# FUZZING-ТЕСТИРОВАНИЕ: ИЩЕМ БАГИ В JIT-КОМПИЛЯТОРЕ И НЕ ТОЛЬКО

Макс Казанцев
max.kazantsev@azul.com

# Немного о нас

- Zing VM – виртуальная машина языка Java от Azul Systems
- LLVM – открытый фреймворк для построения компиляторов
  - *clang*
  - *Xcode*
  - *Kotlin Native*
  - *И другие*
- Falcon – JIT-компилятор в Zing VM, основанный на LLVM

# Что вернёт данная функция?

```
char foo1(char c) {
    for (int i = 0; i < 193; i++) {
        c &= (char) 1;
    }
    return c;
}
```

| | |
|---|---|
| A) 0 или 1 | B) 1 или 2 |
| C) 2 или 3 | D) 3 или 4 |

# Что вернёт данная функция?

```
char foo2(char c) {
    for (int i = 0; i < 193; i++) {
        c &= (char) 1;
        c += (char) 3;
    }
    return c;
}
```

clang v3.8-5.0, -O2

| | |
|---|---|
| A) 0 или 1 | B) 1 или 2 |
| C) 2 или 3 | D) 3 или 4 |

https://godbolt.org/g/kG6qWA

# Естественная реакция на такие баги

# Какую проблему решаем?

- Есть продукт (Falcon), который очень быстро развивается
- Нужно быстро находить регрессии, когда они возникают
- Нужно находить существующие баги раньше всех
- Для этого нужны хорошие тесты

# Виды багов, которые мы ловим

- Падения во время исполнения
- Бесконечное/неприемлемо долгое время работы
- Некорректные результаты

# Почему баги в компиляторах сложно искать?

- Компилятор большой
  - *Объём кода >80 мегабайт*
- Код быстро изменяется
  - *Регулярные ребейзы на свежий LLVM*
  - *Более 70к строк изменений в неделю*
- Некоторые баги проявляются только при соблюдении особых условий
- Даже если компилятор сделал ошибку, она не обязательно будет наблюдаема
- Синергия оптимизаций порождает баги

# Сложности JIT-компиляторов

- Могут использовать данные времени исполнения
  - *Горячие участки кода*
  - *Информация о реальных классах объектов в виртуальных вызовах*
  - *Информация о загруженных классах и т.п.*
- Порядок компиляции/инлайнинга может отличаться от прогона к прогону
- Возможны спекулятивные оптимизации
- Методы перекомпилируются несколько раз

# Такое может случиться с каждым

- Баги есть не только в компиляторах
- Описанная методология подходит и для других продуктов
- Так что не переключайтесь!

# Чего хотят тестировщики?

- Максимум автоматизации
- Однозначно правильное или неправильное поведение
- Минимум ложных срабатываний
- Один инструмент для контроля за старыми и поиска новых багов
- Находить проблемы дома, а не узнавать о них от клиентов

# Чего хотят разработчики?

- Простые и понятные тесты (желательно с исходниками)
- 100% воспроизводимость проблем
- Разумное время исполнения теста
- Детерминированное поведение

