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 шаблонизаторы, 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. Пример шаблона предоставлен на рисунке 1

```
1  <h1>{{.PageTitle}}</h1>
2  <ul>
3      {{range .Todos}}
4          {{if .Done}}
5              <li class="done">{{.Title}}</li>
6          {{else}}
7              <li>{{.Title}}</li>
8          {{end}}
9      {{end}}
10 </ul>
```

Рисунок 1: Пример шаблона на языке Golang

Движок шаблонизатора - программное обеспечение, разработанное для объединения шаблона и данных в финальный документ. При работе шаблонизатора сначала производится лексический и синтаксический 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 выражения, такие как условные операторы, циклы, вызовы функций. Шаблонизатор подставляет значения из доступных ему данных в обозначенные места в шаблоне. Пример того, как должна быть сформирована структура этих данных в программе предоставлен на рисунке 2. Шаблонизатору во время рендеринга шаблона доступна строковая переменная $PageTitle$, её значение он подставит в тег $h1$, и массив структур $Todos$. Конструкция $range$ в шаблоне означает перебор всех элементов массива. Для каждого элемента выведется его поле $Title$ в списке, при этом если значение поля $Done$ будет истинным, то в результат добавится дополнительно $class = "done"$.

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

(a) структура данны для подстановки в шаблон

```
<h1>My TODO List</h1>
<ul>
    <li>Task 1</li>
    <li class="done">Task 2</li>
    <li class="done">Task 2</li>
</ul>
```

(b) результат работы шаблонизатора

Рисунок 2: Пример работы шаблонизатора

1.1.2 Уязвимости в шаблонизаторах

В мире веб-приложений шаблонизаторы часто используются для работы с данными, полученными от пользователей. Рассмотрим типовые примеры атак и известных уязвимостей в шаблонизаторах.

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 предположем, что поле $PageTitle$ задаётся пользователем. Пусть переменная $PageTitle$ содержит строку

$$< script > alert(documnet.cookie) < /script >$$

Тогда если у шаблонизатора отсутствует механизм защиты от xss, то появляется возможность украсть пользовательские куки. Стандартный механизм защиты от атак этого типа - санитизация символов. Санитизация - это применение преобразования(экранирования или кодирования) к спецсимволу в зависимости от контекста подстановки. Экранирование символа - это добавление перед этим символом специальной конструкции, данные после которой интерпретируются безопасно с точки зрения контекста. Кодирование - процесс замены символа на его безопасный эквивалент, то есть символ будет интерпретирован, как данные, а не как какая-то управляющая конструкция. Механизм санитизации реализуют многие современные шаблонизаторы, в некоторых эта опция включена по умолчанию, в некоторых требуется дополнительно указывать необходимость санитизации. В первом случае шаблонизатор может выполнять санитизацию символов, опираясь на контекст, в который попадают данные. Например, стандартный шаболонизатор языка Golang(пакет html/template) может распознавать контексты HTML, CSS, JavaScript и URL, и в них по-разному изменять одну и ту же строку [1] (рисунок 3).

Примеры известных уязвимостей:

- Ошибка в кодировании контекстных переменных в шаблонизаторе Django - CVE-2022-22818[2], при использовании специального тега {% debug %}.

