Научная статья
УДК 004.4
https://doi.org/10.31854/1813-324X-2025-11-4-129-142
EDN:UOKLHB

Реверс-инжиниринг программного обеспечения методом смарт-перебора: пошаговая схема

Константин Евгеньевич Израилов[1] ✉, konstantin.izrailov@mail.ru
Михаил Викторевич Буйневич[2], bmv1958@yandex.ru

[1]Санкт-Петербургский университет государственной противопожарной службы МЧС России,
Санкт-Петербург, 196105, Российская Федерация
[2]МИРЭА – Российский технологический университет,
Москва, 119454, Российская Федерация

Аннотация

Актуальность. *Наличие уязвимостей в программном обеспечении является одной из основных причин возникновения угроз безопасности информации. Противодействие уязвимостям возможно путем их непосредственного поиска в коде программы с его последующим исправлением. Для этого требуется преобразование исполняемого кода в более высокоуровневое и пригодное для поиска и исправления представление (например, в исходный код, алгоритмы, архитектуру и др.); однако существующие решения не могут считаться удовлетворительными по целому ряду причин – поддержка лишь ограниченного набора представлений, получение не тождественного выполняемому псевдокода, невозможность выбора нужной нотации представления. Одним из путей устранения указанных причин является применение полного перебора, который впрочем в случае исходного кода является сверхзатратным по всем параметрам.*
Цель исследования: *разработать менее затратный и более оперативный метод перебора вариантов исходного кода.*
Методы: *количественное и качественное сравнение различных генераторов исходных кодов; формализация метода путем его записи в аналитическом виде.*
Результаты: *предложена 7-шаговая схема метода для подбора экземпляра исходного кода по заданному машинному коду; метод является «умным» (называемый авторами смарт) с позиции оптимальности комбинаций синтаксических конструкций языка программирования. Авторский принцип генерации кода основан на переборе путей по графу синтаксических правил, являющимся представлением формального синтаксиса языка программирования в пространстве. Синтаксис передается в метод в качестве параметра, что делает его шаги полностью инвариантными от языка программирования исходного кода. После генерации множества экземпляров исходного кода производится их компиляция в машинный и сравнение с заданным; при совпадении задача декомпиляции методом умного перебора считается решенной. Предложенный метод может быть адаптирован для других (и, в частности, более высокоуровневых) представлений программы.*
Практическая значимость: *несмотря на очевидную временну́ю затратность перебора при решении такого рода задач, тем не менее, в ряде сценариев применения метод смарт-перебора показал эффективность, сравнимую с работой эксперта, и может непосредственно применяться для реверс-инжиниринга.*
Обсуждение: *существенным усовершенствованием «умного» перебора может стать его качественная оптимизация путем применения искусственного интеллекта в части генетических алгоритмов.*

Ключевые слова: *уязвимости в программном обеспечении, угроза безопасности информации, реверс-инжиниринг, декомпиляция, метод, смарт-перебор, исходный код, машинный код*

Ссылка для цитирования: Израилов К.Е. Буйневич М.В. Реверс-инжиниринг программного обеспечения методом смарт-перебора: пошаговая схема // Труды учебных заведений связи. 2025. Т. 11. № 4. С. 129–142. DOI:10.31854/1813-324X-2025-11-4-129-142. EDN:UOKLHB

Original research
https://doi.org/10.31854/1813-324X-2025-11-4-129-142
EDN:UOKLHB

Reverse Engineering of Software Using the Smart Brute Force Method: Step-by-Step Scheme

Konstantin E. Izrailov[1]✉, konstantin.izrailov@mail.ru
Mikhail V. Buinevich[2], bmv1958@yandex.ru

[1]Saint-Petersburg University of State Fire Service of EMERCOM of Russia,
St. Petersburg, 196105, Russian Federation
[2]MIREA – Russian Technological University,
Moscow, 119454, Russian Federation

Annotation

Introduction: *software vulnerabilities is one of the leading causes of threats to information security. Such vulnerabilities can be countered by directly searching for them in the program code and correcting it. This requires converting the executable code to a higher-level representation that's more suitable for searching and fixes; however, for a number of reasons, existing solutions cannot be considered satisfactory. One of these solutions – an exhaustive search of all possible variants of the source code, converted to a given machine code – is extremely costly in every way.*
Purpose: *developing a less costly and more efficient method of exhaustive searching through source code variants.*
Methods: *quantitative and qualitative comparison of different source code generators, as well as the formalization of this method by writing it in an analytical form.*
Results: *a 7-step scheme for selecting an instance of the source code according to a given machine code is proposed; the authors refer to this method as «smart» because of its optimal combinations of syntactic constructions of the programming language. This method of code generation is based on iterating through paths along the graph of syntactic rules that represent the formal syntax of a programming language in a given space. The syntax is presented as a parameter, which makes its steps completely invariant from the programming language of the source code. After multiple instances of the source code are generated, they are compiled into machine code and compared with the specified instance; if they match, the task of decompilation by smart exhaustive search is considered solved.*
Practical significance: *despite the time cost of using exhaustive searching in solving such tasks, the smart iteration method has shown expert efficiency in a number of application scenarios; thus, it can be directly applied to reverse engineering.*
Discussion: *the qualitative optimization of the "smart" exhaustive search can significantly improve it by genetic algorithms used.*

Keywords: *information security, reverse engineering, decompilation, method, smart brute-force, source code, machine code*

For citation: Izrailov K.E., Buinevich M.V. Reverse Engineering of Software Using the Smart Brute Force Method: Step-by-Step Scheme. *Proceedings of Telecommunication Universities.* 2025; 11(4):129–142. (in Russ.) DOI:10.31854/1813-324X-2025-11-4-129-142. EDN:UOKLHB

Введение

Безопасность программного обеспечения (далее – ПО) считается актуальнейшей проблемой современного мира по причине наличия в нем большого количества киберфизических систем и их информационных потоков. Нарушение безопасности ПО в основном связано с наличием в нем уязвимостей, эксплуатация которых приводит к соответствующим угрозам безопасности информации [1]. При этом сами уязвимости могут быть внедрены на различных структурных уровнях программ (в исходный код, алгоритмы, архитектуру и др.) что существенно затрудняет их поиск. Так, например, если обнаружение уязвимостей в выполняемом коде частично и решается автоматическими средствами,

то уязвимости в архитектуре на данный момент могут быть найдены лишь вручную высококвалифицированным экспертом по безопасности ПО (далее – Эксперт). Ситуация качественно усложняется тем, что, как правило, программа имеет вид машинного кода (далее – МК), который предназначен для непосредственного выполнения ЦПУ и плохо воспринимаем человеком.

Исходя из вышесказанного, можно выделить противоречие предметной области, как противопоставление потребностей Экспертов и имеющихся для этого возможностей в части научно-технического инструментария.

С одной стороны, необходимо понимание принципов и логики работы программного кода для дальнейшего экспертного (а в ряде случаев и автоматического) анализа на предмет выявления в нем уязвимостей различного структурного уровня [2]. При этом требуется дальнейшее устранение найденных уязвимостей, а не только общая оценка безопасности ПО.

С другой стороны, типичное для ПО представление программ в виде МК не позволяет производить их ручной анализ с требуемой эффективностью (т. е. обнаруживая достаточное количество уязвимостей за адекватное время и без задействования достаточно редких Экспертов). Автоматические же средства позволяют выявлять лишь ряд хорошо формализуемых уязвимостей низших структурных уровней, пропуская те, которые расположены в алгоритмах, архитектуре, и тем более в концептуальной модели программы. При этом сам по себе факт наличия уязвимости не приведет к ее устранению, а ручное исправление инструкций МК как имеет существенные технические сложности, так и потенциально может привести к общему нарушению функционирования ПО [3].

Одним из путей разрешения данного противоречия является проведение процедуры *реверс-инжиниринга* (далее – РИ) над МК [4], преобразующей его в исходный код (далее – ИК) [5]; данный процесс традиционно называется *декомпиляцией* (который существенно сложнее дизассемблирования, преобразующего МК в ассемблерный код, содержащий текстовое представление инструкций МК). Нужно отметить, что правильнее говорить про получение псевдоисходного кода – т. е. не гарантированно использованного при получении МК в процессе компиляции; впрочем, далее такое уточнение будем опускать, поскольку оно не влияет на конечное применение РИ, а именно – поиск уязвимостей. Расширенной же процедурой РИ можно считать дальнейшее преобразование программного кода для получения более высокоуровневых представлений – и не только алгоритмов и архитектуры, но даже концептуальной модели или даже идеи (которая является описанием программы, максимально адаптированной под человека). Как результат, полученные представления могут быть проанализированы Экспертами на предмет наличия уязвимостей (и что особенно важно – заложенных на различных этапах программного инжиниринга) [6].

