, а используются лишь для лучшего понимания ИК разработчиком.

Отметим, что получение размера может быть использовано и в случае кодирования хромосомы, как последовательности не только символов или токенов, но и элементов формального синтаксиса языка программирования. Так, например, если рассмотрение текста ИК ограничивается тремя идентификаторами («x», «y» и «z»), а также четырьмя основными арифметическими операциями над их парами («+», «-», «*» и «/»), то формальный синтаксис такого ИК (как совокупность синтаксических правил языка программирования) имеет следующий вид:

```
1   : expression ::=
1.1:    identifier |
1.2:   identifier operator identifier ;

2   : identifier ::=
2.1: 'x' |
2.2: 'y' |
2.3: 'z' ;

3   : operator ::=
3.1: '+' |
3.2: '-' |
3.3: '*' |
3.4: '/' ;
```

И хотя синтаксис для любого разработчика компиляторов является интуитивно понятным, тем не менее, дадим ряд пояснений. Во-первых, идентификаторы («x», «y» и «z») и операторы («+», «-», «*» и «/») являются терминальными символами, поскольку имеют конкретное значение и не могут «раскрываться» через другие символы. Во-вторых, «expression», «identifier» и «operator» являются нетерминальными символами, значения которых заранее

неизвестны, поскольку состоят из комбинации других метапеременных или символов; они определяются через операцию «::=». В-третьих, нетерминальные символы определяются как один из вариантов последовательности других терминальных и нетерминальных символов, задаваемых через операцию альтернативы «|». Префикс для каждой строки (как и ранее, до символа «:») соответствует идентификатору правила, которые как раз могут соответствовать генам особи, определяющим ИК.

Согласно синтаксису, ИК может состоять или из единичных идентификаторов (т.е. «x», «y» и «z») или из комбинаций операций между ними (т.е. «x+x» … «x/z» … «z+x» … «z/z»). Тогда, каждый ИК через хромосому переменной длины можно кодировать последовательностью правил формального синтаксиса, задаваемых соответствующими идентификаторами. Так, ИК «y» соответствует хромосоме [1.1, 2.2], «x+y» – хромосоме [1.2, 2.1, 3.1, 2.2], а «y*z» – хромосоме [1.2, 2.2, 3.3, 2.3]; здесь, «[…]» означает последовательность генов, каждый из которых определяет путь по правилам формального синтаксиса.

Необходимо отметить, что в случае приведенного выше кодирования ИК (т.е. через путь по правилам синтаксиса) задача полного перебора может решаться еще за меньшее количество вариаций (за исключением Примера 1 со сложением двух переменных – там общее число всех вариантов будет таким же).

Несмотря на некоторое количество вариантов представления хромосомы для ИК, далее будет рассмотрено получение зависимости от МК именно количество токенов, поскольку оно, с точки зрения автора, будет иметь теоретическую и практическую значимость не только в рамках ГРИМК, но и для других подобного рода задач.

Метод и прототип

Для получения зависимости размера ИК в токенах от размера МК в байтах был разработан следующий метод (далее – Метод). Его суть заключается в сборе большого количества ИК функций на языке программирования C, их компиляции в МК, вычислении размеров обоих, сборе статистики касательно соответствия этих размеров и определении итоговой зависимости. Данный язык программирования был выбран исходя из его большой популярности для разработки ПО в различных сферах, а также общей сложности проведения реверс-инжиниринга для разработанных на нем программ. Также в качестве компилятора был взят входящий в состав продукта Microsoft Visual Studio Community 2019 (далее – MSVS2019).

Необходимое множество разнообразных функций на языке C было взято из проекта ExeBench, который как раз и ориентирован на предоставление датасета в интересах машинного обучения [28]. Необходимо отметить большую и качественно проведенную работу участниками данного проекта, за что автор текущей статьи им безусловно благодарен.

Далее приведем описание шагов предлагаемого Метода.

Шаг 1. Загрузка dataset с C-функциями

Происходит загрузка структур с метаинформацией в формате JSON, содержащих функции на языке программирования C, предоставляемых в рамках проекта ExeBench.

Шаг 2. Выделение ИК C-функций

Из загруженных JSON-структур выделяется ИК C-функций с назначением им уникальных имен, которые добавляются в единое внутреннее хранилище. Данные имена используются в отладочных целях для однозначной идентификации функций.

Шаг 3. Предобработка ИК C-функций

Производится предобработка ИК C-функций следующим образом:

- удаляются ключевые слова «inline» и «__inline__» перед сигнатурой функции, поскольку в ином случае компилятором не будет сгенерирован МК (исходя из назначения ключевых слов);
- удаляется ключевое слово «static» перед сигнатурой функции, поскольку в ином случае также не будет сгенерирован МК(исходя из назначения ключевого слова);
- делается замена «NULL» на «((void *)0)», поскольку данный макрос не является встроенным в синтаксис компилятора;
- удаляются фрагменты ИК «__attribute__((...))», поскольку они не являются стандартными для языка C и не поддерживаются многими компиляторами;
- удаляются функции с ИК, содержащим ассемблерные вставки (определяемые ключевым словом «__asm__»), поскольку требуется найти зависимость только между ИК и МК;
- опционально, исключаются заданные C-функции (в том случае, если они на основании экспертного анализа ИК признаны аномальными);
- опционально, исключаются функции с ИК, содержащим работы с типами с плавающей точкой («double» и «float»);
- опционально, исключаются функции с ИК, содержащим инициализацию сложных переменных (массивов и строк) в теле функции;
- опционально, исключаются функции с ИК, содержащим работы с дробными числами (например, «10.2»).

