Московский государственный университет имени М.В.Ломоносова
Факультет вычислительной математики и кибернетики
Кафедра системного программирования

Субботин Даниил Николаевич

Поиск некорретного использования библиотечных функций языков C/C++ методами статического анализа

КУРСОВАЯ РАБОТА

Научный руководитель:
старший научный сотрудник ИСП РАН
Гетьман Александр Игоревич

Научный консультант:
к.ф.-м.н.
Бородин Алексей Евгеньевич

Москва, 2022

Аннотация

Поиск некорретного использования библиотечных функций
языков C/C++ методами статического анализа

Субботин Даниил Николаевич

В работе рассмотрены различные ошибки при работе с библиотечными функциями языков C и C++. Описана разработка детекторов для инструмента статического анализа исходного кода Svace, приведены результаты работы данного детектора на тестовых функциях Juliet и реальных проектах с открытым исходным кодом.

Содержание

1 Введение

1.1 Опредение проблемы

В работе рассматриваются функции стандартных библиотек C и C++ (такие как `open`, `close`, `printf`, `scanf` и т.д.). Под некорректным использованием мы будем понимать как некорректное использование самой функции и работы с ее аргуменатми, так и неправильное использование ее результата.

1.1.1 Использование несогласованной функции

Под использованием несогласованной функции будем понимать ситуацию, при которой в программе используется некорректная функция с точки зрения логики работы программы. Такая ситуация может произойти, например, при работе с файловыми дескрипторами и при управлении динамической памятью. Проблема заключается в том, что система типов языков C/C++ позволяет использовать один и тот же тип данных для хранения разных по смыслу объектов. Например, целое число может являться как файловым дескриптором, так и дескриптором сокета или указатель на память может быть создан как с помощью функции malloc, так и с помощью new. Из-за этого возможно использование неправильной функции, несмотря на то, что с точки зрения типов, вызов функции будет корректным. В Листинг 1 приведен пример ошибочного использования функции – открытие файла с помощью `open` и его закрытие с помощью `fclose` вместо функции `close`.

```
void func() {
    int data;
    data = open("file.txt", FLAGS);
    fclose((FILE *)data);
}
```

Листинг 1: Вызов неправильной функции

1.1.2 Неправильное использование результата функции

Ситуация, в которой результат работы библиотечной функции неправильно обрабатывается. Рассматриваются функции, возвращающие целочисленное значение. В этом

случае проблема заключается в том, что как при корректном завершении, так и при завершении с ошибкой, функция возвращает одинаковый объект – целое число. Это позволяет неоднозначно трактовать возвращаемое значение, что может привести к ошибке. Так, в примере ниже, функция `putchar` при ошибке возвращает значение EOF, однако возвращаемое значение сравнивается с 0, что является некорректным. (Листинг 2).

```
void func() {
    int res = putchar((int)'A');
    if (res == 0) {
        exit(1);
    }
}
```

Листинг 2: Неправильная проверка значения

1.2 Статический анализатор Svace

Существует несколько подходов к анализу кода. В работе рассматривается поиск ошибок неправильного использования библиотечных функций с использованием статического анализа исходного кода. Статический анализ проводится без выполнения программы. Кроме того, при статическом анализе исследуются все пути выполнения программы, что позволяет обнаружить ошибки в редко достигаемых участках кода. Данная работа выполнена в рамках инструмента статического анализа *Svace*, разрабатываемый в ИСП РАН.

1.3 Цель и задачи работы

Целью данной работы является создание детектора для инструмента статического анализа *Svace*, обнаруживающего некорретное использование библиотечных функций языков C и C++. В ходе работы были выделены такие задачи: изучить принцип работы *Svace*, разработать и реализовать анализатор для каждого из обозначенных случаев ошибочного использования функций.

