Научная статья
УДК 004.42
DOI:10.31854/1813-324X-2023-9-6-68-82

Методология проведения реверс-инжиниринга машинного кода. Часть 2. Статическое исследование

Константин Евгеньевич Израилов, konstantin.izrailov@mail.ru

Санкт-Петербургский Федеральный исследовательский центр Российской академии наук,
Санкт-Петербург, 199178, Российская Федерация

Аннотация: *Изложены результаты создания единой методологии проведения реверс-инжиниринга машинного кода устройств. Данная, вторая часть цикла статей посвящена статическому исследованию кода с целью восстановления его метаинформации (исходного кода, алгоритмов, архитектуры, концептуальной модели), а также поиска в нем уязвимостей. Проводится обзор научных публикаций на тему существующих методов и средств статического анализа машинного кода. Дается детальное описание и формализация шагов этапа, а также примеры их применения на практике. Частичная схема предлагаемой методологии приводится в графическом виде с указанием получаемых основных и промежуточных результатов.*

Ключевые слова: *реверс-инжиниринг, обратная разработка, программная инженерия, статический анализ, информационная безопасность, уязвимости, методология, схема*

Ссылка для цитирования: Израилов К.Е. Методология реверс-инжиниринга машинного кода. Часть 2. Статическое исследование // Труды учебных заведений связи. 2023. Т. 9. No 6. С. 68–82. DOI:10.31854/1813-324X-2023-9-6-68-82

Methodology for Machine Code Reverse Engineering. Part 2. Static Investigation

Konstantin Izrailov, konstantin.izrailov@mail.ru

Saint-Petersburg Federal Research Center of the Russian Academy of Sciences,
St. Petersburg, 199178, Russian Federation

Abstract: *The creating results a unified methodology for reverse engineering the machine code of devices are presented. This second part of the articles series is devoted to static research of code in order to restore its metainformation (source code, algorithms, architecture, conceptual model), as well as search for vulnerabilities in it. A scientific publications review on the topic of existing methods and tools for static analysis of machine code is carried out. A detailed description and formalization of the steps of the stage is given, as well as examples of their application in practice. A proposed methodology partial diagram is presented in graphical form, indicating the main and intermediate results obtained.*

Keywords: *reverse engineering, backwards engineering, software engineering, static analysis, information security, vulnerabilities, methodology, scheme*

For citation: Izrailov K. Methodology for Machine Code Reverse Engineering. Part 2. Static Investigation. *Proceedings of Telecommun. Univ.* 2023;9(6):68–82. DOI:10.31854/1813-324X-2023-9-6-68-82

Введение

Статья данного цикла посвящена актуальной задаче – реверсу-инжинирингу (далее – РИ) машинного кода (далее – МК), обусловленной необходимостью восстановления принципов и деталей работы программного обеспечения при отсутствии его исходного кода (далее – ИК). Результаты такого исследования оказываются востребованы в следующих случаях. Во-первых, отличие функционала МК от заявленного может означать наличие в нем уязвимостей, потенциально приводящих к информационным угрозам. Во-вторых, потеря ИК программной библиотеки в ряде случаев может затруднить разработку взаимодействующих с ней продуктов, для чего требуется его хотя бы частичное восстановление. В-третьих, одним из способов импортозамещения (естественно, с учетом юридических сторон вопроса) может стать глубокое и всестороннее изучение МК зарубежных продуктов и разработка на их основе отечественных аналогов.

В первой части цикла статей [1] было дано общее представление о предлагаемой автором методологии РИ (далее – Методология), для описания которой вводилась онтологическая модель предметной области. Также был описан 1-й из 4 этапов Методологии, посвященный подготовке объекта исследования, под которым понимается МК всех программ из единой программной системы. Продолжая представление Методологии, во второй части цикла будет описан ее 2-й этап, посвященный наиболее трудоемкой и технически сложной части исследования – статическому анализу (т. е. без выполнения программ), основной задачей которого считается восстановление различной человеко-ориентированной метаинформации, потерянной при компиляции исходного кода в МК.

Обзор работ

Проведем обзор работ, посвященных вопросу РИ в части существующих методов и средств для статического анализа МК; при этом уделим внимание именно отдельным шагам такого анализа, их принципам и особенностям. Все это позволит построить этапы Методологии, относящиеся к статическому исследованию МК.

Работа [2] посвящена обзору различных методов и программных средств, предназначенных для анализа МК в интересах восстановления алгоритмов его работы, а также используемых форматов данных. Отмечается некоторая успешность решения задачи РИ с помощью продукта IDA Pro и различных отладчиков МК, хотя и с некоторыми ограничениями в виде работы с небольшими участками бинарных данных, а также сложности автоматизации процесса. Авторами предложено применение методов статического анализа совместно с динамическими, что дает более полную картину функционала исследуемой программы.

Отдельное внимание в статье уделяется восстановлению формата данных, под которым понимаются границы полей в программном объекте, их группировка, семантика и дополнительная информация (константные значения, связи между полями-указателями и пр.). Часть авторского коллектива в другой работе [3] описывает применение статико-динамического анализа МК для поиска в нем недекларированных возможностей.

В [4] делается сравнительный анализ методов и средств декомпиляции программ (т. е. восстановления их ИК по машинному коду), а также предлагается новый метод определения типов данных в МК. Рассматриваются такие декомпиляторы, как Boomerang, DCC, REC и Hex-Rays (как плагин к IDA Pro). Согласно балльному сравнению, наилучшими показателями обладает Hex-Rays, хотя все утилиты достаточно плохо справляются с задачей корректного восстановления типов данных. В качестве альтернативы авторы описывают разрабатываемую среду декомпиляции, гипотетически (и отчасти на практике) имеющую преимущества среди указанных конкурентов.

В статье [5] описывается метод восстановительной коррекции МК после его дизассемблирования, который с помощью эмуляторов исправляет некоторые ошибки процесса. В процессе анализа МК представляется с помощью графов управления, схем Янова и сетей Петри. Затем, с помощью ряда метрик определяются участки кода, подверженные обфускации, а также имеющие чрезмерно сложную структуру. После этого выявляются потенциально опасные нарушения логики вычислений, зацикливания и «мертвый код». Также осуществляется поиск недекларированных возможностей на основании ряда признаков.

В работе [6] исследуется возможность разбиения МК на отдельные логически связанные блоки в автоматическом режиме, что, в том числе, позволит выбрать вектор исследования кода (как последовательности переходов между его инструкциями) в интересах поиска уязвимостей. Авторы указывают необходимость определения в процессе статического анализа точек входа в программу и входных данных. Также приводятся варианты использования графов вызовов подпрограмм и их матрицы смежности. С каждым типом уязвимости предлагается связать группу матриц, содержащих количество потенциально опасных участков МК.

Исследование [7] направлено на обнаружение вредоносного кода путем РИ МК. Приводится обобщенная схема РИ, состоящая из статического и динамического анализа, первый из которых осуществляется с применением дизассемблеров (например, IDA Pro), а второй – отладчиков (например, OllyDebugger).

Описывается утилита PEiD, анализирующая программу на предмет определения языка программирования, компилятора, упаковщика и другой информации о ней. Если программа оказывается запакованной, то предлагается ее распаковать путем динамического анализа, а затем применить классический статический анализ для поиска уязвимостей.

Согласно сделанным обзорам, существует достаточно ограниченное количество публикаций, посвященных непосредственно статическими методам и средствам обработки МК, и восстановлению из него информации, подходящей для экспертного анализа; при этом, в основном, из МК рассматривается возможность получения ассемблерного листинга, форматов данных, а также частично псевдо-ИК и различных графических схем его алгоритмов. Авторские же публикации в данной области среди всех занимают существенное место и направлены на восстановление большего объема метаинформации о МК, включающей также достаточно высокоуровневую – архитектуру и концептуальную модель программы (или их системы).

Схема Методологии

Онтологическая модель

Для точности описания Методологии ранее в [1] была предложена онтологическая модель предметной области; опуская ее детали и графическую схему, кратко опишем введенные таким образом основные сущности (которые далее будут писаться с большой буквы):

1) Машинный код – представление логики работы программ в виде инструкций процессора;

2) Программа – бинарный файл с МК и дополнительной информацией;

3) Программная система – совокупность взаимодействующих Программ;

4) Образ – представление Программной системы в виде монолитного файла;

5) Устройство хранения – устройство для получения и выполнения загруженного Образа;

6) Метаинформация – сведения о функционале МК (псевдокод, алгоритмы, архитектура, концептуальная модель и т. п.);

7) Реверс-инжиниринг – процесс восстановления Метаинформации о МК;

8) Уязвимость – отличие реализации МК от задуманной или заявленной, приводящее к реализации угроз.

Таким образом, онтологическая модель охватывает всю область РИ, вплоть до уязвимостей в МК.

Графическое представление

Предлагаемая Методология состоит из 4 этапов, шаги 1-го из которых были детально расписаны в предыдущей статье цикла, а 2-й, наиболее объемный, будет описан в текущей. Как и раньше, идентификаторы шагов этапов имеют следующую запись: $X.Y$, где X – номер этапа, Y – порядковый номер выполнения в рамках этапа. Схема Методологии с отражением 2-го этапа в графическом виде представлена на рисунке 1.

