Двунаправленное направляемое конфликтами резюмирование кода в символьной виртуальной машине V#

Седлярский Михаил, 21.M07-мм

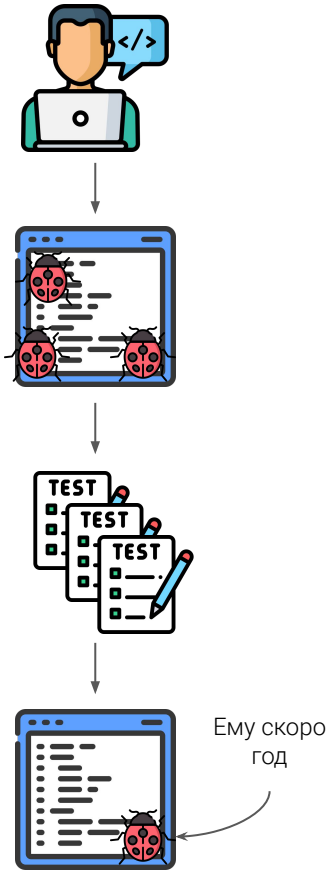
научный руководитель:
Мордвинов Дмитрий Александрович

Предметная область

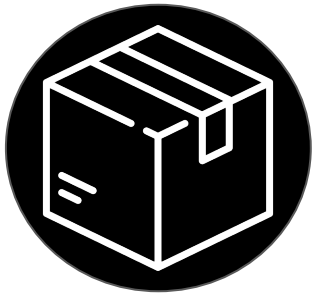
1. Разработчики пишут код
2. Код содержит ошибки
3. Разработчики ~~иногда~~ пишут тесты

Насколько рукописные тесты покрывают код?
Рассмотрены ли все краевые случаи?
В тестах точно нет ошибок?

А стоит ли все тесты писать самостоятельно?



Методы автоматической генерации тестов

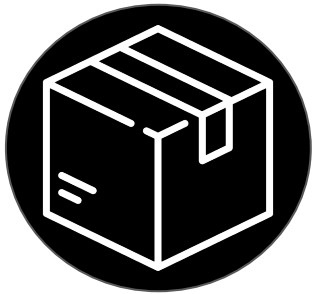


Fuzzing

Fuzzing + инструментация

- American Fuzzy LOP
- Radamsa
- Honggfuzz
- Libfuzzer
- OSS-Fuzz
- Sulley Fuzzing Framework
- boofuzz
- BFuzz
- PeachTech Peach Fuzzer

Методы автоматической генерации тестов



Fuzzing

Fuzzing + инструментация

- American Fuzzy LOP
- Radamsa
- Honggfuzz
- Libfuzzer
- OSS-Fuzz
- Sulley Fuzzing Framework
- boofuzz
- BFuzz
- PeachTech Peach Fuzzer

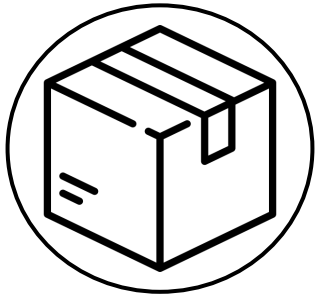
Недостатки фаззинга:

Программа с возможными ошибками

Тестируется случайными наборами данных

Но степень покрытия оставляет желать лучшего

Методы автоматической генерации тестов



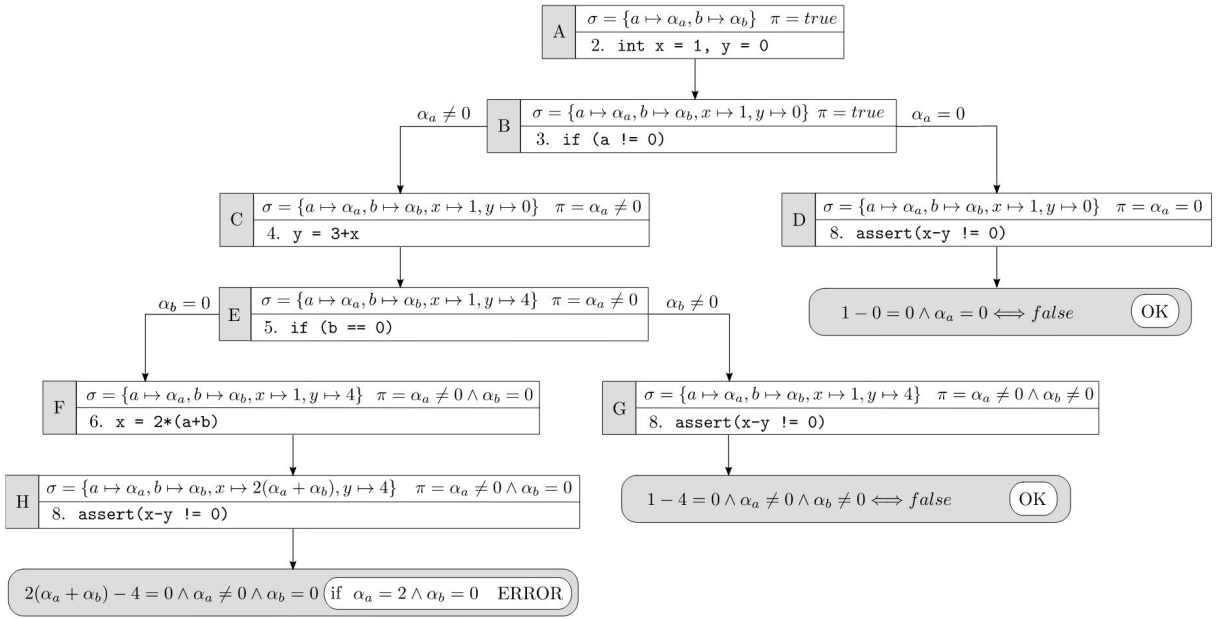
Символьное исполнение

- KLEE
- PEX
- SPF
- JDart
- V#
- CrossHair
- DART
- CATG

Символьное исполнение — это метод анализа программного обеспечения, позволяющий определить какие входные данные приводят к выполнению тех или иных его частей.

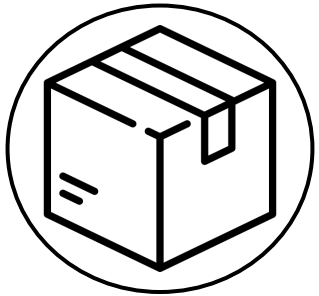
Символьное исполнение. Пример

```
void foobar(int a, int b) {
    int x = 1, y = 0;
    if (a != 0) {
        y = 3 + x;
        if (b == 0)
            x = 2 * (a + b);
    }
    assert (x - y != 0);
}
```



Пример прямого символьного исполнения.
Взят из https://arxiv.org/pdf/1610.00502.pdf

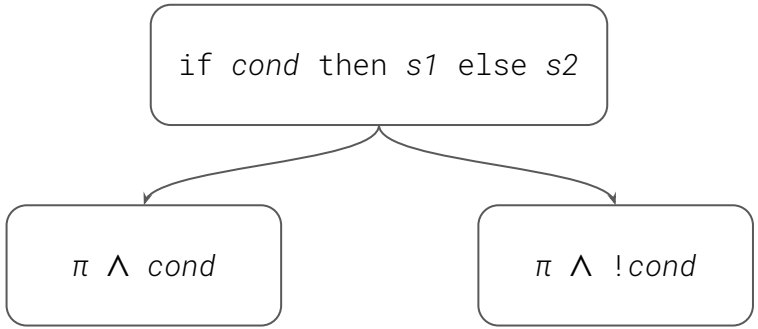
Почему символьное исполнение?



Символьное
исполнение

- Исследуем все возможные состояния программы
- Находим ошибки и генерируем тесты
- Можно доказывать некоторые свойства ПО

Комбинаторный взрыв путей



Если мы встретим ветвление в цикле, то количество путей будет расти экспоненциально.

Невозможно обойти все пути за разумное время. Поэтому часть из них придётся обрезать.

