Кафедра СП
Вопросы к государственному экзамену
Магистерская программа"Компиляторные технологии"

1. Основные понятия дедуктивной верификации. Методы доказательства корректности программ.

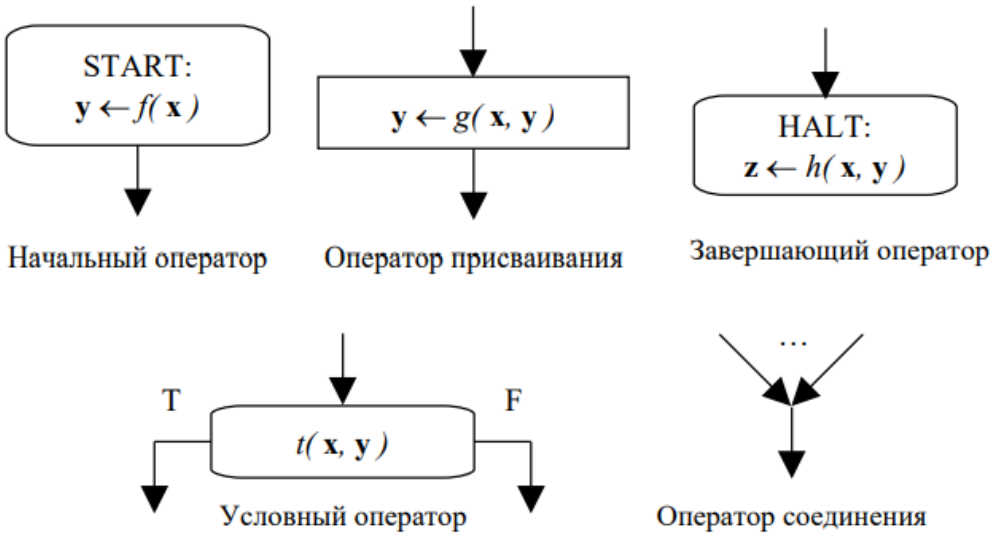
2. Основные понятия дедуктивной верификации. Методы доказательства завершимости программ.

1+2:
Целью любой верификации программы является установление соответствия программы ее требованиям. Дедуктивная верификация устанавливает это соответствие в виде логического вывода утверждения о том, что программа соответствует требованиям. При этом доказывается соответствие программы требованиям на всех входах программы.

Поскольку дедуктивная верификация основана на использовании логических заключений, она должна оперировать с формальными моделями как программ, требований, так и с формально определенным отношением их соответствия.

Математическая модель программы
Переменные программы.
входные, промежуточные и выходные -> x, y, z
Множество значений всех возможных переменных образует универсальный домен D. Кроме того, мы выделим два специальных значения: T (истина) и F (ложь).

Операторы программы
1. Начальный оператор START: y <- f(x). Здесь f является функцией Dx -> Dy
2. Оператор присваивания ASSIGN: y <- g(x, y). Здесь g является функцией Dx * Dy -> Dy
3. Условный оператор TEST: t (x, y). Здесь t является предикатом.
4. Оператор соединения JOIN
5. Оператор завершения HALT: z <- h(x, y). Здесь h является функцией Dx * Dy -> Dz



Блок-схемы

В качестве модели программы мы будем использовать блок-схемы. Блок-схемой называется тройка (V, N, E), где:

V – конечное множество переменных программы,

N – конечное множество операторов блок-схемы,

E – конечное множество связок блок-схемы, помеченных символами T, F или Epsilon.

Заметим, что блок-схема соответствует ориентированному графу, вершинами которого являются операторы программы, а ребрами – ее связки. При этом все ребра помечены одним из трех символов.

Корректно-определенной блок-схемой мы будем называть блок схему удовлетворяющую следующим требованиям:

1. В блок-схеме присутствует ровно один начальный оператор и не менее одного завершающего оператора.
2. Любой оператор находится на ориентированном пути от начального оператора к некоторому завершающему оператору.
3. Число связок, выходящих из каждого оператора, и пометки этих связок соответствуют типу оператора:
   a. Из начального оператора выходит ровно 1 дуга, помеченная символом Eps.
   b. Из оператора присваивания выходит ровно 1 дуга, помеченная символом Eps.
   c. Из условного оператора выходит ровно 2 дуги, причем одна из них помечена символом T, а другая – символом F.
   d. Из оператора соединения выходит ровно 1 дуга, помеченная символом Eps.
   e. Из завершающего оператора не выходит ни одной дуги.
4. Число связок, входящих в каждый оператор, соответствует его типу:
   a. В начальный оператор не входит ни одна дуга.
   b. В оператор присваивания, условный и завершающий оператор входит ровно одна дуга.
   c. В оператор соединения входит не менее одной дуги.

Математическая модель требований
Входной и выходной предикаты
Спецификацией Ф программы над переменными V мы будем называть два предиката:

- входной предикат Phi: Dx -> { T, F }
- выходной предикат Psi: Dx * Dz ->{ T, F }

Выходной предикат (или постусловие) определяет, какие значения выходных переменных программы являются допустимыми (правильными) относительно значений входных переменных. А входной предикат (или предусловие) определяет, при каких значениях входных переменных требуется выполнение ограничений, описанных в выходном предикате

Каждой блок-схеме P мы поставим в соответствие функцию M[P] из входного домена блок-схемы в выходной домен, расширенный специальным значением omega (M[P]: Dx -> Dz+). Если вычисление блок-схемы P на векторе входных переменных x является конечным, то функция M[P] (x) принимает значение h(x,yn), где h – функция завершающего оператора последней конфигурации вычисления, а yn – вектор значений промежуточных переменных из последней конфигурации вычисления. В противном случае функция M[P] (x) принимает значение omega.

Частичная и полная корректность программ
Пусть программа задана своей моделью в виде блок-схемы P, а ее спецификация Ф – предикатами Phi и Psi. Мы будем говорить, что:

- программа P частично корректна относительно Phi и Psi, если для любого вектора значений входных переменных x, такого что Phi(x) = T и M[P](x) != omega, выполнено ограничение Psi(x, M[P](x))=T. Частичную корректность программы P относительно Phi и Psimы будем обозначать {Phi}P{Psi}.

- программа P полностью корректна относительно Phi и Psi, если для любого вектора значений входных переменных x, такого что Phi(x) = T, выполнены ограничения M[P] (x) != omega и Psi(x, M[P](x)) = T. Полную корректность программы P относительно Phi и Psi мы будем обозначать <Phi>P<Psi>.

Заметим, что полная корректность программы P относительно входного предиката Phi и выходного предиката T эквивалентна тому, что программа P завершается всегда, когда вектор значений входных переменных удовлетворяет Phi. В этом случае мы будем говорить, что P завершается на Phi.

Лемма Пусть даны программа P и спецификация Ф = (Phi, Psi). В этом случае <Phi>P<Psi> тогда и только тогда, когда {Phi}P{Psi} и <Phi>P<T>

Исходя из данной леммы, для доказательства полной корректности программы необходимо и достаточно доказать ее частичную корректность и завершаемость.

ЧК: Метод индуктивных утверждений Флойда.
Завершаемость: метод фундированных множеств Флойда.

Метод индуктивных утверждений Флойда

Шаг 1

Выбрать множество *точек сечения* (множество связок такое, что каждый цикл блок-схемы содержит хотя бы одну связку из этого множества).

Шаг 2

Каждой точке сечения сопоставить *индуктивное утверждение*, т.е. предикат $p : D_{\bar{x}} \times D_{\bar{y}} \to \{T,\ F\}$. Псевдосвязке перед START сопоставить $p(x,\ y) \equiv \varphi(x)$. Каждой псевдосвязке после HALT с функцией h сопоставить $p(x,\ y) \equiv \psi(x, h(x, y))$.

Шаг 3

Выписать *условие верификации* для каждого *базового пути* α (т.е. пути без самопересечений, внутри которого нет т.с.) между точками сечения и псевдосвязками (началу пути сопоставлено p_1, концу пути — p_2):
$\forall \bar{x} \in D_{\bar{x}}, \bar{y} \in D_{\bar{y}}$
$\varphi(\bar{x}) \wedge p_1(\bar{x}, \bar{y}) \wedge R_\alpha(\bar{x}, \bar{y}) \Rightarrow p_2(\bar{x}, r_\alpha(\bar{x}, \bar{y}))$

Теорема

Дана произвольная блок-схема P и спецификация для нее $(\varphi,\ \psi)$. Пусть сделаны все шаги метода индуктивных утверждений. Тогда если все выписанные условия верификации истинны, то $\{\varphi\}\ P\ \{\psi\}$.

Замечание: иногда индуктивные утверждения называют *инвариантами циклов* (т.к. они должны быть выполнены всегда, когда вычисление программы находится в точке, куда они приписаны).

Метод фундированных множеств Флойда

Шаг 1

Выбор множества т.с. (все циклические пути имеют т.с.) и фундированного множества $(W, \prec)$.

Шаг 2

Выбор индуктивного утверждения для каждой т.с., выписывание условий верификации для каждого базового пути между точками сечения и псевдосвязкой у START.

Шаг 3

Выбор оценочной функции для каждой точки сечения $(u_A : D_{\bar{x}} \times D_{\bar{y}} \to W', W \subseteq W')$.

