



Московский государственный университет имени М.В. Ломоносова  
Факультет вычислительной математики и кибернетики  
Кафедра информационной безопасности  
Лаборатория безопасности информационных систем

Николайчук Артём Константинович

Обзор методов fuzz-тестирования программ, принимающих на  
вход данные, порождаемые КС-грамматикой

Курсовая работа

Научный руководитель:  
М.Н.С  
А.А.Петухов

Москва, 2022

## Аннотация

В настоящее время разработчики всё больше беспокоятся о безопасности создаваемых приложений. Цена ошибки или бага может очень высокой. Тестирование стало неотъемлемой частью жизненного цикла разработки программного обеспечения. Известно, что даже 100%-ое покрытие исходного кода тестами не гарантирует отсутствие ошибок. Более того, разработчики при написании тестов руководствуются тем, как должна вести себя программа. Из-за этого многие "неочевидные" частные случаи могут быть пропущены. Некоторые программы, такие как парсеры, интерпретаторы, компиляторы, обрабатывают данные, которые имеют сложную структуру. Для них просто невозможно перебрать все варианты входных данных, чтобы удостовериться, что всё работает правильно. Для таких программ разумно применять методы fuzz-тестирования.

# Содержание

<b>1</b>	<b>Введение</b>	<b>4</b>
1.1	fuzz-тестирование . . . . .	4
1.2	Представление входных данных . . . . .	4
<b>2</b>	<b>Цель работы</b>	<b>5</b>
<b>3</b>	<b>Анализ предметной области</b>	<b>6</b>
3.1	Получение начального множества тестов . . . . .	6
3.2	Генерация тестов . . . . .	6
3.2.1	Способы генерации . . . . .	6
3.2.2	Свойства сгенерированных тестов . . . . .	8
3.3	Получение фидбека от запуска теста на SOT . . . . .	8
3.4	Оценка полезности теста . . . . .	8
3.5	Способы обработки тестового множества . . . . .	9
3.6	Методы улучшения тестов . . . . .	9
3.6.1	Уменьшение размера теста . . . . .	9
3.6.2	Мутации . . . . .	10
<b>4</b>	<b>Обзор предметной области</b>	<b>12</b>
4.1	Superion . . . . .	12
4.2	Grammarinator . . . . .	13
4.3	Nautilus . . . . .	13
4.4	EvoGFuzz . . . . .	14
<b>5</b>	<b>Результаты</b>	<b>16</b>

# 1 Введение

## 1.1 fuzz-тестирование

fuzz-тестирование - способ автоматического тестирования программного обеспечения. Фаззер генерирует случайные входные данные и улучшает или изменяет их, затем анализирует работу программы на этих данных и пытается обнаружить потенциальные дефекты или уязвимости программного обеспечения. Фаззеры принято классифицировать по принципу генерации данных:

- Мутационные фаззеры обрабатывают заранее подготовленное множество входных данных. Наиболее популярными изменениями являются заимствованные из биологии мутации и скрещивания. Мутации - это изменение какой-то части входных данных на случайную. При скрещивании выбираются два примера, которые обмениваются друг с другом частью данных.
- Генерационные фаззеры создают новые примеры, основываясь на информации о требуемой структуре входных данных.
- Смешанные фаззеры объединяют в себе два предыдущих подхода. Например, при мутации данные могут меняться не на случайные, а на сгенерированные. Или фаззер может сначала создать пул тестовых данных и к нему применять мутационный метод.

## 1.2 Представление входных данных

На практике оказалось очень удобно задавать структуру входных данных с помощью грамматик. Если мы знаем грамматику, то все возможные инпуты можно представить абстрактным синтаксическим деревом(далее АСТ). Это позволяет избегать синтаксических ошибок на этапе запуска программы. В дальнейшем мы покажем, что такое представление полезно при генерации и мутации данных.