На схеме используются следующие обозначения: прямоугольник с белым фоном – результат применения шага; круг со светло-желтым фоном – идентификатор шага в текущей статье цикла; круг с темно-желтым фоном – идентификатор шага в предыдущей статье цикла; сплошная стрелка – основное действие шага над результатом предыдущего; пунктирная стрелка – дополнительное использование шагом результатов другого шага; прямоугольники с закругленными краями и надписью – группы логически связанных элементов (синий фон – для текущего этапа, серый фон – для последующего этапа); пунктирный прямоугольник с надписью – область шагов определенного этапа. Так, область «Этап 1. Подготовка объекта исследования» соответствует шагам (кроме 2.7 и 2.8), описанным в предыдущей статье цикла, область «Этап 2, 3. Анализ объекта исследования» – шагам, описанным в текущей (для статического исследования) и последующей (для динамического исследования) статьях, область «Этап 4. Документирование» – шаги также для последующей статьи цикла.

Согласно схеме (см. рисунок 1), в данной части статьи будут расписаны Шаги с 2.1 по 2.14; они полностью определяют 2-й этап РИ и предназначены для анализа МК статическими способами. Опишем далее эти шаги более детально.

Этап 2. Статическое исследование

Этап предназначен для выполнения основного анализа МК без его непосредственного выполнения – что отражает статичность исследования. Для каждого шага будет дан пример его применения на практике, используя, по возможности, рассмотренный в предыдущей статье цикла DXE-драйвер UEFI-фазы *BDS* (*аббр. от англ.* Boot Device Selection, *перев. на русс.* Выбор Загрузочного Устройства) [8] с именем *BdsDxe.efi* из материнской платы *PRIME Z490-A* производства *ASUSTeK COMPUTER INC*. Также, примеры будут считаться типовыми без учета специфичных ситуаций, например, когда секции во вредоносной программе имеют заведомо неверный тип для противодействия антивирусным средствам.

Шаг 2.1. Анализ МК для выделения точек входа, секций кода и данных

Шаг является первым для этапа статического исследования, поскольку с него начинается основная ветка РИ МК, полученного из каждой Программы, расположенной в Программной системе Образа, выполняемого на Устройстве хранения. Входом шага являются результаты применения Этапа 1.

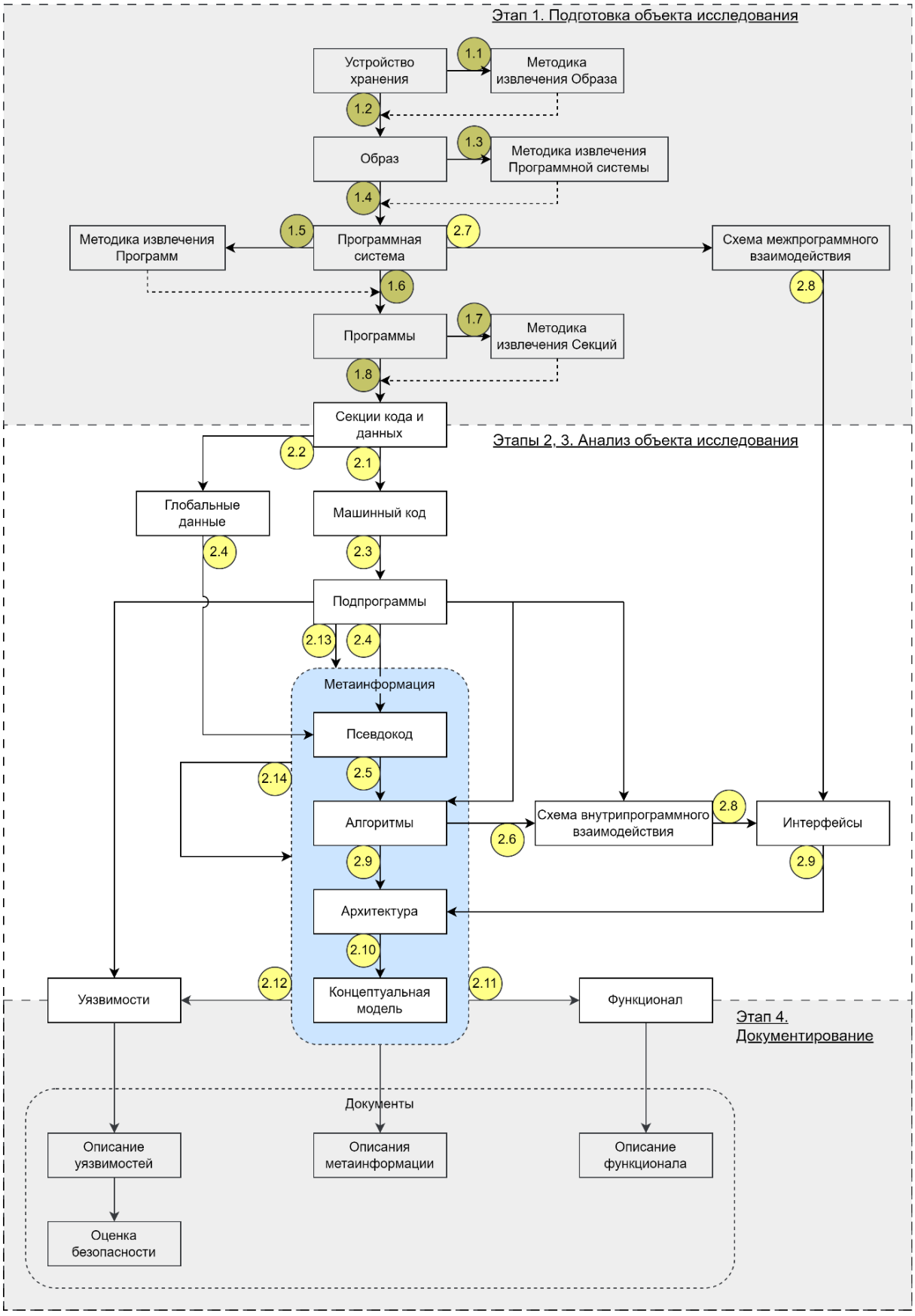

Рис. 1. Схема Методологии реверс-инжиниринга (Этап 2)

Fig. 1. Reverse Engineering Methodology Diagram (Stage 2)

Задачей шага является получение из программных секций с кодом и данными (далее – Секций) содержащегося в них МК. Поскольку Секции, выделенные на Шагах 1.7 и 1.8, имеют определенный тип, определяющий хранящиеся в них данные, то задачей шага является выбор нужных секций и экспорт из них МК.

Формальная запись шага ($Step_{2.1}$) имеет следующий вид:

$$\begin{cases} MachineCodes = Step_{2.1}(Sections) \\ MachineCodes \equiv \{MachineCode_i\} \\ Sections \equiv \{Section_i\} \\ Class^{Section} \epsilon \{Section^{Code}, Section^{Data}\} \end{cases},$$

где $MachineCodes$ – множество блоков с МК, полученных из Секций на шаге; $MachineCode_i$ – i-й блок с МК из множества (для упрощения индекс i далее опустим); $Sections$ – множество секций, полученных из исследуемой Программы на Шаге 1.8; $Section_i$ – i-я секция из множества (для упрощения индекс i далее опустим); $Class^{Section}$ – класс секции, которая содержит код ($Section^{Code}$) или данные ($Section^{Data}$).

Как правило, Секция с выполняемым кодом имеет название «.text», поэтому в большинстве случаев получение МК может заключаться в ее выборе в интерактивном дизассемблере.

Примером шага может быть открытие в продукте IDA Pro драйвера BdsDxe.efi, в результате чего отобразится 7 Секций (называемых также сегментами), одна из которых имеет имя «.text» и класс «CODE», и при переходе на которую будет отображаться содержимое ее МК (рисунок 2).

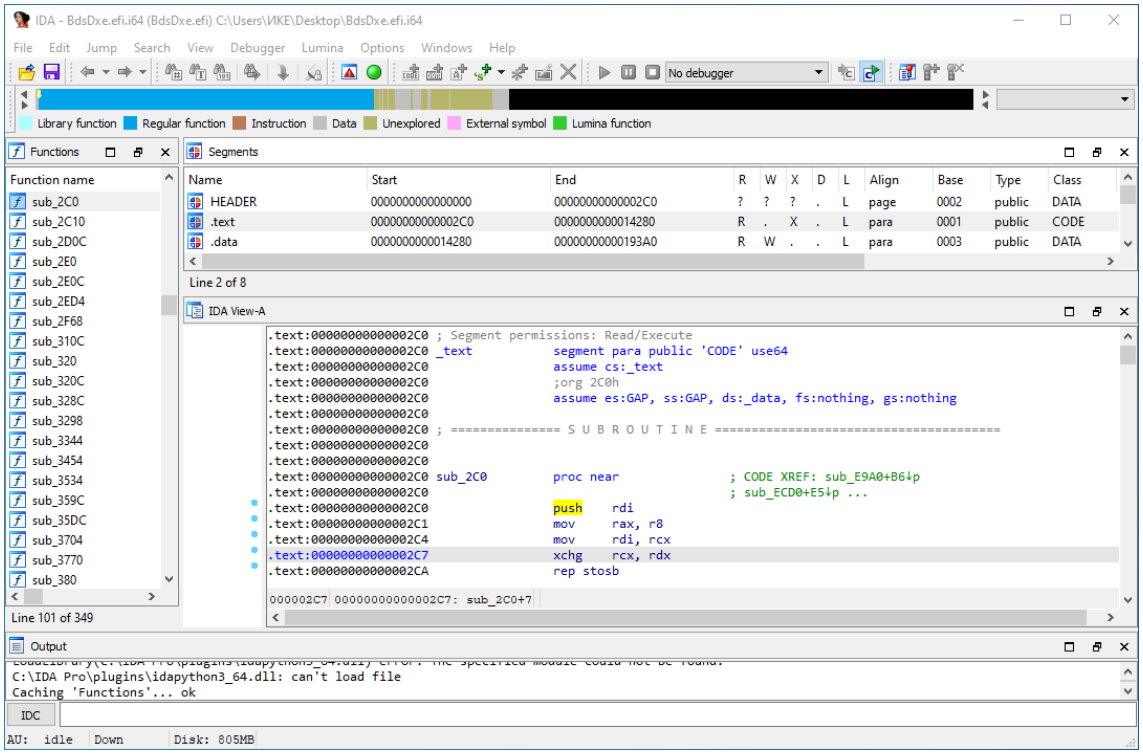

Рис. 2. Пример окон Segments и IDA View утилиты IDA Pro (для Секции кода)