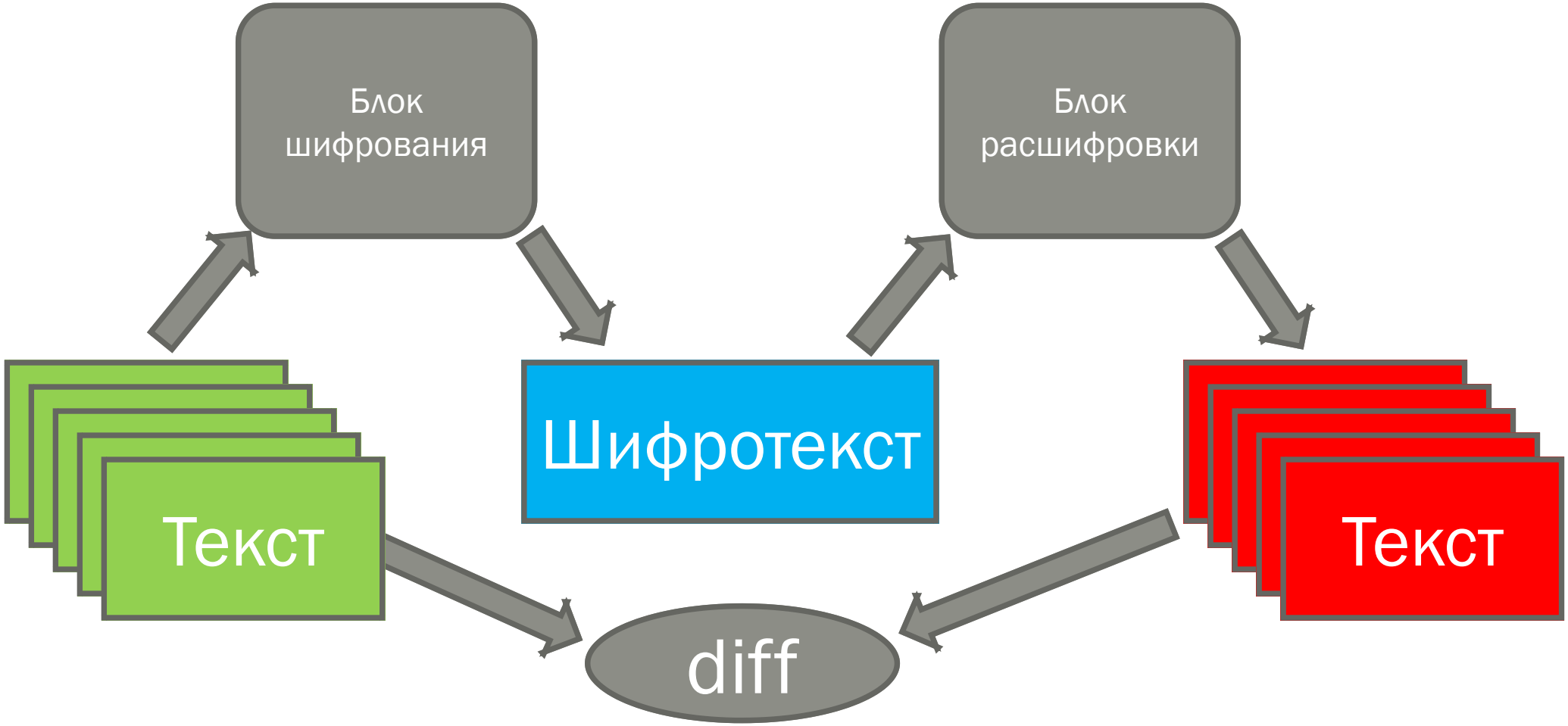
# Как ищут баги в компиляторах?

- Наборы небольших статических тестов (regression, unit и т.п.)
- Реальные приложения
- Coverage-тестирование
- Fuzzing-тестирование

# Fuzzing-тестирование

- От англ. Fuzzy – нечёткий, неопределённый, пушистый
- Состоит в генерации и подаче на вход случайных данных
- Широко применяется там, где требуется высокая степень надёжности

# Как можно тестировать шифрование?



# А оно взлетит?

- Пусть «хороший» тест находит баг с вероятностью $10^{-6}$
- Тогда этот тест не находит баг с вероятностью $0{,}999999$
- Но что, если у нас 5 миллионов тестов?
- Ни одного бага не найдётся с вероятностью $0{,}999999^{5000000} \approx 0{,}007$
- То есть, они найдут хотя бы один баг с вероятностью $> 99\%$

# Существующие фаззеры

- libFuzzer https://llvm.org/docs/LibFuzzer.html
- American Fuzzy Lop http://lcamtuf.coredump.cx/afl/
- Mull Mutation https://github.com/mull-project/mull
- jittester (OpenJDK project)
- Java Fuzzer for Android https://github.com/android-art-intel/Fuzzer
- Java Fuzzer https://github.com/AzulSystems/JavaFuzzer

# Стратегии фаззинга

- Мутация – создавать новые тесты на основе существующих
- Генерация – создавать тесты «с нуля»
- Сочетание этих подходов

# Пример мутирующего фаззинга

| | |
|---|---|
| `Hello, world!` | |
| `Helwo, world!` | Изменение символа |
| `Hel wo, world!` | Вставка символа |
| `Hel wo, word!` | Удаление символа |
| `Hel wo, wordword!` | Дублирование слова |
| `Hel wo, wordword!Hel wo, wordword!` | Дублирование текста |

