



Анализ кода и информационная безопасность

Лекция 04

Статический анализ исходного кода с целью поиска ошибок



МГУ / ВМК / СП

0401

СТАТИЧЕСКИЙ АНАЛИЗ ИСХОДНОГО КОДА ДЛЯ ПОИСКА ОШИБОК

Анализ программ

- Анализ программы — выявление фактов о программе:
 - **статический анализ** — анализ программы без её запуска;
 - динамический анализ — анализ программы, использующий её выполнение (например, тестирование).
- Статический анализ:
 - оптимизация программ (компилятор);
 - рефакторинг (среда разработки);
 - автоматическое документирование (Doxygen);
 - обфускация и деобфускация программ (Dotfuscator, Ariadne);
 - оценка метрик программы (качество, сложность, время работы...);
 - поиск ошибок (FindBugs, Coverity, Klocwork, Svace).

Статический поиск ошибок

- Неформальный подход — поиск часто встречающихся ошибок:
 - перекрёстная проверка кода:
 - выполняется вручную;
 - у людей похожие «слепые пятна» при просмотре кода;
 - анализ на уровне синтаксиса — автоматический поиск ошибочных шаблонов в коде.
- Формальный подход — поиск всех ошибок или доказательство их отсутствия:
 - верификация — формальное доказательство соответствия программы её спецификации:
 - требует построения спецификации;
 - проверка свойств — например, «программа не выбрасывает `NullPointerException`».

Анализ на уровне синтаксиса

- Автоматизация ручной проверки кода.
- Более объективный анализ (нет «слепых пятен»).
- Поиск по достаточно ограниченным (локальным) шаблонам.
- Не подходит для проверки свойств на длинных путях программы.

```
if (X = Y) {  
    // ...  
}
```

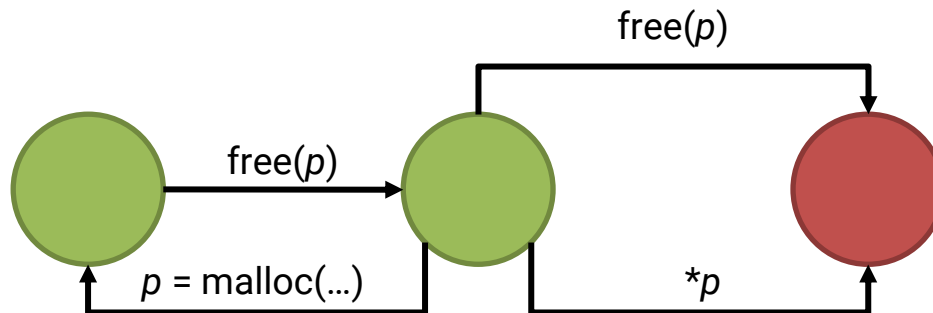
```
for (int X; X < Y; X++) {  
    // ...  
}
```

Верификация

- Золотой стандарт, даёт наибольшую гарантию корректности программы.
- Используются формальные методы:
 - доказательство теорем;
 - проверка моделей.
- Необходима полная спецификация программы.
- Анализ потока данных по всем возможным путям программы.
- Слишком дорогой метод для массового применения:
 - большие временные затраты по разработке спецификации;
 - высокие требования к разработчикам спецификации;
 - процесс проверки может быть очень долгим.
- Применяется в критически важных системах (космос, транспорт, энергетика...).

Проверка свойств

- Не требуется спецификация программы.
- Используются общие стандартные спецификации безопасности — описывается корректная работа с ресурсами:
 - «семафор должен быть освобождён после захвата и захвачен только после освобождения»;
 - «память не должна использоваться после того, как была освобождена, и не должна освобождаться дважды».
- В случае простых свойств для описания могут использоваться конечные автоматы.



