Министерство науки и высшего образования Российской федерации
Федеральное государственное автономное образовательное учреждение высшего образования
«Казанский (Приволжский) Федеральный Университет»
Институт вычислительной математики и информационных технологий
Кафедра системного анализа и информационных технологий

Направление подготовки: 02.03.02 – Фундаментальная информатика и информационные технологии
Профиль: Системный анализ и информационные технологии

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

СИСТЕМА ФАЗЗИНГА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ НА ОСНОВЕ ЭВОЛЮЦИОННОГО ПОДХОДА

Обучающийся 4 курса
группы 09-931 (Редькин В.С.)

Руководитель
ст. преподаватель (Долгов Д.А.)

Заведующий кафедрой системного анализа
и информационных технологий
д-р техн. наук, профессор (Латыпов Р.Х.)

Казань – 2023

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Кибербезопасность стала областью с постоянно растущими бюджетами с обеих сторон – и с точки зрения убытков, понесённых компаниями от кибератак, и с точки зрения затрат на защиту и исследования в области информационной безопасности. Несмотря на большую роль человеческого фактора при проведении многих атак, классические методы, построенные на эксплуатации уязвимостей в программном обеспечении не теряют своей актуальности из-за возможности в случае обнаружения уязвимости в распространённой информационной системе проведения автоматизированных атак на большое число целей. Например, обнаруженная в 2017 году уязвимость cloudbleed, вызывавшая утечку данных из-за ошибки в html-парсере в сервисе Cloudflare, которым пользуются порядка 80% сайтов сети Интернет [1].

Фаззинг – подход к исследованию программы на наличие уязвимостей, заключающийся в автоматической генерации тестовых примеров и наблюдении за поведением программы на сформированных образцах данных с целью обнаружения ошибок работы с памятью, зависаний и другого интересного для исследователя поведения.

Цель настоящей работы - создать систему фаззинга программного обеспечения, использующую основные принципы генетических алгоритмов, которая не требует для своей работы модификации исследуемой программы.

Основные задачи, выполнение которых необходимо для достижения поставленной цели:

– разработать компонент системы, реализующий мутацию входных данных;

– разработать подсистему, осуществляющую трассировку выполняемой программы;

– протестировать систему на уязвимых образцах исполняемых файлов.

1. Генерация входных данных

Для формирования входных данных выделяют два подхода:

– на основе символьного исполнения, заключающийся в построении системы уравнений на основе условий, которые необходимо выполнить для прохождения конкретного пути в программе;

– генеративный, заключающийся в применении простых операций вроде инверсии битов или копирования и удаления сегментов данных для формирования новых образцов.

Далее будут более детально рассмотренны описанные подходы.

1.1. Символьное исполнение

Символьное исполнение – подход, полагающийся на построение и решение систем уравнений на основе условий, встречающихся при прохождении того или иного пути в программе. Рассмотрим следующую программу на языке Python:

```
x = int(input())       # 1
if x > 2:
    print(”big”)       # 2
while x > 0:           # 3
   print(”decrease”)   # 4
   x -= 1

print(”done”)          # 5
```

Если представить каждую точку, в которой происходит изменение потока управления, в виде узла графа, а участки программы, в которые мы можем попасть из текущего, соединить рёбрами, получим граф. В этом графе путь из начальной вершины в конечную будет соответствовать некоторому выполнению программы. Граф, соответствующий программе выше, приведён на рисунке 1.

Далее, рассмотрим какой-нибудь путь в программе, например 1 - 3 - 4 - 3 - 5. Выпишем логические условия, которые нам встречаются на переходах, и преобразования над данными ($\neg(x > 2)$ & $x > 0$ & $x' = x$ & $\neg(x' > 0)$) и решим систему, таким образом выяснив, что необходимое значение $x = 1$.

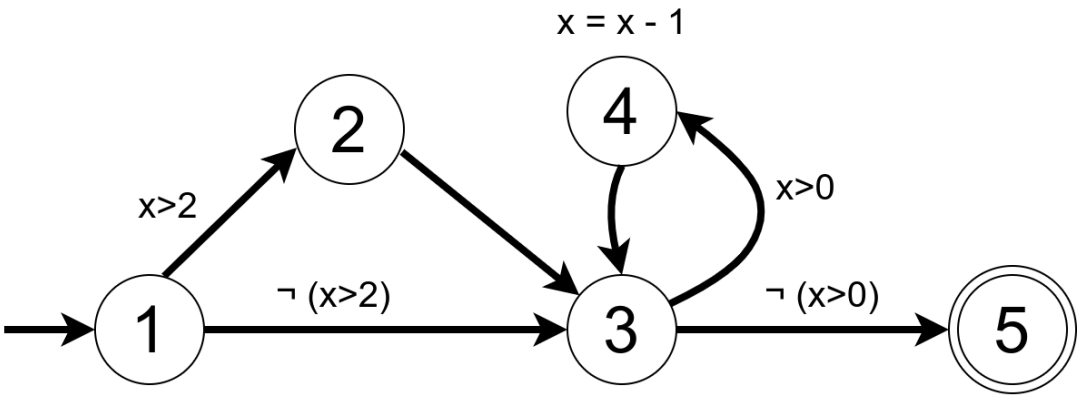

Рисунок 1 – Граф, соответствующий программе

Таким образом возможно явно перебирать пути выполнения, решая для каждого из них систему уравнений и получая приводящий к выбранной траектории пример данных. Данный подход нашёл применение, например, в системе KLEE, применяющей символьное исполнение с измерением покрытия для автоматической генерации тестовых примеров [2]. Данная система реализует символьное исполнение в виде своеобразного интерпретатора биткода llvm для поиска исчерпывающего набора тестовых примеров. Причём, полученные таким образом тесты часто способны опережать в плане покрытия кода тесты, написанные человеком, обладающим знаниями о структуре исследуемой программы.

