УДК 004.056 DOI:10.31854/1813-324X-2021-7-4-95-109

Концепция генетической декомпиляции машинного кода телекоммуникационных устройств

К.Е. Израилов[1, 2]*

[1]Санкт-Петербургский государственный университет телекоммуникаций им. проф. М.А. Бонч-Бруевича, Санкт-Петербург, 193232, Российская Федерация
[2]Санкт-Петербургский федеральный исследовательский центр Российской академии наук, Санкт-Петербург, 199178, Российская Федерация
*Адрес для переписки: konstantin.izrailov@mail.ru

Информация о статье
Поступила в редакцию 01.12.2021
Поступила после рецензирования 20.12.2021
Принята к публикации 21.12.2021

Ссылка для цитирования: 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

Аннотация: *Восстановление корректного исходного кода из машинного в интересах поиска и нейтрализации уязвимостей является актуальнейшей проблемой для области телекоммуникационного оборудования. Применимые для этого техники декомпиляции потенциально достигли своего эволюционного предела. Как результат, требуются новые концепции, способные осуществить качественный скачок в разрешении проблемы. Исходя из этого, в работе предлагается концепция генетической декомпиляции, представляющая собой решение задачи многопараметрической оптимизации в виде итерационного приближения экземпляров исходного кода к «истинному», компилируемому в заданный машинный. Осуществляется проверка данной концепции путем проведения серии экспериментов с разработанным программным прототипом на базовом примере машинного кода. Результаты экспериментов доказывают обоснованность предлагаемой концепции, предполагая тем самым новые инновационные направления обеспечения информационной безопасности в данной предметной области.*

Ключевые слова: *информационная безопасность, телекоммуникационное оборудование, машинный код, уязвимость, реверс-инжиниринг, декомпиляция, искусственный интеллект, генетический алгоритм, концепция*

ВВЕДЕНИЕ

Проблема безопасности телекоммуникационного оборудования актуализировалась в тот же момент, когда для передачи информации стало использоваться программное обеспечение. Причина такой связи заключается в наличии уязвимостей в программном коде, используемом для обработки передаваемых данных. При этом, источник появления уязвимости может быть как случайным (например, ошибка программиста), так и злонамеренным (например, «backdoor»). Вследствие чего телекоммуникационные устройства будут функционировать отличным от изначально задуманного образом, приводя тем самым к классической триаде нарушений информационной безопасности передаваемых данных: конфиденциальности, целостности и доступности. Одним из основных подходов противодействия этому является непосредственный поиск и нейтрализация уязвимостей в программном коде.

Безопасность программного кода

Способы поиска могут носить различный характер, зависящий, в том числе, от точки возникновения уязвимости в программном коде. Например, ее появление при создании архитектуры, реализации алгоритмов или непосредственном переводе кода в бинарное (выполняемое) представление потребует качественно отличных друг от друга способов, имеющих различную эффективность (как совокупность доли выявленных уязвимостей, скорости выявления и затраченных ресурсов). Однако в современных реалиях, когда телекоммуникационное оборудование разрабатывается крупными транснациональными корпорациями, его исходный код (далее – ИК) из соображений «ноу-хау» не предоставляется, а в наличии имеется лишь машинный код (далее – МК). Работа с последним существенно снижает количество применимых способов поиска уязвимостей, основной

причиной чего является практическая невозможность (или по крайней мере, высокая трудоемкость) анализа кода вручную экспертом. Автоматический же поиск [1–3], как правило, позволяет выявлять низкоуровневые уязвимости, затрагивая лишь нижние структурные уровни программы (отдельные выражения, доступ к данным и т. п.) и пропуская наиболее критичные – алгоритмические, архитектурные и концептуальные.

Для расширения использования всего «пула» способов поиска уязвимостей применяется так называемая техника «декомпиляции» (или «реверс-инжиниринга программного кода»), которая и определяет предметную область исследования, изложенного в текущей статье. Суть техники заключается в переводе программного кода из низкоуровневого представления, адаптированного для выполнения автоматом, в высокоуровневое, подходящее для работы с ним человеком [4]. Так, современные декомпиляторы (программные средства, реализующие технику декомпиляции) позволяют преобразовывать МК в некоторый псевдоисходный, из которого гипотетически и был получен данный МК. Стоит отметить, что в полном смысле задача декомпиляции считается нерешаемой, поскольку при компиляции (процессе трансформации ИК в МК) часть метаинформации теряется (комментарии, имена переменных, точный вид алгоритмов подпрограмм, архитектура программных модулей и т. п.) и, как следствие, не может быть восстановлена.

Научное противоречие

Несмотря на значительные успехи в исследовании и разработке декомпиляторов, предметная область содержит следующее научное противоречие в виде противопоставления – потребность ***vs*** возможность.

Потребность. С одной стороны, экспертам для поиска и нейтрализации уязвимостей требуется способ получения ИК, компилируемого в требуемый МК; при этом способ должен быть инвариантным относительно типа процессора выполнения. Первая часть потребности вытекает из того, что, как правило, уязвимость закладывается вручную в момент создания ИК (хотя и может делать небезопасным более высокоуровневые представления программы, такие, как его архитектура или концептуальная модель) [5]. Следовательно, анализ кода, близкого к такому исходному, не только даст эксперту больше информации для поиска уязвимостей, но и позволит ему произвести необходимую модификацию кода, приводящую после компиляции к безопасному МК. Причина второй части потребности заключается в том, что, исходя из огромного количества существующих и постоянно создаваемых процессорных архитектур, адаптация способов декомпиляции под новые процессоры будет не успевать за действиями злоумышленников, эксплуатирующих уязвимости. Это, в частности, обосновывается общей сложностью решения задачи декомпиляции, что требует огромных ресурсо-временных затрат на разработку программных средств.

Возможность. С другой стороны, существующие декомпиляторы (а также идеи, модели и алгоритмы, лежащие в их основе) предназначены для преобразования МК в ИК, который лишь отражает данный МК, а не стремится соответствовать изначально ИК; при этом, возникновение задачи поиска уязвимостей в МК для нового (например, менее популярного или только что созданного) процессора требует разработки декомпилятора «с нуля». Первая часть возможности вытекает из общего подхода к декомпиляции, когда правила преобразования машинных инструкций в конструкции ИК создаются вручную или полуавтоматически с привлечением ограниченного количества высококвалифицированных экспертов. Причина второй части заключается в том, что каждый такой декомпилятор предназначен для восстановления ИК для одного типа машинных инструкций или их близкого набора.

Разрешение противоречия

Разрешение противоречия возможно лишь гипотетически, поскольку существующие подходы лишь улучшают давно применяемые решения, не предполагая какого-либо качественного (т. е. прорывного) скачка. Таким образом, любые исследования, направленные на создание новых подходов в технике декомпиляции, являются безусловно актуальными.

Исследование в данной статье как раз и направлено на проверку авторской концепции декомпиляции, основанной на качественно новых принципах. Естественно, предлагаемый далее подход проверялся на разработанном прототипе декомпилятора для достаточно простых примеров МК; тем не менее, цель статьи заключается именно в проверке самой концепции; такой процесс доказательства имеет более широко известное название, как – PoC (*аббр. от. англ.* Proof of Concept). Естественно, описание полноценного решения не предполагается в данной статье, поскольку имеет существенную сложность для реализации. Однако его получение в будущем позволит практически полностью «свести на нет» указанное противоречие предметной области, разрешив тем самым проблему *гарантированной декомпиляции МК для любого типа процессора выполнения*, которая насчитывает уже несколько десятилетий.

ЭВОЛЮЦИЯ ТЕХНИКИ ДЕКОМПИЛЯЦИИ

Прежде чем описать предлагаемый программный прототип и провести с его помощью проверку концепции, рассмотрим общие этапы эволюции техник декомпиляции, которая послужила предпосылкой для возникновения новой концепции декомпиляции.

Первый этап

На первом этапе решение задачи преобразования МК в ИК производилось экспертами вручную – как ментально, так и с текстово-графическим отображением логики программы в виде блок-схем. Недостатками такого подхода были низкая скорость работы, высочайшие затраты ресурсов (в основном, человеческих) и как результат, большое количество ошибок.

Второй этап

На втором этапе, возникшем вследствие повышения производительности программно-аппаратного обеспечения, стали разрабатываться алгоритмы автоматического преобразования как отдельных блоков машинных инструкций, так и целых их структур, включая даже типы переменных [6]. В этот момент стали появляться первые программные реализации декомпиляторов. Однако, несмотря на существенно выросшую скорость применения способов (до нескольких секунд или минут), при снижении требований к экспертам, анализирующим получаемый ИК, существенного роста эффективности не произошло. Причина этого заключается в том, что разработка самих программных решений оказалась крайне сложной задачей. Также, поскольку алгоритмы преобразования МК в ИК создавались вручную, то росли и ошибки декомпиляции; или же процесс восстановления работал некорректно для непредвиденных входных данных.