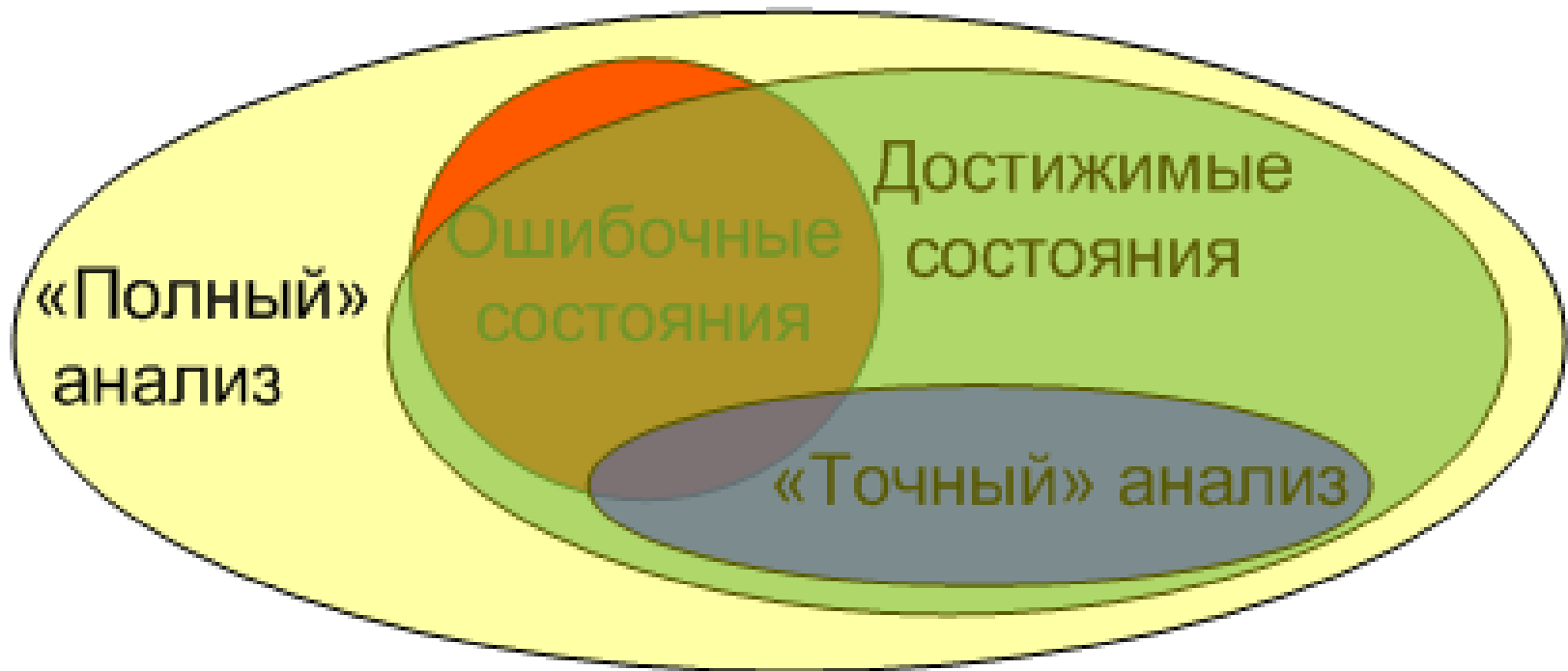
Ограничения формального подхода

- Задача автоматической проверки нетривиального свойства произвольной программы алгоритмически неразрешима (теорема Райса):
 - «есть ли в программе P ошибка класса x ?»;
 - «достигает ли программа P состояния Y ?»;
 - проблема останова.
- Невозможно реализовать универсальную проверку свойств, но для конкретной программы задача разрешима.
- Проверка свойств подразумевает две характеристики:
 - полнота — обнаруживаются все ошибки;
 - безошибочность — отсутствуют ложные срабатывания.
- Поиск ошибок (в отличие от верификации) — отказ от одного или обоих свойств.

Анализ состояний программы

- Отказ от безошибочности:
 - анализ надмножества возможных состояний;
 - гарантирует полноту (все истинные ошибки и, возможно, ложные срабатывания);
 - тривиальная реализация — ошибка в каждом операторе.
- Отказ от полноты:
 - анализ подмножества состояний;
 - гарантирует безошибочность (все выданные ошибки истинные);
 - тривиальная реализация — не выдавать срабатываний.

Анализ состояний программы



Основное ограничение статического анализа

- Экспоненциальный взрыв – быстрый рост числа путей в программе с увеличением объёма кода.
- Каждое ветвление увеличивает число путей в два раза.
- Что можно сказать про циклы?
- Вывод – анализ, проводимый вдоль всех путей, неприменим к реальным программам.
- Возможное решение – группировка путей, объединение состояний в точках слияния нескольких путей:
 - may-анализ – анализ надмножества состояний (объединение);
 - must-анализ – анализ подмножества состояний (пересечение).