Шаг 4

Выписывание условия корректности оценочной функции для каждой точки сечения:
$\forall \bar{x} \in D_{\bar{x}}\ \forall \bar{y} \in D_{\bar{y}} \cdot \varphi(\bar{x}) \wedge p_A(\bar{x}, \bar{y}) \Rightarrow u_A(\bar{x}, \bar{y}) \in W$.

Шаг 5

Выписывание условия завершимости для каждого базового пути между точками сечения (из A в B):
$\forall \bar{x} \in D_{\bar{x}}\ \forall \bar{y} \in D_{\bar{y}} \cdot \varphi(\bar{x}) \wedge p_A(\bar{x}, \bar{y}) \wedge R_\alpha(\bar{x}, \bar{y}) \Rightarrow$
$u_B(\bar{x}, r_\alpha(\bar{x}, \bar{y})) \prec u_A(\bar{x}, \bar{y})$.

Теорема

Дана блок-схема P, спецификация $(\varphi,\ \psi)$. Если все составленные условия верификации, корректности и завершимости истинны, то $\langle\varphi\rangle\ P\ \langle T\rangle$, т.е. блок-схема завершима.

Схема доказательства: по индукции доказать выполнение индуктивных утверждений в точках сечения, из фундированности W сделать вывод об отсутствии бесконечных вычислений.

3. Основные сведения об объектном языке ограничений (OCL): состав OCL-выражения, навигация по ассоциациям, виды коллекций, операции с коллекциями, учёт наследования в выражениях и наследование ограничений. Примеры использования OCL.

Смотри отдельно!

4. Способы объектно-реляционного отображения для классов и атрибутов, бинарных и N-арных ассоциаций, классов ассоциаций, иерархий наследования. Примеры применения этих способов. Моделирование схемы реляционной базы данных с помощью диаграммы классов.

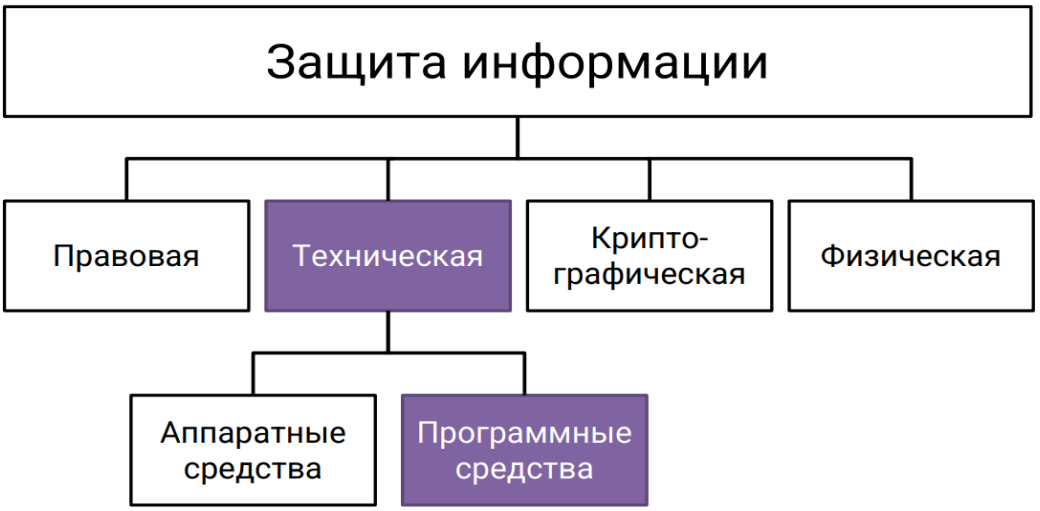
Смотри отдельно!

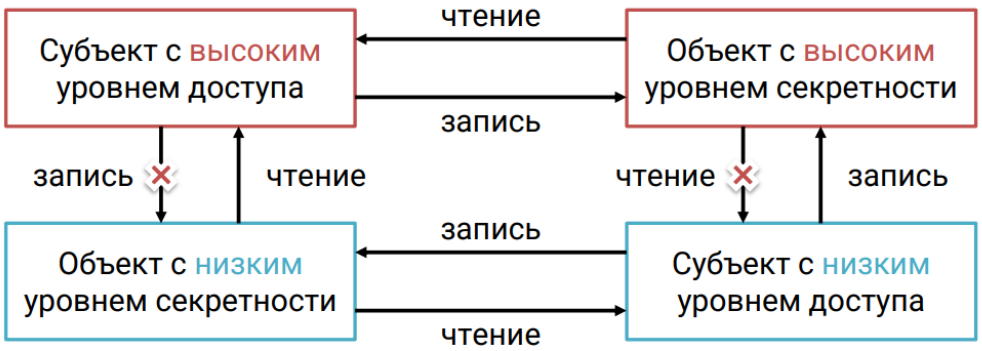
5. Образцы (паттерны) проектирования, их классификация и способ описания. Примеры образцов:структурного, поведенческого и порождающего.

Смотри отдельно!

6. Основные понятия безопасности информации: конфиденциальность, целостность, доступность. Виды защиты информации. Модель Белла-Лападулы. Понятие ошибки, уязвимости в программном обеспечении, примеры.

Конфиденциальность информации — состояние информации, при котором доступ к ней осуществляют только субъекты, имеющие на него право.
Целостность информации — состояние информации, при котором отсутствует любое её изменение либо изменение осуществляется только преднамеренно субъектами, имеющими на него право.
Доступность информации — состояние информации, при котором субъекты, имеющие права доступа, могут реализовать их беспрепятственно.



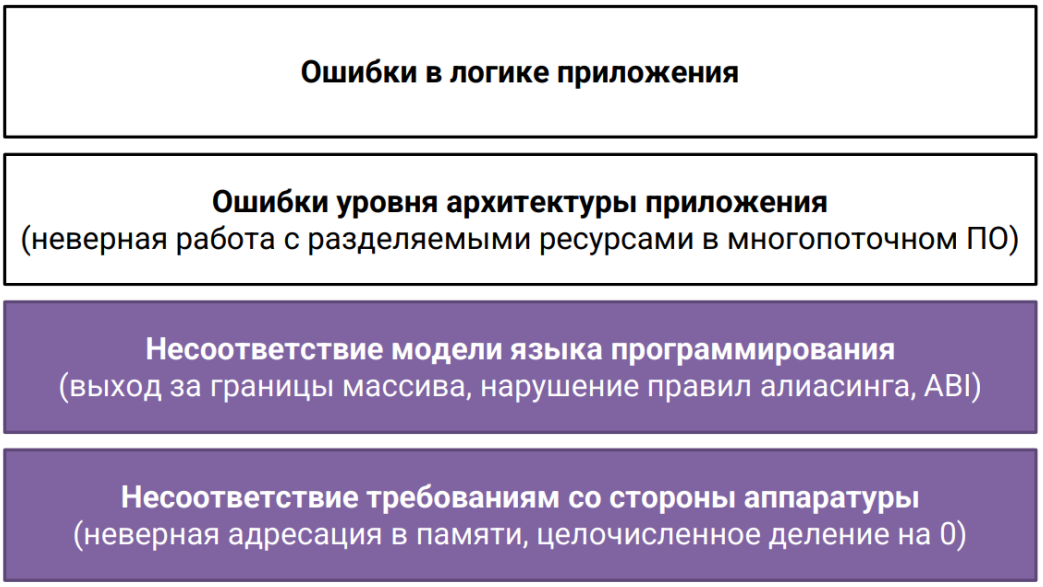


Модель Белла–Лападулы

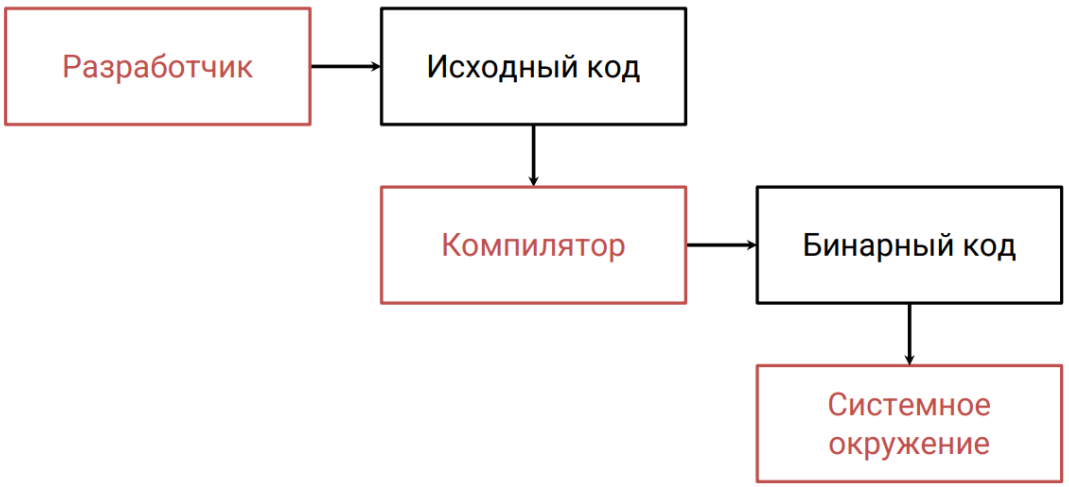
Уязвимость информационной системы — свойство информационной системы, обусловливающее возможность реализации угроз безопасности обрабатываемой в ней информации.

