Анализ кода и информационная безопасность

Лекция 11. Часть 1

МГУ / ВМК / СП

1101

ВОССТАНОВЛЕНИЕ ПОТОКА УПРАВЛЕНИЯ ПРОГРАММЫ

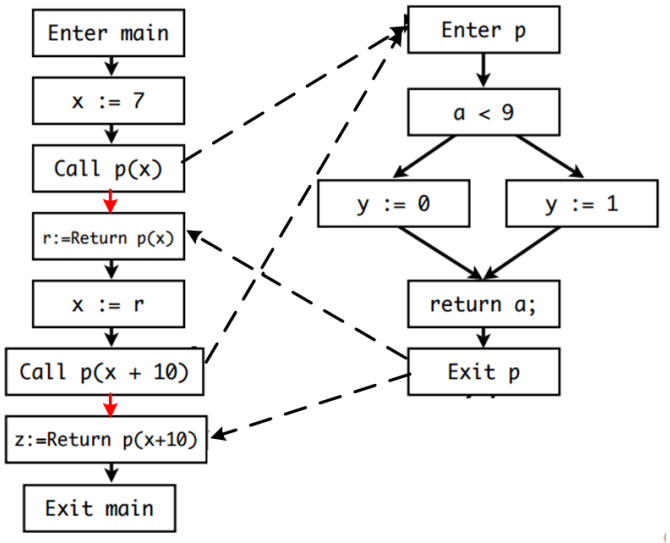
Межпроцедурный граф потока управления (ICFG)

CFG – направленный граф G=(V,E), где V – множество базовых блоков программы, а E = V x V – множество рёбер отображающих передачи управления между базовыми блоками.

ICFG - дополнения по сравнению с CFG:

- Каждому базовому блоку, заканчивающемуся на вызов ставится в соответствие базовый блок возврата из вызова. Он может быть пуст или содержать извлечение результата функции в локальную переменную.
- Два дополнительных пустых блока для каждой функции entry, exit
- 3 новых типа рёбер – рёбра вызова, возврата и «сквозь вызов»
- Все рёбра делятся на два класса:
  - межпроцедурные (вызов и возврат)
  - внутрипроцедурные – все остальные

Межпроцедурный граф потока управления (ICFG)



Определения
Специфика бинарного кода

I – инструкция в бинарном коде

- Addr(I) – адрес инструкции в коде
- Size(I) – размер инструкции в коде
- Next(I) – инструкция, расположенная сразу за I по адресу
- T.Next(I) – следующая за I инструкция в трассе T
- Tartget(I)-инструкция по адресу перехода, если I-передача управления

B – базовый блок в бинарном коде

- I-start – первая инструкция B, I-end - последняя
- Addr (B) – адрес базового блока в бинарном коде, т.е. адрес первой инструкции, входящей в базовый блок
- Size (B) – размер базового блока, т.е. суммарный размер инструкций, входящих в базовый блок, Size (I-start, I-end)

Пример графа потока управления

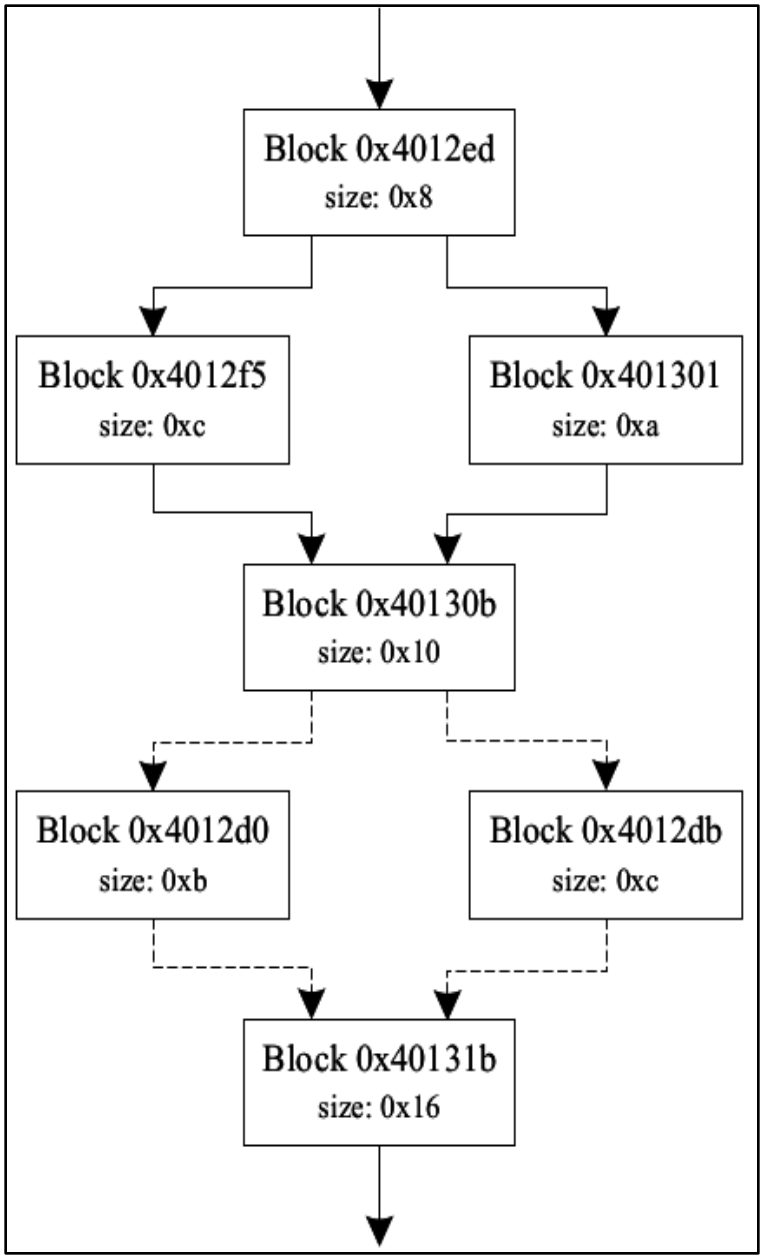
```
int (*foo)(int, int);

int add(int x, int y)
{
    return x + y;
}

int mul(int x, int y)
{
    return x * y;
}
```

```
int test(int a, int b)
{
    if (a < b)
    {
        foo = add;
    }
    else
    {
        foo = mul;
    }
    c = foo(a, b);
    return 0;
}
```

```
0x004012ed:  mov    eax, DWORD PTR [ebp+8]
0x004012f0:  cmp    eax, DWORD PTR [ebp+12]
0x004012f3:  jge    0x401301
0x004012f5:  mov    ds:0x403020, 0x4012d0
0x004012ff:  jmp    0x40130b
0x00401301:  mov    ds:0x403020, 0x4012db
0x0040130b:  sub    esp, 0x8
0x0040130e:  push   DWORD PTR [ebp+12]
0x00401311:  push   DWORD PTR [ebp+8]
0x00401314:  mov    eax, ds:0x403020
0x00401319:  call   eax
0x0040131b:  ...
```