Fig. 2. Segments and IDA View Windows Example of the IDA Pro Utility (for the Code Section)

Согласно рисунку 2, при выборе Секции «.text» с началом по 0x2C0 адресу, во втором окне отображается МК подпрограммы *sub_2C0()*, начинающейся с такого же адреса. Отметим, что точка входа в Программу – т. е. адрес первой выполняемой инструкции МК, также содержится в этой секции.

Шаг 2.2. Анализ секции данных для выделения глобальных данных

Задачей шага является получение из Секций, содержащихся в них глобальных данных, к которым могут быть отнесены такие объекты, как глобальные переменные, константы и пр. Аналогично Шагу 2.1, каждая Секция имеет свой тип, что может быть учтено на данном шаге.

Формальная запись шага ($Step_{2.2}$) имеет следующий вид:

$$\begin{cases} GlobalDatas = Step_{2.2}(Sections) \\ GlobalDatas \equiv \{GlobalData_i\} \end{cases},$$

где $GlobalDatas$ – множество элементов глобальных данных, полученных из Секций на шаге; $GlobalData_i$ – i-й объект глобальных данных из множества (для упрощения индекс i далее опустим).

Как правило, основная Секция с глобальными данными имеет название «.data» (помимо дополнительных, таких, как «.rsrc» для ресурсов, «.xdata» для исключений и т. п.), поэтому в большинстве случаев их получение может заключаться в выборе нужной Секции в интерактивном дизассемблере.

Примером шага может быть продолжение исследования драйвера BdsDxe.efi в IDA Pro, для которого секция «.data» расположена по адресам 0x14280–0x193A и содержит, в том числе, следующие строковые константы (и их адреса):

```
EFI_WARN_UNKNOWN_GLYPH (0x14A10),
EFI_WARN_DELETE_FAILURE(0x14A2A),
EFI_WARN_WRITE_FAILURE (0x14A44);
```

отображение данной информации в IDA Pro представлено на рисунке 3.

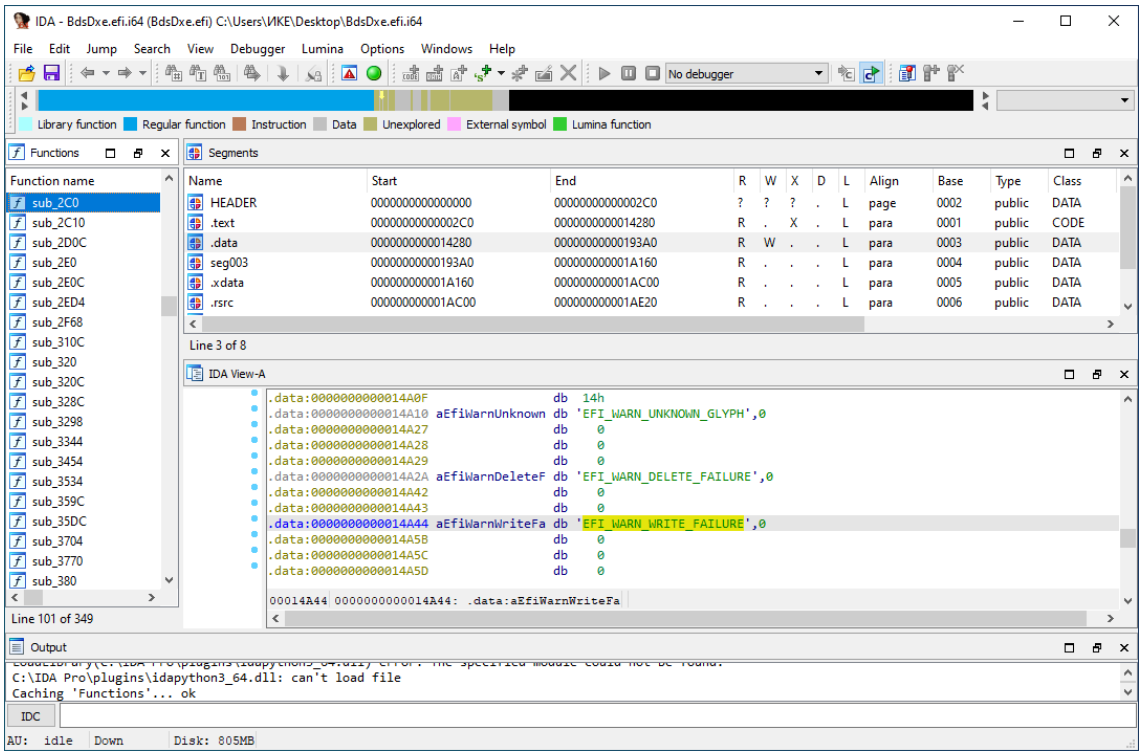

Рис. 3. Пример окон Segments и IDA View утилиты IDA Pro (для Секции данных)

Fig. 3. Segments and IDA View Windows Example of the IDA Pro utility (for the Data Section)

Согласно рисунку 3, при выборе Секции «.data» можно отобразить все глобальные данные Программы для их последующего анализа.

Шаг 2.3. Анализ МК для выделения подпрограмм

Шаг заключается в анализе МК для его деления на отдельные подпрограммы – т. е. многократно исполняемые участки инструкций; как правило, их вызов осуществляется путем перехода на первую инструкцию подпрограммы, а последняя инструкция содержит возврат в точку вызова. Традиционно, такие подпрограммы решают некоторую задачу, используя для этого входные параметры и глобальные переменные, а также возвращая некоторый результат. Если подпрограмма возвращает данные, то она называется *функцией*, иначе – *процедурой*.

Формальная запись шага ($Step_{2.3}$), где $Subroutines$ – множество подпрограмм, полученных из МК на шаге; $Subroutine_i$ – i-я подпрограмма из множества (для упрощения, индекс i далее опустим); $Class^{Subrooutine}$ – класс подпрограммы, соответствующий функции ($Subroutine^{Function}$) или процедуре ($Subroutine^{Procedure}$), имеет вид (1).

$$\begin{cases} Subroutines = Step_{2.3}(MachineCode) \\ Subroutines \equiv \{Subroutine_i\} \\ Class^{Subrooutine} \in \{Subroutine^{Function},\ Subroutine^{Procedure}\} \end{cases}. \quad (1)$$

Задача выделения в МК подпрограмм не является тривиальной, хотя существуют достаточно эффективные ее решения, как правило, внедренные в продукты для дизассемблирования. Примером шага может быть использование утилиты IDA Pro при анализе секций с МК. На рисунке 2 в левой

части экрана присутствует окно «Function name», содержащее все обнаруженные подпрограммы. Большинству из них даны автоматически сгенерированные имена в формате *sub_ADDR*, где *ADDR* – адрес начала подпрограммы (на рисунке 2 отображается их перечень по адресам с 0x2C0 по 0x380).

Шаг 2.4. Анализ МК выбранной подпрограммы для восстановления ее псевдокода

Шаг является наиболее сложным с научной и практической точки зрения, поскольку полноценного решения задачи восстановления ИК (или близких аналогов) из МК не существует. Основная причина этого заключается в том, что при компиляции – т. е. получении МК из ИК, основная часть Метаинформации, с которой работает программист (имена переменных и классов, логика выполнения, формулы вычислений, общая архитектура и т. п.), теряется. Восстановление же этой информации возможно по косвенными признакам; например, если в МК осуществляется многократный возврат выполнения на одну из предыдущих команд, то это может означать наличие логической конструкции языка программирования – *цикла*. Также, если множество подпрограмм расположены по близким адресам и делают взаимные вызовы друг другу, то это может означать наличие конструкции высокоуровневого языка – *экземпляра класса* (или даже логической конструкции архитектуры Программы – модуля).