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

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

```
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 3: Пример санитизации строки O'Reilly: How are you? в разных контекстах

- Уязвимость в шаблонизаторе Mako - CVE-2010-2480[3]. Ошибка в том, что в Mako для санитизации 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 - CVE-2022-40313[4]. Рекурсивный рендеринг в шаблонах Mustache, содержащих пользовательский ввод, в некоторых случаях мог привести к XSS.

1.1.3 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[5]:

- 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.

[3] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-2480
[4] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-40313
[5] https://habr.com/ru/company/dsec/blog/517596/#chto-takoe-fazzing

В рамках поиска ошибок в шаблонизаторах наибольший интерес представляет подход с генерацией входных данных, так как каждый шаблон должен удовлетворять правилам движка, чтобы проходить тривиальные проверки при запуске.

На практике оказалось очень удобно задавать структуру входных данных с помощью грамматик (Пример грамматики на рис. 4). Если известна грамматика, то все возможные входные данные можно представить абстрактным синтаксическим деревом (далее AST - Abstract Syntax Tree). Пример AST представлен в приложении С. Это позволяет избегать синтаксических ошибок на этапе запуска 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 4: 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.
   - Оценить каждый критерий с точки зрения применимости к задаче fuzz-тестирования шаблонизатора.
   - 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

- Разработать фаззер для стандартного шаблонизатора языка Golang, 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.

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

2 Анализ задачи

2.1 Анализ исследуемой функциональности

Уязвимость типа XSS возможна в одном из двух случаев:

1. Неправильное использование шаблонизатора. Например, разработчик программы использовал функцию, которая не осуществляет санитизацию данных, или функцию, отключающую этот механизм безопасности. С точки зрения шаблонизатора ошибки тут нет. Корректное использование шаблонизатора - зона ответственности разработчика.

2. Шаблонизатор сконфигурирован правильно, то есть данные должны быть санитизированы, но этого не происходит. Например, шаблонизатор неверно определил тип контекста, в который подставляются данные, и санитизировал данные под неправильный контекст. Такие ошибки шаблонизатора предлагается искать при помощи fuzz-тестирования.

2.2 Особенности шаблонизатора Golang

В официальной документации указано, что шаблонизатор является безопасным.[6] Как было отмечено ранее, шаблонизатор умеет различать контексты HTML, JS, CSS и URL. Механизм санитизации 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, вычисления логических выражений, получения элементов массива по индексу и получения 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(механизм санитизации символов), который включён по умолчанию, и не требует при использовании никаких дополнительных действий.

Основные особенности шаблонизатора, которые нужно будет учитывать при разработке фаззера:

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

- Язык Golang является строго типзированным языком, и данные, которые обрабатывает шаблон также должны быть строго типзированы. Данные 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 структуры Student. Появляется проблема соответствия названий переменных в шаблоне и названий переменных в структуре, которая передаётся из кода программы.

- Возможность создавать и запускать собственные функции. Задача сгенерировать всевозможные функции изначально не является выполнимой, так как их счётное число.

2.3 Требования к фаззингу шаблонизатора

Исследуемый функционал - корректная санитизация символов шаблонизатором. Чтобы протестировать этот функционал, фаззер должен генерировать разнообразные синтаксически валидные тесты и иметь возможность проверки правильности санитизации. Для корректности создаваемых тестов должна быть решена проблема соответсвия названий и типов данных в шаблоне и в программе, обрабатывающей этот шаблон. Для полноты исследования функциональности санитизации требуется протестировать как можно больше вариантов различных контекстов и обрабатываемых в них данных.

3 Анализ предметной области fuzz-тестирования на основе грамматик

