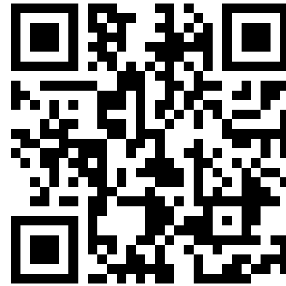


Анализ кода
и информационная безопасность

Лекция 07

МГУ / ВМК / СП

0701

ФАЗЗИНГ. РАЗНОВИДНОСТИ. ГЕНЕТИЧЕСКИЕ АЛГОРИТМЫ В ФАЗЗИНГЕ.

Определение

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.**

Цитаты автора метода

1990: “[Fuzzing] is not a substitute for a formal verification or testing procedures, but rather an inexpensive mechanism to identify bugs...”

1995: “While [fuzzing] is effective in finding real bugs in real programs, we are not proposing it as a replacement for systematic and formal testing.”

2000: “Simple fuzz testing does not replace more extensive formal testing procedures.”

- Barton Miller

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

Простейший пример

- Стандартный запрос HTTP GET
  - `GET /index.html HTTP/1.1`
- Варианты аномальных запросов:
  - `AAAAAA...AAAA /index.html HTTP/1.1`
  - `GET ///////index.html HTTP/1.1`
  - `GET %n%n%n%n%n%n.html HTTP/1.1`
  - `GET /AAAAAAAAAAAAA.html HTTP/1.1`
  - `GET /index.html HTTTTTTTTTTTTTP/1.1`
  - `GET /index.html HTTP/1.1.1.1.1.1.1.1`

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

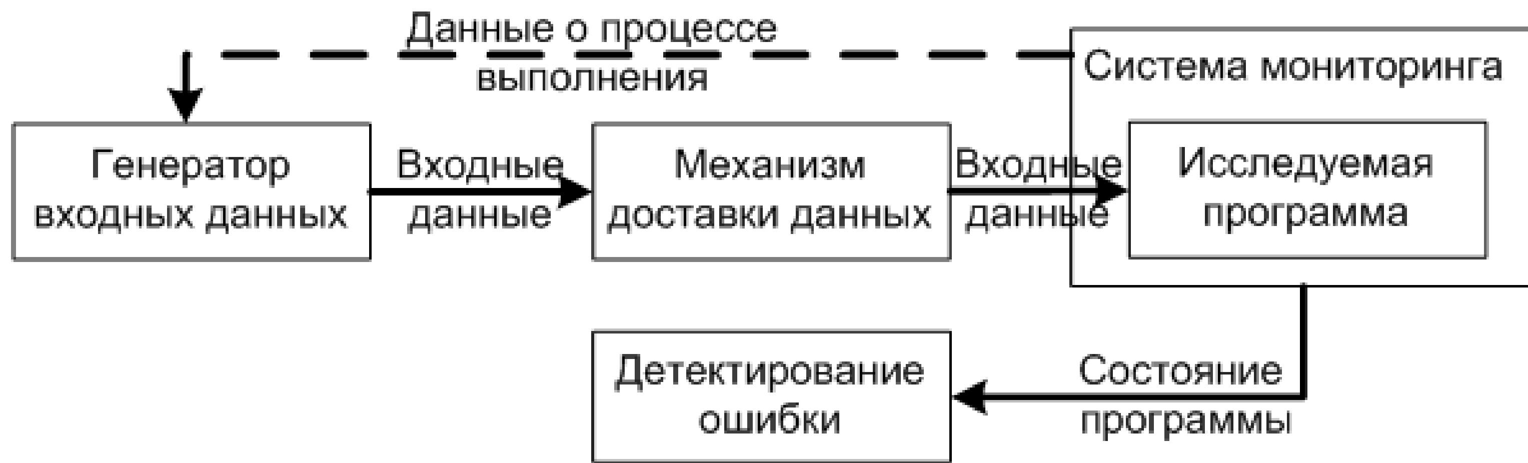
Объекты тестирования

- Форматы файлов
  - PDF, Microsoft word, шрифты TrueType, …
- Сетевые протоколы
  - RDP, VNC, SSL, VoIP, …
- Программные объекты
  - Объекты COM, вызовы API, межпроцессные взаимодействия
- Web-приложения
  - Joomla, WordPress, сайты

Известные инструменты

- SPIKE, Peach, AFL
  - Среда разработки, протоколы, файлы, web
- Basic Fuzzing Framework (BFF)
  - Файлы
- SAGE (Microsoft)
  - Входные данные программ
- AutoFuzz
  - Сетевые протоколы с помощью MITM
- COMRaider
  - Объекты COM
- IOCtrlFuzzer (eSage Lab)
  - Функция API NtDeviceIoControlFile драйверов Windows

Общая схема системы фаззинга



- **Генератор входных данных – поставляет образцы данных для передаче программы. Большое число подходов к генерации. Центральный компонент – качество генерации определяет эффективность тестирования.**
- Механизм доставки данных – представляет образцы в форме, в которой их ожидает получить программа (файлы, …).
- Система мониторинга – определение корректности поведения программы, от реализации зависят классы ошибок, которые могут быть обнаружены.

Механизм доставки данных

Определяет класс программ и их компонент, который может тестироваться данной реализацией фаззинга. Различаются по поддерживаемым «входным интерфейсам» и методам доставки.

- «Входные интерфейсы»:
  - Файлы
  - Сетевые пакеты
  - Переменные среды
  - Параметры запуска (аргументы командной строки)
  - События операционной системы (мышь, клавиатура)
  - Ресурсы операционной системы (данные реестра, …)
- Способы доставки:
  - Неинвазивный (модификация внешних данных)
  - Перехват вызовов и модификация памяти программы

Система мониторинга

Основная характеристика – способность детектировать ошибочную ситуацию.

Виды:

- Локальные – система запущена на том же компьютере, что и анализируемая программа
- Удалённые – система мониторинга может взаимодействовать с тестируемой программой только анализируя входные и выходные данные

Дополнительная характеристика – сведения о процессе выполнении программы на данных входных данных:

- Выполнявшиеся инструкции
- Вызовы библиотечных функций
- ...