Для улучшения качества восстановления псевдокода (т. е., по сути, его соответствия или компилируемости в МК) целесообразно использовать и глобальные данные, полученные на Шаге 2.2. Так, если в МК подпрограммы осуществляется отладочный вывод с помощью специальной процедуры *DebugPrint (name, message),* где *name* – имя информирующего модуля, а *message* – сообщение с информацией, то передача в качестве параметров строковых констант из области глобальных данных позволит понять назначение этой подпрограммы. Например, если name ссылается на строку «LoginModule» (*перев. на русс.* Модуль авторизации), а message на «Invalid password» (*перев. на русс.* Неверный пароль), то с большой вероятностью подпрограмма, выводящая такую отладочную информацию, расположена в функционале по проверке пользовательского пароля при входе в систему. Формальная запись шага (*Step_2.4*), где *PseudoCode* – псевдокод, полученный из МК на шаге, имеет следующий вид:

$$PseudoCode = Step_{2.4}(Subroutine, GlobalDatas).$$

Соответственно, с помощью шага все части МК преобразуются в единый псевдокод, где $PseudoCodes$ – множества псевдокода для всех подпрограмм МК; $PseudoCode_i$ – i-й псевдокод из множества (для упрощения, i далее опустим):

$$PseudoCodes = \{PseudoCode_i\}.$$

Примером шага может быть применение достаточно редких и специализированных для этого утилит декомпиляции (т. е. осуществляющих обратную к компиляции операцию), таких, как плагин HexRays для IDA Pro [9], Ghidra [10] со встроенным декомпилятором или ряд разработок автора [11, 12]. Так, применение HexRays к подпрограмме *sub_1D88()* из DXE-драйвера BdsDxe.efi, имеющей ассемблерный код (в формате строк, где вначале идет адрес инструкции, а затем ее текстовая запись):

```
; Begin of sub_1D88()
0x1D88  sub     rsp, 38h
0x1D8C  mov     edx, 1010000h
0x1D91  call    sub_6A8C
0x1D96  add     rsp, 38h
0x1D9A  retn
; End of sub_1D88()
```

восстановит следующий псевдокод:

```
__int64 __fastcall sub_1D88(__int64 a1)
{
  return sub_6A8C(a1, 0x1010000i64);
},
```

соответствующий вызову другой подпрограммы *sub_6A8C()*, в которой помимо такого же 1-го параметра добавлен второй – 0x1010000.

Однако, поскольку у этих утилит есть ряд недостатков (включая работоспособность простых блоков МК, спорность понятности получаемого псевдокода, отсутствие поддержки сложных программных конструкций), шаг в основном выполняется вручную штучными высококвалифицированными экспертами.

Шаг 2.5. Анализ псевдокода выбранной подпрограммы для восстановления ее алгоритма

Шаг в той или иной степени имеет различные решения, поскольку получение алгоритма по ИК (а не только псевдокоду) считается достаточно частой задачей при анализе последнего. При этом, поскольку ИК имеет структуру выполнения, близкую к алгоритмам (одинаковые точки входа и выхода, логика условных и безусловных переходов, вызовы других подпрограмм и т. п.), то утилиты создания по нему алгоритмов не являются технически сверхсложными. Вид алгоритмов может быть различным – как классическая блок-схема, так и структурограмма [13], Дракон-схемы [14] или даже собственный, специализированный для этого, псевдокод.

Формальная запись шага ($Step_{2.5}$), где $Algorithm$ – алгоритм, полученный из псевдокода на шаге, имеет следующий вид:

$$Algorithm = Step_{2.5}(PseudoCode).$$

Так, с помощью шага, все множество псевдокода может быть преобразовано в его алгоритмы:

$$Algorithms = \{Algorithm_i\},$$

где $Algorithms$ – множества алгоритмов псевдокода; $Algorithm_i$ – i-й алгоритм из множества (для упрощения индекс i далее опустим).

Примером шага может быть применение различных генераторов блок-схем алгоритмов по коду [15], одним из которых является Visustin (на Интернет-ресурсе https://www.aivosto.com/visustin.html), поддерживающий ИК на 49 языках программирования. Естественно, ручное восстановление алгоритма экспертом также считается подходящим способом, имеющим свои достоинства и недостатки перед автоматическими программными.

В качестве условной альтернативы могут применяться IDA Pro или Ghidra, поскольку утилиты позволяют выводить граф потока управления подпрограммы, который тождественен ее алгоритму, но содержащий в элементах блоки МК вместо псевдокода. И хотя такой вид существенно менее понятен эксперту, однако он все же дает определенное представление об алгоритме подпрограммы.

Необходимо отметить, что процесс запутывания кода для понимания (т. н. обфускация) может применяться не только к МК, но и к ИК или же алгоритмам [16], что существенно усложняет задачу восстановления последних.

Шаг 2.6. Анализ алгоритмов подпрограмм для создания схемы внутрипрограммного взаимодействия

Шаг заключается в анализе всех алгоритмов одной Программы для построения схемы их обмена информационными объектами (значениями переменных, включая массивы и структуры) – т. е. внутри программного взаимодействия. Так, если один алгоритм вызывает другой (посредством соответствующей подпрограммы), имеющий аргументы и возвращающий значения, то можно говорить о полноценном двухстороннем внутрипрограммном взаимодействии. Отсутствие аргументов и возвращаемых значений у второго алгоритма соответствует вырожденному случаю взаимодействия посредством «пустых информационных объектов».

Формальная запись шага ($Step_{2.6}$) имеет следующий вид:

$$Scheme^{IntoProgramInteraction} = Step_{2.6}(Algorithms),$$

где $Scheme^{IntoProgramInteraction}$ – схема, описывающая взаимодействие алгоритмов в Программе.

И хотя такая схема близка к графу вызовов подпрограмм, построение которого является достаточно тривиальной задачей для любого анализатора ИК и МК, однако полноценный учет всех входных и выходных параметров требует проведения дополнительного анализа. Такой граф вызовов, отображаемый в IDA Pro в окне WinGraph32, приведен на рисунке 4. Граф содержит взаимные вызовы подпрограмм (с алгоритмами), но без указания передаваемых и возвращаемых значений; для этого требуется дополнительный анализ логики алгоритмов и точек их взаимного вызова.

Шаг 2.7. Анализ Программной системы для создания схемы межпрограммного взаимодействия

Шаг заключается в анализе всех Программ в Программной системе для построения схемы их обмена информационными объектами (параметрами вызовов, содержимым внешних файлов и т. п.) – т. е. межпрограммного взаимодействия [17]. Такой процесс был описан автором ранее в виде соответствующей модели [18], построенной на следующих 5 типах файлов: PE-, ELF-, JBC-, CIL- и скриптовых Программах; а также, на 3 типах взаимодействий: прямом вызове, прямом импорте и косвенном обмене. Так, если IDA Pro вызывает Программу WinGraph32 (в операционной системе Windows), то взаимодействие происходит между двумя PE-Программами путем прямого вызова; если же IDA Pro посредством выполнения пользовательских скриптов создает внешний файл, который затем считывается Python-скриптом, то взаимодействие происходит между PE- и скриптовой Программами путем косвенного обмена.

Формальная запись шага ($Step_{2.7}$) имеет следующий вид:

$$\begin{cases} Scheme^{InterProgramInteraction} = Step_{2.7}(Programs) \\ Programs \equiv \{Program_i\} \end{cases},$$

где $Scheme^{InterProgramInteraction}$ – схема, описывающая межпрограммное взаимодействие в Программной системе; $Programs$ – множества Программ; $Program_i$ – i-я Программа из множества (для упрощения индекс i далее опустим).

Ранее автором (с коллегами) была разработана утилита построения таких схем для PE-Программ и их прямого импорта. Пример визуализации схемы с помощью утилиты Gephi [19] для продукта Git в ОС Windows 10 приведен на рисунке 5. Анализ позволяет сделать вывод, что схема содержит основной центральный узел – для библиотеки kernel32.dll, ряд вторичных центров – для библиотек msvcrt.dll, user32.dll, advapi32.dll, kernel32.dll, а также обособленную группу точек – для библиотеки mscoree.dll (отмечено на рисунке пунктирным квадратом).

Шаг 2.8. Анализ схемы внутрипрограммного и межпрограммного взаимодействия для выявления интерфейсов

Шаг состоит из трех следующих действий. Во-первых, в анализе общей схемы обмена информационными объектам между Программами для определения связывающих их частных каналов; например, скрипт может вызывать PE-Программы с передачей им определенных аргументов командной строки.

