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

Николайчук Артём Константинович

Исследование методов автоматической генерации входных данных для тестирования модулей обработки шаблонов веб-страниц

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

Научный руководитель:
м.н.с
А.А.Петухов

Москва, 2023

Аннотация

В настоящее время разработчики всё больше беспокоятся о безопасности создаваемых приложений. Цена ошибки или бага может быть очень высокой. Тестирование стало неотъемлемой частью жизненного цикла разработки программного обеспечения. Некоторые программы, такие как шаблнизаторы, парсеры, интерпретаторы, компиляторы, обрабатывают данные, которые имеют сложную структуру. Вручную написать тесты с приемлемым покрытием для этих программ не представляется возможным. Для них разумно применять методы генерации тестов, а также fuzz-тестирования поверх этих методов для поиска ошибок. Эти методы подлежат обзору для того, чтобы оценить их применимость для поиска недостатков безопасности шаблонизаторов. Для исследования их применимости разработан фаззер для стандартного шаблонизатора языка Golang.

Содержание

1 Введение

1.1 Введение в предметную область

1.1.1 Шаблонизаторы

Шаблонизаторы - это программные инструменты, которые используются для автоматического генерирования HTML-кода и других статических документов из динамических данных. Они позволяют разработчикам создавать и использовать шаблоны, которые определяют структуру и внешний вид веб-страниц, не привязываясь к конкретным данными. Шаблонизаторы обычно основаны на языках программирования, таких как Python, PHP, Golang, Java и JavaScript. Они широко используются в веб-разработке, чтобы создавать динамические веб-сайты и приложения.

Шаблон - текст, содержащий строковые константы и специальные конструкции шаблонизатора.

```
html(lang="en")
  head
    title Node js Pug
    meta(charset="utf-8")
  body
    h1 #{title}

    if content
      p #{content}
    else
      p No content
```

Рисунок 1: Пример шаблона на языке pug

Движок шаблонизатора - программное обеспечение, разработанное для объединения шаблона и данных в финальный документ. При работе шаблонизатора сначала производится лексический и семантический анализ шаблона. Затем шаблонизатор обрабатывает все встреченные служебные блоки. В них могут быть заложены как простые конструкции - например, подставить значение переменной, так и более сложные выражение, такие как условие, циклы, вызовы функций. Разберём пример с рисунка 2: шаблонизатору во время рендеринга шаблона доступна только переменная *title*, её значение он подставит в тег $h1$, переменной *content* у него нет, поэтому выражение условного оператора будет ложно, и в итоговый шаблон выведется альтернативный вариант условного

```
const pug = require('pug');
var fn = pug.compileFile('simple_template.pug');
var rendered = fn({ title: 'Hello, world!'})
console.log(rendered)
```

(a) программа на языке js, рендерящий шаблон

```
<html lang="en">
  <head>
    <title>Node js Pug</title>
    <meta charset="utf-8"/>
  </head>
  <body>
    <h1>123</h1>
    <p>No content</p>
  </body>
</html>
```

(b) результат

Рисунок 2: Пример работы шаблонизатора

оператора.

1.1.2 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[1]:

- 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.

1.1.3 Представление входных данных

На практике оказалось очень удобно задавать структуру входных данных с помощью грамматик (Пример грамматики на рис. 3). Если известна грамматика, то все возможные входные данные можно представить абстрактным синтаксическим деревом (далее AST - Abstract Syntax Tree). Пример AST представлен на рисунке 4. Это позволяет избегать синтаксических ошибок на этапе запуска