Способы детектирования ошибок

- Удалённые системы
  - Обрыв соединения (TCP) превышения таймаута (UDP)
  - Анализ файлов журнала сервера
- Локальные системы
  - Детектирование аварийного завершения программы:
    - Поиск файлов дампа памяти после завершения
    - Анализ возвращаемого значение
    - Перехват исключений
  - Детектирование зависания - эвристики
  - Детектирование некорректной работы
    - Перехват библиотечных вызовов, анализ корректности параметров (например – работы с памятью)

Основные принципы фаззинга

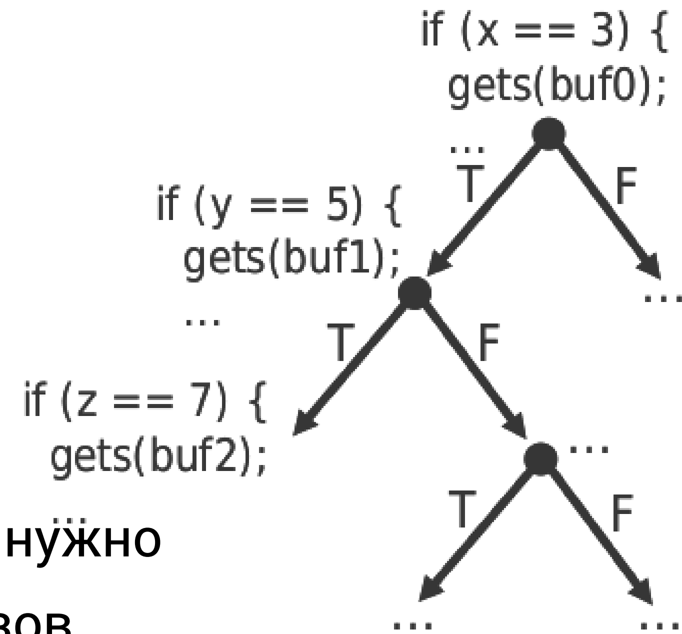
- Автоматическая генерация входных данных
- Входные данные должны быть настолько корректными, чтобы проходить простейшие проверки корректности в программе и настолько некорректными, чтобы вызывать ошибку в процессе их обработки.
- Два основных подхода:
  - На основе мутации – изменения множества изначально данных корректных входов
  - На основе генерации – генерация на основе общего описания формата входных данных (например формата файла)

Поверхностные и глубокие ошибки

```
foo(int x, int y, int z) {
if (x == 3) {      //p = 1/2^32
  gets(buf0);

  if (y == 5) {   //p = p * 1/2^32
    gets(buf1);

    if (z == 7) { //p = p * 1/2^32
       gets(buf2);
    }
  }
}
}
```



Количество состояний, которые нужно перебрать для попадания на вызов

gets(buf0) =	4,294,967,296
gets(buf1) =	18,446,744,073,709,551,616
gets(buf2) =	79,228,162,514,264,337,593,629,020,928

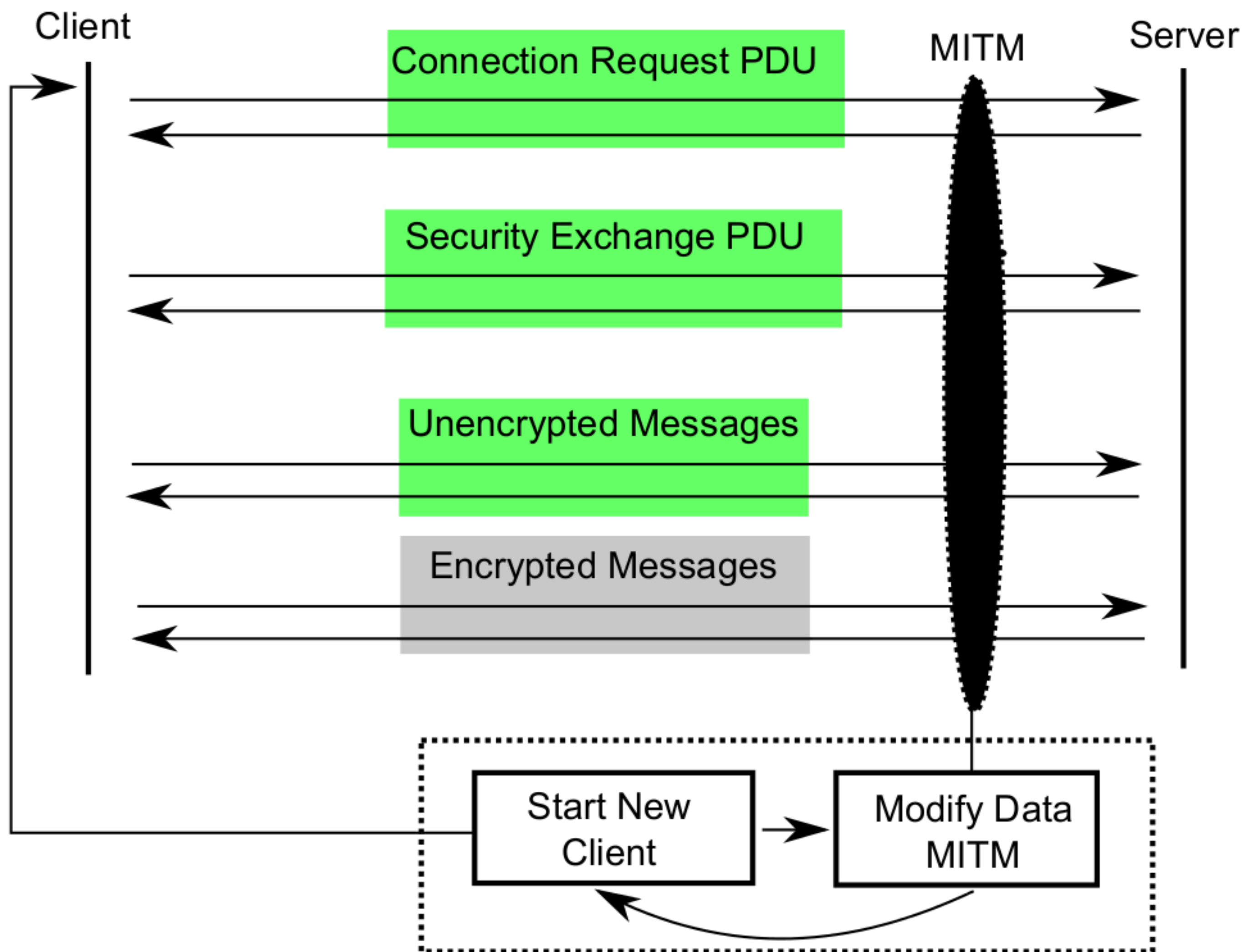
Мутационный фаззинг

- Применяется в случае, если мало или вообще ничего не известно о структуре входных данных
- «Аномалии» добавляются в известные корректные входные данные
- «Аномалии» могут быть полностью случайными или добавляться на основе эвристик. Например:
  - Удаление терминальных символов (0 в Си-строках)
  - Сдвиг данных
- Сильные стороны:
  - Простая настройка и автоматизация
  - Не требуется знание о структуре входных данных
- Слабые стороны:
  - Ограничен набором корректных входных данных
  - Часто не проходит простейший проверки корректности в программе

