Научная статья
УДК 004.4; DOI: 10.61260/2218-130X-2025-4-72-81

ПРОБЛЕМНЫЕ ВОПРОСЫ ПРИМЕНЕНИЯ БОЛЬШИХ ЯЗЫКОВЫХ МОДЕЛЕЙ ДЛЯ ДЕКОМПИЛЯЦИИ МАШИННОГО КОДА С УЯЗВИМОСТЯМИ

✉**Израилов Константин Евгеньевич.**
Санкт-Петербургский университет ГПС МЧС России, Санкт-Петербург, Россия
✉***konstantin.izrailov@mail.ru***

Аннотация. Работа посвящена проблеме наличия уязвимостей в программном обеспечении в условиях отсутствия исходного кода, одним из путей противодействия которым является декомпиляция машинного (выполняемого) кода программ. Рассмотрено применение относительной новой технологии больших языковых моделей для решения данной задачи по восстановлению псевдоисходного кода, подходящего для обнаружения и устранения уязвимостей. Выявлены такие проблемные вопросы предметной области, как неполнота датасета для редких процессорных архитектур, отсутствие гарантии тождественности полученного исходного кода заданному машинному, санация восстанавливаемого исходного кода путем исправления уязвимостей, галлюцинирование в коде и сложность восстановления обфусцированного (в том числе оптимизированного) кода. Для обоснования и демонстрации сути каждого проблемного вопроса приведен практический пример по декомпиляции функций ассемблерного кода с помощью распространенной большой языковой модели DeepSeek-V3.2. Указано негативное влияние проблемных вопросов на итоговую нейтрализацию уязвимостей.

Ключевые слова: безопасность программного обеспечения, уязвимости, реверс-инжиниринг, декомпиляция, искусственный интеллект, проблемные вопросы

Для цитирования: Израилов К.Е. Проблемные вопросы применения больших языковых моделей для декомпиляции машинного кода с уязвимостями // Научно-аналитический журнал «Вестник Санкт-Петербургского университета Государственной противопожарной службы МЧС России». 2025. № 4. С. 72–81. DOI: 10.61260/2218-130X-2025-4-72-81.

Scientific article

PROBLEM ISSUES IN USING LARGE LANGUAGE MODELS FOR DECOMPILATION OF MACHINE CODE WITH VULNERABILITIES

✉**Izrailov Konstantin E.**
Saint-Petersburg university of State fire service of EMERCOM of Russia, Saint-Petersburg, Russia
✉***konstantin.izrailov@mail.ru***

Abstract. This paper examines the problem of software vulnerabilities in the absence of source code. One way to counter them is by decompilation the machine (executable) code of programs. The paper considers the application of a relatively new technology, large language models, to the task of restoring pseudo-source code suitable for detecting and eliminating vulnerabilities. The paper identifies problematic issues in the subject area, such as the incompleteness of the dataset for rare processor architectures, the lack of a guarantee that the obtained source code is identical to the specified machine code, the sanitization of the recovered source code by fixing vulnerabilities, hallucinations in the code, and the difficulty of restoring obfuscated (including optimized) code. To substantiate and demonstrate the essence of each problematic issue, a practical example of decompilation assembly code functions using the widespread large language model DeepSeek-V3.2 is provided. The negative impact of these problematic issues on the final neutralization of vulnerabilities is also indicated.

Keywords: software security, vulnerabilities, reverse engineering, decompilation, artificial intelligence, problem issues

For citation: Izrailov K.E. Problem issues in using large language models for decompilation of machine code with vulnerabilities // Scientific and analytical journal «Vestnik Saint-Petersburg university of State fire service of EMERCOM of Russia». 2025. № 4. P. 72–81. DOI: 10.61260/2218-130X-2025-4-72-81.

