КОНЦЕПЦИЯ ГЕНЕТИЧЕСКОЙ ДЕЭВОЛЮЦИИ ПРЕДСТАВЛЕНИЙ ПРОГРАММЫ. Часть 1

Израилов К. Е.[1]

DOI: 10.21681/2311-3456-2024-1-61-66

Цель исследования: *развитие направления реверс-инжиниринга программ, заключающегося в преобразовании их представлений в одно из предыдущих.*

Методы исследования: *системный анализ, мысленный эксперимент, аналитическое моделирование, многокритериальная оптимизация.*

Полученные результаты: *предложена концепция генетической деэволюции представлений программы, предлагающая процесс их восстановления не обратным способом, т.е. от текущего к предыдущему, а прямым – работая с псевдо-предыдущим представлением и оценивая его близость к исследуемому текущему; принцип концепции основан на решении оптимизационной задачи с помощью генетических алгоритмов.*

В первой части статьи введена онтологическая модель предметной области, в терминах которой предложена высокоуровневая схема (де)эволюции представлений, отражающая преобразования между ними, а также внесение и обнаружение уязвимостей; дано формализованное описание процессов на схеме.

Научная новизна *заключается в качественно новой точке зрения на восстановление представлений – с помощью процесса итеративного подбора предыдущего для соответствия (после эволюции) текущему, при этом, основанного на принципах генетических алгоритмов, а также имеющего полностью формализованный вид.*

Ключевые слова: *концепция, эволюция, реверс-инжиниринг, реинжиниринг, обратная разработка, обратный инжиниринг, генетический алгоритм, уязвимость.*

THE GENETIC DE-EVOLUTION CONCEPT OF PROGRAM REPRESENTATIONS. Part 1

Izrailov K. E.[2]

The goal of the investigation: *development of the programs reverse engineering direction, which consists in transforming their state into one of the previous.*

Research methods: *system analysis, mental experiment, analytical modeling, multicriteria optimization.*

Result: *the genetic de-evolution concept of program representations has been proposed, suggesting a process for their restoration in a backway, i.e. from the current to the previous one, and direct way – working with the pseudo-previous representation and assessing its proximity to the current one being studied; the concept principle is based on solving an optimization problem using genetic algorithms.*

In the first part of the article, a subject area ontological model is introduced, in terms of which a high-level scheme for the (de)evolution of representations is proposed, reflecting transformations between them, as well as the introduction and detection of vulnerabilities; a processes formalized description in the diagram is given.

The scientific novelty *consists in a qualitatively new point of view on the representations restoration – using the iterative process selection of the previous one to correspond (after evolution) to the current one, at the same time, based on the principles of genetic algorithms, and also having a completely formalized form.*

Keywords: *concept, evolution, reverse engineering, reengineering, backward engineering, genetic algorithm, vulnerability.*

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

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

Введение

Реверс-инжиниринг (сокр. реинжиниринг, далее – РИ) программного обеспечения является актуальнейшим направлением в области информационных технологий [1]. Назначением РИ в частном смысле является получение исходного кода программы (а точнее псевдоисходного, лишь близкого к изначальному), как правило имеющего текстовый вид и подходящего для ручного анализа экспертом, из ее машинного кода, имеющего бинарный вид и выполняемого на ЦПУ [2]; как правило, такой процесс называется *декомпиляцией* [3, 4]. В общем смысле, понимаемом автором, РИ предназначен для восстановления метаинформации, утерянной при инжиниринге программы согласно ее жизненному циклу; например, помимо получения исходного кода из машинного возможно восстановить алгоритмы функций программы, ее архитектуру, концептуальную модель и даже (в пределе) саму идею [5, 6].