Сам процесс РИ представляет собой достаточно сложную теоретическую и практическую задачу, подходы к решению которой претерпевали (и, видимо, будут претерпевать) качественные изменения. Так, изначально получение ИК из МК имело ручную форму и осуществлялось соответствующими специалистами – Экспертный подход. Затем появлялись автоматизированные алгоритмы восстановления отдельных конструкций ИК из МК, выделения архитектурных модулей, преобразования представления МК в более человеко-ориентированном виде (например, блок-схемы, в элементах которых все также содержались машинные инструкции) – Алгоритмический подход [7]; впрочем, участие человека осталось также необходимым. Развитие области машинного обучения позволило добавить соответствующие модели и методы и в РИ – Интеллектуальный подход [8]; однако встал вопрос корректности такого преобразования и сбора датасетов. Тем не менее, даже такая длительная эволюция РИ (имеющая даже признаки локальных революций) не принесла какого-либо достаточно эффективного решения данной задачи.

Одним из альтернативных (и достаточно специфических) подходов к решению задачи РИ, инстинктивно и потому научно-необоснованно отвергнутых практически сразу, может быть применение метода полного перебора следующим образом (что, в основном, применяется при непосредственном тестировании ПО на предмет наличия уязвимостей). Поскольку основной задачей РИ является получение ИК соответствующего заданному МК, то можно попытаться перебрать все варианты ИК (путем комбинирования его конструкций), чтобы получить тот, компиляция которого даст МК, идентичный заданному.

Приведем далее грубое качественное сравнение озвученных подходов с позиции эффективности решения задачи РИ, как совокупности трех ее показателей: результативности – точности полученного ИК по заданному МК, оперативности – времени получения ИК, ресурсоэкономности – сохранности затраченных на получение ИК ресурсов (человеческих и машинных); такое сравнение представлено в таблице 1. Одним из важных выводов их качественного сравнения является то, что ни один из подходов не является наилучшим, поскольку имеет как сильные, так и слабые стороны. Например, Эксперт всегда сможет получить МК, соответствующий ИК, но при этом он затратит огромное количество времени и должен будет обладать

большими знаниями и опытом. Применение алгоритмических средств преобразования МК в ИК частично упростит работу Эксперту, но лишь в отдельных случаях (например, при построении графов потока управления или данных [9]). Интеллектуализация (особенно в части машинного обучения, построенного на искусственных нейронных сетях), традиционно, может быть требовательной к аппаратным ресурсам, нуждаться в больших датасетах и не гарантировать корректность результатов. Проведение РИ же полным перебором хотя и гарантированно восстановит ИК, при этом без участия человека и гипотетически не затрачивая аппаратные ресурсы (поскольку последовательный перебор всех экземпляров ИК не требует существенной загрузки ЦПУ и больших объемов оперативной памяти – т. е. задача может решаться в фоновом режиме), но время проведения такой процедуры будет чрезмерно высоким (что в таблице отмечено значением оперативности, как «сверхнизкой»). Сделанный вывод можно считать общеизвестным и не требующим обсуждения.

ТАБЛИЦА 1. Качественное сравнение подходов к реверс-инжинирингу с позиции эффективности

TABLE 1. Qualitative Comparison of Reverse Engineering Approaches from an Efficiency Perspective

Подход	Результативность	Оперативность	Ресурсоэкономность
1. Экспертный	Высокая	Низкая	Низкая
2. Алгоритмический	Средняя	Высокая	Средняя
3. Интеллектуальный	Средняя	Средняя	Средняя
4. Полный перебор	Высокая	Сверхнизкая	Низкая

Второй вывод, следующий из первого, может считаться достаточно новым, который лежит как в основе текущего исследования, так и в авторском направлении – генетическом реверс-инжиниринге [10]. Исходя из того, что полный перебор по двум показателям эффективности (результативности и ресурсоэкономности) превосходит все остальные подходы, то целесообразно попытаться существенно улучшить и оставшийся третий показатель – т. е. оперативность. Еще большей эффективности полного перебора можно достичь за счет сверхнизкой ресурсоэкономности путем реализации алгоритмов подхода инвариантными к языку программирования (далее – ЯП) для создаваемого кода – например, если параметром генератора будет формальный синтаксис в форме Бэкуса – Наура (далее – БНФ) [11]. В ином случае, любая зависимость подхода от ЯП потребует как его модификацию под конкретную нотацию ИК, так и привлечение Экспертов, знакомых с этим языком (например, для настройки правил генерации).

Исходя из выше проведенного качественного сравнения подходов к РИ, а также из особенности решения полным перебором, задача текущего анализа звучит следующим образом – «Исследование возможности конструирования ИК при параметрическом задании синтаксиса ЯП для получения варианта его экземпляра, компилируемого в заданный МК». Суть задачи заключается в том, что необходимо определить, возможно ли создать генератор, который бы оптимальным (с позиции времени работы) образом создавал множество экземпляров ИК и при этом использовал бы синтаксис ЯП, как параметр; т. е. такой генератор должен строиться на синтаксически-инвариантных алгоритмах.

Обоснованность проведения указанного исследования можно подтвердить следующим образом. Во-первых, предложенный подход в принципе практически всегда отвергался, как сколько-либо подходящий для РИ, поэтому любое исследование (даже теоретическое) уже будет обладать некоторой новизной, даже если его результат окажется отрицательным. Во-вторых, качественная оптимизация полного перебора существенно повысит оперативность проведения РИ, что априори сделает подход применимым на практике в некоторых условиях. И, в-третьих, как было упомянуто выше, оптимизированный полный перебор конструкций ИК лежит в основе более сложного (по сути, качественно нового) подхода – генетического реверс-инжиниринга, описанию которого, а также практической реализации и экспериментам уже было посвящено несколько авторских статей [10, 12, 13].

Обзор релевантных работ

Произведем краткий обзор работ, в которых хотя бы частично затрагиваются предпосылки к решению задачи РИ перебором ИК.

Работа [14] описывает продукт TxTea, предназначенный для генерации тестов (известной в англоязычной литературе как Test Case) при проверке работоспособности программ, разрабатываемых на ЯП C++. В основу генератора заложено использование спецификаций ПО. В [15] для этой задачи предлагается дополнительно применять генетическое программирование, а в [16] отдельно рассматривается подзадача кроссовера при генерации тестов. В [17] с помощью генетического про-

граммирования создаются экземпляры ИК для тестирования программных систем в части их взаимодействия с аппаратной частью.

В [18] предлагается инструмент для автоматической генерации ИК по заданным блок-схемам алгоритмов. Удовлетворенность таким механизмом программного инжиниринга, согласно оценкам экспертов и рядовых пользователей (полученных в экспериментах, описанных в работе), показала высокие результаты. Альтернативный подход представлен в [19], где ИК создается по заданным шаблонам. Также для генерации кода от пользователя требуется некоторая информация об обрабатываемых данных.

Исследование [20] посвящено решению противоположной задачи по детектированию автоматически сгенерированного кода. Для этого строится модель машинного обучения для классификации ИК на искусственный и созданный человеком, обученная на соответствующих датасетах. Представлением ИК является дерево абстрактного синтаксиса (далее – ДАС, известное в англоязычной литературе, как AST – *аббр. от Abstract Syntax Tree*), а для классификации используются алгоритмы решающего дерева, случайного леса, наивного Байеса и опорных векторов.

В работе [21] описывается общий подход генетического программирования для создания ИК программ, решающих определенные задачи.

В исследовании [22] предлагается тестировать средства защиты информации путем генерации экземпляров вредоносного программного обеспечения, используя для этого спецификацию ИК, описанную на языке JSON.

Авторы в [23] приводят различные подходы проектирования программных систем, такие, как объектно-ориентированный с использованием шаблонов (что также рассматривается в исследовании [24]) и сервис-ориентированный путем интеграции модулей в рабочие процессы. Как альтернативу, авторы предлагают экспертный поход на базе соответствующих знаний и моделей, главной чертой которого является генерация ИК (которая основывается на многоуровневом наборе правил). Для этого, в частности, вводятся и используются такие понятия, как синтаксис системы (т. е. ЯП), ее прагматика (т. е. требования к ПО) и семантика (т. е. отношение между синтаксисом и прагматикой).

Работа [25] посвящена направлению автоматного программирования, заключающемуся в создании ПО на основе конечного автомата. Так, в исследовании управляющий автомат в виде графа или таблицы (т. е. входные данные) преобразуется в продукционный алгоритм в виде текста (т. е. промежуточное представление), по которому затем синтезируется программа, моделирующая входной автомат (т. е. выходные данные).