Третий этап

На текущий момент, с точки зрения автора, идет третий этап развития техники декомпиляции, который ознаменован повсеместным внедрением искусственного интеллекта. Суть его применения к текущей задаче может заключатся в избавлении человека от необходимости создания всех правил преобразования МК в ИК вручную [7]. При этом, участие человека все равно будет требоваться – для обучения таких интеллектуальных систем или постобработки их результатов. И несмотря на большой потенциал в интеллектуализации подходов [8], суть остается та же – в конечном итоге необходимо создание правил, которые как требуют участия эксперта [9], так и не стремятся получить начальный ИК. При этом процесс обучения и доработки необходимо производить для каждого нового процессора выполнения заново.

Четвертый этап…?

Исходя из общей тенденции этапов эволюции техник декомпиляции, можно спрогнозировать постепенное и вынужденное сокращении участия человека в процессе, полностью отдавая выполнение процесса на откуп автомату (за счет продолжающегося роста аппаратных мощностей современной техники, включая появление суперкомпьютеров, облачных сервисов, распараллеливание вычислений и т. п.). Как результат, должны появиться способы декомпиляции, позволяющие не только снизить потребность в экспертном мнении, но и даже полностью отказаться от него. Именно такая авторская концепция, а также ее подтверждение, и будет продемонстрирована далее.

ГЕНЕТИЧЕСКИЙ АЛГОРИТМ

Идея качественно нового подхода в декомпиляции была описана автором ранее в работе [10]. Суть ее сводится к тому, что не МК преобразуется в ИК, а производится постепенный (многошаговый) поиск ИК, компилируемого в требуемый МК, или максимально близкий к нему. Таким образом, решается задача многопараметрической оптимизации, для чего предлагается использовать генетический алгоритм [11] – аналог происходящих механизмов эволюции в живой природе, хорошо себя зарекомендовавший для подобного рода задач. Суть алгоритма заключается в итеративном подстраивании популяции особей к среде существования путем их мутации, скрещивания и селекции. Принципы работы алгоритма хорошо известны и не требуют дополнительного объяснения. Способ же на базе данного алгоритма назван автором, как «генетическая декомпиляция», что полностью отражает его суть. Стоит отметить, что перенесение особенностей функционирования живой природы в сферу информационных технологий набирает все большую популярность, обосновывая тем самым эффективность такого рода подходов [12].

Разрешение противоречия

Рассмотрим, как повлияет доведение идеи генетической декомпиляции для полноценной реализации на разрешение обозначенного противоречия, и связанную с ним научную проблему.

Во-первых, в результате применения такого способа будет получаться ИК, который гарантированно компилируется в МК – сглаживание частного противоречия между первыми частями потребности и возможности. Во-вторых, реализация декомпиляторов будет практически не зависеть от типа процессора выполнения, поскольку работа генетического алгоритма не предполагает сильной зависимости от особенностей языка исходного кода и набора машинных инструкций – сглаживание частного противоречия между вторыми частями потребности и возможности.

Следовательно, доведение текущего исследования до практической реализации позволит полностью разрешить указанные противоречие и следующую из него проблему. И даже лишь именно этот умозаключительный факт может служить сильнейшей научной мотивацией для проведения текущего исследования.

Принцип работы

Стоит отметить, что принцип работы генетической декомпиляции противоположен классической, поскольку производится попытка компиляции различных экземпляров ИК в МК, а не наоборот. Таким образом, данный способ имеет качественное отличие от существующих.

Адаптация к предметной области

Произведем адаптацию классических генетических алгоритмов к задаче декомпиляции путем сопоставления основных терминов первого с сущностями второй: под особью понимается экземпляр ИК; хромосома особи – текст ИК; ген хромосомы – отдельная конструкция текста (в простейшем случае – символ); популяция – множество сгенерированных экземпляров ИК; поколение (или эпоха) – количество генерации экземпляров ИК из предыдущих (0-е поколение может быть сформировано случайным образом); селекция – отбор части экземпляров, компилируемый ИК которых наиболее близок к заданному декомпилируемому МК; мутация – случайное изменение конструкций ИК; скрещивание – составление нового ИК из двух имеющихся.

Наиболее интересным с научной и сложным с практической точки зрения сущностью является функция приспосабливаемости (иногда еще называемая фитнесс-функцией *от англ*. Fitness), которая показывает, насколько МК, скомпилированный из экземпляра ИК после *N*-поколений, близок к заданному. Также, поскольку преобразование между текстовым ассемблерным кодом (далее – АК) и бинарным МК считается практически тождественным, функция приспосабливаемости может работать только с текстовым представлением инструкций.

Фактически, суть генетического алгоритма сводится к оптимизации функции приспосабливаемости путем подбора ее параметров – генов особи, что тождественно поиску такого ИК, который в точности бы компилировался в заданный МК. Следовательно, результатом решения задачи оптимизации как раз и будет осуществление генетической декомпиляции. При этом, как хорошо видно, в сопоставленных сущностях отсутствует явная привязка как к языку программирования ИК, так и к типу процессора инструкций МК. Единственная зависимость имеется в процессе компиляции, что полностью нейтрализуется использованием соответствующего компилятора, представляющего собой внешнее по отношению к способу программное средство. Таким образом, аналогом выживания особей во враждебной среде для предметной области будет постепенная подстройка «размножающихся» и компилируемых экземпляров ИК к тому (а точнее, одному из набора), который в точности соответствует или максимально близок к заданному МК.

АРХИТЕКТУРА ПРОТОТИПА

Предложенная концепция ранее проверялась с помощью мысленного эксперимента, описанного автором в работе [10], что конечно же, не является достаточным для ее подтверждения. Поэтому был разработан прототип генетического декомпилятора (далее – Прототип), описание которого приводится ниже. Прототип предназначен для восстановления ИК на языке *C*, обработка которого с помощью утилиты компиляции «*cl.exe*» из состава *Microsoft Visual Studio Community* 2019 дает заданный АК (прямой аналог бинарного МК) для 64-битной версии архитектуры x86. Указанные характеристики не являются существенным ограничением, поскольку в теории может использоваться практически любой язык программирования и набор инструкций процессора (в том числе и для виртуальной машины) – это повлияет лишь на выбор компилятора.

Инструменты разработки

Для создания Прототипа использовалась среда разработки *Microsoft Visual Studio Community 2019*, в состав которой входит язык программирования *C#* и программная платформа *.Net Framework* версии 4.7.2. Для реализации генетического алгоритма использовалась библиотека *Genetic* версии 2.2.5 из состава открытой программной платформы *AForge.NET*.

Структурно-функциональная архитектура

Опишем строение Прототипа с помощью структурно-функциональной архитектуры – в виде структуры его модулей и их функциональных связей (рисунок 1).

Архитектура состоит из следующих модулей (см. рисунок 1), функциональные связи которых отражены направленными стрелками и не требуют дополнительных пояснений:

1) модуль запуска, ответственный за начало работы Прототипа, инициализацию структур и т. д.;

2) модуль завершения, ответственный за конец работы, деинициализацию структур и т. д.;

3) модуль ядра генетического алгоритма, являющийся упомянутой библиотекой *Genetic* из состава *AForge.NET*, реализующей основную логику эволюции экземпляров ИК;

4) модуль параметров алгоритма, содержащий настройки генетического алгоритма (например, частоту мутации и скрещивания);

5) модуль обработки эпохи, обеспечивающий получение нового поколения экземпляров ИК;

6) модуль мутации, осуществляющий мутацию экземпляра ИК путем случайного изменения его конструкций;

7) модуль скрещивания, осуществляющий скрещивание двух экземпляров ИК путем выбора случайным образом из их гена конструкций языка для формирования нового экземпляра;