Основными целями применения РИ, естественно, в условиях отсутствия доступного исходного кода, являются следующие. Во-первых, использование программ, созданных недоверенным производителем (недружественными странами, нелегальными и криминальными организациями, недобросовестными конкурентами), приводит ко множеству информационных рисков [7], поскольку такие программы могут содержать уязвимости, делая тем самым свое функционирование отличным от заявленного и/или требуемого [8]. Для чего необходимо получение информации о фактическом функционале программы, что является побочным результатом РИ [9]. Во-вторых, для осуществления эффективного (т.е. с сохранением функционала, в кратчайшие сроки и при адекватных затратах ресурсов) импортозамещения зарубежного программного обеспечения отечественным [10, 11], что особо актуально в условиях неослабевающего санкционного давления, требуется восстановление деталей реализации имеющегося – это является основным назначением РИ. И, в-третьих, при полностью легальной разработке программ, плотно взаимодействующих с внешними библиотеками или программными продуктами, требуется детальное понимание специфики такого информационного обмена; например, форматов входных данных, кодов результата выполнения и т.п. Если разработчик сторонних продуктов обеспечивает своевременную и квалифицированную обратную связь, а также имеет «добротную» документацию, то реализация взаимодействия не представляет особой сложности. Однако в ином случае, например, когда разработчик прекратил поддержку или не выполняет своих обязательств, РИ позволит самостоятельно составить спецификацию на интерфейсы взаимодействия внешних продуктов, что даст возможность реализовать и собственные [12].

На данный момент, РИ в частном смысле (т.е. как декомпиляция машинного кода в исходный) является отдельно стоящей проблемой, не имеющей удовлетворительного научно-практического решения. И хотя существуют средства декомпиляции (наиболее популярным из которых является IDA Pro с плагином Hex-Rays [13–15]), все они поддерживают ограниченный набор ЦПУ машинного кода и выдают далеко не всегда корректный псевдоисходный код; по крайне мере, рекомендуется проверять результат их работы вручную, а также применять дополнительные автоматические средства повышения человеко-ориентированности (например, путем добавления комментариев [16]). РИ в общем же смысле в принципе оставлено без существенного внимания, поскольку получение более высокоуровневых представлений программы (например, алгоритмов) из машинного кода считается второстепенной или несущественной задачей (с чем автор категорически не согласен). При этом, понимание архитектуры программы и ее концептуальной модели (а также и самой идеи) существенно упростило бы достижение трех вышеуказанных целей применения РИ. Сам РИ является достаточно технически сложным процессом – в случае автоматизации резко снижается качество результатов и появляются ограничения в применении, а при его ручном проведении – резко возрастает время и затрачиваемые ресурсы. Одним из путей качественного повышения эффективности данного процесса, с точки зрения автора, является применение искусственного интеллекта, и, в частности, генетических алгоритмов. Для этого все преобразования программы в процесс ее прямого инжиниринга рассматриваются, как части единой эволюции программы – от представления первоначальной идеи до выполняемого машинного кода (через концептуальную модель, архитектуру, алгоритмы, исходный код); обратные же преобразования представлений, логично, могут быть названы деэволюцией программы. Далее будет описана общая концепция такой генетической деэволюции представлений программы, лежащая в основе всего авторского направления исследований.

Онтологическая модель

Для использования единой терминологии введем и опишем следующую онтологическую модель предметной области, содержащую сущности и их взаимосвязи, и представленную на рис. 1; используются следующие обозначения: прямоугольник – сущность, стрелка – взаимосвязь, зеленый фон – объект, синий фон – действие, красный фон – объект из области информационной безопасности.