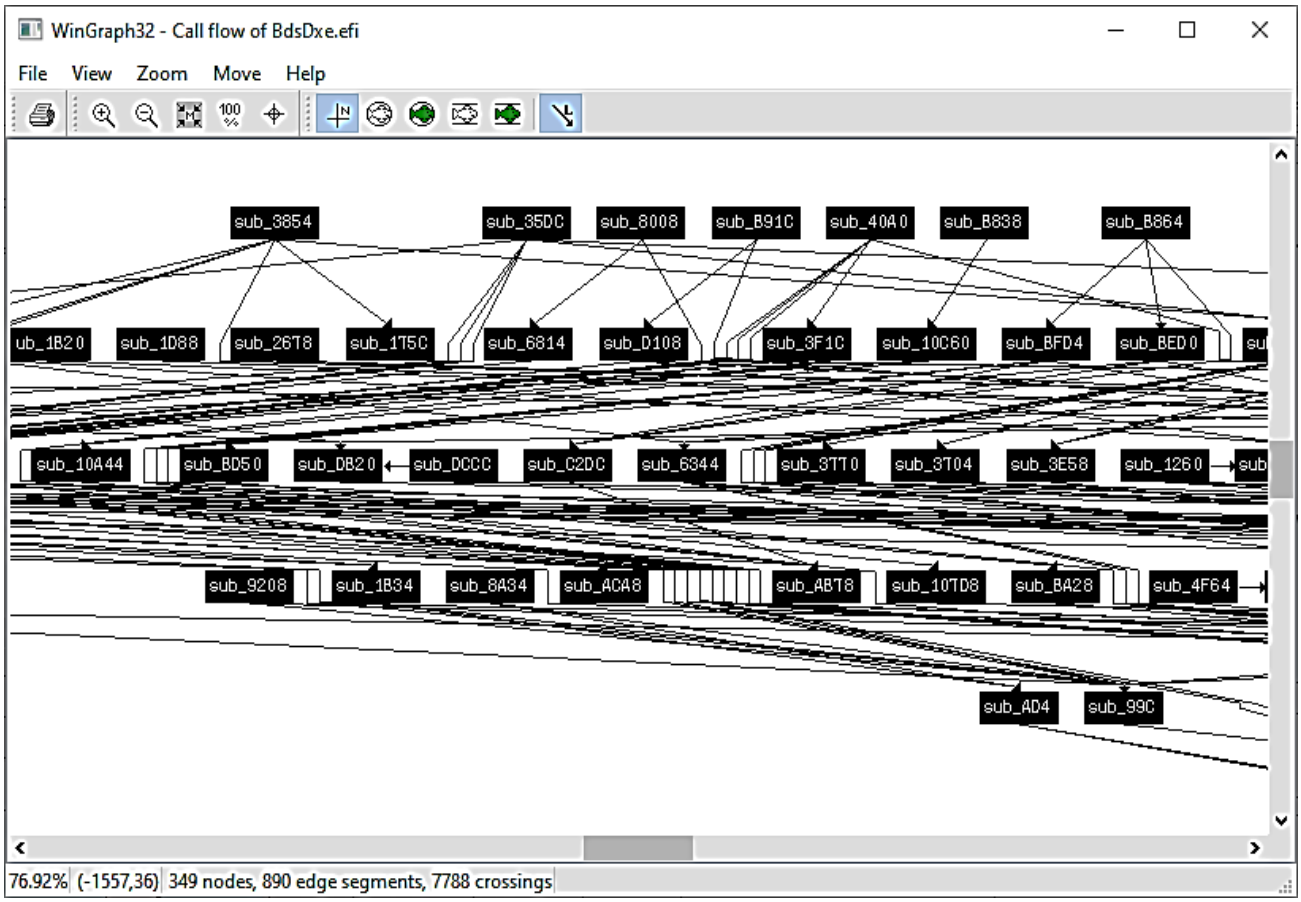

Рис. 4. Пример окна WinGraph32 в утилите IDA Pro

Fig. 4. WinGraph32 Window Example in the IDA Pro Utility

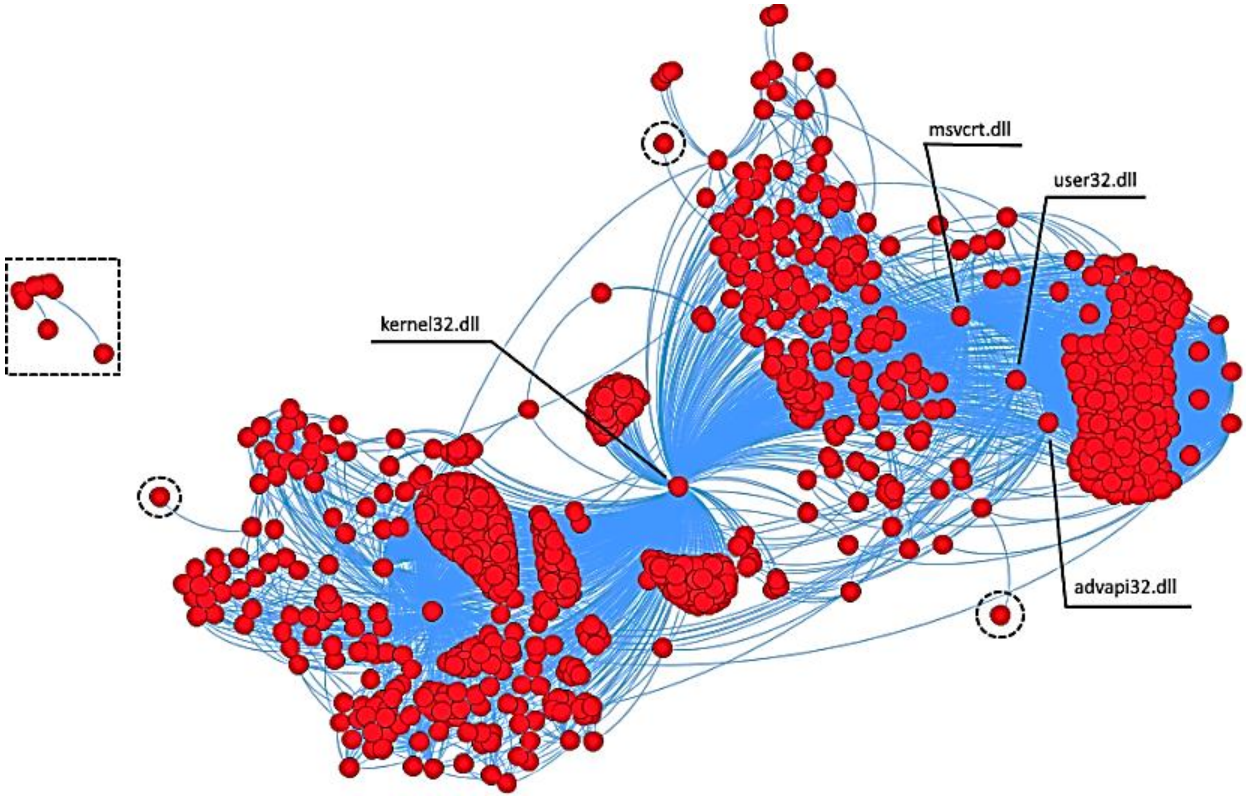

Рис. 5. Пример визуализации межпрограммного взаимодействия Программ в продукте Git

Fig. 5. Visualization of Inter-Program Interaction Example of Programs in the Git Product

Во-вторых, в анализе схемы обмена объектами между подпрограммами в рамках каждой Программы для определения ее входных и выходных частных каналов связи с окружением; например, Программа может анализировать аргументы командной строки и выдавать результат на консоль. И, в-третьих, в формировании набора обобщенных информационных каналов (из частных), которые бы связывали подпрограммы одной Программы с подпрограммами другой; такие каналы будем отождествлять с понятием интерфейсов, поскольку они в некотором смысле размывают границы между Программами и позволяют оперировать лишь множеством подпрограмм во всей Программной системе. Естественно, часть интерфейсов будет «смотреть» наружу из Программной системы, поскольку предназначаются для взаимодействия с «внешним миром» – пользователями, сетью и т. д.

В классической Программе, выполняемой в операционной системе (далее – ОС), входной частью интерфейса является точка входа (указанная в заголовках PE и ELF) – тот адрес МК, который начинает выполняться при запуске; данная подпрограмма получает на вход список аргументов командной строки при запуске, а на выходе возвращает статус завершения работы. В DXE-драйверах присутствует аналогичная подпрограмма, но принимающая на вход указатели на служебные объекты.

Также могут быть и неявные каналы взаимодействия с Программой, такие, как чтение или запись данных во внешний файл. Потенциальным признаком такой подпрограммы, связанной с интерфейсом, является ее терминальность в схеме внутрипрограммного взаимодействия – если она не вызывается другими подпрограммами, то в нее могут передаваться входные данные; аналогично, если она не вызывает другие подпрограммы, то через нее могут возвращаться данные из Программы.

Формальная запись шага ($Step_{2.8}$), где $Interfaces$ – множества интерфейсов Программ; $Interface_i$ – i-й интерфейс из множества (для упрощения индекс i далее опустим), имеет вид (2).

$$\begin{cases} Interfaces = Step_{2.8}(Scheme^{IntoProgramInteraction}, Scheme^{InterProgramInteraction}) \\ INterfaces \equiv \{Interface_i\} \end{cases}. \quad (2)$$

Примером шага может быть сбор информации о точках входа в Программу (указанных в поле AddressOfEntryPoint для PE-заголовка и e_entry для ELF-заголовка), поиска листьев в графовидной схеме внутрипрограммного взаимодействия, анализа кода подпрограмм на предмет наличия API-функций файловой системы (fopen() для открытия файла в стандартной библиотеке языка С) и работы с сетью (socker() для открытия сокета) и др.

Шаг 2.9. Анализ алгоритмов подпрограмм и интерфейсов для восстановления их общей архитектуры

Шаг заключается в анализе алгоритмов для понимания их задача-ориентированности, что позволит объединить близкие с этой позиции подпрограммы в логические единицы архитектуры – модули; связи же между модулями будут определяться внутренним взаимодействием подпрограмм. Дальнейшее же объединение модулей в более иерархические высокоуровневые сущности сформирует остальные логические слои архитектуры вплоть до требуемого уровня абстракции.

Анализ интерфейсов между Программами (если их несколько) позволит соединить модули, полученные из разных Программ, в единую архитектуру, отражающую уже не отдельную Программу, а их целую Программную систему. Графическая иллюстрация такого способа взаимодействия подпрограмм в Программной системе – на рисунке 6.