Классификация команд. Статический подход

С точки зрения набора адресов передачи управления:

- команды, не являющиеся командами передачи управления;
- команды безусловной передачи управления;
- команды условной передачи управления;
- команды вызова подпрограммы;
- команды возврата из подпрограммы;
- команды системного вызова (в т.ч. программные прерывания);
- команды останова.

С точки зрения вычислимости адреса перехода:

- передача управления по абсолютному константному адресу;
- передача управления по константному смещению относительно счётчика команд;
- передача управления по вычисляемому адресу.

Статический алгоритм восстановления CFG

На входе: дизассемблированная программа (инструкции, адреса, метки). Начальная инструкция I-start, текущая инструкция I, предыдущий базовый блок B-prev - пуст, текущий базовый блок B – пуст. S-**множество известных точек входа**, в виде пар (B-prev, I).

1. Извлекаем пару из S, B-prev = S[1], I-start = S[2]. Если S – пусто, останов.
2. next = Next(I). Если next содержит метку – шаг 3. Иначе I = next. Если I - инструкция передачи управления – шаг 3., иначе – шаг 2.
3. Создаём базовый блок B = (Addr (I-start), Size(I-start, I)), ребро E = (B-prev, B), если B-prev – не пуст. Тип ребра определяется типом последней инструкции (call, ret, другая). Если I-вызов или переход и **адрес вычислим добавляем** (B, Tartget(I)) в S.
4. Если I – не передача управления , I = Next(I), I-start = I, B-prev = B, переходим на шаг 2. Иначе – шаг 0.

Статическое восстановление

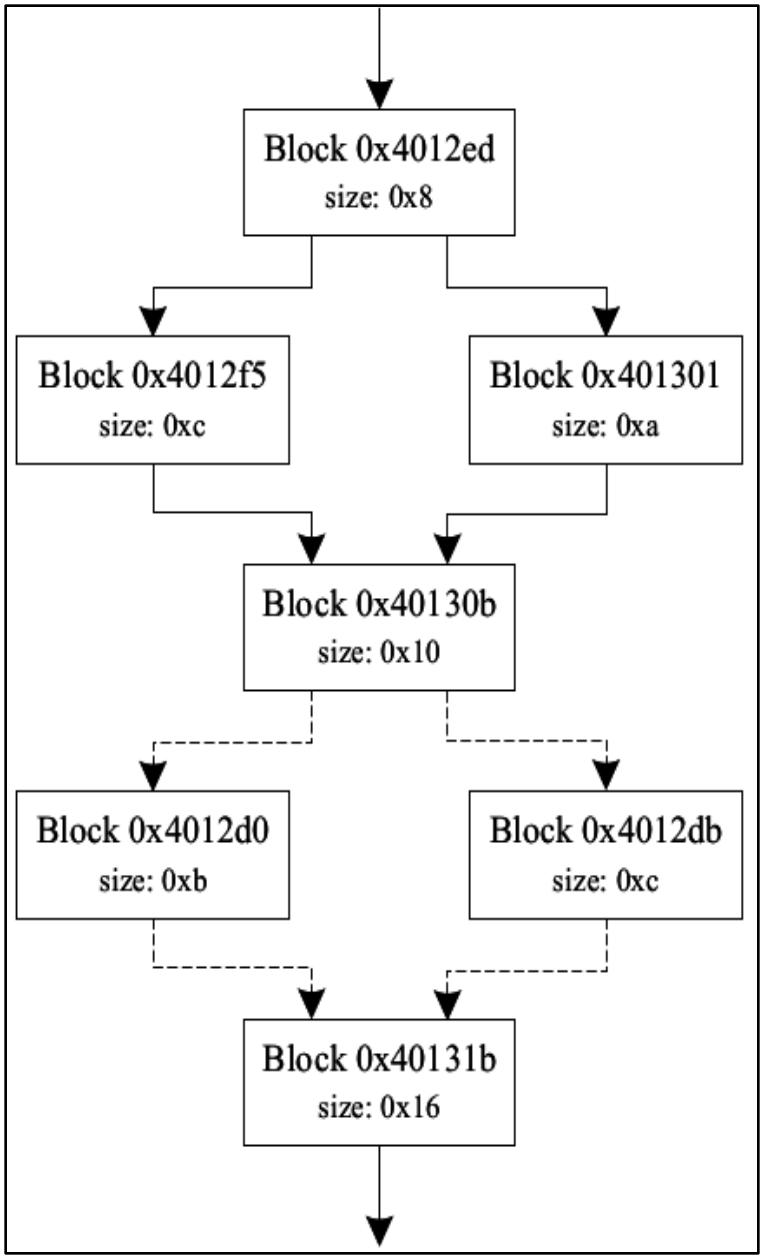
```
int (*foo)(int, int);

int add(int x, int y)
{
    return x + y;
}

int mul(int x, int y)
{
    return x * y;
}
```

```
int test(int a, int b)
{
        if (a < b)
        {
                foo = add;
        }
        else (a < b)
        {
                foo = mul;
        }
        c = foo(a, b);
        return 0;
}
```

```
0x004012ed:  mov    eax, DWORD PTR [ebp+8]
0x004012f0:  cmp    eax, DWORD PTR [ebp+12]
0x004012f3:  jmp    0x401301
0x004012f5:  mov    ds:0x403020, 0x4012d0
0x004012ff:  jmp    0x40130b
0x00401301:  mov    ds:0x403020, 0x4012db
0x0040130b:  sub    esp, 0x8
0x0040130e:  push   DWORD PTR [ebp+12]
0x00401311:  push   DWORD PTR [ebp+8]
0x00401314:  mov    eax, ds:0x403020
0x00401319:  call   eax
0x0040131b:  …
```