# Мутирующий фаззинг программ

```
int main() {
    return 0;
}
```

```
int main() {
    int a = 0;
    return 0;
}
```

# Мутирующий фаззинг программ

```
int main() {
    int a = 0;
    return 0;
}
```

```
int main() {
    int a = 1;
    return 0;
}
```

# Мутирующий фаззинг программ

```
int main() {
    int a = 1;
    return 0;
}
```

```
int main() {
    int a = 1;
    return a;
}
```

# Мутирующий фаззинг программ

```
int main() {
    int a = 1;
    return a;
}
```

```
int main() {
    int b = 15;
    int a = 1;
    return a;
}
```

# Мутирующий фаззинг программ

```
int main() {
    int b = 15;
    int a = 1;
    return a;
}
```

```
int main() {
    int b = 15;
    int a = b + 1;
    return a;
}
```

# Мутирующий фаззинг программ

```
int main() {
    int b = 15;
    int a = b + 1;
    return a;
}
```

```
int main() {
    int b = 15;
    int a = b + 1;
    a++;
    return a;
}
```

# Мутирующий фаззинг программ

```
int main() {
    int b = 15;
    int a = b + 1;
    a++;
    return a;
}
```

```
int main() {
    int b = 15;
    int a = b + 1;
    for (int i = 0; i < N; i++) {
      a++;
    }
    return a;
}
```

# Мутирующий фаззинг программ

```
int main() {
    int b = 15;
    int a = b + 1;
    for (int i = 0; i < N; i++) {
      a++;
    }
    return a;
}
```

```
int main() {
    int b = 15;
    int a = b + 1;
    for (int i = 0; i < N; i++) {
      a++;
      b++;
    }
    return a;
}
```

# Мутирующий фаззинг программ

```
int main() {
    int b = 15;
    int a = b + 1;
    for (int i = 0; i < N; i++) {
      a++;
      b++;
    }
    return a;
}
```

```
int main() {
    int b = 15;
    int a = b + 1;
    for (int i = 0; i < N; i++) {
      a *= b;
      b++;
    }
    return a;
}
```

# Генерация программ

- <Function> → void main() { <Body> }
- <Body> → <Declare><Body> | <Assign><Body> | ε
- <Declare> → int <ID> = <ID>; | int <ID> = <Constant>;
- <Assign> → <ID> = <ID>; | <ID> = <Constant>;

# Генерация программ

```
<Function>
```

# Генерация программ

```
void main() {
      <Body>
}
```

# Генерация программ

```
void main() {
        <Declare>
        <Body>
}
```

# Генерация программ

```
void main() {
      <Declare>
      <Declare>
      <Body>
}
```

# Генерация программ

```
void main() {
      <Declare>
      <Declare>
      <Assign>
      <Body>
}
```

# Генерация программ

```
void main() {
      <Declare>
      <Declare>
      <Assign>
      <Declare>
      <Body>
}
```