Одной из проблем исследования программ посредством символьного исполнения является экспоненциальный рост пространства возможных путей в программе, а также высокая сложность решения полученных в результате символьного исполнения уравнений. В связи с этим на практике символьное исполнение редко применяется, вместо него в большинстве систем используется случайная генерация тестовых примеров. Но одна из особенностей символьного исполнения – явное выделение путей в программе – оказывается очень полезной для исследования более простыми методами, позволяя значительно повысить их эффективность за счёт отбора изменений, приводящих к повышению покрытия программы, что будет рассмотренно в главе 2.

1.2. Генеративный подход

Генеративный подход, иногда именуемый ”умным рандомом”, состоит в применении к существующим образцам данных простых операций, в подавляющем большинстве случаев работающих случайным образом, в надежде получить образцы, на которых программа проявит новое поведение. Для применения данного подхода большое значение имеет начальный набор образцов, из которого фаззер может отбирать участки данных.

Поскольку

Не смотря на свою простоту, этот подход зарекомендовал себя как стандарт в индустрии, так как его применение возможно в условиях отсутствия знания о структуре программы или формате данных, ожидаемых ею на входе.

1.3. Генерация при помощи грамматики

Часто генеративный подход сталкивается с проблемами при работе с программами, вход которых имеет строгую структуру. Например, если мы фаззим интерпретатор языка программирования, подавляющее большинство полученных в результате работы фаззера образцов данных, полученных на основе случайных мутаций, будут отбраковываться модулями лексического и синтаксического анализа, что может привести к неизмеримо большому количеству безуспешных запусков программы.

Более совершенной разновидностью генеративного подхода, пригодной для работы с структурированными данными, является генерация на основе грамматики. Фаззеру на вход подаются правила, задающие общую структуру данных, а вместо простых операций вроде инверсии битов применяется случайный выбор продукций грамматики, в результате чего получаем синтаксическое дерево. Свернув терминальные узлы синтаксического дерева, получим последовательность байт, которую уже можно подавать на вход программе.

Например, рассмотрим следующую грамматику, описывающую математические выражения:

$$Root \rightarrow Number \mid Root\ Operator\ Root$$
$$Number \rightarrow regex(\text{"}0|[0-9]\backslash d+\text{"})$$
$$Operator \rightarrow \text{"}+\text{"} \mid \text{"}-\text{"} \mid \text{"}*\text{"} \mid \text{"}/\text{"}$$

где " + " – терминал, описываемый строкой, а $regex(string)$ – терминал, соответствующий заданному регулярному выражению. Генерацию ввода по грамматике можно проводить следующим образом рекурсивно: находясь в нетерминале N, которому соответствует правило $N \rightarrow E_1 \mid ... \mid E_n$, случайным образом равновероятно выбрать одно из правил вывода E_i, заменить нетерминал N на последовательность терминалов и нетерминалов, и для каждого нетерминала в полученной последовательности операцию повторить. Пример генерации дерева продемонстрирован на рисунке 2.

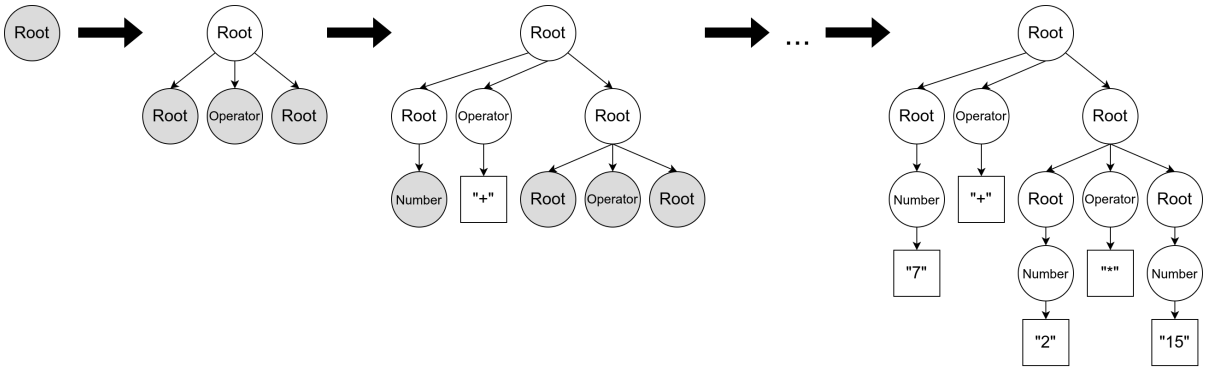

Рисунок 2 – Генерация дерева случайным применением продукций

При этом нам может потребоваться ввести ограничение на глубину результирующего дерева. В таком случае стоит дополнительно учитывать текущий уровень вложенности, а генерацию дерева ограничить конечным числом попыток. При рассмотрении очередного нетерминала проверим, что оставшийся запас вложенности не нулевой, и в случае, если это не так, посчитаем попытку генерации неудачной и сообщим об этом на уровень выше, попытавшись применить другое правило, а успешной будем считать попытку, в результате которой все нетерминалы в правой части правила вывода были успешно сгенерированы.