Угроза безопасности информации — совокупность условий и факторов, создающих потенциальную или реально существующую опасность нарушения безопасности информации.

Ошибки в программах



Ошибки и закладки



Нарушения безопасности в ПО

1. Нарушение **конфиденциальности**:
   - OpenSSL Heartbleed;
   - программная закладка в роутере;
   - шпионаж через веб-камеру в телевизоре.
2. Нарушение **целостности**:
   - программная закладка в роутере;
   - изменение базы данных с паролями через внедрение SQL-кода.
3. Нарушение **доступности**:
   - аварийное завершение программы;
   - отказ в обслуживании — DoS, DDoS.

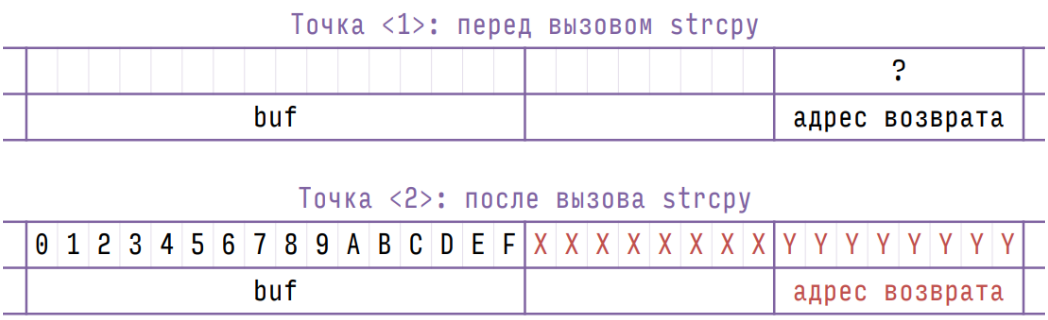
7. Ошибка типа «переполнение буфера». Выполнение произвольного кода на исполнимом стеке. Противодействие выполнению кода на стеке: «канарейка», DEP. Выполнение произвольного кода на неисполнимом стеке. Return-to-libc, return-orientedprogramming (ROP).

- Программа осуществляет запись в буфер, размещенный на стеке, по неверному индексу, превышающему наибольший допустимый.
- Ошибка типична для языков Си и Си++, а также для ассемблера.
- Возможные последствия эксплуатации уязвимости:
    - Нарушение доступности — аварийное завершение или зависание программы;
    - Нарушение конфиденциальности, целостности и доступности — перехват потока управления, внедрение произвольного кода.

```
#include <string.h>

#define BUFSIZE 16

int
main(int argc, char **argv)
{
    char buf[BUFSIZE];

    strcpy(buf, argv[1]);
    return 0;
}
```

? argv[1] = “0123456789ABCDEFXXXXXXXXYYYYYYYY”

```
004003F0:        48 83 EC 18         SUB    RSP, 0x18
004003F4:        48 8B 76 08         MOV    RSI, QWORD [RSI + 8] ; argv[1]
004003F8:        48 89 E7            MOV    RDI, RSP ; buf
004003FB:   <1>  E8 C0 FF FF FF      CALL   strcpy
00400400:   <2>  31 C0               XOR    EAX, EAX
00400402:        48 83 C4 18         ADD    RSP, 0x18
00400406:        C3                  RET
```



Проблемы

1. Нельзя записывать нулевой байт strcpy.
2. При возникновении раннего нуля, старший байт может быть неизвестным.

Решение

1. Начинать шелл-код со следующего байта.
2. Старший байт по стандартной карте адресов нулевой.

Подготовка эксплоита

1. Анализ исходного кода программы, обнаружение ошибки переполнения буфера.
2. Анализ бинарного кода программы, его разметка в соответствии с исходным.
3. Подготовка карты стека на момент переполнения:
   - расположение переменной `buf`;
   - расположение сохранённого адреса возврата.
4. Проработка допустимого формата содержимого эксплоита:
   - шаблон с расположением фиксированных полей (поле адреса возврата);
   - ограничения на значения отдельных байтов.
5. Оформление окончательного вида эксплоита: конкретные значения байтов с внедрённым шелл-кодом.

Условия реализации атаки

1. Исполнимый стек: внедряемый код размещён на стеке. Если система не позволяет выполнять код из диапазона адресов, относящегося к стеку, атака не удастся.
2. Относительно корректное завершение функции: после того, как перезаписан адрес возврата и предшествующие ему значения в стеке, функция должна доработать до команды `ret`, чтобы выполнился внедряемый код.
3. Постоянные адреса: в рамках последовательных запусков программы адреса объектов в стеке не должны меняться, т.к. иначе перезаписанный адрес возврата перестанет быть корректным, и вместо перехвата потока управления можно будет получить лишь аварийное завершение программы.

Противодействие выполнению кода на стеке

Неисполнимый стек (W^X, DEP) — технология, позволяющая помечать сегменты или страницы памяти как неисполняемые. При попытке передать управление на код, размещённый в такой памяти, происходит аварийное завершение процесса.

Аппаратно поддерживается на уровне страничной трансляции во всех процессорах x86-64, а также в более новых x86 и ARM.

В более старых моделях x86 возможна более медленная и сопряжённая с дополнительными ограничениями на исполнимые файлы реализация с использованием сегментной трансляции.

«Канарейка» на стеке

Общая идея: проверять факт перезаписи стека непосредственно перед адресом возврата перед выходом из функции.

«Канарейка» — как правило случайное значение, которое размещается на стеке перед адресом возврата. Перед выходом из функции происходит сравнение значения в стеке с исходным значением. Если значения не совпадают, программа аварийно завершается.

Обход DEP: return-to-libc

Вместо передачи управления на код внутри буфера можно заменить адрес возврата на адрес известной библиотечной функции, например `system` из стандартной библиотеки Си.

```
#include <stdlib.h>

int
system(const char *command);
```

ROP

Гаджет — последовательность команд в исполняемых секциях программы, заканчивающаяся командой RET.

Возвратно-ориентированное программирование (ROP) — техника построения эксплоита, при которой полезная нагрузка формируется в виде цепочки гаджетов с известными постоянным адресами.

8. Статический анализ исходного кода с целью поиска ошибок. Типы обнаруживаемых ошибок. Путь распространения ошибки: source, propagation, sink. Потоковая и контекстная чувствительность. Качество результата анализа: false/true positive/negative. Интерпретация результатов анализа.

Анализ программы — выявление фактов о программе

Статический анализ — анализ программы без её запуска;

Поиск ошибок (FindBugs, Coverity, Klocwork, Svace).

- 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».

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:

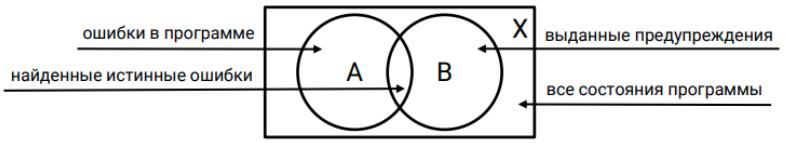
- деление на ноль.

Использование неинициализированных значений.

Ошибки работы с памятью:

- разыменование нулевого указателя;
- выход за границы буфера.

Оценка качества анализа



- B \ A — ложные срабатывания (false positive);
- A \ B — не найденные ошибки (false negative);
- A ∩ B — найденные истинные ошибки (true positive);
- X \ A \ 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: $\frac{|A \cap B|}{|B|}$.

Чувствительность к пути — способность анализа различать разные пути в программе.
Чувствительность к потоку — способность анализа различать порядок следования операторов.
Чувствительность к контексту — способность анализа различать разные вызовы одной функции.

- 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.

9. Применение отладки для оценки возможности эксплуатации уязвимостей. Технологии отладки. Отладка пользовательского кода. Полносистемная отладка в виртуальной машине. Статическое и динамическое инструментирование. Фаззинг. Разновидности фаззинга: черный ящик, белый ящик, серый ящик.

Для эксплоитов

- Сопоставление объектов низкого уровня (адресов, неструктурированных данных) с объектами высокого уровня (функциями, строками кода, типизированными данными).
- Компилятор может выдавать отладочную информацию, описывающую соответствие между сгенерированным бинарным кодом и исходным кодом, из которого он был получен

Отладка пользовательского кода

- “Print-debugging” — добавление в программу вывода значений переменных в окрестности ошибки.
- Бисекция (“wolf fence” algorithm) — способ локализации ошибки:
    - временное отключение частей кода с поиском того момента, когда ошибка перестаёт воспроизводиться;
    - наоборот, последовательное включение частей кода с поиском того момента, когда ошибка начнёт воспроизводиться.
- Применение отладчика:
    - локальная отладка (на той же самой машине);
    - удалённая отладка (через сетевое соединение).
    - Отладчик подключается к выполняющемуся процессу либо к дампу памяти (core dump), полученному при аварийном завершении программы.

Базовый функционал отладчика

- Точка останова (breakpoint) — адрес команды, при достижении которого выполнение программы должно быть приостановлено.
- Точка отслеживания (watchpoint) — адрес ячейки памяти, при обращении к которой на чтение или запись выполнение программы должно быть приостановлено.
- Пошаговое выполнение программы — выполнение одной машинной команды или одной строки исходного кода (с использованием отладочной информации).