Классификация команд. Динамический подход

С точки зрения набора адресов передачи управления:

- команды, не являющиеся командами передачи управления;
- команды безусловной передачи управления;
- команды условной передачи управления;
- команды вызова подпрограммы;
- команды возврата из подпрограммы;
- команды системного вызова (в т.ч. программные прерывания);
- команды останова.

С точки зрения вычислимости адреса перехода:

- передача управления по абсолютному константному адресу;
- передача управления по константному смещению относительно счётчика команд;
- передача управления по вычисляемому адресу.

Динамический алгоритм восстановления CFG

На входе: трасса программы + метки. Начальная инструкция I-start = T-start, текущая инструкция I = I-start предыдущий базовый блок B-prev - пуст, текущий базовый блок B – пуст. Отображение S I -> B. Next(I) – следующая инструкция в трассе.

1. next = T.Next(I). Если next содержит метку – шаг 2 Иначе I = next. Если I - инструкция передачи управления – шаг 2., Иначе – шаг 1.
2. Создаём базовый блок B = (Addr (I-start), Size(I-start, I)), ребро E = (B-prev, B), если B-prev – не пуст. Тип ребра определяется типом последней инструкции (call, ret, другая). Добавляем (I-start, B) в S.
3. I-start = I, B-prev = B. Если I – не передача управления , I = next(I).
4. Осуществляем поиск по I в S – есть ли уже такой базовый блок – пропускаем все его инструкции в трассе, I = I-start = T.next(I-end) переход на шаг 4. Иначе - переходим на шаг 1.

Динамическое восстановление

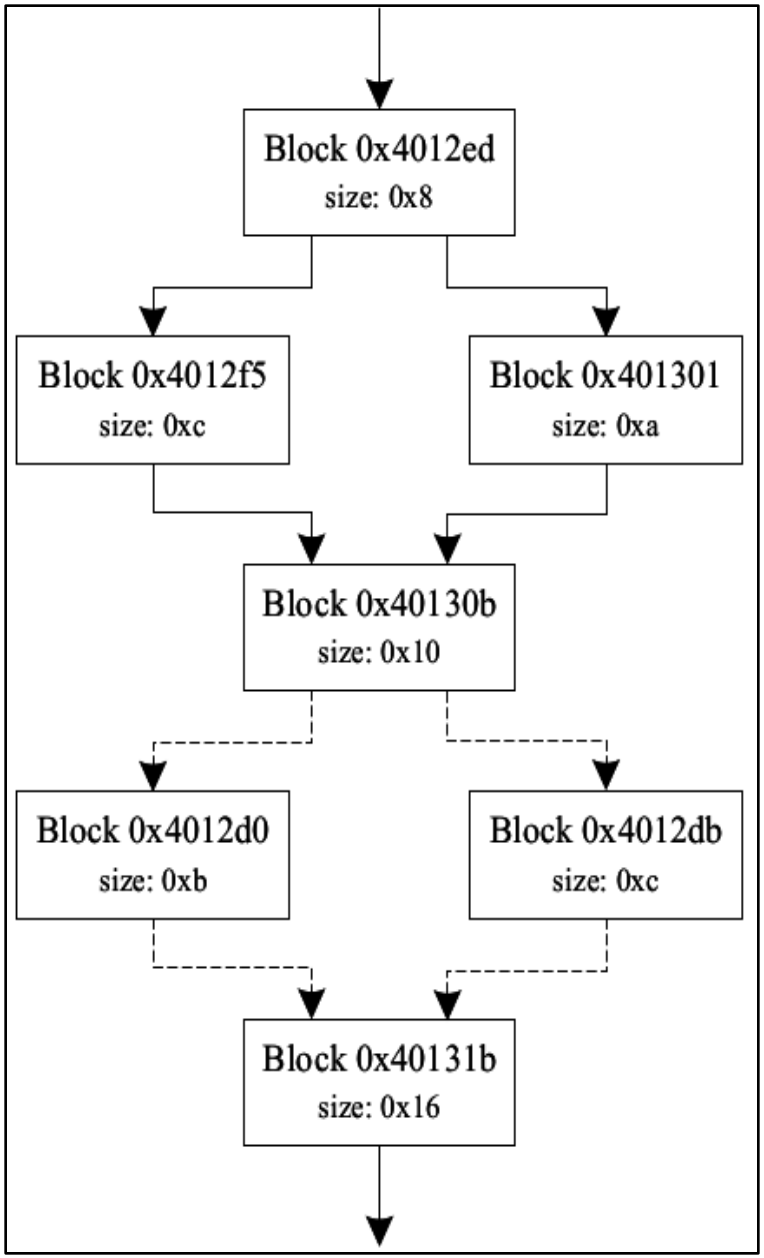
```
int (*foo)(int, int);

int add(int x, int y)
{
      return x + y;
}

int mul(int x, int y)
{
      return x * y;
}
```

```
int test(int a, int b)
{
      if (a < b)
      {
            foo = add;
      }
      else (a < b)
      {
            foo = mul;
      }
      c = foo(a, b);
      return 0;
}
```

```
0x004012ed:  mov    eax, DWORD PTR [ebp+8]
0x004012f0:  cmp    eax, DWORD PTR [ebp+12]
0x004012f3:  jmp    0x401301
0x004012f5:  mov    ds:0x403020, 0x4012d0
0x004012ff:  jmp    0x40130b
0x00401301:  mov    ds:0x403020, 0x4012db
0x0040130b:  sub    esp, 0x8
0x0040130e:  push   DWORD PTR [ebp+12]
0x00401311:  push   DWORD PTR [ebp+8]
0x00401314:  mov    eax, ds:0x403020
0x00401319:  call   eax
0x0040131b:  …
```