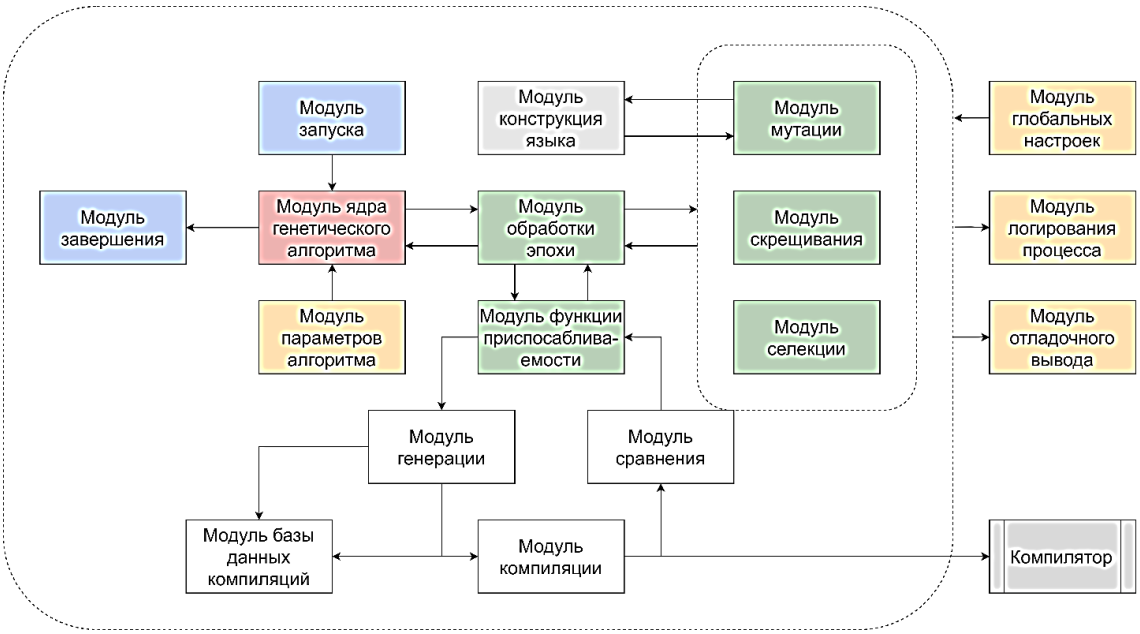

Рис. 1. Структурно-функциональный слой архитектуры прототипа генетического декомпилятора

Fig. 1. Structural and Functional Layer of the Genetic Decompiler Prototype Architecture

8) модуль селекции, осуществляющий выбор наиболее приспособленных экземпляров ИК, т. е. таких, АК которых наиболее близок к заданному;

9) модуль конструкция языка (программирования), содержащий одноименный набор для ИК (т. е. генов), на основании которых происходит мутация;

10) модуль функции приспосабливаемости, возвращающей численный показатель близости АК к заданному;

11) модуль генерации, создающий ИК экземпляра в текстовом виде, готовом для компиляции;

12) модуль базы данных компиляций, носящий оптимизационное назначение и хранящий все полученные ранее результаты компиляции ИК; таким образом, если некоторый ИК уже был скомпилирован ранее, то значение приспосабливаемости для его АК может быть получено из базы данных без запуска ресурсоемкой внешней утилиты;

13) модуль компиляции, запускающий внешнюю утилиту компиляции для получения АК из сгенерированного ИК;

14) модуль сравнения, оценивающий, насколько два АК близки друг к другу (способ сравнения учитывает то, что один из вариантов АК должен в процессе генетической декомпиляции приближаться к другому, заданному);

15) модуль глобальных настроек, содержащий настройки, используемые всеми другими модулями (например, уровень детализации логирования процесса или флаги отладочного вывода);

16) модуль логирования процесса, отображающий весь процесс эволюции поколений экземпляров ИК;

17) модуль отладочного вывода, отображающий информацию об ИК и АК экземпляров; наличие модуля обусловлено сложностью понимания и прогнозирования поведения генетического алгоритма, для чего требуется детализация его работы;

18) компилятор, являющийся внешним программным средством преобразования ИК на конкретном языке программирования в АК, набор инструкций которого полностью совпадает с инструкциями в заданном АК.

Отметим, что элемент архитектуры Компилятор в строгом смысле не является полноценным модулем, поскольку представляет собой полностью обособленное программное средство вне Прототипа. Однако его рассмотрение, как набора интерфейсов работы с файловой системой для компиляции ИК и чтения из файла содержимого его АК, может интерпретироваться как модульная часть Прототипа.

Принцип работы

Опишем достаточно общий алгоритм работы Прототипа. После запуска программы и инициализации параметров генетического алгоритма начинается итеративная генерация поколений экземпляров ИК. В процессе такой эволюции с некоторой частотой (заданной параметрами) в ИК экземпляров происходят мутации путем случайного изменения нескольких их конструкций. Затем часть экземпляров скрещивается, формируя таким образом новый экземпляр, ИК которого получен из случайного набора двух родительских. После этого происходит вычисление приспосабливаемости всех имеющихся экземпляров (часть из которых получена из предыдущего поколения и осталась неизменной, часть подверглась мутации, а часть появилась в результате скрещивания). Экземпляры с наилуч-

шей приспосабливаемостью отбираются для следующего поколения – т. е. осуществляется селекция; остальные же считаются «не выжившими» и отбрасываются. Как было указано выше, наиболее сложным элементом такой схемы является функция приспосабливаемости, поэтому ее алгоритм будет описан далее более детально.

Функция приспосабливаемости

Разработка корректной функции приспосабливаемости, используемой в работе генетического алгоритма, хоть и является сложной задачей, существенно уступает в этом разработке современных декомпиляторов. При этом функцию требуется создать единожды, а затем лишь корректировать.

В Прототипе реализован простейший алгоритм данной функции (в виде одноименного модуля), который хотя и может быть посчитан тривиальным, но даже на нем (как будет показано далее), генетическая декомпиляция обладает удовлетворительной работоспособностью.

Основной принцип работы алгоритма функции базируется на следующих двух идеях.

Идея 1. Подобие двух АК, имеющих текстовый вид, может быть (конечно же в простейшем случае) оценено, как совпадение их количества строк, состава отдельных строк и содержимого всего кода. За каждое такое совпадение начисляется определенное количество баллов, суммирование которых и составляет значение, возвращаемое функцией приспосабливаемости. Таким образом, бóльшие значения функция будет возвращать для тех экземпляров ИК, характеристики АК которых ближе к характеристикам заданного АК; при этом, приоритетом является полное совпадение кодов или хотя бы корреляция их ассемблерных строк. У такой идеи есть существенный недостаток, который (как показало предварительное тестирование Прототипа) ощутимо снижает работоспособность генетической декомпиляции. Поскольку далеко не каждый случайный набор конструкций языка программирования может быть скомпилирован в АК, то для большинства экземпляров ИК функция будет возвращать 0 баллов, так как формально АК экземпляра даже не был получен. Чтобы хотя бы частично нейтрализовать этот недостаток, была выдвинута и реализована следующая идея.

Идея 2. Прототип используется для проверки концепции генетической декомпиляции в ИК на языке *C*, особенностью синтаксиса которого является поддержка так называемой однопроходности, когда для компиляции отдельных конструкций языка отсутствует необходимость в «забегании вперед» по ИК и обработке последующих конструкций. Фактически, АК может генерироваться сразу по мере набора ИК. Такая особенность приводит к тому, что конструкции ИК имеют прямое отображение на инструкции АК (хотя и с некоторым разбросом, поскольку одна конструкция может преобразовываться в несколько инструкций). Как результат, близость конструкций двух ИК в начале и конце текста будет соответствовать близости инструкций в начале и конце АК. Поэтому ИК, начальные и конечные участки которого компилируются в аналогичные участки заданного АК априори будет считаться более приспособленным, чем не имеющий таких участков. Таким образом, целесообразно не просто сравнивать два АК – полученный из ИК и заданный, а оценивать схожесть их начальных и конечных участков (например, постепенным увеличением их длины). Это, в частности, позволит повысить количество успешных компиляций ИК и, тем самым, избавиться от недостатка, названного автором, как «проклятие нулевого значения функции приспосабливаемости для некомпилируемого ИК». К сожалению, подобное соответствие трудно применимо для участков в середине кода, поскольку сложно сопоставить адреса ИК и АК; впрочем, это все же не является невозможным и в будущем гипотетически позволит существенно повысить качество вычисления функции приспосабливаемости.

Псевдокод алгоритмов

Приведем далее алгоритм работы главной функции Прототипа – *MainProcees()*, отражающей работу его модулей (см. рисунок 1). Функция на вход принимает пути к файлам, необходимым для компиляции, и настройки ядра генетического алгоритма; а на выходе возвращает восстановленный ИК, компилируемый в заданный АК (путь к которому указан в составе первого параметра). Принцип работы функции описан ниже с помощью интуитивно понятного псевдокода Алгоритма 1.

```
Алгоритм 1: MainProcess
Input:
  FilePaths – настройки путей к файлам
  GeneticParams - параметры генетического алгоритма
Output:
  Text – исходный код, компилируемый
         в заданный ассемблерный код
Begin
1:  Compier.Init(FilePaths);
2:  Population = GeneticCore(GeneticParams,
                &EvaluateFitness(), &ApplyMutation(),
                &ApplyCrossover(), &ApplySelection());

3:  Do
4:    Population.RunEpoch();
5:    Logger.Write(Population.Chromosomes);
6:  Until Population.IsCompleted()

7:  Text = Population.BestChromosome.ToText;
8:  Return Text;
End
```

Приведем построчное описание работы алгоритма.