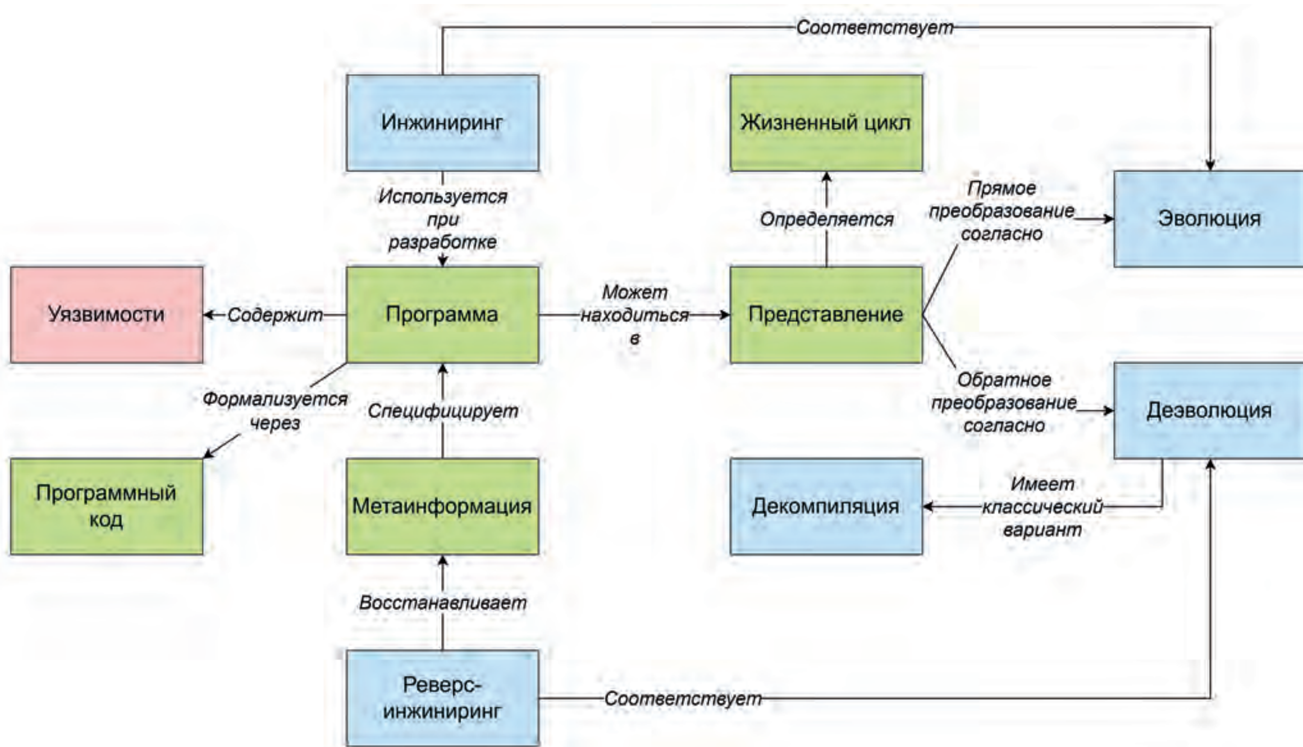

Рис. 1. Онтологическая модель предметной области

Онтологическая модель (см. рис. 1) состоит из следующих элементов (курсивом отмечены связи):

1. Программа – информационный объект, предназначенный (как в данной момент, так и в будущем) для выполнения на ЦПУ и *формализуемый через* программный код;
2. Программный код – формальная запись (текстовая, графическая, иная) реализации функционала по решению определенных задач;
3. Метаинформация о программе – информация, *специфицирующая* программу в человеко-ориентированном виде и не предназначенная напрямую для ее выполнения;
4. Представление программы – типовое состояние (как совокупность формы и содержания [17]), в котором *может находиться* программа в определенный момент своего существования;
5. Жизненный цикл программы – схема, *определяющая* закономерность смены представлений программы;
6. Эволюция представлений программы – *прямое преобразование* представлений программы согласно жизненного цикла (т.е. от первоначальной идеи к конкретной реализации);
7. Деэволюция представлений программы – *обратное преобразование* представлений программы согласно жизненного цикла (т.е. от конкретной реализации к первоначальной идее);
8. Инжиниринг программы – процесс, *используемый при разработке* программы, и соответствующий эволюции ее представлений;
9. Реверс-инжиниринг (реинжиниринг) представления программы – процесс, *восстанавливающий* метаинформацию из программы и *соответствующий* деэволюции ее представлений;
10. Декомпиляция программы – *классический вариант* деэволюции представлений программы по переходу из машинного кода в псевдоисходный код;
11. Уязвимость программы – дефект, содержащийся в программе, в виде отличия ее содержания от «идеального» (изначально задуманного).
12. Опишем предложенную онтологию в практическом аспекте. Центральным звеном модели (см. рис. 1) является программа, формализуемая через программный код и которая в процессе своего существования может находиться в некоторых представлениях (возникших исторически и/или обусловленных типовым процессом программного инжиниринга) – машинного кода, идентичного ему ассемблерного кода, исходного кода, алгоритмов, архитектуры, концептуальной модели и идеи. Каждое представление программы определяется совокупностью формы и содержания, смысл которых неоднократно объяснялся в авторских статьях [5, 6, 17]: форма – внешний вид программы, заданный с использованием определенной нотации (языка программирования, блок-схем, текстового описания и т.п.); содержание – внутренняя логика программы, определяющая ее функционал и не зависящая от способа описания (т.е. от формы). Схема