В работе [26] описываются принцип, методы и программное средство генерации тестов ПО, используя для этого информацию об его ИК. Модель самой программы задается графом потока управления, позволяющего генерировать тесты на основе трасс выполнения.

В работе [27] предлагается решение, также предназначенное для генерации фрагмента ИК, но не путем перебора, а используя аналогичный, описанный на другом ЯП – т. е. осуществление межязыковой трансляции. Синтаксис задается грамматикой YACC (*аббр. от англ*. Yet Another Compiler-Compiler, *досл. перев. на русс*. Еще Один Компилятор Компиляторов), а семантика – специализированным языком. Принцип работы генератора ИК заключается в преобразовании независимого AST в подобное ему, но отражающее синтаксис другого ЯП.

Согласно краткому обзору работ, все они посвящены решению частных задач генерации ИК – созданию задача-ориентированных программ или тестов для их валидации, а также детектированию синтезированного ИК или его преобразование в другую нотацию. Однако все эти решения не подходят для настоящей задачи исследования – конструирование ИК с получением всех его вариаций, одна из которых и будет результатом РИ для заданного МК. Исходя из этого, далее будет описан авторский метод и прототип такого генератора, отличающиеся от существующих полным покрытием всего множества возможных вариантов ИК.

Предлагаемый метод

Место в реверс-инжиниринге

Общая схема применения подхода генерации ИК в интересах РИ представлена на рисунке 1. Необходимо отметить, что такая генерация может осуществляться как полным перебором (чему собственно и посвящена данная статья), так и его оптимизированной версии (что рассматривалась авторами в других статьях и для чего было предложено применять генетические алгоритмы).

Хотя схема является интуитивно понятной и в некотором роде тривиальной, тем не менее дадим ряд пояснений касательно ее работы. Основным элементом схемы (и подхода) является генератор ИК, создающий множество их вариаций. Каждый такой ИК подвергается компиляции, получая тем самым такое же (по размеру) множество МК. Все полученные экземпляры МК сравниваются с исследуемым (т. е. тем, декомпиляцию которого необходимо произвести) с помощью компаратора. Стоит отметить, что применение отдельного компаратора (с не всегда тривиальным алгоритмом) может

потребоваться, поскольку прямое сравнение последовательности байт двух экземпляров МК может быть чрезмерно строгим, т. к. два МК по функционалу могут быть тождественны даже при небольших различиях их бинарной записи. Например, компиляция следующей функции сложения двух чисел (на языке С):

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

для процессорной архитектуры x86 может дать два идентичных по функционалу и различных по ассемблерному представлению экземпляра МК по причине коммутативности операции сложения (т. е. $x + y \equiv y + x$):

```
mov eax, DWORD PTR _x$[ebp]
add eax, DWORD PTR _y$[ebp]
```

и

```
mov eax, DWORD PTR _y$[ebp]
add eax, DWORD PTR _x$[ebp].
```

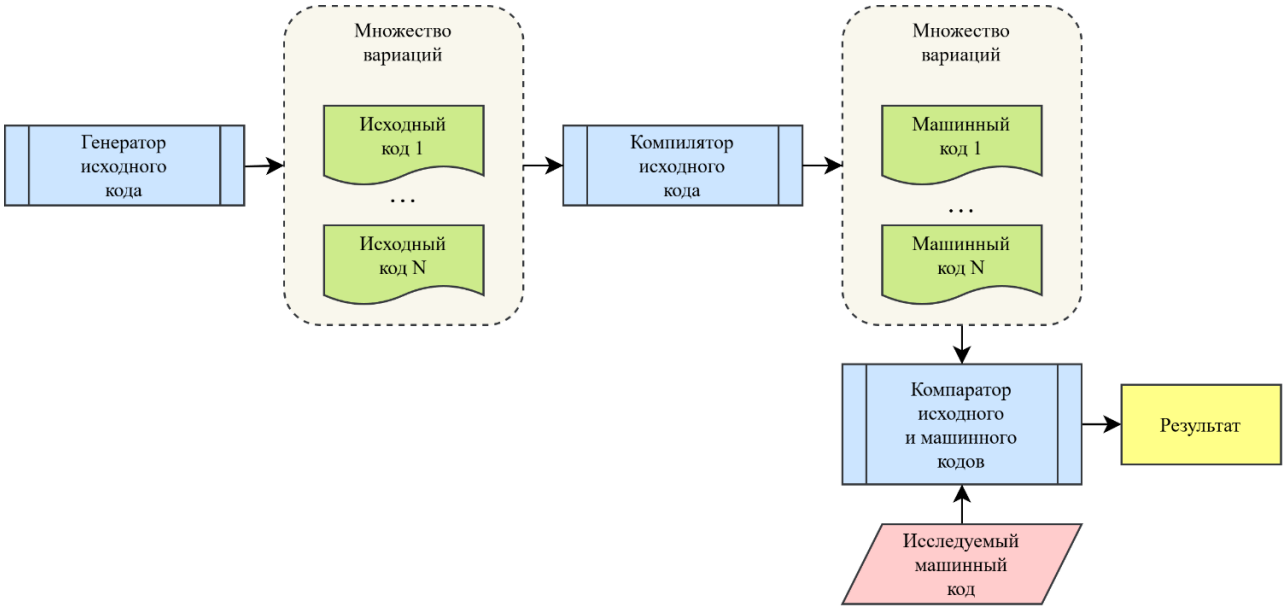

Рис. 1. Схема проведения реверс-инжиниринга путем генерации вариаций исходного кода

Fig. 1. Scheme of Reverse Engineering by Generating the Source Code Variations

Т. е. необходимо подобрать ИК среди вариантов, соответствующих синтаксису (и лексике), состоящему из односимвольных идентификаторов, математических операций и приравнивания.

Соответственно, время работы схемы РИ на базе генератора ИК будет равно произведению количества сгенерированных вариантов на время работы компилятора, даже если на вход ему подается синтаксически неверный код (работой компаратора можно пренебречь).

Генератор 1. Перебор битов символов

Самой простейшей генерацией ИК является, очевидно, перебор битов символов ИК. Так, для выбранного искомого ИК из 5 символов (каждый из которых содержит 8 бит), количество генераций равно $(2^8)^5 = 1099511627776 \approx 1{,}1 \times 10^{12}$.

Оптимизировать данный генератор за счет использования информации об синтаксисе ЯП не получится, поскольку последний не связан с битовым представлением текста программы.

Очевидно, что доля генерации заведомо некомпилируемых экземпляров ИК будет сверхвысокой, что существенно затратит ресурсы как на сам процесс генерации текста, так и на попытки его компиляции.

Генератор 2. Перебор символов ЯП

Развитием Генератора 1 путем оптимизации может быть перебор не бит, а символов, которые являются лексически верными. Если для упрощения принять, что текст ИК соответствует ЯП и может состоять из букв латинского алфавита, цифр, подчеркивания, пробела, знаков пунктуации и специальных символов (из состава «abcdefghijklmn opqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ_01 23456789,.;:?'"()[]{}<>!|/\~+#%&^*-=») общим количеством 92, то количество всех вариаций ИК из 5 символов будет равно $92^5 = 6590815232 \approx 6{,}6 \times 10^9$.

Оптимизировать данный генератор за счет использования информации об синтаксисе ЯП возможно путем выборки среди символов только необходимых – букв алфавита (52), подчеркивания (1), бинарных математических операций (4) и присваивания (1). В этом случае, количество вариаций ИК будет равно $(52 + 1 + 4 + 1)^5 = 656356768 \sim 6{,}6 \times 10^8$.

Подобно Генератору 1, доля генерации некомпилируемых экземпляров ИК будет высокой.

Генератор 3. Перебор деревьев абстрактного синтаксиса ЯП

Одним из развитий генераторов ИК, частично применяемых в генетическом программировании, является использование для создания ИК его представления в виде ДАС, которое частично учитывает правила комбинирования конструкций (таких, как наличие у бинарного оператора двух операндов, отсутствие следования подряд двух идентификаторов переменных и т. п.). ДАС для выбранного искомого ИК имеет форму, представленную на рисунке 2.

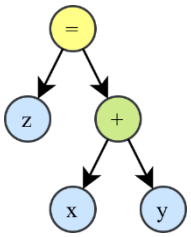

Рис. 2. Дерево абстрактного синтаксиса для примера исходного кода «*z = x + y*»

Fig. 2. Abstract Syntax Tree for the Initial Code Example «z = x + y»

В данном дереве как раз и отражен ряд правил комбинирования конструкций ИК, таких, как следующие: первой конструкцией является операция; операторы «=» и «+» имеют по 2 операнда; явно не показано, но подразумевается, что операнды «+» могут быть выражениями.