В момент, когда выполнение программы приостановлено, отладчик предоставляет возможность просмотра содержимого регистров и памяти.

Поддержка

- Точки останова реализуются через внедрение команды INT 3(аппаратная поддержка) в тело программы по требуемому адресу.
- Команда INT 3 кодируется единственным байтом 0xCC.
- Отладчик сохраняет содержимое первого байта по адресу точки останова и заменяет его в загруженном в память образе программы на команду INT 3.
- При достижении потоком управления команды INT 3 процессор генерирует отладочное прерывание, которое при поддержке со стороны операционной системы перехватывается отладчиком.

- Операционная система приостанавливает выполнение потока, в котором возникло прерывание, до дальнейших указаний от отладчика.

Пошаговое выполнение: Установка бита TF в регистре RFLAGS приводит к тому, что после выполнения каждой очередной команды генерируется отладочное исключение..

Отладка в виртуальной машине
Отладка может выполняться в полносистемном эмуляторе (виртуальной машине), например, в QEMU.
В этом случае отладчик (GDB) подключается не к выполняющемуся на той же машине процессу, а через сетевое соединение к ответной части (GDB debugging stub) в эмуляторе.

Основное преимущество отладки в виртуальной машине — исключение влияния наличия отладчика на поведение отлаживаемой системы.(кроме увеличения времени выполнения)
Основные недостатки — относительная сложность подготовки окружения и семантический разрыв.
Семантический разрыв
- Полносистемная отладка предоставляет полную информацию о состоянии машины.
- Имеется возможность отладки драйверов и ядра ОС: можно обращаться к структурам данных, которые недоступны обычным отладчикам.
- Вместе с тем, отладка осуществляется на низком уровне (значения регистров, неинтерпретируемое содержимое памяти и т.д.).
- Существует семантический разрыв — состояние системы как его «понимает» отладчик и как его понимает оператор, находятся на разных уровнях.

Инструментирование — добавление в существующую программу блоков кода, осуществляющих отслеживание каких-либо её характеристик во время выполнения.

Статическое инструментирование выполняется однократно перед запуском программы. Статическое инструментирование может выполняться:
- на уровне исходного кода — вручную или автоматически;
- на уровне бинарного кода — вручную или автоматически.

“Print-debugging” — пример выполненного вручную статического инструментирования на уровне исходного кода с целью отладки программы.

Динамическое инструментирование бинарного кода (DBI) — разновидность инструментирования бинарного кода, выполняемого во время работы программы.

Динамическое инструментирование бинарного кода родственно JITкомпиляции, т.к. предполагает построение выполняющегося кода «на лету».

Подходы к выполнению инструментирования блока трансляции:
- копирование и аннотирование (C&A — copy-and-annotate);
- «дизассемблирование» и ресинтезирование (D&R: disassembleand-resynthesize).

Фаззинг – подход к тестированию программных систем, при котором на вход системе передаётся большое количество образцов входных данных, сгенерированных другой программой (фаззером). Процесс их обработки контролируется с целью обнаружения ошибок.
Отличие от других видов тестирования:

- «Классическое тестирование» - запуск программы на множестве нормальных входов с целью предотвращения обнаружения ошибок обычными пользователями.
- Фаззинг - запуск программы на множестве аномальных входов, детектирование проблем с целью предотвращение обнаружения эксплуатируемых уязвимостей атакующими.

Схема использования фаззинга

1. Определить источник входных данных программы
2. Случайное мутирование корректного входа или гененерация псевдослучайных данных
3. Использование оракула для мониторинга аварийных завершений
4. Запись входных данных и состояния на которых произошло аварийное завершение

Причины возникновения и использования

- Высокая скорость работы, легко параллелится
- Не требует наличия исходного кода программы
- Не требует написания тестов
    - Ручное тестирование и разработка тестов дороги
- У разработчиков тестов обычно есть «слепые пятна»
- Относительно просто реализуется
- Переносимость/Повторное использование
- Высокая доля истинных ошибок по сравнению со статическими анализаторами

Классификация методов фаззинга

- Фаззинг по методу «чёрного ящика» - генерация входных данных без обратной связи с анализируемой программой – только детектирование ошибок.
- Фаззинг по методу «серого ящика» - генерация входных данных на основании наблюдения за выполнением программы.
- Фаззинг по методу «белого ящика» - генерация входных данных на основе предварительного запуска программы на корректных данных, построения уравнений пути и их решения в ходе *символьного выполнения программы*.

10. Символьное выполнение: основные понятия. Схема работы системы символьного выполнения. Предикат пути, предикат безопасности. Проблема экспоненциального взрыва, стратегии выбора следующего состояния.

Символьное выполнение

- «Выполнение» программы не на конкретных значениях входных данных, а на символьных значениях
- «Выполнение» множества путей программы одновременно: в точке ветвления, зависящей от символьных значений происходит разделение выполнения на две ветви с добавлением ограничений из условия ветвления
- Технология получила быстрое развитие в последнее время благодаря росту вычислительных возможностей и появлению удобных инструментов – решатель STP, общий формат уравнений SMT-LIB2

Представление программы

- Программа представляется в виде бинарного дерева потенциально бесконечной глубины (циклы) – дерево символьного выполнения
- Вершины дерева соответствуют выполнению условных переходов
- Рёбра – выполнению последовательных инструкций
- Каждый путь в дереве описывает эквивалентный класс входных данных
- Формула, описывающая путь – предикат пути

Предикат пути - набор логических формул описывающие прохождение по данному пути выполнения

Предикат безопасности – набор логических формул описывающие нарушение безопасности кода (переполнение буфера)

Схема работы

- Обход путей программы, генерация предикатов путей
- При обнаружении опасной ситуации (например, деление на 0) – добавление предиката безопасности
- Выбор очередной формулы и отправка её решателю
- Если формула содержала предикат безопасности и была решена – получение набора входных данных, активирующих ошибку

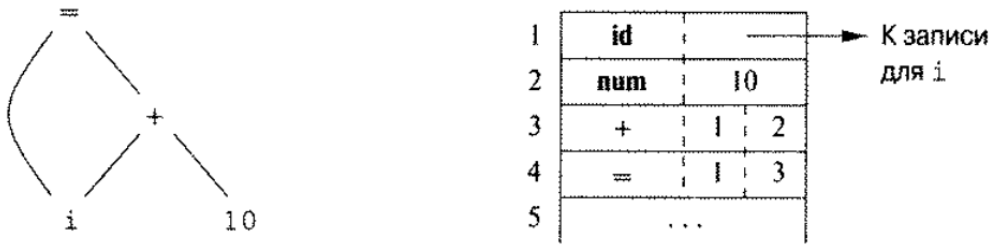
Основные проблемы подхода

- Экспоненциальный взрыв – экспоненциальный рост количества путей
- Моделирование окружения – обработка системных/библиотечных вызовов
- Ограничения решателя – сложность решения уравнений

Стратегии выбора пути

- На основе только структуры кода:
    - Обход в глубину – может «зациклится» в цикле
    - Обход в ширину – очень медленно доходит до содержательного кода при наличии большого количества ветвей
- На основе покрытия кода – выбирать пути с непосещенными инструкциями, или те которые посещались меньше. Позволяет обнаруживать ошибки на редко выполняемых путях, однако может не достигать некоторых инструкций никогда.
- Случайный выбор. Возможности:
    - Всегда выбирать путь случайно
    - Выбирать случайно если долгое время ничего не находится (гибрид)
    - Выбирать случайно в случае равного приоритета путей (гибрид)

11. Метод нумерации значений в пределах базового блока и в пределах процедуры. Реализация метода путем построения ориентированных ациклических графов и использования хеш-таблиц.



а) Ориентированный ациклический граф

б) Массив

Рис. 6.6. Узлы ориентированного ациклического графа для $i = i + 10$, расположенные в массиве

Зачастую узлы синтаксического дерева или ориентированного ациклического графа хранятся в массиве записей, как предложено на рис. 6.6. Каждая строка массива представляет одну запись, а следовательно, один узел. Первое поле каждой записи представляет собой код операции, указывая метку узла. На рис. 6.6, *б* у листьев имеется по одному дополнительному полю, в котором хранится лексическое значение (указатель на запись в таблице символов или константа), а внутренние узлы имеют по два дополнительных поля, указывающих левый и правый дочерние узлы.

Мы обращаемся к узлам с использованием целочисленного индекса соответствующей записи в массиве. Это целочисленное значение исторически носит название *номер значения* (value number) узла или выражения, представленного этим узлом. Например, на рис. 6.6 узел с меткой + имеет номер значения 3, а номера значений его левого и правого дочерних узлов равны соответственно 1 и 2. На практике вместо целочисленных индексов можно использовать указатели на записи или ссылки на объекты, но мы в любом случае будем говорить о ссылке на узел как о "номере значения". Будучи сохраненными с применением подходящей структуры данных, номера значений помогают эффективно строить ориентированные

Предположим, что узлы хранятся в массиве, как показано на рис. 6.6, и что обратиться к каждому узлу можно по его номеру. Назовем *сигнатурой* внутреннего узла тройку $\langle op, l, r \rangle$, где op — метка, l — номер значения левого, а r — правого дочернего узла. Унарный оператор можно рассматривать как имеющий значение $r = 0$.