Сравнение подходов

Статический подход

- Потенциально полное покрытие кода
- «Упираемся» во все вычисляемые адреса

Динамический подход

- Покрытие кода только по трассе выполнения
- Нет проблемы вычисляемых адресов – целевые адреса всех передач управления известны

Сложности, возникающие при построении CFG

- Возможность модификации кода – в разное время по одним и тем же адресам находится разный код (разные базовые блоки)
- Наложение инструкции – код на одном и том же месте исполняется с разными смещениями, что приводит к исполнению разных инструкций и разным базовым блокам
- Неполнота построения меток – неточное разбиение на базовые блоки:
  - Непроанализирован код, выполняющий передачу управления
  - Не вычислен возможный адрес передачи управления

Ошибки межпроцедурности

- Приняли внутрипроцедурное ребро за межпроцедурное
  - «Вынесли» часть кода в отдельную функцию
  - Основная проблема согласованность рёбер – где-то должно быть соответствующее ребро вызова или возврата
- Приняли межпроцедурное ребро за внутрипроцедурное
  - «Инлайнинг»
  - Потенциальные вызовы из других функций в середину функции

- Возможность переходов в середину функций:
  - Функции с несколькими входами в языках (Fortran ENTRY)
  - Общие базовые блоки функций – оптимизация уменьшения кода

Следствия неполноты CFG

Отсутствие базовых блоков:

- неполнота потока данных

Отсутствие рёбер графа

- неполнота потока данных
- отсутствие зависимостей по управлению

Возможны ситуация, когда все базовые блоки есть, а часть рёбер не вычислена:

- Вычисляемые адреса (статика)
- Передача управления не по всем возможным адресам (динамика)

Наиболее частые случаи при динамическом подходе:

- Условный переход, всегда выполнявшийся в одну сторону
- Цикл с пост условием, выполнившийся один раз

Влияние оптимизаций на граф потока управления

- Частичный и полный разворот циклов
- Удаление прозрачных предикатов
- Замена пост условий в циклах над предусловия и наоборот
- Внос и вынос инструкций из/в цикл
- Внос/вынос инструкций из/в ветвления, дублирование базовых блоков в ветвях
- Инлайнинг функций
- Удаление хвостовой рекурсии
- …

Способы запутывания CFG

Увеличение доли вычисляемых переходов

- Замена вычисляемых переходов на невычисляемые

Непрозрачные предикаты:

- Добавление большого количества нереализующихся ветвей ветвей

Виртуальная машина

- Связывание базовых блоков в поток управления выполняет отдельный алгоритм на основе текущего состояния я идентификатора базового блока

