Верификация программного обеспечения. Текущее состояние и проблемы

ИВАННИКОВ Виктор Петрович,
Институт системного программирования РАН (ИСП РАН)
ivan@ispras.ru
http://ispras.ru/

16 апреля 2012 года

Сложность современного ПО

Система	Год	Размер (10^6 LOC)	Размер команды
Windows 3.1	1992	3	
Windows NT 3.5	1994	10	300
Windows NT 4.0	1996	16	800
Windows 2000	1999	30	1400
Windows XP	2001	45	1800
Linux Kernel 2.6.0	2003	5,7	
Open Solaris	2005	9,7	
Mac OS X 10.4	2006	86	
Windows Vista	2007	> 50	
Вся авионика США (бортовая и наземная)	2007	~1000	
Дистрибутив Debian 5.0	2009	323	

Статистика ошибок

- Количество ошибок на 1000 строк кода (до тестирования) остается практически неизменным за 30 лет

Программные системы	**Число ошибок на 1000 строк кода**
Microsoft (до тестирования)	10-20
NASA JPL (до тестирования)	6-9
Среднее по индустрии (продукты)	6-30
Linux	~7
Microsoft (продукты)	0.5
NASA JPL (продукты)	0.003

Верификация и информационная безопасность программ

- Наличие ошибки в программе часто означает наличие уязвимости
- Технологии верификации одновременно являются средствами оценки и обеспечения информационной безопасности программ
- Вывод:
  повышение информационной безопасности программ без развития технологий верификации невозможно

Технологии верификации

- Экспертиза
- Тестирование (динамические испытания)
- Аналитическая верификация
- Статический анализ
- Комбинированные подходы

Экспертиза

- Поиск ошибок, оценка и анализ свойств ПО человеком (обычно группа 2-5 человек)
- Техники
  - Списки важных ограничений и шаблонов ошибок
  - Групповые обсуждения
- Производительность : 100-150 строк / час
  - Не более 2-3 часов в день
- Результаты : выявляется 50-90% ошибок (обнаруживаемых за все время жизни ПО)
- Нужны настоящие эксперты, программисты с менее чем 10-летним стажем могут участвовать лишь как обучающиеся

Тестирование

- Оценка корректности системы по ее работе в выбранных ситуациях, с определенными данными
- Техники:
  - Шаблоны сценариев, перебор и фильтрация
  - Синтез тестов по структуре реализации или модели
- Производительность : сильно зависит от целей, техники, опыта и пр.
  - Microsoft : производительность разработки тестов и тестирования примерно такая же, как у создателей кода, 10 строк/час
- Результаты : выявляется 30-80% ошибок
- Часто используется неопытный персонал, что отрицательно сказывается на качестве

Пример : проект DMS (ИСП РАН)

- Разработка тестов для ОС телефонного коммутатора Nortel Networks с использованием формальных методов
- Размер системы : 250 000 строк, 230 интерфейсных функций
- Трудоемкость : ~ 10 человеко-лет
- Результаты:
  - Тестовый набор использовался для проверки всех новых версий ОС
  - > 300 ошибок в системе с заявленной надежностью 99.9999%
  - 12 ошибок, требующих холодного рестарта
  - ~ 50% всех ошибок найдены в ходе экспертизы
  - Остальные выявлены тестами, сгенерированными из формальных спецификаций и нацеленными на их покрытие

Показатели качества тестирования

- Метрики тестового покрытия
  - Функциональности программы и требований
  - Структуры кода: строк, ветвлений, комбинаций условий в ветвлениях
- Для адекватной оценки нужно сочетание нескольких метрик
- Примеры достигаемого тестового покрытия :
  - Обычное коммерческое ПО : 15-20% ветвлений в коде
  - Системное ПО : 60-80% ветвлений в коде
  - Тесты на базе формальных моделей : 70-80% ветвлений в коде
  - Аналитическая верификация : 100% ветвлений в коде

Аналитическая верификация

- Формальное описание семантики программы и требований и доказательство выполнения требований
- Техники:
  - Метод Флойда (дедуктивный метод)
  - Provers, solvers, интерактивные инструменты
- Инструменты:
  - Мировые лидеры: PVS, Isabelle/HOL, Frama-C
- Примеры приложений: микропроцессоры, ядро операционной системы seL4
- Опыт ИСП РАН: практикум по верификации с помощью Frama-C (~ 100 человеко-часов на программу размером порядка 100 строк)

Пример: верификация ядра ОС seL4

- Ядро встроенной ОС seL4
  - 8700 строк на C, 600 строк на ассемблере
  - Для реализации ~2.5 человеко-года
- Верификация
  - 200 000 строк формальной модели
  - Инструмент: Isabelle/HOL
  - Трудоемкость – 20 человеко-лет (12 чел.)
    Доказано ~10000 лемм

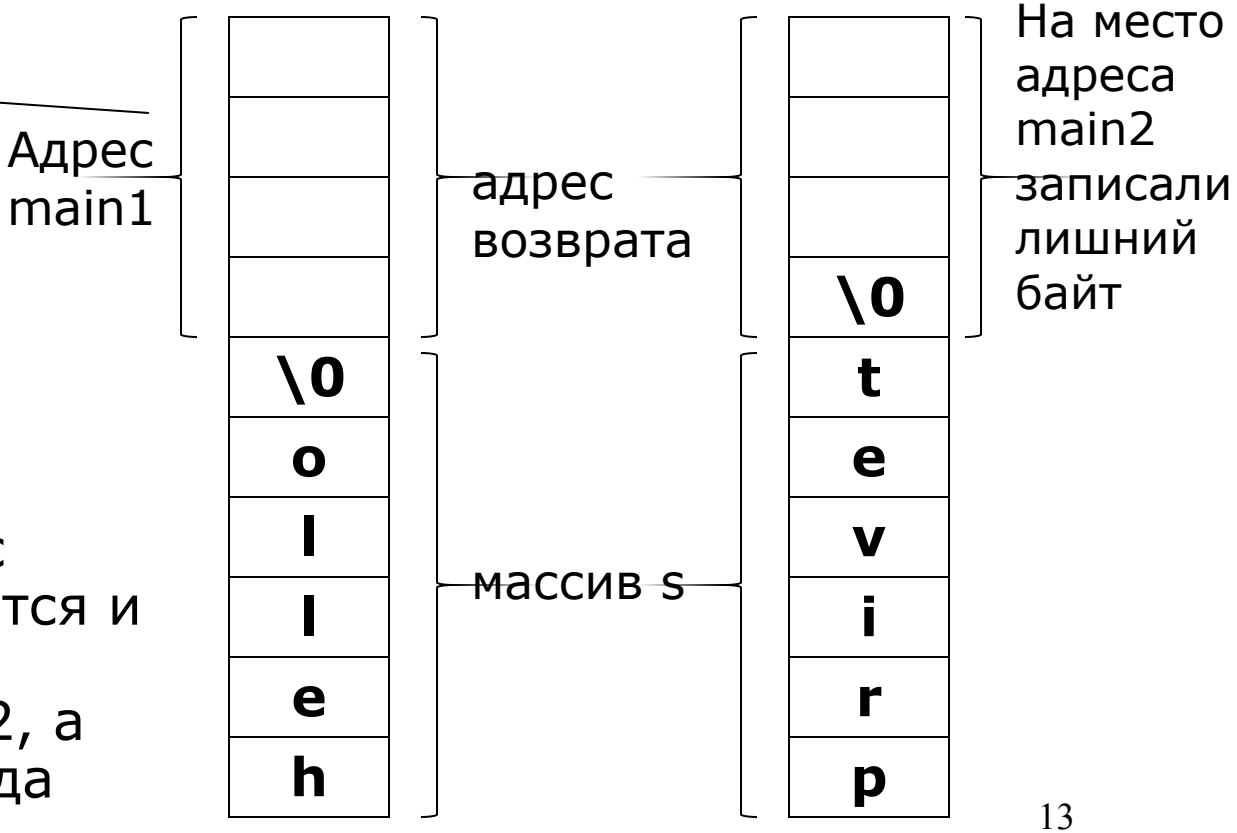
Статический анализ программ

Простейший пример

```
f(char * p)
{     char s[6];
      strcpy(s,p);
}
main1 ()
{       f("hello");
}
main2 ()
{       f("privet");
}
```

Стек после выполнения функции f, вызванной из main1

Стек после выполнения функции f, вызванной из main2



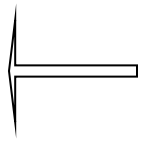
В случае main2 адрес возврата перезапишется и управление будет передано не на main2, а на другой участок кода

Альтернативы решения

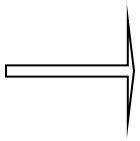
- Тестирование
- Проверки времени выполнения
- Статический анализ программы
- Смешанный

Динамические проверки

```
f(char * p)
{    char s[10];
     strncpy(s,p,10
     );
}
```



```
f(char * p)
{    char
     s[10];
     strcpy(s,p)
     ;
}
```



```
f(char * p)
{    char s[10];
     if
     (strlen(p)<10)
        strcpy(s,p);
}
```

- Можно использовать специальные функции с дополнительными параметрами – ограничениями сверху на объем записываемой информации. Например, вместо strcpy(a,b) использовать strncpy(a,b,n), где n-максимальное количество переписываемых символов
- Вставка проверок подразумевает инструментацию исходного кода и так как такие проверки должны присутствовать во **всех** потенциально опасных местах, **инструментированный код может работать на порядок медленнее исходного**

От динамической защиты к статическому обнаружению

- Даже в случае минимизации количества проверок инструментированный код работает гораздо медленнее исходного
- Среди вставляемых проверок много «бесполезных», то есть тех, которые проверяют заведомо истинные условия
- Необходимо статически обнаружить в тексте программы только те места, в которых действительно возможно нарушение системы защиты

Цели статического анализа – выявление дефектов в программах

Дефекты (ситуации в исходном коде) могут приводить к:

- Уязвимостям* защиты
- Потере стабильности работы программы

*Уязвимость защиты (security vulnerability) – ошибка в тексте программы, которая позволяет пользователю при некоторых сценариях использования программы обходить средства разграничения прав доступа программы или ОС, в которой программа выполняется.