Алгоритм 6.3. Метод номера значения построения узла ориентированного ациклического графа

ВХОД: метка op, узлы l и r.

ВЫХОД: номер значения узла с сигнатурой $\langle op, l, r \rangle$ в массиве.

МЕТОД: выполнить поиск в массиве узла M с меткой op, левым дочерним узлом l и правым дочерним узлом r. Если такой узел найден, вернуть номер значения M. Если нет, создать в массиве новый узел N с меткой op, левым дочерним узлом l и правым дочерним узлом r и вернуть его номер значения. □

Хотя алгоритм 6.3 и выдает на выходе требуемую информацию, сканирование всего массива всякий раз, когда нам требуется получить информацию об одном узле, — метод слишком расточительный, в особенности если в массиве хранятся выражения из всей программы. Более эффективный подход заключается в использовании хеш-таблицы, в которой узлы помещаются в блоки (buckets), в каждом из которых обычно хранится всего лишь несколько узлов. Хеш-таблица представляет собой одну из нескольких структур данных, эффективно поддерживающих словари.[1]

Для построения хеш-таблицы для узлов ориентированного ациклического графа требуется *хеш-функция* h, которая вычисляет индекс блока по сигнатуре $\langle op, l, r \rangle$ способом, который равномерно распределяет сигнатуры по блокам,

12. Исключение частично-избыточных выражений методом анализа потока данных.

ChatGPT (не плохо расписал имхо):

Метод анализа потока данных (Data Flow Analysis) широко используется в компиляторах для оптимизации кода и поиска ошибок. Он позволяет проанализировать поток данных в программе и выявить возможности для оптимизации, такие как исключение частично-избыточных выражений.

Шаги исключения частично-избыточных выражений методом анализа потока данных можно описать следующим образом:

1. Анализируем граф потока управления программы и строим его модель.
2. Для каждого блока базовых операций (Basic Block) вычисляем множество доступных выражений (Available Expressions).
3. Для каждой переменной в программе определяем ее определение (Definition), использование (Use) и живучесть (Liveness).
4. Используя множества доступных выражений и информацию о живучести переменных, определяем частично-избыточные выражения (Partially Redundant Expressions).
5. Выбираем наилучшее место для вычисления выражения и создания временной переменной для хранения результата.
6. Заменяем частично-избыточное выражение ссылкой на временную переменную.
7. Повторяем шаги 2-6 до тех пор, пока есть частично-избыточные выражения в программе.

Как только все частично-избыточные выражения будут исключены, программа будет оптимизирована и ее производительность должна улучшиться.

Избыточность в программе существует в различных формах. Как говорилось в разделе 9.1.4, она может быть в виде общих подвыражений, когда несколько вычислений выражения дают одно и то же значение. Она может быть и в виде выражения, инвариантного относительно цикла, которое вычисляет одно и то же значение на каждой итерации цикла. Избыточность может быть частичной, если она обнаруживается вдоль некоторых, но *не всех*, путей. Общие подвыражения и выражения, инвариантные относительно цикла, можно рассматривать как частные случаи частичной избыточности; таким образом, для устранения различных видов избыточности можно разработать единый алгоритм устранения частичной избыточности.

На рис. 9.30 проиллюстрированы три вида избыточности: общие подвыражения, выражения, инвариантные по отношению к циклу, и частично избыточные выражения. На рисунке показан код как до, так и после выполнения каждого вида оптимизации.

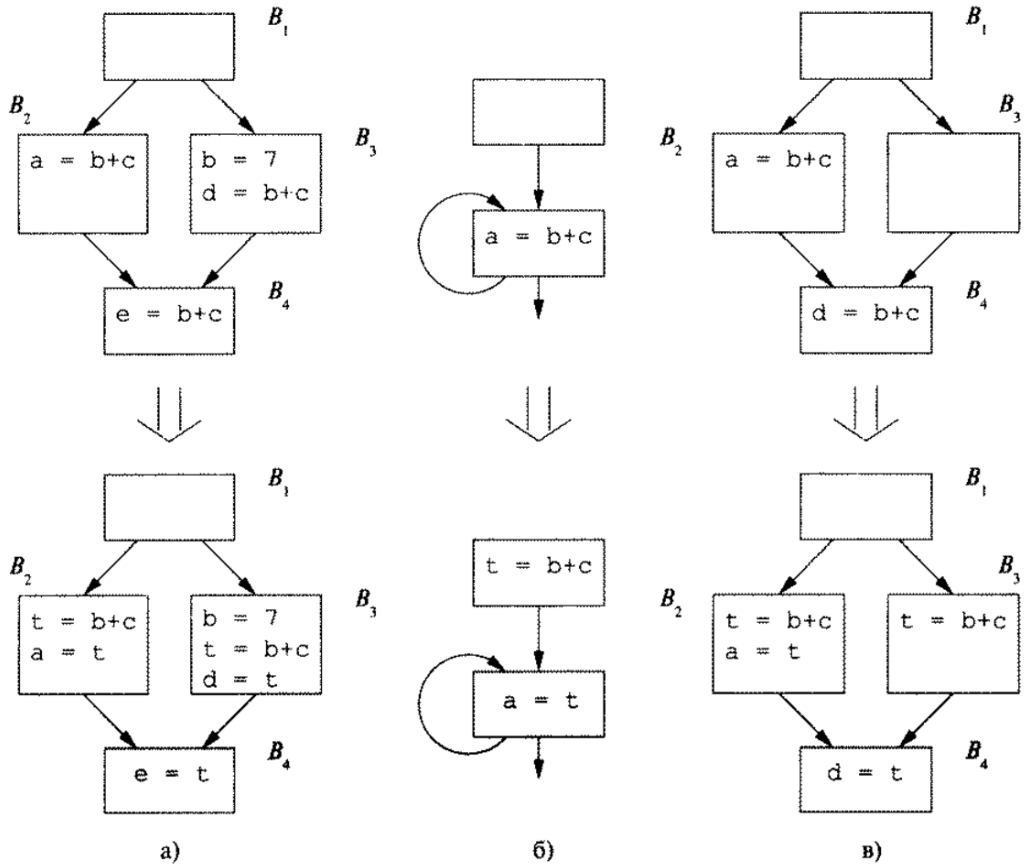


Рис. 9.30. Примеры а) глобального общего подвыражения; б) перемещения кода, инвариантного по отношению к циклу; в) устранения частичной избыточности

При устранении частичной избыточности при помощи единого алгоритма находится множество типов избыточных операций. Этот алгоритм иллюстрирует, как много задач потоков данных могут использоваться при поиске оптимального размещения выражений.

1. Ограничения на размещения накладываются анализом ожидаемости выражений, который представляет собой *обратный* анализ потока данных с пересечением в качестве оператора сбора и определяет, используется ли выражение *впоследствии* каждой точкой программы по *всем* путям.

2. Наиболее ранее размещение выражения определяется точкой программы, в которой выражение ожидаемо, но не доступно. Поиск доступных выражений выполняется при помощи *прямого* анализа потока данных с оператором сбора, представляющим собой пересечение множеств. Этот анализ выясняет, является ли выражение ожидаемым *перед* каждой точкой программы вдоль *всех* путей.

3. Наиболее позднее размещение выражения определяется точкой программы, в которой выражение более не может быть откладываемым. Выражения в некоторой точке программы являются откладываемыми, если вдоль *всех* путей, *достигающих* данной точки программы, не встречается использование этого выражения. Поиск откладываемых выражений выполняется при помощи *прямого* анализа потока данных с оператором сбора, представляющим собой пересечение множеств.

4. Присваивание временным переменным устраняется, если только они не используются *впоследствии* вдоль *некоторого* пути. Поиск используемых выражений выполняется при помощи *обратного* анализа потока данных с оператором сбора, представляющим собой объединение множеств.

13. Граф зависимостей программы: определение, построение, применение.

11.8.2 Графы зависимостей программ

Чтобы обнаружить все возможные при помощи добавления постоянного количества синхронизаций распараллеливания, можно применить "жадное" расщепление исходной программы. Разобьем циклы на максимально возможное количество отдельных циклов, а затем независимо распараллелим каждый из циклов.

Чтобы найти все возможные расщепления цикла, воспользуемся абстракцией *графа зависимостей программы* (program-dependence graph — PDG). Граф зависимостей программы представляет собой граф, узлами которого являются инструкции присваивания в программе, а ребра указывают зависимости данных между инструкциями и их направления. Ребро от инструкции s_1 к инструкции s_2 имеется в том случае, когда имеется зависимость данных между некоторым динамическим экземпляром s_1 и более *поздним* динамическим экземпляром s_2.

Для построения PDG программы мы сначала находим все зависимости данных между каждой парой (не обязательно различных) статических обращений в каждой паре (не обязательно различных) инструкций. Предположим, мы определили, что существует зависимость между обращением $\mathcal{F}_1$ в инструкции s_1 и обращением $\mathcal{F}_2$ в инструкции s_2. Вспомним, что экземпляр инструкции определяется индексным вектором $\mathbf{i} = [i_1, i_2, \ldots, i_m]$, где i_k — индекс цикла k-й вложенности, в котором находится рассматриваемая инструкция.