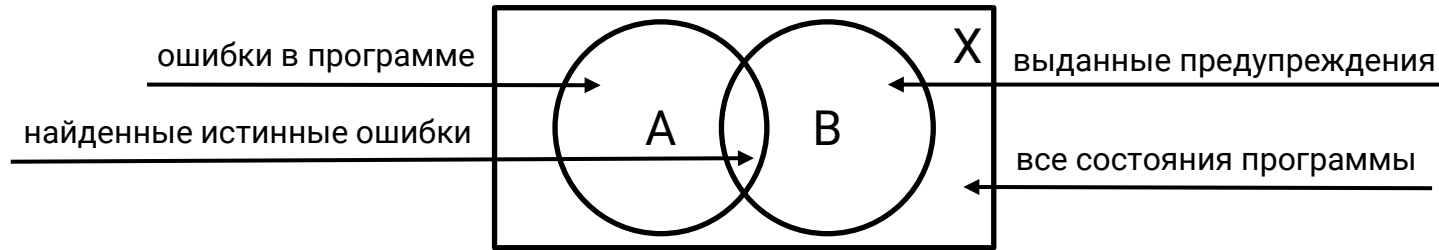
Пример тау-анализа

```
int
f(bool flag)
{
    char *buf;
    int i;

    if (flag) {
        i = 5;
        buf = malloc(10);           // i = 5, sizeof(buf) = 10
    } else {
        i = 15;
        buf = malloc(20);          // i = 15, sizeof(buf) = 20
    }

    // i ∈ { 5, 15 }, sizeof(buf) ∈ { 10, 20 }
    buf[i] = 0;
}
```

Оценка качества анализа



- $B \setminus A$ — ложные срабатывания (false positive);
- $A \setminus B$ — не найденные ошибки (false negative);
- $A \cap B$ — найденные истинные ошибки (true positive);
- $X \setminus A \setminus B$ — состояния без ошибок, не выданные анализатором (true negative);
- Характеристики:
 - **полнота** — доля найденных истинных ошибок среди всех ошибок: $\frac{|A \cap B|}{|A|}$;
 - **точность** — доля найденных истинных ошибок среди всех срабатываний: $\frac{|A \cap B|}{|B|}$.

0402

МОДЕЛИ ПРОГРАММЫ

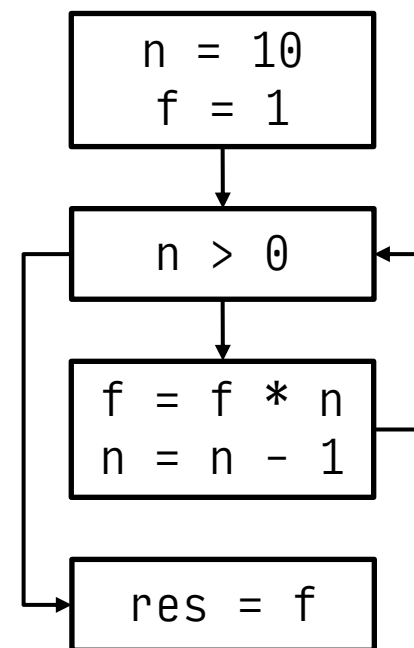
Модели программы

- Исходный код как представление для анализа неудобен.
- Используются модели из области компиляторных технологий.
- Последовательность лексем (лексический анализ).
- Абстрактное синтаксическое дерево (синтаксический анализ).
- Информация о типах (семантический анализ).
- Построение промежуточного представления, возможно, более высокоуровневого, чем в компиляторе.
- Построение модели программы, описывающей потоки управления и данных:
 - граф потока управления (control flow graph);
 - граф вызовов (call graph);
 - граф зависимостей по данным (dependence graph).

Граф потока управления

- Как правило, граф потока управления строится для каждой функции отдельно.
- Анализ по ГПУ без графа вызовов — **внутрипроцедурный**.
- Вершиной может являться базовый блок или отдельный оператор (тогда вершин больше).

```
int n = 10;  
int f = 1;  
  
while (n > 0) {  
    f = f * n;  
    n = n - 1;  
}  
  
res = f;
```



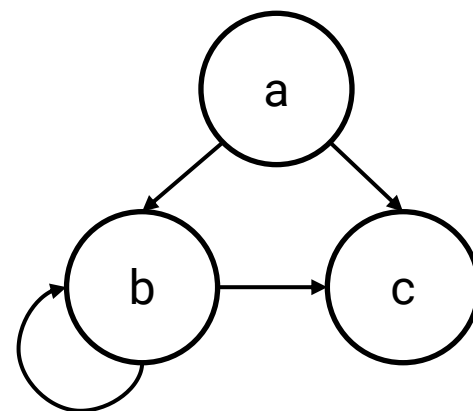
Граф вызовов

- Анализ, учитывающий граф вызовов, — **межпроцедурный**.
- Трудность: какие рёбра следует провести для вызова по указателю?
- Требуется анализ значений указателей.

```
int
a(int x)
{
    if (x) {
        b(1);
    } else {
        c();
    }
}

int
b(int y)
{
    if (y) {
        c(); b(0);
    } else {
        c();
    }
}

void
c(void) {}
```