[1] https://habr.com/ru/company/dsec/blog/517596/#chto-takoe-fazzing

программы. В дальнейшем будет показано, что такое представление полезно при генерации и мутации данных.

```
<start>   ::= <expr>
<expr>    ::= <term> + <expr> | <term> - <expr> | <term>
<term>    ::= <term> * <factor> | <term> / <factor> | <factor>
<factor>  ::= +<factor> | -<factor> | (<expr>) | <integer> | <integer>.<integer>
<integer> ::= <digit><integer> | <digit>
<digit>   ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Рисунок 3: Грамматика арифметических выражений

1.2 Цель работы

Цель данной работы - исследовать применимость методов автоматической генерации входных данных для тестирования модулей обработки шаблонов вебстраниц. Для достижения этой цели необходимо рассмотреть существующие методы fuzz-тестирования, проанализировать их и выбрать наиболее подходящие для шаблонизаторов. Исследовать применимость этих методов экспериментальным способом.

1.3 Постановка задачи

1. Исследовать предметную область fuzz-тестирования на основе грамматик.
   - Сформировать критерии для сравнения методов.
   - Проанализировать существующие решения по выделенным критериям.
   - Дать оценку каждому критерию.
   - Сделать вывод о том, какие методы будут использованы в фаззере.
2. Исследовать существующие инструменты fuzz-тестирования и генерации входных данных по грамматике, и определить, какие из них можно будет переиспользовать при разработке своего фаззера.
3. Разработать прототип фаззера
   - Сделать обзор механизмов безопасности шаблонизаторов и уже найденных в них уязвимостях.
   - Выбрать шаблонизатор, поведение которого будет исследоваться.
   - Разработать фаззер, реализовав определённые на этапе анализа методы.
   - Провести эксперименты, подобрать оптимальные параметры для фаззера, в том числе внести доработки в предложенную архитектуру фаззера.
   - Сделать вывод об эффективности каждого из методов.

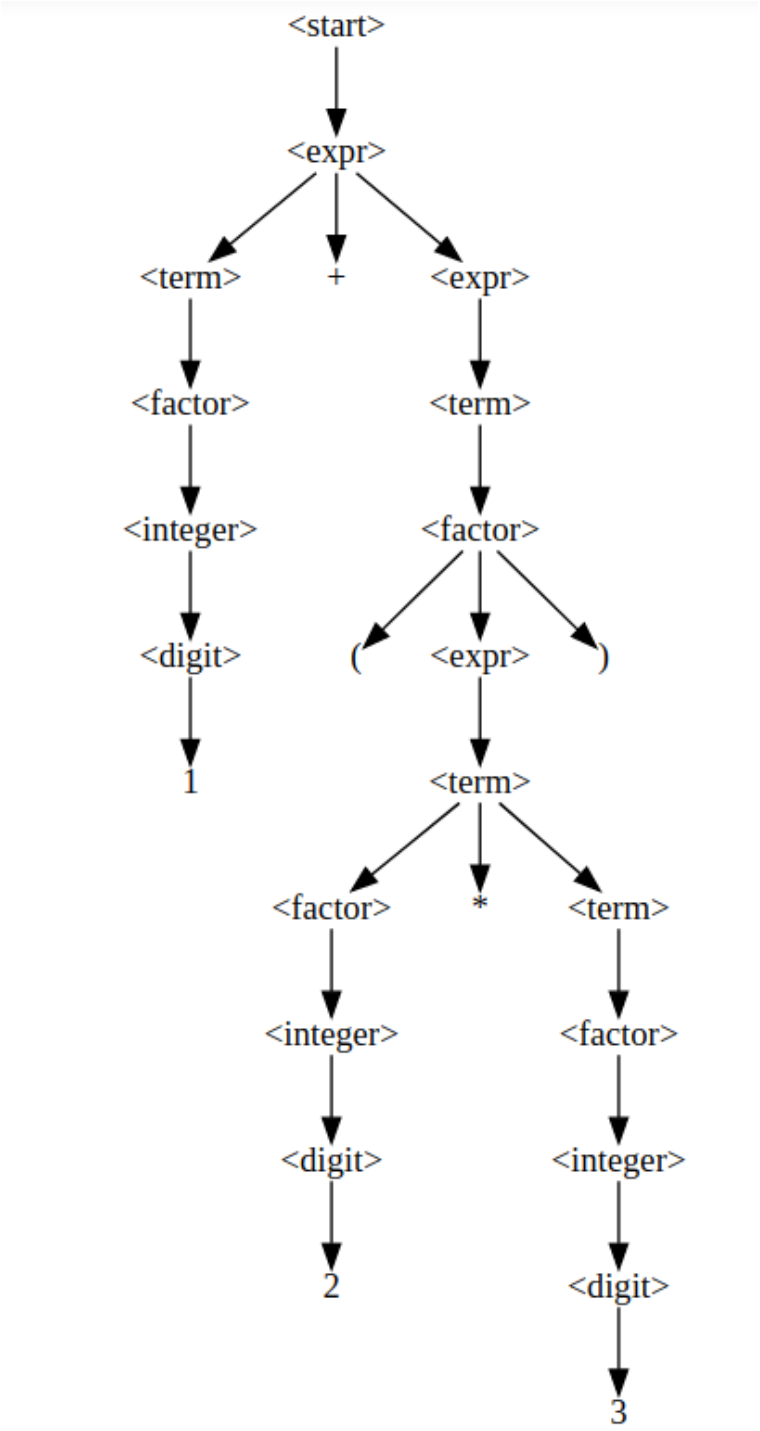

Рисунок 4: AST для выражения 1 + (2 * 3)

4. Сделать вывод о применимости методов fuzz-тестирования для тестирования шаблонизаторов.

2 Анализ предметной области fuzz-тестирования на основе грамматик

Базовый алогоритм[1] работы любого фаззера состоит из шагов:

1. Предобработка тестируемого программного обеспечения(далле SUT - Soft Under Test). Например, она может заключаться в компиляции со специальными флагами или подготовке окружения для тестирования.
2. Получить начальное множество входных данных и создать из него пул тестов.
3. Выбрать из пула один или несколько примеров и получить результат их тестирования в SUT.
4. Решить будет ли полезен этот тест в будущем.
5. Мутировать тест и добавить его в пул.
6. Перейти к шагу 3.

2.1 Получение начального множества тестов

Первый способ получения входных данных - использовать тесты, которые написали разработчики для SUT. 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 SUT. Минус этого способа - тесты разработчиков не всегда доступны.

Второй вариант - собрать пул тестов из примеров в интернете. Например, в случае fuzz-тестировании интерпретатора javascript можно в начальное множество добавлять примеры javasrcipt кода с гитхаба. Таким способом можно получить широкий пул тестов.

Третий - можно сгенерировать множество самостоятельно. В этом случае появляется возможность подтолкнуть фаззер в определённом направлении. Детали процесса генерации описаны ниже.

2.2 Генерация тестов

[1] https://www.fuzzingbook.org/html/Fuzzer.html

2.2.1 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 AST. Шаги алгоритма[1]:

1. Положить в корень дерева стартовую вершину и добавить 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 - часто будут генерироваться похожие тесты, медленно покрываются 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[1]. Если имеется какое-то множество примеров, то каждый из них представляется в виде AST и для каждого нетерминала для каждой продукции подсчитывется частота её встречаемости. По этим частотам можно вычисляется вероятность выбора каждой продукции. Чем чаще встречается переход в тестах, тем чаще он будет использоваться. Плюсы - этот метод позволяет направлять фаззер, путём изменения множества. Выбор символа становится более осмысленным с точки зрения программирования. Минусы - требуется начальное множество тестов.