1. Если существует зависимая пара экземпляров, $\mathbf{i}_1$ инструкции s_1 и $\mathbf{i}_2$ инструкции s_2, и $\mathbf{i}_1$ в исходной программе выполняется до $\mathbf{i}_2$, что записывается как $\mathbf{i}_1 \prec_{s_1 s_2} \mathbf{i}_2$, то в PDG имеется ребро от s_1 к s_2.

2. Аналогично, если существует зависимая пара экземпляров, $\mathbf{i}_1$ инструкции s_1 и $\mathbf{i}_2$ инструкции s_2, и $\mathbf{i}_2 \prec_{s_1 s_2} \mathbf{i}_1$, то в PDG имеется ребро от s_2 к s_1.

Заметим, что возможна ситуация, когда зависимости данных между инструкциями s_1 и s_2 генерируют как ребро от s_1 к s_2, так и ребро от s_2 к s_1.

В частном случае совпадения инструкций s_1 и s_2 $\mathbf{i}_1 \prec_{s_1 s_2} \mathbf{i}_2$ тогда и только тогда, когда $\mathbf{i}_1 \prec \mathbf{i}_2$ ($\mathbf{i}_1$ лексикографически меньше $\mathbf{i}_2$). В общем случае s_1 и s_2 могут быть различными инструкциями, возможно, принадлежащими разным вложенностям циклов.

и s_2 находятся в одном общем цикле. Из-за этой взаимозависимости мы не можем выполнить все экземпляры одной инструкции до другой, а значит, и выполнить расщепление. С другой стороны, если зависимость $s_1 \rightarrow s_2$ — однонаправленная, то можно разделить цикл и сначала выполнить все инструкции s_1, а затем все инструкции s_2.

Граф зависимостей программы облегчает определение того, можно ли разделить инструкции в цикле. Инструкции, соединенные в цикл в PDG, разделены быть не могут. Если $s_1 \rightarrow s_2$ — зависимость между двумя инструкциями в цикле, то некоторый экземпляр s_1 должен быть выполнен до некоторого экземпляра s_2, и наоборот. Заметим, что такая взаимозависимость наблюдается, только если s_1

14. Проблемы статического анализа объектно-ориентированных языков (C++, Java). Поток управления в присутствии исключений. Вызовы по указателю и их анализ. Понятие одевиртуализации.

Проблемы:

Использование динамического полиморфизма означает, что определенный метод может вызываться для разных объектов, и это может быть известно только во время выполнения. Это затрудняет точный анализ кода на этапе компиляции.

Также в C++ есть проблемы, связанные с использованием указателей и ссылок, которые могут привести к неопределенному поведению во время выполнения. Такие проблемы могут быть трудными для обнаружения статическим анализом.

В Java, статический анализ также имеет свои ограничения относительно анализа потока управления. В частности, использование динамической загрузки классов может привести к тому, что методы и поля будут известны только во время выполнения

В присутствии исключений поток управления может значительно изменится. Требуется рассматривать несколько вариантов развития событий одновременно и анализировать несколько путей.

Проблемы с вызовом по указателю в том, что достоверно может быть не известен адрес, по которому мы будем переходить. Мы можем только попытаться как-то сузить в общем случае область переходов с помощью анализа указателей. Естественно восстановить гарантировано поток управления в этом случае мы не можем.

Одевиртуализация (деобфускация) в статическом анализе - это процесс обратного преобразования двоичного кода в исходный код, который был замаскирован, возможно, с помощью техник обфускации.
Обфускация используется для защиты программного кода от анализа и понимания его работы. Однако, в некоторых случаях, необходимо провести анализ защищенного кода, например, для обнаружения уязвимостей или создания более эффективных алгоритмов.

В статическом анализе одевиртуализация может быть выполнена с помощью специальных инструментов, которые выявляют и извлекают скрытый код из программы. Эти инструменты работают на уровне машинного кода, выполняя различные декодирования и дешифрования, чтобы получить исходный код программы. Однако, этот процесс может быть сложным и затратным, особенно если оригинальный код был сильно обфусцирован.

15. Инструментирование при динамическом анализе: инструментирование исходного кода программ при компиляции, динамическая двоичная трансляция.

Dynamic analysis can help with some situations when static cannot

+ Прочти нет false positives ошибок
- Дорогой анализ
- Находит ошибки только на конкретном пути исполнения.

Статическое инструментирование
Инструментирование — добавление в существующую программу блоков кода, осуществляющих отслеживание каких-либо её характеристик во время выполнения.

Статическое инструментирование выполняется однократно перед запуском программы.
Статическое инструментирование может выполняться:

- на уровне исходного кода — вручную или автоматически;
- на уровне бинарного кода — вручную или автоматически.

“Print-debugging” — пример выполненного вручную статического инструментирования на уровне исходного кода с целью отладки программы.

Динамическое инструментирование бинарного кода (DBI) — разновидность инструментирования бинарного кода, выполняемого во время работы программы. Динамическое инструментирование бинарного кода родственно JITкомпиляции, т.к. предполагает построение выполняющегося кода «на лету».
Подходы к выполнению инструментирования блока трансляции:

- копирование и аннотирование (C&A — copy-and-annotate); PIN
- «дизассемблирование» и ресинтезирование (D&R: isassembleand-resynthesize). Valgrind

16. Информационная безопасность. Шифрование данных. Криптографическая стойкость. Симметричная криптография. Блочный шифр (DES) и его режимы. Ассиметричные схемы (RSA и Диффи-Хеллмана). Код аутентификации (MAC). Цифровая подпись (DSA).

Безопасность информации (данных) — состояние защищенности информации (данных), при котором **обеспечены ее** (их) конфиденциальность, доступность и целостность.

1. **Конфиденциальность:** обеспечение доступа к информации только авторизованным пользователям.
2. **Целостность**: обеспечение достоверности и полноты информации и методов ее обработки.
3. **Доступность**: обеспечение доступа к информации и связанным с ней активам авторизованных пользователей по мере необходимости.

Национальный стандарт РФ «Защита информации. Основные термины и определения» (ГОСТ Р 50922-2006).;

Шифрование данных

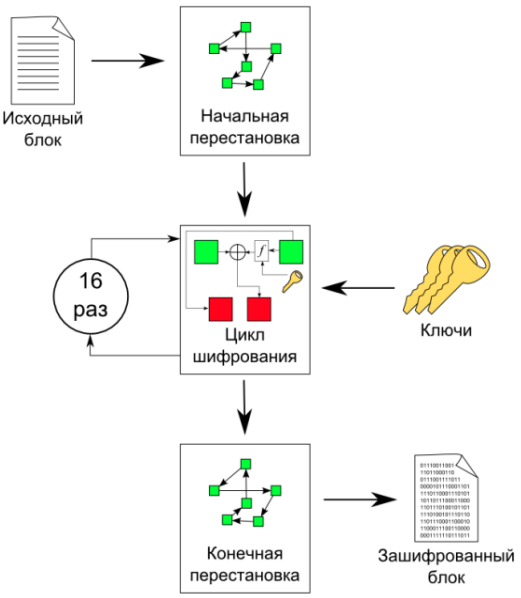
- Криптография – наука о методах обеспечения конфиденциальности, целостности данных, аутентификации, невозможности отказа от авторства
- Шифрование – обратимое преобразование открытого (исходного) текста на основе секретного алгоритма или ключа в шифрованный текст
- Принцип Керкгоффса (Kerckhoffs's) 1883, максима Шеннона (Shannon) 1949 – алгоритмы шифрования общедоступны; секретны только ключи