Граф зависимостей

- Оператор s **зависит по данным** от оператора t , если существует путь от t к s и значение какой-либо переменной определяется в t и используется в s (**DU-зависимость**).
- Отношение «зависит по данным» может быть представлено в виде графа, в вершинах которого – операторы, а рёбра соответствуют наличию зависимости.
- На практике помимо DU-зависимостей интересны также:
 - чтение-запись (**UD-зависимость**);
 - запись-запись (**DD-зависимость**).
- В графе зависимостей также представлены зависимости по управлению.

Граф зависимостей

Какие рёбра в данном примере?

```
int n = 10;
int f = 1;

while (n > 0) {
    f = f * n;
    n = n - 1;
}

res = f;
```

`n = 10`

`f = 1`

`n > 0`

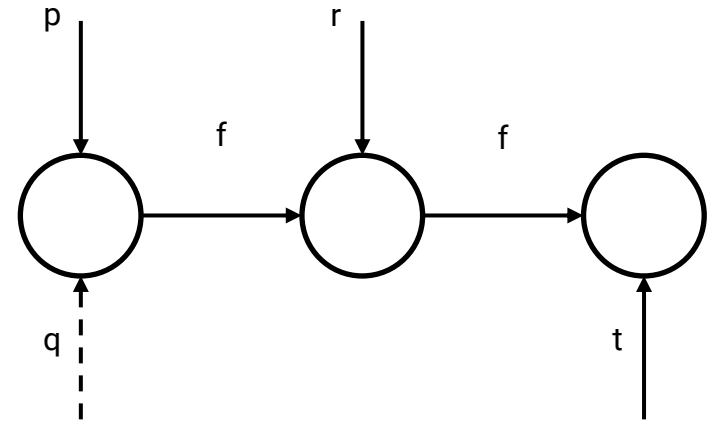
`f = f * n`

`n = n - 1`

`res = f`

Граф указателей

```
r = malloc(...);  
p->f = r;  
t = malloc(...);  
if (...) { q = p };  
r->f = t;
```



Связь между анализом потоков управления и данных

- Внутривпроцедурный анализ потока управления (ГПУ) обеспечивает:
 - определение последовательности анализируемых конструкций.
- Межпроцедурный анализ потока управления (ГВ) обеспечивает:
 - передачу параметров в вызываемые функции и возвращаемого значения в точку возврата.
- Анализ потока данных (ГЗ) обеспечивает:
 - определение возможных значений переменных во всех точках программы;
 - определение недостижимых переходов;
 - определение возможных целей вызова функции по указателю.

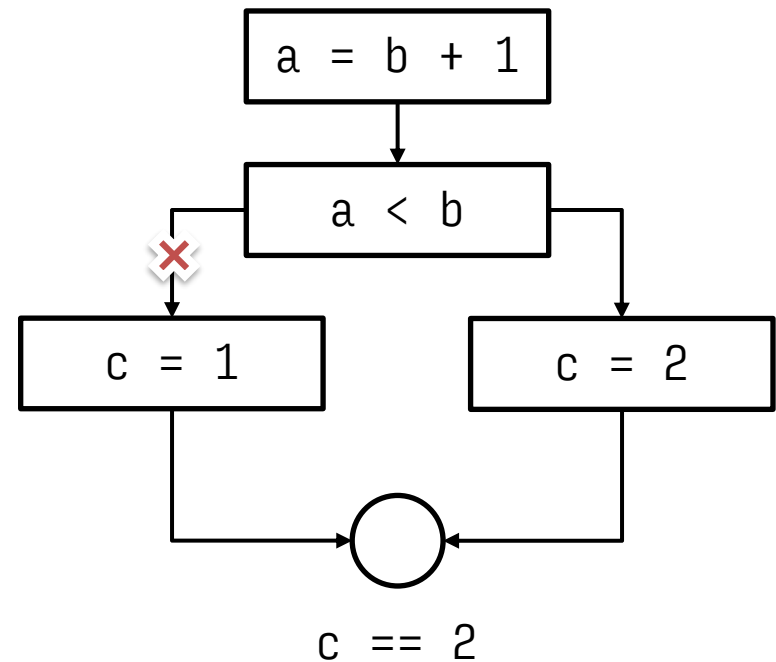
Недостижимые переходы

```
a = b + 1;  
if (a < b) {  
    c = 1;  
} else {  
    c = 2;  
}
```

Поток данных

- Всегда верно, что $a > b$.