2 Обзор Svace

Анализ с использованием *Svace* происходит в несколько этапов. При анализе программы происходит перехват команды запуска компилятора и компоновки. Затем запускается компилятор *Svace*, который генерирует внутреннее представление для исходного кода программы. Файлы внутреннего представления сохраняются на диск для последующего анализа. При запуске анализа к внутреннему представлению применяется набор правил, предоставляемых подключенными детекторами для обнаружения определенных видов дефектов. Результатом анализа является список предупреждений.

Для анализа кода строится граф вызовов, который после обходится, начиная с листьев. Каждая функция анализируется только один раз и на основе собранной информации, создаётся резюме, которое содержит описание наиболее важных побочных эффектов от её вызова в произвольном контексте [1]. При этом детекторы могут сохрать в резюме нужные им данные. Такой подход позволяет избежать повторного анализа, поскольку при каждом анализе вызова функций будет использоваться только их резюме.

Для поддержки расширения в *Svace* выделяется структура, содержащая общие части для всех видов анализов, и модули, ответственные за поиск предупреждений определённого типа. Эти модули регистрируются в диспетчере расширений и затем получают уведомления о нужных им событиях, например вызов конкретной функции, разыменование указателя, операция сравнения и т.д. Для описания анализируемых свойств значений в *Svace* используются атрибуты. Каждый модуль работает со своим набором атрибутов, которые отражают интересующие свойства анализируемого кода.

Для анализа библиотечных функций в *Svace* используются спецификации. Спецификация – это определение функции, которое написано на анализируемом языке. Их наличие позволяет анализировать программы, не имея доступа к исходному коду используемых в ней библиотек. Для создания таких спецификаций в *Svace* используются особые *спецфункции*. С их помощью можно задавать необходимые атрибуты переменным в описании библиотечных функций [2]. Таким образом, каждый модуль работает со своим набором спецфункций и реагирует, если в спецификации вызванной функции была интересующая его спецфункция. В качестве примера рассмотрим спецификацию функции `strcat`:

```
char *strcat(char *s, const char *append) {
    sf_set_trusted_sink_ptr(s);
    sf_append_string(s, append);
    sf_null_terminated(s);
}
```

Листинг 3: Спецификация для strcat

В данном коде содержатся 3 спецфункции: sf_set_trusted_sink_ptr() — данная спецфункция показывает, что ее аргумент должен быть из надежного источника, иначе данный указатель может вызвать уязвимость, sf_append_string() — данная спецфункция показывает, что выполнено добавление одной строки к другой, sf_null_terminated() — данная спецфункция показывает, что строка заканчивается нулевым символом.

Основной анализ *Svace* является потоковочувствительным. Это значит, что анализ учитывает порядок инструкций. При таком подходе все атрибуты распространяются по графу потока управления функции. Поскольку в разных точках графа один и тот же атрибут может иметь разные значения, то в точках слияния путей графа для каждого атрибута анализатор вызывает функцию слияния, которая должна определить, каким образом несколько разных значений атрибута должны быть объединены для дальнейшего распространения.

3 Реализация

3.1 Детектор для проверки аргументов

Для обнаружения использования несогласованных функций был разработан потокочувствительный детектор `LIB_FUNC_INCOMPATIBLE`. Все анализируемые функции были разделены по типу принимаемого ими аргумента:

- `SocketCategory` — для функций работы с сокетами;
- `FileHandlerCategory` — для функций, работающих на низком уровне с файлами;
- `StdioHandlerCategory` — для функций, работающих на низком уровне напрямую с файловыми дескрипторами;
- `FilePointerCategory` — для функций, работающих с файловыми указателями;
- `MallocCategory` — для malloc-подобных функций;
- `NewCategory` — для функций динамического выделения и освобождения объектов;
- `NewArrayCategory` — для функций динамического выделения и освобождения массивов.

В ходе анализа существующих функций [3] было выяснено, что существуют функции, которые могут применяться как к сокету, так и к файлу, например, функции `write` и `close`. Поскольку категория у переменной может принимать только единственное значение, было принято решение выделить такие функции в отдельную категорию (`StdioHandlerCategory`).