После генерации дерева необходимо свернуть его в последовательность байт, которую можно подать программе. Для этого можно также использовать рекурсивный обход, сворачивая поддеревья слева направо, в результате чего мы выпишем все терминалы. Например, для рассматриваемого дерева (рисунок 3) терминалами будут ”7”, ”+”, ”2”, ”*”, ”15”, которые свернутся в строку ”7+2*15”.

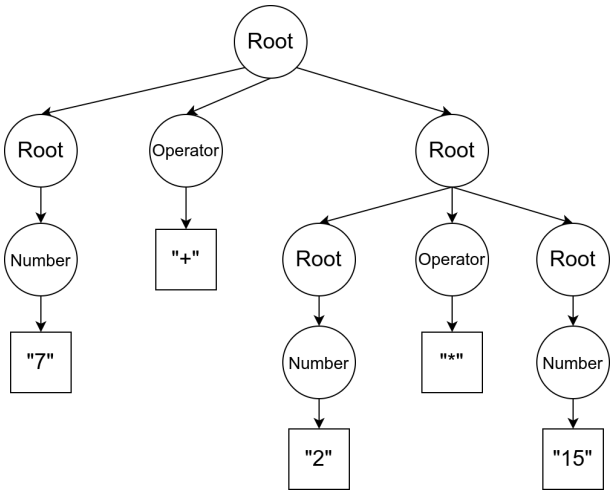

Рисунок 3 – Дерево, построенное по заданной грамматике

1.4. Мутации дерева

Введение промежуточного представления позволяет реализовать новые, более высокоуровневые мутации, способные учитывать структуру ввода. Если мы сохраним в дереве информацию о том, какое правило вывода было использовано для генерации того или иного поддерева, станет возможным, во-первых, регенерировать в существующем дереве его участки, просто стирая поддерево с корнем в некотором узле и заново применяя правило продукции случайным образом; во-вторых, мы получим возможность сохранять и в дальнейшем повторно использовать удачно сгенерированные образцы деревьев соответствующих типов, которые привели к выявлению нового поведения программы.

Далее, для обнаружения ошибок в парсере мы можем позволить с небольшой вероятностью производить вставку дерева, являющегося результатом применения продукции, которая не ожидается в данном участке

входа, дабы не создавать жёстких ограничений на пространство поиска фаззера, а скорее рассматривать заданные правила как рекомендации.

Наконец, возможно сочетание с классическими мутационными алгоритмами за счёт применения мутаций к терминальным вершинам в синтаксичеком дереве. Кроме того, в случае, если нам необходимо обнаруживать ошибки в самом парсере исследуемой программы, мы можем добавить возможность применять примитивные мутации, работающие на уровне байт, к элементам дерева, которые грамматика не даёт мутировать, например константы или ключевые слова. Для этого для каждого терминала будем дополнительно хранить информацию об области, которой он соответствует в итоговом свёрнутом виде, а при применении мутации найдём все терминалы, к которым она применяется и изменим их. Такой подход позволит сохранить байтовые мутации при работе с деревьями в привычном виде. Пример подобного можно увидеть на рисунке 4 – модификация применяется к соответствующим участкам литералов B и C.