Поток управления



Вызов функции через указатель

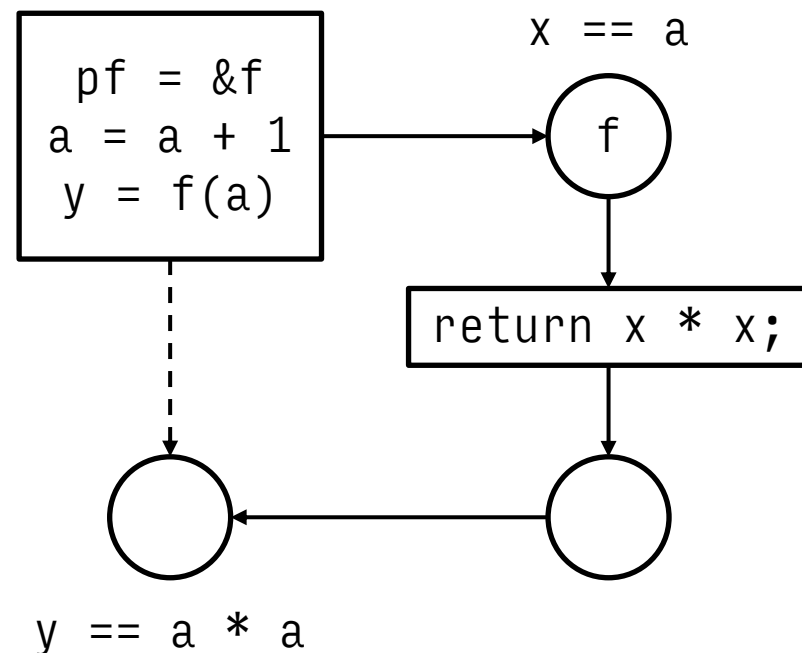
```
int
main(void)
{
    pf = &f;
    // ...
    a = a + 1;
    y = *pf(a);
    // ...
}

int
f(int x)
{
    return x * x;
}
```

Поток данных

- *pf* ссылается на *f*.

Поток управления

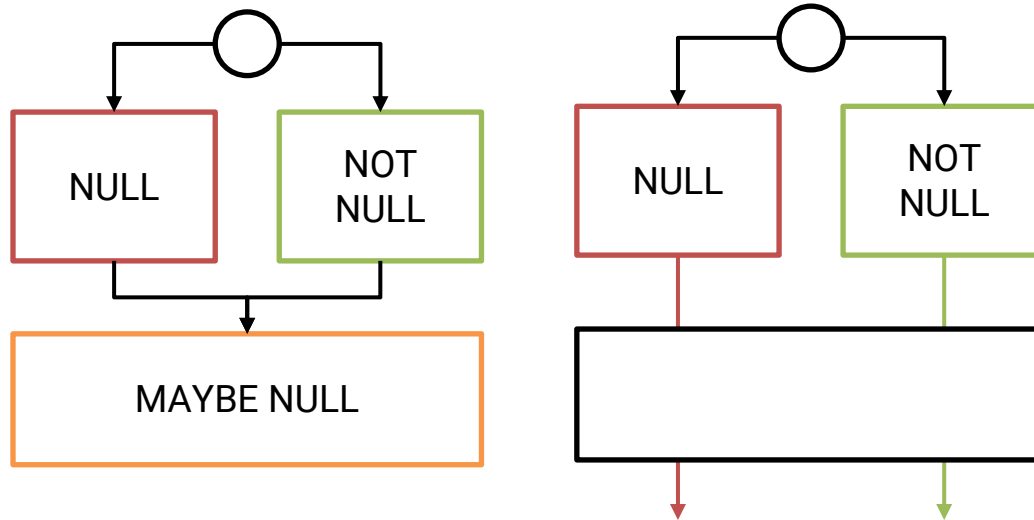


Алиасы (псевдонимы)

- Набор различных выражений, которые обозначают одну и ту же переменную.
- Примеры конструкций, порождающих алиасы:
 - использование указателей, в частности, указателей на функции;
 - элементы массивов (индексные выражения);
 - объединения (union).
- Алиасы представляют проблему для анализа потоков данных и управления:
 - состояние какой переменной меняется в инструкции?
 - какая функция (набор функций) будет (или может быть) вызвана по указателю в данной точке вызова?
- Анализ алиасов (в частности, анализ указателей).