Необходимость в последних четырех опциональных действиях шага связана с тем, что C-код в ряде

случаев генерирует аномально большой МК, что негативно повлияет на формулу прогнозирования размера ИК; для этого, такой код изначально исключается из обработки Методом.

Шаг 4. Компиляция ИК С-функций

ИК каждой функции копируется в С-файл («filename.c»), который компилируется (утилитой «cl.exe») без оптимизации (ключ «/Od») с получением только объектных файлов (ключ «/c»), содержащих МК (ключ «/Fo»); для отладочных целей генерируется и ассемблерный файл (ключ «/Fa»). Итоговой строкой компиляции является следующая: «cl.exe filename.c /c /Od /Fafilename.asm /Fofilename.obj». Зависимость между ИК и МК для других ключей компилятора (например, включающих оптимизацию по размеру или скорости) также является интересной научно-практической задачей, но будет рассмотрена в дальнейших исследованиях.

Шаг 5. Вычисление размера ИК (в токенах)

Производится разбиение текста ИК на отдельные языковые токены, измерение в которых длины текста является более корректным (или целесообразным), поскольку устраняет влияние размеров пользовательских названий (имен функций, переменных, типов), которые никак не отражаются на логике работы программы. Так, например, два ИК с текстами «void f() {}» и «void f1234567890() {}» содержат 11 и 21 символ соответственно, хотя фактически, они абсолютно идентично описывают пустую функцию; количество же токенов для этих ИК одинаково и равняется шести.

Шаг 6. Вычисление размеров МК

Производится вычисление размера кода в машинном (бинарном) представлении, для чего используется объектный файл с МК, сгенерированный в результате компиляции. Поскольку объектный файл помимо самих инструкций CPU содержит и другую информацию (что определяется заголовком файла), то его необходимо «распарсить», выделить секцию с кодом и получить ее размер.

Все полученные размеры (ИК и МК) заносятся во внутреннее хранилище. В случае ошибки, ее текст также сохраняется, а функция помечается как некомпилируемая. Затем, для каждого размера МК вычисляется минимальное, максимальное и среднее значение размера соответствующего ему ИК; дополнительно, в хранилище сохраняется количество элементов этих списков.

Шаг 7. Вывод таблицы зависимости

Производится вывод зависимости размеров МК и соответствующих им количества токенов (минимального, максимального, среднего) в ИК в табличном виде для последующей визуализации. В Методе такая таблица предназначена для загрузки в Microsoft Excel для полу-автоматического анализа.

Шаг 8. Определение формулы зависимости

Производится определение формулы зависимости между размерами ИК (в токенах) и МК. Для этого в Методе используется инструментарий, встроенный в Microsoft Excel, в части построения трендов по предопределенному закону (в настройках точечных диаграмм).

Реализация

Метод был реализован в виде программного прототипа (далее – Прототип), выполняющего все шаги, кроме 8-го. Для разработки Прототипа использовался язык программирования Python 3.10, а также следующие библиотеки: json – для работы с файлами в JSON формате (т.е. содержащих С-функции), subprocess – для запуска внешних процессов (т.е. утилиты «cl.exe»); nltk – для разделения текста ИК функции на список токенов; coff – для парсинга получаемых объектных файлов формата The Common Object File Format (сокр. COFF) и выделения в них секций с МК; signal – для перехвата нажатия «Ctrl+C» с целью пользовательского завершения работы.

Прототип в автоматическом режиме сканирует заданную директорию на предмет наличия JSON-файлов с С-функциями, позволяет загружать и обновлять внутреннее хранилище функциями из новых JSON-файлов, делает промежуточные сохранения внутреннего хранилища во внешний файл, управляет пропуском С-функций с аномальными размерами, а также выводит лог своей работы на консоль.

Эксперимент

Опишем далее эксперимент, проведенный с применением Метода и, соответственно, Прототипа.

Исходные данные и параметры

В эксперименте были взяты следующие исходные данные и параметры:

- датасет с С-функциями скачивался по Интернет-адресу https://huggingface.co/datasets/jordiae/exebench/tree/main (размер отобранных файлов составил 12.5 Гб);
- С-функции с типами с плавающей точкой, дробными числами и инициализацией сложных переменных исключались;
- экспертно было исключено 18 функций с аномально длинным ИК, который не приводил к генерации МК соизмеримого размера; функции имели следующее содержание: конкатенацию большого числа символов и текстовых строк, применение неиспользуемых макросов, создание длинных строк, оперирование сложными выражениями, длинные комментарии;
- для расчета зависимости брался МК размером не более 300 байт, поскольку для большего размера соответствующих экземпляров С-функций было

менее 10, что можно считать недостаточным для получения корректной статистике;
- количество токенов ИК считалось, как усредненное значение множества всех ИК (для определённого размера МК);
- для определения формулы зависимости размера ИК (в токенах) от размера МК использовался степенной тренд (согласно терминологии Microsoft Excel).