# Генерация программ

```
void main() {
      <Declare>
      <Declare>
      <Assign>
      <Declare>
}
```

# Генерация программ

```
void main() {
	int a = 3;
	<Declare>
	<Assign>
	<Declare>
}
```

# Генерация программ

```
void main() {
	int a = 3;
	int b = a;
	<Assign>
	<Declare>
}
```

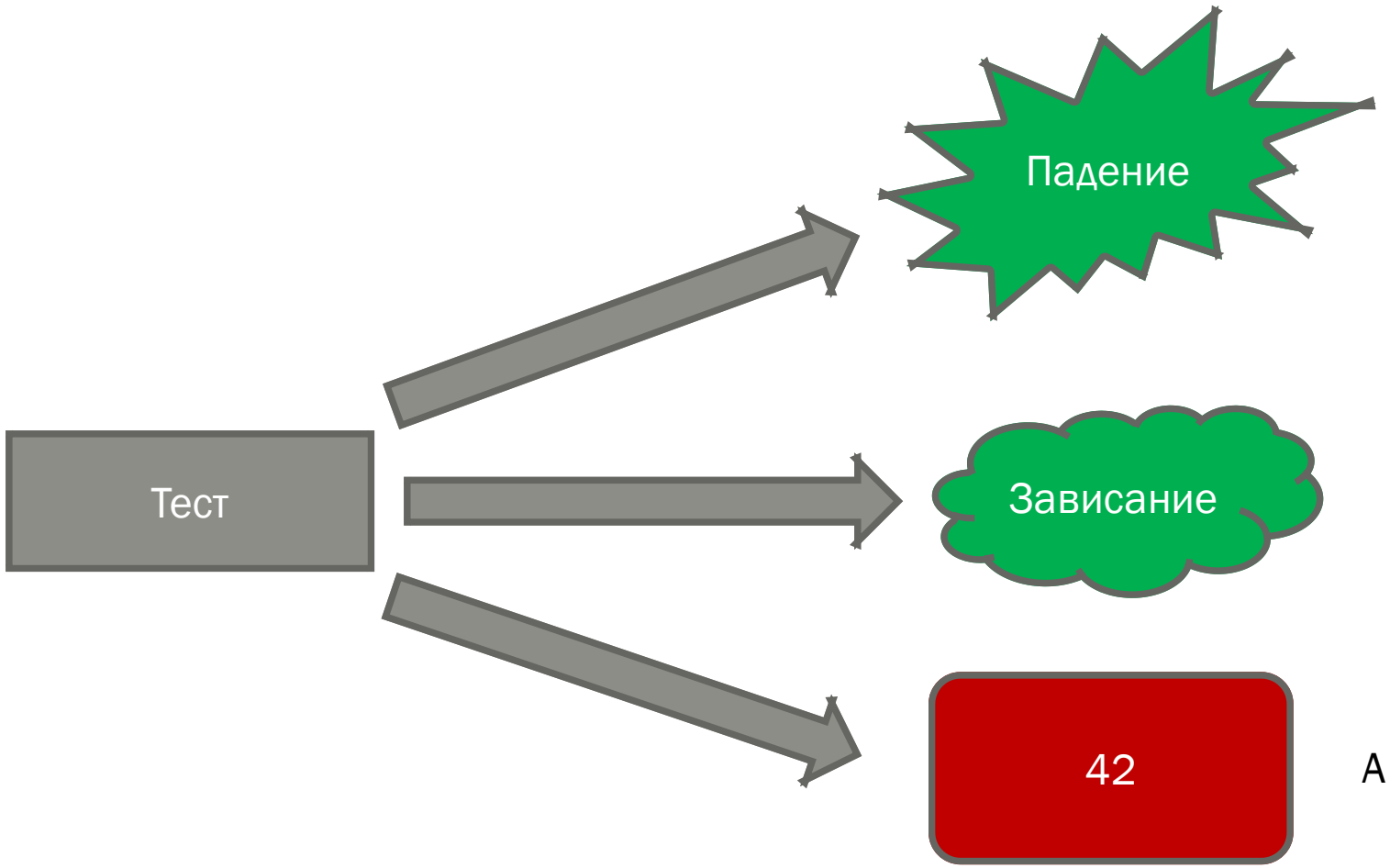
# Генерация программ

```
void main() {
	int a = 3;
	int b = a;
	a = 18;
	<Declare>
}
```