Для наглядности представления архитектуры предположим, что в Программе типа «арифметический калькулятор» были выделены и интерпретированы 8 подпрограмм со следующими задачами: сложение, умножение, вычитание и деление чисел; ввод чисел; вывод результата; логирование работы; сообщение об ошибке.

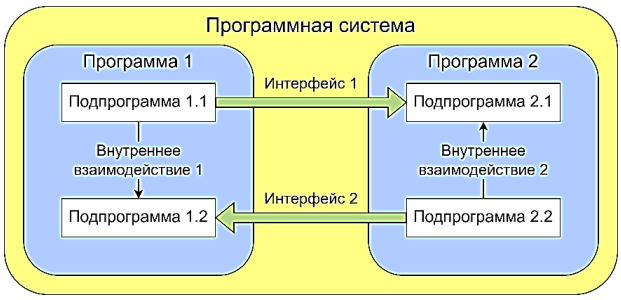

Рис. 6. Графическая иллюстрация способа взаимодействия подпрограмм в Программной системе

Fig. 6. Graphic Illustration of the Interaction Subroutines Method in the Software System

Тогда на данном шаге первые 4 подпрограммы могут быть объединены в модуль «Операции», подпрограммы 5 и 6 – в модуль «Взаимодействие с пользователем»; а 7 и 8 – в модуль «Системный функционал». Затем первые 2 модуля можно объединить в более высокоуровневый модуль – «Основной функционал», построив таким образом иерархическую архитектуру. При этом в архитектуре будут отмечены входной и выходной интер-

фейсы к пользователю – в модуле с подпрограммами 5 и 6, а также выходной интерфейс к ОС – в модуле с подпрограммами 7 и 8. Необходимо заметить, что приведенный способ записи архитектуры является одним из возможных [20], поскольку существуют более сложные представления, например, с применением UML; однако, здесь и далее будем рассматривать именно такой простой вид архитектуры.

Формальная запись шага ($Step_{2.9}$), состоящая из двух частей – получение частной архитектуры каждой Программы ($Step_{2.9}.1$) и общей архитектуры всей Программной системы ($Step_{2.9.2}$), имеет следующий вид:

$$\begin{cases} Architecture = Step_{2.9.1}(Subroutine) \\ Architectures \equiv \{Architecture\} \\ \overline{Architecture} = Step_{2.9.2}(Architectures, Interfaces) \end{cases},$$

где $Architecture$ – частная архитектура МК отдельной Программы, полученная на шаге; $Architectures$ – множества архитектур для всех Программ; $\overline{Architecture}$ – общая архитектура всей Программной системы, полученная на шаге.

Примером шага может быть построение одноуровневой архитектуры для DXE-драйвера *BdsDxe.efi* в виде совокупности 3-х модулей: «BDS-фаза» – реализующего логику одноименной UEFI фазы, «Поддержка языка» – поддерживающего настройки языка интерфейса, и «Запись аппаратных ошибок» – устанавливающего уровень поддержки записей об аппаратных ошибках. Соответственно, точка входа в Программу расположена в 1-м модуле, который вызывает оставшиеся 2.

Необходимо отметить, что в случае наличия лишь одной Программы интерфейсы между Программами будут отсутствовать, а, следовательно, частная и общая архитектуры окажутся тождественны: $Architecture \equiv \overline{Architecture}$.

Шаг 2.10. Анализ архитектуры для восстановления ее концептуальной модели

Шаг заключается в анализе модулей, их связей и интерфейсов (т. е. архитектуры) Программной системы для восстановления ее концептуальной модели, под которой понимается совокупность взаимосвязанных основополагающих сущностей, представляющая «смысл» всей системы с высокоуровневой позиции. В принципе, концептуальная модель близка к онтологической, хотя и предназначена для отражения специфики конкретной Программы, а не всей предметной области.

Формально запись текущего шага ($Step_{2.10}$), где $Model^{Conceptual}$ – концептуальная модель Программы, имеет следующий вид:

$$Model^{Conceptual} = Step_{2.10}(\overline{Architecture}).$$

Примером шага для архитектуры *BdsDxe.efi* может быть обобщение модулей архитектуры и получение совокупности следующих элементов концепции – «Человек», «BDS-фаза» и «Настройки модуля»; при этом 1-й элемент связан со 2-м, который связан с 3-м. По аналогии с архитектурой, в случае наличия одной Программы концептуальная модель, будет относиться только к ней.

Шаг 2.11. Анализ восстановленной Метаинформации для описания функционала МК

Шаг заключается в составлении итогового функционала МК [21] на основании анализа Метаинформации различного типа, восстановленной в результате предыдущих шагов – псевдокода после 2.4, алгоритмов после 2.5, архитектуры после 2.9 и концептуальной модели после 2.10. Таким образом, будет получен основной результат исследования, при этом касающийся не только данного этапа, а всего РИ, поскольку последующие этапы носят уточняющий и документирующий характер. В простейшем случае описание функционала может быть перечислением внутренних действий Программ (по алгоритмам их подпрограмм), некоторых деталей их реализации (по псевдокоду), а также решаемых задач (по архитектуре), которые сформулированы на едином базисе сущностей (по концептуальной модели).

Формальная запись шага ($Step_{2.11}$), где $Metainfo$ – все Метаинформация, восстановленная из МК Программ на шаге, имеет вид (3).

$$\begin{cases} Functionality = Step_{2.11}(Metainfo) \\ Metainfo \equiv \langle PseudoCodes, Algorithms, \overline{Architecture}, Model^{Conceptual} \rangle \end{cases}. \quad (3)$$

Примером шага для DXE-драйвера *BdsDxe.efi* может быть анализ архитектуры из связанных с задачами модулей – «BDS-фаза», «Поддержка языка», «Запись аппаратных ошибок», которые согласно концептуальной модели обеспечивают взаимодействие пользователя с BDS-фазой UEFI, работающей согласно настройкам (из NVRAM); а все детали решения этих задач могут быть сформулированы на основании логики работы конкретных подпрограмм и их псевдокода. В результате будет получено человеко-ориентированное описание драйвера, хорошо понятное эксперту.

Шаг 2.12. Поиск Уязвимостей по восстановленной Метаинформации

Шаг заключается в анализе восстановленной Метаинформации на предмет наличия в ней Уязвимостей, поскольку одним из целевых назначений РИ

является проверка Программы на предмет ее безопасности. Так, каждый тип Метаинформации связан со своим структурным уровнем Уязвимости [22]: псевдокод – с низким, алгоритм – со средним, архитектура – с высоким и концептуальная модель – со сверхвысоким или концептуальным. И хотя для поиска Уязвимостей уже существует ряд программных средств, однако их наибольшая результативность оказывается для низкоуровневых и частично среднеуровневых Уязвимостей; для поиска остальных классов Уязвимостей, как правило, требуется привлечение экспертов по безопасности программного кода.

Формальная запись шага ($Step_{2.12}$) имеет следующий вид:

$$\begin{cases} Vulnerabilities = Step_{2.12}(Metainfo) \\ Vulnerabilities \equiv \{Vulnerability_i\} \\ Class_{Vulnerability} \in \begin{Bmatrix} Vulnerability^{LowLevel}, \\ Vulnerability^{MediumLevel}, \\ Vulnerability^{HighLevel}, \\ Vulnerability^{ConceptualLevel} \end{Bmatrix} \end{cases},$$

где $Vulnerabilities$ – множества Уязвимостей, обнаруженных на шаге; $Vulnerability_i$ – i-я Уязвимость из множества (для упрощения, индекс i далее опустим); $Class_{Vulnerability}$ – класс, связанный с Уязвимостями согласно их структурного уровня:

– низкоуровневой ($Vulnerability^{LowLevel}$);
– среднеуровневой ($Vulnerability^{MediumLevel}$);
– высокоуровневой ($Vulnerability^{HighLevel}$);
– концептуальной ($Vulnerability^{ConceptualLevel}$).

Примером шага для *BdsDxe.efi* в виде Уязвимостей, которые обнаруживаемы по различной Метаинформации, может быть следующий. Применение переменных меньшего размера, чем возможные значения, в псевдокоде могут привести к хранению неверных настроек BDS-фазы. Ошибки в логике выбора наилучшего раздела для загрузки ОС после BDS фазы могут привести к невозможности запуска всей системы. Отсутствие связи модуля BDS-фазы с настройками будет означать невозможность управлять логикой ее работы. Если в Образ материнской платы внедрен модуль доверенной загрузки, выполненный в виде DXE-драйвера [23], то отсутствие в концептуальной модели связи DXE-драйвера *BdsDxe.efi* (который как раз отвечает за выбор загрузочного устройства) с этим модулем скорее всего будет означать небезопасность загрузки ОС, поскольку отсутствует механизм управления ею.

Шаг 2.13. Применение машинного обучения к подпрограммам для дополнительного восстановления Метаинформации

Шаг является достаточно технически сложным и мало затрагиваемым в современных научных публикациях, поскольку заключается в применении моделей и методов машинного обучения для восстановления псевдокода из МК подпрограмм (а также глобальных данных) – т. е. является некой «интеллектуальной» декомпиляцией [24]. Таким образом, уже полученный псевдокод (на Шаге 2.4) может быть дополнен на данном шаге новыми сведениями. Шаг может реализовываться путем классификации различных последовательностей инструкций МК по соответствующим конструкциям в языках программирования среднего и высокого уровня. Также в интересах этого возможно применение авторского метода генетической декомпиляции [25], основанного на использовании генетических алгоритмов для оптимального подбора конструкций исходного кода, в точности компилируемых в исследуемый МК.