перехода между представлениями определяется жизненным циклом программы: например, в процессе компиляции исходного кода получается ассемблерный код, который после ассемблирования (и линковки) преобразуется в машинный. При инжиниринге программы, когда из ее начальной идеи в голове человека получается машинный код, готовый для выполнения на ЦПУ, происходит постепенная эволюция представлений. Обратным процессом является РИ, при котором из текущего представления программы (как правило, машинного или ассемблерного кода) получается псевдоисходный код, за которым возможно получить алгоритмы, архитектуру и т.д.; для этого, в частности, восстанавливается метаинформация о программе (а точнее, о ее необходимом представлении), такая, как деление на подпрограммы (с аргументами и возвращаемыми значениями), в ряде случаев их имена, детали алгоритмов, схема взаимодействия между программными модулями и т.п. В частности, реинжиниринг путем восстановления предыдущих представлений программы позволяет искать заложенные в них уязвимости [18]. Классическим примером деэволюции представлений является декомпиляция, которая заключается в преобразовании (как правило, с помощью специализированных программных средств) программы из машинного кода в псевдоисходный. Отметим, что под программным кодом понимается любое «задание» программы в строгом (или формализованном) виде, т. е. как классический исходный код, так и бинарный машинный код, а также детализированные блок-схемы, полное и корректное описание архитектуры и т. п.

Далее будет дано представление предлагаемой концепции генетической деэволюции представлений программы с помощью двух следующих схем:

1) (де)эволюции всего множества представлений, описывающей процесс эволюционных изменений программы;
2) генетической деэволюции двух смежных представлений, описывающей получение предыдущего из них по имеющемуся с применением генетических алгоритмов.

Схемы (де)эволюции представлений

Исходя из введенной онтологической модели и ее наложения на практику, логику (де)эволюции представлений программы можно представить в виде следующей схемы (см. рис. 2); используются следующие обозначения: $\boldsymbol{Form_X}$ и $\boldsymbol{Content_X}$ – форма и содержание X-го представления (т.е. имеющего идентификатор X).

Дадим ряд достаточно очевидных пояснений к схеме (де)эволюции представлений (см. рис. 2). Переход от текущего представления (X-1 и X) к следующему представлению (X и X+1) осуществляется в процессе эволюции (согласно схеме жизненного цикла) с помощью некоторого способа синтеза (X-1→X и X→X+1); последним могут быть как программные решения (классические компиляторы и ассемблеры), так и экспертные действия (создание архитектуры, перевод ее в алгоритмы,