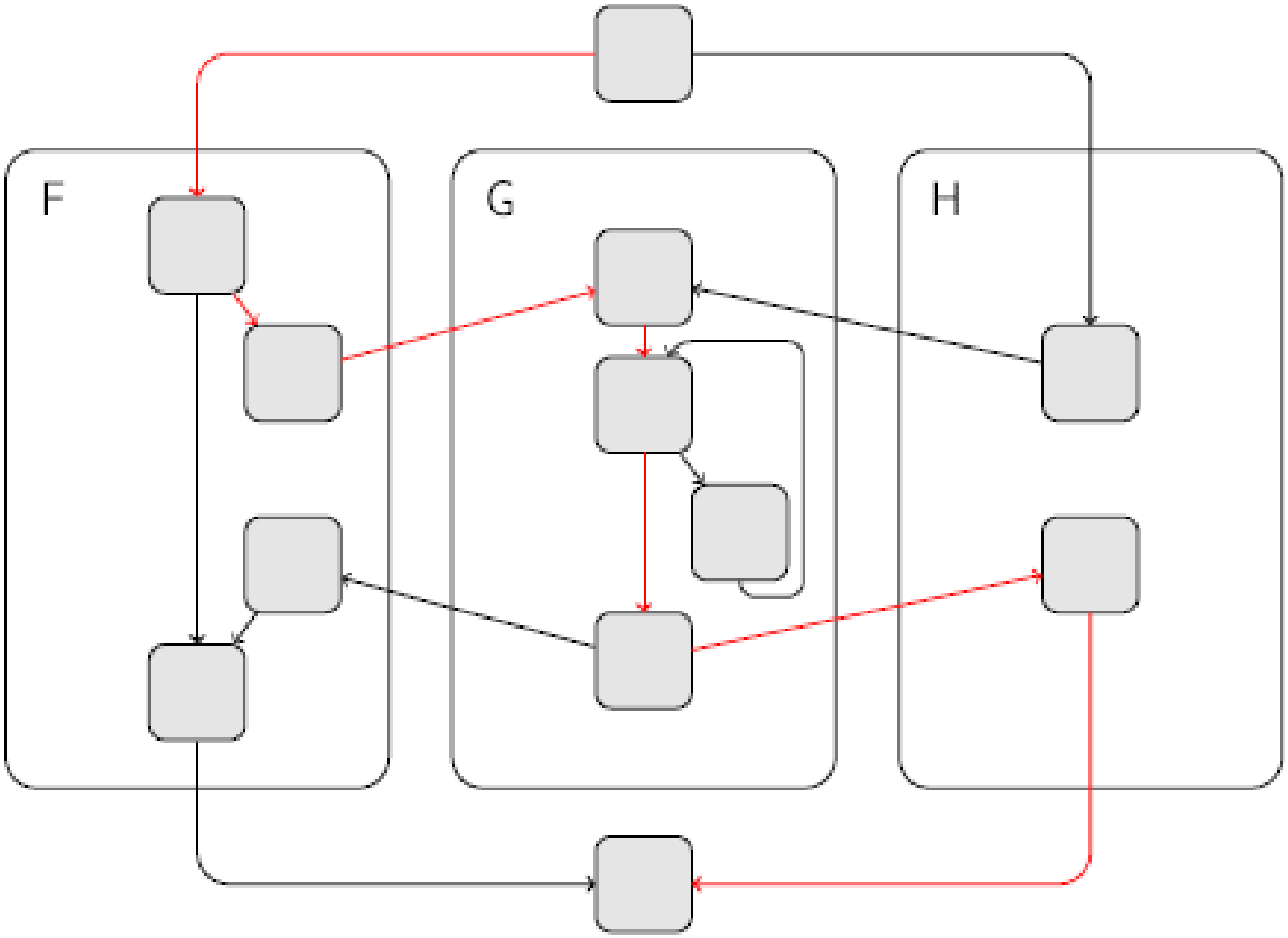
Использование CFG

1. Обнаружение недостижимого кода
2. Построение достигающих определений
3. Построение зависимостей по данным и по управлению
4. Выявление высокоуровневых управляющих конструкций (циклы с пред/постусловием, switch, …)
5. …

Важное свойство путей в CFG, влияющее на точность последующего анализа:

Пути в графе потока управления должны быть *реализуемыми*

Пример нереализуемого пути



Автомат с магазинной памятью

M = (K, A, P, s, F, M, m), где

K – конечное множество состояний автомата

A – допустимый входной алфавит, из которого формируются строки, считываемые автоматом

P – набор правил перехода между состояниями:
<K x A x S> -> <K x S>

s – начальное состояние

F – множество конечных состояний

M – алфавит стека

m – нулевой символ стека

Автомат смотрит текущий символ (из A) на входе, своё текущее состояние (из K), последний символ на стеке (из S) ищет подходящее правило и осуществляет переход в новое состояние (из K) и возможно действие с последним символом стека (из S)

Автомат для оценки реализуемости путей в CFG

Входной алфавит – множество рёбер E в CFG

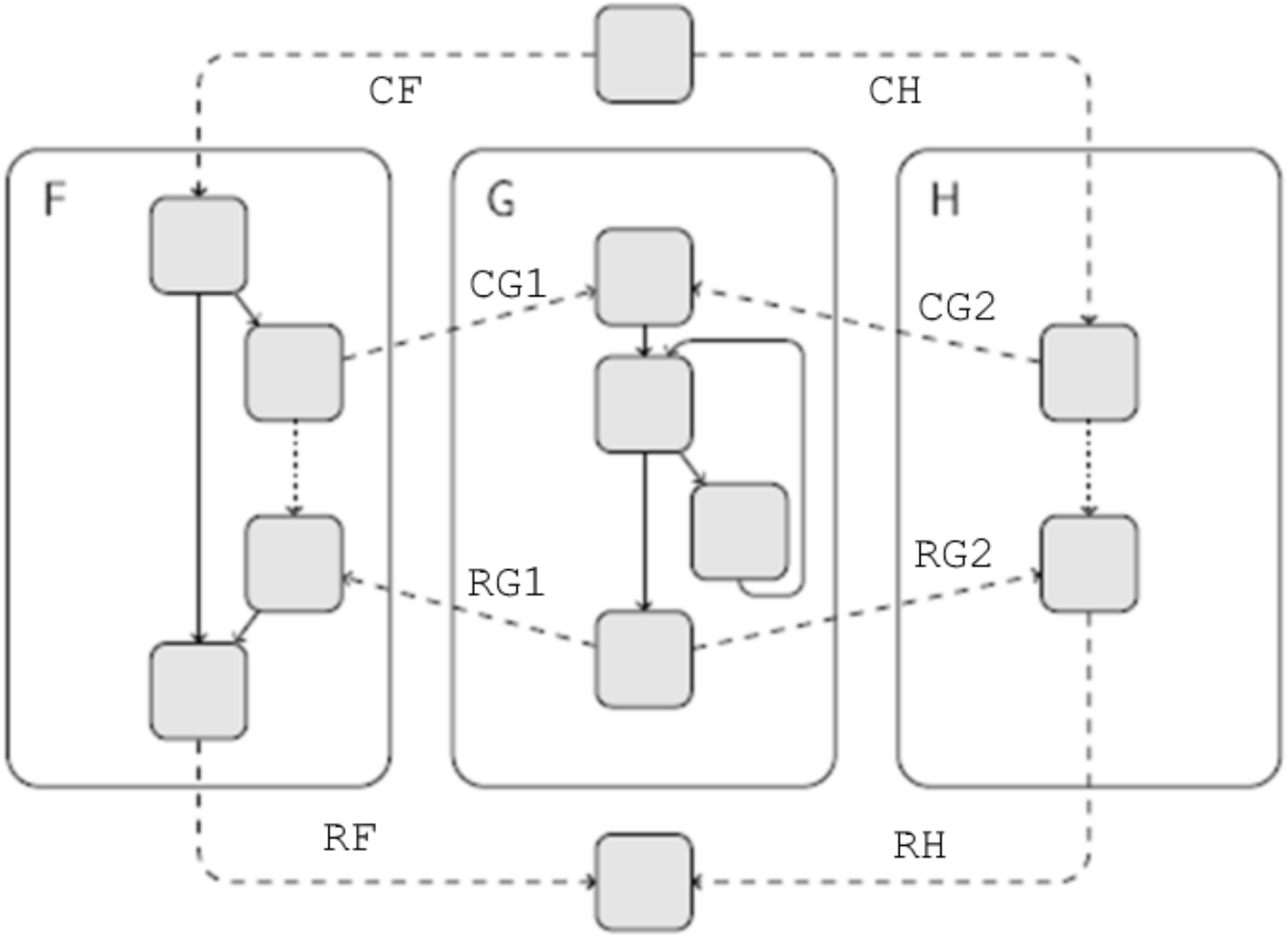
Состояний всего два – допустимый путь 1 (начальное) и недопустимый 0. Оба состояния – конечные.