Формальная запись шага ($Step_{2.13}$), в которой $PseudoCode'$ – псевдокод подпрограммы, уточненный в процессе статического исследования на шаге; $MachineLearning$ – применяемые для уточнения модели и методы машинного обучения; «→» – операция изменения сущности (в данном случае обновление Метаинформации); «∪» – объединение сущностей (в данном случае Метаинформации и уточнений в псевдокоде), имеет вид (4).

$$\begin{cases} PseudoCode' = Step_{2.13}(Subroutine, GlobalDatas, MachineLearning) \\ Metainfo \rightarrow Metainfo \cup PseudoCode' \end{cases}. \quad (4)$$

Примером шага для подпрограммы *sub_1D88()* из Шага 2.4 может быть подбор с применением генетического алгоритма конструкций единственной строки ее псевдокода таким образом, чтобы она компилировалась в 5 инструкций ее МК.

Шаг 2.14. Применение машинного обучения к восстановленной Метаинформации для ее уточнения

Как и Шаг 2.14, данный шаг является также мало изученным, хотя для восстановления Метаинформации ряд задач, решаемых с помощью машинного обучения, подходят достаточно хорошо. Поскольку основная часть Метаинформации на данный момент РИ уже была восстановлена (в основном ручным способом, хотя и с применением автоматизирующих средств), то шаг носит уточняющий характер – т. е. дополняет или корректирует их, например, следующим образом. Во-первых, типовые наборы конструкций псевдокода могут быть отнесены к более абстрактным конструкциям алгоритмов. Во-вторых, подобная классификация в купе с кластеризацией позволяет выделять близкие подпрограммы (по их алгоритмам) в отдельные модули и их высокоуров-

невые группы [26]. И, в-третьих, аналогичным образом можно получить концептуальную модель Программной системы (или отдельной Программы) из ее архитектуры, определив категории всех верхних модулей и образовав их логические связи.

Формальная запись текущего шага ($Step_{2.14}$), где $Metainfo'$ – Метаинформация, уточненная в процессе статического исследования на шаге; $MachineLearning$ – применяемые для уточнения модели и методы машинного обучения, имеет следующий вид:

$$\begin{cases} Metainfo' = Step_{2.14}(Metainfo, MachineLearning) \\ Metainfo \rightarrow Metainfo \cup Metainfo' \end{cases},$$

Пример шага для DXE-драйвера *BdsDxe.efi* может заключаться в следующем. Во-первых, псевдокод для подпрограммы *sub_1D88()*, состоящий из вызова другой подпрограммы *sub_6A8C()* с двумя параметрами, может быть с помощью машинного обучения проклассифицирован как следующая конструкция блок-схемы алгоритма – «вызов предопределенного процесса (функции) с передачей параметров». Во-вторых, множество подпрограмм, вызывающих друг друга и расположенных в близком диапазоне адресов, могут быть отнесены к одному модулю архитектуры Программы (как частной архитектуре Программной системы); вызовы же подпрограмм между различными диапазонами будут означать связи между модулями, а их названия (или описания) могут быть сформированы из совокупности строковых констант в МК этих модулей (например, с помощью моделей Word2Vec [27]). И, в-третьих, сущности концептуальной модели их взаимосвязи могут быть получены путем категорирования названия (или описания) модулей архитектуры с учетом их иерархии.

Заключение

В данной части цикла статей был описан 2-й этап методологии реверс-инжиниринга, являющийся наиболее технически сложным. Для этого, в частности, сделан обзор существующих методов и средств статического анализа машинного кода.

Новизна соответствует указанной в первой части цикла и заключается в системности и масштабности охвата решения задачи реверс-инжиниринга. Теоретическая и практическая значимость аналогична, но в отличие от первой части, посвященной подготовке МК, относится к его статическому исследованию.

Продолжением исследования должно стать описание оставшихся 3-го и 4-го этапов методологии реверс-инжиниринга, а также представление итоговой схемы процесса и сведение шагов всех этапов в единую систему.

Окончание следует…

Список источников

1. Израилов К.Е. Методология проведения 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. Падарян В.А., Гетьман А.И., Соловьев М.А., Бакулин М.Г., Борзилов А.И., Каушан В.В. Методы и программные средства, поддерживающие комбинированный анализ бинарного кода // Труды Института системного программирования РАН. 2014. Т. 26. № 1. С. 251–276.

3. 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

4. Долгова К.Н., Чернов А.В., Деревенец Е.О. Методы и алгоритмы восстановления программ на языке ассемблера в программы на языке высокого уровня // Проблемы информационной безопасности. Компьютерные системы. 2008. № 3. С. 54–68.

5. Новиков В.А., Ломако А.Г., Еремеев М.А., Петренко А.С. Выявление и нейтрализация недекларированных возможностей программ // Proceedings of the 2017 Symposium on Cybersecurity of the Digital Economy (CDE'17, Иннополис, Россия, 19–20 сентября 2017). Санкт-Петербург: Издательский Дом «Афина», 2017. С. 284–287.

6. Ревнивых А.В., Велижанин А.С. Методика автоматизированного формирования структуры дизассемблированного листинга // Кибернетика и программирование. 2019. № 2. С. 1–16. DOI:10.25136/2306-4196.2019.2.28272

7. 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 (ICAC3, Mumbai, India, 03–04 December 2021). IEEE, 2022. PP. 1–5. DOI:10.1109/ICAC353642.2021.9697150

8. Черчесов А.Э. Фазы загрузки UEFI и способы контроля исполняемых образов // Вопросы защиты информации. 2018. № 2(121). С. 51–53.

9. Zhang D., Zhang Z., Jiang B., Tse T.H. The Impact of Lightweight Disassembler on Malware Detection: An Empirical Study // Proceedings of the 42nd Annual Computer Software and Applications Conference (Tokyo, Japan, 23–27 July 2018). IEEE, 2018. PP. 620–629, DOI:10.1109/COMPSAC.2018.00094

10. David A.P. Ghidra Software Reverse Engineering for Beginners: Analyze, identify, and avoid malicious code and potential threats in your networks and systems. UK: Packt Publishing Ltd, 2021. 322 p.

11. Буйневич М.В., Израилов К.Е. Автоматизированное средство алгоритмизации машинного кода телекоммуникационных устройств // Телекоммуникации. 2013. № 6. С. 2–9.

12. Буйневич М.В., Израилов К.Е. Метод алгоритмизации машинного кода телекоммуникационных устройств // Телекоммуникации. 2012. № 12. С. 2–6.

13. Селиверстова И.А. Разработка программного обеспечения построения XML описания кода // Современные научные исследования и инновации. 2016. № 2(58). С. 102–104.

14. Митькин С.Б. Автоматное программирование на языке Дракон // Программная инженерия. 2019. Т. 10. № 1. С. 3–13. DOI:10.17587/prin.10.3-13

15. Вохмин А.А., Евдокимова О.А., Малявко А.А. Визуально-графическая система программирования на основе разработки блок-схем алгоритмов. Конвертирование текстов программ на различных языках программирования в блок-схемы и обратно // Южно-Сибирский научный вестник. 2021. № 3(37). С. 49–57. DOI:10.25699/SSSB.2021.37.3.013

16. Pakonen A. Obfuscation of function block diagrams // Proceedings of the 28th International Conference on Emerging Technologies and Factory Automation (ETFA, Sinaia, Romania, 12–15 September 2023). IEEE, 2023. PP. 1–7. DOI:10.1109/ETFA54631.2023.10275363

17. Ипатов П.С. Технологии межпрограммных интерфейсов // Science Time. 2016. № 9(33). С. 115–118.

18. Буйневич М.В., Ганов Г.А., Израилов К.Е. Интеллектуальный метод визуализации взаимодействий программ в интересах аудита информационной безопасности операционной системы // Информатизация и связь. 2020. № 4. С. 67–74.

19. Yang J., Cheng C., Shen S., Yang S. Comparison of complex network analysis software: Citespace, SCI2 and Gephi // Proceedings of the 2nd International Conference on Big Data Analysis (Beijing, China, 10–12 March 2017). IEEE, 2017. PP. 169–172. DOI:10.1109/ICBDA.2017.8078800

20. Gardazi S.U., Shahid A.A. Survey of software architecture description and usage in software industry of Pakistan // Proceedings of the International Conference on Emerging Technologies (Islamabad, Pakistan, 19–20 October 2009). IEEE, 2009. PP. 395–402. DOI:10.1109/ICET.2009.5353137

21. Sharma K., Dubey S.K., Gaurav P., Prachi. Functionality Assessment of Software System using Fuzzy Approach // Proceedings of the 8th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions) (ICRITO, Noida, India, 04–05 June 2020). IEEE, 2020. PP. 1206–1209. DOI:10.1109/ICRITO48877.2020.9197795

22. Kotenko I., Izrailov K., Buinevich M., Saenko I., Shorey R. Modeling the Development of Energy Network Software, Taking into Account the Detection and Elimination of Vulnerabilities // Energies. 2023. Vol. 16. Iss. 13. P. 5111. DOI:10.3390/en16135111

23. Израилов К.Е., Покусов В.В. Создание программной объектно-ориентированной платформы для разработки UEFI модулей // X международная научно-техническая и научно-методическая конференция «Актуальные проблемы инфотелекоммуникаций в науке и образовании» (АПИНО 2021, Санкт-Петербург, Россия, 24–25 февраля 2021). Санкт-Петербург: СПбГУТ. 2021. Т. 2. С. 246–250.