В основе анализа лежат энциклопедия фаззинга - fuzzingbook[7] и статьи с google scholar, в которых описываются современные подходы к fuzz-тестированию на основе грамматик. Fuzzingbook - сборник всех базовых идей и концепций, применяемых в fuzz-тестировании.

Согласно fuzzingbook, базовый алогоритм[8] работы любого фаззера состоит из следующих 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.

Ниже подробно описаны идеи, которые можно применять в каждом из пунктов. Обзор сформирован на основе источников - [1]-[8]. Для каждой идеи указано, из какой статьи она взята.

3.1 Получение начального множества тестов

Первый способ получения входных данных[3] - 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[3] - 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.

[7] https://www.fuzzingbook.org/

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

Третий[2] - можно сгенерировать множество самостоятельно. 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 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[6]. Плюсы - простая реализация. Минусы - часто будут генерироваться похожие тесты, медленно покрываюся 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.

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

- 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.

- Ещё один способ - задать вероятности пропорционально количеству возможных поддеревьев в продукции[6]. Если в вершине потенциально много поддеревьев, она будет чаще выбираться. Плюсы - позволяет наиболее полно покрыть грамматику. Минусы - требуется предподсчёт 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 SUT, тем сложнее его анализировать, и тем дольше будет его обработка[2].

- Чем тест сложнее и рекурсивнее, тем вероятнее найти на этом тесте ошибку в SUT[2].

Эти свойства не противоречат друг другу. Рассмотрим примеры для грамматики с рисунка 4. Выражение "5+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.

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

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

- Покрытие кода(line coverage)[3] и покрытие функций(function coverage)[9] - 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.

- Покрытие веток(путей)[9] - 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.

3.4 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(на основе статьи [2]):

- 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.

3.5 Способы обработки тестового множества

Существует две стратегии обработки тестового множества:

1. Первая, наиболее часто встречающаяся - обрабатывать каждый тест по отдельности[9]. Это позволяет распараллелить процесс fuzz-тестирования, что серьёзно его ускоряет.

2. Вторая стратегия основана на теории эволюции Дарвина[2]. 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 минимизации:

- Самый простой способ заключается в построении AST теста и поочерёдном удалении поддеревьев[6]. Если после удаления поддерева, полезность теста не уменьшилась, то старый тест заменяется на новый, и продолжается процесс минимизации. Плюсы - неплохая скорость 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[3]. 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[6]. 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 построения теста[5]. Этот способ применим, если используется генерация 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 AST, то удобно описывать мутации изменениями над AST.

- Скрещивание тестов друг с другом[4]. 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[3]. 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[6]. 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[6]. 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[2 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

4 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.

Для этого требуется оценить каждый из инструментов по критериям, описанным в предыдущем пункте, и сделать вывод о возможности переиспользования этого инструмента при разработке фаззера. Критерии:

1. Получение начального пула тестов
2. Способ генерации тестов
3. Метрики, получаемые от запуска теста
4. Оценка полезности теста
5. Стратегия обработки тестов
6. Мутации и способы улучшения тестов

Для каждого из пунктов требуется ответить на вопросы: как инструмент решает задачу, какая конкретно реализация используется.

4.1 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.

4.2 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.

4.3 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.

4.4 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.

4.5 Выводы