Алфавит стека – множество рёбер типа вызов (C).

Правила переходов:

- Обычным рёбрам соответствуют переходы не меняющие состояние, но не стек <1, A, S> -> <1, S>
- Рёбрам типа «вызов функции», соответствуют переходы, добавляющие в стек ребро точки вызова <1, C, S> -> <1,S + C>
- Рёбрам типа «возврат из функции», соответствуют переходы: убирающие из стека состояние в точке вызова <1, C, S> -> <1, S - C>, если ребро возврата соответствует символу C в стеке.
- <1, C, S> -> <0, S> стек не меняется и автомат переходит в состояние “недопустимый путь”, если символ не соответствует или стек пуст

Автомат с магазинной памятью

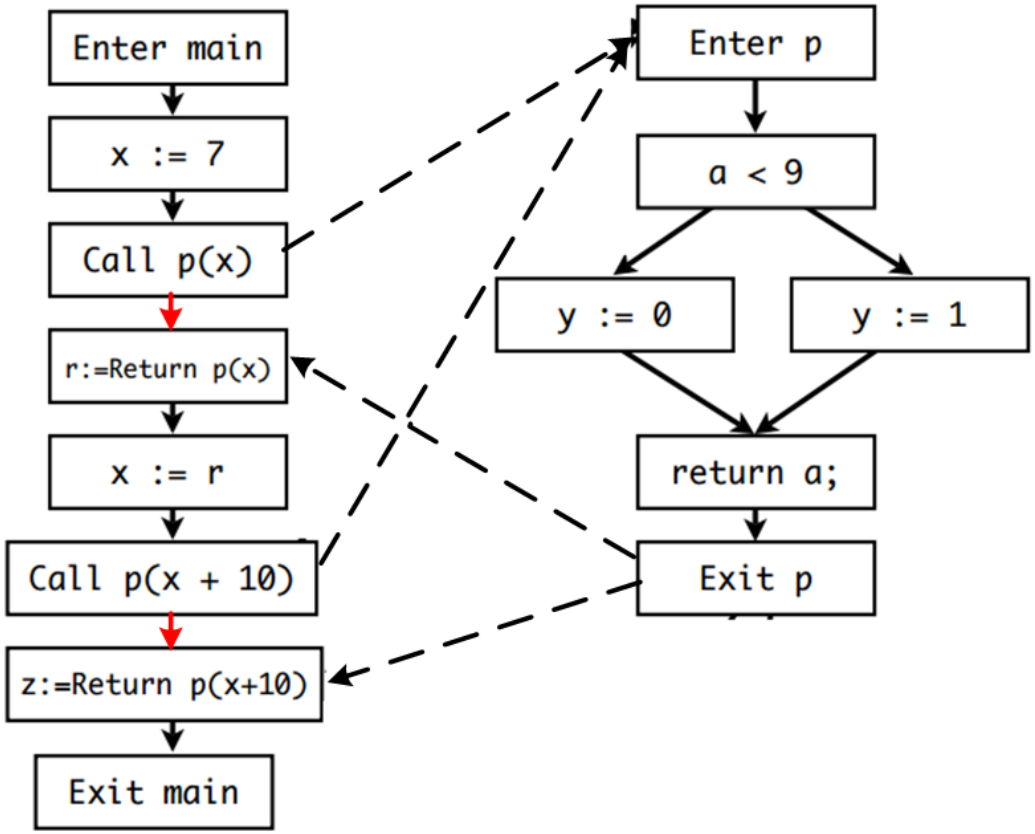


1102

ВОССТАНОВЛЕНИЕ ФУНКЦИЙ. ИДЕНТИФИКАЦИЯ БИБЛИОТЕЧНЫХ ФУНКЦИЙ.

Функции в исполняемом коде

- Адрес начала функции
- Диапазоны адресов, содержащих код функции
- Инструкция(-ии) возврата из функции



Базовый алгоритм восстановления функций

1. На входе алгоритма – восстановленный межпроцедурный граф потока управления (ICFG).
2. Каждое межпроцедурное ребро вызова соответствует вызову функции.
3. Граф, между рёбрами вызова и возврата – CFG функции. Адреса базовых блоков – адреса кода функции.
4. Адрес(а) входа в функцию – адрес(а), в который ведёт ребро из Entry- блока.
5. Адреса инструкций возврата – адреса последних инструкций блоков, из которых выходят рёбра, ведущие в Exit-блок.

Неточности ICFG, влияющие на восстановление функций

- Внутрипроцедурные ребра, пинятые за межпроцедурные
  - Инструкции CALL и RET, использующиеся как JMP
  - Обычно используется при обфускации
- Межпроцедурные ребра, принятые за внутрипроцедурные
  - Вызов или возврат с помощью инструкции JMP
  - Заглушки при вызове внешних функций
  - Пример - возврат из функции Microsoft Windows XP, KiSystemService

    `POP EDX` – выталкивание адреса из стека в регистр EDX

    `JMP EDX` – переход по адресу в регистре EDX

Восстановление стека вызовов по трассе

Идея:

- Поиск всех инструкций передачи управления, которые могут осуществлять вызов и возврат
- Поиск пар инструкций, таких, что:
  - Вторая инструкция осуществляет переход на инструкцию следующую за первой (`addr+size`)
  - Пара инструкция соответствует стеку вызовов (нет незавершённых вызовов)
  - Адрес перехода первой инструкции – адрес функции (`funcAddr`)