## 2 Цель работы

Цель данной работы - сделать значимых методов и средств, используемых при fuzz-тестировании программ, обрабатывающих структурированные входные данные. Для достижения этой цели требуется:

- Сформировать критерии для сравнения методов
- Проанализировать существующие решения по выделенным критериям
- Дать оценку каждому критерию

## 3 Анализ предметной области

Базовый алгоритм работы любого фаззера состоит из шагов:

1. Обработать Soft Under Test(далее SOT)
2. Получить начальное множество входных данных и создать из него пул тестов
3. Выбрать из пула один или несколько тестов и посмотреть результат их обработки SOT
4. Решить будет ли полезен этот тест в будущем
5. Мутировать тест и добавить его в пул
6. Перейти к шагу 3

### 3.1 Получение начального множества тестов

- Взять тесты, которые писали разработчики для SOT. Этот способ позволяет сразу получить хорошее покрытие кода. При обнаружении бага разработчики исправляют его и часто добавляют тест, который проверяет работоспособность программы в этом месте. В этом смысле тесты, как начальное множество, позволяют сразу добираться до "слабых"мест в SOT. Минус этого способа - тесты разработчиков не всегда доступны.
- Собрать пул тестов из примеров в интернете. Например, в случае fuzz-тестировании интерпретатора javascript можно в начальное множество добавлять примеры javascript кода с гитхаба. Таким способом можно получить широкий пул.
- Можно самим сгенерировать это множество. В этом случае появляется возможность подтолкнуть фаззер в определённом направлении. Детали процесса генерации описаны ниже.

### 3.2 Генерация тестов

#### 3.2.1 Способы генерации

Алгоритм создания тестовых данных опирается на знание их структуры. Как было отмечено ранее эту структуру удобно задавать контекстно-свободной грамматикой. В этом случае процесс генерации нового теста заключается в построении его АСТ. Шаги алгоритма:

1. Положить в корень дерева стартовую вершину и положить её в очередь вершин.
2. Взять текущую вершину из очереди.
3. Если текущая вершина - терминальная, то перейти к следующей вершине.
4. Каким-то способом выбрать продукцию из правила вывода для текущего нетерминала и добавить все символы из неё в очередь.
5. Перейти к шагу два.

Все известные методы опираются при выборе продукции нетерминала в пункте 4 на вероятности, то есть каждой продукции каждого нетерминала задаётся вероятность её выбора. Разные алгоритмы отличаются друг от друга способом задания этой вероятности.

- Выбирать продукцию равновероятно для каждого нетерминала. Плюсы - простая реализация. Минусы - будет часто генерировать похожие тесты, медленно покрывает всевозможные ветки деревьев. Например: если у стартового символа одна из продукций - один терминальный символ, то большая часть сгенерированных тестов будет состоять из этого символа.
- Алгоритм построения похожих. Если есть какое-то множество примеров, то нужно каждый из них представить в виде АСТ и для каждого нетерминала для каждой продукции подсчитать частоту её встречаемости. По этим частотам можно вычислить вероятности выбора каждой продукции. Чем чаще встречается переход в тестах, тем чаще мы его будем использовать. Плюсы - позволяет направлять фаззер, путём изменения множества. Выбор символа становится более осмысленным с точки зрения программирования. Минус - нужно начальное множество тестов.
- Алгоритм построения отличных. Отличие этого способа от предыдущего - вероятность обратно пропорциональна частоте встречаемости, то есть чем чаще встречается переход в тестах, тем реже он будет генерироваться. Это позволит "мыслить" фаззеру в противоположном направлении. Плюсы - чаще будем покрывать неожиданные пути в грамматике. Минусы - чаще будем попадать в неинтересные символы (например в javascript часто будет вызываться return).
- Задать вероятности пропорционально количеству возможных поддеревьев в продукции. Если в вершине потенциально много поддеревьев, будем ходить туда чаще. Плюсы - позволит наиболее полно покрыть грамматику. Минусы - требуется предсчёт количества деревьев для каждого нетерминала.

### 3.2.2 Свойства сгенерированных тестов

Сгенерированные тесты должны не только быть синтаксически корректными, но и обладать полезными для fuzz-тестирования свойствами.

- Нужно стремиться создавать короткие тесты. Чем тест длиннее - тем дольше он будет выполняться в SOT, тем сложнее его анализировать, тем дольше будет его обработка.
- Чем тест сложнее и рекурсивнее, тем вероятнее найти на нём ошибку SOT.

### 3.3 Получение фидбека от запуска теста на SOT

Получение обратной связи от запуска теста - важная часть фаззера. Именно она часто позволяет определять значимость входного набора данных. Метрики, которые полезно оценивать:

- Самая весомая метрика - падение SOT с некоторыми видами исключений. Например, `double free` в языке программирования `c++`. Зачастую такие ошибки означают, что найден баг и этот тест нужно дополнительно исследовать.
- Покрытие кода(`line coverage`) и покрытие функций(`function coverage`) - подсчёт количества строк/функций, которые покрыл тест. В контексте множества тестов можно выявлять те, которые попадают в новые строки/функции. Разным строкам и функциям можно давать разный вес при подсчёте метрики. Это позволяет направлять фаззер в сторону исследования этих функций.
- Покрытие веток(путей) - более сложный вариант предыдущей метрики. При подсчёте учитывается последовательность выполнения строк кода или вызовов функций. Плюсы - покрытие веток гораздо более информативная метрика, чем предыдущая. Минусы - число веток растёт очень быстро с увеличением количества кода в SOT.