Также для ускорения работы Прототипа при обработке большого количества метаинформации и функций из ExeBench, все данные (как исходные, так промежуточные и конечные) располагались на виртуальном диске в памяти (размером 16 Гб).

Ход выполнения

Лог работы Прототипа при проведении Эксперимента представлен ниже; пометкой «...» отмечены строки, аналогичные предыдущей (см. схему 1).

Следуя логу, процесс работы Прототипа занял 1 часа 33 минут и 25 секунд. При этом было загружено 219077 С-функций, из которых для компиляции было подготовлено 200037 экземпляров. Из всех С-функций было успешно скомпилировано 82451 экземпляров, а 117587 привели к различным ошибкам. Таким образом, для построения зависимости было получено примерно 82.5 тысяч соотношений количества токенов в ИК и соответствующих им размеров МК.

Результаты

В результате применения Прототипа и отбора ИК с размером не более 300 токенов для построения зависимости было использовано 80787 экземпляров, что составляет 80787 / 82451 = 98.0% от их общего количества и представляет собой достаточно репрезентативную для оценки выборку.

При формировании непосредственной зависимости размера ИК от МК было учтено, что объектные файлы (с расширением «*.obj») после компиляции С-функций в MSVS2019 имели формат COFF, а их инструкции для CPU содержались в секциях со служебным названием «.text$mn». Таким образом, вычисляемый размер МК существенно отличался от размера самого объектного файла.

Полученный график зависимости размера МК от ИК (с линейным трендом) представлен на Рисунке 1; используются следующие обозначения графиков: «Мин.» – минимальное количество токенов, «Макс.» – максимальное количество токенов, «Сред.» – усредненное количество токенов, «Линейная (Сред.)» – линейный тренд для усредненного количества токенов, полученный с помощью встроенного инструментария Microsoft Excel (его формула и погрешность указаны в верхней правой части графика).

Таким образом, формула зависимости размера ИК в токенах (SCT_{Size}) от размера МК (MC_{Size}) имеет следующий усредненный вид (см. правую верхнюю часть Рисунка 1):

$$SCT_{Size} = 0.6057 \times MC_{Size} + 9.8242,$$

при этом достоверность аппроксимации составляет достаточно высокое значение – $R^2 = 0.9543$.

Схема 1

```
(2024.06.09 17:05:03) Loading dataset from 'Dataset\real_test\data_0_
time1678114487_default.jsonl' ... OK (2132 items, {'WithRealError': 2})
...
(2024.06.09 17:07:13) Loading dataset from ' Dataset\train_synth_simple_io\
data_0_time'1677914260_default.jsonl' ... OK (4786 items, {'WithRealError':
5214})
(2024.06.09 17:07:18) Dataset preparing (219077) ... OK ({'Skip with asm': 478,
'Skip with double/float type': 13569, 'Skip with array init': 2438, 'Skip by
position': 18, 'Skip with fractional number': 2536})
(2024.06.09 17:07:26) Dataset compiling (200038 items) ...
(2024.06.09 17:07:26)   0) Compile function '[data_0_time1678114487_default:0]
num2str()' ... Succeeded
...
(2024.06.09 18:38:25)     200037) Compile function '[data_0_time1677914260_
default:9993] icosd()' ... Succeeded
(2024.06.09 18:38:25)     200038) Compile function '[data_0_time1677914260_
default:9996] get_current_frame()' ... Failed
(2024.01.24 22:39:55)   Saving to 'a_dataset_3.json' ... OK
(2024.06.09 18:38:28) ... OK ({'All': 200038, 'Skipped': 0, 'Compiled': 200038,
'Succeeded': 82451, 'Failed': 117587})
```

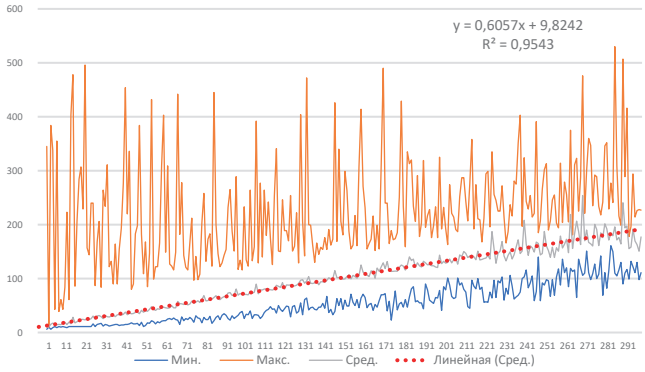

Рис. 1. Зависимость размера МК (в байтах) от размера ИК (в токенах)

Ограничения результатов

Приведем далее основные ограничения Метода и Прототипа, а также их обоснование и пути устранения.