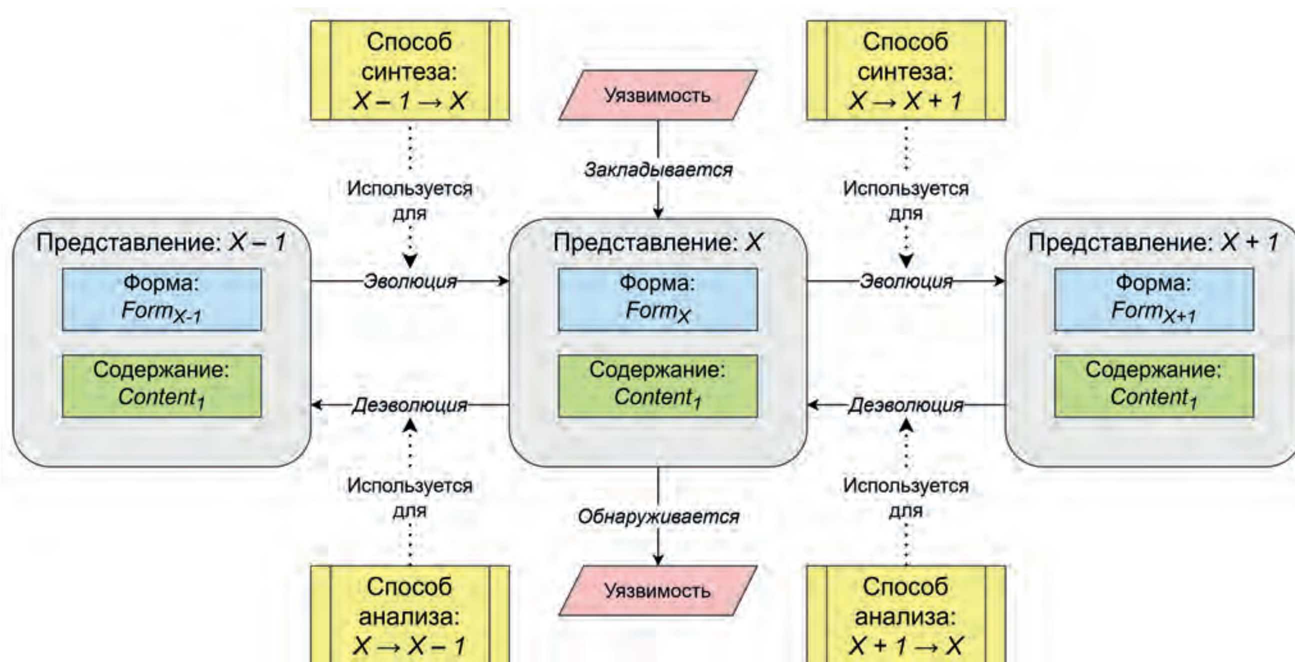

Рис. 2. Общая схема (де)эволюции представлений программы

кодирование программы), а также их замена на генеративный искусственный интеллект [19, 20]. При этом содержание представления не должно меняться, поскольку оно определяет функционал программы, который в конечном представлении должен соответствовать задуманному изначально – таким образом все содержания тождественны созданному в 1-м представлении: $\forall X{:}Content_X{=}Content_1$. Форма представления же наоборот подвергается изменениям – от человеко-ориентированного (понятного эксперту и не подходящего для выполнения ЦПУ) к машинно-ориентированному (применимого только для непосредственного выполнения автоматом) – таким образом, формы представлений неидентичны ($\forall X,Y,X{\neq}Y{:}Form_X{\neq}Form_Y$), поскольку претерпевают последовательные изменения ($Form_{X\text{-}1}{\longrightarrow}Form_X{\longrightarrow}Form_{X+1}$). Изменение содержания (случайное или намеренное) в некотором представлении означает появление уязвимостей в программе (т.е. $\exists X,Y{:}Content_X{\neq}Content_Y$); поиск же уязвимостей наиболее эффективен именно в том представлении, в котором она была внесена, поскольку в результате эволюции форма уязвимости с объективной неизбежностью также будет видоизменяться. Обратный же процесс получения предыдущих представлений из исходного ($X{+}1{\longrightarrow}X$ *и* $X{\longrightarrow}X\text{-}1$) является их деэволюцией. Как было указано выше, данный процесс является наиболее сложным, и именно на качественно новый подход к его реализации и направлено настоящее исследование.

Все процессы, приведенные на схеме, могут быть записаны следующим формальным образом.

Каждое представление программы (***R***, аббр. от англ. ***Representation***) является совокупностью формы (перев. на англ. ***Form***) и содержания (перев. на англ. ***Content***):

$$R \equiv \langle Form, Content\rangle,$$

что также может быть записано через отдельные компоненты ***R*** (в случае конкретного представления указывается его идентификатор в нижнем левом индексе):

$$\begin{cases} R^{Form} \equiv Form \\ R^{Content} \equiv Content \end{cases}.$$

Процесс эволюции представлений программы может быть записан как:

$$R_x \rightrightarrows R_{x+1},$$

где «$\rightrightarrows$» – операция эволюции; x – идентификатор текущего представления; а $x{+}1$ – последующего (соответственно, $x\text{-}1$ – идентификатор предыдущего представления).

Автором выделено несколько типов представлений (перев. на англ. ***Type***), которые используются при классическом инжиниринге программ [5], выполняемых напрямую на ЦПУ (т.е. без применения виртуальных машин [21]):

$$Type^R \in \begin{Bmatrix} Type^R_{Идея}, Type^R_{КонцептуальнаяМодель} \\ Type^R_{Архитектура}, Type^R_{Алгоритмы} \\ Type^R_{ИсходныйКод}, Type^R_{МашинныйКод} \end{Bmatrix},$$

где $Type^R$ – тип представления ***R***, $Type^R_{...}$ – идентификатор типа, соответствующий указанному в нижнем индексе представлению; представление ассемблерного кода (и его тип) будет опущено, поскольку оно достаточно хорошо преобразуется в/из машинного и может не учитываться.

Аналогичным образом, процесс деэволюции представлений можно записать следующим образом:

$$R_{x\text{-}1} \leftleftarrows R_x.$$

где «$\leftleftarrows$» – операция деэволюции.