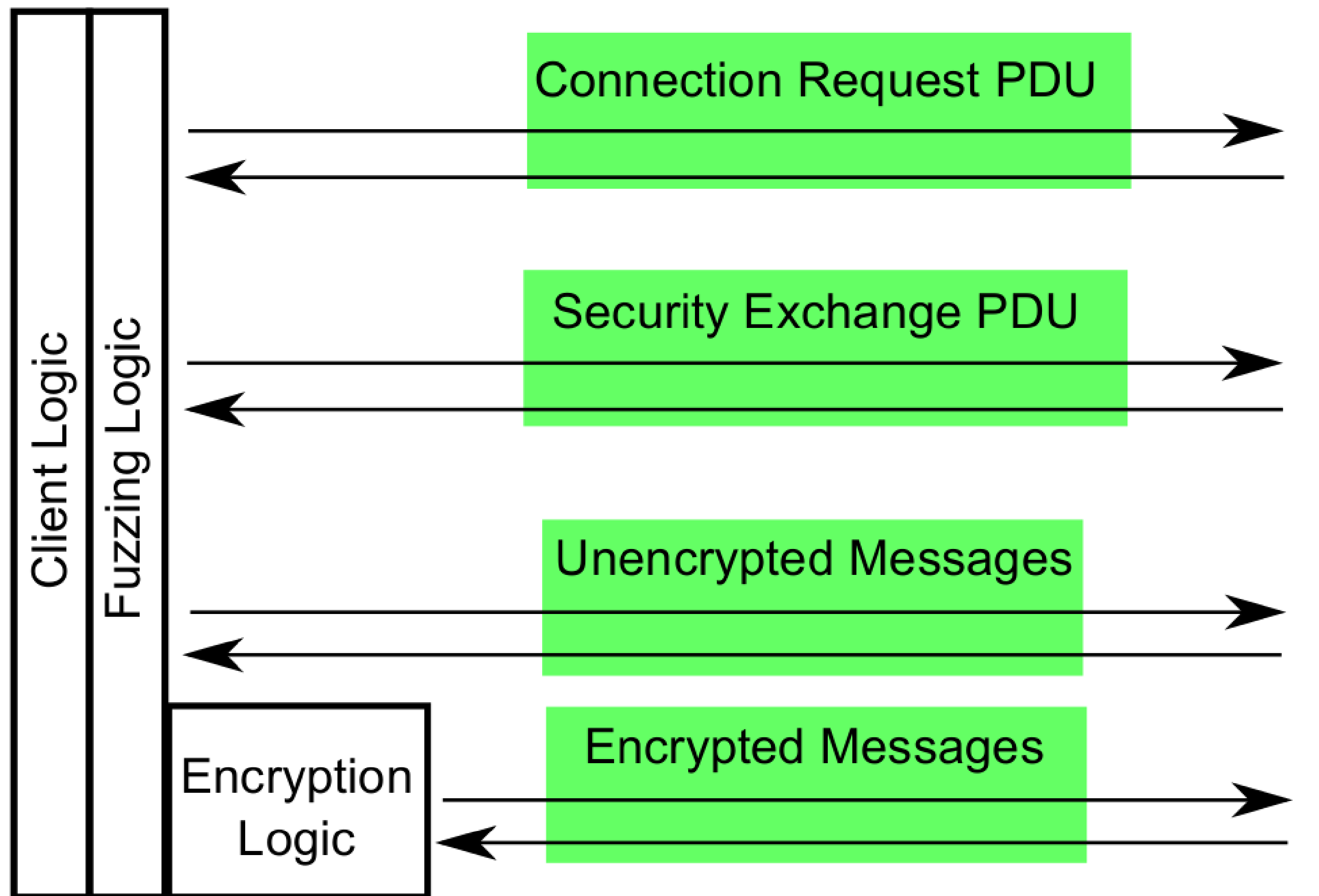
Пример: фаззинг PDF-просмотровщика

1. Запуск поисковика для поиска pdf файлов (~миллиард результатов)
2. Загрузка набора корректных файлов
3. Использование инструмента фаззинга:
   1. Считывание файла
   2. Мутация файла
   3. Запуск программы с передачей мутированного файла
   4. Если программа завершается аварийно – записать вход, процесс выполнения и передать на дальнейший анализ

Пример: фаззинг сетевого протокола



Пример: фаззинг сетевого протокола (2)



Фаззинг на основе генерации

- Корректные входы создаются на основе описания формата файла или протокола, например RFC
- «Аномалии» добавляются в различные структурные части входных данных, например поля заголовка файла
- Сильные стороны:
    - Большая полнота покрытия
    - Возможность поддержки зависимостей в данных, например контрольных сумм
- Слабые стороны:
    - Необходима спецификация входных данных
    - Написание генератора – сложная задача

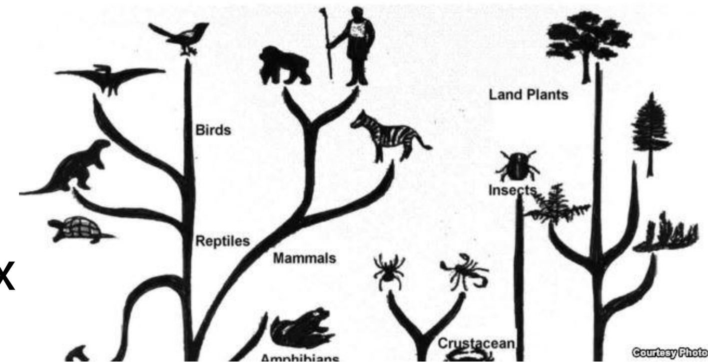
Развитие методов генерации

- Полностью случайные входные данные – случайный выбор места и содержания «аномалии» в случае мутационного подхода или генерация случайной строки.
- Использование шаблонов – участков входных данных, которые мутируют или генерируются независимо.
- Использование структур – разбиение всего входа на поля и последовательности, с указанием зависимостей между их содержимым (поля длины, контрольные суммы).
- Описание части *входного языка* программы с помощью грамматики и создание входов, которые являются корректными словами языка.
- Использование эвристик – учёт покрытия кода, количества потенциально опасных вызовов.
- Применение генетических алгоритмов.

Генетические алгоритмы в фаззинге

Требуется обратная связь от системы мониторинга для оценки «жизнеспособности» кандидатов. Обычно используется в мутационном подходе. Алгоритм:

1. Получение текущего набора входов. Вначале – набор корректных данных.
2. Мутирование текущего набора входов – получение следующего поколения входных данных.
3. Мониторинг выполнения на каждом из кандидатов. Отбор «жизнеспособных мутаций» на основе метрик:
   - Лучшее **покрытие кода**
   - Выполнение вызовов потенциально опасных функций (memcpy)
4. «Смешивание» наиболее «жизнеспособных мутаций». Переход к пункту 1.