Рассматриваемые виды дефектов

- Переполнение буфера
- Format string: недостаточный контроль параметров при использовании функций семейства printf/scanf
- Tainted input: некорректное использование непроверяемых на корректность пользовательских данных
- Разыменование нулевого указателя
- Утечки памяти
- Использование неинициализированных данных

Преимущества статического анализа

- Автоматический анализ многих путей исполнения одновременно
- Обнаружение дефектов, проявляющихся только на редких путях исполнения, или на необычных входных данных (которые могут быть установлены злоумышленником в процессе атаки)
- Возможность анализа на неполном наборе исходных файлов
- Отсутствие накладных расходов во время выполнения программы

Первое поколение анализаторов

- Flowfinder, ITS4, RATS, Pscan (распространяются бесплатно)
- CodeSurfer (инструмент для обнаружения уязвимостей на базе CodeSurfer’а недоступен)
- FlexeLint (продается на рынке), Splint (распространяется бесплатно)

Недостатки первого поколения

- Большое число ложных срабатываний – 90% и выше
- Пропуск реальных уязвимостей
- Необходима ручная проверка результатов работы, которая требует привлечения значительных ресурсов (материальных, людских, временных)

Современные анализаторы

- Coverity Prevent
- Klockwork Insight
- Svace

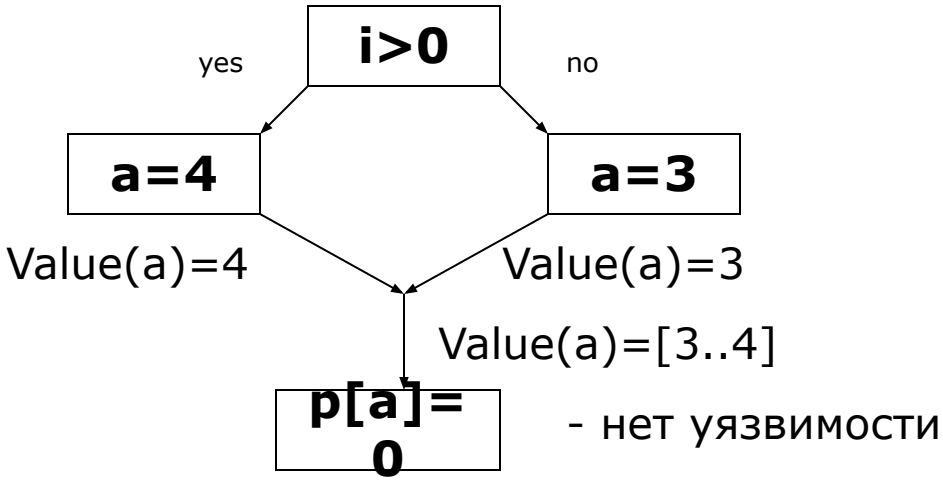
Наш подход

Для обнаружения уязвимостей мы предлагаем применять следующее:

- Межпроцедурный data-flow анализ с итерациями на внутрипроцедурном уровне
- Анализ указателей
- Анализ интервалов и значений целочисленных объектов
- Контекстно-зависимый (использующий индивидуальные входные параметры для каждой точки вызова) анализ

Целочисленный анализ на основе интервалов

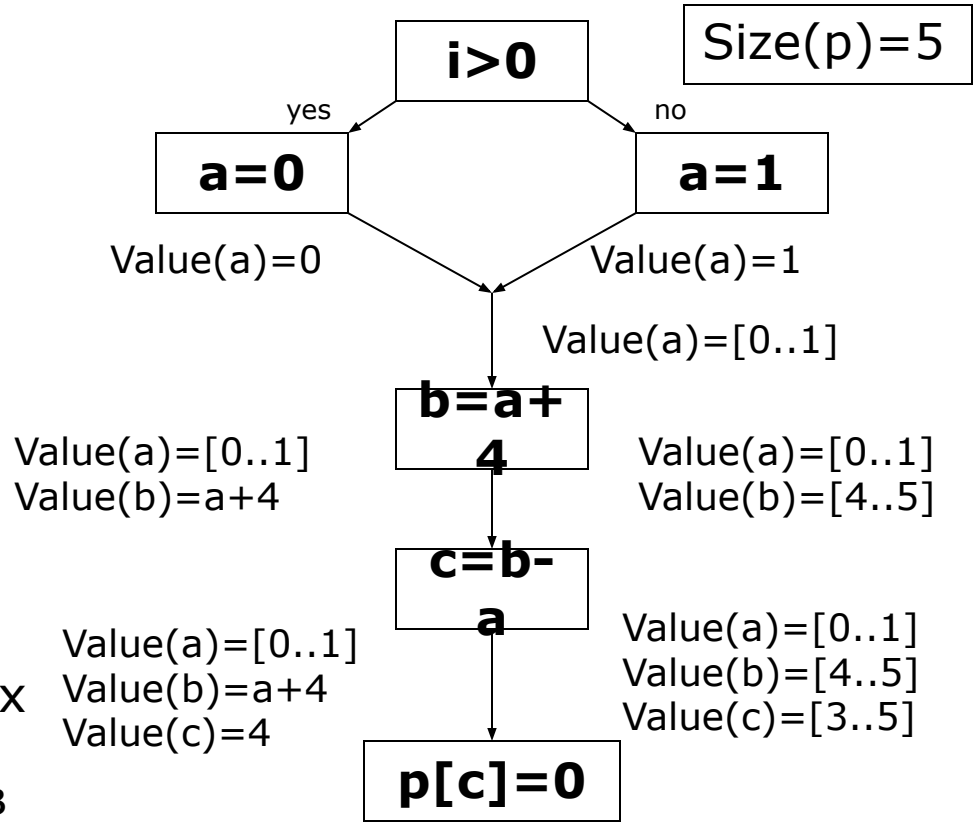
```
void f(int i)
{    char p[5];
     int a;
     if (i>0)
          a=4;
     else
          a=3;
     p[a]=0;
}
```



- Целью целочисленного анализа программы является получение необходимой информации о значении целочисленных атрибутов объектов программы (значение переменной, размер массива и т.д.)
- Чем точнее и полнее представляются целочисленные значения, тем точнее возможно проведение поиска уязвимостей

Недостатки целочисленного анализа интервалов

```
void f(int i)
{     char p[5];
      int a,b,c;
      if (i>0)
            a=0;
      else
            a=1;
      b=a+4;
      c=b-a;
      p[c]=0
}
```



- Потеря точности в некоторых случаях.
- Отсутствие информации о связях между переменными: в случае анализа на основе интервалов в последней инструкции примера будет обнаружена ложная уязвимость.

Необходимость межпроцедурного анализа

```
f(char *p, char *s)
{   strcpy(p,s);  // произойдет копирование 6 байт, включая        //конец
    строки
}

main()
{   char *p;
    p=malloc(5);
    f(p,”hello”);
}
```

В приведенном примере указатель на объект размером 5 байт передается в функцию f, поэтому без межпроцедурного анализа эта информация никаким образом не попадет на вход вызову функции strcpy и ошибку не зафиксируется

Необходимость анализа указателей

```
…
char a[10];
char * p;
p=a;
p[10]=0;
…
```

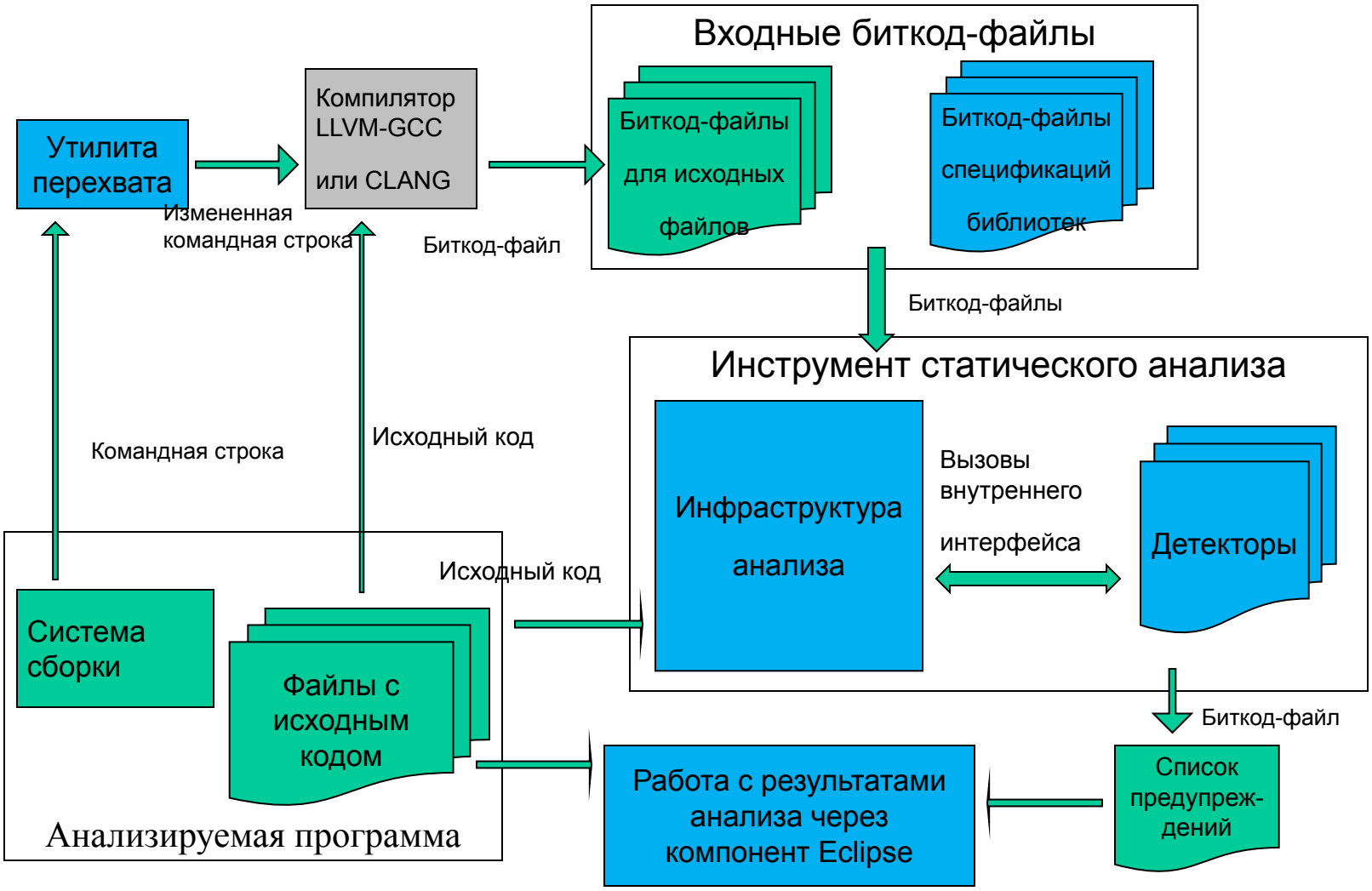
Для данного примера в случае отсутствия анализа указателей невозможно сделать никаких предположений о размере массива, на который указывает p и, следовательно, невозможно определить наличие ошибки

