МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМЕНИ Н.Г. ЧЕРНЫШЕВСКОГО»

Кафедра теоретических основ компьютерной безопасности и криптографии

Отслеживание потока данных в исходном коде программ на языке C

АВТОРЕФЕРАТ
дипломной работы

студента 6 курса 631 группы
специальности 10.05.01 Компьютерная безопасность
факультета компьютерных наук и информационных технологий
Сенокосова Владислава Владимировича

Научный руководитель
старший преподаватель _____ А.А. Лобов
19.01.2026 г.

Заведующий кафедрой
д. ф.-м. н., профессор _____ М.Б. Абросимов
19.01.2026 г.

Саратов 2026

ВВЕДЕНИЕ

Цифровые технологии стали неотъемлемой частью жизни, трансформируя взаимодействие, работу и доступ к информации. Они упрощают решение задач, что приводит к росту онлайн-сервисов – от электронной коммерции до соцсетей. Однако цифровизация сопровождается серьёзными угрозами безопасности. Ошибки в программной логике могут стать причиной утечек конфиденциальных данных, наносящих репутационный и финансовый ущерб.

Особую сложность представляет анализ крупных проектов, где ручной аудит кода невозможен из-за ограничений по времени и бюджету. Наиболее эффективным решением являются инструменты статического анализа, позволяющие автоматизировать поиск уязвимостей на ранних этапах разработки.

Актуальность работы обусловлена широким использованием языка C в критически важных системах (ОС, встроенные системы, серверные приложения). Низкоуровневый характер C даёт разработчику гибкость, но требует высокой дисциплины работы с памятью, что повышает риск уязвимостей, включая утечки информации.

Цель работы – исследование и разработка методов обнаружения потенциальных утечек данных в исходном коде на C.

Задачи работы:

1. Изучить проблему утечек данных и их последствия.
2. Проанализировать теоретические основы статического анализа кода.
3. Сравнить возможности современных статических анализаторов.
4. Исследовать методы и алгоритмы статического анализа.
5. Изучить особенности языка C, связанные с управлением памятью.
6. Разработать статический анализатор для C-кода, способный находить утечки в отдельных файлах и проектах.

Дипломная работа состоит из введения, 3-х разделов, заключения, списка использованных источников и 2-х приложения. Общий объем работы – 108 страниц, из них 50 страниц – основное содержание, включая 21 рисунок, список использованных источников из 13 наименований.

КРАТКОЕ СОДЕРЖАНИЕ

В первом разделе «Актуальность проблемы утечек данных в современной кибербезопасности» исследуется проблема утечек информации как устойчивой и значимой угрозы в условиях цифровой трансформации. Проводится анализ современной цифровой среды, характеризующейся экспоненциальным ростом информационных систем, онлайн-сервисов и объёмов обрабатываемых конфиденциальных данных.

Утечка данных определяется как нарушение конфиденциальности информации, приводящее к её несанкционированному раскрытию или выходу за пределы контролируемой среды. Источники утечек классифицируются на внешние (злоумышленники) и внутренние (пользователи, ошибки разработки и эксплуатации ПО).

Анализируется современный ландшафт киберугроз, где утечки данных занимают центральное положение наряду с вредоносным ПО и сетевыми атаками. Устанавливается, что значительная доля инцидентов обусловлена не целенаправленными атаками, а дефектами программной реализации, некорректной обработкой данных и нарушениями бизнес-логики. Подчёркивается, что подобные уязвимости часто латентны на этапе разработки и манифестируют в фазе эксплуатации.

Исследуются статистические тенденции инцидентов, демонстрирующие устойчивый рост как количественных показателей, так и масштаба последствий. Согласно данным, в 4 квартале 2024 года общее число инцидентов увеличилось на 5% относительно предыдущего квартала и на 13% в годовом исчислении. Компрометация массивов персональных, коммерческих и служебных данных приводит к существенным финансовым потерям, регуляторным санкциям, судебным разбирательствам и длительной эрозии доверия.

На основе аналитических отчётов InfoWatch проводится обобщение статистики, подтверждающее, что утечки данных стабильно входят в число наиболее распространённых инцидентов, причём существенная их часть

коррелирует с ошибками программного обеспечения. Отмечается проблематика позднего обнаружения утечек, многократно увеличивающего совокупный ущерб.