Сравнение фаззеров. Оценка полноты тестирования

- Одна из используемых метрик – покрытие кода
- Виды покрытия кода:
  - по строкам кода – какая доля строк кода программы выполнялась в процессе фаззинга
  - по ветвям выполнения – по какой доле ветвей в условных переходов была передано управление
  - по путям выполнения – какая доля всех путей выполнения программы была изучена

Пример

```
1 if( a > 2 )
2     a = 2;
3 if( b > 2 )
4     b = 2;
```

Количество входов для полного покрытия:

- 1 – по строкам
- 2 – по ветвям
- 4 – по путям

Ограничения применимости подхода с покрытием кода

- Код может быть покрыт без возникновения ошибки

```
mySafeCpy(char *dst, char* src){
    if(dst && src)
        strcpy(dst, src);
}
```

- Код проверки ошибок тяжело покрыть, если не заниматься этим целенаправленно

```
ptr = malloc(sizeof(blah));
if(!ptr)
    ran_out_of_memory();
```

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.**

0702

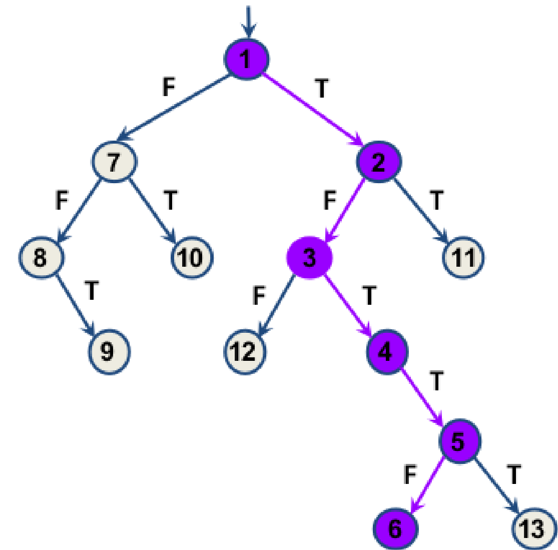
СИМВОЛЬНОЕ ВЫПОЛНЕНИЕ. ПРЕДИКАТЫ ПУТИ И БЕЗОПАСНОСТИ. ОГРАНИЧЕНИЯ.

Что такое символьное выполнение?

- Технология предложенная в 70-х для тестирования программ
- «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

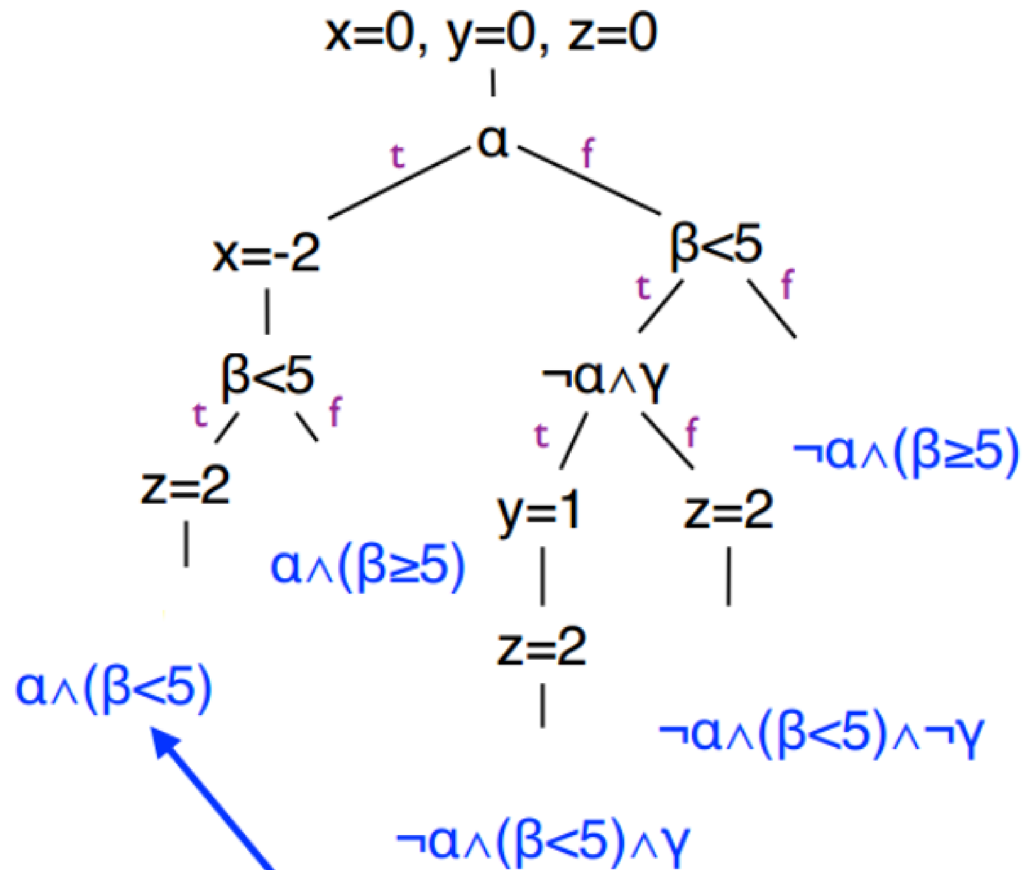


Дерево символьного выполнения

Символьные значения

int a = α, b = β, c = γ;

```
int x = 0, y = 0, z = 0;
if (a) {
  x = -2;
}
if (b < 5) {
  if (!a && c)  { y = 1; }
  z = 2;
}
assert(x+y+z!=3)
```



Предикат пути

Сравнение конкретного и символьного выполнения

Программа	Конкретное выполнение	Символьное выполнение
`int foo(int i){`	i = 1	i_{input}
`int j = 2*i;`	i = 1, j = 2	$i = i_{input},\ j = 2 * i_{input}$
`i = i++;`	i = 2, j = 2	$i = i_{input} + 1,\ j = 2 * i_{input}$
`i = i * j;`	i = 4, j = 2	$i = 2 * i_{input} \wedge 2 + 2 * i_{input}$
`if ( i < 1 )`		$i = - 2 * i_{input} \wedge 2 - 2 * i_{input}$ $(2 * i_{input} \wedge 2 + 2 * i_{input} < 1)$
`i = -i ;`		$i = 2 * i_{input} \wedge 2 + 2 * i_{input}$ $(2 * i_{input} \wedge 2 + 2 * i_{input} >= 1)$
`return i;`	return 4	
`}`		