Для fuzz-тестирования шаблонизаторов предлагается использовать наиболее отличившиеся идеи. Среди будущих мутаций должны присутствовать изменение вероятностей 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.

5 Описание работы реализованного фаззера

Общую схему разработанного фаззера можно видеть на рисунке 5. Рассмотрим каждый из этапов подробнее.

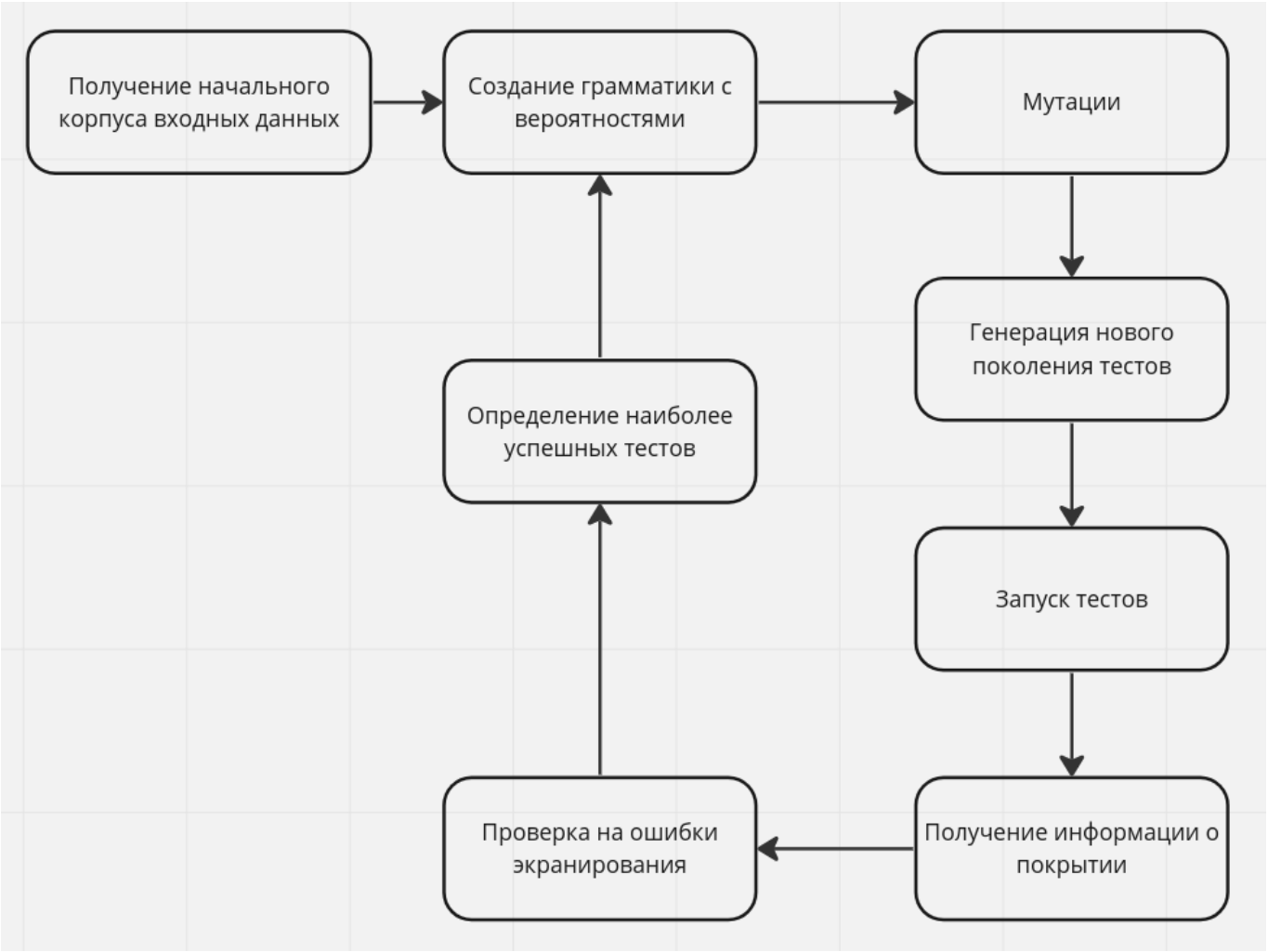

Рисунок 5: Схема работы фаззера

5.1 Получение грамматики шаблонизатора

Для работы фаззера необходима грамматика, по которой он будет создавать тестовые данные. За основу взята грамматика, найденная на гитхабе[9]. В процессе исследования в ней потребовались значительные преобразования для использования в фаззере. Во-первых, она была дополнена, чтобы соответствовать актульным возможностям шаблонизатора. Во-вторых, было заменено правило, которое генерировало произвольный текст между управляющими конструкциями шаблонизатора, на правило, которое создаёт различные контексты и места, в которых может встретиться уязвимость xss. Такие места были взяты из тестов инструмента DOMPurify[10].

[9] https://gist.github.com/bluven/054e59be17d758ccce76bdf432c9d08c
[10] https://github.com/cure53/DOMPurify/blob/main/test/test-suite.js

Для выявленной на этапе анализа задачи проблемы соответствия названий переменных в шаблоне и в структуре данных предлагается следующее решение: для каждого стандартного типа данных зафиксировать названия 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. Для реализации этой идеи в грамматике было изменено правило генерации названий переменных и функций с произвольных строковых значений на фиксированные.

5.2 Получение начального корпуса входных данных

Как было отмечено в результатах анализа, фаззер создаёт более интересные входные данные, если вероятности в грамматике соответсвуют реальным вероятностям использования переходов грамматики. Для начльного множества тестов были собраны 34 шаблона из настоящих проектов с сайта github[11]. Перед тем, как фаззер сможет обработать эти шаблоны, они проходят предварительную обработку: все символы между управляющими конструкциями заменяются на строку TEXT. Сами символы не имеют значения для фаззинга, важен только контекст, в котором будет выполняться логика шаблонизатора, а в 95% случаев в данных от пользователей - это обычный HTML контекст.

5.3 Создание грамматики с вероятностями

Входные данные для этого этапа - обычная грамматика, множество шаблонов и (опционально) грамматика с вероятностями из прошлой эпохи фаззинга. На выходе должна получиться новая грамматика с вероятностями. Фаззер действует следующим образом:

1. Нужно получить количество переходов для каждого из правил грамматики. Antlr4[12] - инструмент, который генерирует парсер и лексер по грамматике. При этом у него есть возможность встроить в парсер произвольный код на языке программирования python. Каждый из шаблонов парсится при помощи скрипта, сгенерированного инструментом antlr4, при этом происходит подсчёт количества переходов в каждом из нетерминалов грамматики.

2. Далее нужно преобразовать количество переходов в вероятности по формуле $probability(A => A_i) = \frac{cnt(A=>A_i)}{\sum_{i=1}^{N} cnt(A=>A_i)}$, где $probability(A => A_i)$ - вероятность перехода из нетерминала A в продукцию A_i, а $cnt(A => A_i)$ - количество таких переходов, полученное в пункте 1.

[11] https://github.com/search?l=Go&q=html%2Ftemplate&type=code

[12] https://github.com/antlr/antlr4

3. (опциональный шаг) Следующий шаг разработан в рамках данной работы на основании запусков фаззера. Иногда бывает полезно учитывать грамматику из предыдущих эпох, чтобы не упираться в локальные максимумы. Поэтому было принято решение попробовать не просто заново пересчитывать вероятности, а добавить в формулу вероятности из прошлой эпохи с коэффициентом λ(коэффициент устаревания). Тогда формула примет следующий вид: $probability(A => A_i) = \lambda \cdot \frac{cnt(A=>A_i)}{\sum_{i=1}^{N} cnt(A=>A_i)} + (1 - \lambda) \cdot old\_probability(A => A_i)$. Данное преобразование не нарушает свойств вероятности.

4. (Только для первой эпохи) Начальные вероятности для правил выбора названий переменных и функций, а также правил подстановки контекста выставлены вручную и примерно соответвуют равномерному распределению. Это связано с тем, что данные правила были добавлены в грамматику в рамках решения задачи, и нет данных, по которым можно было бы автоматически подсчитать их вероятности.

5.4 Мутации

Мутации применяются только если не было изменений в покрытии в течение двух или более эпох. В фаззере реализовано два типа мутаций, какая из них будет применена - выбирается случайным образом.