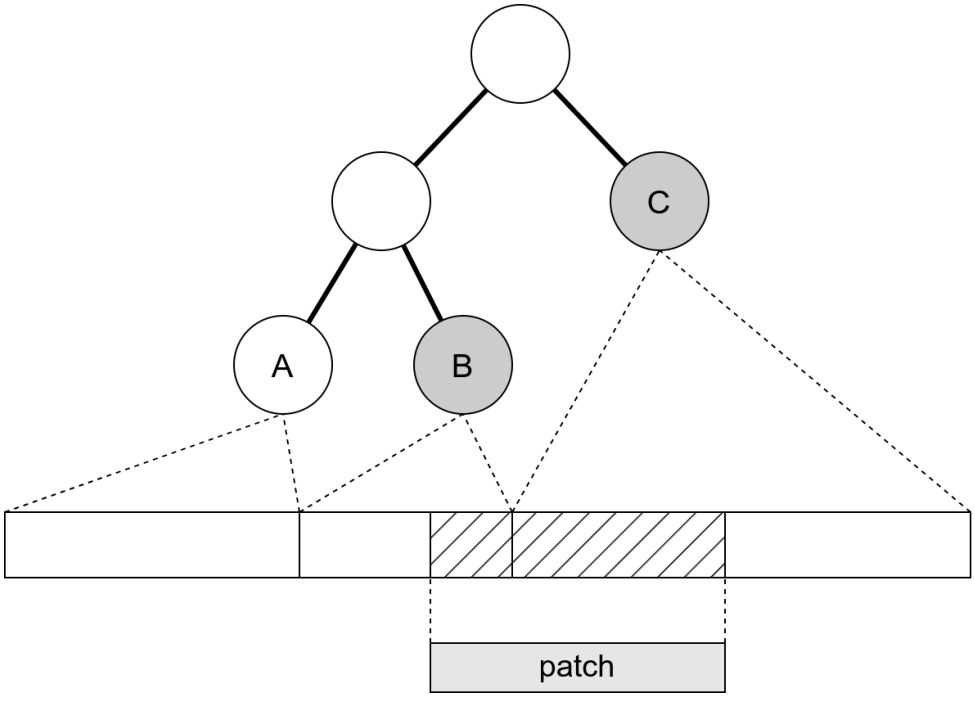

Рисунок 4 – Модификация терминалов на основе диапазонов

1.5. Генетические алгоритмы

Одним из подходов к решению задач поиска является применение генетических алгоритмов. Данный подход сводится к введению метрики (fitness function), и поиска решения, её улучшающего, за счёт применения к популяции вариантов решения мутаций (случайных изменений, выполняемых как правило при формировании новых образцов) и скрещивания (рекомбинация частей решений-кандидатов). Метрика указывает, насколько хорош или близок к оптимальному конкретный вариант решения задачи, но за счёт добавления в неё дополнительных слагаемых также можно вводить мягкие ограничения на сложность решений, например ограничивать глубину дерева при помощи штрафов. Генетические алгоритмы оказываются полезны в задачах, которые допускают неточное, приближённое решение.

Генетические алгоритмы оказываются полезны в том числе и в фаззинге. Примером подобного подхода является iFuzzer, предназначенный специально для фаззинга интерпретаторов языков программирования [3]. Данный фаззер руководствуется описанием грамматики целевого языка программирования для генерации тестовых примеров и использует подходы генетических алгоритмов – fitness function, мутацию и кроссинговер – для выбора существующих и генерации новых образцов, за счёт чего он способен создавать разнообразные образцы корректных с точки зрения целевого парсера программ. Применение fitness function позволяет ограничить разрастание генерируемых образцов за счёт введения штрафов, зависящих от размера программы, таким образом стремясь к генерации примеров наименьшей длины и наиболее разнообразной популяции.

Также для поддержания семантической корректности при мутации данный фаззер использует подход с переиспользованием литералов, заключающися в ограничении выбора имён переменных из уже

существующих в синтаксическом дереве и переименовании переменных при модификации синтаксического дерева в процессе мутации. Это становится возможным благодаря явной разметке в грамматике участков, являющихся именованными сущностями.

2. Трассировка

Важным компонентом, значительно ускоряющим процесс фаззинга, является измерение покрытия кода программы при запуске очередного тестового примера. Существует несколько подходов для контроля исследуемой программы и измерения покрытия, они будут рассмотрены далее.

2.1. Статическая инструментация

Статическая инструментация программы, полагающаяся на применение специальных библиотек и компиляторов, добавляющих в программу инструкции, на которые затем ориентируется фаззер для точного выяснения траектории выполнения программы.

Плюсом такого подхода является быстрота проведения фаззинга (например, в программе может быть искусственно выделена та или иная секция, подвергаемая тестированию в бесконечном цикле, за счёт чего отпадает необходимость в трате ресурсов на постоянный запуск новых процессов и загрузки библиотек).

Минус данного подхода состоит в необходимости наличия доступа к исходному коду программы и необходимости дополнительной работы, заключающейся в подключении специальных заголовочных файлов, выделении тестируемых участков программы, а также компиляции при помощи специальных инструментов.

Одним из фаззеров, использующих статическую инструментацию, является American Fuzzy Lop, или коротко afl [4]. Данная инструмент предоставляет большой набор подходов, позволяющих сделать фаззинг быстрее и эффективнее:

– afl-gcc и afl-clang – специальные компиляторы, предназначенные для генерации исполняемых файлов с дополнительной инструментацией, используемой фаззером. Помимо прочего, они могут производить

дополнительное действия по ”укреплению” (hardening) исполняемых файлов, что позволяет более эффективно обнаруживать ошибки в работе с памятью;

– afl-trim

2.2. QEMU mode

В тех случаях, когда доступа к исходному коду программы нет, необходимо прибегать к другим методам, если мы хотим иметь возможность использовать покрытие программы. Одним из вариантов, применяемых в том числе и в afl, является запуск программы в эмуляторе. Исполняемый файл запускается в легковесном эмуляторе, а при выполнении инструкций, отвечающих за изменение потока выполнения программы, например инструкций `jmp` или `call`, информация об текущем значении указателя инструкций записывается, за счёт чего мы можем построить путь в пространстве состояний программы. Фаззер afl реализует данный подход при помощи модифицированного эмулятора QEMU, при этом происходит эмуляция только кода в пространстве пользователя, а выполнение системных вызовов и взаимодействие с ядром происходит как обычно.

Преимуществом данного подхода является его универсальность – мы больше не нуждаемся в доступе к исходному коду и не обязаны выполнять компиляцию специализированными инструментами, фаззер можно применить к любому исполняемому файлу. Кроме того, за счёт применения эмулятора у нас появляется возможность проводить фаззинг исполняемых файлов, скомпилированных для архитектуры процессора, отличающейся от таковой на нашей машине, например фаззить программы под arm64 на компьютерах с процессорами на архитектуре intel.

Очевидной проблемой, возникающей при применении эмулятора, является негативное влияние на скорость выполнения программы, что особенно важно при генеративном подходе. В случае использования afl и QEMU замедление оказывается в 2-5 раз. Для снижения этого влияния

авторы afl предлагают в том числе механизм fork server, сводящийся к модификации исполняемого файла за счёт внедрения перед точкой входа небольшого фрагмента машинного кода, выполняющего в бесконечном цикле fork для создания собственных копий. Это позволяет за счёт механизма copy on write в linux выполнить загрузку динамических библиотек только один раз и таким образом увеличить долю времени, затрачиваемую на выполнение исследуемого кода.