В качестве источника ИК для С-функций выбран определенный датасет (т.е. ExeBench) по следующим причинам. Во-первых, подобных датасетов, содержащих отдельные функции, которые могут компилироваться без программного окружения (например, заголовочных файлов, включаемых с помощью препроцессорной директивы «#include») достаточно мало. Например, если взять любой крупный проект с множеством функций, то хотя он и будет компилируемым целиком, но отдельные функции будут «тянуть» за собой другие, в том числе библиотечные. Во-вторых, датасет ExeBench уже имеет достаточно хорошую структуру, поскольку состоит из JSON-файлов с ИК функциями, ассемблерным кодом под ряд компиляторов (кроме используемого в составе MSVS2019), ошибками компиляции и дополнительной метаинформацей. И, в-третьих, С-функции датасета не относятся к какой-либо определенной области ПО, а являются усредненными для программной инженерии.

Исходно, в датасете ExeBench был приведен ассемблерный код, полученный компиляторами GCC с различными оптимизациями и для ряда CPU (например, x86 и ARM), по которому можно было бы сгенерировать и соответствующие объектные файлы. Тем не менее, компилятор в составе MSVS2019 является полноценным средством получения МК [29]. При этом, теоретически, размеры ИК на одном языке программирования и МК для некоторого CPU будут слабо зависеть от выбора конкретного средства компиляции.

Как было указано, под размером ИК могут пониматься различные метрики – количество элементов текста (например, символы строки «int x = 0;»), лексических объектов (например, список токенов – TOK_INT, TOK_IDENT(«x»), TOK_ASSIGN, TOK_CONST(«0»), TOK_SEMICOLON), синтаксических конструкций (например, подграф в Abstract Syntax Tree [30] – AS_DECLARATION (AS_ASSIGN(AS_IDENT(«x»), AS_CONST(«0»))) и т.п. Однако, подсчет количества токенов ИК по сравнению с количеством символов приведет к увеличению производительности генетического алгоритма, поскольку генерация ИК из лексически верных фрагментов текста (а не случайной последовательности текста) будет с большей вероятностью приводить к компилируемому экземпляру. Использование более абстрактных сущностей (подграфов синтаксических конструкций, соответствующих формальному синтаксису языка) является возможным, но и более сложным, что и будет исследовано автором в дальнейшем.

В эксперименте при построении зависимости размера ИК от МК были обнаружены некоторые аномально высокие размеры МК в объектных файлах, что, однако, было обосновано особенностями кодирования ИК и спецификой генерируемого МК. Исходя из того, что количество таких аномалий (3 штуки) является несущественным по сравнению с общим количеством рассмотренных экземпляров (82282), а значение аномального размера является единичным и превышает средний не более чем в 10 раз (см. Рисунок 1), то и на формулу зависимости они не оказывают существенного влияния. По этой же причине, при расчете пропускался ИК, в котором использовались типы double и float, а также присутствовала динамическая инициализация массивов в теле функций.

Полученная формула зависимости размера ИК от размера МК имеет следующую достаточно простую линейную форму:

$$SC_{Size} = A \times MC_{Size} + B,$$

где A и B – некоторые коэффициенты. Впрочем, это является закономерным и вполне отражающим реальность, поскольку увеличение конструкций в ИК логично ведет к соизмеримому увеличению инструкций в МК.

Заключение

В работе приводится авторский альтернативный подход к декомпиляции МК с получением ИК, анализ которых на предмет наличия уязвимостей может существенно повысить безопасность любого ПО. Суть подхода (сокращенно ГРИМК) заключается в применении искусственного интеллекта в части генетических алгоритмов для итеративного приближения ИК к такому представлению, которое бы компилировалось в нужный МК. Одной из задач ГРИМК является определение размера исходного ИК (соответствующего длине хромосомы в терминологии генетического алгоритма), чему и посвящено данное исследование.

Основным результатом текущей работы является Метод (и Прототип), позволяющий с использованием статистических данных определить зависимость между размерами ИК и МК отдельно взятых функций (для этого используется открытый датасет – ExeBench). Также получена непосредственная формула зависимости размеров, а именно следующая:

$$SC_{Size} = 0.6057 \times MC_{Size} + 9.8242.$$

Теоретическая значимость исследования заключается в установлении прямой зависимости между размером ИК и МК для типовых функций. Практическая значимость состоит в возможности подбора параметров различных алгоритмов (включая генетические в рамках ГРИМК), которым необходимо прогнозировать размеры представлений ПО при взаимообратном преобразовании между ИК и МК.

Продолжением работы должно стать получение зависимостей между размерами ИК и МК С-функций для различных режимов работы компиляторов и инструкций CPU, уточнение формулы зависимости исходя из особенностей МК, а также выбор более сложных сущностей для конструирования ИК. Также планируется распространение описанного подхода ГРИМК и на другие, даже не смежные, области (например, для интеллектуальной адаптации графических интерфейсов под задачи пользователей [31] или обнаружения сложно-взаимодействующих уязвимостей [32, 33]).