- Алгоритм построения отличных[1]. Отличие этого способа от предыдущего - вероятность выбора становится обратно пропорциональна частоте встречаемости, то есть чем чаще встречается переход в тестах, тем реже он будет генерироваться. Это позволяет двигаться фаззеру в противоположном направлении. Плюсы - чаще будут встречаться неожиданные пути в грамматике. Минусы - чаще будут попадаться неинтересные символы. Например, в javascript часто будет вызываться return.

[1] https://www.fuzzingbook.org/html/Grammars.html#A-Simple-Grammar-Fuzzer

- Ещё один способ - задать вероятности пропорционально количеству возможных поддеревьев в продукции. Если в вершине потенциально много поддеревьев, она будет чаще выбираться. Плюсы - позволяет наиболее полно покрыть грамматику. Минусы - требуется предпосчёт количества деревьев для каждого нетерминала.

2.2.2 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 SUT, тем сложнее его анализировать, тем дольше будет его обработка.

- Чем тест сложнее и рекурсивнее, тем вероятнее найти на нём ошибку SUT.

Эти свойства не противоречат друг другу. Рассмотрим примеры для грамматики с рисунка 3. Выражение "5+5+5+5+5+5+5+5+5+5+5"является длинным, но не сложным. Скорее всего при парсинге будет вызвана функция обработки знака "+"несколько раз последовательно, что лишь увеличит время работы программы. Другой пример - выражение "(5 + (5) + ((5 + 5) + 5))". Оно короче предыдущего, но при обработке скобок некоторые функции будут вызваны рекурсивно, что увеличивает вероятность найти ошибку.

2.3 Получение обратной связи от запуска теста на SUT

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 - возникновение в SOT искомых исключений, например double free[1] в языке программирования c++. Зачастую такие ошибки означают, что найден баг и этот тест нужно дополнительно исследовать вручную.

- Покрытие кода(line coverage) и покрытие функций(function coverage) - подсчёт количества строк/функций, которые покрыл тест. В контесте 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.

[1] https://owasp.org/www-community/vulnerabilities/Doubly_freeing_memory

- Покрытие веток(путей) - более сложный вариант предыдущей метрики. При подсчёте учитывается последовательность выполнения строк кода или вызывов функций. Плюсы - покрытие веток гораздо более информативная метрика, чем предыдущая. Минусы - число веток растёт очень быстро с увеличением количества кода в SUT.

2.4 Оценка полезности теста

После запуска теста требуется узнать полезен ли этот тест или от него можно отказаться. Для этого нужно научиться сравнивать различные входные данные друг с другом. Для этого введём функцию $value : \mathbb{X} \to \mathbb{R}$, 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 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.

2.5 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.

2.6 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.

2.6.1 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 минмизации:

- Самый простой способ заключается в построении AST теста и поочерёдном удалении поддеревьев. Если после удаления поддерева, полезность теста не уменьшилась, то старый тест заменяется на новый, и продолжается процесс минимизации. Плюсы - неплохая скокорость работы. С небольшой вероятностью тест существенно укоротится. 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, оно заменяется 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 генерация тестов. Тогда этап уменьшения можно опустить.

2.6.2 Мутации

Мутации являются двигателем фаззера, позволяют ему эволюционировать. Если фаззер "застрял"на каком-то этапе, то мутации могут сделать шаг в сто-

рону и процесс fuzz-тестирования продолжится. Если тесты представимы в виде AST, то удобно описывать мутации изменениями над AST.

- Скрещивание тестов друг с другом. Берустя два успешных теста, представляются в виде AST, и поддерево одного теста заменяется на поддерево другого согласно правилам грамматики. Такое изменение, например, позволяет быстро увеличивать покрытие веток SUT.

- Создаётся пул поддеревьев. Для каждой вершинки в AST теста считается какая-то дополнительная информация. Поддерево этой вершинки может быть заменено только на поддерево из пула с такой же информацией. Например, информацией может быть тип нетерминала, количество идентификаторов. В пул тестов добавляются поддеревья 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 SUT, например, на создание вложенных циклов или рекурсии.