В строке 1 инициализируются пути модуля компиляции *Compiler*.

В строке 2 создается объект популяции *Population*, инициализируемый параметрами генетического алгоритма и функцией вычисления приспосабливаемости *EvaluteFitness()*, процедурами мутации *ApplyMutation()*, скрещивания *ApplyCrossover()* и селекции *ApplySelection()*. Принцип работы этих процедур достаточно тривиален и был приведен ранее при описании соответствующих модулей. Алгоритм работы функции приспосабливаемости более подробно будет рассмотрен далее. Также в этот момент формируется начальная (0-я) популяция.

В строке 3 начинается цикл по эволюционному развитию популяций экземпляров ИК.

В строке 4 производится получение новой популяции путем применения мутации, скрещивания и селекции, а также оценивания приспосабливаемости экземпляров.

В строке 5 производится логирование информации о текущей популяции для отслеживания процесса эволюции.

В строке 6 заканчивается цикл, начатый в строке 2. Условием окончания является получение популяции, неизменной во времени – т. е. все экземпляры которой имеют одинаковое значение функции приспосабливаемости. Данное значение не обязательно является максимальным, поскольку генетический алгоритм способен находить локальные максимумы функции, которые не являются глобальными, что представляет собой отдельную задачу, как правило решаемую подстройкой параметров работы.

В строке 7 в переменную *Text* генерируется текст ИК, полученный из экземпляра последней популяции, обладающей наилучшей приспосабливаемостью – т. е. любой из популяций, следуя условию окончания цикла в строке 6.

В строке 8 полученный текст ИК возвращается из функции.

Приведем далее алгоритм работы функции приспосабливаемости *EvaluateFitness()*, идеи работы которой были описаны выше. Функция на вход принимает хромосому экземпляра, представлявшего собой список конструкций его ИК в виде текстовых токенов, а на выходе возвращает балльное значение приспосабливаемости экземпляра (чем выше значение, тем ближе АК экземпляра к заданному). Принцип работы функции описан с помощью интуитивно понятного псевдокода Алгоритма 2.

Приведем построчное описание работы алгоритма.

В строке 1 инициализируется базовое значение приспосабливаемости *Fitness* (путем приравнивания его к нулю).

```
Algorithm 2: EvaluateFitness
Input:
  Tokens – ИК в виде списка токенов текста
           (т.е. конструкций языка)
Output:
  Fitness – значение фитнес функции для экземпляра ИК
Begin
 1:  Fitness = 0;
  :
 2:  For i = 0 .. Tokens.Length Do
 3:    tokensPart = Tokens.Take(i + 1);
 4:    text = tokensPart.ToText();
 5:    fit = CompileAndEvaluatePartialFitness(text);
  :
 6:    Fitness += COEFF_DIRECT_PARTIAL *
                  fit * (i + 1);
 7:  End For
  :
 8:  For i = 0 .. Tokens.Length Do
 9:    tokensPart = Tokens.Skip(i + 1);
10:    text = tokensPart.ToText();
11:    fit = CompileAndEvaluatePartialFitness(text);
  :
12:    Fitness += COEFF_REVERT_PARTIAL *
                  fit * (Tokens.Length - i - 1);
13:  End For
  :
14:  Return Fitness;
End
```

В строке 2 начинается цикл по увеличению значений индекса *i* с 0 до количества токенов из входного параметра.

В строке 3 выделяется часть начальных токенов (*tokensPart*) длиной текущего индекса цикла плюс 1.

В строке 4 из выделенных токенов формируется текст (*text*) ИК.

В строке 5 вычисляется значение приспосабливаемости сформированного текста с использованием функции алгоритма *CompileAndEvaluatePartial Fitness()* для получения частичной приспосабливаемости от области текста ИК (которая будет описана далее).

В строке 6 значение приспосабливаемости для выделенной части токенов прибавляется к общему значению приспосабливаемости с применением корректирующего коэффициента *COEFF_DIRECT_PARTIAL*. Также, чем из большего числа токенов был сформирован текст ИК, тем на бóльшую величину будет умножено и значение его приспосабливаемости – стимулируя тем самым генетический алгоритм формировать более «удачные» последовательности ген, соответствующих скомпилированному АК, близкому к заданному.

В строке 7 оканчивается цикл, начатый в строке 2.

Строки с 8-й по 13-ю практически полностью идентичны строкам с 2-й по 7-ю с тем отличием, что в строке 3 выделяется часть конечных, а не начальных токенов.

В строке 14 вычисленное значение приспосабливаемости возвращается из функции.

Приведем далее алгоритм работы функции частной приспосабливаемости *CompileAndEvaluatePartialFitness()*, используемой алгоритмом *EvaluateFitness()*. Функция на вход принимает текст ИК, а на выходе возвращает балльное значение приспосабливаемости соответствующего экземпляра. Принцип работы функции описан с помощью интуитивно понятного псевдокода Алгоритма 3.

```
Алгоритм 3: CompileAndEvaluatePartialFitness
Input:
  Text – текст ИК
Output:
  Fitness – значение фитнес функции для экземпляра ИК
Begin
 1:  Fitness = 0;
  :
 2:  If Text In Database.Keys Then
 3:    Fitness = Database[Text];
 4:  Else
 5:    Compiler.Write(Text);
 6:    Status = Compiler.Execute();
  :
 7:    If Status == True Then
 8:       Fitness += SCORE_CODE_IS_COMPILED;
  :
 9:      Lines = Compiler.Load();
10:      LinesOrig = Compiler.LoadOriginal();
  :
11:      IF Lines.Count == LinesOrig.Count Then
12:        Fitness += SCORE_SAME_NUMBER_OF_LINES;
13:      End If
  :
14:      MinCount = Min(Lines.Count,
                        LinesOrig.Count);

15:      For i == 0 .. MinCount Do
16:        If Lines[i] == LinesOrig[i] Then
17:          Fitness += SCORE_SAME_LINE;
18:        End If
19:      End For
  :
20:      If Lines.Count == LinesOrig.Count
         And Lines.Count == MinCount Then
21:        Fitness += SCORE_SAME_TEXT;
22:      End If
23:    End If
  :
24:    Database[Text] = Fitness;
25:  End If
  :
26:  Return Fitness;
End
```

Приведем построчное описание работы алгоритма.

В строке 1 инициализируется базовое значение приспосабливаемости *Fitness* (путем приравнивания его к нулю).

В строке 2 начинается основная ветка условия, выполняемая, если в базе данных (*Database*) присутствует значение приспосабливаемости для текста ИК (*Text*), переданного в алгоритм в качестве параметра.

В строке 3 возвращается ранее посчитанное значение приспосабливаемости для текста ИК.

В строке 4 начинается альтернативная ветка условия, выполняемая, если текст ИК не найден в базе данных.

В строке 5 происходит запись текста в файл временного ИК для последующей компиляции; для этого используется объект компиляции *Compiler*.

В строке 6 производится компиляция ИК, записанного во временный файл, в АК, записываемый в другой временный файл, с возвращением результата процесса в переменную *Status*.

В строке 7 начинается основная ветка условия, выполняемая, если статус компиляции оказался успешным.

В строке 8 увеличивается значение приспосабливаемости на заданное число баллов *SCORE_CODE_IS_COMPILED* – стимулируя тем самым генетический алгоритм формировать экземпляры с компилируемым ИК.

В строке 9 считываются строки АК из временного файла (полученного в строке 6) в массив *Lines*.

В строке 10 считываются строки заданного АК в массив *LinesOrig*.

В строке 11 начинается основная ветка условия, выполняемая, если количество ассемблерных строк скомпилированного и заданного кода равны.

В строке 12 увеличивается значение приспосабливаемости на заданное число баллов *SCORE_SAME_NUMBER_OF_LINES* – стимулируя тем самым генетический алгоритм формировать экземпляры с таким же количеством ассемблерных строк, как и у заданного.

В строке 13 оканчивается ветка проверки, начатая в строке 11.

В строке 14 вычисляется минимальное количество строк среди скомпилированного и заданного АК с сохранением значения в *MinCount*.