Спецификация окружения

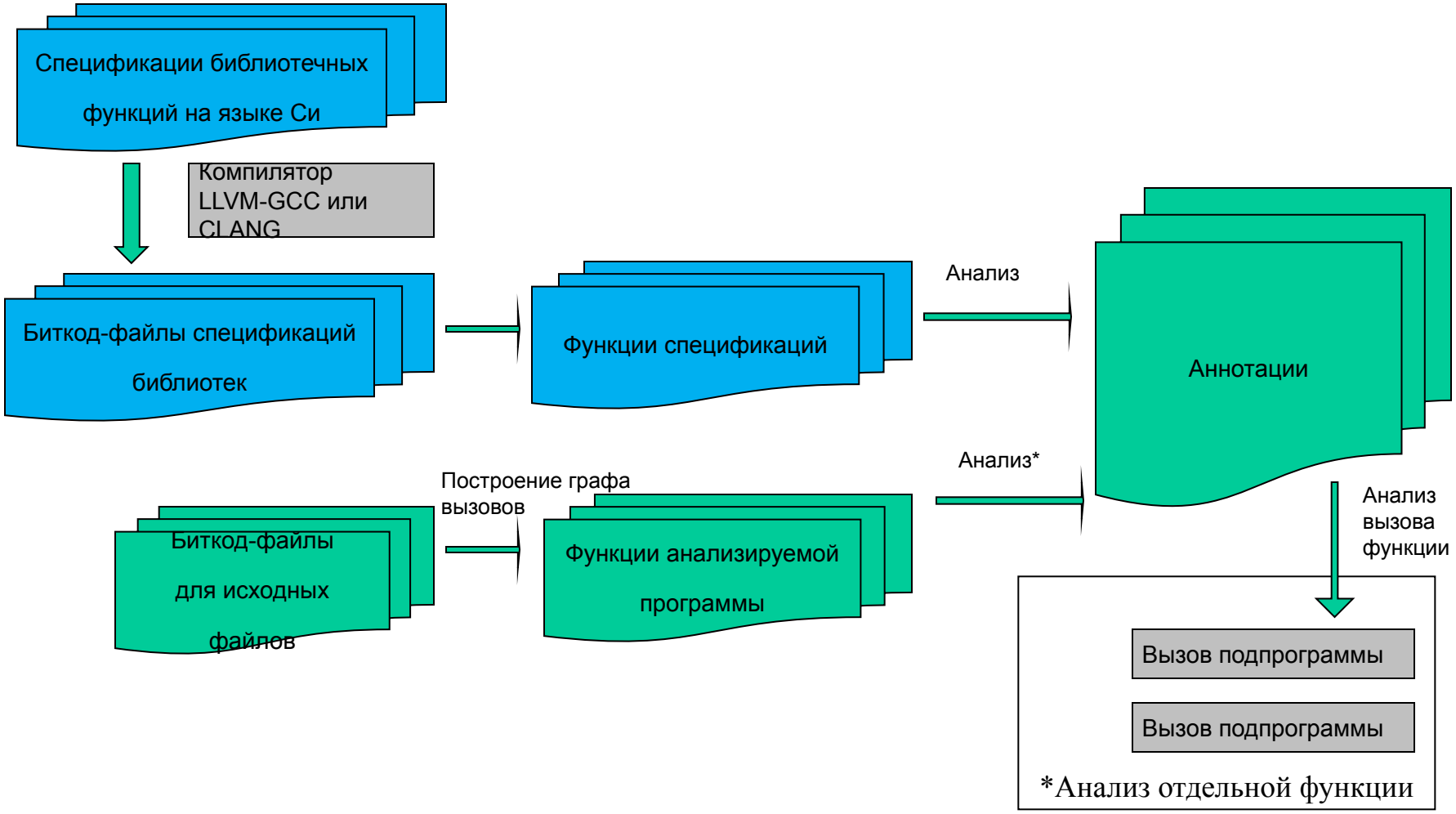
```
char* strcpy(char *dst, const char *src) {
    char d1 = *dst;//dst и src
    char d2 = *src;// разыменовываются в функции

    //необходимо проверить на корректность src
    //(в случае пользовательского ввода)
    sf_set_trusted_sink_ptr(src);

    //содержимое src копируется в dst
    sf_copy_string(dst, src);

    //возвращается входной параметр
    return dst;
}
```

Внутренние обработчики инфраструктуры анализа

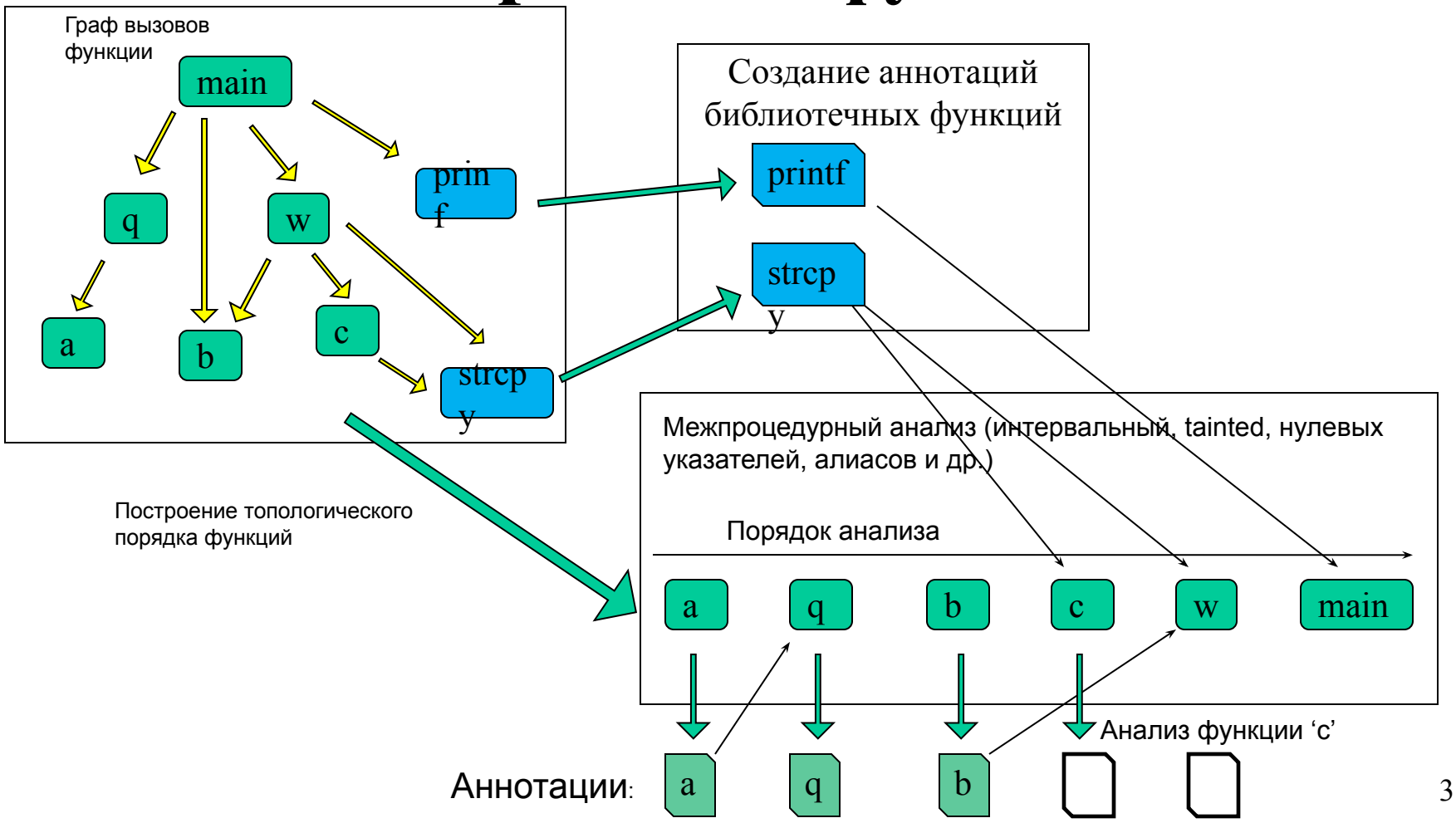
Схема работы Svace



Построение аннотаций



Топологический порядок обработки функций



Время анализа

Проект	Строк Си и Си++ кода	Время анализа*	Строк в секунду
libxml++	5245	39 с	134
gperf	6516	33 с	197
aide	13945	58 с	240
ffmpeg	75521	1 м 20 с	944
gstreamer.0.10	101954	2 м 43 с	625
qtiplot	158771	5 м 07 с	517
openssl	240264	19 м 51 с	202
postgresql-8.4	503815	23 м 38 с	355
firefox	2862047	2ч 35 м 40 с	306
android	11 млн.	13 ч 58 м	218