- Применимы стандартные AFL[1] 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.

[1] https://github.com/google/AFL

3 Анализ инструментов предметной области

Цель анализа - рассмотреть работу инструментов, которые применяются при fuzz-тестировании программ или решают задачу генерации входных данных. В обзоре рассмотрены самые цитируемые актуальные статьи с google scholar.

3.1 Superion

Superion[3] - 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 парсятся и из их поддеревеьев формируется множество для мутаций. Берётся не больше 10000 поддеревеьев 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. Тестирование проводилось на парсерах xml и интерпретаторах javascript.

3.2 Grammarinator

Grammarinator[5] - 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 премя генерации контролируется размер тестов. Для каждого символа грамматики производится предподсчёт 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 интерпертатора 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 - деревья из пула используются, как поддеревья генерируемых тестов.

3.3 Nautilus

Nautilus[6] - фаззер, сочетающий в себе способность генерировать входные данные по грамматике и ориентироваться 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:

   - Используется две стратегии уменьшения тестов. Минимизация 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, заменяя их по одной за раз. Из этих стратегий лучше работает первая.
   - К каждому тесту применяется одна из следующих 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.

3.4 EvoGFuzz

В EvoGFuzz[2] 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:
   $f(x) = \infty$, если тест упал с ошибкой
   $f(x) = expansions(x)^2/length(x)$,
   $expansions(x)$ - количество расширений(нетерминальных символов)
   $length(x)$ - длина теста в символах.
   Использование такой функции позволяет более сложным тестам получать большую полезность, а длинные тесты штрафуются. Плюсы: функция просто вычисляется, учитывает свойства, которые требуется 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.

3.5 Выводы

Для fuzz-тестирования шаблонизаторов предлагается использовать наиболее отличившиеся идеи. Среди будущих мутаций должны присутствовать изменение пероятностей на случайные, скрещивание и замена поддерева на случайное, потому что они выделяются среди остальных. Для генерации тестов можно использовать алгоритм построения похожих, вероятности вычислить по реальным программам. Для борьбы с семантическими ошибками можно использовать идеи из Grammarinator. Среди стратегий обработки тестов нет явно выделяющейся, но идеи, описанные в EvoGFuzz, мало изучены, предлагается исследовать именно их.

4 Анализ задачи

4.1 Уязвимости в шаблонизаторах

В мире веб-приложений шаблонизаторы часто используется для форматирования данных, полученных от пользователей. Рассмотрим типовые примеры атак и известных уязвимостей в шаблонизаторах.

4.1.1 Уязвимости типа XSS

Атака xss возможна, если данные полученные от пользователя некорректно обрабатываются и попадают в html-разметку. В качестве примера рассмотрим шаблон из рисунка 1 и представим, что поле *title* задаёт пользователь. Если переменная *title* содержаит строку $< script > alert(documnet.cookie) < /script >$ и у шаблонизатора отсутствует механизм защиты от xss, то появляется возможность украсть пользовательские куки. Стандартный механизм защиты от атак этоги типа - экранирование символов. Экранирование - это процесс замены опасных для безопасности символов на безопасные, которые могут быть безопасно отображены в документе, и является важной составляющей безопасности веб-приложений. Этот механизм реализуют многие современные шаблонизаторы, в некоторых эта опиция включена по умолчанию, в некоторых нужно дополнительно указывать необходимость экранирования. В первом случае шаблонизатор мможет выполнять экранирование символов, опираясь на контекст, в который попадают данные. Например, стандартный шаболонизатор языка(пакет html/template) Golang может распознавать контексты HTML, CSS, JavaScript и URL, и в них по-разному экранировать одну и ту же строку [2] (рисунок 5).

