УДК 004.4:004.5:004.8
DOI 10.17513/snt.40771

К ВОПРОСУ О ПРИМЕНЕНИИ LLM-ДЕКОМПИЛЯЦИИ ДЛЯ ПОИСКА УЯЗВИМОСТЕЙ В МАШИННОМ КОДЕ ПРОГРАММ

Буйневич М. В. ORCID ID 0000-0001-8146-0022,
Израилов К. Е. ORCID ID 0000-0002-9412-5693

Федеральное государственное бюджетное образовательное учреждение высшего образования «Санкт-Петербургский университет ГПС МЧС России», Санкт-Петербург, Российская Федерация, e-mail: bmv1958@yandex.ru

Задача декомпиляции, заключающаяся в преобразовании машинного (выполняемого) кода программы в исходный, который более адаптирован для выявления уязвимостей, является крайне востребованной в сфере обеспечения безопасности программного обеспечения. Для ее решения рассматривается возможность применения искусственного интеллекта в части больших языковых моделей (Large Language Model, LLM). Цель исследования – установление границ применимости LLM для противодействия уязвимостям программного обеспечения, представленного машинным кодом. Проводится серия экспериментов (из 5 «прогонов») по декомпиляции подпрограммы, содержащей уязвимость типа «переполнение буфера» и заданной в шестнадцатеричной форме, пятью следующими языковыми моделями: GPT-5, DeepSeek-R1, Qwin3-Max, GigaChat 2.0 и Yandex GPT 5.1 Pro. Выполнена оценка экземпляров исходного кода, получаемых каждой из языковых моделей, по следующим критериям: компилируемость, тождественность изначальному коду, качество (как понятность эксперту); по результатам тестирования произведено ранжирование моделей. Показано, что существует проблема «исчезновения» уязвимости из восстанавливаемого подобным образом исходного кода, присутствующей в изначальном машинном. Указана новизна исследования, его теоретическая и практическая значимость, намечены перспективы проведения более масштабных и строгих экспериментов за счет увеличения количества тестов и их «прогонов», а также более тонкой настройки моделей, в том числе специально разработанных для задач декомпиляции и поиска уязвимостей в машинном коде программ.

Ключевые слова: безопасность программного обеспечения, машинный код программы, уязвимость, декомпиляция, большие языковые модели (LLM), эксперимент

ON THE APPLICATION OF LLM-BASED DECOMPILATION FOR VULNERABILITY DETECTION IN PROGRAM MACHINE CODE

Buinevich M. V. ORCID ID 0000-0001-8146-0022,
Izrailov K. E. ORCID ID 0000-0002-9412-5693

Federal State Budgetary Educational Institution of Higher Education «Saint-Petersburg university of State fire service of EMERCOM of Russia», Saint-Petersburg, Russia, e-mail: bmv1958@yandex.ru

The decompilation task – converting program machine (executable) code into source code more amenable to vulnerability detection – is of critical importance in the field of software security. To address this task, the use of artificial intelligence, specifically large language models (LLMs), is being explored. This study aims to establish the limits of LLM applicability for countering vulnerabilities in software represented as machine code. A series of experiments (consisting of five runs) is conducted on the decompilation of a subroutine containing a buffer overflow vulnerability, provided in hexadecimal form, using five language models: GPT-5, DeepSeek-R1, Qwen3-Max, GigaChat 2.0, and Yandex GPT 5.1 Pro. The instances of source code produced by each language model are evaluated according to the following criteria: compilability, functional equivalence to the original code, and quality (i.e., intelligibility to an expert). Based on the experimental results, the models are ranked. The study reveals the problem of the "disappearance" of the vulnerability present in the original machine code from the source code reconstructed in this manner. The novelty of the research is highlighted, along with its theoretical and practical significance. Prospects for further, more extensive and rigorous experiments are outlined, including increasing the number of test cases and runs, as well as more fine-tuning of models, including those specifically designed for decompilation and vulnerability detection in program machine code.