Понятие о криптографической стойкости

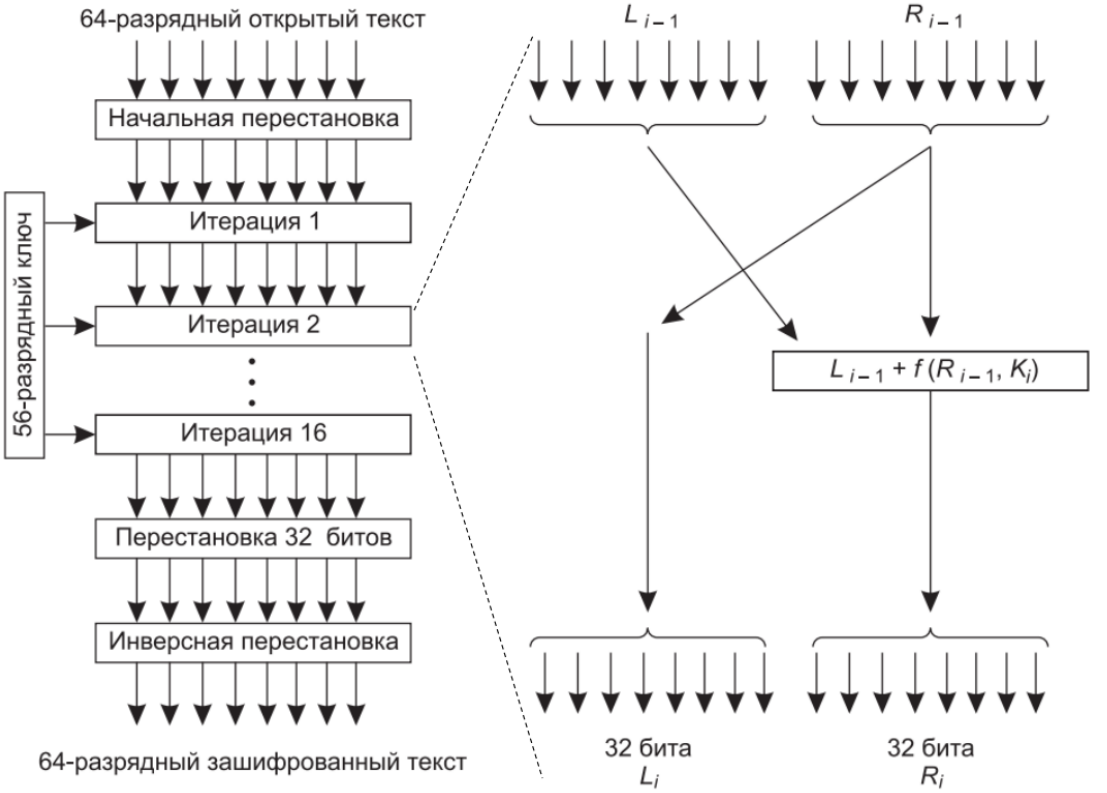
- Криптографическая стойкость – способность криптографического алгоритма противостоять криптоанализу
- Абсолютно стойкие системы – криптосистема не может быть раскрыта (дешифрована) ни теоретически, ни практически даже при наличии у атакующего бесконечно больших вычислительных ресурсов
- Достаточно стойкие системы – потенциальная возможность дешифрования существует (обратное не доказано) – оценка стойкости выполняется в расчёте на определённый момент времени последовательно с двух позиций:
    1. вычислительная сложность полного перебора
    2. известные на данный момент слабости (уязвимости) криптосистемы и их влияние на вычислительную сложность – доказуемая стойкость
    - доказательство стойкости криптосистемы сводится к решению определённой трудно решаемой математической проблемы, положенной в основу алгоритма

DES – Data Encryption Standard

- Блочный, симметричный шифр
- Разработан: в январе 1977
- Размер блока: 64 бит
- Размер ключа: 56 бит
- Шифрование одного блока:
  - Начальная перестановка
  - 16 циклов шифрования
  - Конечная (обратная) перестановка



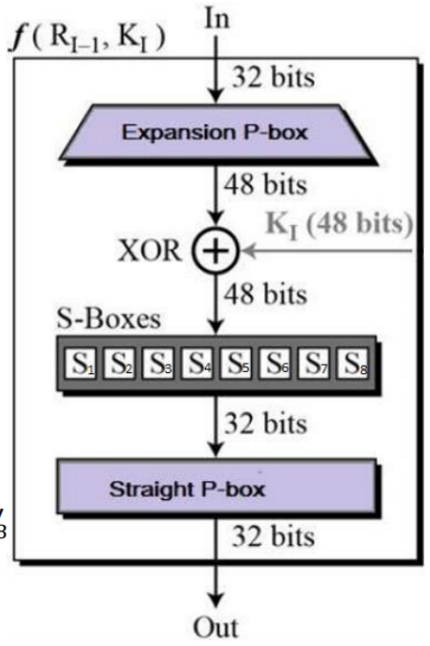
DES: шифрование блока



DES: функция шифрования f

https://academic.csuohio.edu/yuc/security/Chapter_06_Data_Encription_Standard.pdf

- Функция расширения $E(R_{i-1})$
  - расширяет вектор размером 32 бит до 48 бит путем повторения некоторых бит
- Побитовое сложение по модулю 2 расширенного вектора $E(R_{i-1})$ с ключом этого цикла K_i
  - Исходный ключ (56 бит) → 16 ключей, каждый по 48 бит, по одному ключу на итерацию
  - Результат сложения: $B_1B_2B_3B_4B_5B_6B_7B_8$
- Преобразование S (48 бит → 32 бит)
  - $B_1B_2B_3B_4B_5B_6B_7B_8 \rightarrow B'_1B'_2B'_3B'_4B'_5B'_6B'_7B'_8$
  - B_i (6 бит) → B'_i (4 бит), $i = 1, \ldots, 8$
- Перестановка P



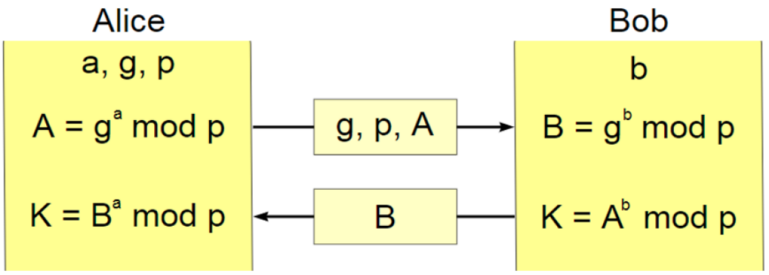
Тройное шифрование DES (Triple DES)

- Два ключа ($K1$,$K2$)
- Три этапа (E,D, E)
- Ключа длиной 112 бит (= 2 * 56) вполне достаточно
- Обратная совместимость с существующими DES системами с одним ключом ($K2$ = $K1$)

Протокол Диффи-Хеллмана

- Позволяет двум и более сторонам получить общий секретный ключ, используя не защищенный от прослушивания канал связи
- Основан на трудоемкости задачи вычисления *дискретного логарифма*: – обращение функции g^x в некоторой конечной мультипликативной группе G.

Протокол Диффи-Хеллмана



- *a, b* – натуральные числа – **закрытые ключи**
- *p* – случайное простое число
- *g* – первообразный корень по модулю *p*
  - образующий элемент *мультипликативной группы кольца вычетов* по модулю *p*
    - следовательно, числа *g* и *p* взаимно просты

Схема RSA

- Основана на трудоемкости задачи факторизации (разложения на множители) произведения двух простых чисел
- Для обеспечения достаточного уровня защищенности требуется ключ длиной, по крайней мере, 1024 бита
- Схема слишком медленная, чтобы шифровать большие объемы данных – широко применяется для распространения ключей

Описание схемы RSA

- Выбирается два **различных** больших простых числа p и q (обычно длиной по 1024 бита)
- Вычисляется: $n = p \cdot q, (p-1) \cdot (q-1) = \varphi(n)$
- Выбирается число e – открытая экспонента, **взаимно простое** с числом $\varphi(n)$
- Вычисляется $d = e^{-1} mod(\varphi(n))$
- Для сообщения m: $0 \leq m \leq n-1$:
  - Шифрование:
    - $c = m^e (mod\ n)$
    - *(e, n)* – открытый ключ
  - Дешифрование – используется теорема Эйлера:
    - $m = c^d (mod\ n)$
    - *d* – закрытый ключ

Код аутентификации сообщения

- Используя все биты сообщения, отправитель генерирует избыточный код (дополнение) и передает его вместе с сообщением
- Прежде чем принять сообщение как подлинное, получатель выполняет проверку: – содержимое сообщения и его дополнение являются согласованными
- Дополнение называется кодом аутентификации сообщения: MAC – Message Authentication Code

Какие задачи решает MAC

1. Целостность сообщения – в процессе передачи не было непреднамеренной модификации сообщения
2. Подлинность сообщения – в процессе передачи не было преднамеренной модификации сообщения

Цифровая подпись

Аналог «рукописной подписи» для цифровых документов

Требования:

1. получатель может проверить объявленную личность отправителя

банк может установить подлинность клиента, а клиент – подлинность банка

2. отправитель не может позднее отрицать содержимое сообщения
3. получатель не может позднее изменить подписанное сообщение

Digital Signature Algorithm (DSA)

Общие параметры схемы:

1. $H(x)$ – криптографическая хэш-функция
2. L, N – длина открытого и закрытого ключей
   - *L = 1024, N = 160*
   - *L = 3072, N = 256*
3. $\boldsymbol{q}$ – простое число длиной N бит
4. $\boldsymbol{p}$ – простое число длиной L бит такое, что $p - 1$ делится на q
5. $h \in (1, p-1)$ – произвольное число
6. $\boldsymbol{g} = h^{\frac{p-1}{q}} (mod\ p), g \neq 1$
   - если $g = 1$, то выбрать другое h

Параметры схемы для **каждого пользователя**:

1. закрытый ключ $\boldsymbol{x}$ – случайно выбранное число: $0 < x < q$
2. открытый ключ $\boldsymbol{y} = g^x (mod\ p)$

Схема цифровой подписи DSA: генерация подписи сообщения

1. Для каждого сообщения m генерируется случайное число k: $1 < k < q$
2. Вычислить $r = \left(g^k mod\ p\right) mod\ q$
   - Если $r = 0$, вернуться на шаг 1 и сгенерировать новое случайное число k
3. Вычислить $s = k^{-1}(H(m) + x{\cdot}r)\ mod\ q$
   - Если $s = 0$, вернуться в п.1 и сгенерировать новое случайное число k
4. Цифровая подпись: (r, s)

Схема цифровой подписи DSA: проверка подписи

- Подпись заведомо неверна, если не выполнено хотя бы одно из условий:
  1. $0 < r < q$
  2. $0 < s < q$
- Далее:
  - $w = s^{-1} mod\ q$
  - $u_1 = H(m){\cdot}w\ mod\ q$
  - $u_2 = r{\cdot}w\ mod\ q$
  - $v = (g^{u_1} y^{u_2} mod\ p) mod\ q$
  - цифровая подпись (r, s) верна $\Leftrightarrow v = r$

17. Понятие анонимности пользователя в сети. Идентификаторы пользователя в сети на разных уровнях (устройства, ОС, ПО). Подходы к деанонимизации и способы защиты. Концепция анонимных сетей (Mix и Tor). Луковая маршрутизация. Виды атак на анонимные сети.