2.3. Динамическая инструментация

Динамическая инструментация программы полагается на использование методов, схожих с таковыми, применяемыми в отладчиках - для сбора информации о траектории выполнения программы применяются точки останова, в которых записывается состояние регистра счётчика команд. Мы точно также можем работать с уже готовым исполняемым файлом и не иметь исходного кода исследуемой программы.

Проблемой динамической инструментации является серьёзное влияние на скорость выполнения программы, вызванное необходимостью обрабатывать большое число прерываний и системных вызовов при общении между исследуемой программой и программой-трассировщиком, из-за чего время выполнения увеличивается пропорционально числу попадания указателя инструкций на точку останова.

Для снижения этого влияния могут применяться различные методы, например Coverage-guided tracing [5]. Данный подход предлагает вместо создающего серьёзную вычислительную нагрузку полного отслеживания траектории выполнения выявлять только факт посещения новых, ранее не обследованных участков программы. В данном случае мы исходим из предположения, что львиная доля тестовых примеров не вносит вклада в обнаружение новых участков программы, а вместо этого проходит по уже известным путям, и процент таких примеров по мере исследования программы увеличивается, а вероятность обнаружить непосещённый участок

снижается.

2.4. Трассировка при помощи ptrace

Для контроля выполнения программ операционная система Linux предоставляет системный вызов ptrace [6]. Он позволяет:

– читать память дочернего процесса через `ptrace(PTRACE_PEEKDATA, addr)` и регистры при помощи `ptrace(PTRACE_GETREGS)`, за счёт чего мы получаем возможность наблюдать состояние программы, например отслеживать состояние регистра указателя инструкций;

– модифицировать память и регистры дочернего процесса при помощи `ptrace(PTRACE_POKEDATA, addr, data)` и `ptrace(PTRACE_SETREGS, addr, data)`, за счёт чего мы получаем возможность управлять выполнением программы, а также модифицировать её код прямо во время выполнения;

– продолжать выполнение дочернего процесса, выполняя `ptrace(PTRACE_CONT)` для просто запуска, `ptrace(PTRACE_SYSCALL)` для запуска процесса до выполнения им системного вызова и `ptrace(PTRACE_SINGLESTEP)` для запуска с выполнением единственной инструкции. Последний вариант в том числе позволяет обрабатывать точки останова через добавление в код программы инструкций, вызывающих прерывания, что используется в дебаггерах, и в том числе пригодится в дальнейшем.

Основное применение данного системного вызова – реализация с его помощью дебаггеров и других инструментов, например инструментов аудита, отслеживающих системные вызовы. Типичная последовательность использования ptrace выглядит следующим образом:

1) родительский процесс (tracer) создаёт свою копию при помощи `fork`;

2) созданный дочерний процесс, возможно, выполняет некоторый код

инициализации и сообщает, что хочет стать целью отслеживания через `ptrace(PTRACE_TRACEME);`

3) созданный дочерний процесс выполняет запуск целевого исполняемого файла через ещё один системный вызов `execve`;

4) родительский процесс ожидает сообщений от дочернего процесса через системные вызовы вроде `waitpid`. На основании полученной информации родительский процесс понимает, в каком состоянии находится дочерний, взаимодействует с ним и отдаёт команды о продолжении выполнения.

Применяя `ptrace`, мы можем отслеживать процесс выполнения программы с нужным уровнем гранулярности путём расстановки точек останова в начале функций или базовых блоков. Алгоритм обработки точек останова выглядит следующим образом:

1) программа-трассировщик выбирает участки кода, в которых выставляются точки останова. Это можно сделать, при помощи анализа исполняемого файла, добывая информацию из таблицы символов для обнаружения функций или анализируя непосредственно код программы, тем самым выявляя инструкции, изменяющие поток выполнения програмы;

2) исследуемый исполняемый файл загружается в память через `fork` и `execve`, как было описано выше, и ставится на паузу;

3) байт инструкции, расположенный по интересующему нас адресу заменяется на байт $0xCC_{16}$, что соответствует ассемблерной инструкции `INT 3`, вызывающей прерывание точки останова, а прежнее значение по этому адресу запоминается;

4) когда все инструкции прерываний расставлены, дочерний процесс запускается, программа-трассировщик ожидает появления прерываний;

5) при возникновении прерывания трассировщик может узнать, где оно возникло, из регистра счётчика команд. Выполнив необходимую логику, программа-трассировщик заменяет команду на ранее сохранённое значение;

6) происходит запуск дочерней программы в режиме выполнения одной инструкции, за счёт чего происходит выполнение кода, изначально заложенного в программу, после чего управление снова передаётся трассировщику;

7) трассировщик снова заменяет байт на `INT 3`, тем самым возвращая точку останова в программу, после чего можно продолжить выполнение дочерней программы.

2.5. Преимущества и ограничения

Трассировка при помощи точек останова является лишь одним из доступных подходов, и данный подход имеет как преимущества, так и недостатки. Из плюсов, выгодно выделяющих его на фоне других ранее перечисленных можно выделить:

– отсутствие необходимости в модификации исследуемой программы, и за счёт этого более низкий порог входа. Благодаря отсутствию необходимости применения инструментов вроде специального компилятора или работы с исходным кодом мы получаем систему, которую достаточно запустить на уже имеющемся у пользователя исполняемом файле;

– приемлемая скорость выполнения. За счёт запуска программы средствами, предоставляемыми самой операционной системой, мы имеем скорость выполнения, по большому счёту совпадающую с таковой при запуске той же программы пользователем. Единственный источник замедления – обработка системных вызовов и межпроцессного взаимодействия при работе с `ptrace`.