При таком конструировании общее количество ИК из 5 символов в форме «Идентификатор Оператор Идентификатор Оператор Идентификатор» (т. е. той, которая соответствует искомому ИК) можно вычислить с помощью комбинаторики. Так, ИК состоит из следующего количества элементов: бинарных математических операторов для языка программирования C (т. е. «+-*/») – 4, односимвольных идентификаторов (т. е. «*a*…*z*», «*A*…*Z*» и «0_») – 53, оператора присваивания («=») – 1. Тогда общее число ДАС будет равно (52 + 1) × (4 + 1) × × (52 + 1) × (4 + 1) × (52 + 1) = 3721925 ≈ 3,8 × 10^6. Доля генерации некомпилируемых экземпляров ИК хотя и будет средней, однако она все также существенно скажется на низкой оперативности подхода; при этом при слишком жестких ограничениях на шаблон ДАС часть вариантов ИК не будут сгенерированы в принципе (например, унарные операции). Информация о синтаксисе ЯП в генераторе уже используется и, следовательно, этим путем его соптимизировать не удастся.

Необходимо отметить, что если два первых генератора не зависели от синтаксиса ЯП, то в данном учитывается логика построения математических выражений, что автоматически снижает его применимость для всего многообразия ЯП.

Генератор 4. Перебор веток синтаксиса ЯП

Последний генератор в приведенном пути эволюции может быть получен исправлением недостатков предыдущего – третьего, а именно следующих: снижение до нуля создания синтаксически неверных ИК, генерация всех возможных вариаций ИК и инвариантность алгоритмов к конкретному ЯП. Все это можно достичь с помощью авторской идеи – генерации ИК по формальному синтаксису ЯП, который является лишь параметром для обобщенных алгоритмов генератора. Таким образом, будет осуществлен «синтаксический перебор» ИК, а не «лексический», который применялся ранее – полностью (в Генераторах 1 и 2) и частично (в Генераторе 3).

Синтаксис, описываемый ИК, подобный искомому, приведен на Листинге в нотации Бэкуса–Наура (префикс с «:» на конце указывает номер строки с отдельным правилом, а «…» – сокращенные для компактности символы):

```
1: expr_asgn ::= ident, '=', expr_oper ;
2: expr_oper ::= ident, oper, ident ;
3: oper ::= '+' | '-' | '*' | '/' ;
4: ident ::= 'a' | ... | 'z' | 'A' ... 'Z' | '_' .
```

В синтаксисе Листинга используются следующие элементы:

– терминалы (зеленого цвета): бинарные математические операции, символы латинского алфавита, подчеркивание и приравнивание (например, «+», «x» и «=»);

– нетерминалы (синего цвета): expr_asgn для выражения присваивания, expr_oper для математического выражения, ident для односимвольного идентифиватора (например, в строке 2 «exor_oper» задается через последовательность двух идентификаторов «ident» и операции между ними «oper»);

– «::=» (черного цвета): для определения правила раскрытия нетерминала через другие нетерминалы и терминалы (например, в строке 3 «oper» задается через математические операции);

– «|» (черного цвета): для указания нескольких последовательностей, через которые может определяться нетерминал (например, в строке 4 «ident» задается как один из символов алфавита или подчеркивание).

Очевидно, что главным нетерминалом, с которого происходит сопоставление ИК к ЯП, является заданный в 1-й строке – «expr_asgn». Такого рода формальные записи синтаксиса ЯП (часто в расширенных формах), как правило, применяются при построении современных компиляторов.

Произведем расчет количества экземпляров ИК, которые могут быть сгенерированы путем полного перебора по данному синтаксису. Синтаксис задает

строгую структуру ИК, состоящую из идентификатора, за котором всегда следует знак «=» и последовательность из идентификатора, математической операции и еще одного идентификатора. Таким образом, общее число комбинаций равно (52 + 1) × 1 × (52 + 1) × 4 × (52 + 1) = 595508 ≈ 6 × 10^5; данное значение можно считать *истинным количеством экземпляров ИК*, среди которых искался тот, который соответствует заданному МК (т. е. пределом оптимизации генераторов в текущих условиях подбора). Еще раз подчеркнем, что важной особенностью данного генератора является перебор всех возможных ИК без создания заведомо некорректных и полная (при определенной реализации) независимость алгоритмов от синтаксиса ЯП. Именно данный генератор положен в основу дальнейшего развития исследования.

Подведем итог оценки всех четырех генераторов (определяемых индексом $i \in [1..4]$), указав для них общее количество генерируемых экземпляров ИК ($Total_i$) и долю излишне сгенерированных ИК относительно истинного количества ($Error_i$ в процентах):

– генератор 1: $Total_1 \approx 1{,}1 \times 10^{12}$ и $Error_1 \approx 99{,}9999$;

– генератор 2: $Total_2 \approx 6{,}6 \times 10^9$ и $Error_2 \approx 99{,}991$ для неоптимизированной версии, $Total_2 \approx 6{,}6 \times 10^8$ и $Error_2 \approx 99{,}9093$ – для оптимизированной;

– генератор 3: $Total_3 \approx 2{,}4 \times 10^6$ и $Error_3 \approx 84$;

– генератор 4: $Total_4 \approx 6 \times 10^5$ и $Error_4 \approx 0$.

Следуя проведенной оценке генераторов, можно сделать вывод, что первые два практически не применимы для подбора ИК в рамках РИ из-за огромного количества излишне генерируемых экземпляров. Генератор 3 показывает лучшие результаты, хотя и создает подходящим лишь каждый 6-й экземпляр; при этом, как указывалось, в основу его алгоритмов заложен конкретный синтаксис ЯП. Последний же генератор может считаться эталонным, что, впрочем, следует из принципа его условно идеальной (с позиции перебора конструкция ЯП) работы.

Схема метода

В основу метода работы (далее – Метод) предлагаемого «умного» генератора ИК (далее – Генератор) заложен обход направленного графа синтаксических правил (далее – ГСП), описывающего все возможные пути разбора (а, следовательно, и генерации) выходных текстов ИК. Использование именно графа (вместо дерева) обусловлено тем, что один и тот же нетерминал может использоваться в нескольких правилах, для чего потребуются переходы с одной вершины графа на другую, расположенную в параллельной ветке. Сам обход ГСП осуществляется в глубину (т. е. от самой верхней вершины по всем «детям», пока каждый из них, включая его «детей», не будет пройден) с тем отличием от классических обходов, что одна и та же вершина может быть пройдена несколько раз – это необходимо для генерации ИК с многократно повторяющимися конструкциями (например, «$y = x + (x + (\ldots x)..)$»), заданными в синтаксисе рекурсивным способом.

Пошаговая схема Метода с добавленным функционалом для проведения РИ представлена на рисунке 3.

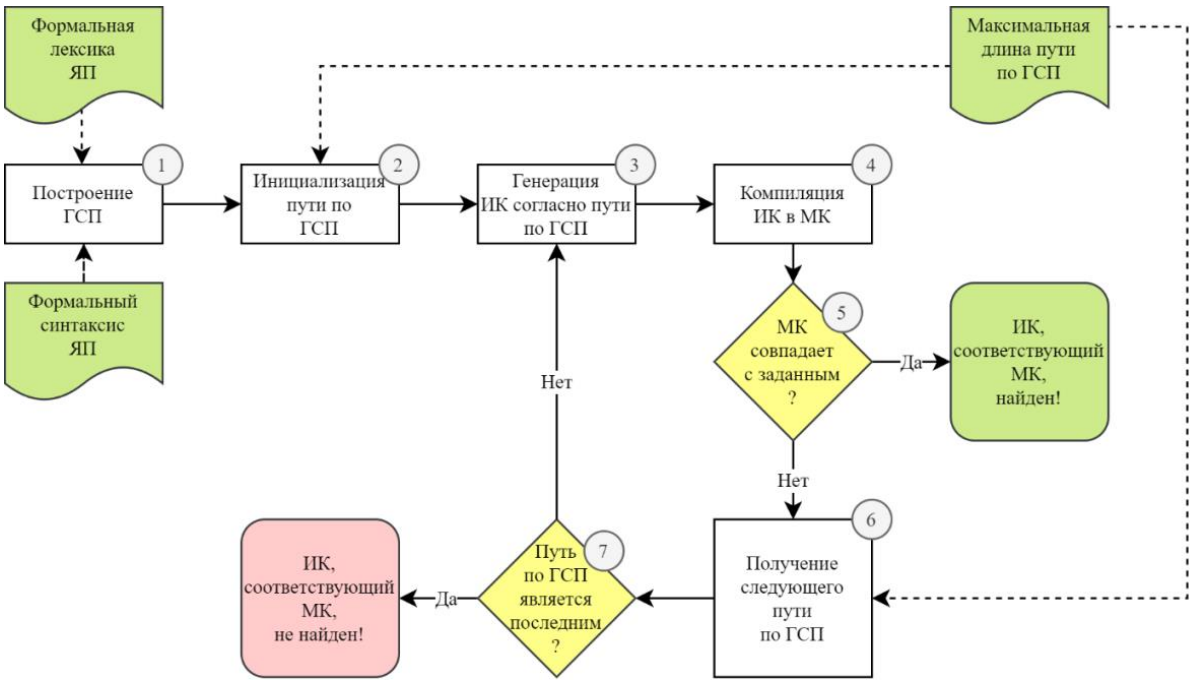

Рис. 3. Схема работы Метода с функционалом для реверс-инжиниринга

Fig. 3. Scheme of the Method Operation with Functionality for Reverse Engineering

Опишем приведенную на рисунке 3 схему Метода, как последовательность шагов (номера которых указан в окружности в правой верхней части фигур).