Применение символьного выполнения

- Основная цель: определение семантики программы
- Приложения:
  - Определение нереализуемых путей
  - Генерация тестовых наборов входных данных для увеличения покрытия кода
  - Доказательство эквивалентности двух фрагментов кода
  - Поиск ошибок и уязвимостей

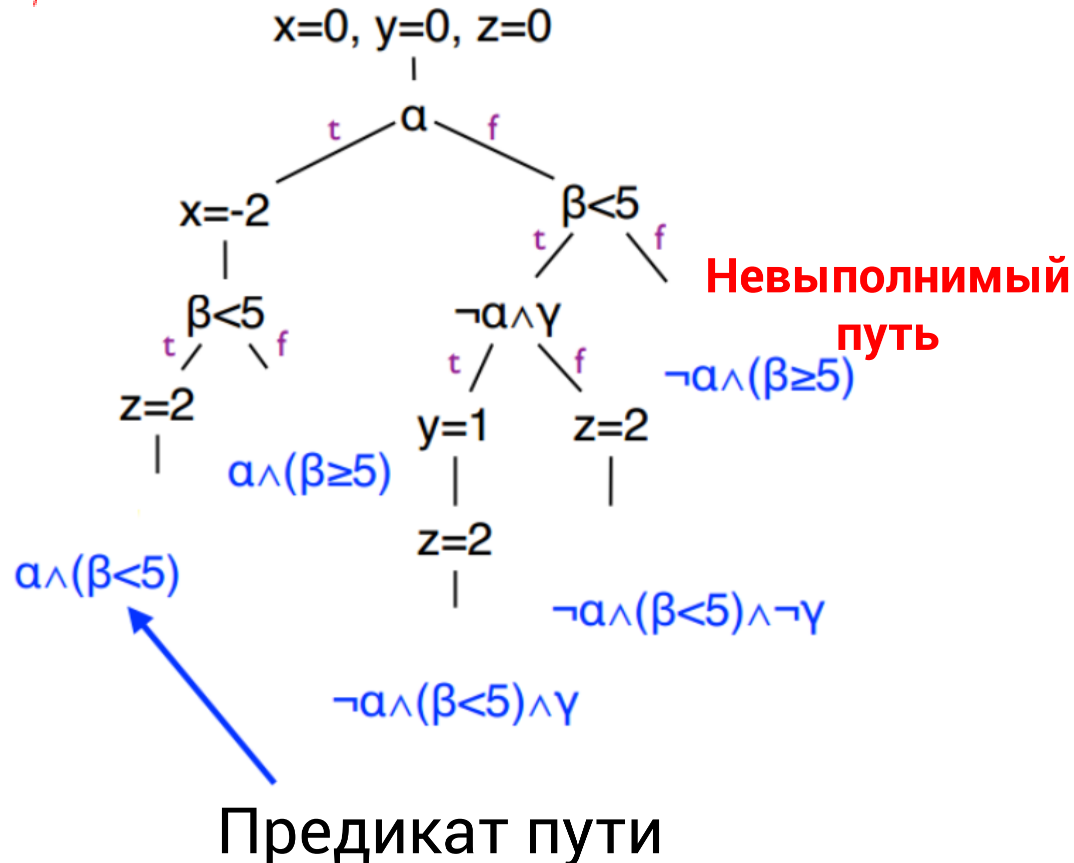
Определение нереализуемых путей

Символьные значения

```
int a = α, b = β, c = γ;
```

Пусть есть условие: $\alpha = \beta$

```
int x = 0, y = 0, z = 0;
if (a) {
  x = -2;
}
if (b < 5) {
  if (!a && c)  { y = 1; }
  z = 2;
}
assert(x+y+z!=3)
```

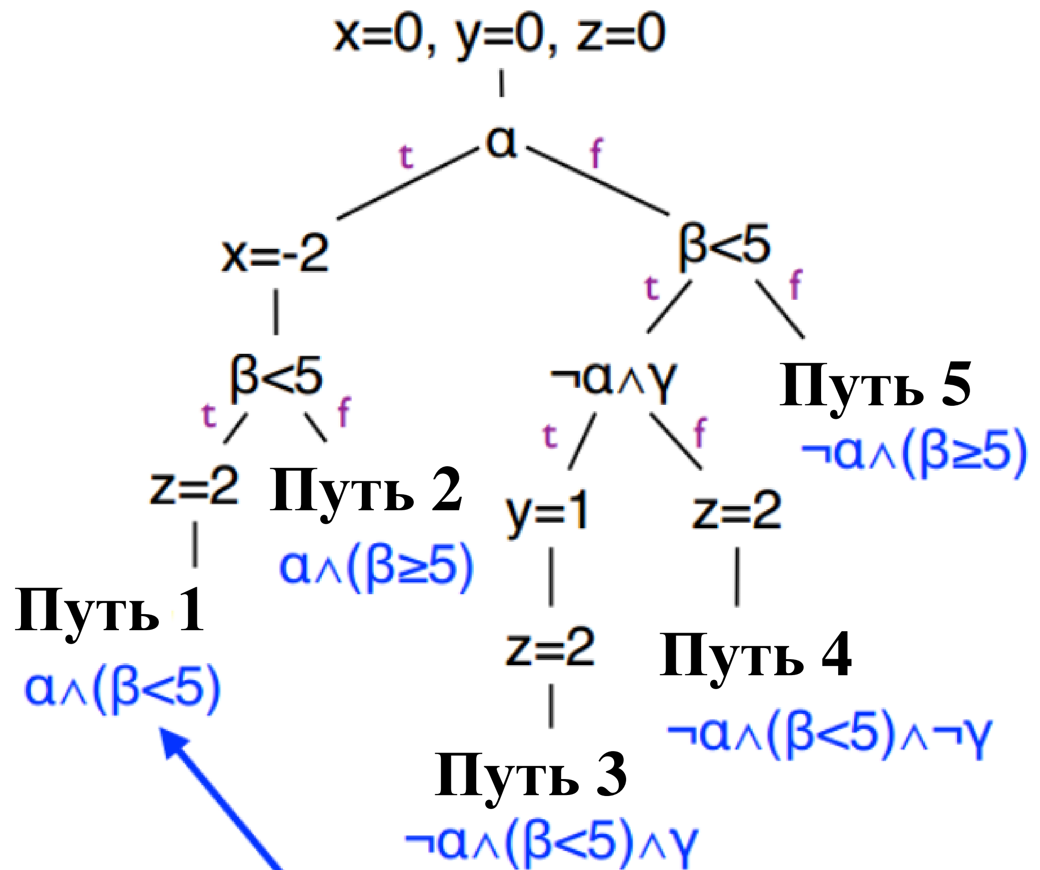


Генерация тестовых входных данных

Символьные значения

int a = α, b = β, c = γ;

```
int x = 0, y = 0, z = 0;
if (a) {
  x = -2;
}
if (b < 5) {
  if (!a && c)  { y = 1; }
  z = 2;
}
assert(x+y+z!=3)
```



Путь 1: $\alpha = 1, \beta = 1$

Путь 2: $\alpha = 1, \beta = 6$

Путь 3: ...