Keywords: software security, program machine code, vulnerability, decompilation, large language models (LLMs), experiment

Введение

Большие языковые модели (далее – LLM, *аббр. от англ.* Large Language Model) показали свою работоспособность во множестве качественно разных областей. Естественно, проводятся исследования по применению моделей в области обеспечения безопасности программ, и в частности – для противодействия уязвимостям программного обеспечения. Так, одной из актуальных задач является преобразование машинного кода программы в исходный; данный процесс классически называется *декомпиляцией* [1]. Целью такого преобразования является получение представления программы, подходящего для ручного (экспертом) или автоматического поиска уязвимостей. Однако сама возможность применения LLM для де-

компиляции кода с уязвимостями еще недостаточно хорошо изучена, и именно этому вопросу посвящено исследование, приводимое в текущей статье.

Цель исследования – установление границ применимости LLM для противодействия уязвимостям программного обеспечения, представленного машинным кодом.

Материалы и методы исследования

Подходы к декомпиляции

На сегодняшний день уже существует некоторое количество способов декомпиляции машинного кода. Так, исторически первым считается трудоемкий ручной [2], требующий привлечения редких специалистов в предметной области. Накопление опыта в процессе применения способа позволило перейти к следующему, основанному на автоматизированном применении алгоритмов преобразования процессорных инструкций в исходный код программы; так, например для этого применяется известная среда дизассемблирования (с функциональностью по декомпиляции) IDA Pro [3]. Искусственный интеллект, и в частности машинное обучение, также показали определенную эффективность при решении отдельных задач декомпиляции [4; 5]. Существуют исследования по применению более строгих алгоритмов получения исходного кода по машинному, основанные на полном переборе [6] и сведении процесса декомпиляции к оптимизационной задаче [7]. Также стремительный рост вычислительных мощностей, приведший к повсеместному использованию LLM, позволил предположить востребованность таких моделей и в области декомпиляции [8].

Тем не менее у каждого из способов есть свои сильные и слабые стороны, что не позволяет говорить о полном и эффективном решении задачи декомпиляции. Так, ручной способ теоретически сможет из любого машинного кода получить его исходный, хотя и потребует критически большого времени. Применение искусственного интеллекта [9], и в том числе LLM, позволит получить решение задачи за меньшее время, но возникнет дополнительная подзадача по проверке его достоверности. С этой позиции среди способов выделяется авторский, основанный на решении оптимизационной задачи подбора конструкций исходного кода для соответствия всей программы машинному коду [10; 11], поскольку он гарантирует получение кода, компилируемого в заданный; тем не менее реализация способа требует улучшения в части снижения времени сходимости алгоритма.

Из указанных способов наименее изученным с позиции применения для задачи декомпиляции остается основанный на LLM. Поэтому интересной с научной и практической точки зрения можно считать оценку того, насколько большие модели в принципе применимы для данной задачи, особенно применительно к коду с уязвимостями. Для этого далее будет проведен базовый эксперимент по декомпиляции кода подпрограммы, заданной в бинарном виде (т. е. даже не как текст с ассемблерными инструкциями) и содержащей классическую уязвимость переполнения буфера.

Результаты исследования и их обсуждение

Тестовый пример

В качестве примера для эксперимента был использован фрагмент исходного кода с классической уязвимостью, а также его представление в ассемблерном и бинарном виде.

Исходный код

Исходный код подпрограммы для эксперимента приведен в листинге 1 (в начале каждой строки указан ее порядковый номер); к его особенностям относится то, что он содержит уязвимость типа «переполнение буфера» [12].

```
1: void test(const char *str) {
2:   char buff[12];
3:   int i = 0;
4:   for (; i < 12; ++i) {
5:     buff[i] = str[i];
6:   }
7:   buff[i] = '\x0';
8: }
```

Листинг 1. Подпрограмма с уязвимостью типа «переполнение буфера» (исходный код)

Так, следуя исходному коду (листинг 1), подпрограмма не выполняет никаких действий, кроме заполнения временного массива символов (переменная «buff») 12 символами из строки, указатель на которую передается в качестве аргумента (параметр «str»). Затем в конец новой строки (т.е. в 13-й символ, т. к. по окончании цикла значение переменной «i» равно 12 при учете нумерации с 0-го индекса) записывается конец строки (байт со значением «0»). Уязвимость заключается в том, что в подпрограмме «test()» под переменную «buff» выделено 12 байт, а в конце подпрограммы запись идет в 13-й байт, т. е. происходит выход за пределы массива и «затирание» других данных с потенциально опасными последствиями.