* Не учитывается время компиляции проекта

Сравнение с Coverity Prevent

Тип предупреждения	Истинные срабатывания Svace	Истинные срабатывания Prevent	Воспроизведено истинных срабатываний Prevent, %
Переполнение буфера	60%	10%	100%
Работа с динамически выделенной памятью	50%	70%	20%
Разыменование NULL	70%	60%	50%
Испорченный ввод	70%	70%	80%
Неинициализированные данные	60%	40%	50%
Несоотвестие типов возвращаемых значений	60%	90%	30%
Состояние гонки	90%	90%	80%
Передача по значению	100%	100%	100%
Другие (более 30)	50%	70%	30%

Текущее состояние

- Анализ программ на Си\Си++.
- Информация об исходном коде собирается при помощи компилятора LLVM-GCC (или CLANG)
- Нет ограничений на размер программы (линейная масштабируемость)
- Полностью автоматический анализ
- Поддержка пользовательских спецификаций функций
- Набор спецификаций стандартных библиотечных функций (Си, Linux)
- Набор подсистем поиска различных дефектов (переполнение буфера, разыменование нулевого указателя и др.)
- Набор подсистем поиска дефектов расширяем
- Графический пользовательский интерфейс, реализованный в виде расширения среды Eclipse

Avalanche: Обнаружение ошибок при помощи динамического анализа

Динамический и статический анализ кода

- Динамический анализ - анализ программы во время выполнения
- Статический анализ - анализ без выполнения программы

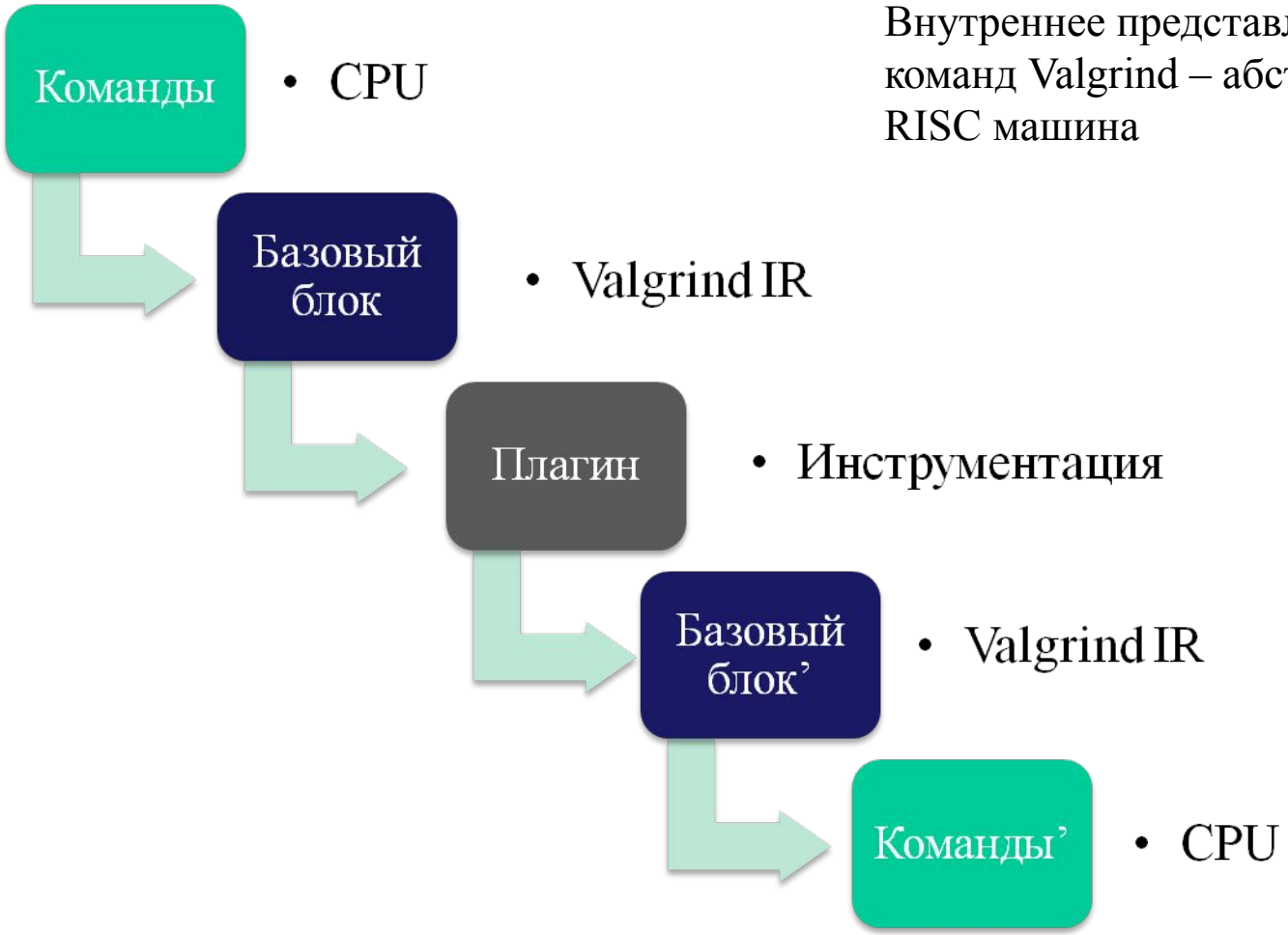
Обнаружение ошибок при помощи анализа программ

- Динамический анализ
  - Требуется набор входных данных и/или среда выполнения
  - Высокие требования к ресурсам
  - Высокая точность обнаружения
- Статический анализ
  - Работает на исходном или бинарном коде
  - Анализ абстрактной модели
  - Хорошая масштабируемость
  - Ложные срабатывания

Valgrind

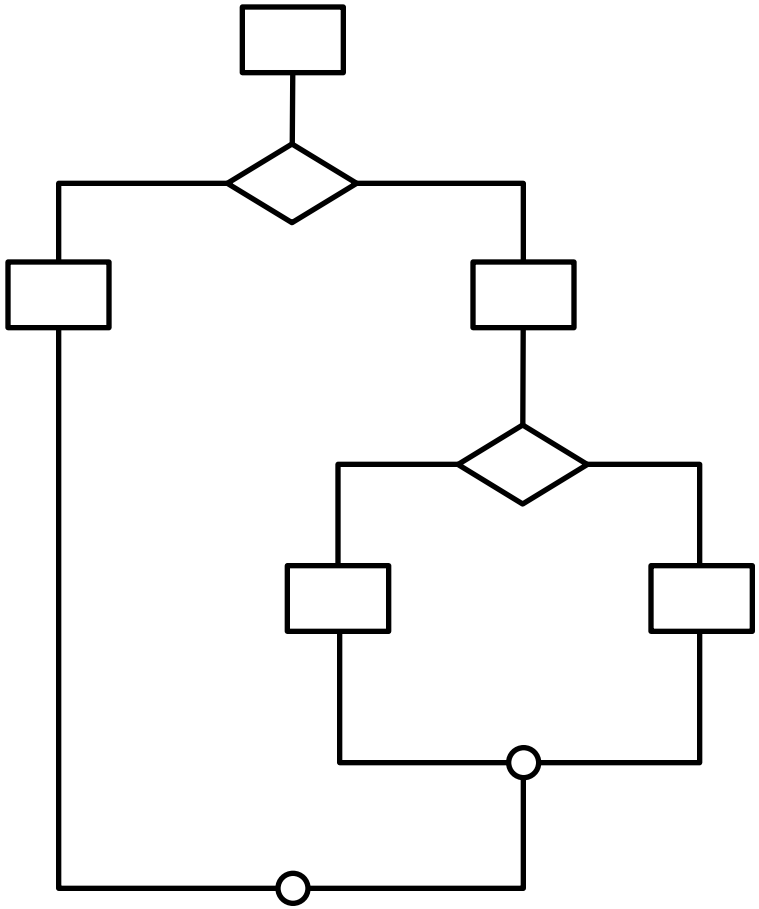
- Фреймворк динамической инструментации
- Обнаруживаемые ошибки:
  - Утечки памяти
  - Ошибки работы с динамической памятью
  - Неинициализированные данные
  - Ошибки в многопоточных программах