Литература

1. *Абдуллин Т. И., Баев В.Д ., Буйневич М. В., Бурзунов Д. Д., Васильева И. Н., Галиуллина Э. Ф. и др. Цифровые технологии и проблемы информационной безопасности: монография. СПб: СПГЭУ 2021. 163 с.*
2. *Katz D. S., Ruchti J., Shulte E. Using recurrent neural networks for decompilation // The proceedings of 25th International Conference on Software Analysis, Evolution and Reengineering (Campobasso, Italy, 20–23 March 2018). 2018. PP. 346–356. DOI: 10.1109/SANER.2018.8330222.*
3. *Fokin A., Troshina K., Chernov A. Reconstruction of class hierarchies for decompilation of C++ programs // The proceedings of 14th European Conference on Software Maintenance and Reengineering (Madrid, Spain, 15–18 March 2010). 2011. PP. 240–243. DOI: 10.1109/CSMR.2010.43.*
4. *Ревнивых А. В., Велижанин А. С. Методика автоматизированного формирования структуры дизассемблированного листинга // Кибернетика и программирование. 2019. № 2. С. 1–16. 10.25136/2306-4196.2019.2.28272*
5. *Poudyal S., Dasgupta D. AI-powered ransomware detection framework // The proceedings of Symposium Series on Computational Intelligence (Canberra, ACT, Australia, 01–04 December 2020). 2021. PP. 1154–1161. DOI: 10.1109/SIKI47803.2020.9308387.*
6. *Израилов К. Е. Методология реверс-инжиниринга машинного кода. Часть 3. Динамическое исследование и документирование. Труды учебных заведений связи. 2024. Т. 10. № 1. С. 86–96. DOI: 10.31854/1813-324X-2024-10-1-86-96.*
7. *Израилов К. Е. Методология реверс-инжиниринга машинного кода. Часть 2. Статическое исследование. Труды учебных заведений связи // 2023. Т. 9. № 6. С. 68–82. DOI: 10.31854/1813-324X-2023-9-6-68-82.*
8. *Израилов К. Е. Методология реверс-инжиниринга машинного кода. Часть 1. Подготовка объекта исследования // Труды учебных заведений связи. 2023. Т. 9. № 5. С. 79–90. DOI: 10.31854/1813-324X-2023-9-5-79-90.*
9. *Kotenko I., Izrailov K., Buinevich M. The Method and Software Tool for Identification of the Machine Code Architecture in Cyberphysical Devices // Journal of Sensor and Actuator Networks. 2023. Vol. 12. Iss. 1. PP. 11. DOI: 10.3390/jsan12010011*
10. *Частикова В. А., Чич А. И. Генетические алгоритмы и генетическое программирование: особенности реализации // Перспективы науки. 2019. № 1 (112). С. 13–16.*
11. *Xia B., Ge Y., Yang R., Yin J., Pang J., Tang C. BContext2Name: naming functions in stripped binaries with multi-label learning and neural networks // The proceedings of 10th International Conference on Cyber Security and Cloud Computing (CSCloud) / 9th International Conference on Edge Computing and Scalable Cloud (Xiangtan, Hunan, China, 01-03 July 2023). 2023. PP. 167–172. DOI: 10.1109/CSCloud-EdgeCom58631.2023.00037.*
12. *A. Jaffe, J. Lacomis, Schwartz E. J., Goues C. L., Vasilescu B. Meaningful variable names for decompiled code: a machine translation approach // The proceedings of 26th International Conference on Program Comprehension (Gothenburg, Sweden, 27 May 2018 – 03 June 2018). 2020. PP. 20–2010.*
13. *Shudrak M., Zolotarev V. The new technique of decompilation and its application in information security // The proceedings of Sixth UKSim/AMSS European Symposium on Computer Modeling and Simulation (Malta, Malta, 14–16 November 2012). 2013. PP. 115–120. DOI: 10.1109/EMS.2012.20.*
14. *Alrabaee S., Choo K. -K. R., Qbea'h M., Khasawneh M. BinDeep: binary to source code matching using deep learning // The proceedings of 20th International Conference on Trust, Security and Privacy in Computing and Communications (Shenyang, China, 20–22 October 2021). 2022. PP. 1100–1107. DOI: 10.1109/TrustCom53373.2021.00150.*
15. *Katz D. S., Ruchti J., Schulte E. Using recurrent neural networks for decompilation // The proceedings of 25th International Conference on Software Analysis, Evolution and Reengineering (Campobasso, Italy, 20–23 March 2018). 2018. PP. 346–356. DOI: 10.1109/SANER.2018.8330222.*
16. *Ahmed T., Devanbu P., Sawant A. A. Learning to find usages of library functions in optimized binaries // IEEE Transactions on Software Engineering. 2021. Vol. 48. No. 10. PP. 3862–3876. DOI: 10.1109/TSE.2021.3106572.*
17. *Badri M., Badri L., Flageol W., Toure F. Source code size prediction using use case metrics: an empirical comparison with use case points // Innovations in Systems and Software Engineering. 2016. Vol. 13. PP. 143–159. DOI: 10.1007/s11334-016-0285-7.*
18. *Тютюнников Н. Н. Оценка размера программного средства с учетом адаптированного и повторно используемого исходного кода в модели COCOMO II // Фундаментальные и прикладные исследования: проблемы и результаты. 2014. № 11. С. 136–141.*