Шаг 1. Построение ГСП

На первом шаге происходит построение ГСП, согласно которому будет генерироваться ИК. Поскольку алгоритмы генерации должны быть инвариантными к синтаксису, то последний задан в формализованном виде и передается в Генератор через параметр. Также, поскольку синтаксис оперирует токенами ЯП (базовыми единицами синтаксиса), необходимо задать их текстовое написание через формальную лексику (также, как параметр Генератора); впрочем, синтаксис и лексика могут описываться совместно в одном представлении. Действия шага по построению ГСП (*аббр*. SRG *от англ*. Syntax Rule Graph) в формальном виде могут быть записаны следующим образом:

$$SRG = BuildSRG(LanguageSyntax, LanguageLexica),$$

где *BuildSRG*(…) – операция построения ГСП; *LanguageSyntax* – формальный синтаксис ЯП; *LanguageLexica* – формальная лексика ЯП.

ГСП для примера формального синтаксиса из Листинга приведен на рисунке 4; использованы следующие цвета фонов фигур: серый – последовательность терминалов и нетерминалов в рамках одного правила, зеленый – терминал, синий – основной нетерминал, желтый – ссылочный нетерминал, определенный в другом месте графа; также, сплошная стрелка задает правила, а пунктирная – ссылки между нетерминалами.

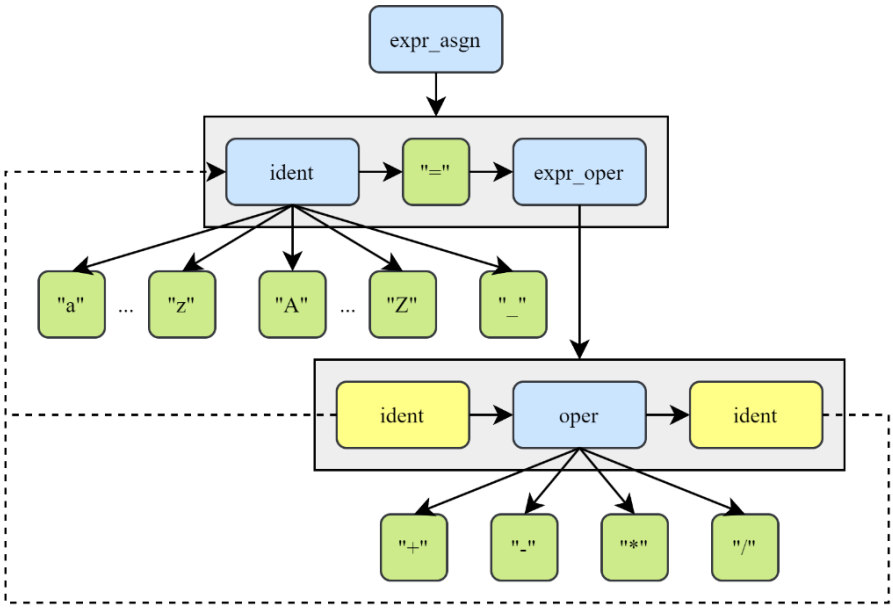

Рис. 4. Представление графа синтаксических правил для Листинга

Fig. 4. Representation of the Syntax Rules Graph for Listing

Отметим, что пример ГСП на рисунке 4 выглядит не как классический граф (с множеством вершин и ребер между ними), а представляет более сложную структуру, где часть вершин сами являются графами («ident» → "=" → expt_oper» и «ident → oper → ident»); это сделано лишь для упрощения понимания сути ГСП, что в реальном графе может быть реализовано через служебные узлы.

На рисунке 4 дадим ряд пояснений касательно соответствия представления ГСП Листингу.

Нетерминал «expr_asgn» имеет одного «ребенка», представляющего последовательность элементов синтаксиса – нетерминала «ident», терминала приравнивания и нетерминала «expr_oper»; таким образом, выражение присваивания может быть записано, как идентификатор, за которым идет символ «=» и математическое выражение.

Нетерминал «ident» имеет 53 «ребенка», соответствующих терминалам-символам латинского алфавита и подчеркиванию; таким образом, идентификатор может быть записан как один из этих символов. Нетерминал «expr_oper» имеет одного «ребенка», представляющего последовательность элементов синтаксиса – нетерминала «ident», нетерминала операции и еще одного нетерминала «ident»; таким образом, математическое выражение может быть записано, как два идентификатора (тождественных такому же нетерминалу, но определенному в параллельной ветке ГСП), между которыми стоит бинарная операция.

Нетерминал «oper» имеет 4 «ребенка», соответствующих терминалам-символам бинарных математических операций; таким образом, операция может быть записана как один из этих символов. Следуя интерпретации примера ГСП, можно сделать вывод, что в нем присутствуют четыре вершины (3 из которых соответствуют одному нетерминалу), в которых есть альтернатива движения по синтаксису – «ident» и «oper»; остальные ветки описывают однозначные пути продвижение по вершинам графа.

Шаг 2. Инициализация пути по ГСП

На втором шаге происходит первоначальная инициализация пути для обхода всех ветвей ГСП. Само представление пути для оптимизации записи, в Генераторе задается, как выбор каждой альтернативной ветки с помощью ее локального порядкового номера (далее, для упрощения, каждую такую ветку в ГСП будем называть альтернативой).

Например, дойдя при обходе до «oper», дальнейшей вершиной может быть одна из четырех (согласно четырем выходящим ребрам, ведущим к терминалам «+», «-», «*» и «/»), что в пути будет задаваться числом от 1 до 4. Соответственно, отсутствие альтернативных ребер (как для «expr_asgn», «expr_oper») в пути никак обозначать не надо; формально для таких случаев будет только один выбор ребра с индексом 1. Также, исходя из цикличности практически всех нетривиальных ГСП, необходимо ограничить глубину обхода, чтобы обеспечить теоретическую и практическую завершимость работы метода с генерацией ИК определенной (в смысле не бесконечной) длины; для этого одним из параметров Генератора является максимальная длина пути (далее – МДП).

Действия шага по выбору первоначального пути (*Path* с порядковым номером 1) по ГСП в формальном виде могут быть записаны следующим образом:

$$Path_1 = InitPath(SRG, MaxDeep),$$

где *InitPath*(…) – операция получения 1-го пути по ГСП; *MaxDeep* – МДП по ГСП.

Для Листинга и, соответственно, рисунка 4 начальный путь имеет следующую запись:

$$Path_1 = [1, 1, 1, 1],$$

где «[…]» – последовательность выбора альтернатив; четыре значения «1» – выбор пути по первому альтернативному ребру для каждого имеющего их родительского узла (т. е. согласно обходу ГСП в глубину при посещении последовательно нетерминалов «ident», «ident», «oper» и «ident»).

Шаг 3. Генерация ИК согласно пути по ГСП

На третьем шаге по текущему пути генерируется ИК путем обхода ГСП с выбором соответствующих альтернатив.

Согласно рисунку 4, посещение вершин будет следующим:

1) «expr_asgn»;
2) «ident» с последующим выбором альтернативы согласно 1-му элементу пути (от 1 до 53);
3) один из символов алфавита или подчеркивание;
4) «=»;
5) «expr_oper»;
6) аналогично 2), но с выбором альтернативы согласно 2-му элементу пути;
7) один из символов алфавита или подчеркивание;
8) «oper» с последующим выбором альтернативы согласно 3-му элементу пути (от 1 до 4);
9) один из символов математической операции;
10) аналогично 2) и 6), но выбором альтернативы согласно 4-му элементу пути;
11) один из символов алфавита или подчеркивание.