Введение

Получение исходного кода программы по ее машинному с последующим поиском и устранением уязвимостей (то есть их нейтрализация) является хорошо известным и часто применяемым подходом повышения информационной безопасности программного обеспечения. Данный процесс, называемый *декомпиляцией* (как антагонист компиляции), имеет различные способы проведения, ни один из которых на сегодняшний день не занял лидирующую позицию. Так, ручной способ восстановления исходного кода крайне трудоемок [1], применение экспертных алгоритмов преобразования охватывает не все вариации выполняемого кода [2, 3], машинное обучение применимо лишь для частных задач области [4], а более строгие способы перебора (в том числе авторский, решающий оптимизационную по подбору конструкций исходного языка до требуемых машинных с помощью генетических алгоритмов [5], названный *генетической декомпиляцией*) требуют некоторой доработки в части повышения оперативности.

С другой стороны, относительно новая технология применения больших языковых моделей (англ. Large Language Model, LLM) показала свою перспективность в огромном количестве областей. Тем не менее, границы работоспособности данной технологии в задачах декомпиляции были недостаточно рассмотрены и оценены [6], что является упущением для предметной области и частично будет устранено в данной статье далее. Основной же упор при рассмотрении будет делаться не только на восстановление исходного кода по машинному, но и корректное отражение в нем уязвимостей.

Проблемные вопросы

Достаточно глубокое знакомство автора с задачей декомпиляции [7], проведение исследований в области безопасности программного кода, а также понимание основ LLM [8] позволили выделить следующие потенциальные проблемные вопросы технологии (указываемые далее с префиксом «ПВ_»), каждый из которых рассматриваются далее; сам же процесс будет коротко называться *LLM-декомпиляцией*. Также, в качестве тестируемой LLM используется DeepSeek версии 3.2 (в режиме «Глубокое мышление») от одноименной компании. Указанная LLM, неспециализированная для задач анализа кода, будет применяться для восстановления исходного кода по заданному ассемблерному, который в определенной степени может считаться тождественным бинарному или выполняемому представлению программы. Запросом же для применения LLM будет следующий – «Декомпилируй следующий ассемблерный код для процессорной архитектуры x86-64 в код на языке программирования C:», за которым шел соответствующий ассемблерный код. Также LLM использовалась в режиме со стандартными настройками, а, например, не в состоянии генерации наиболее вероятных ответов (то есть когда ее параметр «температуры» равен 0).

ПВ_1. Неполнота датасета для редких архитектур.

Большинство современных LLM обучается на огромных массивах данных [9], включающих в том числе информацию о машинном коде для различных процессорных архитектур. Однако редкость архитектуры автоматически уменьшает и объем связанных с ней обучающих данных, что ведет к снижению точности декомпиляции и с помощью самой LLM. Такой эффект будет особенно проявляться, если исходный код и результаты его компиляции не являются распространенными.

Известные алгоритмы, такие как рекурсивная функция вычисления факториала числа, с большой вероятностью будут корректно декомпилированы различными LLM под практически любые архитектуры. Поэтому в качестве примера можно рассмотреть модификацию данной функции путем изменения множителя в зависимости от остатка деления текущего рекурсивно вычисляемого значения на 3 – прибавление 0,5 в случае остатка 1 и вычитание 0,5 в случае остатка 2 (Листинг 1); соответственно, функция будет возвращать нецелое число в формате double (то есть двойной точности).

Листинг 1. Модифицированная функция рекурсивного вычисления факториала числа (исходный код).

```
double test_1(int n) {
  if (n == 0 || n == 1)
    return 1;

  if (n % 3 == 1)
    return ((double)n + 0.5) * test_1(n - 1);
  if (n % 3 == 2)
    return ((double)n - 0.5) * test_1(n - 1);

  return n * test_1(n - 1);
}
```

С большой вероятностью приведенная вариация «факториала» (Листинг 1) неизвестна большинству LLM, что затруднит ее декомпиляцию.

В результате применения LLM с указанным ранее запросом исходный код был корректно восстановлен. Однако после применения LLM к ассемблерному коду для малораспространенного процессора eZ80 от компании Zilog (с соответствующим изменением самого запроса), используемого в портативных графических калькуляторах от компании Texas Instruments, модель не смогла корректно восстановить исходный код, выдавая результаты, подобные приведенному в Листинге 2.