- Первая - изменение вероятностей в одной из строчек на случайные.
- Вторая - изменение всех вероятностей в грамматике на обратные, то есть генерация данных методом наименее похожих.

5.5 Создание шаблонов новой эпохи

Для генерации шаблонов по грамматике с вероятностями исползуется инструмент tribble[13]. Количество тестов и глубина их AST - параметры, которые указываются в инструменте. Значения этих параметров перебирались для поиска оптимальных.

5.6 Запуск тестов

Для запуска тестов помимо шаблона нужно сгенерировать код на языке Golang, который будет обрабатывать этот шаблон. При этом требуется сгенерировать структуру, которая будет подана на вход шаблонизатору, как пользовательские данные. Большинство полей структуры заполняется строковыми константами - векторами xss атак, при помощи которых в будущем будет определяться отсутствие санитизации символов. Эти векторы взяты из тестов инструмента

[13] https://github.com/havrikov/tribble

DOMPurify [14]. Для решения проблемы бесконечного числа функций, которые можно использовать в шаблоне, предлагается определить только две и в шаблоне генерировать только их вызовы: первая возвращает одну из предварительно выбранных строковых констант, вторая просто возвращает свой первый аргумент.

Дополнительно каждый тест запускается дважды - первый раз, как отдельный юнит-тест модуля html/template, второй раз - как отдельный юнит-тест модуля text/template. Таким образом, каждый шаблон запускается несколько раз с различными входными данными.

5.7 Получение информации о покрытии

Для получение информации о покрытии используется встроенные в язык программирования Golang методы[15]. Тест запускается командой go test -covermode count. После прогона теста создастся файл с информацией о том, какие строчки исследуемого кода были выполнены при запуске, и сколько раз это произошло. Покрытие всех запусков одного шаблона объединяется в один файл.

5.8 Поиск ошибок санитизации

В случае успешного выполнения теста, создастся html-документ, который нужно проверить на наличие необработанного вектора атаки xss. Для этого используется инструмент puppeteer[16]. Puppeteer - это высокоуровневая библиотека Node.js, разработанная командой Chrome для автоматизации и контроля над веб-браузером. Он предоставляет простой и удобный API для работы с Chromium (или Google Chrome) через сетевой протокол DevTools.Puppeteer обладает широким набором функций для автоматизации веб-браузера, включая навигацию по страницам, заполнение и отправку форм, создание скриншотов и PDF-файлов, эмуляцию устройств и многое другое. Кроме того, Puppeteer поставляется с функциональностью для исполнения страниц, которая позволяет эмулировать действия пользователя на веб-странице, такие как нажатие кнопок, ввод текста, скроллинг и т.д., что обеспечивает более реалистичное взаимодействие с веб-сайтом. Именно эта возможность используется в фаззере. Все векторы атак исполняют одну и ту же javascipt функцию - alert. При помощи puppeteer, html-документ открывается в headless-браузере, при этом отслеживается событие всплывающего окна - по сути вызов функции alert. Если событие произошло - значит найдена потенциальная уязвимость. Шаблон откладывается, чтобы можно было верифицировать уязвимость вручную.

[14] https://github.com/cure53/DOMPurify/blob/main/test/test-suite.js

[15] https://go.dev/blog/cover

[16] https://pptr.dev/

5.9 Отбор успешных шаблонов

Этот шаг нужен для того, чтобы сформировать список шаблонов, которые будут влиять на дальнейшую работу фаззера. Нужно оценить каждый из шаблонов на основании результатов запуска. Функция полезности шаблона зависит от покрытия кода и наличия ошибки санитизации, и выглядит следующим образом:

$$f(template) = \begin{cases} 1 & \text{если покрыты новые строчки кода или обнаружена уязвимость} \\ 0 & \text{в противном случае} \end{cases}$$

В новое поколение шаблон попадает тогда и только тогда, когда его функция полезности равна 1.

6 Эксперименты

Цель экспериментов - найти оптимальные параметры фаззера и сделать вывод о применимости методов fuzz-тестирования для тестирования шаблонизаторов.

6.1 Методика проведения экспериментов