Valgrind: общая схема работы

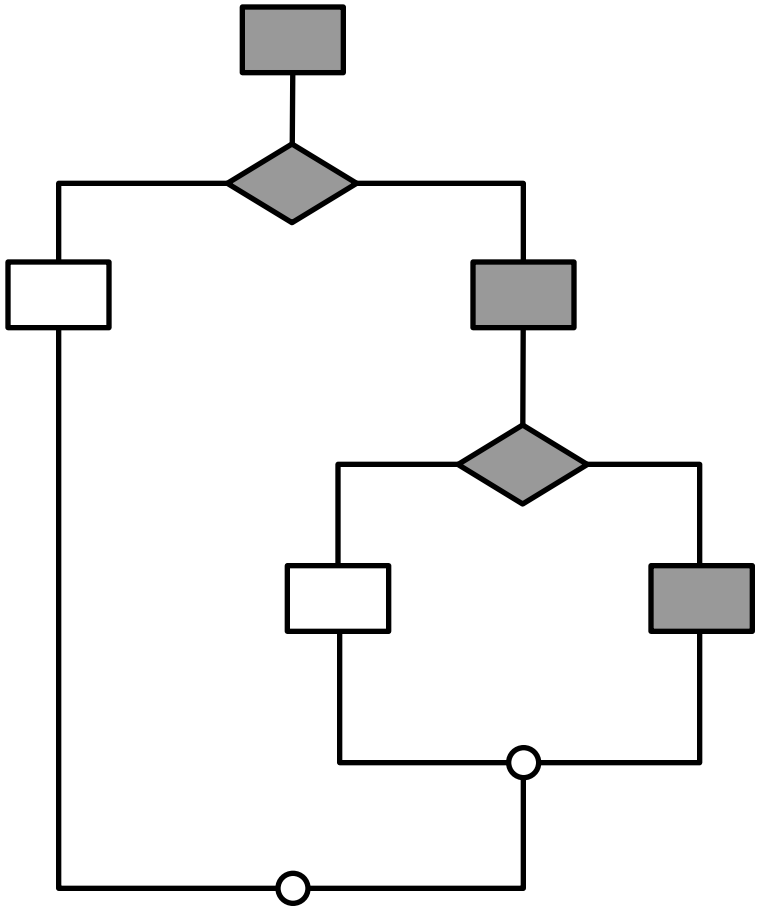


Внутреннее представление команд Valgrind – абстрактная RISC машина

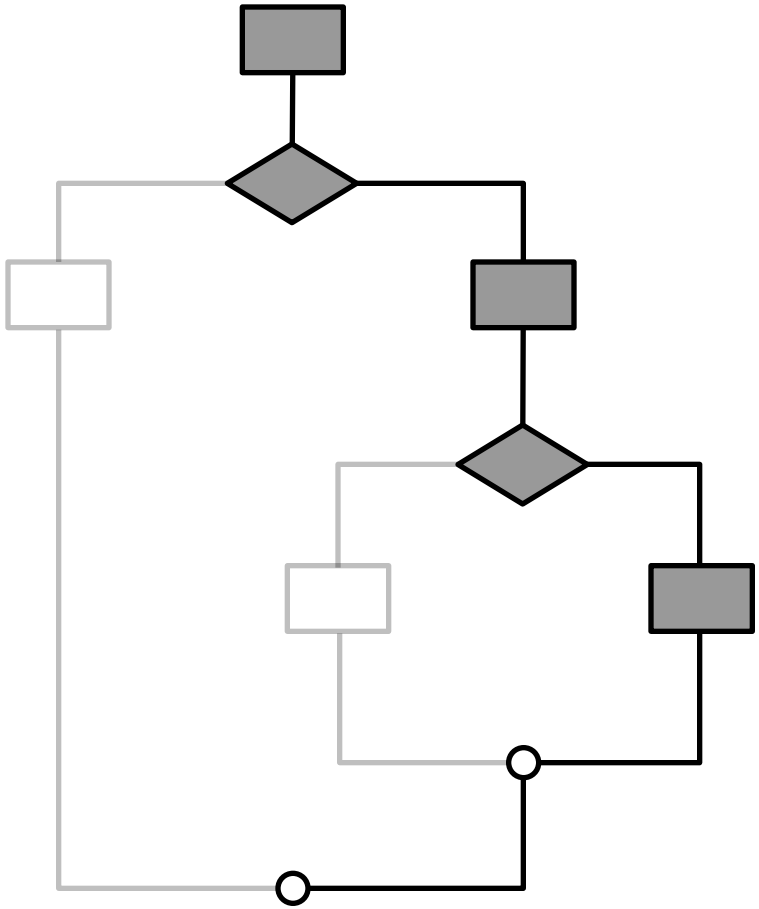
Трасса выполнения программы



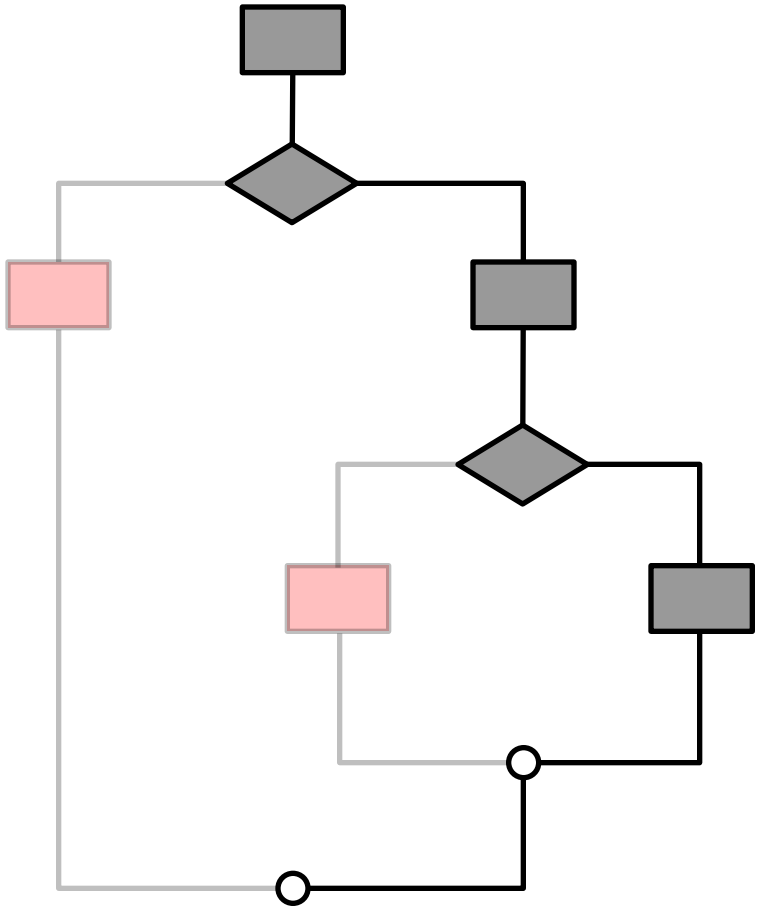
Трасса выполнения программы



Трасса выполнения программы



Трасса выполнения программы



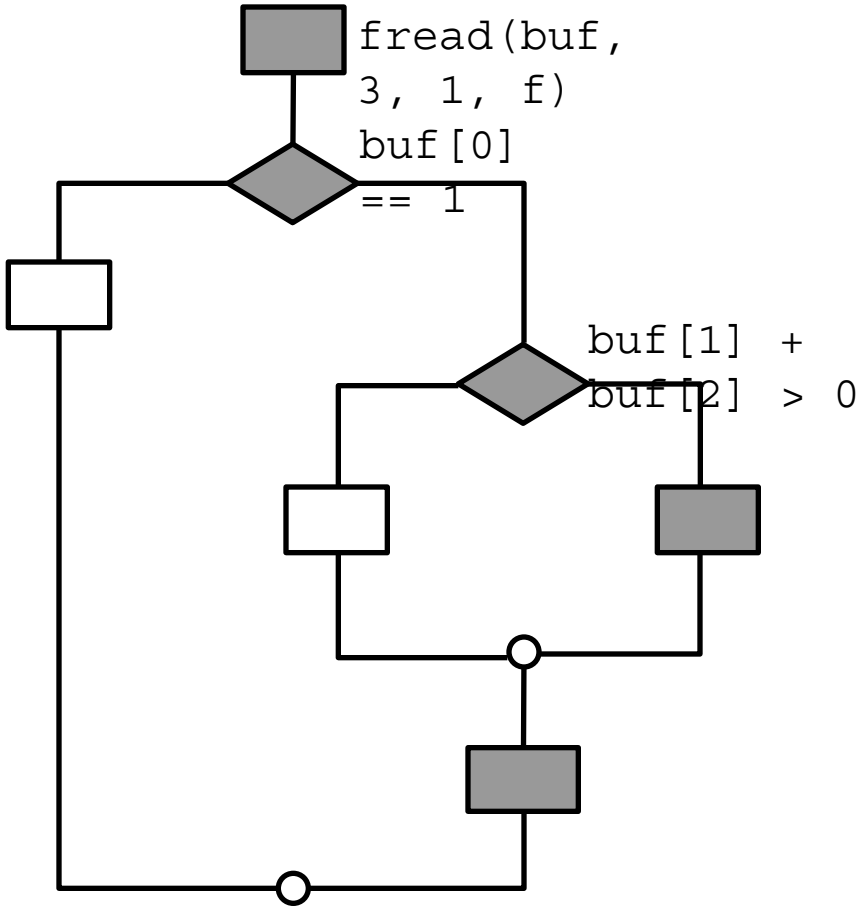
Работы в этой области

- EXE tool, Stanford University - символические вычисления
- SAGE framework, Microsoft Research - white-box fuzz testing
- KLEE, LLVM project

Пример

```
char *names[] = { “one”, “two”, ...};

char buf[3];
fread(buf, 3, 1, f);    // чтение 3-х байт из файла

if (buf[0] == 1) {    // ветвление #1
     int index;
     if (buf[1] + buf[2] > 0) {    // ветвление #2
          index = ...;
     }
   name = names[index];  // index не инициализирован
                         // выход за границы массива
     ...
} else {
     ...
}
```