Ассемблерный код

Ассемблерный код подпрограммы из примера получен с помощью компилятора из состава «Microsoft (R) C/C++ Optimizing Compiler Version 19.29.30159 for x86», входящего в продукт «Microsoft Visual Studio Community 2019», с помощью следующей командой строки:

```
> cl.exe test.c /Fa /GS-
```

где «cl.exe» – утилита декомпиляции, «test.c» – файл с исходным кодом, «/Fa» – ключ для генерации ассемблерного кода, «/GS-» – ключ для отключения встраивания в код защиты от переполнения буфера.

Здесь следует отметить, что для полноценного построения выполняемого кода необходимо наличие функции начала выполнения программы – «main()», поэтому данная функция с пустым телом была добавлена в файл «test.c».

Полученный ассемблерный код, соответствующий подпрограмме «test()», приведен в листинге 2.

```
_TEXT SEGMENT
_buff$ = -16   ; size = 12
_i$ = -4       ; size = 4
_str$ = 8      ; size = 4
_test PROC
; Line 1
  push  ebp
  mov   ebp, esp
  sub   esp, 16 ; 00000010H
; Line 3
  mov   DWORD PTR _i$[ebp], 0
; Line 4
  jmp   SHORT $LN4@test
$LN2@test:
  mov   eax, DWORD PTR _i$[ebp]
  add   eax, 1
  mov   DWORD PTR _i$[ebp], eax
$LN4@test:
  cmp   DWORD PTR _i$[ebp], 12 ;
0000000cH
  jge   SHORT $LN3@test
                ...
```

Продолжение

```
                ...
; Line 5
  mov   ecx, DWORD PTR _str$[ebp]
  add   ecx, DWORD PTR _i$[ebp]
  mov   edx, DWORD PTR _i$[ebp]
  mov   al, BYTE PTR [ecx]
  mov   BYTE PTR _buff$[ebp+edx], al
; Line 6
  jmp   SHORT $LN2@test
$LN3@test:
; Line 7
  mov   ecx, DWORD PTR _i$[ebp]
  mov   BYTE PTR _buff$[ebp+ecx], 0
; Line 8
  mov   esp, ebp
  pop   ebp
  ret   0
_test   ENDP
_TEXT ENDS
```

Листинг 2. Подпрограмма с уязвимостью типа «переполнение буфера» (ассемблерный код)