19. Частикова В. А., Чич А. И. Генетические алгоритмы и генетическое программирование: особенности реализации // Перспективы науки. 2019. № 1 (112). С. 13–16.
20. Архипов А. Н., Панов А. В. Применение кода Грея в генетическом алгоритме при кодировании признаков, представляемых целыми числами // ИТ-Стандарт. 2020. № 4 (25). С. 25–30.
21. Вавилина Е. А., Варламова С. А., Чеснов В. В. Исследование влияния изменения параметров генетического алгоритма на скорость решения задачи о рюкзаке // Информационные технологии в управлении и экономике. 2021. № 1 (22). С. 15–22.
22. Файзуллин Р. Ф. Потенциал генетических алгоритмов в задачах покрытия территории группой БЛА // Вестник РГГУ. Серия: Информатика. Информационная безопасность. Математика. 2024. № 1. С. 36–50. DOI: 10.28995/2686-679X-2024-1-36-50.
23. Kotenko, I., Izrailov, K., Buinevich, M., Saenko I., Shorey R. Modeling the Development of Energy Network Software, Taking into Account the Detection and Elimination of Vulnerabilities // Energies. 2023. Vol. 16. Iss. 13. PP. 5111. DOI: 10.3390/en16135111.
24. Kaleybar H. J., Davoodi M., Brenna M., Zaninelli D. Applications of genetic algorithm and its variants in rail vehicle systems: a bibliometric analysis and comprehensive review // Access. 2023. Vol. 11. PP. 68972–68993. DOI: 10.1109/ACCESS.2023.3292790.
25. Yu C. -Y., Huang C. -Y., Utilizing multi-objective evolutionary algorithms to optimize open source software release management // IEEE Access. 2023. Vol. 11. PP. 112248–112262. DOI: 10.1109/ACCESS.2023.3323615.
26. Jiacheng L., Lei L. A hybrid genetic algorithm based on information entropy and game theory // IEEE Access. 2020. Vol. 8. PP. 36602–36611. DOI: 10.1109/ACCESS.2020.2971060.
27. Bin Z., Zhichun G., Qiangqiang H. A genetic clustering method based on variable length string // The proceedings of 2nd International Conference on Safety Produce Informatization (Chongqing, China, 8-30 November 2019). 2020. PP. 460–464. DOI: 10.1109/IICSPI48186.2019.9095977.
28. Armengol-Estapé J., Woodruff J., Brauckmann A., Magalhães J. W. de S., O'Boyle M .F. P. ExeBench: an ML-scale dataset of executable C functions // The proceedings of 6th ACM SIGPLAN International Symposium on Machine Programming New York, NY, USA, 13 June 2022). 2022. PP. 50–59. DOI:10.1145/3520312.353486
29. Pashinska-Gadzheva M. Comparison of compiler efficiency with SSE and AVX instructions // The proceedings of International Conference Automatics and Informatics (Varna, Bulgaria, 06–08 October 2022). 2022. PP. 56–59. DOI: 10.1109/ICAI55857.2022.9960080.
30. Si G., Zhang Y., Li M., Jing S. Malicious code utilization chain detection scheme based on Abstract Syntax Tree // The proceedings of 6th Advanced Information Technology, Electronic and Automation Control Conference (Beijing, China, 03–05 October 2022). 2022. PP. 1108–1111. DOI: 10.1109/IAEAC54830.2022.9929773.
31. Курта П. А., Израилов К. Е. Обзор способов построения динамических адаптивных интерфейсов и их интеллектуализация // Научно-аналитический журнал «Вестник Санкт-Петербургского университета Государственной противопожарной службы МЧС России». 2023. № 4. С. 119–132. DOI: 10.61260/2218-130X-2024-2023-4-119-132.
32. Буйневич М. В., Израилов К. Е. Антропоморфический подход к описанию взаимодействия уязвимостей в программном коде. Часть 1. Типы взаимодействий // Защита информации. Инсайд. 2019. № 5 (89). С. 78–85.
33. Буйневич М. В., Израилов К. Е. Антропоморфический подход к описанию взаимодействия уязвимостей в программном коде. Часть 2. Метрика уязвимостей // Защита информации. Инсайд. 2019. № 6 (90). С. 61–65.