У фаззера есть 3 основных параметра, для которых требуется найти оптимальные значения. Это максимальная глубина AST генерируемого шаблона, количество шаблонов в каждом поколении и коэффициент устаревания. Также требуется сравнить и оценить полезность применяемых мутаций. Оптимизируемая метрика - покрытие строк в процентах. Для поиска оптимальных параметров используется итеративный подход. Начальное приближение - глубина 10, количество шаблонов 10, коээфициент устаревания - 0.8. Далее осуществляется поиск оптимального значения для каждого из параметров по отдельности в следующем порядке: глубина, количество шаблонов в поколении, коэффициент. Для каждого возможного значения параметра фаззер работает 100 эпох, после чего фиксируется значение покрытия. Во время работы фаззера используются обе мутации. Среднее время одного прогона - 4 часа.

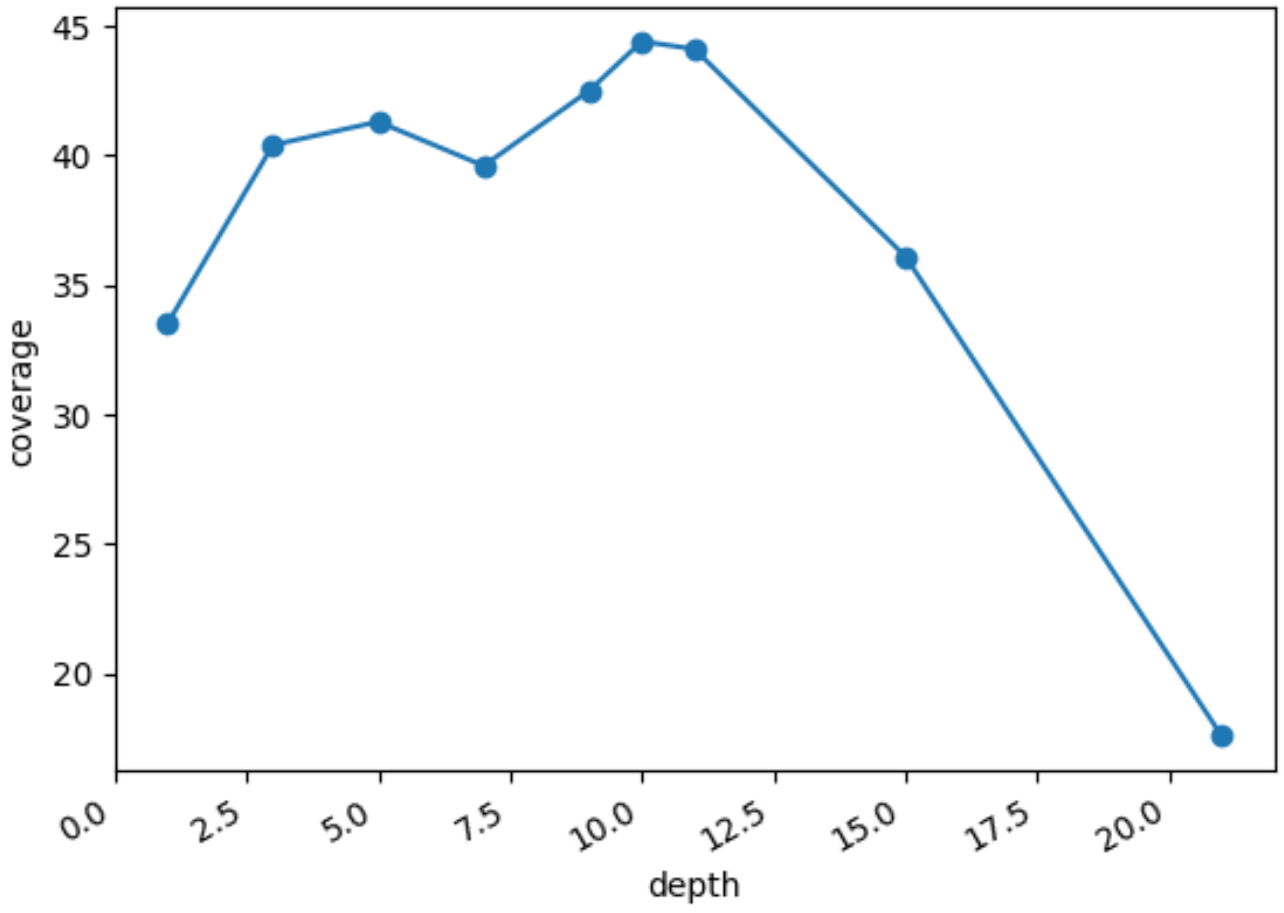

Рисунок 6: Зависимость покрытия от глубины AST

Оптимальное значение глубины генерируемых тестов - 10(рисунок 6). При такой глубине дерева, каждый шаблон содержит в себе несколько(2-3) контек-

стов и конструкций шаблонизатора. Если глубина меньше 10, генерируются недостаточно разнообразные шаблоны. Если больше 10, то возрастает вероятность ошибки обработки теста шаблонизатором.

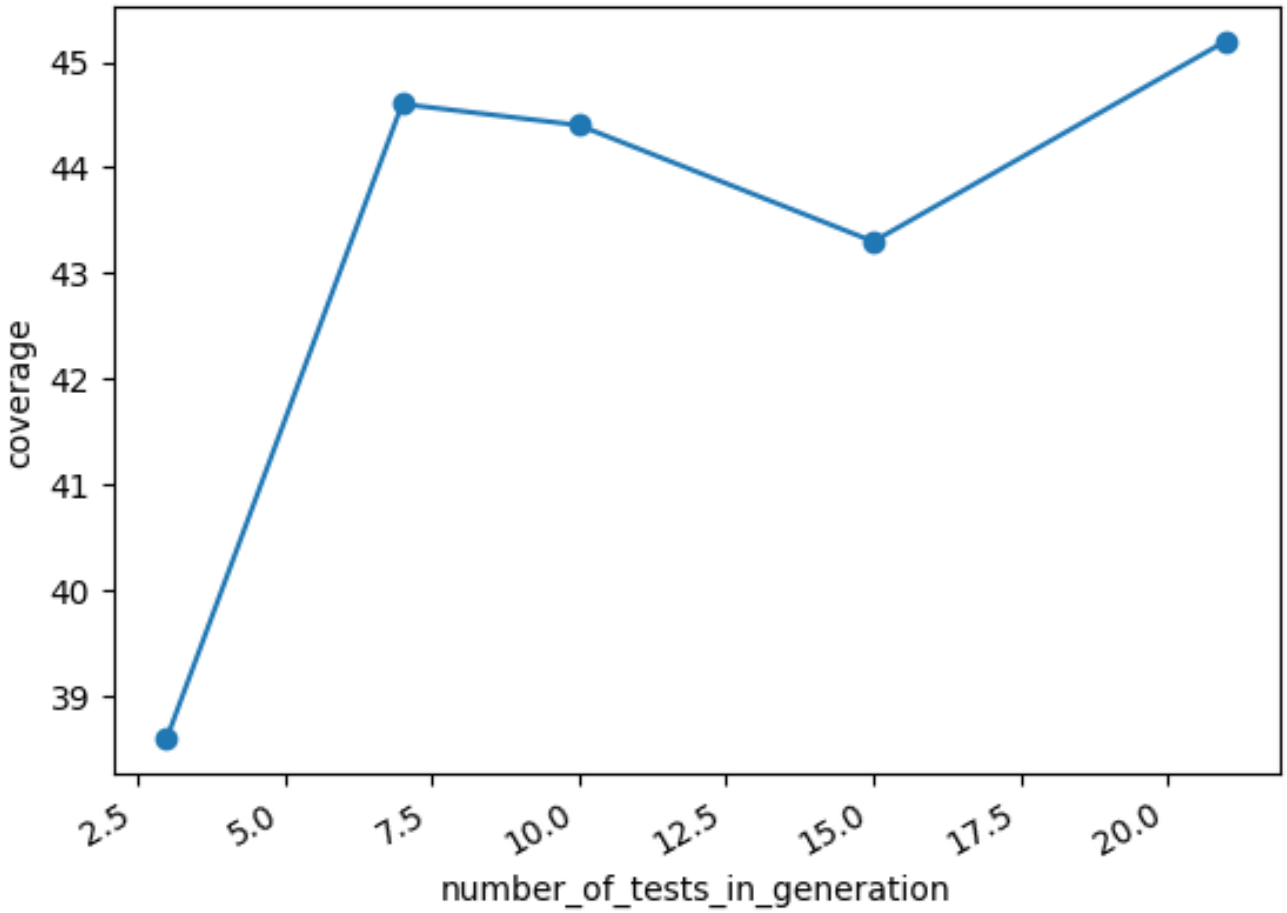

Рисунок 7: Зависимость покрытия от количества тестов в поколении

На графике(рисунок 7) видно, что метрика почти не изменяется, если тестов хотя бы 7. Самый выский показатель - 45.2% достигнут при количетсве тестов - 21. Это объясняется тем, что в этом случае за 100 эпох было запущено в 3 раза больше тестов по сравнению со случаем, когда в поколении 7 тестов. Времени было также потрачено в 3 раза больше. Таким образом, можно сделать вывод, что оптимальное значение шаблонов в поколении - 7.

Оптимальное значение коэффициента - 0.75 (рисунок 8). Действительно, при таком значении коэффициента, в сгенерированных тестах будет чаще срабатывать именно то правило грамматики, благодаря которому обнаружилось покрытие новых строк кода. В то же время остальные строчки частично сохранят вероятности прошлых эпох, что позволит и дальше генерировать разнообразные шаблоны.

На гистограмме(рисунок 9) демонстрируется сравнение эффективности разных мутаций. Наиболее полезна - мутация измененения вероятностей на случайные, так как она даёт наибольший прирост функции покрытия. Мутация изменения всех вероятностей в грамматике на обратные не даёт существенного улучшения. Наиболее оптимальный вариант - комбинировать эти мутации во время фаззинга, чаще используя первый тип.

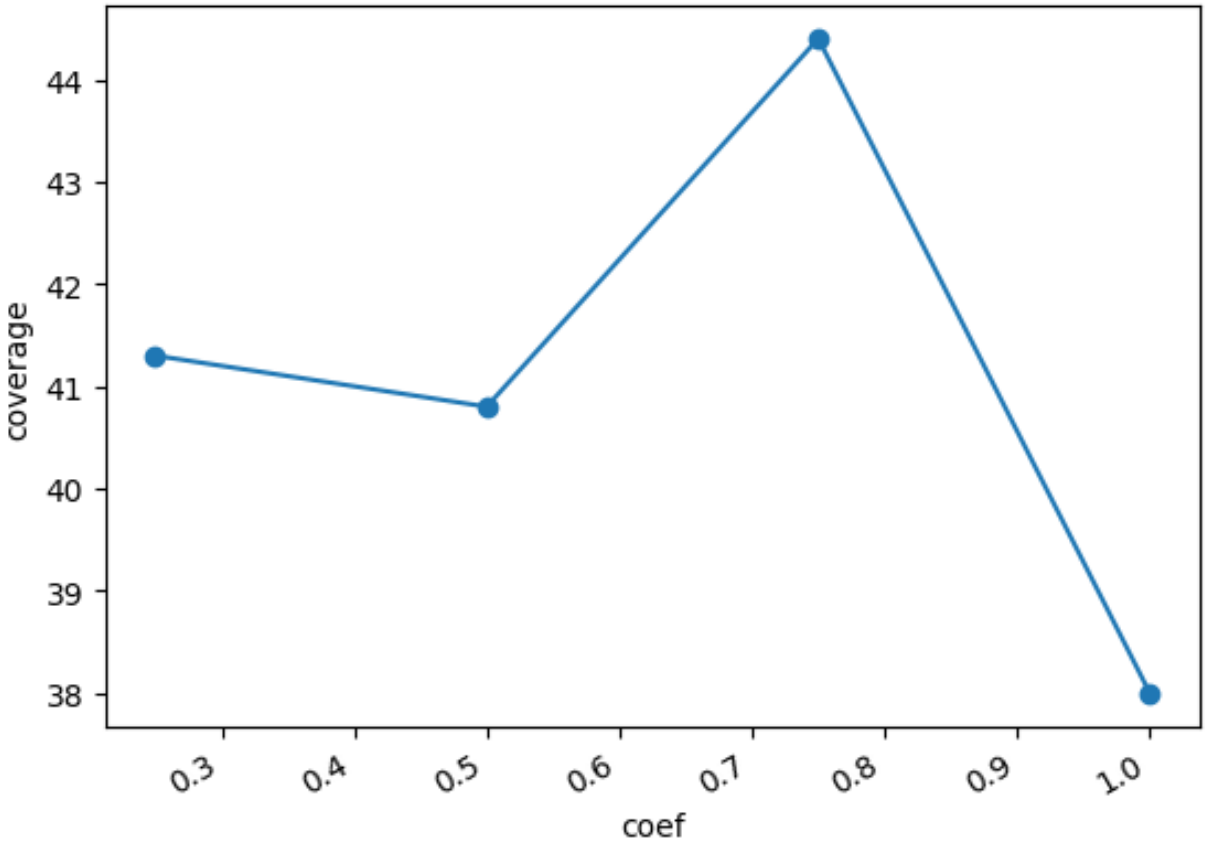

Рисунок 8: Зависимость покрытия от коэффициента устаревания

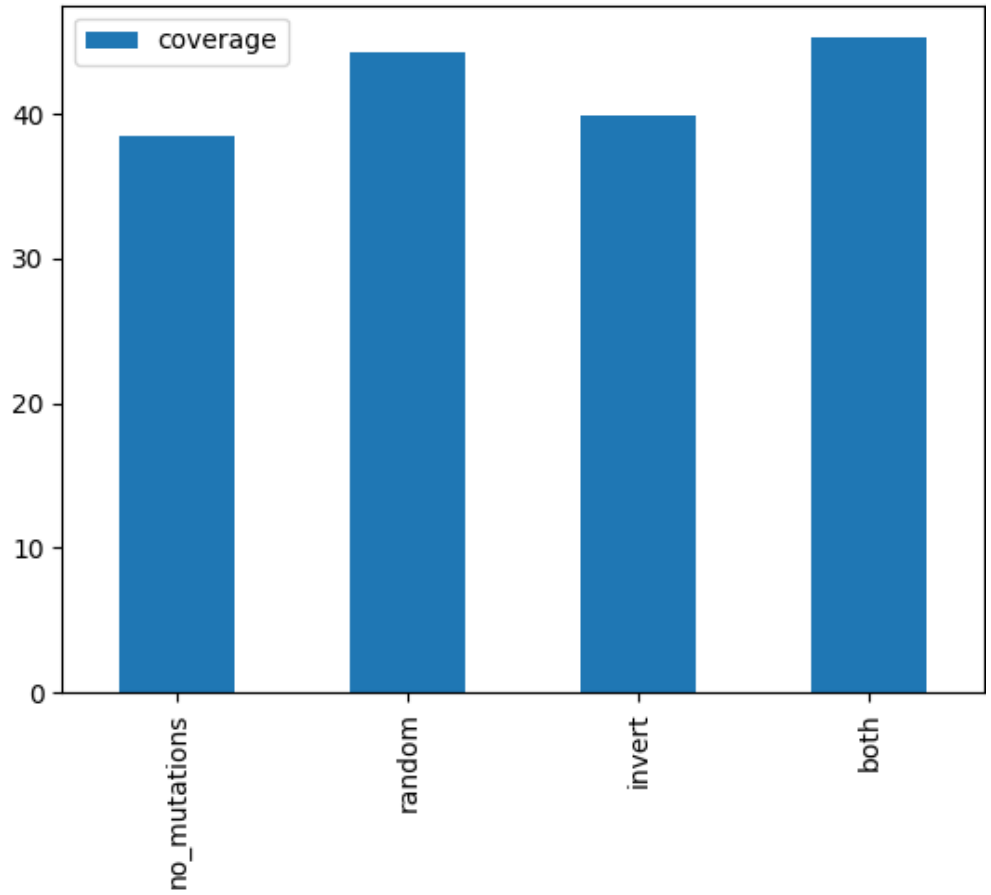

Рисунок 9: Зависимость покрытия от используемых мутаций

Затем было проведено несколько запусков на лучших параметрах. Самый успешный из них набрал 45.6% покрытия кода. Значение покрытия для каждой эпохи показано на рисунке 10

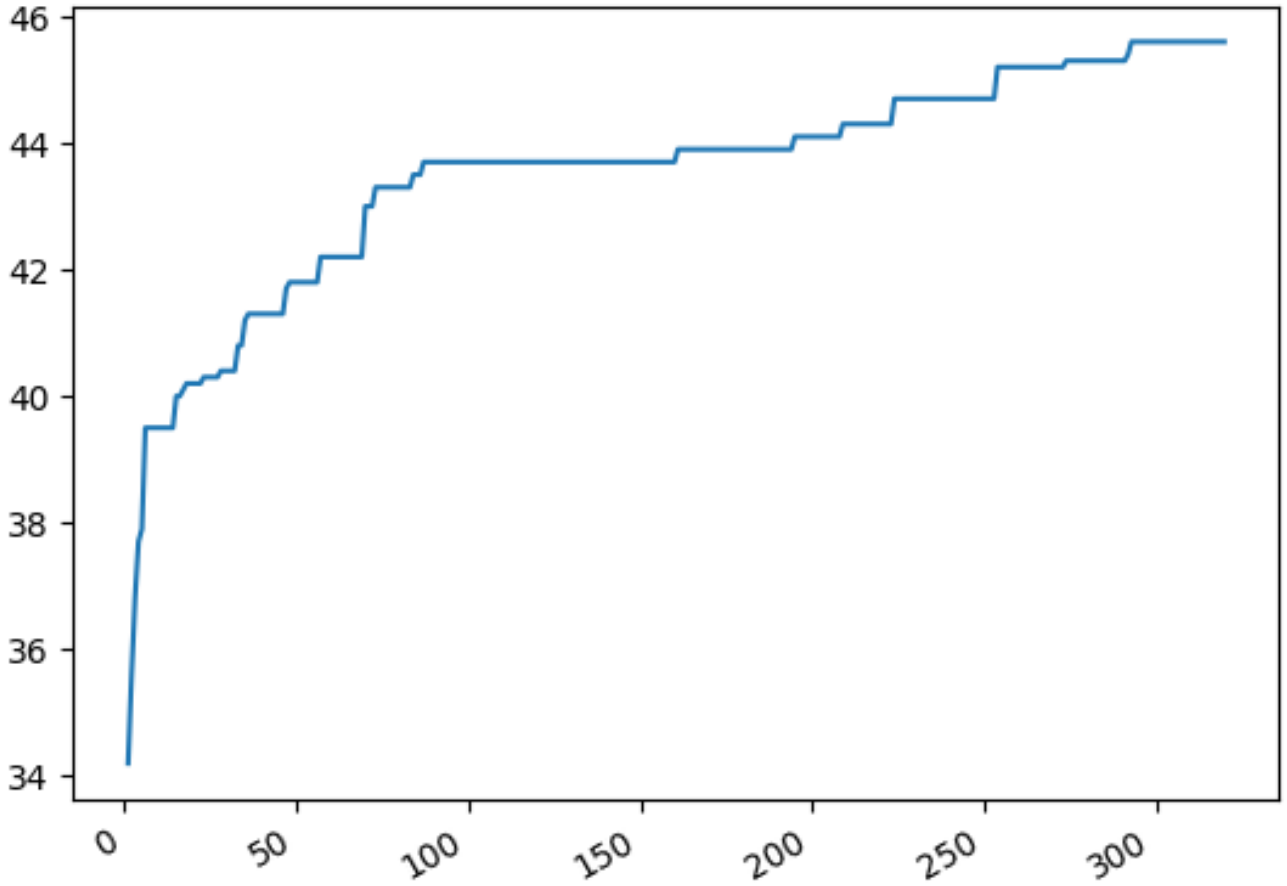

Рисунок 10: Лучший запуск

6.2 Найденные недостатки

Недостаток в функции template.Execute(wr io.Writer, data any)[17]. Если при выполнении шаблона или записи его выходных данных возникает ошибка, выполнение останавливается, но частичные результаты рендеринга шаблона будут записаны в буфер. Данный недостаток был автоматически обнаружен фаззером. Фаззер в процессе работы создаёт код на go, который использует эту функцию, записывая все шаблоны в один и тот же файл. Этот файл открывается для записи в начале программы. Важно отметить, что содержимое файла не очищается, а переписывается новым шаблоном. Первым шагом фаззер записал в файл шаблон, в которым все символы были корректно санитизированы. Затем в начало этого файла из-за ошибки при работе функции записалась только часть следующего шаблона. В результате в конце файла осталась последняя часть первого шаблона, что привело к срабатыванию функции alert, то есть возможной уязвимости. Данная ситуация маловероятна. Базовая рекомендация для разработчиков - не использовать буффер в случае ненулевой ошибки функции template.Execute.

[17] https://pkg.go.dev/html/template#Template.Execute

6.3 Анализ результатов

Покрытие кода оказалось на довольно низком уровне - 45.6%. После анализа участков кода, которых фаззеру не удалось достичь, было выявлено, что в пакетах text/template и html/template присутствуют вспомогательные функции, которые не являются объектами исследования. После их удаления, процент покрытия стал равен 50.8%. Если рассматривать только код, связанный с исследуемой функциональностью, а именно определением контекста и санитизацией символов, то покрытие равно 55.8%. Причины, по которым часть кода осталась непокрытой:

- Недостаточно разнообразное множество контекстов, встроенных в грамматику. Шаблонизатор гранулярно обрабатывает все символы в тесте, поэтому для более полного покрытия нужно добавлять контексты, основываясь на исходном коде программы. Например, не был протестирован код санитизации символов в параметре ссылки(в html-атрибуте href)