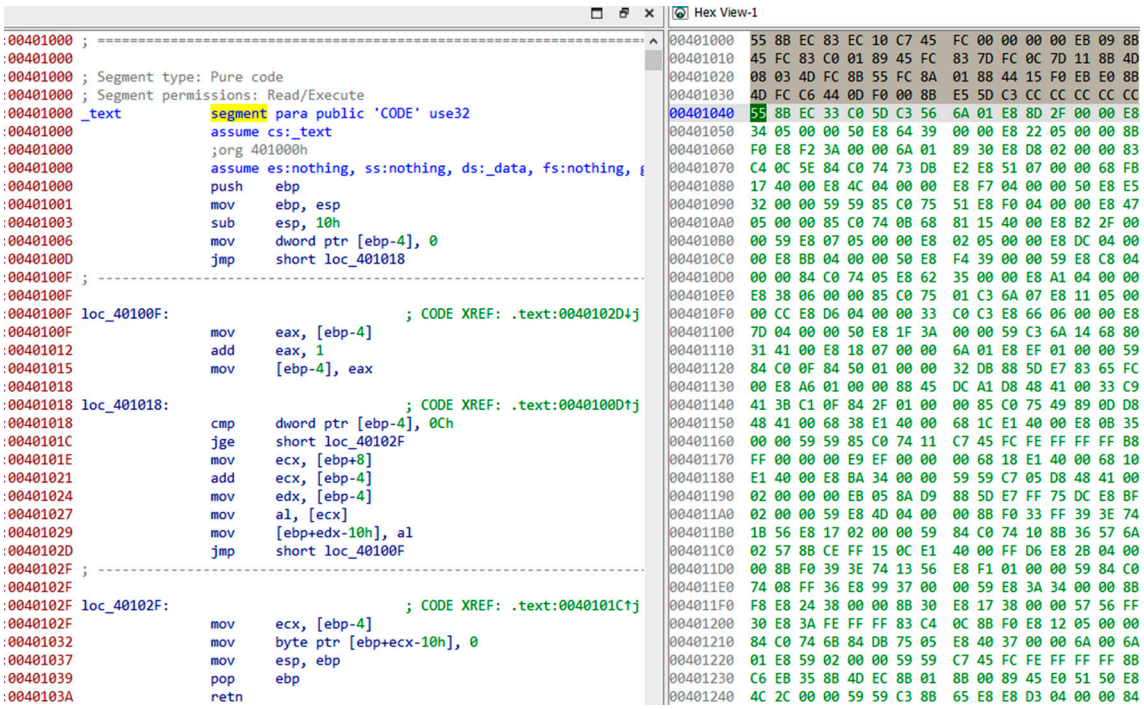
Как можно заметить, участки ассемблерного кода (листинг 2) соответствуют аналогичным конструкциям исходного кода (листинг 1). Соответственно, уязвимость переполнения буфера отражается в строках «_buff$ = -16» – массив «buff» начинается со смещением -16, «_i$ = -4» – переменная «i» расположена со смещением -4, оставляя тем самым для «buff» 12 байт свободного пространства, и «mov BYTE PTR _buff$[ebp+ecx], 0» – запись в 13-й элемент массива, что затирает тем самым область переменной «i» и фактически изменяет ее значение. Естественно, к каким-либо негативным последствиям (например, аварийному завершению программы) данный код не приведет, т. к. после «уязвимого» изменения переменной «i» она дальше нигде не используется, однако в более сложных программах (например, если эта переменная возвращается из функции или переполнение изменяет стек с адресами возврата из подпрограмм) ситуация может быть существенно иной.

Бинарный код

Получение бинарного кода программы возможно различными способами, одним из которых является ее дизассемблирование в среде IDA Pro [13]; результат приведен на рисунке.

На рисунке в левом окне отображается ассемблерный код подпрограммы «test()» (хотя название в явном виде не указано, т. к. эта информация теряется в процессе компиляции), а в правой части (окно «Hex View-1») – ее бинарный код с окончанием в виде 5 байтов со значением «0xCC» (выделено серым фоном). Таким образом, бинарное представление подпрограммы состоит из 59 байт, представленных в листинге 3.

И именно эта входная информация (т. е. текст листинга 3) будет являться частью запроса (т. н. промпт) к LLM с целью декомпиляции подпрограммы.



Отображение подпрограммы с уязвимостью типа «переполнение буфера» в IDA Pro

```
5 8B EC 83 EC 10 C7 45 FC 00 00 00 00 EB 09 8B 45 FC 83 C0 01 89 45 FC
83 7D FC 0C 7D 11 8B 4D 08 03 4D FC 8B 55 FC 8A 01 88 44 15 F0 EB E0 8B
4D FC C6 44 0D F0 00 8B E5 5D C3
```

Листинг 3. Подпрограмма с уязвимостью типа «переполнение буфера» (бинарный код в шестнадцатеричном формате)

Эксперимент

Проведем эксперимент по многократной декомпиляции бинарного кода подпрограммы «test()» в исходный код каждой из LLM.

В качестве распространенных LLM, которые будут далее применяться для декомпиляции листинга 3, были отобраны следующие пять, являющиеся наиболее распространенными (с указанием их производителя): GPT-5 (OpenAI), DeepSeek-R1 (DeepSeek), Qwin3-Max (Alibaba), GigaChat 2.0 (Сбербанк), Yandex GPT 5.1 Pro (Yandex) [14]; т.е. 3 зарубежные и 2 российские разработки.

В качестве запроса используем фразу «Декомпилируй следующие байты HEX-формата подпрограммы для процессора x64-86 в код на языке программирования C», за которой последуют байты в шестнадцатеричном (т. е. HEX) виде из листинга 3.