References

1. Abdullin T.I., Baev V.D., Bujnevich M.V., Burzunov D.D., Vasil'eva I.N., Galiullina Je.F. i dr. Cifrovye tehnologii i problemy informacionnoj bezopasnosti: monografija. SPb: SPGJeU 2021. 163 s.
2. Katz D. S., Ruchti J., Shulte E. Using recurrent neural networks for decompilation // The proceedings of 25th International Conference on Software Analysis, Evolution and Reengineering (Campobasso, Italy, 20–23 March 2018). 2018. PP. 346–356. DOI: 10.1109/SANER.2018.8330222.
3. Fokin A., Troshina K., Chernov A. Reconstruction of class hierarchies for decompilation of C++ programs // The proceedings of 14th European Conference on Software Maintenance and Reengineering (Madrid, Spain, 15–18 March 2010). 2011. PP. 240–243. DOI: 10.1109/CSMR.2010.43.
4. Revnivyh A. V., Velizhanin A. S. Metodika avtomatizirovannogo formirovanija struktury dizassemblirovannogo listinga // Kibernetika i programmirovanie. 2019. № 2. S. 1–16. 10.25136/2306-4196.2019.2.28272
5. Poudyal S., Dasgupta D. AI-powered ransomware detection framework // The proceedings of Symposium Series on Computational Intelligence (Canberra, ACT, Australia, 01-04 December 2020). 2021. PP. 1154–1161. DOI: 10.1109/SIKI47803.2020.9308387.
6. Izrailov K. E. Metodologija revers-inzhiniringa mashinnogo koda. Chast' 3. Dinamicheskoe issledovanie i dokumentirovanie. Trudy uchebnyh zavedenij svjazi. 2024. T. 10. № 1. S. 86–96. DOI: 10.31854/1813-324X-2024-10-1-86-96.
7. Izrailov K. E. Metodologija revers-inzhiniringa mashinnogo koda. Chast' 2. Staticheskoe issledovanie. Trudy uchebnyh zavedenij svjazi // 2023. T. 9. № 6. S. 68–82. DOI: 10.31854/1813-324X-2023-9-6-68-82.
8. Izrailov K. E. Metodologija revers-inzhiniringa mashinnogo koda. Chast' 1. Podgotovka ob#ekta issledovanija // Trudy uchebnyh zavedenij svjazi. 2023. T. 9. № 5. S. 79–90. DOI: 10.31854/1813-324X-2023-9-5-79-90.
9. Kotenko I., Izrailov K., Buinevich M. The Method and Software Tool for Identification of the Machine Code Architecture in Cyberphysical Devices // Journal of Sensor and Actuator Networks. 2023. Vol. 12. Iss. 1. PP. 11. DOI: 10.3390/jsan12010011
10. Chastikova V. A., Chich A. I. Geneticheskie algoritmy i geneticheskoe programmirovanie: osobennosti realizacii // Perspektivy nauki. 2019. № 1 (112). S. 13–16.
11. Xia B., Ge Y., Yang R., Yin J., Pang J., Tang C. BContext2Name: naming functions in stripped binaries with multi-label learning and neural networks // The proceedings of 10th International Conference on Cyber Security and Cloud Computing (CSCloud) / 9th International Conference on Edge Computing and Scalable Cloud (Xiangtan, Hunan, China, 01–03 July 2023). 2023. PP. 167–172. DOI: 10.1109/CSCloud-EdgeCom58631.2023.00037.
12. A. Jaffe, J. Lacomis, Schwartz E. J., Goues C. L., Vasilescu B. Meaningful variable names for decompiled code: a machine translation approach // The proceedings of 26th International Conference on Program Comprehension (Gothenburg, Sweden, 27 May 2018 – 03 June 2018). 2020. PP. 20–2010.
13. Shudrak M., Zolotarev V. The new technique of decompilation and its application in information security // The proceedings of Sixth UKSim/AMSS European Symposium on Computer Modeling and Simulation (Malta, Malta, 14-16 November 2012). 2013. PP. 115–120. DOI: 10.1109/EMS.2012.20.