24. Yu S.-Y., Achamyeleh Y.G., Wang C., Kocheturov A., Eisen P., Al Faruque M.A. CFG2VEC: Hierarchical Graph Neural Network for Cross-Architectural Software Reverse Engineering // Proceedings of the 45th International Conference on Software Engineering: Software Engineering in Practice (Melbourne, Australia, 14–20 May 2023). IEEE, 2023. PP. 281–291. DOI:10.1109/ICSE-SEIP58684.2023.00031

25. 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

26. Израилов К.Е., Умаралиев И.В. Гипотетический метод восстановления модулей архитектуры машинного кода с целью выявления высокоуровневых уязвимостей // XII международная научно-техническая и научно-методическая конференция Актуальные проблемы инфотелекоммуникаций в науке и образовании (АПИНО 2023, Санкт-Петербург, Россия, 28 февраля – 01 марта 2023). Санкт-Петербург: СПбГУТ, 2023. Т. 1. С. 577–581.

27. Wang R., Shi Y. Research on application of article recommendation algorithm based on Word2Vec and Tfidf // The Proceedings of International Conference on Electrical Engineering, Big Data and Algorithms (Changchun, China, 25–27 February 2022). IEEE, 2022. PP. 454–457. DOI:10.1109/EEBDA53927.2022.97448244

References

1. Izrailov K. Methodology for Machine Code Reverse Engineering. Part 1. Preparation of the Research Object. *Proceedings of the Telecommun. Univ.* 2023;9(5):79–90. DOI:10.31854/1813-324X-2023-9-5-79-90

2. Padaryan V.A., Getman A.I., Solovev M.A., Bakulin M.G., Borzilov A.I., Kaushan V.V. Methods and software tools supporting combined binary code analysis. *Proceedings of ISP RAS*. 2014;26(1):251–276.

3. Bugerya A.B., Yefimov V.Yu., Kulagin I.I., Padaryan V.A., Solovev M.A., Tikhonov A.Yu. Program complex for detecting undeclared capabilities in the absence of source code. *Proceedings of ISP RAS*. 2019;31(6):33–64. DOI:10.15514/ISPRAS-2019-31(6)-3

4. Dolgova K.N., Chernov A.V., Derevenets Ye.O. Methods and algorithms for restoring assembly language programs into high-level language programs. *Information Security Problems. Computer Systems*. 2008;3:54–68.

5. Novikov V.A., Lomako A.G., Yeremeev M.A., Petrenko A.S. Identification and neutralization of undeclared program features. *Proceedings of the 2017 Symposium on Cybersecurity of the Digital Economy, CDE'17, 19–20 September 2017, Innopolis, Russia.* St. Petersburg: Afina Publ.; 2017. p.284–287.

6. Revnivikh A.V., Velizhanin A.S. Automated Formation Methodology of disassembled listing structure. *Cybernetics and Programming*. 2019;2:1–16. DOI:10.25136/2306-4196.2019.2.28272

7. 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, ICAC3, 03–04 December 2021, Mumbai, India.* IEEE; 2022. p.1–5. DOI:10.1109/ICAC353642.2021.9697150

8. Cherchesov A.E. UEFI boot phases and how to control executable images. *Voprosy zashchity informatsii.* 2018;2(121): 51–53.

9. Zhang D., Zhang Z., Jiang B., Tse T.H. The Impact of Lightweight Disassembler on Malware Detection: An Empirical Study. *Proceedings of the 42nd Annual Computer Software and Applications Conference, 23–27 July 2018, Tokyo, Japan.* IEEE; 2018. p.620–629. DOI:10.1109/COMPSAC.2018.00094

10. David A.P. Ghidra *Software Reverse Engineering for Beginners: Analyze, identify, and avoid malicious code and potential threats in your networks and systems*. Packt Publishing Ltd; 2021. 322 p.

11. Buinevich M.V., Izrailov K.E. Automated tool for machine code algorithmization of telecommunication devices. *Telekommunikatsii*. 2013;6:2–9.

12. Buinevich M.V., Izrailov K.E. Algorithmization method for machine code of telecommunication devices. *Telekommunikatsii*. 2012;12:2–6.

13. Seliverstova I.A. Development of software for building XML code description. Modern scientific researches and innovations. 2016;2(58):102–104.

14. Mitkin S.B. Automata programming in the Dragon language. Software Engineering. 2019;10(1):3–13. DOI:10.17587/prin.10.3-13

15. Vokhmin A.A., Yevdokimova O.A., Malyavko A.A. Visual-graphical programming system based on the development of algorithm flowcharts. Converting program texts in different programming languages into flowcharts and back again. *South-Siberian Scientific Bulletin*. 2021;3(37):49–57. DOI:10.25699/SSSB.2021.37.3.013

16. Pakonen A. Obfuscation of function block diagrams. *Proceedings of the 28th International Conference on Emerging Technologies and Factory Automation, ETFA, 12–15 September 2023, Sinaia, Romania.* IEEE; 2023. p.1–7. DOI:10.1109/ETFA54631.2023.10275363

17. Ipatov P.S. Technologies of interprogram interfaces. *Science Time*. 2016;9(33):115–118.

18. Buinevich M.V., Ganov G.A., Izrailov K.E. An intelligent method for visualizing program interactions in the interest of operating system information security auditing. *Informatization and communication.* 2020;4:67–74.

19. Yang J., Cheng C., Shen S., Yang S. Comparison of complex network analysis software: Citespace, SCI2 and Gephi. *Proceedings of the 2nd International Conference on Big Data Analysis Beijing, 10–12 March 2017, Beijing, China.* IEEE; 2017. p.169–172. DOI:10.1109/ICBDA.2017.8078800

20. Gardazi S.U., Shahid A.A. Survey of software architecture description and usage in software industry of Pakistan. *Proceedings of the International Conference on Emerging Technologies, 19–20 October 2009, Islamabad, Pakistan.* IEEE; 2009. p.395–402. DOI:10.1109/ICET.2009.5353137

21. Sharma K., Dubey S.K., Gaurav P., Prachi Functionality Assessment of Software System using Fuzzy Approach. *Proceedings of the 8th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions), ICRITO, 04–05 June 2020, Noida, India.* IEEE; 2020. p.1206–1209. DOI:10.1109/ICRITO48877.2020.9197795

22. Kotenko I., Izrailov K., Buinevich M., Saenko I., Shorey R. Modeling the Development of Energy Network Software, Taking into Account the Detection and Elimination of Vulnerabilities. *Energies.* 2023:16(13):5111. DOI:10.3390/en16135111

23. Izrailov K.E., Pokusov V.V. Creation of software object-oriented platform for UEFI modules development. *Proceedings of the X International Scientific-Technical and Scientific-Methodical Conference on Actual Problems of Infotelecommunications in Science and Education, 24–25 February 2021, St. Petersburg, Russia.* St. Petersburg: The Bonch-Bruevich Saint-Petersburg State University of Telecommunications Publ.; 2021. vol.2. p.246–250.

24. Yu S.-Y., Achamyeleh Y.G., Wang C., Kocheturov A., Eisen P., Al Faruque M.A. CFG2VEC: Hierarchical Graph Neural Network for Cross-Architectural Software Reverse Engineering. *Proceedings of the 45th International Conference on Software Engineering: Software Engineering in Practice, 14–20 May 2023, Melbourne, Australia.* IEEE; 2023. p.281–291. DOI:10.1109/ICSE-SEIP58684.2023.00031

25. Izrailov K.E. The concept of genetic decompilation machine code telecommunication devices. *Proceedings of the Telecommun. Univ*. 2021;7(4):95–109. DOI:10.31854/1813-324X-2021-7-4-95-109

26. Izrailov K.E., Umaraliev I.V. Hypothetical method of restoring machine code architecture modules in order to detect high-level vulnerabilities. *Proceedings of the XII International Scientific-Technical and Scientific-*Methodical *Conference on Actual Problems of Infotelecommunications in Science and Education, 28 February – 01 March 2023, St. Petersburg, Russia*. St. Petersburg: The Bonch-Bruevich Saint-Petersburg State University of Telecommunications Publ.; 2023. vol.1. p.577–581.

27. Wang R., Shi Y. Research on application of article recommendation algorithm based on Word2Vec and Tfidf. *Proceedings of the International Conference on Electrical Engineering, Big Data and Algorithms, 25–27 February 2022, Changchun, China*. IEEE; 2022. p.454–457. DOI:10.1109/EEBDA53927.2022.9744824

Статья поступила в редакцию 25.10.2023; одобрена после рецензирования 21.11.2023; принята к публикации 22.11.2023.
The article was submitted 25.10.2023; approved after reviewing 21.11.2023; accepted for publication 22.11.2023.

Информация об авторе:

ИЗРАИЛОВ Константин Евгеньевич

кандидат технических наук, доцент, старший научный сотрудник лаборатории проблем компьютерной безопасности Санкт-Петербургского Федерального исследовательского центра Российской академии наук

https://orcid.org/0000-0002-9412-5693