Трасса:

```
addr:   JMP funcAddr
        …
        …
        JMP addr+size
```

Восстановление функций по трассе исполнения

1. Вначале стек пар <инструкция, адрес перехода> пуст.
2. Проход по трассе выполнения сверху вниз.
3. Поиск инструкций перехода, получение адреса перехода (адрес следующей инструкции в трассе).
4. Просмотр всего снизу вверх стека, поиск инструкции, для которой адрес перехода является «следующим»
5. При обнаружении:
   - считаем пару инструкций вызовом (адрес её перехода – адрес функции) и возвратом.
   - Все инструкции в стеке после «вызова» - удаляются (они не являются вызовами при условии корректности стека)
6. Инструкция в стеке не обнаружена:
   - добавляем текущую инструкцию в стек, как возможную инструкцию вызова с её адресом перехода

Проблемы восстановления функций по трассе

1. Стеки соответствуют нитям выполнения – требуется предварительное разделение трассы
2. Заглушки при вызове внешних функций и оптимизации внешних вызовов этим подходом не обнаруживаются
3. Возникновение прерываний в моменты вызова и возврата – трасса сразу после вызова или прерывания содержит обработчик прерывания:
   - В случае вызова – сложность с определением адреса функции
   - В случае возврата – сложность с определением вызова, которому соответствует возврат
4. Требуется предварительно найти возвраты из прерываний (каскады прерываний, удвоение инструкций)

Распознавание библиотечных функций

- Значительная часть кода – стандартные функции (например, ‘Hello world’ может содержать 58)
- Сборок библиотечных функций очень много: большое количество библиотек, разные версии, сборка разными компиляторами с разными настройками
- Дополнительный бонус - идентификация функций позволяет восстановить семантику кода и параметров в окрестностях их вызова, т.к. описание функций и их параметров можно найти в описании библиотек (например, описание WinAPI можно найти в MSDN)

Идея технологии FLIRT (IDA Pro)

Создание базы сигнатур библиотечных функций. Условия:

- База не должна занимать много места (с учётом количества функций)
- Поиск известных функций должен быть достаточно быстрым
- Отсутствие ложных срабатываний
- Максимально возможное разделение похожих функций

Особенности технологии FLIRT

- Рассматриваются библиотеки на Си\Си++
- Сигнатура отдельной функции – первые 32 байта тела функции
- В случае если значение байта может меняться, используется символ ‘.’

```
558BEC0EFF7604.........59595DC3558BEC0EFF7604.........59595DC3 _registerbgidriver
558BEC1E078A66048A460E8B5E108B4E0AD1E9D1E980E1C0024E0C8A6E0A8A76 _biosdisk
558BEC1EB41AC55604CD211F5DC3..................................... _setdta
558BEC1EB42FCD210653B41A8B5606CD21B44E8B4E088B5604CD219C5993B41A _findfirst
```

Особенности технологии FLIRT (2)

- Для компактности хранения и ускорения поиска используется структра «Префиксное дерево»

```
558BEC
      0EFF7604.........59595DC3558BEC0EFF7604.........59595DC3 _registerbgidriver
      1E
        078A66048A460E8B5E108B4E0AD1E9D1E980E1C0024E0C8A6E0A8A76 _biosdisk
      B4
        1AC55604CD211F5DC3                                     _setdta
        2FCD210653B41A8B5606CD21B44E8B4E088B5604CD219C5993B41A _findfirst
```

- Если первые 32 байта функций одинаковые – они хранятся в одном листе дерева. Для их различения используется значение CRC16 с 33 байта по первый переменный байт.

```
558BEC
      56
        1E
          B8....8ED8
                   33C050FF7608FF7606.........83C406
                                                    8BF083FEFF
                    0. _chmod   (20 5F33)
                    1. _access  (18 9A62)
```

Алгоритм поиска сигнатур

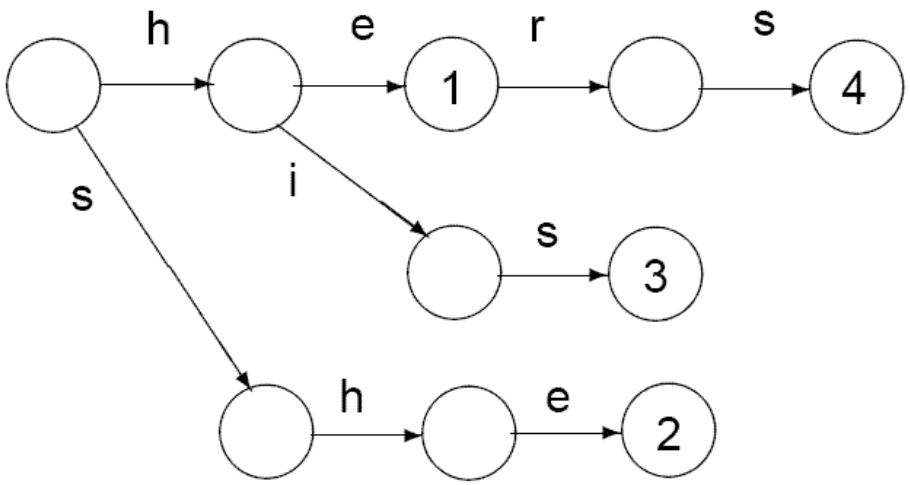
- Для поиска в исполняемом файле функций по заданному дереву сигнатур используется алгоритм Ахо-Корасик .
- Данный алгоритм используется для массового поиска строк в тексте.
- В случае технологии FLIRT:
  - строка –> сигнатура функции
  - текст –> сегмент кода в исполняемом файле
- Для поиска данный алгоритм использует структуру модифицированное префиксное дерево

Алгоритм Ахо-Корасик

- Алгоритм строит конечный автомат на основе строк, которые требуется найти (словарь)
- Затем на вход автомату передаётся текст.
- Автомат получает по очереди все символы текста и переходит по рёбрам с соответствующими символами.
- Если автомат пришёл в конечное положение, соответствующая строка присутствует в тексте.