14. *Alrabaee S., Choo K. -K. R., Qbea'h M., Khasawneh M. BinDeep: binary to source code matching using deep learning // The proceedings of 20th International Conference on Trust, Security and Privacy in Computing and Communications (Shenyang, China, 20–22 October 2021). 2022. PP. 1100–1107. DOI: 10.1109/TrustCom53373.2021.00150.*
15. *Katz D. S., Ruchti J., Schulte E. Using recurrent neural networks for decompilation // The proceedings of 25th International Conference on Software Analysis, Evolution and Reengineering (Campobasso, Italy, 20-23 March 2018). 2018. PP. 346-356. DOI: 10.1109/SANER.2018.8330222.*
16. *Ahmed T., Devanbu P., Sawant A. A. Learning to find usages of library functions in optimized binaries // IEEE Transactions on Software Engineering. 2021. Vol. 48. No. 10. PP. 3862–3876. DOI: 10.1109/TSE.2021.3106572.*
17. *Badri M., Badri L., Flageol W., Toure F. Source code size prediction using use case metrics: an empirical comparison with use case points // Innovations in Systems and Software Engineering. 2016. Vol. 13. PP. 143–159. DOI: 10.1007/s11334-016-0285-7.*
18. *Tjutjunnikov N. N. Ocenka razmera programmnogo sredstva s uchetom adaptirovannogo i povtorno ispol'zuemogo ishodnogo koda v modeli COCOMO II // Fundamental'nye i prikladnye issledovanija: problemy i rezul'taty. 2014. № 11. S. 136–141.*
19. *Chastikova V. A., Chich A. I. Geneticheskie algoritmy i geneticheskoe programmirovanie: osobennosti realizacii // Perspektivy nauki. 2019. № 1 (112). S. 13–16.*
20. *Arhipov A. N., Panov A. V. Primenenie koda Greja v geneticheskom algoritme pri kodirovanii priznakov, predstavljaemyh celymi chislami // IT-Standart. 2020. № 4 (25). S. 25–30.*
21. *Vavilina E. A., Varlamova S. A., Chesnov V. V. Issledovanie vlijanija izmenenija parametrov geneticheskogo algoritma na skorost' reshenija zadachi o rjukzake // Informacionnye tehnologii v upravlenii i jekonomike. 2021. № 1 (22). S. 15–22.*
22. *Fajzullin R. F. Potencial geneticheskih algoritmov v zadachah pokrytija territorii gruppoj BLA // Vestnik RGGU. Serija: Informatika. Informacionnaja bezopasnost'. Matematika. 2024. № 1. S. 36–50. DOI: 10.28995/2686-679X-2024-1-36-50.*
23. *Kotenko, I., Izrailov, K., Buinevich, M., Saenko I., Shorey R. Modeling the Development of Energy Network Software, Taking into Account the Detection and Elimination of Vulnerabilities // Energies. 2023. Vol. 16. Iss. 13. PP. 5111. DOI: 10.3390/en16135111.*
24. *Kaleybar H. J., Davoodi M., Brenna M., Zaninelli D. Applications of genetic algorithm and its variants in rail vehicle systems: a bibliometric analysis and comprehensive review // Access. 2023. Vol. 11. PP. 68972–68993. DOI: 10.1109/ACCESS.2023.3292790.*
25. *Yu C. -Y., Huang C. -Y., Utilizing multi-objective evolutionary algorithms to optimize open source software release management // IEEE Access. 2023. Vol. 11. PP. 112248–112262. DOI: 10.1109/ACCESS.2023.3323615.*
26. *Jiacheng L., Lei L. A hybrid genetic algorithm based on information entropy and game theory // IEEE Access. 2020. Vol. 8. PP. 36602–36611. DOI: 10.1109/ACCESS.2020.2971060.*
27. *Bin Z., Zhichun G., Qiangqiang H. A genetic clustering method based on variable length string // The proceedings of 2nd International Conference on Safety Produce Informatization (Chongqing, China, 8-30 November 2019). 2020. PP. 460–464. DOI: 10.1109/IICSPI48186.2019.9095977.*
28. *Armengol-Estapé J., Woodruff J, Brauckmann A., Magalhães J. W. de S., O'Boyle M. F. P. ExeBench: an ML-scale dataset of executable C functions // The proceedings of 6th ACM SIGPLAN International Symposium on Machine Programming New York, NY, USA, 13 June 2022). 2022. PP. 50–59. DOI:10.1145/3520312.353486*
29. *Pashinska-Gadzheva M. Comparison of compiler efficiency with SSE and AVX instructions // The proceedings of International Conference Automatics and Informatics (Varna, Bulgaria, 06–08 October 2022). 2022. PP. 56–59. DOI: 10.1109/ICAI55857.2022.9960080.*
30. *Si G., Zhang Y., Li M., Jing S. Malicious code utilization chain detection scheme based on Abstract Syntax Tree // The proceedings of 6th Advanced Information Technology, Electronic and Automation Control Conference (Beijing, China, 03-05 October 2022). 2022. PP. 1108–1111. DOI: 10.1109/IAEAC54830.2022.9929773.*
31. *Kurta P. A., Izrailov K. E. Obzor sposobov postroenija dinamicheskih adaptivnyh interfejsov i ih intellektualizacija // Nauchno-analiticheskij zhurnal «Vestnik Sankt-Peterburgskogo universiteta Gosudarstvennoj protivopozharnoj sluzhby MChS Rossii». 2023. № 4. S. 119–132. DOI: 10.61260/2218-130X-2024-2023-4-119-132.*
32. *Bujnevich M. V., Izrailov K. E. Antropomorficheskij podhod k opisaniju vzaimodejstvija ujazvimostej v programmnom kode. Chast' 1. Tipy vzaimodejstvij // Zashhita informacii. Insajd. 2019. № 5 (89). S. 78–85.*
33. *Bujnevich M. V., Izrailov K. E. Antropomorficheskij podhod k opisaniju vzaimodejstvija ujazvimostej v programmnom kode. Chast' 2. Metrika ujazvimostej // Zashhita informacii. Insajd. 2019. № 6 (90). S. 61–65.*