Отдельный подраздел посвящён критической роли программного обеспечения на языке C в инфраструктурном контексте. Доказывается его широкое применение в разработке операционных систем, встроенных решений и высокопроизводительных компонентов. Устанавливается, что ошибки управления памятью, работы с указателями и межпроцедурной передачи данных являются доминирующим источником уязвимостей, потенцирующих утечки информации. Делается вывод о необходимости автоматизированных средств статического анализа ввиду низкоуровневой природы языка, требующей высокой дисциплины разработки.

Во втором разделе исследуются теоретические аспекты возникновения утечек информации на уровне программной логики и методологии их обнаружения.

Анализируются основные механизмы детектирования: динамический и статический анализ. Динамический анализ, основанный на наблюдении за поведением программы во время выполнения, позволяет выявлять реализуемые сценарии утечек, но требует наличия тестовых данных и не обеспечивает полного покрытия путей выполнения. Статический анализ, осуществляемый без запуска программы путём изучения её исходного кода, является более эффективным на этапах проектирования и разработки, поскольку выявляет потенциальные уязвимости.

Вводится понятие статического анализатора кода — программного средства для выявления дефектов и угроз безопасности путём анализа исходного кода. Рассматриваются ключевые этапы его работы: парсинг, построение абстрактного синтаксического дерева (AST), анализ вызовов функций и отслеживание потоков данных между переменными и модулями.

Определяются преимущества статического анализа: возможность раннего обнаружения уязвимостей в жизненном цикле ПО, отсутствие

необходимости выполнения программы, анализ всех потенциальных путей выполнения, масштабируемость на большие кодовые базы. Подчёркивается способность метода выявлять сложно воспроизводимые при тестировании ошибки, включая транзитивные утечки данных.

Отмечаются ограничения метода: риск ложноположительных срабатываний, сложность анализа динамических структур данных, указателей и внешних библиотек (особенно характерная для языка C), а также зависимость точности от совершенства алгоритмов и полноты описания опасных источников и приёмников данных.

Проводится обзор существующих решений для поиска утечек, анализируются их функциональные возможности и ограничения применительно к языку C. Устанавливается, что многие инструменты не в полной мере поддерживают обнаружение сложных сценариев утечек, связанных с транзитивными вызовами и межфайловым взаимодействием, что снижает их эффективность в реальных проектах.

В заключении раздела рассматриваются фундаментальные подходы статического анализа:

- анализ потоков данных (Data-Flow Analysis);
- taint-анализ (Taint Analysis);
- построение графов вызовов функций (Call Graph Construction);
- межпроцедурный анализ (Interprocedural Analysis).

Делается вывод, что комбинация указанных подходов позволяет повысить точность детектирования утечек информации и адекватно учитывать архитектурные особенности программ, реализованных на языке C.

Третий раздел посвящён практической реализации статического анализатора для отслеживания потоков данных в исходном коде программ на языке C.

В начале раздела описываются используемые инструменты разработки и программные средства. Обосновывается выбор технологий,

обеспечивающих корректный разбор исходного C-кода и построение абстрактного синтаксического дерева.

Абстрактное синтаксическое дерево (AST) – это иерархическое представление структуры программы, используемое для анализа её логики и взаимосвязей между элементами кода.

Далее рассматриваются применённые подходы анализа программы. Подробно описывается алгоритм детектирования опасных функций, анализ их аргументов и определение источников конфиденциальных данных. Особое внимание уделяется транзитивному анализу, позволяющему выявлять цепочки вызовов, через которые данные могут утекать опосредованно.

Отдельно описывается межфайловый анализ, обеспечивающий отслеживание потоков данных между различными файлами и модулями проекта. Этот подход позволяет выявлять утечки, распределённые по нескольким компонентам программы.

Приводится описание функциональных возможностей разработанного анализатора, включая:

- конфигурируемый список опасных функций;
- анализ аргументов вызовов;
- многоформатную систему отчетности;
- визуализацию результатов анализа;
- поддержку анализа заголовочных файлов и библиотек.

Конфигурационный файл для программы представлен на рисунке 1. Он содержит в себе всю необходимую информацию о путях, где хранятся компиляторы, входных и выходных файлах, режимах работы и опасных функциях с их описанием:

Рисунок 1 – Содержимое конфигурационного файла json_launch.json

В качестве тестовой программы будет рассмотрена следующая (см. рис. 2):

```c
#include <stdio.h>
#include <locale.h>

const int INTER_VAal = 10000;
int *buff = 345;
int bbb;
#define sdf;

void new_fun(int type_1) {
    int new_1 = 1331;
    printf("Helloo world %d", new_1);
}

int main() {

    setlocale(LC_ALL, "");

    FILE *input_file, *output_file;
    char ch;
    FILE *name = "input.txt";
    float as;
    input_file = fopen(name, "r");
    output_file = fopen("output.txt", "w");

    if (input_file == NULL || output_file == NULL) {
        printf("Error.\n");
        return 1;
    }

    while ((ch = fgetc(input_file)) != EOF) {
        fputc(ch, output_file);
    }

    fclose(input_file);
    fclose(output_file);

    printf("Copy data from input.txt to output.txt.\n");

    new_fun(INTER_VAal);
    return 0;
}
```