Чувствительность анализа

Чувствительность к пути – способность анализа различать разные пути в программе.



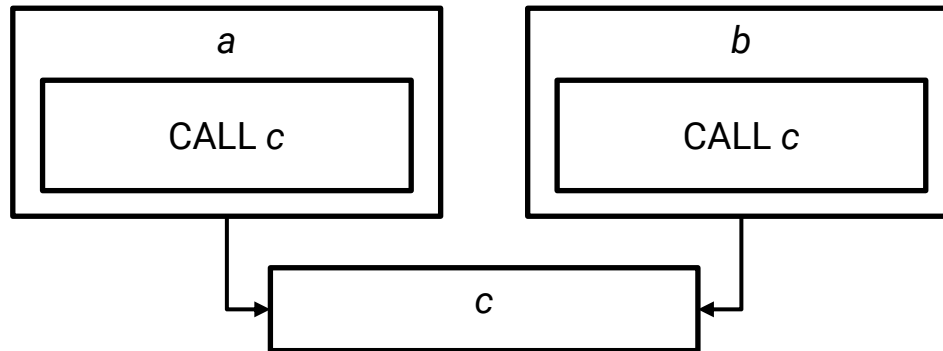
Чувствительность анализа

Чувствительность к потоку — способность анализа различать порядок следования операторов.

- «Указатель p может указывать на x ».
- «Указатель p может указывать на x перед выполнением оператора S ».

Чувствительность анализа

Чувствительность к контексту — способность анализа различать разные вызовы одной функции.



0403

КЛАССЫ ОБНАРУЖИВАЕМЫХ ОШИБОК - ПУТЬ РАСПРОСТРАНЕНИЯ ОШИБКИ

Классы обнаруживаемых ошибок

- Ошибки работы с ресурсами:
 - утечки памяти или других ресурсов;
 - неверная последовательность операций (например, двойное освобождение);
 - ошибки при работе с многопоточными примитивами.
- Ошибки ввода/вывода:
 - форматная строка.
- Арифметические ошибки:
 - деление на ноль.
- Использование неинициализированных значений.
- Ошибки работы с памятью:
 - разыменованние нулевого указателя;
 - выход за границы буфера.

Пример утечки памяти

```
int
f(void)
{
    int *p = malloc(SIZE);
    // ...
    return 0;
}
```

Утечка динамической
памяти при выходе из
функции

```
int
f(void)
{
    int *p = malloc(SIZE);
    int *q = malloc(SIZE);
    // ...
    if (flag) p = q;
    // ...
}
```

Утечка динамической
памяти при потере
указателя

Пример использования неинициализированных переменных

```
int
f(void)
{
    int i;
    if (i > 0) {
        // ...
    }
    // ...
}
```

Использование
неинициализированной
переменной

```
int
f(_Bool flag)
{
    int *p;
    if (flag) *p = 0;
    // ...
}
```

Разыменованное
неинициализированное
указателя

Пример ошибок ввода-вывода

```
int
f(void)
{
    char value[50];
    scanf("%49s", value);
    // ...
    printf(value);
    return 0;
}
```

Использование внешних
данных в качестве
форматной строки

```
int
f(void)
{
    char *s = malloc(10);
    // ...
    gets(s);
}
```

Использование опасных
функций

Путь распространения ошибки

- Цель обнаружения ошибки — её устранение.
- Ошибка обнаруживается в месте её проявления в программе (выполнение некорректного действия).
- Исправление может требоваться в другом месте.
- **Путь распространения ошибки** — поток данных в программе, приведший к ошибке, делится на 3 части:
 - **источник (source)** — место инициализации переменных, значения которых привели к ошибке;
 - **распространение (propagation)** — операторы, участвовавшие в обработке/передаче значений, которые привели к ошибке;
 - **сток, место проявления ошибки (sink)** — оператор, приводящий к ошибке.