Листинг 2. Модифицированная функция рекурсивного вычисления факториала числа (декомпилированный код).

```
float test_1(int n) {
  if (n == 0 || n == 1) {
    return -1.0f;
  }

  int rem = n % 3;
  if (rem == 1) {
    return (2.0f * n) * test_1(n - 1);
  } else if (rem == 2) {
    return (2.0f * n) * test_1(n - 1);
  } else {
    return (float)n * test_1(n - 1);
  }
}
```

Эксперимент был многократно повторен с получением аналогичных результатов.

Следовательно, можно утверждать, что распространенность процессорной архитектуры имеет существенное влияние на корректность LLM-декомпиляции. Впрочем, необходимо отметить, что ассемблерный код сам по себе был сложнее, поскольку состоял из 152 строк, в отличие от аналогичного для x86-64 – из 84 строк.

С точки зрения уязвимостей в программном коде, описанное поведение LLM-декомпиляции означает, что они будут более успешно нейтрализованы лишь для популярных процессорных архитектур (для которых, фактически, и так разработано достаточное количество необходимых средств и методик); однако программное обеспечение для более редких архитектур, потенциально используемое в специализированных устройствах критических областей, будет оставаться небезопасным.

ПВ_2. Отсутствие гарантии тождественности исходного кода.

Применение большинства «нестрогих» способов декомпиляции машинного кода в исходный имеет важный недостаток, основанный на отсутствии доказательства тождественности этих двух кодов. Кроме LLM к такого рода способам могут быть отнесены ручной и основанный на машинном обучении; при этом, генетическая декомпиляция гарантирует корректность восстановленного исходного кода, так как это является условием завершения ее работы, а алгоритмические способы предполагают экспертно заданные сценарии преобразования, не допускающие вероятностных отклонений (если не содержат ошибок в работе).

В качестве обоснования проблемности данного вопроса проведен эксперимент по LLM-декомпиляции ассемблерного кода вычисления факториала числа без использования рекурсии, исходный код функции которого приведен в Листинге 3.

Листинг 3. Функция безрекурсивного вычисления факториала числа (исходный код).

```
int test_2(int n) {
  int x = 1;
  do {
    x *= n;
    n -= 1;
  } while (n > 0);
  return x;
}
```

Вводится ограничение, что функция должна быть работоспособна для всех $n \geq 1$, поскольку для $n = 0$ она вернет 0, хотя формально $0! = 1$.

Результат многократной LLM-декомпиляции привел к получению исходного кода, с незначительными отклонениями идентичного приведенному в Листинге 4.

Листинг 4. Функция безрекурсивного вычисления факториала числа (декомпилированный код).

```
int test_2(int n) {
  int result = 1;
  while (n != 0) {
    result *= n;
    n--;
  }
  return result;
}
```

Восстановленный код формально соответствует изначально заданному, однако функция при $n = 0$ хотя и возвращает более корректное значение $0! = 1$, но в изначальном коде логика была иной.

Несмотря на успешность декомпиляции важно отметить различие кода в Листингах 3 и 4, заключающееся в изменении конструкции цикла – с «do {…} while» на «while {…}». Как результат, ассемблерный код, получаемый компиляцией Листинга 3 будет структурно отличен от кода, полученного из Листинга 4 – появится дополнительная вторая метка и переход на нее перед первой; данное отличие представлено в таблице (метки выделены жирным шрифтом).

Таблица

Отличие ассемблерного кода при компиляции исходного кода в Листингах 3 и 4

Результат компиляции Листинга 3	Результат компиляции Листинга 4

`test_2:`
`...`
`.L2:`
`...`
`Cmp DWORD PTR [rbp-20], 0`
`Jne .L2`
`...`
	`test_2:`
	`...`
	`Jmp .L2`
	`.L3:`
	`...`
	`.L2:`
	`Cmp DWORD PTR [rbp-20], 0`
	`Jne .L3`
	`...`

Следовательно, хотя LLM и сгенерировала исходный код с логикой, практически тождественной изначальному, однако, корректность его восстановления не удастся проверить компиляцией и тривиальным сравнением ассемблерных кодов – структура кода настолько поменялась, что эти изменения существенно отразились и в машинных инструкциях.