В то же время можно выделить ряд недостатков и ограничений:

– необходимость проведения, возможно довольно сложного, анализа исследуемого файла. В том случае, если мы хотим иметь покрытие на уровне базовых блоков, нам необходимо проводить анализ всего исполняемого машинного кода;

– вытекающая из предыдущего пункта более низкая универсальность.

Анализ, который мы можем провести, не всегда будет способен полноценно извлечь заложенный в программу функционал, и не позволит работать с программами, скомпилированными при помощи инструментов обфускации, в которых может, например, применяться модификация исполняемого кода или переходы по адресам, не совпадающим с адресами выделенных во время анализа инструкций (поскольку довольно большая доля байтовых последовательностей могут интерпретироваться как команды в наборе команд x86 с учётом поддержки инструкций переменной длинны и того факта, что платформа x86 не накладывает требований на выравнивание инструкций);

– проблемы при работе со статически слинкованными исполняемыми файлами – фаззер просто не будет знать, какая часть программы является важной и должна тестироваться и будет уделять большую долю времени анализу библиотечного кода вместо интересующей нас пользовательской части. Эта проблема частично разрешается предоставлением пользователю возможности указывать области исполняемых файлов, в которые будут добавляться точки останова, но это в свою очередь означает дополнительный анализ и конфигурацию со стороны пользователя.

Резюмируя, можно сказать, что выбранный способ трассировки предоставляет ряд преимуществ, но конечному пользователю стоит знать и о недостатках, ввиду которых в некоторых случаях, вероятно, стоит сделать выбор в пользу другого решения. Тем не менее стоит отметить, что фаззер покрывает основной пользовательский сценарий – написанные на языках C или C++ программы, для которых не проводилась обфускация и выполнялась динамическая линковка.

3. Выбор исходных образцов и стратегий мутации

При применении генеративного подхода решающую роль играет выбор образцов, которые станут источником для генерации, а также самих стратегий мутации. Например, применение битфлипов при фаззинге интерпретатора не даст пройти стадию корректно работающего лексического анализа, но в то время эта же стратегия может помочь обнаружить ошибки в парсере изображений. Для выбора исходных образцов необходимо выделение уникальных путей в программе и предпочтение мутации образцов, дающих наибольшее покрытие. При выборе стратегий мутации оказываются полезны некоторые методы решения задач о принятии решений в условии неопределённости, в частности для задачи о многоруких бандитах. Далее будут рассмотрены эти вопросы.

3.1. Выбор образцов на основе покрытия

Частой проблемой при фаззинге является медленное исследование отдалённых участков программы, связанное с тем, что фаззер не обладает информацией о том, каким путём может быть достижим тот или иной её участок и в следствие этого вынужден выбирать, какой образец мутировать случайным образом. Часто применяемой эвристикой для борьбы с этим является предпочтение тех образцов, которые имеют наибольшее покрытие программы в надежде, что их мутирование позволит продвинуться дальше.

Фаззеры формулируют пути в программе по-разному. Afl в частности вместо просто понимая покрытия как количества посещённых функций, базовых блоков или других структурных элементов рассматривает программу как конечный автомат и отслеживает переходы между его состояниями. При этом ещё и выполняется подсчёт каждого типа переходов и происходит разделение числа переходов на эмпирически подобранные классы эквивалентности, за счёт чего, с одной стороны, мы получаем, например, возможность отыскивать нередко приводящие к неожиданному поведению

двойное исполнение блоков кода, и с другой стороны не проводим различий между примерами, при обработке которых используются многократно работающие циклы.

Аналогично afl предлагается использовать выделение классов эквивалентности, выделяя однократно, двукратно и многократно посещённые участки программы. Путь в программе представляется как словарь, ключами в котором являются адреса посещённых участков программы, а значениями – классы эквивалентности. Данный подход, помимо прочего, позволяет убирать из программы точки останова после попадания адреса в класс многократно выполняющихся участков после выполнения трёх и более раз, за счёт чего влияние на скорость выполнения программы будет ограниченно числом точек останова и не будет постоянно возрастать, но в то же время мы по-прежнему имеем возможность узнать, был ли достигнут новый её участок.

Выбор кандидата на мутацию происходит с вероятностью, задаваемой простой формулой:

$$p_i = \frac{n_i}{\sum_{k=1}^{m} n_k},$$

где n_i это число уникальных точек в программе, посещённых при запуске i-го примера, а p_i - вероятность выбора i-го примера.

3.2. Задача о многоруких бандитах

Задача о многоруких бандитах является одной из классических задач принятия решений в условиях неопределённости, а также обучении с подкреплением и находит применение в разных областях, например в медицине как потенциально более эффективный способ поиска лекарств или в рекламе для выбора наиболее подходящих рекламных баннеров [7]. В своём классическом виде задача формулируется следующим образом: имеется K случайных распределений с математическими ожиданиями μ_i и стандартными отклонениями σ_i, соответствующих игральным автоматам

(отсюда название). На каждом ходу игрок выбирает один из автоматов и тянет за ручку, тем самым получая некоторый выигрыш, определяемый соответствующим распределением. Целью игрока является максимизация выигрыша. При этом параметры распределений изначально неизвестны, игроку доступна лишь оценка этих параметров на основании своих предыдущих выборов и наблюдавшихся результатов, перед игроком стоит выбор между использованием какого-то автомата, который он считает наилучшим, и исследованием остальных распределений в попытке собрать больше информации и потенциально найти более хороший вариант.