- Использование в значениях переменных только векторов атаки xss. Из-за этого, например, остался не исследован код, работающий с числовыми константами, который теоретически может влиять на ход исполнения программы.

В ходе тестирования не удалось найти существенных недостатков в функциональности санитизации символов в шаблонизаторе. Это может быть связано как с высокой квалификацией разработчиков, так и с недостаточным покрытием исходного кода.

6.4 Выводы

В целом, идея использования грамматики с вероятностями для создания шаблонов себя полностью оправдала. Сгенерированные тесты получаются довольно сложными. С грамматикой очень удобно работать и строить вокруг неё весь процесс тестирования. В результате fuzz-тестирования был покрыт весь код, достижимый данной грамматикой. Слабым местом фаззера стала генерация кода, который вызывает функцию обработки шаблона.

- Использование мутаций над грамматикой с вероятностями позволяет выбираться из точек экстремумов.

- Выбор начальных вероятностей по реальным программам позволяет достигать довольно высокого покрытия на ранних эпохах тестирования .

- Использование обратной связи существенно ускоряет процесс фаззинга и позволяет автоматически направлять фаззер в сторону ещё не исследованного кода

- Разработанный метод с коэффициентом устаревания позволяет эффективно комбинировать вероятности прошлой и текущей эпохи. То есть новые вероятности учитывают разнообразие тестов старой эпохи, и новое покрытие текущей.
- Процесс fuzz-тестирования полностью автоматизирован
- Разработанный фаззер работает крайне медленно. При стандартных параметрах на 100 эпох тратится примерно 5 часов. Каждый сгенерированного шаблон требуется открыть инструментом puppeteer, что очень сильно замедляет работу и является узким местом всего фаззера.

Объекты дальнейшего исследования:

- Улучшение грамматики, в частности добавление в неё новых контекстов.
- Использовать вместо метрики покрытия строк кода, метрику покрытия веток кода. Это позволит более точно управлять фаззером и анализировать его работу.
- Ускорить процесс проверки наличия ошибки санитизации.
- Добавление новых методов мутаций.

7 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 генерации входных данных. На его основе предложен алгоритм для fuzz-тестирования шаблонизаторов.
- Разработан полностью автоматический фаззер для стандартного шаблонизатора языка программирования Golang.
- Проанализирована работа фаззера.
- Сделаны выводы о применимости fuzz-тестирования для поиска ошибок в шаблонизаторе стандартной библиотеки языка Golang.

Список литературы

[1] Inputs from Hell Generating Uncommon Inputs from Common Samples [Электронный ресурс]. URL: https://arxiv.org/pdf/1812.07525.pdf (дата обращения: 05.05.2023)

[2] Evolutionary Grammar-Based Fuzzing [Электронный ресурс]. URL: https://arxiv.org/pdf/2008.01150.pdf (дата обращения: 05.05.2023)

[3] Superion: Grammar-Aware Greybox Fuzzing [Электронный ресурс]. URL: https://arxiv.org/pdf/1812.01197.pdf (дата обращения: 05.05.2023)

[4] Fuzzing With Optimized Grammar-Aware Mutation Strategies [Электронный ресурс]. URL: https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=9469897 (дата обращения: 05.05.2023)

[5] Grammarinator: A Grammar-Based Open Source Fuzzer [Электронный ресурс]. URL: Grammarinator (дата обращения: 05.05.2023)

[6] NAUTILUS: Fishing for Deep Bugs with Grammars [Электронный ресурс]. URL: Nautilus (дата обращения: 05.05.2023)

[7] The Fuzzing Book [Электронный ресурс]. URL: https://www.fuzzingbook.org/ (дата обращения: 05.05.2023)

[8] Systematically Covering Input Structure [Электронный ресурс]. URL: https://publications.cispa.saarland/2971/1/ase19-paper381-published.pdf (дата обращения: 05.05.2023)

[9] American fuzzy lop - a security-oriented fuzzer [Электронный ресурс]. URL: https://github.com/google/AFL (дата обращения: 05.05.2023)

[10] Golang documentation [Электронный ресурс]. URL: https://pkg.go.dev/ (дата обращения: 05.05.2023)

[11] Using type inference to make web templates robust against XSS [Электронный ресурс]. URL: https://rawgit.com/mikesamuel/sanitized-jquery-templates/trunk/safetemplate.html#problem_definition (дата обращения: 05.05.2023)

[12] The Definitive ANTLR 4 Reference [Электронный ресурс]. URL: https://pragprog.com/titles/tpantlr2/the-definitive-antlr-4-reference/ (дата обращения: 05.05.2023)

A Грамматика с вероятностями