Рисунок 2 – Программа test_1.c опасными функциями для тестирования программы parse_nodes.py

```
ПРЯМЫЕ ВЫЗОВЫ ОПАСНЫХ ФУНКЦИЙ

[1] new_fun ──▶ printf("Helloo world %d", new_1)
  │ └ <Файл: ./Tests/test_1.c, Строка: 11, Столбец: 5, Символ: 173>

[2] main ──▶ fopen(name, "r")
  │ └ <Файл: ./Tests/test_1.c, Строка: 24, Столбец: 18, Символ: 479>

[3] main ──▶ fopen("output.txt", "w")
  │ └ <Файл: ./Tests/test_1.c, Строка: 25, Столбец: 19, Символ: 516>

[4] main ──▶ printf("Error.\n")
  │ └ <Файл: ./Tests/test_1.c, Строка: 29, Столбец: 9, Символ: 711>

[5] main ──▶ printf("Copy data from input.txt to output.txt.\n")
  │ └ <Файл: ./Tests/test_1.c, Строка: 42, Столбец: 5, Символ: 1148>

[6] main ──▶ fgetc(input_file)
  │ └ <Файл: ./Tests/test_1.c, Строка: 34, Столбец: 18, Символ: 967>

[7] main ──▶ fputc(ch, output_file)
  │ └ <Файл: ./Tests/test_1.c, Строка: 35, Столбец: 9, Символ: 1005>

[8] main ──▶ fclose(input_file)
  │ └ <Файл: ./Tests/test_1.c, Строка: 39, Столбец: 5, Символ: 1095>

[9] main ──▶ fclose(output_file)
  │ └ <Файл: ./Tests/test_1.c, Строка: 40, Столбец: 5, Символ: 1120>
```

Рисунок 3 – Данные о прямом вызове функций

В результате был получен такой результат report.txt, в котором есть указание прямых вызовов (см. рис. 3):

Цепочки вызовов функций показаны на рисунке 4

```
                    ПОЛНЫЕ ЦЕПОЧКИ ВЫЗОВОВ ДО ОПАСНЫХ ФУНКЦИЙ

Формат: Функция(аргументы) [файл:строка] → ... → ОпаснаяФункция(аргументы)

[1] ОБЫЧНЫЙ ВЫЗОВ
──────────────────────────────────────────────────────────────────────────────
   new_fun()
   └─ Файл: ./Tests/test_1.c:11
      ↓
   printf("Helloo world %d", new_1)
   └─ <Файл: ./Tests/test_1.c, Строка: 11, Столбец: 5, Символ: 173>

[2] ОБЫЧНЫЙ ВЫЗОВ
──────────────────────────────────────────────────────────────────────────────
   main()
   └─ Файл: ./Tests/test_1.c:39
      ↓
   fclose(input_file)
   └─ <Файл: ./Tests/test_1.c, Строка: 39, Столбец: 5, Символ: 1095>

[3] ОБЫЧНЫЙ ВЫЗОВ
──────────────────────────────────────────────────────────────────────────────
   main()
   └─ Файл: ./Tests/test_1.c:34
      ↓
   fgetc(input_file)
   └─ <Файл: ./Tests/test_1.c, Строка: 34, Столбец: 18, Символ: 967>

[4] ОБЫЧНЫЙ ВЫЗОВ
──────────────────────────────────────────────────────────────────────────────
   main()
   └─ Файл: ./Tests/test_1.c:35
      ↓
   fputc(ch, output_file)
   └─ <Файл: ./Tests/test_1.c, Строка: 35, Столбец: 9, Символ: 1005>

[5] ОБЫЧНЫЙ ВЫЗОВ
──────────────────────────────────────────────────────────────────────────────
   main()
   └─ Файл: ./Tests/test_1.c:29
      ↓
   printf("Error.\n")
   └─ <Файл: ./Tests/test_1.c, Строка: 29, Столбец: 9, Символ: 711>

[6] ПРЯМАЯ ОПАСНОСТЬ
──────────────────────────────────────────────────────────────────────────────
   main()
   └─ Файл: ./Tests/test_1.c:44
      ↓
   new_fun(INTER_VAal)
   └─ <Файл: ./Tests/test_1.c, Строка: 44, Столбец: 5, Символ: 1208>

[7] ОБЫЧНЫЙ ВЫЗОВ
──────────────────────────────────────────────────────────────────────────────
```