Проведем серию из пяти экспериментов для одного тестового примера для каждой из отобранных LLM с фиксацией полученных результатов (естественно, перед каждым применением контекст LLM будет очищаться).

Результаты первого применения LLM для восстановленного исходного кода приведены в листинге 4 (без дополнительного ручного редактирования).

Как видно из листингов, первые три и последняя LLM корректно восстановили исходный код, однако признаки уязвимости были утрачены – размер массива «buff» был увеличен до 16, избавив тем самым его от переполнения 13-м нулевым символом. В случае же GigaChar 2.0, код оказался некорректным – в цикле копируются не 12, а 13 символов (т. к. условием выхода является «i <= 12», вместо «i < 12»); также вместо массива для хранения переменной используется прямая работа с памятью через указатель (т. е. «((char *)((intptr_t) i-16)»). При этом код в листинге 4(г) сложен для понимания экспертом в связи с тем, что не восстановлена переменная «buff», а также используются излишние библиотека «stdio.h» и тип «intptr_t».

а) GPT-5

```
void func(const char *src) {
  char buf[16];
  int i = 0;
   while (++i < 12) {
    buf[i] = src[i];
  }
   buf[i] = '\0';
}
```

б) DeepSeek-R1

```
void subroutine(char *param) {
  char buffer[16]; // Бу-
фер на 16 байт (выделено в стеке)
  int i = 0; // Счетчик/индекс (ebp-0x4)
  // Копирование 12 байт из param в buffer
  for (i = 0; i < 12; i++) {
    buffer[i] = param[i];
  }
   buffer[i] = 0; // Запись нулевого
байта в конец строки
}
```

в) Qwin3-Max

```
void function(const char *src) {
  char buffer[16];
  int i = 0;
  do {
    if (i >= 12) break;
      buffer[i] = src[i];
    i++;
  } while (1);
  buffer[12] = '\0';
}
```

г) GigaChat 2.0

```
#include <stdio.h>
void func(int param1) {
  int i = 0;
  // Основной цикл обработки
  while (i <= 12) {
    char ch = *(char *)(param1 + i);
     // Запись обработанного символа
    *((char *)((intptr_t)i - 16)) = ch;
     ++i;
  }
  // Завершение цикла записью заверша-
ющего нуля
  ((char *)((intptr_t)i - 16))
[0] = '\0';
}
```

д) GPT 5.1 Pro

```
void copy_string_limited(const char *src) {
  char buffer[16]; // Локальный буфер на 16 байт (выделение: sub rsp, 0x10)
  int i = 0; // Счётчик цикла (mov dword [rbp-0x4], 0)
  // Цикл: копируем до 12 символов или до нулевого байта
  while (i < 12) {
    buffer[i] = src[i]; // Копирование байта (mov al, [rcx]; mov [rbp+rdx-
0x10], al)
    if (src[i] == 0) { // Если встретили нулевой байт — выходим
      break;
    }
    i++; // Инкремент счётчика (add eax, 1)
  }
   buffer[i] = 0; // Добавляем нулевой байт в конец (mov byte [rbp+rcx-0x10], 0)
   // Здесь buffer доступен, но в оригинальном коде он не используется дальше
   // (вероятно, функция предназначена для дальнейшей работы с buffer)
}
```

Листинг 4. Результат декомпиляции подпрограммы с уязвимостью типа «переполнение буфера» с помощью LLM

Результаты эксперимента

Результаты всех пяти экспериментов по декомпиляции подпрограммы «test()» с помощью каждой из LLM приведены в таблице. Для каждого теста через символ «|» указана его корректность с точки зрения компилируемости, тождественности изначальному в листинге 1 с учетом наличия уязвимости и качества восстановленного кода; значения по каждому критерию могут быть следующими: «+» – да, «–» – нет, «0» – частично.