Поиск ошибок и уязвимостей

x – входное значение
char buf[42];

x может быть любым

if x>0

$(x > 0)$

f t

if x*x < 0xffffff

$(x > 0) \wedge (x*x < 0xffffff)$

f t

strcpy(buf, input)

Критерий на переполнение буфера:

$(x > 0) \wedge (x*x < 0xffffff) \wedge (strlen(input >= 42))$

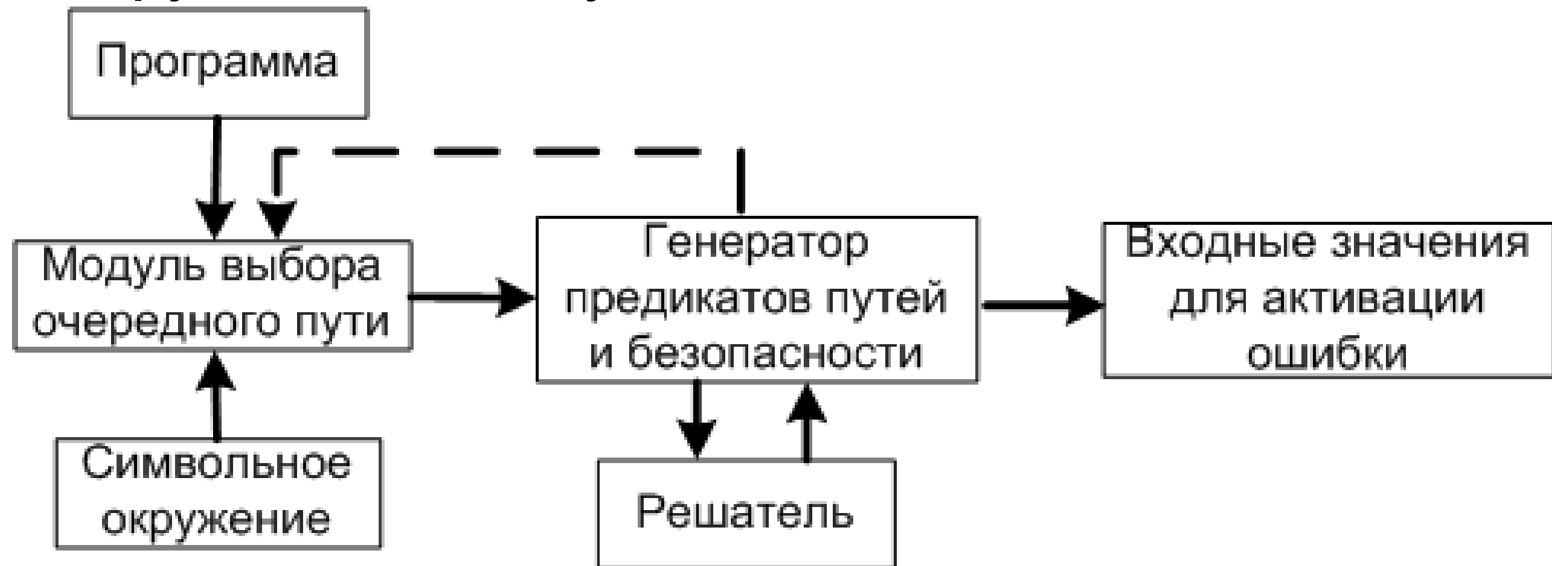
Предикат пути - набор логических формул описывающие прохождение по данному пути выполнения

Уравнение разрешимо?
x=70, input = “AAAAA…”
42 символа

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

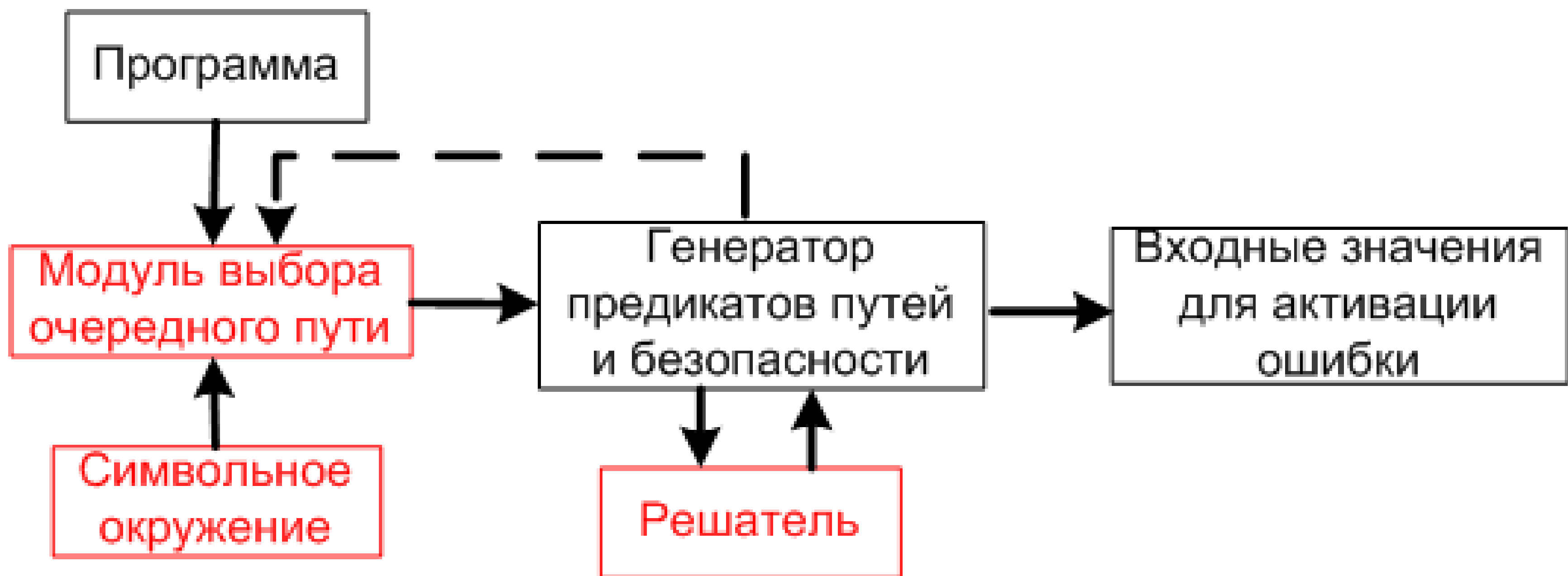
Общая схема системы поиска ошибок и уязвимостей