Путь распространения ошибки

```
int
f(void)
{
    int i;          // источник
    // ...
    if (i > 0) {   // сток
        // ...
    }
    // ...
}
```

Использование
неинициализированной
переменной

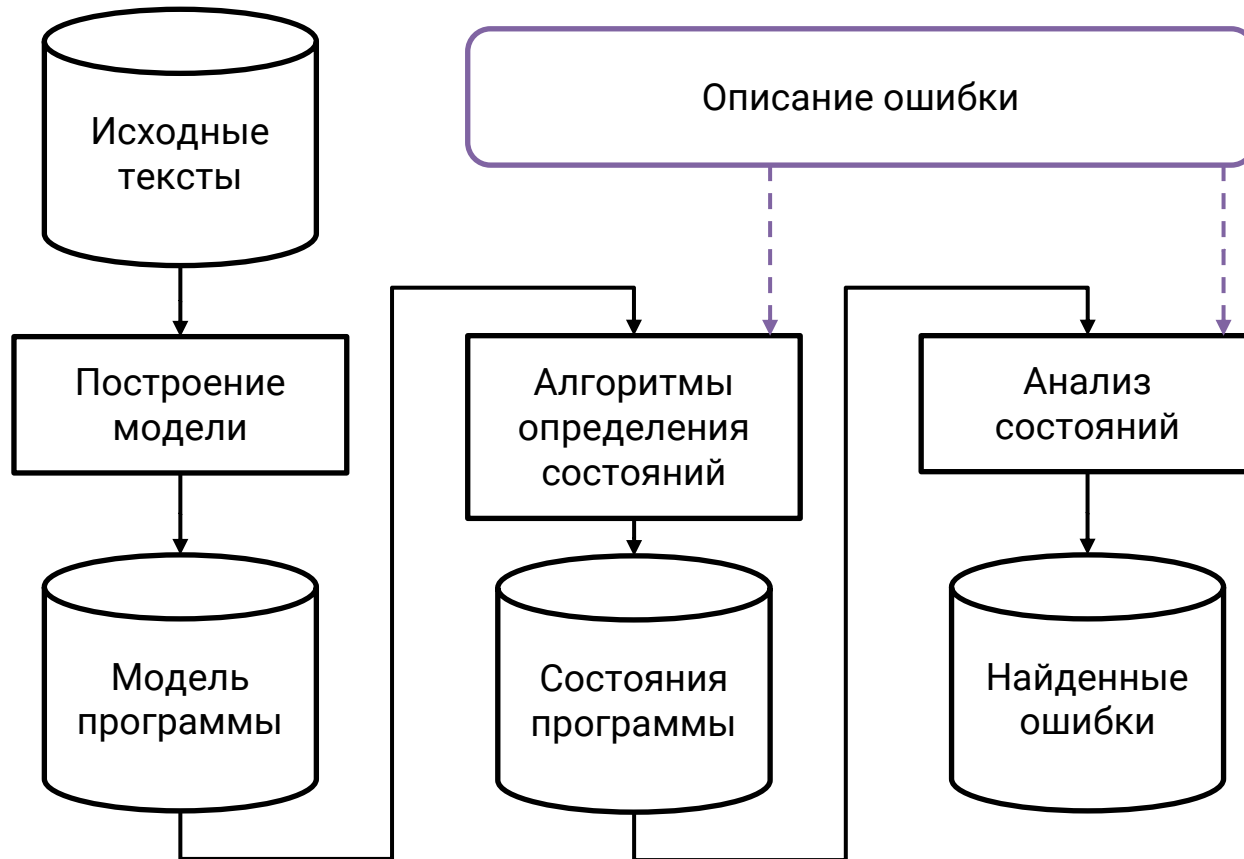
```
int
f(_Bool flag)
{
    char *buf;
    int j;
    int i = 4;     // источник 1
    if (flag) {
        // источник 2
        buf = malloc(10);
    } else {
        // источник 3
        buf = malloc(20);
    }
    j = i + 10;   // распростр.
    buf[j] = 0;  // сток
}
```

Переполнение буфера

0404

АЛГОРИТМ ПОИСКА ОШИБОК - ОГРАНИЧЕНИЯ АНАЛИЗА

Общая схема поиска ошибок



Описание ошибки

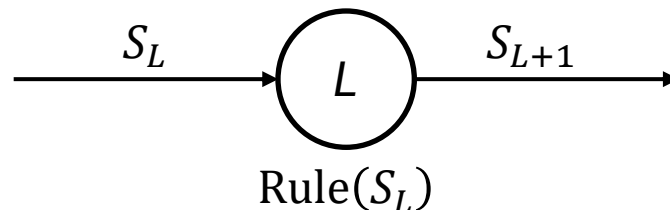
- Каждый тип ошибки характеризуется:
 - списком источников — операторов, порождающих значения, впоследствии приводящие к ошибке;
 - списком правил — описанием того, как различные операторы влияют на значение, приводящее к ошибке;
 - списком стоков — операторов, приводящих к проявлению ошибки, и задающим условия её возникновения.
- Пример для ошибки «разыменование нулевого указателя»:
 - источники — операторы инициализации указателей;
 - правила — операторы присваивания;
 - стоки — операторы разыменования указателей, при условии, что значение может быть равно нулю.