Результаты декомпиляции подпрограммы с уязвимостью типа «переполнение буфера» различными LLM

LLM	Тест 1	Тест 2	Тест 3	Тест 4	Тест 5
GPT-5	+ | – | +	+ | – | +	+ | + | +	+ | – | +	+ | – | +
DeepSeek-R1	+ | – | +	– | – | +	+ | – | +	+ | – | +	+ | – | +
Qwin3-Max	+ | – | +	+ | – | +	+ | – | +	+ | – | +	– | – | +
GigaChat 2.0	– | – | –	– | – | +	– | – | –	– | – | –	– | – | –
Yandex GPT 5.1 Pro	+ | – | +	– | – | +	+ | – | +	+ | 0 | +	+ | – | +

Дадим ряд комментариев касательно проведенного эксперимента и его результатов.

Во-первых, корректный код с уязвимостью был восстановлен только с помощью GPT-5 на третьем «прогоне». Yandex GPT 5.1 Pro на четвертом «прогоне» также корректно восстановил код с уязвимостью типа «переполнение буфера»; однако, после ряда собственных уточнений (или «размышлений»), LLM исправила его на код с массивом размером 16, потеряв тем самым признак уязвимости (поэтому в соответствующей ячейке таблицы указано значение «0»). Необходимо отметить, что существуют строгие «неинтеллектуальные» методы статического анализа бинарного кода, позволяющие выявлять данный тип уязвимостей [15]. Во-вторых, на одном из «прогонов» GPT-5 и Qwin3-Max восстановили массив размером 13 (а не 16, как в большинстве остальных), что можно считать наиболее близким к корректному результату. В-третьих, GigaChat 2.0 достаточно часто добавляла излишнее использование внешних библиотек и конструкций, что ухудшало восприятие кода.

Суммирование значений по критериям и «прогонам», указанных в таблице, позволяет получить следующую интегральную оценку (суммируя значения по первым двум критериям и усредняя по третьему) каждой из LLM:

1) GPT-5 – 5 | 1 | + ;
2) DeepSeek-R1 – 4 | 0 | + ;
3) Qwin3-Max – 4 | 0 | + ;
4) GigaChat 2.0 – 0 | 0 | – ;
5) Yandex GPT 5.1 Pro – 4 | 0.5 | + .

Таким образом, лишь модель GPT-5 в 20% случаев корректно восстановила код с уязвимостью, а Yandex GPT 5.1 Pro сделала это частично. Кроме GigaChat, все остальные LLM в 80% получали полностью работоспособный код с нейтрализованной уязвимостью (т. е. переполнения буфера не происходило) и имели высокое качество кода. При этом GigaChat «проиграла» остальным LLM по 1-му и 3-му критериям.

Выполним простейшее ранжирование LLM, назначив каждому из значений критериев баллы: за «+» – 1, за «0» – 0.5 и за «–» – 0. В результате, отсортированный по сумме баллов список LLM будет следующим: 1) GPT-5 (11 баллов); 2) Yandex GPT 5.1 Pro (9.5 баллов); 3) DeepSeek-R1 (9 баллов); 4) Qwin3-Max (9 баллов); 5) GigaChat 2.0 (1 балл). Таким образом, наилучшие показатели LLM оказались у GPT-5, за которой идут практически с одинаковым количеством баллов Yandex GPT 5.1 Pro, DeepSeek-R1 и Qwin3-Max, от них с большим отрывом отстает GigaChat 2.0.

Заключение

В работе проведен эксперимент по декомпиляции подпрограммы, заданной в бинарном виде (через шестнадцатеричную запись), с помощью пяти популярных LLM общего предназначения, и оценены его результаты на предмет восстановления исходного кода подпрограммы, содержащей уязвимость типа «переполнение буфера». С одной стороны, результаты эксперимента показали работоспособность LLM-технологии и в этом, более сложном, случае (а, например, не при декомпиляции более «осмысленного» ассемблерного кода). С другой стороны, практически во всех пяти «прогонах» восстановленный код не содержал уязвимости, поскольку был скорректирован LLM до безопасного состояния.

Новизна исследования состоит в том, что впервые приведены натурные эксперименты по применению LLM в интересах не только декомпиляции бинарного кода, но и с целью выявления в нем уязвимостей.

Теоретическая значимость состоит в том, что установлена принципиальная возможность и определены границы применимости LLM-технологии для данной задачи из области информационной безопасности; практическая же значимость заключается в оценке базовой работоспособности популярных LLM для классической задачи обнаружения уязвимости типа «переполнение буфера».