В строке 15 начинается цикл по увеличению значений индекса i с 0 до минимального количества ассемблерных строк, сохраненных в *MinCount*.

В строке 16 начинается основная ветка условия, выполняемая, если строка с индексом i скомпилированного и заданного АК совпали.

В строке 17 увеличивается значение приспосабливаемости на заданное число баллов *SCORE_SAME_LINE* – стимулируя тем самым генетический алгоритм формировать экземпляры, АК которых содержит строки, совпадающие со строками заданного.

В строке 18 оканчивается ветка условного выполнения, начатая в строке 16.

В строке 19 оканчивается цикл, начатый в строке 15.

В строке 20 начинается основная ветка условия, выполняемая, если все строки скомпилированного и заданного АК совпали (т. е. ИК экземпляра полностью компилируется в заданный АК).

В строке 21 увеличивается значение приспосабливаемости на заданное число баллов *SCORE_SAME_TEXT* – стимулируя тем самым генетический алгоритм сохранять в поколении экземпляры с ИК, в точности компилируемые в заданный АК.

В строках 22, 23 и 25 оканчивается ветка условного выполнения, начатая в строках 20, 7 и 4 соответственно.

В строке 24 происходит сохранение текста ИК и значение его приспосабливаемости в базе данных.

В строке 26 вычисленное значение приспосабливаемости возвращается из функции.

Интерфейс

Поскольку Прототип создан исключительно для проверки концепции генетической декомпиляции, то он имеет минимально необходимый интерфейс. Прототип представляет собой консольное приложение, принимающее в качестве параметров пути к следующим файлам:

– шаблон, служащий основой для генерации новых вариантов ИК;

– заданный АК, который является целевым для работы декомпиляции и с которым сравниваются все другие АК;

– ИК, из которого получается заданный АК (в процессе работы Модуля запуска); этот ИК считается отсутствующим в задаче декомпиляции, однако он упрощает тестирование Прототипа для различных вариантов заданного АК;

– временный ИК, генерируемый согласно файлу шаблона и хромосоме экземпляра ИК;

– временный АК, компилируемый из временного ИК;

– внешний компилятор, вызываемый Прототипом для получения временного АК из временного ИК.

Также прототип принимает следующие настройки (включая параметры алгоритма генетической декомпиляции):

– частота мутаций в виде доли особей одной популяции, хромосомы которых участвуют в мутации (по умолчанию равно 0,2 или 20 %);

– частота скрещиваний в виде доли особей одной популяции, хромосомы которых участвуют в скрещивании (по умолчанию равно 0,2 или 20 %);

– размер одной популяции в виде количество особей популяции, как изначальной, так и отбираемой в результате селекции (по умолчанию равно 10).

Как хорошо можно увидеть, какая-либо специфика языка программирования или типа процессора во входных параметрах отсутствует, а вся взаимосвязь между ИК и АК (и, соответственно, МК) определяется используемым внешним компилятором (последний параметр в списке путей). Вывод всей информации (полученного результата, ошибок, лога процесса и отладочной информации) Прототипом осуществляется в текстовую консоль.

ЭКСПЕРИМЕНТ

Для непосредственной проверки концепции генетической декомпиляции проведем ряд базовых экспериментов с помощью разработанного Прототипа.

Ограничения

Для упрощения проведения эксперимента и отсечения несущественного функционала Прототипа, а также ставя целью именно подтверждение концепции, а не полноценное тестирование решения, введем следующие 6 ограничений.

Ограничение 1. Будем восстанавливать АК для отсутствующего ИК, представляющего следующее тело функции, суммирующей 3 аргумента (на языке *C*):

```
Function: TestFunct
1: int test (int x, int y, int z) {
2:   int ret = x + y + z;
3:   return ret;
4: }
```

Согласно коду, функция *funct()* в качестве аргументов принимает три переменные (типа *Integer*) – *x*, *y* и *z*, затем их суммирует во временную переменную *ret* и возвращает ее значение.

Ограничение 2. Будем с помощью генетического алгоритма создавать популяции экземпляров ИК, соответствующих только самому математическому выражению, состоящему из следующей последовательности конструкций языка *C* – «*x+y+z*». Таким образом, синтаксическое «обрамление» выражение в виде сигнатуры функции и конструкции возврата ее значения через временную переменную учитываться не будет.

Ограничение 3. Для сравнения скомпилированного и заданного АК будем использовать только его часть, относящуюся к вычислению выражения, указанного в Ограничении 2. Эта часть в ассемблерной записи имеет следующий вид, расположенный между метками строк 2 и 3 функции *TestFunct()*:

```
; Line 2
      mov     eax, DWORD PTR y$[rsp]
      mov     ecx, DWORD PTR x$[rsp]
      add     ecx, eax
      mov     eax, ecx
      add     eax, DWORD PTR z$[rsp]
      mov     DWORD PTR ret$[rsp], eax
; Line 3
```

Ограничение 4. В качестве возможных конструкций будем использовать их достаточно малый набор, состоящий из 6 элементов – токенов языка: имена переменных – «*x*», «*y*», «*z*» и «*w*», а также математические выражения – «+» и «–».

Ограничение 5. Фактом окончания эволюции (т. е. ситуация, когда функция *Population.IsCompleted()* в псевдокоде алгоритма *MainProcess* возвращает *True*) будем считать получение подряд 100 поколений, экземпляры которого имеют одинаковое значение приспосабливаемости – т. е. эволюция оказывается

в некотором экстремуме, из которого не может выйти (или другими словами, в ситуации, когда экземпляры ИК не могут улучшить свою приспосабливаемость).

Ограничение 6. Все предустановленные значения коэффициентов, баллов, условий окончания декомпиляции и пр. получены эмпирически, а их уточнения являются отдельно стоящей задачей, выходящей за рамки текущего исследования.

Сценарий 1. Базовая проверка работоспособности

Перед запуском Прототипа оценим возможности решения подобной задачи полным перебором. В этом случае необходимо составить выражение из 5 конструкций (см. Ограничение 2), каждая из которых может быть одним из множества 6 элементов (см. Ограничение 4). Простые вычисления позволяют получить полное число комбинаций текстов ИК при переборе, как $6 \times 6 \times 6 \times 6 \times 6 = 7776$. А учитывая тот факт, что любой ИК должен быть преобразован в МК внешним компилятором (даже несмотря на то, что большинство вариантов попросту не смогут быть скомпилированы), увеличение числа комбинаций существенно повысит время декомпиляции.

Произведем выполнение Прототипа по базовому сценарию – декомпиляции одиночного АК, в соответствии с указанными ограничениями и параметрами генетического алгоритма по умолчанию. Это приведет к выводу в консоль следующего лога (часть несущественных строк пропущено, что обозначено в логе строкой «...»).

```
[ Epoch: 0 ] MAX : { x x x x x } => 201 (10:0)
[ Epoch: 1 ] MAX : { x x x + x } => 706 (2:0)
...
[ Epoch: 10 ] MAX : { x x x + x } => 706 (10:2)
[ Epoch: 11 ] MAX : { - y x + x } => 808 (2:0)
...
[ Epoch: 14 ] MAX : { - y + + x } => 1717 (1:0)
[ Epoch: 15 ] MAX : { - + x + x } => 1810 (1:0)
...
[ Epoch: 25 ] MAX : { - + x + x } => 1810 (1:0)
[ Epoch: 26 ] MAX : { y + x + x } => 1982 (1:0)
...
[ Epoch: 32 ] MAX : { y + x + x } => 1982 (1:0)
[ Epoch: 33 ] MAX : { z + y + x } => 1989 (2:0)
...
[ Epoch: 43 ] MAX : { z + y + x } => 1989 (10:7)
[ Epoch: 44 ] MAX : { x + y + x } => 1997 (1:0)
..
[ Epoch: 55 ] MAX : { x + y + x } => 1997 (10:3)
[ Epoch: 56 ] MAX : { x + y + z } => 7002 (1:0)
...
[ Epoch: 63 ] MAX : { x + y + z } => 7002 (8:0)
[ Epoch: 64 ] MAX : { x + y + z } => 7002 (10:1)
...
[ Epoch: 163 ] MAX : { x + y + z } => 7002 (10:100)
```

Строки лога имеют следующий формат:

```
[ Epoch: N_EPOCH ] MAX : { TEXT } => FITHESS
(N_BEST:N_EPOCH_WITH_BEST)
```

где *N_EPOCH* означает номер поколения (0 – для изначального, неподверженного мутациям и скрещиваниям); *TEXT* – текст ИК для экземпляра с наилучшей приспосабливаемостью; *FITNESS* – ее значение; N_*BEST* – количество особей, имеющих наилучшую приспосабливаемость в текущем поколении; *N_EPOCH_WITH_BEST* – количество поколений, все особи которого имеют наилучшую приспосабливаемость (используется для определения факта завершения процесса декомпиляции).