- 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



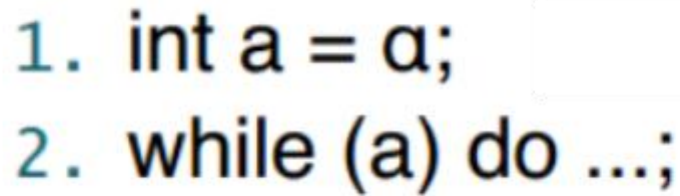
Экспоненциальный взрыв

- Экспоненциальный рост по точкам ветвления. Пример – 3 переменных, 3 точки ветвления, 8 путей

```
1. int a = α, b = β, c = γ;
2. if (a) ... else ...;
3. if (b) ... else ...;
4. if (c) ... else ...;
```

- символьные переменные

- Циклы, содержащие в условиях символьные переменные. Пример – потенциально 2 ^ 32 - 1 путей через цикл



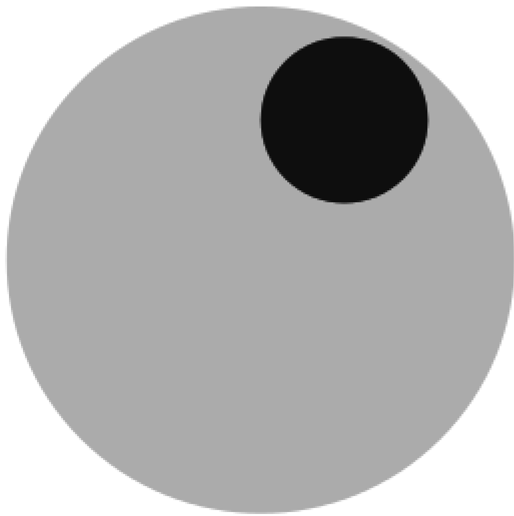
- Важность выбора стратегии обхода

Стратегии выбора пути

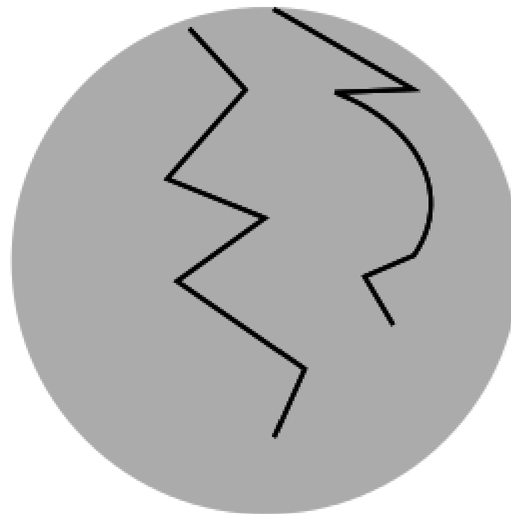
- На основе только структуры кода:
  - Обход в глубину – может «зациклится» в цикле
  - Обход в ширину – очень медленно доходит до содержательного кода при наличии большого количества ветвей
- На основе покрытия кода – выбирать пути с непосещенными инструкциями, или те которые посещались меньше. Позволяет 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)

Стратегии выбора пути (2)

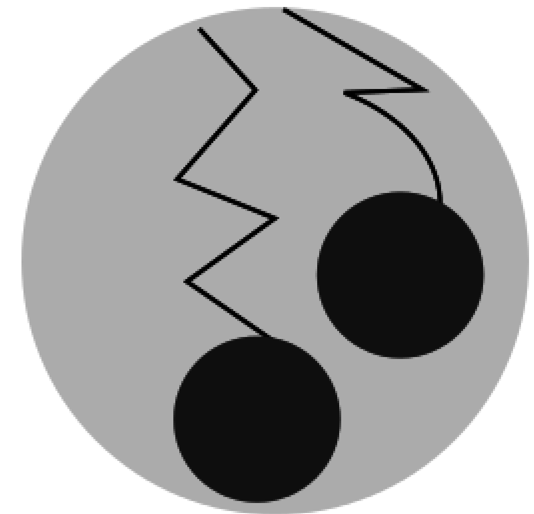
- Выполнение программы можно рассматривать как направленный ациклический граф
- Вершины – состояния программы
- Рёбра – переходы между состояниями
- Задача выбора пути – задача обхода графа



Обход на основе покрытия кода



Случайный обход



Гибридный подход

Моделирование окружения

- Библиотечные вызовы. Например, sin(x)
- Системные вызовы. Например, open(file)
- Возможный подход:

```
int fd  = open("t.txt", O_RDONLY);
```

Если параметры конкретные – передача ОС

```
int fd  = open(sym_str, O_RDONLY);
```

Иначе – предоставить модель, обрабатывающую символьные файлы

- Инициализация программы символьной файловой системой, поддерживающей N символьных файлов. При открытии N + 1-го – ошибка
- Цель – исследование всех возможных корректных взаимодействий с окружением

Ограничения решателя

Большая часть времени – решение уравнений:

- Дорогая процедура (NP-полный алгоритм)
- Много уравнений – вызов на каждой ветви

Две простые оптимизации (используются в KLEE):

- Удаление нерелевантных ограничений
- Кэширование решений

Удаление нерелевантных ограничений

На практике каждая ветвь зависит от малого числа переменных:

```
…
…
if (x < 10) {
    …
}
```