Пример

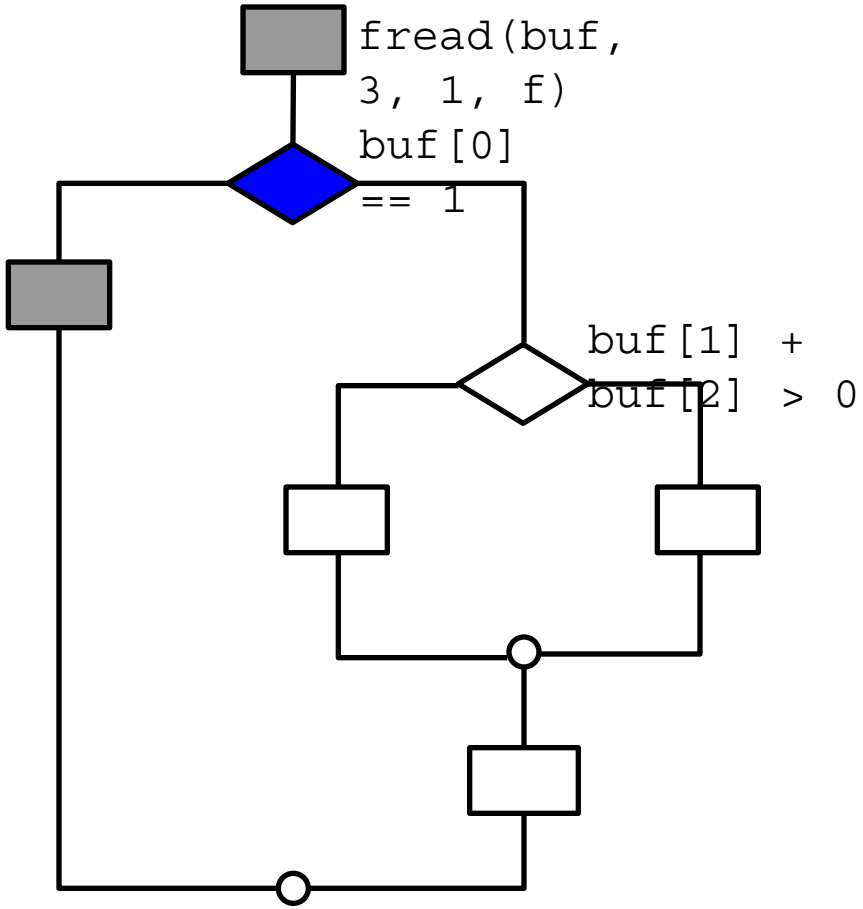


x1, x2, x3: byte

x1 = 1
x2 + x3 > 0

x1 = 1
x2 = 100
x2 = 200

Пример



Система уравнений:

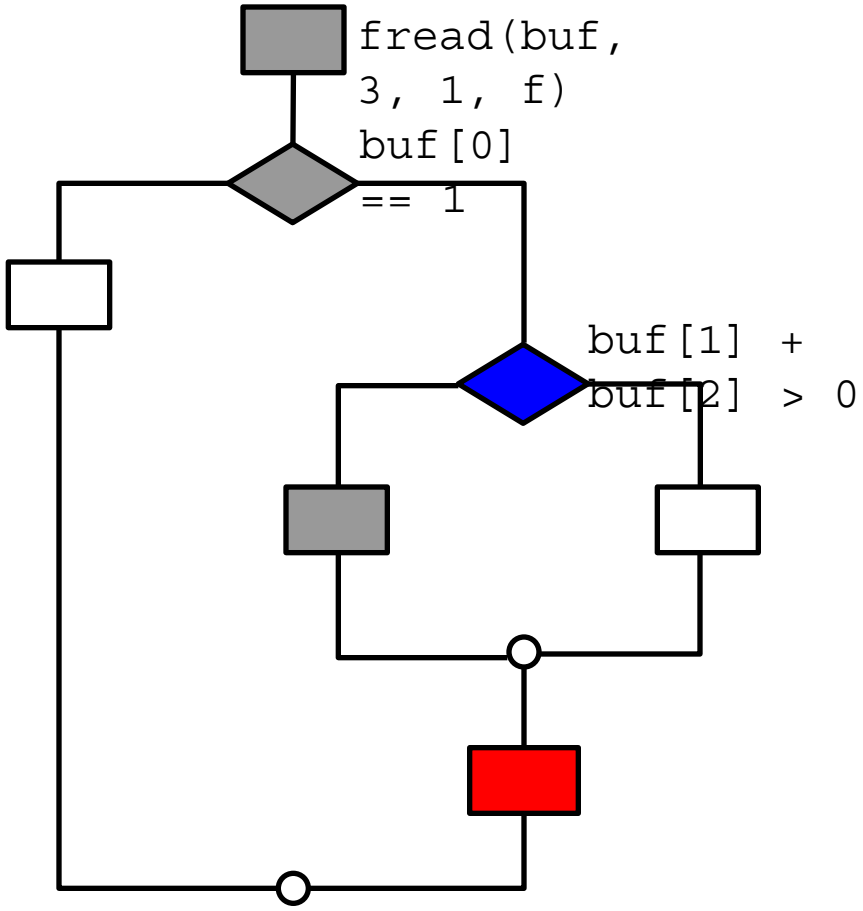
x1, x2, x3: byte

¬(x1 = 1)

Решение:

x1 = 2

Пример



Система уравнений:

x1, x2, x3: byte

x1 = 1

¬(x2 + x3 > 0)

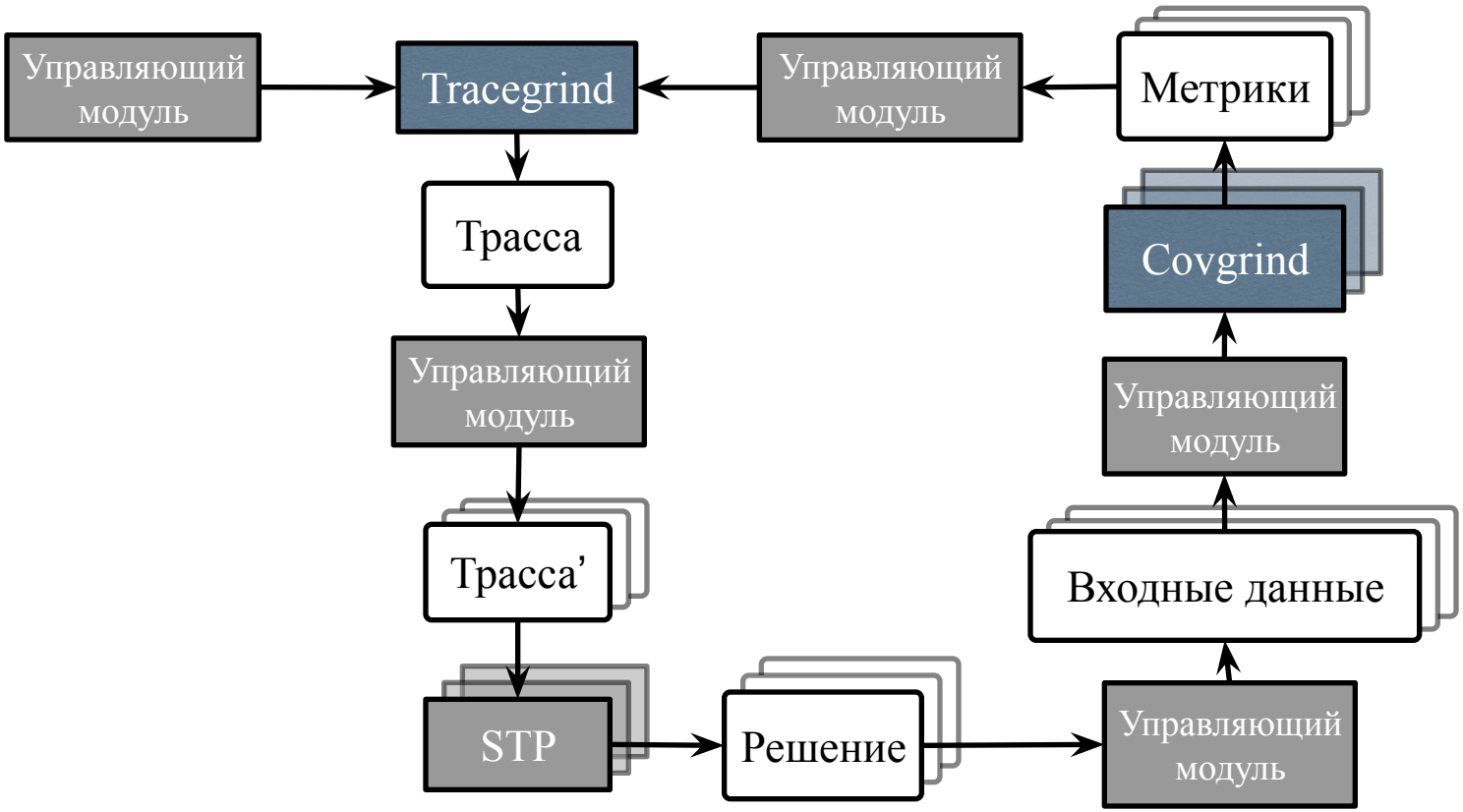
Решение:

x1 = 1
x2 = 0
x3 = 0

Avalanche

- Отслеживает поток помеченных (потенциально опасных) данных
- Изменяет входные данные, чтобы спровоцировать ошибку, или обойти новые части программы
- Обнаруживает критические ошибки - разыменование нулевого указателя, деление на ноль, неинициализированные данные, ошибки работы с памятью
- Генерирует набор входных данных для каждой найденной ошибки

Avalanche: итерация



Управляющий модуль Avalanche

- Координация работы других компонентов
- Обход различных путей исполнения программы, инвертирование условий
- Поддержка параллельного и распределенного анализа

Tracegrind

- Отслеживает поток помеченных данных в программе
- Все данные прочитанные из внешних источников (файлы, сетевые сокеты, аргументы коммандной строки, переменные окружения)
- Переводит трассу выполнения в булевскую формулу (STP утверждения)

Tracegrind

- Моделирует оперативную память, регистры и временные переменные при помощи бит-векторов
- Моделирует комманды при помощи операций и утверждений STP

Covgrind

- Вычисление метрики покрытия кода программы (количество новых ББ покрытых на текущей итерации)
- Перехват сигналов (обнаружение критических ошибок)
- Обнаружение ошибок работы с памятью при помощи Memcheck
- Обнаружение бесконечных циклов при помощи таймаутов

STP - Simple Theorem Prover

- SAT решатель (основан на MiniSat)
- Проект с открытым исходным кодом
- Поддерживает бит-векторы, широкий набор операций

Проект

- Опубликован на Google Code http://code.google.com/p/avalanche
- Лицензии:
  - Valgrind и Memcheck - GPL v2
  - STP - MIT license
  - Драйвер Avalanche, Tracegrind, Covgrind - Apache license

Avalanche: возможности

- Поддержка клиентских сетевых сокетов
- Поддержка переменных окружения и параметров коммандной строки
- Поддержка платформ x86/Linux и amd64/Linux, ARM/Linux, Android
- Поддержка параллельного и распределенного анализа

Результаты

- Более 15-ти ошибок на проектах с открытым исходным кодом
- Ошибки подтверждены и/или исправлены разработчиками

mencoder	NPD
wget	NPD
swtool	NPD
libmpeg2	DBZ
gnash	UE
audiofile	IL
libsndfile	DBZ
libjpeg	DBZ
libmpeg3	NPD
libquicktime	NPD, IL
libwmf	NPD
mono	NPD
parrot	NPD, IL
llvm	NPD

Null Pointer Dereference (разуменование нулевого указателя) = NPD
Division By Zero (деление на ноль) = DBZ
Unhandled Exception (необработанная исключительная ситуация) = UE
Infinite Loop (бесконечный цикл) = IL

Планы на будущее

- Улучшение прозводительности
- Поддержка новых источников входных данных (серверные сетевые сокеты, и т. д.)
- Поддержка новых типов ошибок (многопоточные приложения)

UniTESK

Разработка теста по-простому

- Случай тестирования отдельной функции:
  - Подобрать набор входных (тестовых) данных
  - Вычислить ожидаемый результат для каждого из тестовых данных
    А если это трудно или невозможно, например для функции random() ?
  - Запустить тест с каждым из тестовых данных, сопоставить результат с ожидаемым
  - Принять решение о продолжении или завершении тестирования
    А каков критерий завершения ?