```
Context                          {{.}} After
{{.}}                            O'Reilly: How are &lt;i&gt;you&lt;/i&gt;?
<a title='{{.}}'>                O&#39;Reilly: How are you?
<a href="/{{.}}">                O&#39;Reilly: How are %3ci%3eyou%3c/i%3e?
<a href="?q={{.}}">              O&#39;Reilly%3a%20How%20are%3ci%3e...%3f
<a onx='f("{{.}}")'>             O\x27Reilly: How are \x3ci\x3eyou...?
<a onx='f({{.}})'>               "O\x27Reilly: How are \x3ci\x3eyou...?"
<a onx='pattern = /{{.}}/;'>     O\x27Reilly: How are \x3ci\x3eyou...\x3f
```

Рисунок 5: Пример экранирования строки O'Reilly: How are you? в разных контекстах

Примеры известных уязвимостей:

- Ошибка в экранировании контекстных переменных в шаблонизаторе Django[3], при использовании специального тега {% debug %}.

[2] https://pkg.go.dev/html/template#hdr-Contexts

[3] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-22818

- Уязвимость в шаблонизаторе Mako[4]. Ошибка в том, что в Mako для экранирования использовалась функция стандартной библиотеки языка программирования Python, которая некорректно обрабатывала символ кавычки '. Атакующий мог проэксплуатировать этот недостаток, воспользовавшись функцией onload для любого html-тега. Пример эксплойта - ' onload=alert(1) a='

- Уязвимость в шаблонизаторе Mustache[5]. Рекурсивный рендеринг в шаблонах Mustache, содержащих пользовательский ввод, в некоторых случаях мог привести к XSS.

4.1.2 Уязвимости типа SSTI

SSTI (Server-side template injection) - это уязвимость, которая возникает, когда злоумышленник может выполнить произвольный код на стороне сервера, используя уязвимости шаблонизатора.

Шаблонизатор Jinja подвержен такой уязвимости, если происходит вставка пользовательских данных в шаблон без предварительной проверки на потенциально опасные символы. Например, если в приложении есть возможность ввода данных пользователем, которые затем используются для формирования HTML-шаблона на сервере, то злоумышленник может вставить в поле ввода код, который будет выполнен на сервере. Рассмотрим пример такой уязвимости. В коде программы(рисунок 6), в формировании шаблона используются данные, полученные от пользователя. Если подставить в уязвимый параметр name строку " 1 + 1 то сложение выполнится.