В процессе (де)эволюции происходит лишь изменение формы представления программы, а ее содержание (естественно, в случае неизменности функционала программы – отсутствия привнесенных уязвимостей) остается неизменным, т.е.

$$\forall x,y{:}x{\neq}y{:}\begin{cases} R_x^{Form} \neq R_y^{Form} \\ R_x^{Content} = R_y^{Content} \end{cases},$$

где x и y – два различных идентификатора представлений; соответственно, если идентификаторы будут одинаковыми, то и представления программы полностью совпадут.

Для осуществления эволюции ($\rightrightarrows$) представлений применяется соответствующий способ (***W***, аббр. от англ. ***Way***, перевод на русс. Путь); таким образом, саму операцию можно представить (с помощью символа «:») в виде функции:

$$\rightrightarrows{:}R_{x+1} = E\,(R_x, W_x),$$

где W_x – способ синтеза нового представления из x-го.

Обратную же к эволюции операцию, а именно – деэволюцию ($\leftleftarrows$), можно записать аналогичным образом с использованием диакритического знака отрицания (черты над символом):

$$\leftleftarrows{:}R_{x\text{-}1} = \overline{E}\,(R_x, \overline{W}_x),$$

где $\overline{W}$ – обратный к синтезу способ получения предыдущего представления из x-го путем анализа.

Внесение уязвимостей (***V***, аббр. от англ. ***Vulnerability***), меняющее, как указывалось, содержание представления программы (без перехода к новой форме), можно записать как:

$$\begin{cases} R_x \longrightarrow \widehat{R_x} \\ R_x^{Form} = \widehat{R_x^{Form}} \\ V = \widehat{R_x^{Content}} \setminus R_x^{Content} \\ V \notin \varnothing \end{cases},$$

где «$\longrightarrow$» – операция преобразования представления; R_x^{Form} и $R_x^{Content}$ – форма и содержание -го представления; «$\widehat{\square}$» – диакритический знак для обозначения

сущности (в данном случае, представление) с уязвимостью; «\» – оператор разности двух множеств (в данном случае, содержания представления с и без уязвимости); Ø – пустое множество (в данном случае, уязвимостей).

Таким образом, описание внесения уязвимости представляет собой систему следующих уравнений:

1) преобразование безопасного представления в имеющее уязвимость;
2) равенство форм этих представлений;
3) определение уязвимости, как различия содержаний этих представлений;
4) указание на существование уязвимости (т.е. ее отличие от пустого множества).

Аналогичным образом, обнаружение уязвимости можно записать в виде функции ее детектирования (D, аббр. от англ. $\boldsymbol{Detection}$):

$$V = D_x\,(\widehat{R_x}),$$

где D_x – операция детектирования уязвимости в небезопасном представлении $\widehat{R_x}$, работающая с x-ым представлением. Естественно, реинжиниринг и поиск уязвимостей существенно усложнится в случае применения защит от анализа кода [22].