# Генерация программ

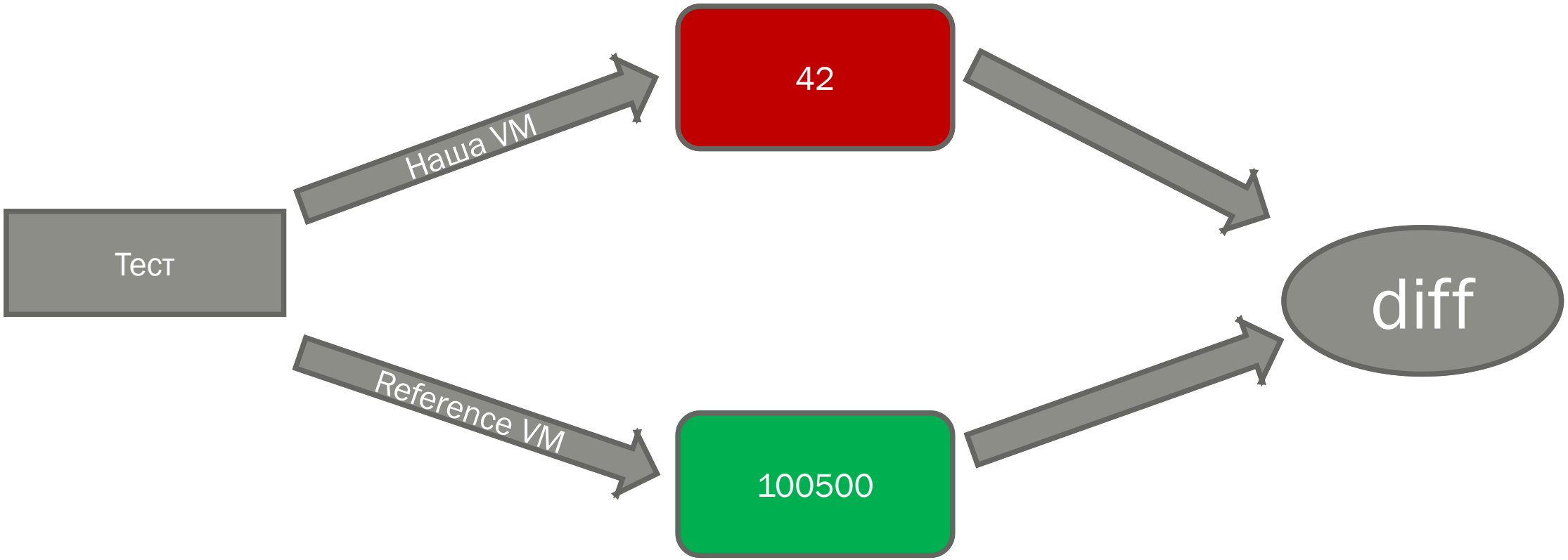
```
void main() {
	int a = 3;
	int b = a;
	a = 18;
	int c = a;
}
```

# Как проверять результат?




А это правильно?

# Нужна reference-имплементация!



# Fuzzer, вперёд!

Но помни, что нужны:

- 100% воспроизводимость проблем
- Разумное время исполнения теста
- Разумный объём кода

# Хорош ли такой тест? Пример 1

```
int test() {
    int a = <...>;
    int b = <...>;
    return a;
}
```

# Пример 1

```
int test() {
    int a = // Посчитано верно
    int b = // Посчитано с ошибкой
    return a;
}
```

# Все переменные участвуют в ответе

```
int test() {
    int a = // Посчитано верно
    int b = // Посчитано с ошибкой
    return a + b;
}
```

# Хорош ли такой тест? Пример 2

```
int test() {
    int a = <...>
    int b = <...>
    if (rnd.nextInt(100500) == 42)
        return b;
    return a;
}
```

# Пример 2

```
int test() {
    int a = // Посчитано верно
    int b = // Посчитано с ошибкой
    if (rnd.nextInt(100500) == 42)
        return b;
    return a;
}
```

# История JavaFuzzer

- Создан в Intel как средство тестирования Dalvik и ART VM
- Следует стратегии генерации
- Язык программирования – Ruby
- Код открыт: https://github.com/android-art-intel/Fuzzer
- Адаптирован для тестирования серверной/десктопной Java в Azul
- Расширен новыми возможностями
- Код также открыт: https://github.com/AzulSystems/JavaFuzzer
- Активно используется для стабилизации Falcon

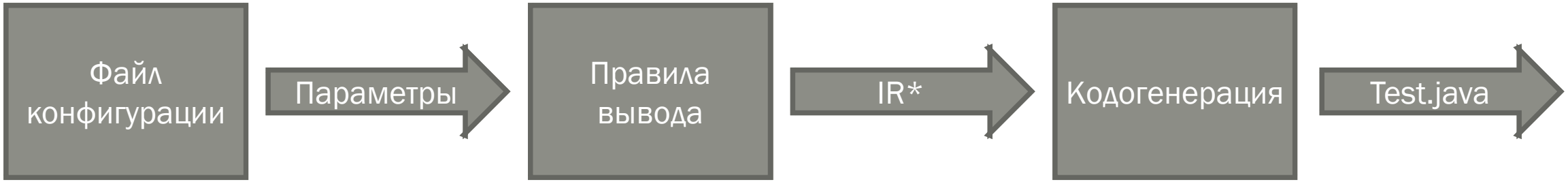
# Поддерживаемые конструкции

- Циклы (for, while, do-while, enhanced for; break, continue)
- Ветвления (if, switch)
- Вызовы методов (рекурсия запрещена)
- Арифметика (целочисленная, floating-point, логические операции)
- Исключения, try-catch-finally
- Массивы (одномерные, многомерные)
- Классы и объекты
- Потоки (в ограниченном режиме, только краши)
- Переопределение методов (в ограниченном режиме, нет abstract)