```
start = template @@ 1.0;
template = (Text @@ 0.5908346972176759 | GoAction @@ 0.40916530278232405) (Text @@ 0.5908346972176759 | GoAction @@ 0.40916530278232405)* ;
GoAction = CommentAction @@ 0.012 | PipelineAction @@ 0.636 | IfAction @@ 0.204 | RangeAction @@ 0.052 | TemplateAction @@ 0.084 | BlockAction @@ 0.0 | WithAction @@ 0.0 | defineAction  @@ 0.012 ;
defineAction = LD Define TemplateName RD  @@ 1.0;
CommentAction = LD CommentBegin AnyText CommentEnd RD  @@ 1.0;
PipelineAction = LD ( Pipeline @@ 0.9937106918238994 | VarDeclarePipeline @@ 0.006289308176100629 | VarAssignPipeline  @@ 0.0) RD  ;
IfAction = LD If Pipeline RD template ( LD Else (If Pipeline)? RD template )? End  @@ 1.0;
RangeAction = LD Range ( Pipeline @@ 0.46153846153846156 | VarDeclarePipeline  @@ 0.5384615384615384) RD template ( LD Else RD template)? End  ;
TemplateAction = LD Template TemplateName (Pipeline)? RD  @@ 1.0;
BlockAction = LD Block Name (Pipeline)? RD  @@ 1.0;
WithAction = LD With Pipeline RD template ( LD Else RD template)? End  @@ 1.0;
Pipeline = argument ( Pipe argument )* @@ 0.9877049180327869 | MethodCall @@ 0.012295081967213115 | FuncCall  @@ 0.0 ;
VarDeclarePipeline =  Variable (Comma Variable)? Assignment Pipeline  @@ 1.0;
VarAssignPipeline =  Variable Equal Pipeline  @@ 1.0;
argument = Constant @@ 0.0 | Nil @@ 0.0 | Variable @@ 0.018587360594795539 | Fields @@ 0.8884758364312267 | FuncCall @@ 0.048327137546468404 | (LeftParenthesis argument RightParenthesis Fields? )  @@ 0.0037174721189591076 ;
Variable = Dollar Name @@ 1.0;
Fields = (Variable)? Dot Name ( Dot Name)?  @@ 1.0;
FuncCall =  globalFunctions @@ 1.0;
globalFunctions = (And argument argument) @@ 0.23076923076923078 | Index argument (argument)* @@ 0.07692307692307693 | Slice argument (argument)* @@ 0.0 | Len argument @@ 0.07692307692307693 | Not argument @@ 0.0 | Or argument argument @@ 0.07692307692307693 | Printf argument (argument)* @@ 0.
0 | Eq argument argument @@ 0.15384615384615385 | Ne argument argument @@ 0.3076923076923077 | Lt argument argument @@ 0.0 | Le argument argument @@ 0.0 | Gt argument argument| Ge argument argument @@ 0.07692307692307693 | RandomString @@ 0.0 | Print argument  @@ 0.0 ;
MethodCall = ( Variable @@ 0.0 | Fields  @@ 1.0) ( argument )*  ;
End = LD end RD  @@ 1.0;
LD = BlockStart (Dash " ")?  @@ 1.0;
RD = (Dash)?  BlockEnd  @@ 1.0;
Text = Href @@ 0.3| Default @@ 0.1 | Img @@ 0.1 | Style @@ 0.2 | Js @@ 0.3 ;
Style = "<style>" GoAction "</style>" @@ 1.0;
Js = "<script>" GoAction "</script>" @@ 1.0;
Img = "<img src=xx:" GoAction ">" @@ 1.0;
Default = "<br>" GoAction "</br>" @@ 1.0;
Href = AStart GoAction AEnd | HrefStart GoAction HrefEnd @@ 1.0;
AStart = "<a href=\"#\">" @@ 1.0;
AEnd = "</a>" @@ 1.0;
HrefStart = "<a href=\"" @@ 1.0;
HrefEnd = "\">abc</a>" @@ 1.0;
Template = "template " @@ 1.0;
Block = "block " @@ 1.0;
With = "with " @@ 1.0;
CommentBegin = "/*" @@ 1.0;
CommentEnd = "*/" @@ 1.0;
If = " if " @@ 1.0;
Else = " else " @@ 1.0;
Range = " range " @@ 1.0;
Pipe = "|" @@ 1.0;
Comma = "," @@ 1.0;
Assignment = ":=" @@ 1.0;
Equal = "=" @@ 1.0;
Nil = "nil" @@ 1.0;
LeftParenthesis = "(" @@ 1.0;
RightParenthesis = ")" @@ 1.0;
Dollar = "$" @@ 1.0;
Dot = "." @@ 1.0;
end = "end" @@ 1.0;
BlockStart = "{{" @@ 1.0;
Dash = "-" @@ 1.0;
BlockEnd = "}}" @@ 1.0;
Constant = "CONSTANT" @@ 1.0;
Define = "define " @@ 1.0;
And = " and " @@ 1.0;
Eq = " eq " @@ 1.0;
Lt = " lt " @@ 1.0;
Le = " le " @@ 1.0;
Ne = " ne " @@ 1.0;
Gt = " gt " @@ 1.0;
Ge = " ge " @@ 1.0;
Len = " len " @@ 1.0;
Not = " not " @@ 1.0;
Or = " or " @@ 1.0;
RandomString = " random_string " @@ 1.0;
Print = " print " @@ 1.0;
Index = " index " @@ 1.0;
Slice = " slice " @@ 1.0;
Printf = " printf " @@ 1.0;
AnyText = "Comment " @@ 1.0;
TemplateName = " \"keking_template_name\" " @@ 1.0;
Name = "A " @@ 0.3 | "B " @@ 0.28 | "C " @@ 0.20 | "LocalName " @@ 0.02 | "D " @@ 0.20;
```

В Пример сгенерированного шаблона

```
<style>
{{ if .D }}
<br>{{.A }}</br>
{{end}}
</style>
{{ and .B .A }}
<script>{{.D }}</script>
<img src=xx:{{ if .A }}<style>
{{.B }}</style>{{end}}>
<style>{{.C }}</style>
{{ if .B }}{{.A }}{{ else }}
<a href="{{template  "keking_template_name" }}">abc</a>
{{end}}
{{.A }}
<br>{{.C }}</br>
<a href="{{.A }}">abc</a>
```

C Пример AST для выражения 1 + (2 * 3)