Префиксное дерево для набора слов

{he, she, his, hers}



Свойства алгоритма Ахо-Корасик

- Самовосстановление – переход в оптимальное состояние для дальнейшего поиска, если строка не совпала.
- Для этого используются *суффиксные ссылки* – рёбра с пустым символом – автомат становится недетерминированным
- Каждому состоянию соответствует слово, которое в него привело
- Для каждого состояния вычисляется самый длинный суффикс, который ведёт в какое-либо другое состояние
- В это состояние проводится ссылка

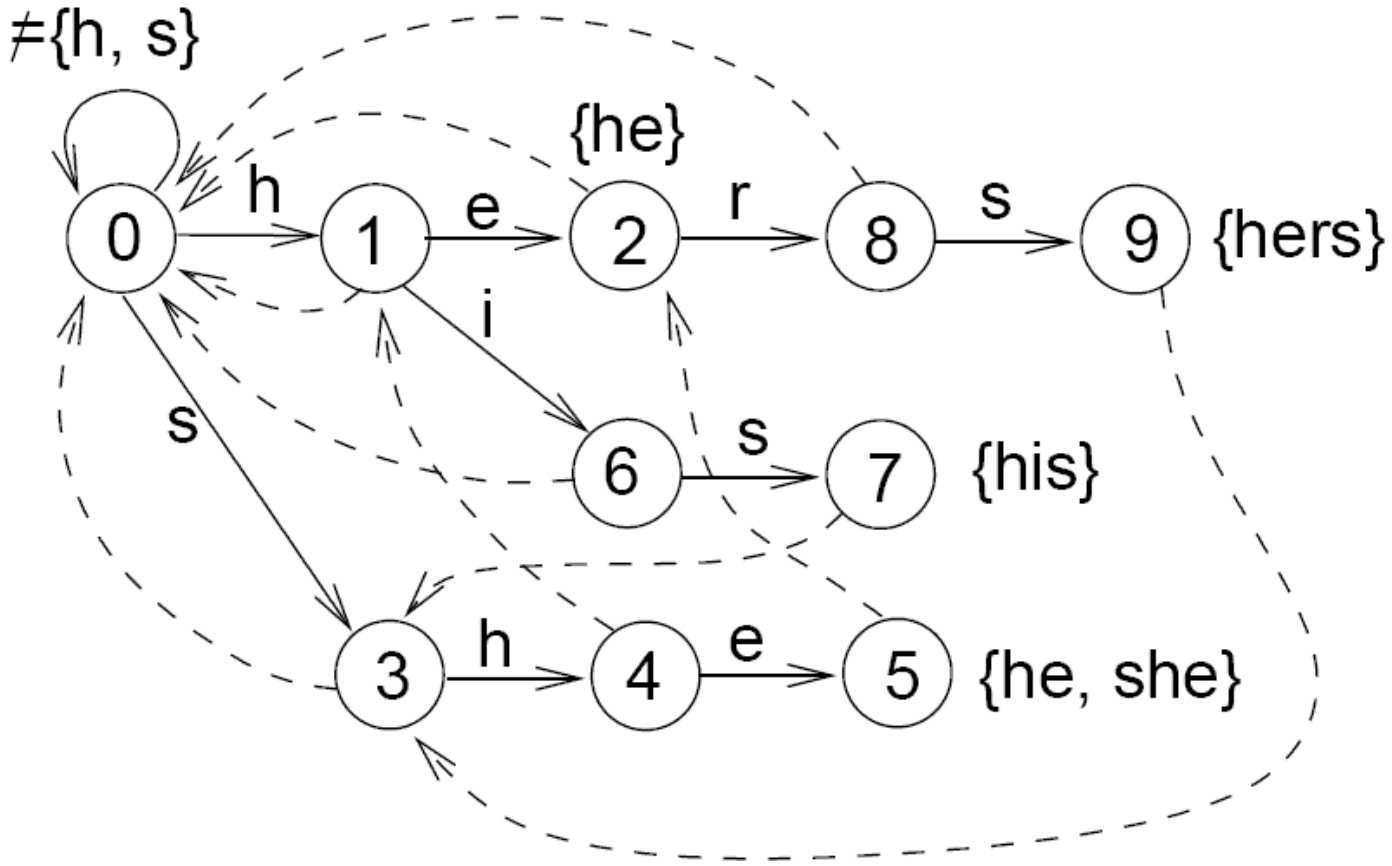
Свойства алгоритма Ахо-Корасик (2)

- Преобразование недетерминированного конечного автомата в детерминированный в общем случае приводит к значительному увеличению количества вершин.
- Автомат поиска можно превратить в детерминированный, не создавая новых вершин: если для вершины v некуда идти по символу c, проходимся по суффиксной ссылке ещё и ещё раз — пока либо не попадём в корень, либо будет куда идти по символу c.

Свойства алгоритма Ахо-Корасик (3)

- Детерминизация увеличивает количество **конечных** вершин. А именно: если суффиксные ссылки из вершины v ведут в конечную u, сама v тоже объявляется конечной.
- Для этого используются *конечные ссылки* : конечная ссылка ведёт на ближайшую по суффиксным ссылкам конечную вершину. Пройдя по конечным ссылкам, находим все совпавшие строки.

Пример модифицированного префиксного дерева



Модифицированное префиксное дерево для набора слов {he, she, his, hers}

Литература к лекции

Основные источники

1. Xiaozhu Meng, Barton P. Miller. Binary code is not easy. Proceedings of the 25th International Symposium on Software Testing and Analysis, 2016
2. T. Reps, A. Lal, and N. Kidd. Program analysis using weighted pushdown systems. In FSTTCS, 2007.
3. Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. BYTEWEIGHT: Learning to recognize functions in binary code. In USENIX Security Symposium, 2014.
4. IDA F.L.I.R.T. Technology: In-Depth

   www.hex-rays.com/products/ida/tech/flirt/in_depth.shtml