### 3.4 Оценка полезности теста

После запуска теста нужно полезен ли этот тест или от него можно отказаться. Для этого нужно научиться сравнивать различные входные данные друг с другом. Для этого введём функцию  $value : \mathbb{X} \rightarrow \mathbb{R}$ , которая каждому тесту ставит в соответствие его численную оценку. Конкретных реализаций этой функции может быть много. Приведём основные параметры, от которых она может зависеть:

- При наличии исключения при запуске, значение функции становится бесконечным.



- Чем больше покрытие кода, тем больше значение функции. Если определяется, что тест покрывает новый участок кода, то функцию можно сделать равной бесконечности.
- Чем короче тест, тем больше значение функции.
- Чем тест разнообразнее, то есть чем больше символов грамматики он покрывает, тем лучше.

### 3.5 Способы обработки тестового множества

Существует две стратегии обработки тестового множества:

1. Первая - наиболее часто встречающаяся - обрабатывать каждый тест по отдельности. Это позволяет распараллелить процесс fuzz-тестирования, что серьёзно его ускоряет.
2. Вторая стратегия основана на теории эволюции Дарвина. Выбирается множество тестов. Для них всех рассчитывается функция полезности. Затем начинается процесс "выживания". Какой-то процент (например 10) тестов с наибольшей функцией полезности объявляется выжившими. Остальные случайным образом делятся на группы, в которых выживает несколько сильнейших. Потом среди оставшихся несколько случайных тестов объявляются выжившими. Тесты, которые не выжили отбрасываются.

### 3.6 Методы улучшения тестов

Все успешные тесты необходимо преобразовывать для продолжения процесса fuzz-тестирования. Целями улучшения могут быть:

- Поиск наиболее оптимального теста с точки зрения фаззера, обладающего теми же свойствами, что и улучшаемый. Другими словами - оптимизация процесса fuzz-тестирования.
- Дальнейшее продвижение фаззера, в том числе выход из локальных экстремумов.

#### 3.6.1 Уменьшение размера теста

Как было отмечено ранее, у коротких тестов есть ряд преимуществ по сравнению с длинными. Уменьшенный тест должен сохранить все полезные свойства длинного. Например, если старый тест покрывает какую-то новую функцию, то и новый должен покрывать эту функцию. Стандартные методы:

- Самый простой способ - построить для теста его АСТ и поочерёдно удалять поддеревья. Если после удаления поддерева, полезность теста не уменьшилась, то заменяем старый тест на новый и продолжаем процесс уменьшения.

Плюсы - неплохая скорость работы. Если повезёт, то можно существенно укоротить тест. Минусы - удаление поддерева может сделать тест синтаксически некорректным. Этот способ охватывает только очень локальные изменения теста.

- Уменьшение поддерева. Способ похож на предыдущий, только вместо отбрасывания поддерева, мы заменяем его на более короткое. Новый тест всегда будет синтаксически корректным, но перебор всех поддеревьев может занять длительное время.
- Рекурсивное замещение поддерева. По сути этот способ является эвристикой предыдущего. Если в вершине у нетерминала  $F$  есть сын  $F$ , то производится замена поддерева текущей вершины на поддерево сына. Такое изменение оставит тест синтаксически корректным и приведёт к его упрощению для фаззера. Минус - этот способ может быть редко применим.
- Контролировать размер во время построение теста. Этот способ применим, если используется генерации тестов. Тогда этап уменьшения можно опустить.

### 3.6.2 Мутации

Мутации являются двигателем фаззера, позволяют ему эволюционировать. Если фаззер "застрял" на каком-то этапе, то мутации могут сделать шаг в сторону и процесс fuzz-тестирования продолжится. Если тесты представимы в виде АСТ, то удобно описывать мутации изменениями над АСТ.