```
if request.args.get('name'):
    name = request.args.get('name')
template = '''
    <!doctype html>
    <h1>Hello %s!</h1>
    <form>
        <input type="text" name="name">
    </form>
''' % (name)
return render_template_string(template)
```

Рисунок 6: Пример щаблона, узявимого к SSTI

[4]https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-2480
[5]https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-40313

4.1.3 Вывод

Уязвимость типа SSTI является следствием ошибки разработчика при использовании шаблонизатора. Такие ошибки не интересны с точки зрения безопасности самого шаблонизатора.

Уязвимость типа XSS возможна в одном из двух случаев:

- Экранирование не должно было быть выполнено - либо шаблонизатор не гарантирует экранирование, либо была использована специальная функция, отключающая этот механизм безопасности. Это неправильная конфигурация шаблонизатора, то есть с точки зрения программы ошибки тут нет.
- Шаблонизатор сконфигурирован правильно, но специальные символы не экранируются. Такие ошибки предлагается искать при помощи fuzz-тестирования.

4.2 Выбор шаблонизатора

Для тестирования был выбран стандартный шаблонизатор языка Golang[6]. В документации указано, что шаблонизатор является безопасным.[7] Как было отмечено ранее, шаблонизатор умеет различать контексты HTML, JS, CSS и URL. Механизм экранирования символов включён по умолчанию. На данный момент в шаблонизаторе не было найдено ни одной уязвимости. Шаблонизатор реализует довольно обширный список синтаксических конструкций, основные из них:

- Условный оператор if/else
- Оператор цикла range
- Встроенные функции сравнения данных, логические высказывание, получение значени по индексу и получение подмассива. Помимо встроенных, шаблонизатор позволяет создавать свои функции и использовать их в шаблоне.
- Возможность создавать переменные внутри шаблона и использовать их значения.

Таким образом, шаблонизатор реализует нужный механизм безопасности, который включён по умолчанию, и не требует при использовании никаких дополнительных действий.

Рассмотрим особенности шаблонизатора, которые нужно будет учитывать при разработке фаззера.

- Язык Golang является строго типизированным языком, и данные, которые обрабатывает шаблон также должны быть строго типизированны. Данные

[6] https://pkg.go.dev/html/template

[7] https://pkg.go.dev/html/template#hdr-Security_Model