# Свойства сгенерированных программ

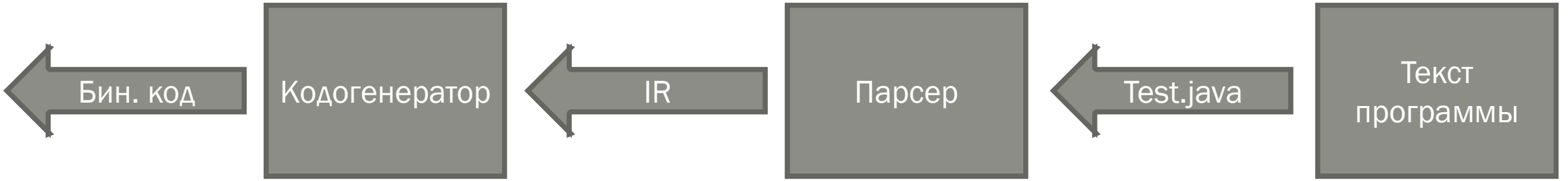
- Гарантированно завершаются за конечное время
- Всегда выдают один и тот же результат
- Вычисляют контрольные суммы с участием всех использованных переменных
- Как показывает практика, неплохо ищут баги

# Устройство JavaFuzzer




IR* - внутреннее представление (internal representation)

# Для сравнения, устройство парсера



# Файл конфигурации

- Ограничения (макс/мин)
- Допустимые типы и их вероятности
- Операции и их вероятности

```
#       class       weight  weight-in-loop   scale-down-factor
statements: {
    ForLoopStmt:     [48,         24,         1.5 ],
    WhileDoStmt:     [16,          8,         1.5 ],
    EnhancedForStmt: [ 4,          2,         1.5 ],
    ContinueStmt:    [ 0,          1,           1 ],
    BreakStmt:       [ 0,          1,           1 ],
    IfStmt:          [ 3,          4,         1.5 ],
    SwitchStmt:      [ 1,          1,           1 ],
    AssignmentStmt:  [ 1,         80,           1 ],
    IntDivStmt:      [ 0,          1,           1 ],
    ReturnStmt:      [ 0,          1,           1 ],
    TryStmt:         [ 1,          1,           2 ],
    ExcStmt:         [ 1,          1,           2 ],
    VectStmt:        [ 0,          8,           1 ],
    InvocationStmt:  [ 1,          1,           1 ],
    CondInvocStmt:   [ 1,          1,           2 ],
    SmallMethStmt:   [ 10,         1,           2 ],
    NewThreadStmt:   [ 40,         1,           2 ]
```

# Пример сгенерированной программы

```
public static void vMeth1() {

    int i7=54703, i8=-14, i18=-32644, i19=4, i20=8, i21=-1, iArr1[]=new int[N];
    short sArr[]=new short[N];
    long lArr1[]=new long[N];

    FuzzerUtils.init(sArr, (short)28174);
    FuzzerUtils.init(iArr1, 5);
    FuzzerUtils.init(lArr1, -50290L);

    for (i7 = 11667; i7 > 225; i7--) {
        switch ((i7 % 3) + 74) {
        case 74:
            for (i18 = 4; i18 < 175; i18++) {
                for (i20 = i7; i20 < 2; ++i20) {
                    double d1=-20.76874;
                    sArr[i18] = (short)(((long)(d1 / (i7 | 1)) >> iArr1[i7]) * i20);
                    i19 = (int)8563533763630855434L;
                    Cls1.fFld += ((lArr1[i7 + 1] - (++i8)) + i18);
                    i19 += (int)(-Cls.instanceCount);
                    if (i7 != 0) {
                        vMeth1_check_sum += i7 + i8 + i18 + i19 + i20 + i21 + FuzzerUtils.checkSum(sArr) +
                            FuzzerUtils.checkSum(iArr1) + FuzzerUtils.checkSum(lArr1);
                        return;
                    }
                    try {
                        iArr1[i20] = (240 / Test.iFld1);
                        iArr1[i20 - 1] = (-44739 % i21);
                        i19 = (Test.iFld1 / -34797);
                    } catch (ArithmeticException a_e) {}
                    i8 += (i20 | Test.instanceCount);
                    Test.iFld1 = (i21--);
                }
            }
            break;
        case 75:
            try {
                Test.iFld1 = (5441 / i18);
                iArr1[i7] = (i20 % -49571);
                i19 = (i8 % 130);
            } catch (ArithmeticException a_e) {}
        case 76:
            i8 |= (int)((-Cls.instanceCount) - (iArr1[i7]--));
            break;
        default:
            i8 -= i18;
        }
    }
    vMeth1_check_sum += i7 + i8 + i18 + i19 + i20 + i21 + FuzzerUtils.checkSum(sArr) + FuzzerUtils.checkSum(iArr1)
        + FuzzerUtils.checkSum(lArr1);
}
```

