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 –

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.

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.

1.2.1. 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:

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

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 $N \to E_1 \mid ... \mid E_n$, 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.

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 следующего дерева (Рисунок 1) терминалами будут ”7”, ”+”, ”2”, ”*”, ”15”, которые свернутся в строку ”7+2*15”.

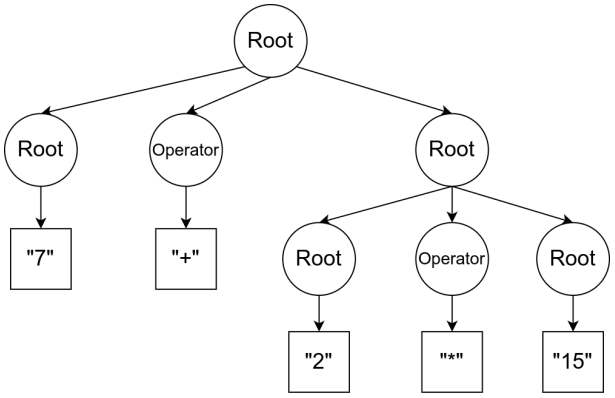

Рисунок 1 – Дерево, построенное по заданной грамматике

При этом возможно сочетание с классическими мутационными алгоритмами за счёт внедрения в результирующее синтаксическое дерево блоков, содержащих двоичные данные.

1.2.2. Генетические алгоритмы

Примером подобного подхода является iFuzzer, предназначенный специально для фаззинга интерпретаторов языков программирования [2]. 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, заключающийся 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 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 [3]. Данная инструмент предоставляет большой набор подходов, позволяющих сделать фаззинг быстрее и эффективнее:

– afl-gcc – специальный компилятор, предназначенный для генерации исполняемых файлов с дополнительной инструментацией, используемой фаззером. Помимо прочего, afl-gcc может производить дополнительное мероприятия по ”укреплению” (hardening) исполняемых файлов, что

позволяет более эффективно обнаруживать ошибки в работе с памятью;

– afl-trim

2.2. 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. В отличие от предыдущего подхода, в данном случае возможна работа с уже готовым исполняемым файлом, мы можем 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 [4]. 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.

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

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

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