Действия шага по генерации ИК (*SourceCode*) согласно пути по ГСП в формальном виде могут быть записаны следующим образом:

$$SourceCode_{Path} = GenerateSourceCode(Path, SGR),$$

где *GenerateSourceCode*(…) – операция генерации ИК по пути.

Очевидно, что для первого выбранного пути [1, 1, 1, 1] будет сгенерирован следующий ИК (*SourceCode*):

$$SourceCode_{[1,1,1,1]} = "a = a + a".$$

Шаг 4. Компиляция ИК в МК

На четвертом шаге производится получение МК (*MachineCode*), соответствующего сгенерированному ИК (*SourceCode*) согласно текущему пути (*Path*), путем операции компиляции (*Compile*):

$$MachineCode_{Path} = Compile(SourceCode_{Path});$$

для чего можно использовать стандартные утилиты компиляции (Microsoft Visual C++, GNU Compiler Collection, Intel C++ compiler и др.).

Шаг 5. Проверка «МК совпадает с заданным?»

Пятый шаг содержит проверку, необходимую для успешного завершения работы метода. Так, если скомпилированный МК совпадает с тем, РИ которого необходимо произвести (*MachineCode*), то, следовательно, задача восстановления ИК завершена успешно:

$$MachineCode_{Path} = \underline{MachineCode} \Longrightarrow Result = Result_{Successful},$$

где *Result* – результат РИ; $Result_{Successful}$ – факт успешности РИ.

Шаг 6. Получение следующего пути по ГСП

На шестом шаге осуществляется получение следующего пути из текущего по ГСП с помощью последовательного перебора всех альтернатив. Действия шага по выбору нового по ГСП (*GetNextPath*) в

формальном виде могут быть записаны следующим образом:

$$Path_{(i+1)} = GetNextPath(Path_i, SGR, MaxDeep),$$

где $Path_{(i+1)}$ – следующий i+1-й путь; $Path_i$ – текущий i-й путь. Аналогично Шагу 2, для ограничения глубины обхода, как и на Шаге 2 используется параметр МДП (*MaxDeep*).

Если весь ГСП обойден и следующий путь отсутствует, то он примет специальное значение из пустой последовательности [ ]:

$$Path_{(N+1)} = [\,],$$

где N – число всех возможных путей по ГСП; N+1 – путь, следующий за последним (т. е. несуществующий).

Важно отметить, что изменение одного элемента пути может повлиять не только на значение последующих элементов, но и на их количество (т. е. длину пути), поскольку все подграфы ГСП могут иметь различную глубину обхода – от 1 до бесконечности (в случае циклов).

Шаг 7. Проверка «Путь по ГСП является последним?»

Седьмой шаг содержит проверку, необходимую неуспешного завершения работы метода. Так, если путь по ГСП является последним (т. е. пройдены все возможные пути), и при этом нужного МК и ИК не найдено, то, следовательно, задача восстановления ИК не смогла быть решена:

$$Path = [\,] \Longrightarrow Result = Result_{Unsuccessful},$$

где [ ] – пустая последовательность, обозначающая отсутствие пути (возвращается *GetNextPath*); $Result_{Unsuccessful}$ – факт неуспешности РИ.

Анализ формальных записей всех шагов Метода (что таким образом определяет аналитическую модель, лежащую в основе его работы) позволяет утверждать об его инвариантности к синтаксису ЯП искомого ИК и архитектуры ЦПУ исследуемого МК; причины такого вывода заключатся в том, что синтаксис в формальном виде задается параметром, а архитектура никак не учитывается (за счет применения внешней утилиты компиляции).

Заключение

В интересах решения задачи реверс-инжиниринга программного обеспечения были рассмотрены различные классические (экспертный, алгоритмический, интеллектуальный) и авторский (полного перебора) подходы, качественное сравнение которых с позиции эффективности не позволило выделить какой-либо, абсолютно превосходящий остальные. Как результат, было предложено развитие подхода полного перебора, а также оптимизация входящего в него генератора исходного кода, что позволило получить метод «умного» перебора, являющийся основным научным результатом текущего исследования. Данный метод принимает на вход формальный синтаксис ЯП и МДП по его графу, производит перебор всех веток ГСП, генерацию ИК, компиляцию его в МК и сравнение с исследуемым (т. е. тем, РИ которого необходимо произвести).

Оригинальность Метода состоит в применении для РИ МК в ИК нового подхода, который хотя и был известен, но не применяется по причине высоких временных затрат [29]. Так, хотя ближайшей областью к решаемой задаче является генетическое программирование, его назначение и, следовательно, алгоритмы, предназначены для других целей. Также отличительными особенностями Метода являются следующие: инвариантность к синтаксису ЯП искомого ИК и архитектуре ЦПУ исследуемого МК (в отличие от этого, популярный продукт для РИ IDA Pro [30] на момент июля 2024 года поддерживает лишь 5 типов ЦПУ – x86, x64, ARM32, ARM64 и PowerPC); полная автоматизация, позволяющая его использовать не «штучными» высококвалифицированными Экспертами по ИК и МК, а операторами, владеющими лишь основами РИ (современные автоматические декомпиляторы позволяют получать как правило только С-подобный псевдо-код, а их результаты требуют проверки Экспертом на корректность).

Теоретическая значимость работы заключается в развитии подхода к реверс-инжинирингу, принцип которого диаметрально противоположен существующим, являясь при этом инвариантным к синтаксисам входных и выходных представлений кода. Практическая значимость работы состоит в получении теоретической и алгоритмической базы для создания программных решений, позволяющих проверить работоспособность метода на реальных примерах.

Продолжением исследования будет реализация метода в виде программного прототипа, проверка его работоспособности, а также, проведение ряда экспериментов с цель общей оценки эффективности и выявления сильных и слабых сторон подхода.

Список источников

1. Tan T.-T., Wang B.-S., Tang Y., Zhou X. Crash Analysis Mechanisms in Vulnerability Mining Research // Proceedings of the 4th International Conference on Computer and Communication Systems (Singapore, Singapore, 23–25 February 2019). IEEE, 2019. PP. 355–359. DOI:10.1109/CCOMS.2019.8821775

2. Chondamrongkul N., Sun J., Warren I. Automated Security Analysis for Microservice Architecture // Proceedings of the *International Conference on Software Architecture Companion (Salvador*, Brazil, 16–20 March 2020). IEEE, 2020. PP. 79–82. DOI:10.1109/ICSA-C50368.2020.00024

3. Iannone E., Guadagni R., Ferrucci F., De Lucia A., Palomba F. The Secret Life of Software Vulnerabilities: a Large-Scale Empirical Study // IEEE Transactions on Software Engineering. 2023. Vol. 49. Iss. 1. PP. 44–63. DOI:10.1109/TSE.2022.3140868. EDN:GKKIKO

4. Fu J., Zhang K., Zheng J., Li W., Zhu Y. Research and Application of Grey Box Detection Technology Based on Reverse Engineering and Dynamic Pollution Diffusion // Proceedings of the 7th Information Technology and Mechatronics Engineering Conference (Chongqing, China, 15–17 September 2023). IEEE, 2023. PP. 2380–2384. DOI:10.1109/ITOEC57671.2023.10291380

5. Devine T.R., Campbell M., Anderson M., Dzielski D. SREP+SAST: A Comparison of Tools for Reverse Engineering Machine Code to Detect Cybersecurity Vulnerabilities in Binary Executables // Proceedings of the International Conference on Computational Science and Computational Intelligence (Las Vegas, USA, 14–16 December 2022). IEEE, 2022. PP. 862–869. DOI:10.1109/CSCI58124.2022.00156

6. Bhardwaj V., Kukreja V., Sharma C., Kansal I., Popali R. Reverse Engineering-A Method for Analyzing Malicious Code Behavior // Proceedings of the International Conference on Advances in Computing, Communication, and Control (Mumbai, India, 03–04 December 2021). IEEE, 2021. PP. 1–5. DOI:10.1109/ICAC353642.2021.9697150

7. Израилов К.Е., Покусов В.В. Архитектура программной платформы преобразования машинного кода в высокоуровневое представление для экспертного поиска уязвимостей // Электронный сетевой политематический журнал «Научные труды КубГТУ». 2021. № 6. С. 93–111. EDN:AIOUWF

8. Буйневич М.В., Израилов К.Е., Покусов В.В., Тайлаков В.А., Федулина И.Н. Интеллектуальный метод алгоритмизации машинного кода в интересах поиска в нем уязвимостей // Защита информации. Инсайд. 2020. № 5(95). С. 57–63. EDN:HIHDOM

9. Cummins C., Fisches Z.V., Ben-Nun T., Hoefler T., O'Boyle M.F.P., Leather H. ProGraML: A Graph-based Program Representation for Data Flow Analysis and Compiler Optimizations // Proceedings of the 38th International Conference on Machine Learning (PMLR, 18–24 July 2021). 2021. Vol. 139. PP. 2244–2253.