С точки зрения уязвимостей в программном коде такое существенное изменение структуры программы означает, что если LLM ошибочно добавит уязвимый код или, наоборот, не отобразит его, то проверка данного факта будет представлять отдельно стоящую сложную задачу. Таким образом, отсутствует полная гарантия как наличия уязвимости в программе, так и ее отсутствия.

ПВ_3. Санация восстанавливаемого исходного кода.

Достаточно специфичным эффектом является внесение исправлений в логику кода самой LLM, исходя из типовых шаблонов или конструкций, присутствующих в обучающем датасете. Опасность данного эффекта с позиции информационной безопасности заключается в том, что уязвимости в коде могут автоматически устраняться – то есть будет произведена «санация», что не позволит их обнаружить и устранить в изначальном ассемблерном коде. Продемонстрирован данный эффект с помощью исходного и его ассемблерного кодов в Листингах 5 и 6, суть уязвимости заключается в выходе индекса массива за допустимый диапазон [10].

Листинг 5. Функции с уязвимостью выхода за границы массива (исходный код).

```
int test_3(void) {
  char buff[4] = {1, 2, 3, 4};
  return buff[4];
}
```

Листинг 6. Функции с уязвимостью выхода за границы массива (ассемблерный код).

```
test_3:
  push    rbp
  mov     rbp, rsp
  mov     DWORD PTR [rbp-4], 67305985
  movzx   eax, BYTE PTR [rbp+0]
  movsx   eax, al
  pop     rbp
  ret
```

Следуя коду в Листинге 5, в первой строке тела функции «test_3()» создается локальная переменная-массив «buff» размером четыре байта, которая затем инициализируется значениями 1, 2, 3 и 4, что отражается в ассемблерном коде в Листинге 6 с помощью единой 4-байтной записи числа 67 305 985 в память по адресу «rbp-4». Затем во второй строке из функции возвращается пятый индекс элемента массива (поскольку их нумерация начинается с нулевого) – «buff [4]», что в Листинге 6 записывается через доступ к памяти по адресу «rbp+0», который отстает на пять байт от адреса «rbp-4». Таким образом, функция вернет не последний элемент массива – число 4, а некоторое значение из памяти, расположенной за массивом, что может содержать конфиденциальные данные (или вообще вызвать программное исключение при выполнении).

Многократное применение LLM-декомпиляции к Листингу 6 позволило получить исходный код, подобный (с некоторыми вариациями) приведенному в Листинге 7.

Листинг 7. Функции с уязвимостью выхода за границы массива (декомпилированный код).

```
int test_3() {
  int value = 67305985;
  return *(char *)&value;
}
```

Согласно результату декомпиляции (Листинг 7), хотя инициализация массива и была корректно восстановлена (поскольку число 67 305 985 в шестнадцатеричной записи соответствует 0x04030201, то есть четырем байтам – 4, 3, 2 и 1), однако декомпилированная функция возвращает не значение в памяти за пределами массива «buff», а его первый элемент, то есть 4. Следовательно, информация об уязвимости в ассемблерном коде после LLM-декомпиляции была потеряна. Данный эффект создает опасные предпосылки к необеспечению безопасности программного обеспечения, когда, с одной стороны, эксперт применяет LLM для декомпиляции и обнаружения уязвимостей, а с другой стороны, LLM сама стремится их скрыть от эксперта.

ПВ_4. Галлюцинирование в коде.

Еще одним эффектом, связанным с LLM, считается так называемое возникновение галлюцинаций [11] – выводов моделей, которые или противоречат исходным данным запроса, или не могут быть проверены. Данный термин применим и для задачи LLM-декомпиляции, а одним из его проявлений может быть восстановление вещественных чисел, пример исходного кода с которыми приведен в Листинге 8.

Листинг 8. Функции с вещественными числами (исходный код).

```
double test_4(int x, int y) {
  return x * 0.2 + y * 2.0;
}
```

Данная функция (Листинг 8) является тривиальной, возвращающей взвешенную сумму от двух аргументов. Сложность декомпиляции для LLM заключается в форме хранения числа нецелого 0,2 типа double, запись которого в ассемблерном коде приведена в Листинге 9.