# К счастью, их можно сокращать

```
1   public static void vMeth1() {
2       int i7=54703, iArr1[] = new int[N];
3       for (i7 = 228; i7 > 225; i7--) {
4           switch ((i7 % 3)) {
5           case 0:
6               for (int i18 = 4; i18 < 175; i18++) {
7                   for (int i20 = i7; i20 < 2; ++i20) {
8                       if (i7 != 0) {
9                           return;
10                      }
11                      iArr1[i20 - 1] = 1;
12                  }
13              }
14          default:
15          }
16      }
17      vMeth1_check_sum += i7;
18  }
```

# Как попробовать?

- https://github.com/AzulSystems/JavaFuzzer
- `rt.sh -r results -p test -sp -conf config.yml 1000`

# Упражнение 1: циклы с FP счётчиком

```
void runFPCounterExample() {
    int x = 0;
    for (float f = 262_000; f < 262_200; f += 0.01f) {
        x++;
    }
    System.out.println(x);
}
```

Что будет напечатано?

| | |
|---|---|
| A) 20000 | B) 20463 |
| C) ArithmeticException | D) Программа зависнет |

# Такая программа зависнет

| Floating point | Bin | Hex |
|---|---|---|
| `f = 262144.0f;` | `01001000100000000000000000000000` | `0x48800000` |
| `step = 0.01f;` | `00111100001000111101011100001010` | `0x3c23d70a` |
| `f + step = 262144.0f;` | `01001000100000000000000000000000` | `0x48800000` |

# Упражнение 2: побитовый перевод

```
double foo() { … }

void testNaN() {
    double x = foo();
    long y = Double.doubleToRawLongBits(x);
    System.out.println(y);
}
```

```
Наша VM: -2251799813685248
Reference VM: -2038219679989760
```

Есть ли баг?

| | |
|---|---|
| A) Да | B) Нет |
| C) Невозможно определить | D) Программа зависнет |

# Упражнение 2: побитовый перевод

```
double foo() { return 0.0 / 0.0; }

void testNaN() {
    double x = foo();
    long y = Double.doubleToRawLongBits(x);
    System.out.println(y);
}
```

```
Наша VM: -2251799813685248
Reference VM: -2038219679989760
```

Есть ли баг?

| | |
|---|---|
| A) Да | B) Нет |
| C) Невозможно определить | D) Программа зависнет |

# Упражнение 2: бага нет

- Нечисел много, разрешается вернуть любое из них
- `-2251799813685248 = 0xfff8000000000000`
- `-2038219679989760 = 0xfff8c24000000000`

# В заключение

- Генерация тестов – эффективный способ находить баги в компиляторе
- Баги есть даже в старых, проверенных компиляторах
- Не всякий тест хорош для поиска багов
- Генерируя простые тесты, можно столкнуться с ложной тревогой

# Полезные ссылки

- Zing VM: https://www.azul.com/products/zing/virtual-machine/
- LLVM Project: http://llvm.org/
- Falcon: https://www.azul.com/press_release/falcon-jit-compiler/
- Java Fuzzer for Android: https://github.com/android-art-intel/Fuzzer
- Java Fuzzer: https://github.com/AzulSystems/JavaFuzzer

# Благодарность

Спасибо за внимание!