Символьное исполнение. Прямое и обратное

- *Прямое символьное исполнение*. Если нас интересует исполнение конкретной части кода, то в общем случае мы не знаем как быстро прямой анализ дойдёт до нужного места и дойдёт ли вообще.

- *Обратное символьное исполнение*. Начинает работу не с точки входа в программу (т.е. не с её начала), а с некоторой целевой локации кода, и далее движется в обратном направлении.

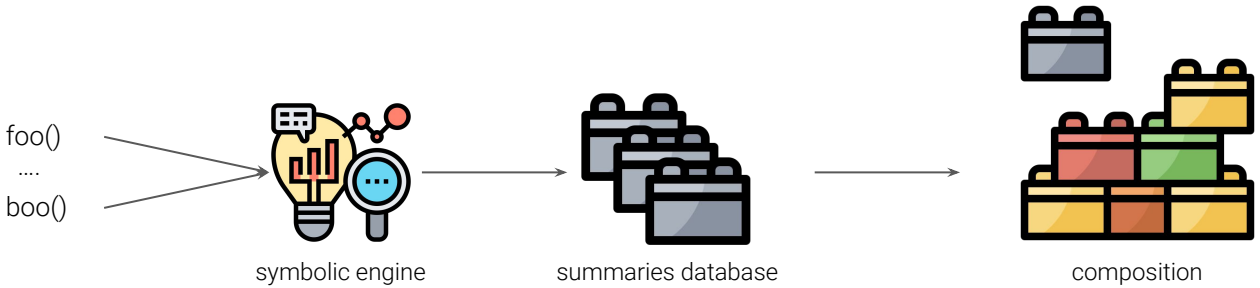
- Что если идти с двух сторон навстречу друг другу?

Двунаправленное символьное исполнение

Поочередное использование шагов прямого и обратного символьного исполнения, может устранить проблемы обоих подходов. Их правильная комбинация позволяет не только быстро генерировать тестовые данные для труднодостижимых локаций кода, но и выявлять такие локации в ходе исполнения.

Композиционное символьное исполнение и резюмирование

- Тела циклов и функций можно анализировать только один раз
- Полученные результаты можно переиспользовать
- Часть ограничений пути можно построить из готовых “рюзюме”
- Сокращаем пространство поиска



Как же резюмировать код?

Как кратко обобщить программу (функцию)?

- Аппроксимации функции сверху: мы опишем все возможные состояния кода, но возможно, включим и те, что на самом деле никогда не достигаются.
- В какой форме лучше всего описывать состояния? Очевидно, что наивное перечисление здесь не подойдёт.
- Предлагается использовать формулы логики первого порядка. Например:

```
int abs(int x) {
    return x < 0 ? -x : x;
}
```

$$x \neq \texttt{INT\_MIN} \Rightarrow \texttt{abs}(x) \geq 0$$

Данная модель включает в себя вполне бессмысленные состояния (x = -4 и abs(x) = 9), но может доказать недостижимость таких как: abs(x) = -1404

Как же резюмировать код?

Как построить аппроксимацию исследуемой функции?

Планируется дополнить алгоритм двунаправленного символьного исполнения.

После обнаружения невыполнимого ограничения пути будет находится минимальное unsat-ядро формулы и на основе атомов полученного ядра строится обобщение исследуемого участка кода.

Постановка задачи

1. Провести детальный обзор существующих подходов резюмирования кода и стратегий ветвлений в символьном исполнении
2. Разработать алгоритм двунаправленного направляемого конфликтами резюмирования кода
3. Реализовать алгоритм в символьной виртуальной машине V#.
4. Провести эксперименты, доказывающие эффективность предложенного алгоритма и его реализации

Данный проект проводиться в рамках символьной виртуальной машины V# для платформы .net core

https://github.com/VSharp-team/VSharp

План работ на ближайший год

1. Реализация извлечения минимальных unsat-ядер в V#
2. Реализация двунаправленных эвристик на основе unsat-ядер
3. Реализация базы данных лемм
4. Реализация механизма распространения относительно индуктивных лемм
5. Апробация полученного решения

Спасибо за внимание!