Листинг 9. Запись числа 0,2 в формате double (ассемблерный код).

```
.long   -1717986918
.long   1070176665
```

Суть такой записи (Листинг 9) заключается в том, что восемь указанных байт в шестнадцатеричной записи для double-формата (нецелое число двойной точности) имеют вид 0x3fc999999999999a, где первый бит соответствует *знаку* числа (0 для «+», 1 для «–»),

следующие – 11 битам *порядка*, а оставшиеся 52 – *мантиссе*; итоговая же формула вычисления значения является следующей – $\text{знак} \times \left(\frac{1+\text{мантисса}}{2^{52}}\right) \times 2^{\text{порядок}-1023}$.

Применение LLM-декомпиляции к Листингу 8 получает код, близкий к нему по логике, но отличный по константным значениям, он приведен в Листинге 10.

Листинг 10. Функции с вещественными числами (декомпилированный код).

```
double test_4(int a, int b) {
  return 2.5 * a + 2 * b;
}
```

Повторение LLM-декомпиляции для ассемблерного кода функции «test_4» привело к получению аналогичных результатов с некоторой вариацией в конкретном значении коэффициента перед переменной «a».

Следовательно (Листинг 10), при декомпиляции исходного кода с вещественными числами, записанными в формате двойной точности, можно говорить о галлюцинировании LLM.

Эффект галлюцинирования может иметь более существенные проявления на уязвимости в программном коде, например, добавляя несуществующие из них или, в редких случаях, скрывая присутствующие. При этом даже незначительное изменение восстановленного кода не только усложнит поиск уязвимостей, но и после устранения получит хотя и безопасный код, но несоответствующий изначальному.

ПВ_5. Сложность восстановления обфусцированного кода.

Обфускация кода, классически применяемая для его «запутывания» и усложнения анализа [12], будет оказывать существенное влияние и на результативность LLM-декомпиляции. Операцией с подобным эффектом, но иным предназначением, может считаться компиляция исходного кода с оптимизацией, поскольку, например, при ускорении кода или уменьшении его размера, логика выполнения и выбор ассемблерных инструкций может существенно изменяться.

Для проверки данного утверждения можно воспользоваться практически корректно восстанавливаемым исходным кодом функции из Листинга 3 (кроме $n = 0$), к которому последовательно применим максимальный уровень оптимизации (ключ «-O3» для компилятора «gcc») и обфускацию (с помощью онлайн-ресурса https://www.pelock.com/obfuscator/ в демо режиме с тройным проходом); LLM-декомпилированный код представлен в Листинге 11.

Листинг 11. Функция безрекурсивного вычисления факториала числа после оптимизации и обфускации (декомпилированный код).

```
int test_5(int dwParam) {
  int eax, edx, ecx;

  eax = dwParam;
  while (1) {
    if (eax <= 0) {
      return eax;
    }

    ecx = eax - 1;
    if (ecx == 0) {
      return eax;
    }

    edx = ecx;
```

```
      if (edx == 0) {
        break;
      }

      ecx = edx - 1;
      if ((ecx & 1) == 0) {
        eax *= edx;
      } else {
        eax *= ecx;
      }

      if (eax > 0) {
        continue;
      } else {
        return eax;
      }
    }

    return eax;
  }
```

Изучение кода (Листинг 11) позволяет сделать вывод о том, что он не обладает изначальным функционалом по вычислению факториала числа. При этом различные проведения данного теста и изучение хода размышлений LLM говорят о том, что она не способна работать с такого рода большим и запутанным кодом; при этом в ряде случаев для меньшего уровня обфускации она выдавала близкие к корректному результаты, но не гарантируя правильность и указывая формулировку «Поэтому я напишу код для факториала» и «Но в данном случае, это лучшее, что я могу сделать». Следовательно, усложнение кода не только существенно «запутывает» LLM-декомпиляцию, но и потенциально приводит к галлюцинированию в том смысле, что восстановленный код становится догадкой.

Исходя из того, что злоумышленники для затруднения обнаружения уязвимостей в программе производят обфускацию ее кода (полностью или частично), то LLM-декомпиляция в этом случае не может считаться удовлетворительным решением, поскольку будет приводить к получению заведомо ошибочных результатов, не позволяя оценить опасность программного обеспечения должным образом.