Рисунок 4 – Цепочки вызовов функций и подробным описанием

Детальный отчет report.txt содержит более содержательный отчет, с указанием функций, где были найдены опасные функции см. рис. 5

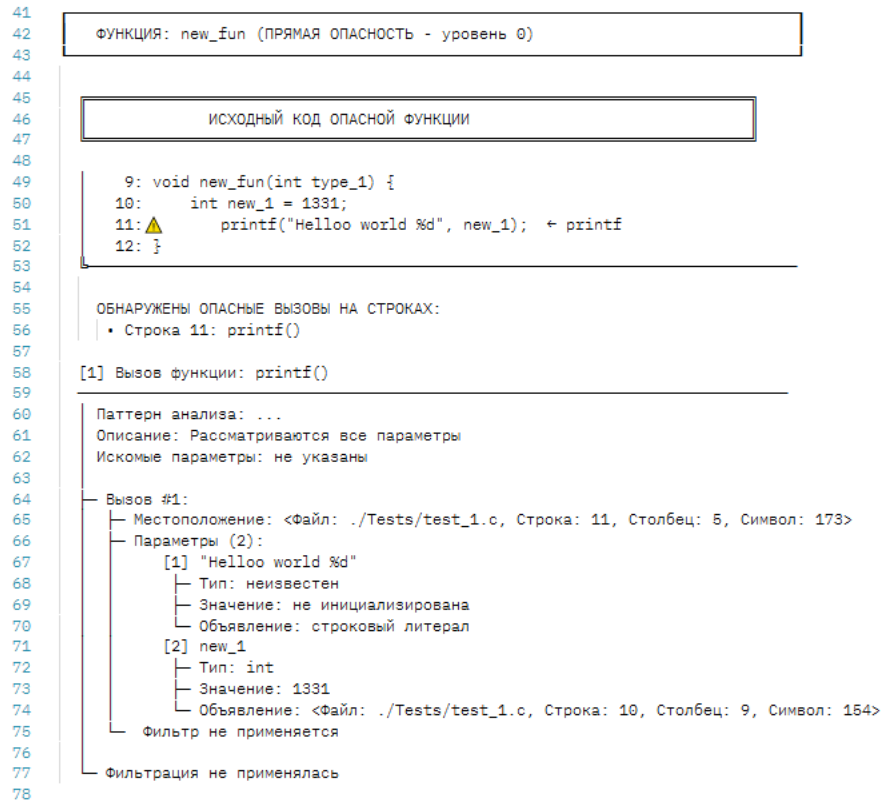

Рисунок 5 – Подробная статистика, представленная в report.txt

Результаты тестирования наглядно доказали эффективность разработанного статического анализатора. Инструмент корректно выполняет свою основную функцию: обнаруживает участки кода, где конфиденциальные данные могут попасть в опасные функции, как напрямую, так и через сложные цепочки вызовов между модулями. Программный комплекс готов к решению прикладных задач аудита безопасности C-программ, а модульная архитектура и многоформатная система отчётов открывают возможности для его адаптации и интеграции в современные CI/CD-конвейеры.

ЗАКЛЮЧЕНИЕ

Проведённое исследование комплексно рассмотрело проблему утечек данных в ПО, особенно в программах на C, лежащих в основе критической инфраструктуры. Изучение статистики инцидентов подтвердило актуальность разработки автоматизированных средств защиты. Теоретический анализ методов статического анализа (потоки данных, символьное выполнение, taint-анализ) заложил основу для практической реализации. Сравнение существующих инструментов выявило необходимость поддержки транзитивного и межфайлового анализа для обнаружения сложных уязвимостей.

В результате разработан статический анализатор для C-кода, способный находить не только прямые вызовы опасных функций, но и сложные сценарии утечек благодаря транзитивному и межфайловому анализу. Ключевое достижение – возможность отслеживать цепочки вызовов через несколько функций и модулей, выявляя уязвимости, не обнаруживаемые базовыми методами.

Тестирование подтвердило работоспособность системы для проектов разной сложности. Инструмент готов к интеграции в процесс разработки для автоматизированного аудита безопасности и может стать основой для более сложных систем. Перспективы развития включают расширение языковой поддержки, увеличение глубины анализа и адаптацию к современным парадигмам программирования.