Продолжение следует…

Литература

1. Израилов К. Е. Методология реверс-инжиниринга машинного кода. Часть 1. Подготовка объекта исследования. Труды учебных заведений связи. 2023. Т. 9. № 5. С. 79–90. DOI: 10.31854/1813-324X-2023-9-5-79-90
2. Bhardwaj V., Kukreja V., Sharma C., Kansal I., Popali R. Reverse Engineering-A Method for Analyzing Malicious Code Behavior // In proceedings of the International Conference on Advances in Computing, Communication, and Control (Mumbai, India, 2021, 03-04 December 2021). PP. 1–5. DOI: 10.1109/ICAC353642.2021.9697150
3. Mauthe N., Kargén U., Shahmehri N. A Large-Scale Empirical Study of Android App Decompilation // In proceedings of the IEEE International Conference on Software Analysis, Evolution and Reengineering (Honolulu, HI, USA, 09-12 March 2021). 2021. PP. 400–410. DOI: 10.1109/SANER50967.2021.00044
4. Borrello P., Easdon C., Schwarzl M., Czerny R., Schwarz M. CustomProcessingUnit: Reverse Engineering and Customization of Intel Microcode // In proceedings of the IEEE Security and Privacy Workshops (San Francisco, CA, USA, 25-25 May 2023). 2023. PP. 285–297. DOI: 10.1109/SPW59333.2023.00031
5. Израилов К. Е. Моделирование программы с уязвимостями с позиции эволюции ее представлений. Часть 1. Схема жизненного цикла // Труды учебных заведений связи. 2023. Т. 9. № 1. С. 75–93. DOI:10.31854/1813-324X-2023-9-1-75-93
6. Израилов К. Е. Моделирование программы с уязвимостями с позиции эволюции ее представлений. Часть 2. Аналитическая модель и эксперимент // Труды учебных заведений связи. 2023. Т. 9. № 2. С. 95–111. DOI:10.31854/1813-324X-2023-9-2-95-111
7. Самарин Н. Н. Модель безопасного функционирования программного обеспечения, формализующая контроль использования памяти и обращений к ней процессора // Наукоемкие технологии в космических исследованиях Земли. 2021. Т. 13. № 1. С. 68–79.
8. Язов Ю. К., Соловьев С. В. Методология оценки эффективности защиты информации в информационных системах от несанкционированного доступа / Санкт-Петербург: Издательство «Наукоемкие технологии». 2023. 258 с.
9. Афанасов А. К., Цой А. И. Извлечение данных Android-приложения WhatsApp // Процессы управления и устойчивость. 2021. Т. 8. № 1. С. 246–253.
10. Полонский А. М. Импортозамещение программного обеспечения и организация обучения студентов с использованием отечественного или свободного программного обеспечения // Актуальные проблемы экономики и управления. 2022. № 2 (34). С. 65–82.
11. Исаев Р. А. Проблемы и перспективы отечественного аналитического программного обеспечения в условиях реализации программ импортозамещения // Промышленные АСУ и контроллеры. 2020. № 11. С. 10–22.
12. Шарков И. В. Метод восстановления протокольных автоматов по бинарному коду // Труды Института системного программирования РАН. 2022. Т. 34. № 5. С. 43–62.
13. Cao K., Leach K. Revisiting Deep Learning for Variable Type Recovery // In proceedings of the IEEE/ACM 31st International Conference on Program Comprehension (Melbourne, Australia, 15-16 May 2023). 2023. PP. 275–279. DOI: 10.1109/ICPC58990.2023.00042
14. Кусаинов А. Р., Глазырина Н. С. Обзор инструментов статического анализа программного кода // Colloquium-Journal. 2020. № 32–1(84). С. 48–52.
15. Xu Z., Wen C., Qin S. Type Learning for Binaries and Its Applications // In proceedings of the IEEE Transactions on Reliability. 2019. Vol. 68, No. 3. PP. 893–912. DOI: 10.1109/TR.2018.2884143
16. Rani P., Birrer M., Panichella S., Ghafari M., Nierstrasz O. What Do Developers Discuss about Code Comments? // In proceedings of the IEEE 21st International Working Conference on Source Code Analysis and Manipulation (Luxembourg, 27-28 September 2021). 2021. PP. 153–164. DOI: 10.1109/SCAM52516.2021.00027
17. Израилов К. Е., Татарникова И. М. Подход к анализу безопасности программного кода с позиции его формы и содержания // Актуальные проблемы инфотелекоммуникаций в науке и образовании (АПИНО-2019): сборник научных статей VIII Международной научно-технической и научно-методической конференции (Санкт-Петербург, 27-28 февраля 2019 г.). 2019. С. 462–467.
18. Фомин А. И., Хапилина Д. А., Горлищев И. А., Олимпиенко К. В. Инженерный анализ наличия программных закладок в программном обеспечении при отсутствии исходных кодов // Научная мысль. 2019. Т. 7. № 1 (31). С. 123–126.
19. Саколик А. ChatGPT и разработка программного обеспечения // БИТ. Бизнес & Информационные технологии. 2023. № 2 (125). С. 38–41.
20. Ebert C., Louridas P. Generative AI for Software Practitioners // IEEE Software. 2023. Vol. 40. No. 4. PP. 30–38. DOI: 10.1109/MS.2023.3265877
21. Majidha Fathima K. M., Santhiyakumari N. A Survey on Evolution of Cloud Technology and Virtualization // In proceedings of the Third International Conference on Intelligent Communication Technologies and Virtual Mobile Networks (Tirunelveli, India, 04-06 February 2021). 2021. PP. 428–433. DOI: 10.1109/ICICV50876.2021.9388639.
22. Маркин Д. О., Макеев С. М. Система защиты терминальных программ от анализа на основе виртуализации исполняемого кода // Вопросы кибербезопасности. 2020. № 1 (35). С. 29–41. DOI: 10.21681/2311-3456-2020-01-29-4