Разработка теста по-простому (2)

- Случай тестирования группы функции, класса/объекта, модуля с несколькими интерфейсами (обычно есть переменные состояния и побочный эффект):
  - Подобрать набор входных (тестовых) данных для каждой функции
  - Вычислить ожидаемый результат для каждого из тестовых данных
    Новая проблема – как учесть побочный эффект?
  - Вызвать каждую функцию с каждым из тестовых данных в различных состояниях модуля, сопоставить результат с ожидаемым
    Какие состояния считать различными, как прийти в нужное состояние (построить тестовую последовательность), если побочный эффект вычислить невозможно?
  - Принять решение о продолжении или завершении тестирования
    А каков критерий завершения ?

Типичные размеры тестовых наборов

- Ядро ОС Linux (LTP)
  - 18 Mbyte (при этом покрывает менее половины строк кода)
- Библиотеки стандарта OS Linux (LSB)
  - Более 100 тысяч вариантов
  - Более 80 Mbyte
- Для компилятора C (например, ACE или Perennial)
  - Более 40-80 тысяч вариантов
  - Более 1 Gbyte

Не удивительно, что на тестирование тратиться существенная доля усилий (в Майкрософте около 70%), а средняя плотность ошибок, которая считается приемлемой – 2-3 ошибки на 1 тысячу строк кода.

Решения UniTESK

- Исходная точка построения теста – формализация программного контракта в форме пред- и пост-условий
  - пред- и пост-условия определяют тестовые оракулы и критерии полноты покрытия
- Тестовую последовательность конструировать не вручную, а на основе интерпретации модели теста
- Нотации – максимально приближенные к языкам программирования

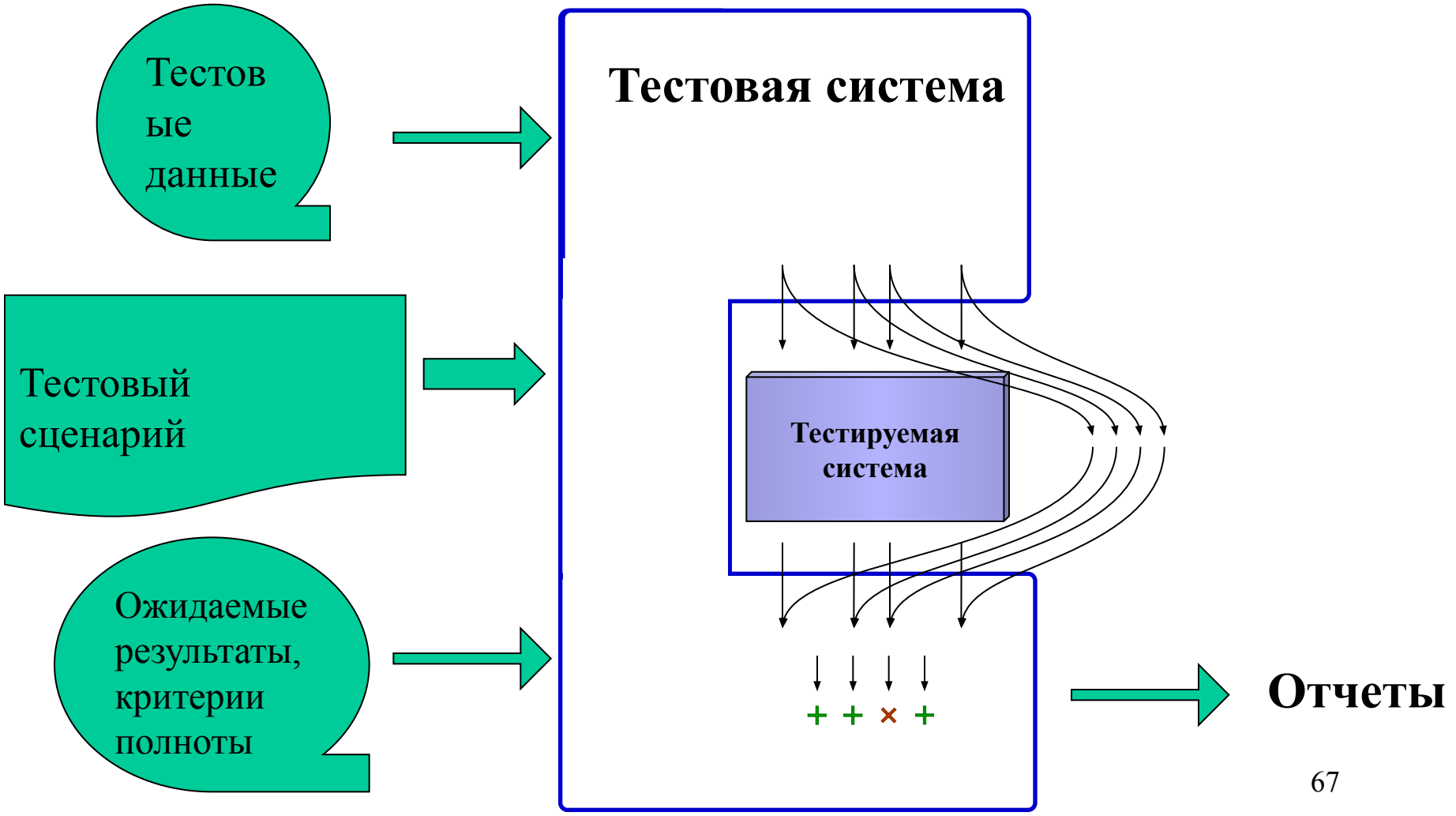
Формализация требований

- Выделение модельного состояния
- Описание формального контракта операций
  - Предусловия – задают область определения
  - Постусловия – задают основные ограничения на результаты работы операций
  - Инварианты – ограничения целостности данных, общая часть всех пред- и постусловий
  - Аксиоматическая часть контракта (если нужно)

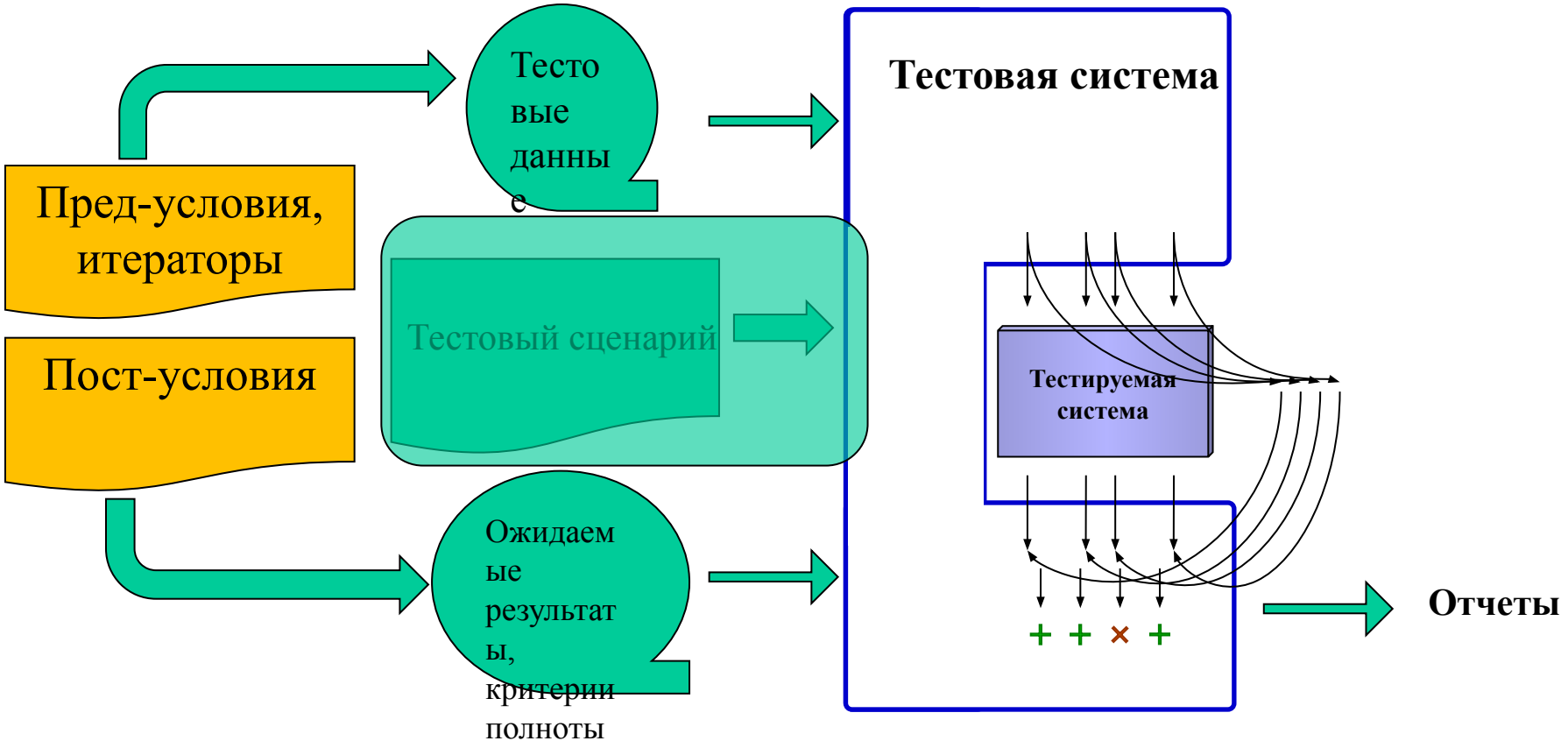
Простой пример спецификации в JavaTESK

```
public specification class SqrtSpecification {
  public specification double sqrt(double x) {
    pre { return x >= 0; }
    post  {
      branch SingleCase;
      return     sqrt * sqrt == x;
    }

  public specification int sqrt(int x)  {
    pre { return x >= 0; }
    post  {
      branch SingleCase;
      return     sqrt * sqrt <= x
              && (sqrt + 1) * (sqrt + 1 ) > x ;
    }
  }
}
```