10. Израилов К.Е. Концепция генетической декомпиляции машинного кода телекоммуникационных устройств // Труды учебных заведений связи. 2021. Т. 7. № 4. С. 95–109. DOI:10.31854/1813-324X-2021-7-4-95-109. EDN:AIOFPM

11. Tonis R.B.M. Automating Scientific Paper Screening with Backus-Naur Form (BNF) Grammars // Didactica danubiensis. 2024. Vol. 4. Iss. 1. PP. 46–57.

12. Израилов К.Е. Концепция генетической деэволюции представлений программы. Часть 1 // Вопросы кибербезопасности. 2024. № 1(59). С. 61–66. DOI:10.21681/2311-3456-2024-1-61-66. EDN:CBCKRF

13. Израилов К.Е. Концепция генетической деэволюции представлений программы. Часть 2 // Вопросы кибербезопасности. 2024. № 2(60). С. 81–86. DOI:10.21681/2311-3456-2024-2-81-86. EDN:JUBPML

14. Hamberger P., Klammer C., Luger T., Moser M., Pfeiffer M., Piereder C. Specification-Based Test Case Generation for C++ Engineering Software // Proceedings of the International Conference on Software Maintenance and Evolution (ICSME, Bogotá, Colombia, 01–06 October 2023). IEEE, 2023. PP. 519–529. DOI:10.1109/ICSME58846.2023.00066

15. Sato Y. Specification-Based Test Case Generation with Constrained Genetic Programming // Proceedings of the 20th International Conference on Software Quality, Reliability and Security Companion (QRS-C, Macau, China, 11–14 December 2020). IEEE, 2020. PP. 98–103. DOI:10.1109/QRS-C51114.2020.00027

16. Huang C., Zhou H., Zhao H., Cai W., Zhou Z.Q., Jiang M. On the Usefulness of Crossover in Search-Based Test Case Generation: An Industrial Report // Proceedings of the 29th Asia-Pacific Software Engineering Conference (APSEC, Japan, 06–09 December 2022). IEEE, 2022. PP. 417–421. DOI:10.1109/APSEC57359.2022.00054

17. Schwachhofer D., Angione F., Becker S., Wagner S., Sauer M., Bernardi P., Polian I. Optimizing System-Level Test Program Generation via Genetic Programming // Proceedings of the European Test Symposium (ETS, The Hague, Netherlands, 20–24 May 2024). IEEE, 2024. PP. 1–4. DOI:10.1109/ETS61313.2024.10567817

18. Supaartagorn C. Web application for automatic code generator using a structured flowchart // Proceedings of the International Conference on Software Engineering and Service Science (ICSESS, Beijing, China, 24–26 November 2017). IEEE, 2017. PP. 114–117. DOI:10.1109/ICSESS.2017.8342876

19. Shinde K., Sun Y. Template-Based Code Generation Framework for Data-Driven Software Development // Proceedings of the 4th Intl Conf on Applied Computing and Information Technology / 3rd Intl Conf on Computational Science / Intelli-gence and Applied Informatics / 1st Intl Conf on Big Data, Cloud Computing, Data Science & Engineering (ACIT-CSII-BCD, Las Vegas, USA, 12–14 December 2016). IEEE, 2016. PP. 55–60. DOI:10.1109/ACIT-CSII-BCD.2016.023

20. Shimonaka K., Sumi S., Higo Y., Kusumoto S. Identifying Auto-Generated Code by Using Machine Learning Techniques // Proceedings of the 7th International Workshop on Empirical Software Engineering in Practice (IWESEP, Osaka, Japan, 13 March 2016). IEEE, 2016. PP. 18–23. DOI:10.1109/IWESEP.2016.18

21. Igwe K., Pillay N. Automatic programming using genetic programming // Proceedings of the Third World Congress on Information and Communication Technologies (WICT 2013, Hanoi, Vietnam, 15–18 December 2013). IEEE, 2013. PP. 337–342. DOI:10.1109/WICT.2013.7113158

22. Бирюков Д.Н., Дудкин А.С., Захаров О.О. Способ тестирования средств защиты информации на основе применения многовариантной генерации исходного кода по заданной функциональной спецификации // Труды Военно-космической академии имени А.Ф. Можайского. 2022. № 684. С. 113–122. EDN:BJWKLG

23. Самохвалов Э.Н., Ревунков Г.И., Гапанюк Ю.Е. Генерация исходного кода программного обеспечения на основе многоуровневого набора правил // Вестник Московского государственного технического университета им. Н.Э. Баумана. Серия Приборостроение. 2014. № 5(98). С. 77–87. EDN:SVZLSL

24. Соколов А.П., Макаренков В.М., Першин А.Ю., Лаишевский И.А. Разработка программного обеспечения генерации кода на основе шаблонов при создании систем инженерного анализа // Программная инженерия. 2019. Т. 10. № 9-10. С. 400–416. DOI:10.17587/prin.10.400-416. EDN:CHYPRE

25. Довгаль В.М., Корольков О.Ф., Чаплыгин А.А., Королькова В.О. К вопросу решения проблемы автоматической генерации кода программ по заданному управляющему продукционному алгоритму // В мире научных открытий. 2012. № 1(25). С. 220–235. EDN:PBBWKP

26. Андрианова А.А., Ицыксон В.М. Технология анализа исходного кода программного обеспечения и частичных спецификаций для автоматизированной генерации тестов // Системы и средства информатики. 2014. Т. 24. № 2. С. 99–113. DOI:10.14357/08696527140207. EDN:SJHATL

27. Саух А.М., Хмельнов А.Е. Трансляция фрагментов исходных текстов программ с использованием спецификаций синтаксиса и семантики языков программирования // Вестник Новосибирского государственного университета. Серия: Информационные технологии. 2013. Т. 11. № 3. С. 53–62. EDN:RCHBLB

28. Haq I.U., Caballero J.A. Survey of Binary Code Similarity // ACM Computing Surveys. 2021. Vol. 54. Iss. 3. PP. 1–38. DOI:10.1145/3446371. EDN:KEPQCC

29. Куделя В.Н. Методы перечисления путей в графе // Наукоемкие технологии в космических исследованиях Земли. 2023. Т. 15. № 5. С. 28–38. DOI:10.36724/2409-5419-2023-15-5-28-38. EDN:HQEASN

30. Кусаинов А.Р., Глазырина Н.С. Обзор инструментов статического анализа программного кода // Colloquium-Journal. 2020. № 32-1(84). С. 48–52. EDN:JXSKQX

References

1. Tan T.-T., Wang B.-S., Tang Y., Zhou X. Crash Analysis Mechanisms in Vulnerability Mining Research. *Proceedings of the 4th International Conference on Computer and Communication Systems, 23–25 February 2019, Singapore, Singapore*. IEEE; 2019. p.355–359. DOI:10.1109/CCOMS.2019.8821775

2. Chondamrongkul N., Sun J., Warren I. Automated Security Analysis for Microservice Architecture. *Proceedings of the International Conference on Software Architecture Companion, 16–20 March 2020, Salvador, Brazil*. IEEE; 2020. p.79–82. DOI:10.1109/ICSA-C50368.2020.00024

3. Iannone E., Guadagni R., Ferrucci F., De Lucia A., Palomba F. The Secret Life of Software Vulnerabilities: a Large-Scale Empirical Study. *IEEE Transactions on Software Engineering*. 2023;49(1):44–63. DOI:10.1109/TSE.2022.3140868. EDN:GKKIKO

4. Fu J., Zhang K., Zheng J., Li W., Zhu Y. Research and Application of Grey Box Detection Technology Based on Reverse Engineering and Dynamic Pollution Diffusion. *Proceedings of the 7th Information Technology and Mechatronics Engineering Conference, 15–17 September 2023, Chongqing, China.* IEEE; 2023. p.2380–2384. DOI:10.1109/ITOEC57671.2023.10291380

5. Devine T.R., Campbell M., Anderson M., Dzielski D. SREP+SAST: A Comparison of Tools for Reverse Engineering Machine Code to Detect Cybersecurity Vulnerabilities in Binary Executables. *Proceedings of the International Conference on Computational Science and Computational Intelligence, 14–16 December 2022, Las Vegas, USA*. IEEE; 2022. p.862–869. DOI:10.1109/CSCI58124.2022.00156

6. Bhardwaj V., Kukreja V., Sharma C., Kansal I., Popali R. Reverse Engineering-A Method for Analyzing Malicious Code Behavior. *Proceedings of the International Conference on Advances in Computing, Communication, and Control, 03–04 December 2021, Mumbai, India*. IEEE; 2021. p.1–5. DOI:10.1109/ICAC353642.2021.9697150

7. Izrailov K.E., Pokusov V.V. Software platform architecture for converting machine code into a high-level representation for expert search of vulnerabilities. *Scientific Works of the Kuban State Technological University.* 2021;6:93–111. (in Russ.) EDN:AIOUWF

8. Buinevich M.V., Izrailov K.E., Pokusov V.V., Tailakov V.A., Fedulina I.N. An intelligent method of machine code algorithmization for vulnerabilities search. *Zašita informacii. Inside.* 2020;5(95):57–63. (in Russ.) EDN:HIHDOM

9. Cummins C., Fisches Z.V., Ben-Nun T., Hoefler T., O'Boyle M.F.P., Leather H. ProGraML: A Graph-based Program Representation for Data Flow Analysis and Compiler Optimizations. *Proceedings of the 38th International Conference on Machine Learning, PMLR, 18–24 July 2021, vol.139*. 2021. p.2244–2253.