- Скрещивание тестов друг с другом. Возьмём два успешных теста, представим их в виде АСТ и заменим поддерево одного теста на поддерево другого согласно правилам грамматики. Такое изменение, например, позволяет быстро увеличивать покрытие веток SOT.
- Создадим пул поддеревьев. Для каждой вершинки в АСТ теста считаем какую-то дополнительную информацию и меняем, только на поддерево из пула с такой же информацией. Например, информацией может быть тип нетерминала, количество идентификаторов. В пул тестов нужно добавлять поддерева из тестов, которые дают существенное улучшение полезности. Эта мутация позволяет обращать больше внимания на прогресс в фаззере. Например, если тестом покрыта какая-то новая функция, то среди новых тестов будет много тех, которые эту функцию исследуют. Сохранение дополнительной информации позволяет увеличить вероятность попадания в новую функцию.
- Рекурсивное дублирование поддерева. Это обратное действие одному из способов уменьшения. Эта мутация направлена на усложнение структуры выполняемого кода в SOT. Например, создание вложенных циклов или рекурсии.

- Применимы стандартные AFL мутации - бит флип, замена "интересных значений". Они более случайны, их имеет смысл применять, когда остальные не работают. После применения этой мутации тест может стать синтаксически некорректным.
- Замена поддерева на случайно сгенерированное. Эта мутация - адаптированная для fuzz-тестирования с грамматикой версия предыдущего пункта. Основное отличие - тест останется корректным.
- При использовании метода генерации тестов по вероятностной грамматике можно изменять вероятности генерации продукции для конкретного нетерминала. Можно изменять на случайные, можно просто немного прибавить или отнять вероятности у нескольких символов. Новые тесты могут сильно отличаться от предыдущих. Плюс такой мутации - не нужно работать с самими тестами, достаточно просто изменить вероятность при генерации.

## 4 Обзор предметной области

Цель обзора - оценить критерии из пункта 3, то есть добавить им плюсы и минусы. В обзоре рассмотрены самые цитируемые актуальных статьи с google scholar про фаззеры по грамматикам.

### 4.1 Superion

Superion - grey-box fuzzer, созданный на основе AFL. Является линией отсчёта для остальных рассматриваемых фаззеров. Разбор Superion по критериям:

1. Подразумевается, что начальный пул тестов уже имеется.
2. Генерация тестов не используется.
3. В Superion используется стандартное покрытие AFL - покрытие веток. В реализации происходит подсчёт всех веток, которые покрыл тест, и количество проходов по ним.
4. Функция полезности - при увеличении покрытия, тест обозначается успешным.
5. Каждый тест обрабатывается по отдельности.
6. Улучшение тестов:
  - Для уменьшения размера тестовых данных используется самая простая стратегия - удаление поддеревьев. В статье делается следующий вывод: "Таким образом, несмотря на относительно низкий коэффициент обрезки, эта стратегия обрезки с учетом грамматики может значительно улучшить коэффициент валидности для тестовых входных данных после обрезки, что облегчает и ускоряет дальнейшую работу с ними."
  - Используется мутация скрещивания тестов. Реализация: берётся текущий тест и случайный из очереди длины не более 10000 байт. Они парятся и из их поддеревьев формируется множество для мутаций. Берётся не больше 10000 поддеревьев и длина не более 200 байт. Затем каждое поддерево текущего теста заменяется на каждое поддерево из множества, формируя новый тест.  
Плюсы: очень подробный перебор возможных тестов.  
Минусы: несмотря на ограничения тестов получается очень много. Довольно большая проблема фаззера - время подготовки данных. Если программа выполняется  $t$  секунд, то время на мутации примерно  $t/3$ .
  - Применяется мутация замены по словарю. В каждое корректное, согласно грамматике, место вставляется значение из множества стандартных для грамматики конструкций. Делается предположение, что все токены должны состоять из цифр и букв. Словарь токенов можно либо составить вручную, либо взять самые популярные токены из множества

тестов. При мутации новый токен либо вставляется между двумя старыми, либо заменяет один из них.

В статье сравнивали различные мутации. Самыми удачными оказались мутация замены поддеревьев и перезапись токенов при помощи словаря, определённого человеком.

Superion хорошо себя показал при сравнении с обычным AFL и jsfunfuzz - специальным фаззером для js. Тестировали на интерпретаторах xml и javascript.

## 4.2 Grammarinator

Grammarinator - инструмент, позволяющий генерировать тестовые данные по имеющейся грамматике. Так как это неполноценный фаззер имеет смысл описать только некоторые критерии.