В данном детекторе в качестве атрибута будет выступать категория переменной. Этот атрибут определяет спецфункция `sf_lib_arg_type(void *var, const char *category_type_name)`. В качестве параметров она принимает имя переменной и название категории, к которой эта переменная относится. Например, в спецификации функции `operator delete()` спецфункция показывает, что принимаемый параметр *ptr* должен иметь категорию `NewCategory`, а в функции `malloc()` – что возвращаемое значение имеет категорию `MallocCategory` (Листинг 4).

```
void operator delete(void* ptr) {
    sf_lib_arg_type(ptr, "NewCategory");
}

void* malloc(int size) {
    void* x;
    sf_lib_arg_type(x, "MallocCategory");
    return x;
}
```

Листинг 4: Пример спецификации

В основе алгоритма лежит метод межпроцедурного анализа на основе создания и последующего анализа резюме для функций. Детектор сохраняет в резюме информацию о категории формальной переменной. Поскольку *Svace* обходит граф вызовов функций в таком порядке, чтобы вызываемая функция анализировалась перед вызывающей, это значение распространяется до точки вызова функции. Обнаружение ошибок происходит при анализе вызова функции, которая имеет спецфункцию для данного плагина. Фактические переменные получают значение атрибута формальных переменных, и в случае, если эти категории не совпадают, создается предупреждение. Например, в следующем коде переменная p после вызова `malloc` получает атрибут `MallocCategory`, поэтому последующий вызов `delete` приведет к созданию предупреждения, поскольку в спецификации `delete` формальный параметр имеет атрибут `NewCategory` (Листинг 5).

```
void func() {
    int *p = (int *) malloc(sizeof(int));
    delete p;
}
```

Листинг 5: Неправильная работа с памятью

Также возможно обнаружение ошибок на этапе слияния путей графа потока управления. Если атрибут функции получает разные значения на разных путях выполнения, то в точке слияния создается предупреждение. В качестве примера рассмотрим код:

```
void func(bool boo) {
    int *ptr;
    if (boo) {
        ptr = (int *) malloc(sizeof(int));
```

```
    } else {
        ptr = new int;
    }
}
```

Листинг 6: Некорректное использование в условном операторе

Переменная ptr в разных ветках условного оператора инициализуется по-разному и дальнейшая работа с этим указателем будет некорректной, функция слияния может определить это, сравнив значения атрибутов, и выдать предупреждение.

3.2 Детектор для проверки возвращаемых значений

В данной работе рассматриваются только функции, которые возвращают целочисленное значение. Для обнаружения неправильных проверок возвращаемых значений функций был разработан детектор `LIB_WRONG_CHECK`. В этом детекторе функции были разделены по категориям по смыслу возвращаемого значения [3]:

- `EOF` — для функций, которые возвращают EOF в случае ошибочной работы;
- `Scan` — подмножество предыдущей категории. В этот класс были выделены функции, предназначенные для чтения данных с использованием строки формата и которые могут возвращать целые числа больше нуля;
- `NonZero` — для функций, которые возвращают ненулевое значение в случае ошибочной работы;
- `Negative` — для функций, которые возвращают отрицательное значение в случае ошибочной работы.

Кроме того, для функций категории `Scan` атрибут запоминает количество значений, которые должны быть считаны. Это необходимо для определения неправильного сравнения. Аналогично предыдущему плагину, в этом плагине атрибут так же может иметь только единственное значение. Этот атрибут определяет спецфункция, которая в качестве параметров она принимает имя переменной и название категории, к которой эта переменная относится.