10. Izrailov K. Genetic Decompilation Concept of the Telecommunication Devices Machine Code. *Proceedings of Telecommunication Universities*. 2021;7(4):10–17. (in Russ.) DOI:10.31854/1813-324X-2021-7-4-95-109. EDN:AIOFPM

11. Tonis R.B.M. Automating Scientific Paper Screening with Backus-Naur Form (BNF) Grammars. *Didactica danubiensis*. 2024;4(1):46–57.

12. Izrailov K.E. The genetic de-evolution concept of program representations. Part 1. *Voprosy kiberbezopasnosti.* 2024;1(59): 61–66. (in Russ.) DOI:10.21681/2311-3456-2024-1-61-66. EDN:CBCKRF

13. Izrailov K.E. The genetic de-evolution concept of program representations. Part 2. *Voprosy kiberbezopasnosti.* 2024;2(60): 81–86. (in Russ.) DOI:10.21681/2311-3456-2024-2-81-86. EDN:JUBPML

14. Hamberger P., Klammer C., Luger T., Moser M., Pfeiffer M., Piereder C. Specification-Based Test Case Generation for C++ Engineering Software. *Proceedings of the International Conference on Software Maintenance and Evolution, ICSME, 01–06 October 2023, Bogotá, Colombia*. IEEE; 2023. p.519–529. DOI:10.1109/ICSME58846.2023.00066

15. Sato Y. Specification-Based Test Case Generation with Constrained Genetic Programming. *Proceedings of the 20th International Conference on Software Quality, Reliability and Security Companion, QRS-C, 11–14 December 2020, Macau, China*. IEEE; 2020. p.98–103. DOI:10.1109/QRS-C51114.2020.00027

16. Huang C., Zhou H., Zhao H., Cai W., Zhou Z.Q., Jiang M. On the Usefulness of Crossover in Search-Based Test Case Generation: An Industrial Report. *Proceedings of the 29th Asia-Pacific Software Engineering Conference, APSEC, 06–09 December 2022, Japan*. IEEE; 2022. p.417–421. DOI:10.1109/APSEC57359.2022.00054

17. Schwachhofer D., Angione F., Becker S., Wagner S., Sauer M., Bernardi P., Polian I. Optimizing System-Level Test Program Generation via Genetic Programming. *Proceedings of the European Test Symposium, ETS, 20–24 May 2024, The Hague, Netherlands.* IEEE; 2024. p.1–4. DOI:10.1109/ETS61313.2024.10567817

18. Supaartagorn C. Web application for automatic code generator using a structured flowchart. *Proceedings of the International Conference on Software Engineering and Service Science, ICSESS, 24–26 November 2017, Beijing, China*. IEEE; 2017. p.114–117. DOI:10.1109/ICSESS.2017.8342876
19. Shinde K., Sun Y. Template-Based Code Generation Framework for Data-Driven Software Development. *Proceedings of the 4th Intl Conf on Applied Computing and Information Technology / 3rd Intl Conf on Computational Science / Intelligence and Applied Informatics / 1st Intl Conf on Big Data, Cloud Computing, Data Science & Engineering, ACIT-CSII-BCD, 12–14 December 2016, Las Vegas, USA*. IEEE; 2016. p.55–60. DOI:10.1109/ACIT-CSII-BCD.2016.023
20. Shimonaka K., Sumi S., Higo Y., Kusumoto S. Identifying Auto-Generated Code by Using Machine Learning Techniques. *Proceedings of the 7th International Workshop on Empirical Software Engineering in Practice, IWESEP, 13 March 2016, Osaka, Japan*. IEEE; 2016. p.18–23. DOI:10.1109/IWESEP.2016.18
21. Igwe K., Pillay N. Automatic programming using genetic programming. *Proceedings of the Third World Congress on Information and Communication Technologies, WICT 2013, 15–18 December 2013, Hanoi, Vietnam*. IEEE; 2013. p.337–342. DOI:10.1109/WICT.2013.7113158
22. Biryukov D.N., Dudkin A.S., Zakharov O.O. A method for testing information security tools based on the use of multivariate source code generation according to a given functional specification. *Proceedings of the Mozhaisky Military Space Academy.* 2022;684:113–122. (in Russ.) EDN:BJWKLG
23. Samohvalov E.N., Revunkov G.I., Gapanyuk Yu.E. Source code generation of software based on multilevel set of rules. *Herald of the Bauman Moscow State Technical University. Series Instrument Engineering.* 2014;5(98):77–87. (in Russ.) EDN:SVZLSL
24. Sokolov A.P., Makarenkov V.M., Pershin A.Yu., Laishevskiy I.A. Development of template-based code generation software for development of computer-aided engineering system. *Software Engineering*. 2019;10(9-10):400–416. (in Russ.) DOI:10.17587/prin.10.400-416. EDN:CHYPRE
25. Dovgal V.M., Korolkov O.F., Chaplygin A.A., Korolkova V.O. Considering of solving automatic code generation problem basing on given control production algorithm. *In the World of Scientific Discoveries.* 2012(1);25:220–235. (in Russ.) EDN:PBBWKP
26. Andrianova A., Itsykson V. Source code and partial specifications analysis for automated generation of unit tests. *Systems and Means of Informatics.* 2014;24(2):99–113. (in Russ.) DOI:10.14357/08696527140207. EDN:SJHATL
27. Saukh A.M., Hmelnov A.E. Source code fragments translation based on programming languages syntax and semantics specifications. *Vestnik NSU. Series: Information Technologies.* 2013;11(3):53–62. (in Russ.) EDN:RCHBLB
28. Haq I.U., Caballero J.A. Survey of Binary Code Similarity. *ACM Computing Surveys*. 2021;54(3):1–38. DOI:10.1145/3446371. EDN:KEPQCC
29. Kudelya V.N. Methods for enumerating paths in a graph. *H&ES Research.* 2023;15(5):28–38. (in Russ.) DOI:10.36724/2409-5419-2023-15-5-28-38. EDN:HQEASN
30. Kussainov A.R., Glazyrina N.S. Overview of static program code analysis tools. *Colloquium-Journal*. 2020;32-1(84):48–52. (in Russ.) EDN:JXSKQX

Статья поступила в редакцию 09.07.2025; одобрена после рецензирования 09.08.2025; принята к публикации 26.08.2025.
The article was submitted 09.07.2025; approved after reviewing 09.08.2025; accepted for publication 26.08.2025.

Информация об авторах:

ИЗРАИЛОВ Константин Евгеньевич — кандидат технических наук, доцент, профессор кафедры прикладной математики и безопасности информационных технологий Санкт-Петербургского университета государственной противопожарной службы МЧС России https://orcid.org/0000-0002-9412-5693

БУЙНЕВИЧ Михаил Викторович — доктор технических наук, профессор, профессор кафедры КБ-4 МИРЭА – Российского технологического университета https://orcid.org/0000-0001-8146-0022

Буйневич М.В. является членом редакционного совета журнала «Труды учебных заведений связи» с 2016 г., но не имеет никакого отношения к решению опубликовать эту статью. Статья прошла принятую в журнале процедуру рецензирования. Об иных конфликтах интересов авторы не заявляли.

Buinevich M.V. has been a member of the journal "Proceedings of Telecommunication Universities" Editorial Council since 2016, but has nothing to do with the decision to publish this article. The article has passed the review procedure accepted in the journal. The author has not declared any other conflicts of interest.