Как хорошо видно по логу, процесс декомпиляции завершился на 163-м поколении, притом получив верный результат – ИК выражения «$x + y + z$».

Интерпретация процесса декомпиляции

Восстановим и интерпретируем эволюцию особей ИС в процессе генетической декомпиляции, используя лог, полученный ранее.

Поколение 0 является изначальным (соответствующим некомпилируемому тексту – «x x x x x»), приспосабливаемость (т. е. близость к заданному АК) экземпляров ИК которого равна 201. Гены экземпляров фактически могут быть сформированы случайным образом и не представляет особого научно-практического интереса.

В поколении 1 произошли изменения (здесь и далее под ними подразумеваются мутация и скрещивание), в результате которых 4-й токен был изменен с «x» на «+», что повысило максимальную приспосабливаемость до 706. Естественно, возможно были и другие мутации, которые не отражаются, поскольку лог содержит текст ИК только с локально максимальной приспосабливаемостью.

В поколении 11 первый токен был изменен с «x» на «-», а второй – «x» на «y», увеличив тем приспосабливаемость до 808.

Затем, вплоть до поколения 43, гены (т. е токены ИК) экземпляров претерпевали изменения, представляя собой ошибочную ветку эволюции. Причина ошибочности заключается в том, что искомый ИК начинается с конструкции «x» вместо присутствующего в поколениях «-», чего эволюции удалось достичь лишь в 44-м поколении.

Затем, продолжая мутировать, скрещиваться и селектировать, было сформировано поколение 56, как раз и представляющее искомый ИК. И хотя в данном поколении присутствовал всего лишь один экземпляр, имеющий глобально максимальную приспосабливаемость, равную 7002, однако уже все экземпляры 64-го поколения обладали таким значением.

На протяжении последующих 100 поколений (т. е. до 163-го включительно) все экземпляры также имели одинаковую приспосабливаемость, что и посчиталось алгоритмом, как завершение работы.

Важным выводом, который можно сделать из анализа и интерпретации лога, является устойчивость генетической декомпиляции к отклонениям эволюции по «ошибочным» веткам развития (т. е. отход от истинного ИК, компилируемого в заданный МК); такое отклонение соответствует поколениям с 11 по 43. Это говорит о потенциальной достижимости алгоритмом глобального максимума функции приспосабливаемости даже в случае неудачных мутаций и скрещиваниях, основанных на генераторе случайных чисел.

Подсчет Прототипом количества реальных компиляций ИК (и его частей согласно работе алгоритма *EvaluteFitness()* в процессе вычисления приспосабливаемости) позволил получить значение 449 (естественно, с учетом работы Модуля базы данных компиляции, имеющего оптимизационное назначение). Таким образом, применение генетической декомпиляции по сравнению с полным перебором токенов ИК сократило количество ресурсоемких вызовов внешнего компилятора в 7776/449 ~ 17 раз. При этом, очевидно, что добавление в список возможных конструкций языка всего их множества катастрофически (для времени выполнения) увеличит время полного перебора, хотя гипотетически на скорость работы Прототипа повлияет существенно меньше.

Сценарий 2. Вычисление результативности

Естественно, ожидание того, что алгоритм Прототипа будет всегда давать верные результаты, даже при учете всех 6 введенных ограничений, было бы слишком амбициозным. И простейшее тестирование путем многократного запуска Прототипа подтверждают это – в ряде случаев «выживает» популяция, ИК экземпляров которой на протяжении 100 поколений обладает одинаково высокой приспосабливаемостью, при этом не компилируясь в заданный МК. Поэтому далее проведем эксперимент по нахождению результативности Прототипа, как доли его запусков, приведших к верному ИК (т. е. соответствующих заданному МК) по отношению ко всем запускам.

В процессе данного сценария было произведено 10000 запусков Прототипа в специальном исследовательском режиме, выводящем полученный ИК (с префиксом *Text*), значение его приспосабливаемости (с префиксом *Fitness*) и количество поколений (с префиксом *Epoaches*). Время эксперимента на персональном компьютере средней мощности составило около 17 мин. Часть полученного лога представлена ниже.

```
(XX.XX.2021 18:41:49)  Text = x + y + z, Fitness =
7002, Epoches = 133
(XX.XX.2021 18:42:01)  Text = - y + - y, Fitness =
1816, Epoches = 146
(XX.XX.2021 18:42:13)  Text = x + y + z, Fitness =
7002, Epoches = 227
...
(XX.XX.2021 18:58:30)  Text = x + y + z, Fitness =
7002, Epoches = 176
(XX.XX.2021 18:58:30)  Text = x + y + z, Fitness =
7002, Epoches = 288
(XX.XX.2021 18:58:30)  Text = x + y + z, Fitness =
7002, Epoches = 149
```

Результаты статистической систематизации лога приведены в таблице 1, имеющей следующие столбцы:

1) № – номер решения, соответствующий одному из максимумов функции приспосабливаемости, определенных алгоритмом, как глобальный;

2) Текст ИК – текст на языке *C*, компиляция которого получает МК, наиболее близкий (с точки зрения генетического алгоритма) к требуемому;

3) Количество решений – общее число решений с одинаковым текстом ИК, полученных генетическим алгоритмом;

4) Доля решения – округленная процентная доля решений с данным ИК по отношению ко всем решениям (т. е. к 10000);

5) Значение приспосабливаемости – количество баллов, полученных при сравнении МК, скомпилированного из экземпляров ИК полученного решения, с заданным МК;

6) Среднее количество поколений – количество итераций эволюции, после которой значение приспосабливаемости экземпляров одного поколения стабилизировалось (т. е. не менялось 100 раз);

7) Первое истинное поколение – номер поколения, экземпляры которого обладали одинаковой приспосабливаемостью в течение последующих 100 итераций (значение равно среднему количеству итераций минус 100).

ТАБЛИЦА 1. Результаты статистической систематизации лога работы Прототипа

TABLE 1. Results of the Prototype's Log Statistical Systematization

1	2	3	4	5	6	7
1	x + y + z	7374	0,74 %	7002	181	81
2	x - y + z	1	≈ 0 %	1998	140	40
3	x + y – z	1537	0,15 %	1997	152	52
4	- y - - y	1088	11 %	1816	167	67

Согласно результатам табличного анализа можно сделать следующие выводы.

Во-первых, всего было найдено 4 решения, каждое из которых соответствует одному из максимумов функции приспосабливаемости. При этом все решения представляют собой компилируемый код, что гипотетически еще раз подтверждает потенциал предложенной концепции.

Во-вторых, одно из решений является истинным (под номером 1), поскольку его ИК компилируется в АК, тождественный заданному.

В-третьих, доля верных решений составляет 74 %, что превышает долю ближайшего конкурента (под номером 3) в 7374/1537 ~5 раз. Следо-

вательно, значение 74 % может считаться результативностью Прототипа генетической декомпиляции. Графическая интерпретация всех полученных решений представлена на рисунке 2 (с соблюдением примерных соотношений межу значениями приспосабливаемости).

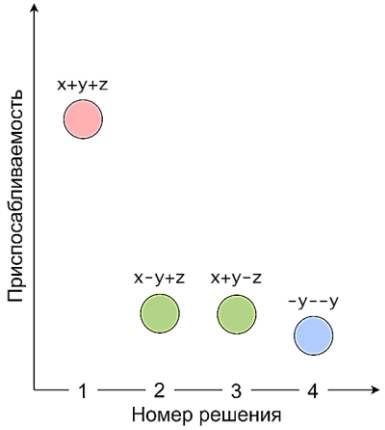

Рис. 2. Приспосабливаемость полученных решений при множественной генетической декомпиляции

Fig. 2. Adaptability of the Obtained Solutions for Multiple Genetic Decompilation

Также, при установлении факта, что алгоритм нашел локальный, а не глобальный максимум (в остальных $\frac{1+1537+1088}{10000} \sim 0{,}26$, т. е. 26 % случаев), достаточно будет осуществить декомпиляцию еще несколько раз, поскольку уже после второго запуска процент ошибки составит $\left(\frac{1+1537+1088}{10000}\right)^2 \sim 0{,}07$, т. е. 7 %. Такой прием позволит существенно увеличить результативность Прототипа, доведя его практически до 100 %.