Данный плагин реагирует на каждое условный переход в коде и проверяет, имеет ли функция в левой части условия интересующий атрибут – то есть имеет ли функция

отношение к выделенным классам. Если да, то анализуется правая часть выражения и знак сравнения. Основная проблема этих функций заключается в том, что как при правильной работе, так и при ошибочной работе, функция корретно завершается и возвращает какое-то значение. При этом определить результат ее работы можно только обработав ее возвращаемое значение. Поэтому была проанализирована семантика всех возможных возвращаемых значений и составлены правила, по которым определяется некорретное использование:

- `для EOF` — если результат работы функции сравнивается с EOF, то допустимы только знаки сравнения == и !=. Если возвращаемое значение сравнивается с 0, то допустим любой знак, кроме равенства. Во всех остальных случая выражение игнорируется;

- `для Scan` — поскольку эта категория является расширением предыдущей, то сначала выполняется проверка для EOF-класса функций. Если ошибочное сравнение не обнаружено, то из атрибута читается значение, записанное на этапе создания резюме, которое отображает, сколько полей содержит строка формата. Это значение сравнивается со значением в правой части выражения. В случае несовпадения, выдается предупреждение. Знак сравнения при этом не играет никакой роли;

- `для NonZero` — если сравнение происходит с 0, то допустимы только знаки сравнения == и !=, иначе выдается предупреждение. Если результат работы сравнивается не с 0, то плагин игнорирует это выражение;

- `для Negative` — поскольку стандарт определяет только знак возвращаемого значения, то сравнение с -1 является ошибочным. В случае, когда происходит любое сравнение не с 0, выдается предупреждение.

Работу данного плагина можно продемонстрировать на примере функции `scanf`, которая в результате работы возвращает количество записанных аргументов:

```
void func() {
    const char *s;
    int n;
```

```
    if (sscanf(s, "some_format_string", &n, &n) < 3)
        ...
    if (sscanf(s, "some_format_string", &n, &n) == 1)
        ...
    if (sscanf(s, "some_format_string", &n, &n) != 2)
        ...
}
```

Листинг 7: Пример анализируемого кода

В первом и втором случае число аргументов, которое ожидается считать и число, с которым это значение сравнивается, различаются, поэтому в данных случаях будет выдано предупреждение. В третьем случае всё корректно – функция ожидает считать 2 аргумента, и ее возвращаемое значение сравнивается с 2. В задачи данного плагина не входило обнаружение несоответствий между строкой формата и аргументами – для этого уже разработан другой детектор.

4 Результаты

4.1 Juliet

Чтобы оценить эффективность детекторов, нужно собрать информацию об их истинных и ложных срабатываниях. Однако для реальных проектов таких данных просто не существует, поэтому для оценки работоспособности реализованных детекторов использовался набор тестов Juliet, в котором заранее известно количество ошибок и их местоположение в коде [4]. Juliet представляет собор набор небольших программ на языках C и C++ и демонстрирует типовые классы ошибок. Для каждого из реализованных плагинов имеется соответсвующий набор тестов в Juliet. Результаты работы детекторов приведены в таблице 2.

Название детектора	**Всего дефектов**	**Найдено дефектов**	**Покрытие**
LIB_WRONG_CHECK	270	216	80%
LIB_FUNC_INCOMPATIBLE	48	42	87%

Таблица 1: Результаты работы на Juliet

При этом часть тестов являются не совсем корректными. Например, в следующем коде первое сравнение помечено в Juliet как ошибочное, а второе – как правильное:

```
void func() {
    if (rename(old_name, new_name) == 0) {
        printf("rename failed!");
    }

    if (rename(old_name, new_name) != 0) {
        printf("rename failed!");
    }
}
```

Листинг 8: Некорректный тест в Juliet

В этом примере используется функция, которая при успешном переименовании файла возвращает 0, а при неуспешном – ненулевое значение. Поэтому, корректным будет сравнение результата работы с 0 как на равенство, так и на неравенство. Поскольку все детекторы разрабатываются для анализа реальных проектов, а не для покрытия

тестовых наборов функций, то из-за наличия таких тестов в Juliet, покрытие меньше, чем могло быть.