Анализ программ на основе состояний

- Определяются состояния объектов программы во всех её точках (т.е. между парами соседних операторов).
- Объектами выступают переменные (в т.ч. указатели), файлы, сетевые сокеты:
 - переменные — возможные значения;
 - указатели — равенство нулю, объекты, на которые может ссылаться указатель;
 - файл — открыт/закрыт, режим, положение файлового указателя.
- **Состояние программы** — возможные состояния всех объектов, видимых в данной точке программы $L: S_L = \{(object, state)\}$.

Алгоритмы определения состояний

- Для анализа состояний разных объектов применяются:
 - алгоритм анализа значений;
 - алгоритм анализа указателей;
 - алгоритм анализа ресурсов...
- Алгоритмы анализа представлены в виде правил для конструкций программы (разновидностей операторов).
- Каждое правило формирует уравнение над состояниями объектов программы: $\text{Rule}(S_L): S_L \rightarrow S_{L+1}$.
- Строится система уравнений, решение которой даёт состояния объектов программы во всех точках.



Алгоритм поиска ошибки

1. Найти в модели программы операторы-источники ошибки.
2. Проинициализировать состояния объектов в соответствии с классом ошибки.
3. Распространить состояния объектов по графу потока управления программы с учётом графа вызовов в соответствии с правилами ошибки – получить состояния в каждой точке программы .
4. Найти в модели программы операторы, которые могут приводить к проявлению ошибки.
5. Проверить в них состояние программы в соответствии с условиями возникновения ошибки.

Преимущества статического анализа

- Раннее обнаружение ошибок – во время разработки.
- Код при этом может не собираться и не запускаться.
- Полное покрытие кода.
- Не требуется задавать тестовые входные данные.
- Лучше проверяет участки кода, сложные для тестирования (обработка ошибок).

- Однако:
 - статический анализ не является заменой тестирования (не проверяет логику программ).

Ограничения статического анализа

- Отсутствие знаний о входных значениях (неопределённые значения переменных).
- «Экспоненциальный рост» числа путей в программе.
- Большая длина отдельных путей в программе (циклы, рекурсивные вызовы).
- Сложность анализа циклов и рекурсивных вызовов (сколько раз выполнялись?).
- Проблема алиасов.
- Возможность отсутствия части кода программы (библиотек).
- Большой объём данных описывающих состояния программы (возможные значения).

Возможные упрощения

- Сокращение числа анализируемых путей выполнения:
 - совместный анализ путей выполнения программы.
- Сокращение длины анализируемых путей:
 - ограничение на глубину стека вызовов;
 - ограничение на число итераций при анализе циклов.
- Сокращение объёма данных, связанных с путями:
 - сокращение количества анализируемых объектов, например представление массива одним объектом;
 - компактное хранение состояний объектов: вместо $\{ 1, 2, \dots, 10 \}$ – интервал $[1, 10]$.
- Возможность аннотации внешних функций в виде некоторой суммы их влияния на данные:
 - «OpenFile() возвращает созданный объект типа файл или 0».

Автоматизация

- Уровень автоматизации:
 - количество входных данных и действий, требуемых от пользователя перед началом анализа;
 - возможность предоставления дополнительных данных для уточнения анализа;
 - дополнительная постобработка выдаваемых ошибок.
- Входные данные и действия:
 - необходимость аннотации внешних функций;
 - необходимость создания специальной сборки проекта.
- Уточнение:
 - дополнительная аннотация кода.
- Постобработка выдаваемых ошибок:
 - возможность ранжирования выдаваемых ошибок по серьёзности, вероятности истинности и т.д.;
 - возможности по проверке истинности выданных ошибок.

Литература к лекции

Основные источники

1. Flemming Nielson, Hanne R. Nielson, Chris Hankin. Principles of Program Analysis / Springer, 1999.
2. Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, Dawson Engler. A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World // Communications of the ACM, 2010, vol. 53, no. 2, pp. 66-75.
3. William R. Bush, Jonathan D. Pincus, David J. Sielaff. A Static Analyzer for Finding Dynamic Programming Errors // Software – Practice and Experience, 2000, vol. 30, issue 7, pp. 775-802.
4. Альфред В. Ахо, Моника С. Лам, Рави Сети, Джеффри Д. Ульман. Компиляторы. Принципы, технологии и инструментарий / Вильямс, 2015.