В-четвертых, значение приспосабливаемости истинного решения (под номером 1) отличается от остальных (под номером 2, 3 и 4) практически в 3,5 раза, что дает потенциал для снижения количества ошибок алгоритма (т. е. выбора неверных максимумов) путем корректировки параметров самого генетического алгоритма для «покидания» границ локального экстремума (например, увеличением размера популяции, частоты мутаций, механизма скрещивания и т. п.).

И, в-пятых, первое поколение экземпляров, завершающих генетическую декомпиляцию, не превышает 100, что означает достаточно быстрое эволюционное развитие.

Сценарий 3. Оценка оптимизации компиляции

Основной операцией, снижающей оперативность работы Прототипа, является использование внешнего компилятора. При этом, если бы даже механизм преобразования ИК в АК был встроен в Прототип, то он все равно занимал бы значительную долю времени. В интересах этого был создан Модуль базы данных компиляций, хранящий значения приспосабливаемости для всех скомпилированных ранее ИК. Оценим его возможности по оптимизации путем выполнения эксперимента по сценарию, заключающемуся в отслеживании количества реальных компиляций в процессе генетической декомпиляции. Полученная зависимость в виде гистограммы представлена на рисунке 3.

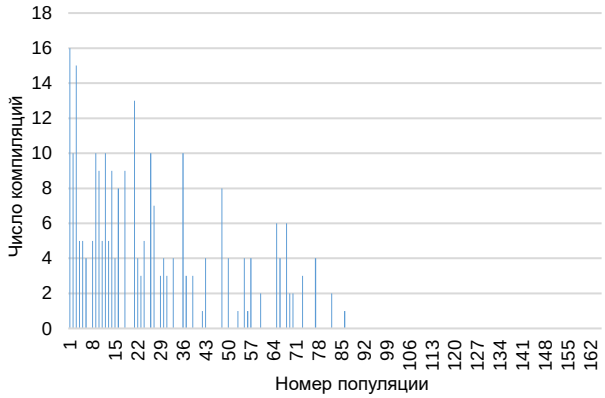

Рис. 3. Зависимость количества компиляций исходного кода от номера популяции

Fig. 3. The Dependence of Compilations' Source Code Number on Population Number

Как хорошо видно (см. рисунок 3), снижение количества компиляций с преобразованием ИК в АК имеет снижающийся тренд (линейный или обратно экспоненциальный). Так, уже после 90-го поколения реальная компиляция отсутствует полностью, все последующие значения функции приспосабливаемости берутся из базы данных; что существенно понижает скорость декомпиляции.

Сценарий 4. Подбор параметров генетического алгоритма

Оценим то, насколько подстройка генетического алгоритма влияет на его результативность. Для этого проведем эксперимент по сценарию, в котором осуществляется перебор параметров генетического алгоритма следующим образом:

– размер популяции меняется от 2 до 20 с шагом 1 (по умолчанию было 10);

– доля мутаций меняется от 0,0 до 1,0 с шагом 0,1 (по умолчанию было 0,2);

– размер селекций меняется от 0 до 1,0 с шагом 0,1 (по умолчанию было 0,2).

Следовательно, в ходе проведенного эксперимента, перебиралось $19 \times 11 \times 11 = 2299$ комбинаций параметров.

Для каждого набора параметров осуществлюсь 100-кратное проведение декомпиляции (т. е. генерация ИК и вычисление приспосабливаемости), что позволило получить достаточно адекватную усредненную результативность Прототипа.

Таким образом, в ходе эксперимента было произведено $2299 \times 100 = 229900$ запусков Прототипа, а общее время эксперимента заняло примерно 430 мин.

По мере увеличения размера популяции (особенно при высоких значениях доли мутации и скрещивания) росло и время работы алгоритма. Это объяснимо тем, что при высоких долях большее число особей в популяции подвергается изменениям, требующим вычислений функции приспособленности, что сказывается на времени выполнении, даже несмотря на оптимизацию за счет применения базы данных. Однако итоговый рост времени декомпиляции не оказался критичным.

Используя полученный лог работы Прототипа, была построена зависимость минимальной, средней и максимальной результативности декомпиляции от размера популяции при всех значениях долей мутации и скрещивания (рисунок 4).

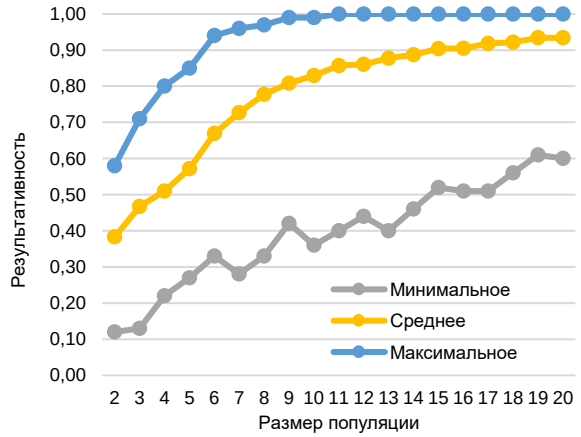

Рис. 4. Зависимость результативности генетической декомпиляции от размера популяции

Fig. 4. The Dependence between Genetic Decompilation's Performance on Population Size

Анализ лога работы Прототипа и графика на рисунке 4 позволяет сделать следующие предварительные выводы. Уже при размере популяций, равной 11, максимальное значение результативности достигает 1,0 – при долях мутаций и скрещиваний, равных парам значений (0,9; 0,1) и (1,0; 0,00), соответственно. Таким образом, «правильный» выбор параметров генетического алгоритма позволяет со 100 % вероятностью определять ИК, соответствующий заданному МК. Также прослеживается общая тенденция к увеличению среднего значения результативности, обосновывая то, что увеличение размера популяции приведет и к заметному повышению качества работы генетической декомпиляции. При этом, даже минимальный размер популяции, равный 2, в среднем, практически в половине случаев (а, точнее, следуя графику на рисунке 4 в 38 %) обеспечивает успешную декомпиляцию. Также на размерах популяции, более 15, доля результативности превышает середину в 50 %. Эта популяция может считаться точкой бифуркации в работоспособности Прототипа, поскольку даже при самой неудачной комбинации остальных параметров генетического алгоритма доля верных решений будет превалировать над долей неверных.

Анализ результатов

Исходя из результатов проведенных экспериментов, концепцию генетической декомпиляции можно считать подтвержденной, что следует из следующих утверждений.

Согласно Сценарию 1 эксперимента Прототип обладает базовой работоспособностью, а интерпретация работы генетической декомпиляции является адекватной и закономерной.

Согласно Сценарию 2 эксперимента Прототип имеет существенную результативность, позволяя в 3 из 4 раз корректно производить декомпиляцию. При этом процент успеха может быть повышен достаточно простыми техническими приемами, включающими подбор параметров.

Согласно Сценарию 3 реализованная оптимизация позволяет в процессе эволюционного развития существенно повысить скорость работы за счет сокращения реальных вызовов внешнего компилятора (что практически невозможно в случае применения полного перебора).

Согласно Сценарию 4 подбор оптимальных параметров генетического алгоритма (размер популяции, доля мутаций и скрещиваний) гипотетически позволяет достичь верного нахождения ИК в 100 % случаев.

Также в экспериментах не затронута подстройка константных коэффициентов и баллов, которые также могут оказать существенное влияние на успешность декомпиляции.

ЗАКЛЮЧЕНИЕ

На текущем этапе исследования проверялась концепция генетической декомпиляции, для чего был реализован соответствующий Прототип, на котором проводился ряд сценариев из состава эксперимента. Результаты последних могут служить обоснованием подтверждения концепции, хотя и пример АК (а, следовательно, ИК и МК) можно считать недостаточно сложным.

В текущем состоянии реализация полноценного генетического декомпилятора по предложенной концепции вряд ли осуществима, поскольку существует ряд проблем, не позволяющих проверить работоспособность Прототипа на более сложных примерах. Тем не менее, с точки зрения автора, устранение проблем носит чисто технический (и, в некотором смысле, логический) характер, не ставя под сомнения верность концепции в целом.

Продолжением исследования должно стать всестороннее тестирование Прототипа с целью технического (а не концептуального) усовершенствования его слабых сторон для проведения первых тестов на МК реальных программных продуктов. Это позволит не только полностью доказать состоятельность предложенной концепции, но и произве-

сти сравнение Прототипа, соответствующего новому 4-му этапу эволюции техник декомпиляции, с существующими аналогами, относящимися ко 2-му и 3-му этапам. В результате можно будет оценить перспективность общего вектора развития изложенной теоретико-практической «мысли» в данной предметной области.