```
<h1>{{.PageTitle}}</h1>
<ul>
    {{range .Todos}}
        {{if .Done}}
            <li class="done">{{.Title}}</li>
        {{else}}
            <li>{{.Title}}</li>
        {{end}}
    {{end}}
</ul>
```

(a) простой шаблон на языке Golang

```
data := TodoPageData{
    PageTitle: "My TODO list",
    Todos: []Todo{
        {Title: "Task 1", Done: false},
        {Title: "Task 2", Done: true},
        {Title: "Task 3", Done: true},
    },
}
```

(b) структура данных, для подстановки в шаблон

Рисунок 7: Пример работы с шаблонизатором Golang

в шаблон передаются одной структурой, обращаться к ним нужно через название полей структуры. Наример, конструкция в шаблоне {{ .Student.Name }} обращается к полю Student входной структуры, а затем к полю Name. Появляется проблема соответствия названий переменных в шаблоне и названий переменных в структуре, которая передаётся из кода программы. Предлагается следующее решение: для каждого стандартного типа данных зафиксировать название переменных, которые ему соответствуют. Но при этом добавить возможность иногда обращаться к несуществующим переменным для полноты покрытия(шаблонизатор должен корректно обрабатывать данную ситуацию и выдавать ошибку). Фиксирование названий не отразится на чистоте эксперимента, потому что по прежнему проверяется весь функционал шаблонизатора.

- Возможность создавать и запускать собственные функции. Задача сгенерировать всевозможные функции изначально не является выполнимой, так как их счётное число. Поэтому предлагается определить только две функции: первая возвращает одну из предварительно выбранных строковых констант, вторая просто возвращает свой первый аргумент.

5 Результаты

В рамках данной работы были получены следующие результаты:

- Сделан обзор методов fuzz-тестирования программ, принимающих на вход данные, порождаемые КС-грамматикой. Выделены критерии для сравнения таких фаззеров.
- Произведён анализ существующих инструментов. На его основе предложен алгоритм для fuzz-тестирования шаблонизаторов.

Список литературы

[1] Inputs from Hell Generating Uncommon Inputs from Common Samples [Электронный ресурс]. URL: https://arxiv.org/pdf/1812.07525.pdf (дата обращения: 18.05.2022)

[2] Evolutionary Grammar-Based Fuzzing [Электронный ресурс]. URL: https://arxiv.org/pdf/2008.01150.pdf (дата обращения: 18.05.2022)

[3] Superion: Grammar-Aware Greybox Fuzzing [Электронный ресурс]. URL: https://arxiv.org/pdf/1812.01197.pdf (дата обращения: 18.05.2022)

[4] Fuzzing With Optimized Grammar-Aware Mutation Strategies [Электронный ресурс]. URL: https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=9469897 (дата обращения: 18.05.2022)

[5] Grammarinator: A Grammar-Based Open Source Fuzzer [Электронный ресурс]. URL: Grammarinator (дата обращения: 18.05.2022)

[6] NAUTILUS: Fishing for Deep Bugs with Grammars [Электронный ресурс]. URL: Nautilus (дата обращения: 18.05.2022)

[7] The Fuzzing Book [Электронный ресурс]. URL: https://www.fuzzingbook.org/ (дата обращения: 18.05.2022)

[8] Systematically Covering Input Structure [Электронный ресурс]. URL: https://publications.cispa.saarland/2971/1/ase19-paper381-published.pdf (дата обращения: 18.05.2022)

[9] American fuzzy lop - a security-oriented fuzzer [Электронный ресурс]. URL: https://github.com/google/AFL (дата обращения: 24.05.2022)