Заключение

В работе проведена серия тестов по декомпиляции ассемблерного кода с помощью неспециализированной LLM в интересах дальнейшего обнаружения и устранения уязвимостей. Основной упор при тестировании данной технологии сделан на рассмотрение проблемных вопросов процесса декомпиляции, а также определение границ его применимости. Новизна исследования заключается в расширении знаний касательно применимости LLM в области безопасности программного обеспечения. Значимость же состоит в выявлении и изучении достаточно новых эффектов, снижающих эффективность обеспечения информационной безопасности с помощью LLM. Продолжением исследования должно стать более глубокое рассмотрение каждого из проблемных вопросов (а также близких к ним, но относящихся к другим способам декомпиляции [13]) и выработка предложений касательно их разрешения.

Список источников

1. Касперски К. Техника отладки программ без исходных текстов. СПб.: БХВ-Петербург, 2005. 832 с.

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

3. Израилов К.Е. Алгоритмизация машинного кода телекоммуникационных устройств как стратегическое средство обеспечения информационной безопасности // Национальная безопасность и стратегическое планирование. 2013. № 2 (2). С. 28–36.

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. Washington, 2015. P. 611–626.

5. Израилов К.Е. Генетический реверс-инжиниринг программ для поиска уязвимостей // Научно-аналитический журнал «Вестник Санкт-Петербургского университета Государственной противопожарной службы МЧС России». 2025. № 1. С. 109–119. DOI: 10.61260/2218-130X-2025-1-109-119.

6. LLM4Decompile: Decompiling Binary Code with Large Language Models / H. Tan [et al.] // The proceeding of Conference on Empirical Methods in Natural Language Processing. Miami, 2024. P. 3473–3487. DOI: 10.18653/v1/2024.emnlp-main.203.

7. Израилов К.Е. Концепция генетической декомпиляции машинного кода телекоммуникационных устройств // Труды учебных заведений связи. 2021. Т. 7. № 4. С. 10–17. DOI: 10.31854/1813-324X-2021-7-4-95-109.

8. Yin X., Ni C., Wang S. Multitask-Based Evaluation of Open-Source LLM on Software Vulnerability // Transactions on Software Engineering. Vol. 50. № 11. P. 3071–3087. DOI: 10.1109/TSE.2024.3470333.

9. Galadima H.S., Doherty C., Brennan R. Towards LLM-based Synthetic Dataset Generation of Cyber Incident Response Process Logs // The proceedings of Cyber Research Conference. Carlow, 2024. P. 1–4. DOI: 10.1109/Cyber-RCI60769.2024.10939563.

10. Calatayud B.M., Meany L. A comparative analysis of Buffer Overflow vulnerabilities in High-End IoT devices // The proceedings of 12th Annual Computing and Communication Workshop and Conference. Las Vegas, 2022. P. 0694–0701. DOI: 10.1109/CCWC54503.2022.9720884.

11. Комашко М.Н. ChatGPT, текст, информация: критический анализ // Труды по интеллектуальной собственности. 2024. Т. 50. № 3. С. 118–128. DOI: 10.17323/tis.2024.22306.

12. Милушев Э.Х., Батунин Я.В., Попов А.А. Методы обфускации кода: сравнительный анализ // Наукосфера. 2025. № 5-2. С. 1–6. DOI: 10.5281/zenodo.15574433.

13. Израилов К.Е. Проблемные вопросы генетической деэволюции представлений программы для поиска в них уязвимостей и рекомендации по их разрешению // Труды учебных заведений связи. 2025. Т. 11. № 1. С. 84–98. DOI: 10.31854/1813-324X-2025-11-1-84-98.

References

1. Kasperski K. Tekhnika otladki programm bez iskhodnyh tekstov. SPb.: BHV-Peterburg, 2005. 832 s.

2. Aeshin I.T. Revers-inzhiniring programmnogo produkta s ispol'zovaniem IDA Pro // Aktual'nye problemy aviacii i kosmonavtiki. 2018. T. 3. № 4 (14). S. 808–809.