→

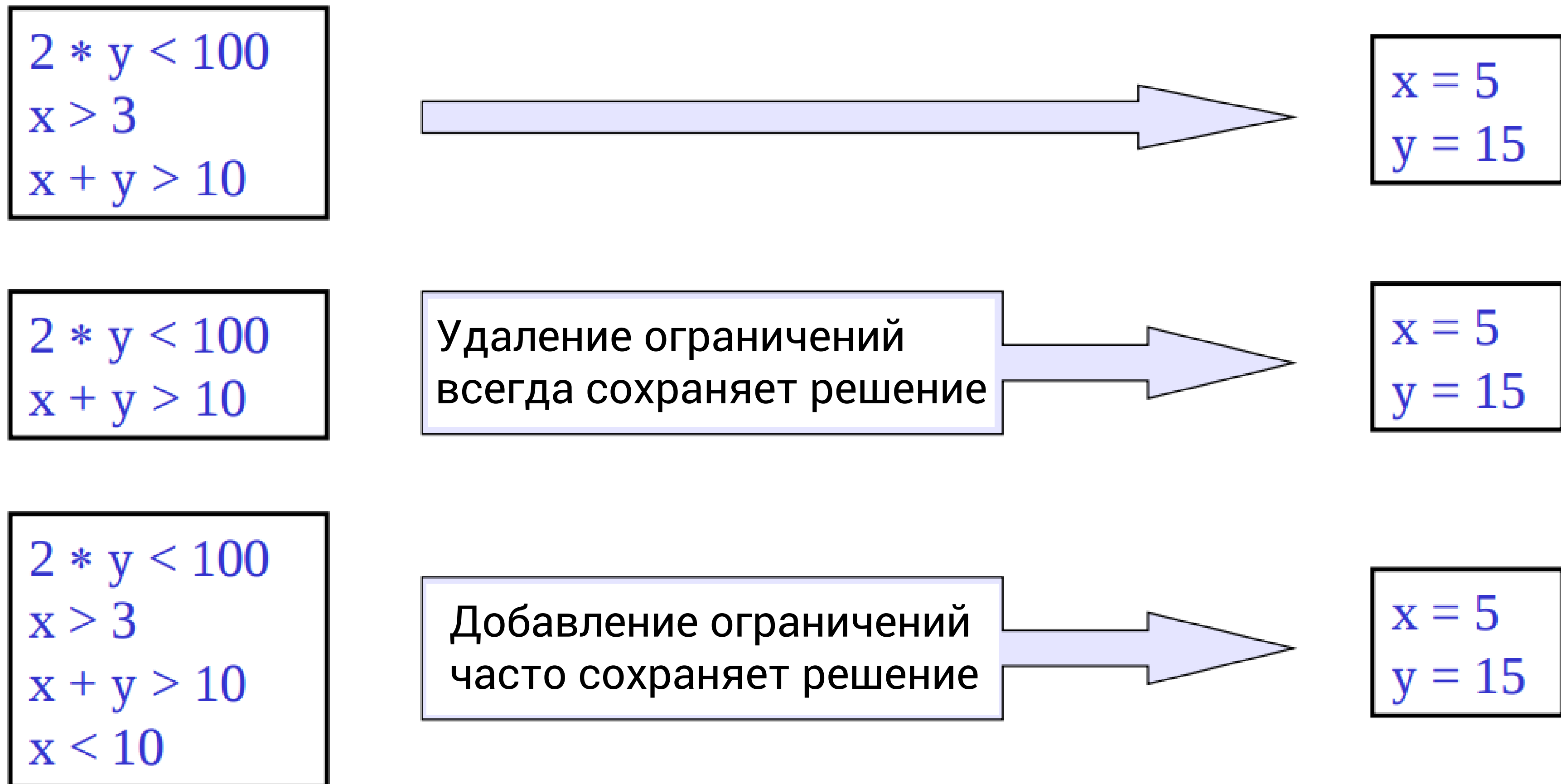
$x + y > 10$

~~$z \& -z = z$~~

$x < 10$?

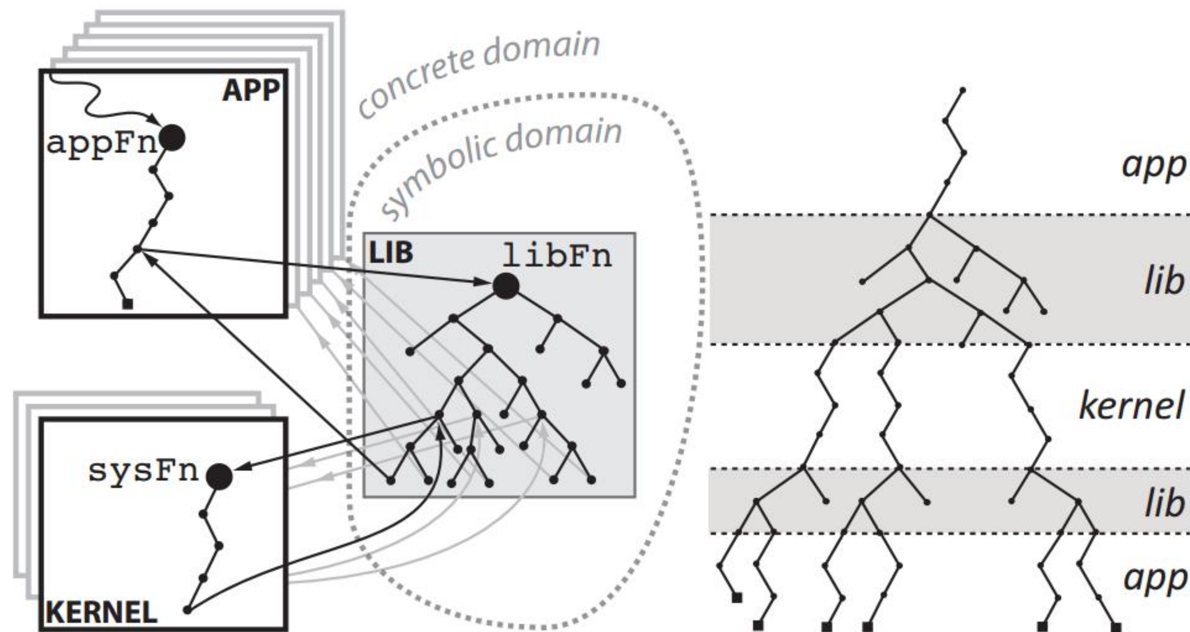
Кэширование решений

Статичное множество ветвей: частое повторение множества ограничений



Развитие подхода. Смешанное выполнение.

Используется в инструменте S2E



Используется в инструменте S2E

- Переход «символьное-конкретное»
- Переход «конкретное-символьное»
- Проблема «чрезмерного ограничения»

Основная литература

1. Амини П., Саттон М., Грин А. Fuzzing: исследование уязвимостей методом грубой силы. — Символ-Плюс, 2009.
2. Richard McNally, Ken Yiu, Duncan Grove and Damien Garhardy. Fuzzing: The state of the art, 2012.
3. Edward J. Schwartz, Thanassis Avgerinos, David Brumley. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but might have been afraid to ask, 2010

Дополнительная литература

1. Patrice Godefroid, Michael Y. Levin, David Molnar. Automated Whitebox Fuzz Testing, 2008.
2. S. Gorbunov and A. Rosenbloom. AutoFuzz: Automated Network Protocol Fuzzing Framework, 2010
3. V. Chipounov, V. Kuznetsov, G. Candea. S2E: A Platform for In-Vivo Multi-Path Analysis of Software Systems, 2011
4. C. Cadar, D. Dunbar, D. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs, 2008