Продолжением работы должно стать проведение более расширенных экспериментов (например, увеличение количества тестов и их «прогонов», настройки моделей и др.), а также создание специализированной языковой модели (как большой, так и малой) для задач декомпиляции, а в перспективе – и поиска уязвимостей.

Список литературы

1. Трошина Е. Н. Исследование и разработка методов декомпиляции программ: автореф. дис. ... канд. физ.-мат. наук. Москва, 2009. 24 с. EDN: NLCUUD.

2. Касперски К. Техника отладки программ без исходных текстов. СПб.: БХВ-Петербург, 2005. 832 с. ISBN: 5-94157-229-8. URL: https://ibooks.ru/bookshelf/335097/reading (дата обращения: 10.03.2026).

3. Аешин И. Т. Реверс-инжиниринг программного продукта с использованием IDA Pro // Актуальные проблемы авиации и космонавтики. 2018. Т. 3. № 4 (14). С. 808–809. EDN: VTUAPE.

4. Shin E. C. R., Song D., Moazzezi R. Recognizing functions in binaries with neural networks // The proceedings of 24th USENIX Conference on Security Symposium (USA, Washington, D.C., 2015 August 12-14). 2015. P. 611–626. ISBN: 978-1-931971-232. URL: https://people.eecs.berkeley.edu/~dawnsong/papers/Recognizing%20functions%20in%20binaries%20with%20neural%20networks_augsut%202015.pdf (дата обращения: 10.03.2026).

5. He J., Ivanov P., Tsankov P., Raychev V., Vechev M. Debin: Predicting Debug Information in Stripped Binaries // The proceedings of SIGSAC Conference on Computer and Communications Security (New York, NY, USA, 15-19 October 2018). 2018. 1667–1680. DOI: 10.1145/3243734.3243866.

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

7. Израилов К. Е. Генетический реверс-инжиниринг программ для поиска уязвимостей // Вестник Санкт-Петербургского университета Государственной противопожарной службы МЧС России. 2025. № 1. С. 109-119. EDN: UHFRBI. DOI: 10.61260/2218-130X-2025-1-109-119.

8. Tan H., Luo Q., Li J., Zhang Y. LLM4Decompile: Decompiling Binary Code with Large Language Models // The proceeding of Conference on Empirical Methods in Natural Language Processing (Miami, Florida, 12–16 November 2024). 2024. P. 3473–3487. DOI: 10.18653/v1/2024.emnlp-main.203.

9. Katz D. S., Ruchti J., Schulte E. Using recurrent neural networks for decompilation // The proceedings of 25th International Conference on Software Analysis, Evolution and Reengineering (Campobasso, Italy, 20-23 March 2018). 2018. P. 346-356. DOI: 10.1109/SANER.2018.8330222.

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

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

12. Дудина И. А., Малышев Н. Е. Об одном подходе к анализу строк в языке СИ для поиска переполнения буфера // Труды Института системного программирования РАН. 2018. Т. 30. № 5. С. 55-74. EDN: YQOZXF. DOI: 10.15514/ISPRAS-2018-30(5)-3.

13. Вдовченко Г. П. Метод и анализ вредоносных программ // Международный журнал информационных технологий и энергоэффективности. 2025. Т. 10. № 7-1 (57). С. 18-22. EDN: VNMIIY.

14. Коробцов В. И., Овсянников И. В., Сачков Д. И. Автоматическая генерация надежного программного кода с помощью генеративных предобученных трансформеров (GPT) // Информационные технологии и математическое моделирование в управлении сложными системами. 2024. № 1 (21). С. 52-59. EDN: KLVTWY.

15. Ding S., Yuan J. Identifying buffer overflow vulnerabilities based on binary code // IEEE International Conference on Computer Science and Automation Engineering (Shanghai, China, 10-12 June 2011). 2011. DOI: 10.1109/CSAE.2011.5952950.

Конфликт интересов: Авторы заявляют об отсутствии конфликта интересов.

Conflict of interest: The authors declare that there is no conflict of interest.

Финансирование: Авторы заявляют об отсутствии внешнего финансирования.

Financing: The research was performed without external funding.