4.2 Реальные программы

4.2.1 busybox

busybox – это набор утилит для командной строки Linux [5]. В результате анализа этого проекта было найдено некорректное использование возвращаемого значения функции `scanf`. В качестве параметров для записи ей было передано 3 аргумента, однако результат работы сравнивается с 2 (Листинг 9).

```
if (len == 4 && sscanf(date_str, "%2u%2u%2u""%2u%2u%c" + 9, &ptm->
    tm_hour, &ptm->tm_min, &end) >= 2) {
    ...
}
```

Листинг 9: Ошибка в busybox

4.2.2 Tizen

Tizen v6.5 – это открытая операционная система на базе ядра Linux [6]. В результате анализа было найдено несколько похожих ошибок. Их суть сводится к тому, что сначала целое число используется как дескриптор файла, а затем – как сокет. Пример ошибочного кода представлен в (Листинг 10).

```
static int generate(int fd, void *ret_data, size_t length) {
    ...
    //здесь fd используется как файл
    lseek(fd, (off_t) 0, SEEK_SET);
    ...
    if ((r = pa_loop_write(fd, ret_data, length, NULL)) < 0 || (
       size_t) r != length) {
    ...
    }
    ...
}
```

```
ssize_t pa_loop_write(int fd, const void *buf, size_t count, int *
   type)
    ...
    //здесь fd используется как сокет
    if ((r = send(fd, buf, count, MSG_NOSIGNAL)) < 0) {
        ...
    }
}
```

Листинг 10: Ошибка в Tizen

4.3 Обобщенные результаты

Поскольку для Tizen детекторы выдали много срабатываний, то для оценки их работы были случайно выбраны и размечены предупреждения для каждого детектора. Для этого вычислилась хэш-сумма каждого предупреждения, затем эти значения были упорядочены по возрастанию, и для разметки были выбраны первые 20 предупреждений.

Название детектора	**Всего срабатываний**	**Выборка**	**% Истинных**
LIB_FUNC_INCOMPATIBLE	171	20	80%
LIB_WRONG_CHECK	965	20	60%

Таблица 2: Результаты работы на Tizen

Более низкая точность второго детектора связана с тем, что спецификации функций не совсем точно отражают все возможные случаи обработки их возвращаемых значений, поэтому выше количество ложных срабатываний.

5 Заключение

В процессе работы были исследованы ошибки при использовании библиотечных функций. Для поиска таких ошибок были разработаны детекторы для инструмента статического анализа Svace, созданы спецфункции для спецификации функций, а также реализованы атрибуты для классификации переменных и возвращаемых значений функций.

В дальнейшем планируется улучшить анализ, дополнив и уточнив спецификации, а также расширив множество анализируемых функций.

Список литературы

[1] *Е., Бородин А.* Поиск уязвимостей небезопасного использования помеченных данных в статическом анализаторе Svace / Бородин А. Е., Горемыкин А. В., Вартанов С. П., Белеванцев А. А. // *Труды Института Системного Программирования РАН.* — 2021. — Т. 33, № 1. DOI: 10.15514/ISPRAS-2014-26(1)-7.

[2] *Иванников, В. П.* Статический анализатор Svace для поиска дефектов в исходном коде программ / В. П. Иванников, А. А. Белеванцев, А. Е. Бородин, В. Н. Игнатьев, Д. М. Журихин, А. И. Аветисян, М. И. Леонов // *Труды Института Системного Программирования РАН.* — 2014. — Т. 26, № 5. DOI: 10.15514/ISPRAS-2014-26(1)-7.

[3] Интерактивная система просмотра системных руководств. `https://www.opennet.ru/man.shtml`.

[4] Набор тестов Juliet. `https://samate.nist.gov/SRD/testsuite.php`.

[5] busybox GIT Repositories. `https://github.com/mirror/busybox`.

[6] Tizen Project Staging GIT Repositories. `https://git.tizen.org`.