Общая схема тестирования



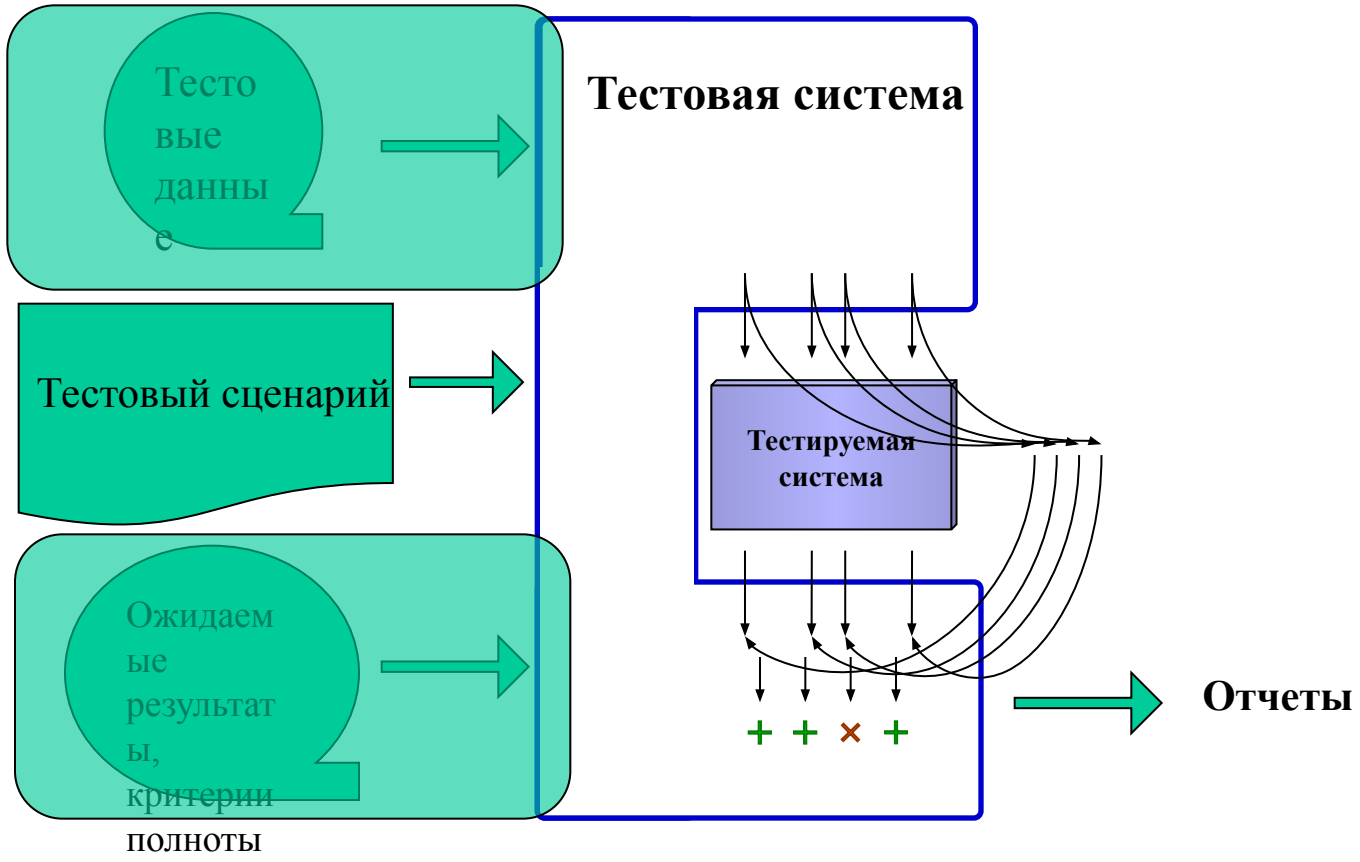
Общая схема. Данные, оракул, покрытие



Общая схема. Тестовый сценарий

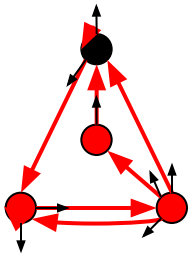
Варианты:

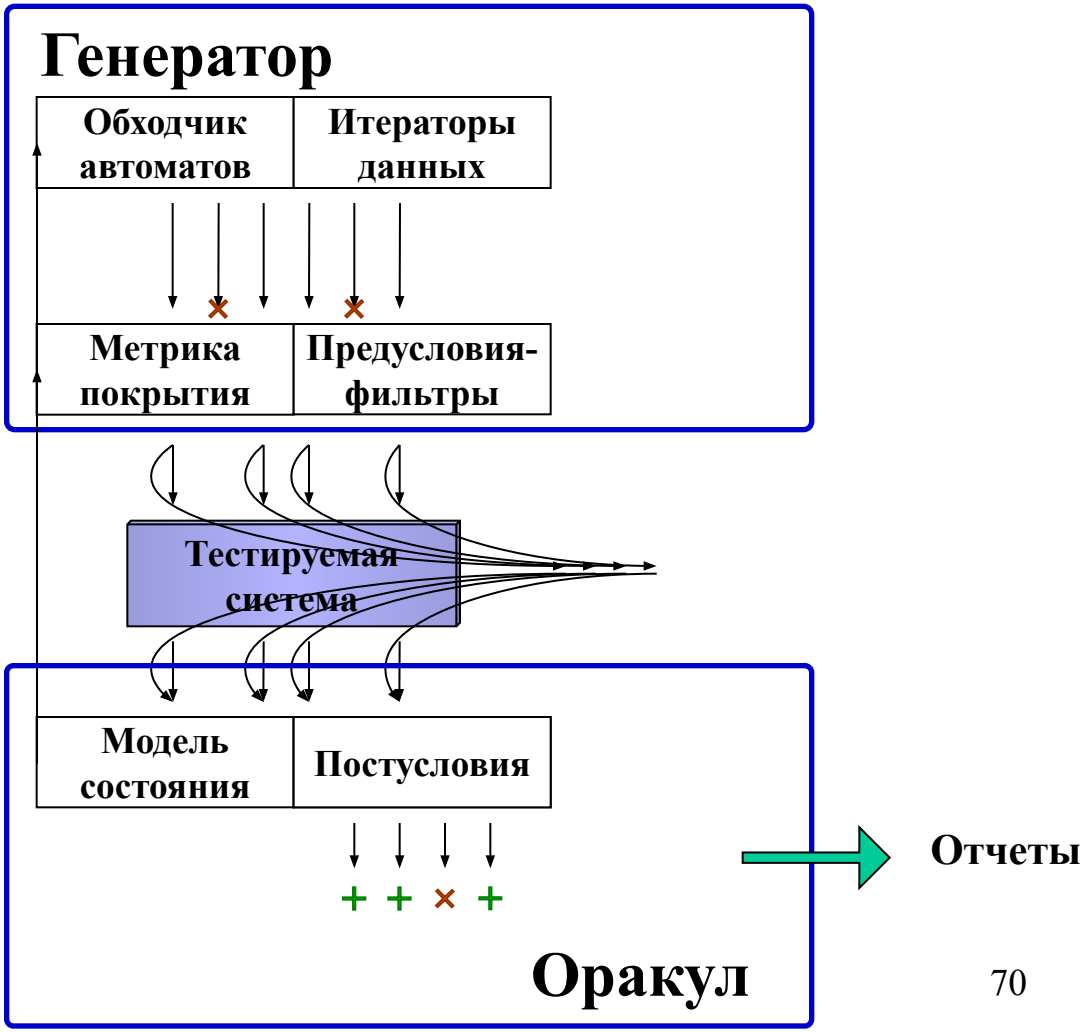
- написать вручную
- описать и обойти КА
- обходить и строить КА налету



Общая схема UniTESK

Генерация тестовой последовательности налету





Применение UniTESK

- Операционные системы
  - Ядро ОС телефонной станции 1994-1997
  - Linux Standard Base 2005-2010
  - Тестовый набор для ОС 2000 (НИИСИ) 2005-...
- Протоколы
  - IPv6 Microsoft Research 2000-2001
  - Мобильный IPv6 (в Windows CE 4.1) 2002-2003
  - IPv6 Октет 2002
  - Тестовый набор для IPsec 2004-2008
  - Тестовый набор для SMTP 2010
- Оптимизаторы компиляторов Intel 2001-2003
- Оптимизатор трансляции графических моделей 2005
- Информационные системы
  - Компоненты CRM-системы 2004
  - Биллинговая система и EAI 2005-…
- Микропроцессоры
  - Процессоры Комдив 64 (НИИСИ) 2006-...
  - Компоненты процессоров и шин (МЦСТ) 2010-...

Спасибо!