Анонимность пользователя в сети

- Защита от наблюдения со стороны Интернет-провайдера (а также тех, кто его контролирует):
  – цензура
  – контекстная реклама
  – осуществление общественной, гражданской или политической деятельности
- Защита от наблюдения со стороны посещаемого ресурса

Бывает:

- Социальная анонимность – распространение (осознанное/неосознанное) в сети персональной информации самим пользователем
- Техническая анонимность:
  – контроль за хранением и безопасностью персональных данных посредством специальных технических средств (как программных, так и аппаратных)
  – минимизация возможности утечки персональных данных

Анонимность субъекта (anonymity):

– злоумышленник не может с достаточной степенью точности идентифицировать субъект в рамках некоторого множества субъектов со схожим набором атрибутов

Виды анонимности

- Анонимность отправителя:

– ни получатель сообщения, ни атакующий, наблюдающий за некоторым участком сети (между отправителем и получателем), не могут установить отправителя сообщения

- Анонимность получателя:

– атакующий, наблюдающий за некоторым участком сети, не может установить получателя сообщения

- Анонимность взаимодействия:

– атакующий, наблюдающий за некоторым участком сети, не может достоверно установить, от какого отправителя и какому получателю доставляется конкретное сообщение

- K-анонимность:

– пользователь сети не может на практике отличаться, по крайней мере, от $(K - 1)$ других пользователей сети, когда K достаточно большое
– вводится для количественной оценки уровня анонимности, предоставляемого системой анонимизации отдельному пользователю

Идентификаторы пользователя в сети

Устройство пользователя
– Адресные признаки устройства
- MAC-адрес сетевого интерфейса
- IP-адрес, назначенный сетевому интерфейсу

ПО пользователя
– Операционная система (реализация стека протоколов TCP/IP):
- Windows 10, Ubuntu 20.04, iOS 13.2.2, …

– Сетевое приложение:
- Веб-браузер – User-Agent, Cookies, цифровой отпечаток, …

Обеспечение технической анонимности

Задача:
– скрыть факт взаимодействия между двумя узлами сети для некоторого потока сетевых пакетов:
- Скрыть адресные признаки узла-отправителя и узла-получателя
- Скрыть идентификаторы уровня ОС
- Сделать цифровой отпечаток пользователя «стандартным»

Сокрытие адресных признаков:
– отказаться от прямой передачи данных от узла-отправителя к узлу-получателю
- осуществлять передачу данных через один или несколько промежуточных узлов

Анонимные сети

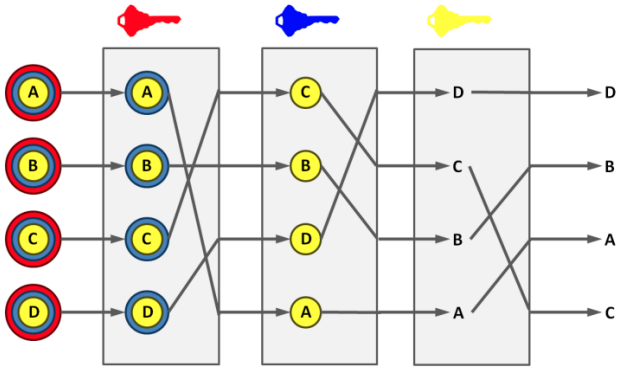
Обобщают подход на основе использования промежуточных узлов
- Используется цепочка из нескольких узлов

Ни один промежуточный узел не знает одновременно адреса источника и адреса назначения
- большинство промежуточных узлов не знает ни того, ни другого адреса

Mix networks:
- Каждый узел знает только свою часть маршрута:
  – вложенное шифрование с использованием публичных ключей всех участвующих в передаче узлов – защита от отдельных скомпрометированных узлов
- Впервые предложены в 1981 году by David Chaum – https://chaum.com/wp-content/uploads/2021/12/chaummix.pdf
- Mix-сети относятся к группе анонимных сетей с высокой задержкой – это ограничивает их применимость
  – ОК – для электронной почты
  – НЕОК – для анонимного просмотра веб-страниц/видео

Анонимные сети: Mix

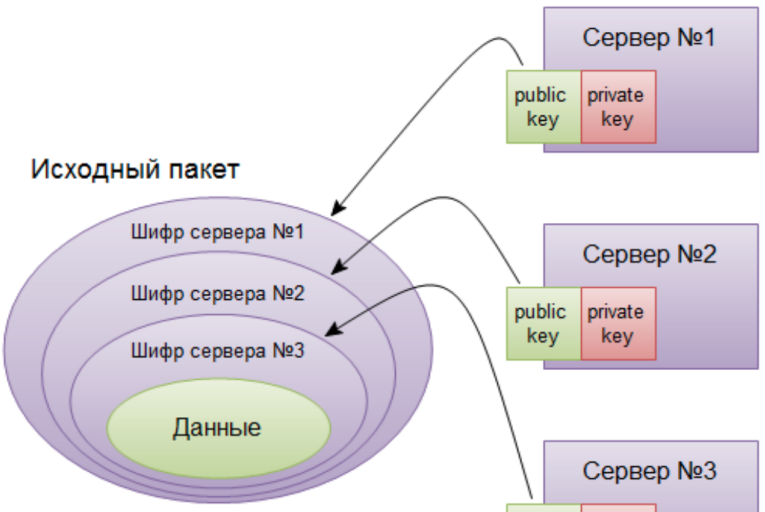
- Используется цепочка прокси-серверов, каждый из которых:
  1. получает сообщения от **большого числа источников**
  2. **изменяет порядок отправки** сообщений
  3. посылает **в измененном порядке** сообщения следующему прокси-серверу или получателю



Анонимные сети: The Onion Router

- Децентрализованная сеть, поддерживаемая добровольцами
- Исходный код в открытом доступе (октябрь 2003)
- Считается одним из самых надежных способов обеспечения анонимности в сети Интернет

«Луковая» маршрутизация



1. Каждый отдельный уровень шифрования дешифруется очередным маршрутизатором
2. Из дешифрованных данных извлекается адрес следующего маршрутизатора
3. Сообщение отправляется на следующий маршрутизатор

Атаки на анонимные сети

1. **Атака на кодировку сообщения.** Если сообщение не меняет свое содержимое при прохождении по анонимной сети, весь его маршрут может быть восстановлен путем сравнения передаваемого содержимого в разных точках анонимной сети
2. **Атака на длину сообщения**. Если сообщение сохраняет свой размер при передаче по сети, его маршрут может быть восстановлен с некоторой вероятностью путем сравнения длин передаваемых сообщений в разных точках анонимной сети
3. **Атака на воспроизведение**. Атакующий может воспроизводить данные ранее переданных сообщений, ожидая, что сеть передаст эти пакеты по тому же маршруту, и атакующий сможет связать входящие и исходящие пакеты
4. **Атака путем сговора**. Некоторые участники анонимной сети объединяются для нарушения анонимности остальных участников
5. **Атака на переполнение**. Атакующий переполняет отдельные каналы анонимной сети большим количеством сообщений, чтобы путем оценки характеристик передачи некоторых сообщений сузить круг пользователей, к которым эти сообщения могут относиться.
6. **Временные атаки**. Атакующий пытается оценить продолжительность соединения путем наблюдения фактов установки и окончания соединения в различных точках входа и выхода сети
7. **Атака на объем данных**. Атакующий пытается сопоставить общий объем передаваемых данных в точках входа и выхода сети
8. **Атака профилирования**. подразумевает долговременный анализ соединений некоторого набора пользователей – анализ обычно является комбинацией временных атак и атак на объем данных

Деанонимизирующие признаки ОС: стек TCP/IP Основаны на пробелах в спецификациях протоколов, и, следовательно, различных реализациях в разных ОС:

– IPv4: генерация значения поля ID при фрагментации
- выбор начального значения TTL
- установка флага DF для пакетов, не требующих фрагментации

– ICMP:

Destination Port Unreachable
- размер фрагмента не фиксирован

– TCP:
- выбор начального значения sequence number
- ответы на некорректные комбинации флагов
- набор опций (порядок передачи, поддержка)

Деанонимизация ОС: активное и пассивное сканирование

- Активное сканирование:
  – Злоумышленник формирует и отправляет сообщения с заданными свойствами, после чего анализирует полученные ответы
  – Обладает большей точностью
  – Может быть обнаружено системами обнаружения вторжений (IDS)
  – Известный представитель: Nmap
- Пассивное сканирование:
  – Злоумышленник анализирует прослушиваемый трафик, ничего не отправляя в сеть
  – Является менее точным
  – Не может быть обнаружено – Известный представитель: p0f