3. Izrailov K.E. Algoritmizaciya mashinnogo koda telekommunikacionnyh ustrojstv kak strategicheskoe sredstvo obespecheniya informacionnoj bezopasnosti // Nacional'naya bezopasnost' i strategicheskoe planirovanie. 2013. № 2 (2). S. 28–36.

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. Washington, 2015. P. 611–626.

5. Izrailov K.E. Geneticheskij revers-inzhiniring programm dlya poiska uyazvimostej // Nauchno-analiticheskij zhurnal «Vestnik Sankt-Peterburgskogo universiteta Gosudarstvennoj protivopozharnoj sluzhby MCHS Rossii». 2025. № 1. S. 109–119. DOI: 10.61260/2218-130X-2025-1-109-119.

6. LLM4Decompile: Decompiling Binary Code with Large Language Models / H. Tan [et al.] // The proceeding of Conference on Empirical Methods in Natural Language Processing. Miami, 2024. P. 3473–3487. DOI: 10.18653/v1/2024.emnlp-main.203.

7. Izrailov K.E. Koncepciya geneticheskoj dekompilyacii mashinnogo koda telekommunikacionnyh ustrojstv // Trudy uchebnyh zavedenij svyazi. 2021. T. 7. № 4. S. 10–17. DOI: 10.31854/1813-324X-2021-7-4-95-109.

8. Yin X., Ni C., Wang S. Multitask-Based Evaluation of Open-Source LLM on Software Vulnerability // Transactions on Software Engineering. Vol. 50. № 11. P. 3071–3087. DOI: 10.1109/TSE.2024.3470333.

9. Galadima H.S., Doherty C., Brennan R. Towards LLM-based Synthetic Dataset Generation of Cyber Incident Response Process Logs // The proceedings of Cyber Research Conference. Carlow, 2024. P. 1–4. DOI: 10.1109/Cyber-RCI60769.2024.10939563.

10. Calatayud B.M., Meany L. A comparative analysis of Buffer Overflow vulnerabilities in High-End IoT devices // The proceedings of 12th Annual Computing and Communication Workshop and Conference. Las Vegas, 2022. P. 0694–0701. DOI: 10.1109/CCWC54503.2022.9720884.

11. Komashko M.N. ChatGPT, tekst, informaciya: kriticheskij analiz // Trudy po intellektual'noj sobstvennosti. 2024. T. 50. № 3. S. 118–128. DOI: 10.17323/tis.2024.22306.

12. Milushev E.H., Batunin Ya.V., Popov A.A. Metody obfuskacii koda: sravnitel'nyj analiz // Naukosfera. 2025. № 5-2. S. 1–6. DOI: 10.5281/zenodo.15574433.

13. Izrailov K.E. Problemnye voprosy geneticheskoj deevolyucii predstavlenij programmy dlya poiska v nih uyazvimostej i rekomendacii po ih razresheniyu // Trudy uchebnyh zavedenij svyazi. 2025. T. 11. № 1. S. 84–98. DOI: 10.31854/1813-324X-2025-11-1-84-98.

Информация о статье:
Статья поступила в редакцию: 24.10.2025; одобрена после рецензирования: 23.11.2025;
принята к публикации: 25.11.2025

Information about the article:
The article was submitted to the editorial office: 24.10.2025; approved after review: 23.11.2025;
accepted for publication: 25.11.2025

Информация об авторах:
Израилов Константин Евгеньевич, профессор кафедры прикладной математики и безопасности информационных технологий Санкт-Петербургского университета ГПС МЧС России (196105, Санкт-Петербург, Московский пр., д. 149), кандидат технических наук, доцент, e-mail: konstantin.izrailov@mail.ru, https://orcid.org/0000-0002-9412-5693

Information about authors:
Izrailov Konstantin E., professor of the department of applied mathematics and information technology security of Saint-Petersburg university of State fire service of EMERCOM of Russia (196105, Saint-Petersburg, Moskovsky ave., 149), candidate of technical sciences, associate professor, e-mail: konstantin.izrailov@mail.ru, https://orcid.org/0000-0002-9412-5693