Необходимо отметить, что автор понимает всю смелость предложенной концепции, гипотетически способной раз и навсегда решить проблему декомпиляции, притом инвариантной не только от языка программирования ИК, но и от мнемоники записи АК, а также процессора выполнения МК. В связи с этим, автор готов адекватно воспринять критику проведенного исследования, его идей и результатов, а также выслушать любые авторитетные точки зрения касательно проблематики затронутой предметной области.

ФИНАНСИРОВАНИЕ

Работа выполнена при частичной финансовой поддержке бюджетной темы 0073-2019-0002.

Список используемых источников

1. Гурин Р.Е. Обзор и анализ инструментов, которые осуществляют верификацию бинарного кода программы // Новые информационные технологии в автоматизированных системах. 2014. № 17. С. 514–518.
2. Тихонов А.Ю., Аветисян А.И. Комбинированный (статический и динамический) анализ бинарного кода // Труды Института системного программирования РАН. 2012. Т. 22. С. 131–152.
3. Каушан В.В. Поиск ошибок выхода за границы буфера в бинарном коде программ // Труды Института системного программирования РАН. 2016. Т. 28. № 5. С. 135–144.
4. Бугеря А.Б., Ефимов В.Ю., Кулагин И.И., Падарян В.А., Соловьев М.А., Тихонов А.Ю. Программный комплекс для выявления недекларированных возможностей в условиях отсутствия исходного кода // Труды Института системного программирования РАН. 2019. Т. 31. № 6. С. 33–64. DOI:10.15514/ISPRAS-2019-31(6)-3
5. Buinevich M., Izrailov K., Vladyko A. Metric of vulnerability at the base of the life cycle of software representations // Proceedings of the 20th International Conference on Advanced Communication Technology (ICACT, Chuncheon, South Korea, 11–14 February 2018). IEEE, 2018. PP. 1–8. DOI:10.23919/ICACT.2018.8323940
6. Трошина Е.Н., Чернов А.В. Восстановление типов данных в задаче декомпилирования в язык С // Прикладная информатика. 2009. № 6(24). С. 99–117.
7. 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.
8. Obert J., Loffredo T. Efficient Binary Static Code Data Flow Analysis Using Unsupervised Learning // Proceedings of the 4th International Conference on Artificial Intelligence for Industries (AI4I, Laguna Hills, USA, 20–22 September 2021). IEEE, 2021. PP. 89–90. DOI:10.1109/AI4I51902.2021.00030
9. Jia X., Bin Z., Chao F., Chaojing T. An Automatic Evaluation Approach for Binary Software Vulnerabilities with Address Space Layout Randomization Enabled // Proceedings of the International Conference on Big Data Analysis and Computer Science (BDACS, Kunming, China, 25–27 June 2021). IEEE, 2021. PP. 170–174. DOI:10.1109/BDACS53596.2021.00045
10. Израилов К.Е. Применение генетических алгоритмов для декомпиляции МК // Защита информации. Инсайд. 2020. № 3(93). С. 24–30.
11. Загинайло М.В., Фатхи В.А. Генетический алгоритм как эффективный инструмент эволюционных алгоритмов // Инновации. Наука. Образование. 2020. № 22. С. 513–518.
12. Максимова Е.А. Модель состояний субъектов критической информационной инфраструктуры при деструктивных воздействиях в статичном режиме // Труды учебных заведений связи. 2021. Т. 7. № 3. С. 65–72. DOI:10.31854/1813-324X-2021-7-3-65-72.

* * *

The Genetic Decompilation Concept of the Telecommunication Devices Machine Code

K. Izrailov[1, 2]

[1]The Bonch-Bruevich Saint-Petersburg State University of Telecommunications, St. Petersburg, 193232, Russian Federation
[2]St. Petersburg Federal Research Center of the Russian Academy of Sciences, St. Petersburg, 199178, Russian Federation

Article info
DOI:10.31854/1813-324X-2021-7-4-95-109
Received 1st December 2021
Revised 20th December 2021
Accepted 21st December 2021

For citation: Izrailov K. The Genetic Decompilation Concept of the Telecommunication Devices Machine Code. *Proc. of Telecom. Universities.* 2021;7(4):95–109. (in Russ.) DOI:10.31854/1813-324X-2021-7-4-95-109

Abstract: *Reverse engineering correct source code from a machine code to find and neutralize vulnerabilities is the most pressing problem for the field of telecommunications equipment. The decompilation techniques applicable for this have potentially reached their evolutionary limit. As a result, new concepts are required that can make a quantum leap in problem solving. Proceeding from this, the paper proposes the concept of genetic decompilation, which is a solution to the problem of multiparameter optimization in the form of iterative approximation of instances of the source code to the "original" one which will compile to the given machine code. This concept is tested by conducting a series of experiments with the developed software prototype using a basic example of machine code. The results of the experiments prove the proof of the concept, thereby suggesting new innovative directions for ensuring information security in this subject area.*

Keywords: *information security, telecommunications equipment, machine code, vulnerability, reverse engineering, decompilation, artificial intelligence, genetic algorithm, proof of concept.*

FUNDING

The reported study was partially funded by the budget project 0073-2019-0002.

References

1. Gurin R.E. Review and Analysis of Tools that Verify the Binary Code of the Program. *Novye informatsionnye tekhnologii v avtomatizirovannykh sistemakh*. 2014;17:514–518. (in Russ.)
2. Tikhonov A.Yu., Avetisyan A.I. Combined (Static and Dynamic) Analysis of Binary Code. *Proceedings of the Institute for System Programming of the RAS*. 2012;22:131–152. (in Russ.)
3. Kaushan V.V. Buffer Overrun Detection Method in Binary Code. *Proceedings of the Institute for System Programming of the RAS*. 2016;28(5):135–144. (in Russ.)
4. Bugerya A.B., Efimov V.Yu., Kulagin I.I., Padaryan V.A., Solovev M.A., Tikhonov A.Yu. A Software Complex for Revealing Malicious Behavior in Untrusted Binary Code. *Proceedings of the Institute for System Programming of the RAS*. 2019;31(6):33–64. (in Russ.) DOI:10.15514/ISPRAS-2019-31(6)-3
5. Buinevich M., Izrailov K., Vladyko A. Metric of vulnerability at the base of the life cycle of software representations. *Proceedings of the 20th International Conference on Advanced Communication Technology, ICACT, 11–14 February 2018, Chuncheon, South Korea*. IEEE; 2018. p.1–8. DOI:10.23919/ICACT.2018.8323940
6. Troshina K.N., Chernov A.V. Type Reconstruction for C Decompilation. *Journal of Applied Informatics*. 2009;6(24):99–117. (in Russ.)
7. 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. *Zaŝita informacii. Inside*. 2020;5(95):57–63. (in Russ.)
8. Obert J., Loffredo T. Efficient Binary Static Code Data Flow Analysis Using Unsupervised Learning. *Proceedings of the 4th International Conference on Artificial Intelligence for Industries, AI4I, 20–22 September 2021, Laguna Hills, USA)*. IEEE; 2021. p.89–90. DOI:10.1109/AI4I51902.2021.00030
9. Jia X., Bin Z., Chao F., Chaojing T. An Automatic Evaluation Approach for Binary Software Vulnerabilities with Address Space Layout Randomization Enabled. *Proceedings of the International Conference on Big Data Analysis and Computer Science, BDACS, 25–27 June 2021, Kunming, China*. IEEE; 2021. p.170–174. DOI:10.1109/BDACS53596.2021.00045
10. Izrailov K.E. Applying of Genetic Algorithms to Decompile Machine Code. *Zaŝita informacii. Inside*. 2020;3(93):24–30. (in Russ.)
11. Zaginailo M.V., Fatkhi V.A. Genetic Algorithm as an Effective Tool for Evolutionary Algorithms. Innovatsii. Nauka. Obrazovanie. 2020;22:513–518. (in Russ.)
12. Maximova E.A. Model of the States of Critical Information Infrastructure Subjects under Destructive Influences in Static Mode. *Proc. of Telecom. Universities.* 2021;7(3):65–72. (in Russ.) DOI:10.31854/1813-324X-2021-7-3-65-72.

Сведения об авторе:

ИЗРАИЛОВ Константин Евгеньевич

кандидат технических наук, доцент кафедры защищенных систем связи Санкт-Петербургского государственного университета телекоммуникаций им. проф. М.А. Бонч-Бруевича, старший научный сотрудник Санкт-Петербургского федерального исследовательского центра Российской академии наук,
konstantin.izrailov@mail.ru
https://orcid.org/0000-0002-9412-5693