Если проводить параллели с генерацией данных для фаззинга, то вместо игровых автоматов мы выбираем среди набора стратегий мутации, а в качестве выигрыша рассматривается то, как много участков программы удалось покрыть сгенерированным при применении мутации образцом.

3.3. Алгоритмы для задачи о многоруких бандитах

Для измерения качества алгоритма как правило применяется метрика, отражающая общее количество недополученного выигрыша за T шагов, формулируемая следующим образом:

$$R_T = T\mu^* - \sum_{t=1}^{T} \mu_j(t), \text{ где } \mu^* = \max_{i=1,\dots,K} \mu_i$$

то есть μ^* это математическое ожидание наилучшего распределения.

Часто применяемыми алгоритмами для задачи о многоруких бандитах являются ϵ-жадный (epsilon-greedy) алгоритм и алгоритм Softmax. Помимо них есть и теоретически более эффективные, например, UCB (Upper Confidence Bound), но как показывают экспериментальные данные, простые эвристики показывают себя на уровне, а иногда и лучше. Две распространённые эвристики будут рассмотренны далее.

Epsilon-greedy является простой эвристикой, осуществляющей выбор между случайным исследованием и использованием наилучшего на данный момент варианта. Алгоритм выбора действий при использовании epsilon-

greedy алгоритма можно описать следующим образом:

$$\text{действие на шаге } T+1 = \begin{cases} \arg\max\limits_i \hat{\mu}_i(T) \text{ с вероятностью } 1-\epsilon \\ \text{случайное с вероятностью } \epsilon, \end{cases}$$

где $\hat{\mu}_i(T)$ - оценки средних значений, накопленные за предыдущие T шагов. Таким образом, данный алгоритм либо выполняет случайное исследование, либо выбирает тот из вариантов, который выглядит наиболее эффективным с точки зрения оценки среднего выигрыша.

Другой алгоритм, Softmax, в некотором роде обобщает epsilon-greedy. Для выбора варианта используется распределение Больцмана, формулируемое следующим образом:

$$p_i(T+1) = \frac{e^{\hat{\mu}_i(T)/\tau}}{\sum_{j=1}^{K} e^{\hat{\mu}_j(T)/\tau}},$$

где $p_i(T+1)$ – вероятность выбора на $T+1$ шаге i-го варианта, $\hat{\mu}_i(T)$ – оценки средних значений за предыдущие T шагов, а τ – температурный коэффициент. За счёт τ мы можем регулировать случайность алгоритма выбора. При стремлении τ к нулю алгоритм ведёт себя как простой жадный алгоритм, то есть всегда выбирается наиболее предпочтительный вариант, а при стремлении τ к бесконечности влияние средних значений снижается и выбор становится более случайным.

В данной работе выбор сделан в пользу softmax алгоритма и пользователю предоставлена возможность конфигурирования параметра τ, поскольку он относительно прост для понимания и его настройка при необходимости не должна вызвать у пользователя серьёзных затруднений, но в то же время он показывает себя довольно хорошо на практике, и кроме того решает основную проблему в виде исключения неэффективных стратегий мутации.

4. Описание разработанной системы

В данной секции будет рассмотрена общая архитектура фаззера, описан процесс конфигурации и использования фаззера конечным пользователем.

4.1. Общая архитектура

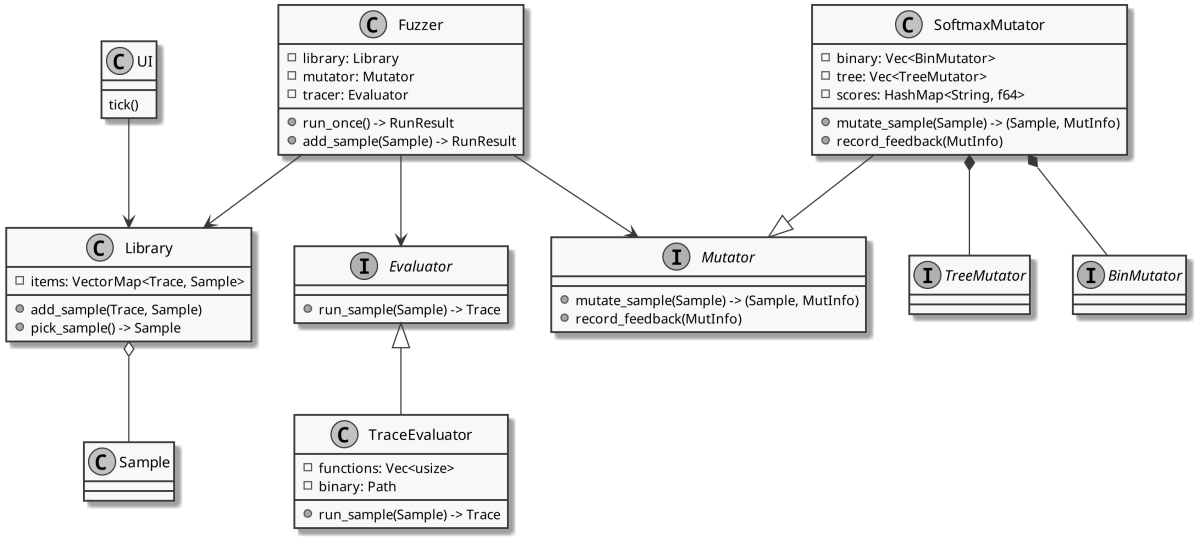

Рисунок 5 – Диаграмма классов системы

Разработанную систему можно условно поделить на компоненты, которые также приведены на рисунке 5 выше:

– библиотека образцов, ответственная за хранение состояния фаззера и выбор очередного кандидата на мутацию на основании хранимой информации о покрытии, как было описано выше;

– подсистема, ответственная за генерацию данных, выполняющая мутацию и кроссинговер для выбранного ранее образца на базе имеющейся библиотеки. Данный компонент постепенно изменяет параметры распределения вероятностей выбора из имеющихся мутаторов в соответствии с полученными данными о покрытии;

– подсистема, осуществляющая запуск исполняемого файла. При запуске через файл или стандартный ввод (в зависимости от конфигурации) передаётся очередной образец, выполняется трассировка исполняемого файла при помощи ptrace, как было описано ранее, и в результате мы получаем данные о покрытии (Trace);

– подсистема, ответственная за вывод информации на экран. Пользователю предоставляется информация о том, как протекает процесс тестирования для информирования и возможной диагностики неполадок.

В качестве языка реализации был использован язык системного программирования Rust [8], поскольку он предоставляет доступ ко всем необходимым системным вызовам и интерфейсам операционной системы и в то же время обладает субъективно высоким уровнем удобства.

4.2. Конфигурирование

Для конфигурирования фаззера используется формат Toml. В едином файле fuzz.toml пользователь указывает исполняемый файл, который хочет тестировать, описывает, что будет подаваться на вход, а также при необходимости указывает дополнительные данные, например параметры для алгоритма выбора методов мутации или параметры отдельных мутаторов. Простейшая конфигурация выглядит следующим образом:

```
[binary]
# путь до исполняемого файла
path = ”./exif”
# указание, как данные будут подаваться на вход
# (stdin по-умолчанию)
pass_style = ”file”

[input]
# папка с образцами
seeds = ”./examples”
```

Система допускает задание входа либо в виде готовых примеров и в таком случае будет использовать исключительно бинарные мутации, как это делают классические фаззеры, либо в виде грамматики, и в таком случае пользователю дополнительно нужно будет описать вход в отдельном файле в виде контекстно-свободной грамматики. Грамматика задаётся набором альтернативных продукций вида `N -> P_1|...|P_N`, где `N` это нетерминал, а `P_i` – один или более элемент из следующего набора:

– другой нетерминал, записывающийся по привычному для языков программированиям правилам: первый символ является буквой или

символом подчёркивания, а последующие буквами, цифрами или символами подчёркивания (например, `root` или `Number`);

– фиксированная строка, ограниченная двойными кавычками (например, `"SELECT"`). Данный элемент оказывается полезным для описания грамматик, порождающих текстовый ввод;

– строка, задаваемая регулярным выражением. Для задания регулярного выражения обычную строку следует заключить в re(...), например `re("[a-z][a-z]+")`. Стоит отметить, что функционал регулярных выражений можно эмулировать и с помощью продукций, данный функционал введён для повышения удобства;

– фиксированная последовательность байт, задаваемая в шестнадцатиричном виде с префиксом 0x, например `0x89504e47`. Данный тип токенов может быть полезен при описании бинарных форматов файлов, например для задания магических чисел – приведённый ранее пример является magic для файлов формата png;

– произвольный набор байт, задаваемый литералом bytes. Позволяет указать либо сегмент фиксированной длины (например `bytes(4)`) либо сегмент с длиной в некотором диапазоне (`bytes(5 8)`);

– специальный токен Nothing, позволяющий указывать пустые продукции (это полезно для записи опциональных секций через правило вида `Marker -> 0x9F | Nothing`).

Пример грамматики, составленной по данным правилам:

```
root -> "SELECT " select_target " FROM " name;
select_target -> "*" | args;
args -> name | args ", " name;
name -> re("[a-z][a-z0-9]*");
```

ЗАКЛЮЧЕНИЕ

Текст нашего умного заключения будет написан вот тут.

СПИСОК ЛИТЕРАТУРЫ

1) Incident report on memory leak caused by Cloudflare parser bug. — URL: https://blog.cloudflare.com/incident-report-on-memory-leak-caused-by-cloudflare-parser-bug/ (дата обр. 27.02.2023).

2) Cadar C., Dunbar D., Engler D. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs // Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation. — San Diego, California : USENIX Association, 2008. — C. 209—224. — (OSDI'08).

3) IFuzzer: An Evolutionary Interpreter Fuzzer Using Genetic Programming / S. Veggalam [и др.] // European Symposium on Research in Computer Security. — 2016.

4) American fuzzy lop (2.52b). — URL: https://lcamtuf.coredump.cx/afl/ (дата обр. 27.02.2023).

5) Nagy S., Hicks M. Full-Speed Fuzzing: Reducing Fuzzing Overhead through Coverage-Guided Tracing // 2019 IEEE Symposium on Security and Privacy (SP). — 2019. — C. 787—802.

6) ptrace(2) - Linux manual page. — URL: https://man7.org/linux/man-pages/man2/ptrace.2.html (дата обр. 27.02.2023).

7) Kuleshov V., Precup D. Algorithms for multi-armed bandit problems // ArXiv. — 2014. — T. abs/1402.6028.

8) Rust Programming Language. — URL: https://www.rust-lang.org/ (дата обр. 27.02.2023).