1. При создании тестов используется равновероятная грамматика с небольшими улучшениями. После выбора какого-то правила вероятность его повторного выбора уменьшается. Такой подход помогает направлять поколение к менее посещаемым частям грамматики. Во время генерации контролируется размер тестов. Для каждого символа грамматики производится подсчёт минимального размера поддерева и при генерации выбираются только те символы, поддеревья которых не превысят заданную длину теста. В Grammarinator используется алгоритм переиспользования токенов. Это позволяет уменьшить количество тестов с семантическими ошибками. Например, при fuzz-тестировании интерпретатора javascript будет сгенерировано меньше тестов с ошибкой "использование необъявленной переменной".
2. Создаётся пул тестов, которые используются для ускорения создания тестовых примеров с помощью эволюционных методов. Одна из возможностей заключается в выполнении случайной рекомбинации деревьев из пула для создания новых тестовых примеров. Другой вариант - взять деревья из пула и использовать их, как поддеревья генерируемых тестов.

## 4.3 Nautilus

Nautilus - фаззер, сочетающий в себе способность генерировать входные данные по грамматике и ориентирующийся на покрытие кода. Разбор Nautilus по критериям:

1. Начальный пул тестов не нужен.
2. Тестировались два варианта генерации: равновероятный по грамматике и равномерный по количеству возможных поддеревьев. В статье методы тестировали на разных интерпретаторах: для некоторых лучше работал второй метод, для некоторых одинаково.

3. Используемая метрика - покрытие строк кода.
4. Функция полезности - при увеличении покрытия, тест обозначается успешным.
5. Тесты обрабатываются по отдельности.
6. Улучшение тестов:
  - Используются две стратегии уменьшения тестов. Уменьшение поддерева - для каждого нетерминала изменяется поддерево на минимально возможное поддерево в этом нетерминале и проверяется покрытие. Далее применяется рекурсивная минимизация. Её цель состоит в том, чтобы уменьшить количество рекурсий, идентифицируя рекурсии и заменяя их по одной за раз. Из этих стратегий лучше работает первая.
  - К каждому тесту применяется один тип мутаций.
    - (a) Случайное поддерево изменяется на случайно сгенерированное поддерево.
    - (b) Каждая вершина меняется на дерево, сгенерированное по смежным правилам.
    - (c) Рекурсивная мутация - если у нетерминала, есть сын, совпадающий с нетерминалом, повторяем его  $2^n$  раз,  $n$  - небольшое.
    - (d) Скрещивание деревьев. Скрещивание происходит только между тестами, покрывающими новые участки кода.
    - (e) Мутации АФЛ.

В начале процесса fuzz-тестирования лучше всего себя показывает первый тип мутаций. Через некоторое время скрещивание обгоняет его.

## 4.4 EvoGFuzz

В EvoGFuzz идея строить тесты по грамматике с вероятностями является ключевой. Удалось спроектировать фаззер так, что почти весь код является реализацией генерации теста. Мутации опираются на вероятности в грамматике.

1. Требуется множество тестов для создания грамматики с вероятностями.
2. Для генерации входных данных используется грамматика с вероятностями. При создании контролируется размер входных данных.
3. При запуске теста требуется получать только информацию об исключениях.
4. Функция полезности выглядит следующим образом:

$$f(x) = \infty, \text{ если тест упал с ошибкой}$$

$$f(x) = \text{expansions}(x)^2 / \text{length}(x),$$

$\text{expansions}(x)$  - количество расширений (нетерминальных символов)

$length(x)$  - длина теста в символах.

Используя эту функцию более сложные тесты получают лучший скор, а длина штрафуются. Плюсы: просто вычисляется, учитывает свойства, которые мы хотим получить от теста. Минусы: Тесты могут получаться длинными в терминах грамматики.

5. Используется стратегия эволюции тестов. Запускается всё множество тестов и для каждого вычисляется функция полезности. Остаётся примерно 5% тестов, с максимальной функцией качества. Среди не вошедших в топ 5% случайным образом формируется 10 групп по 10 тестов и победители остаются для дальнейшей работы.
6. Единственная мутация - для каждого нового поколения выбирается случайный нетерминал в грамматике и вероятности перехода из него меняются на случайные.

## 5 Результаты

В рамках данной работы были получены следующие результаты:

- Выделены критерии для сравнения фаззеров программ, принимающих на вход данные, порождаемые КС-грамматикой
- Сделан обзор существующих решений.