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 $x = 1$.

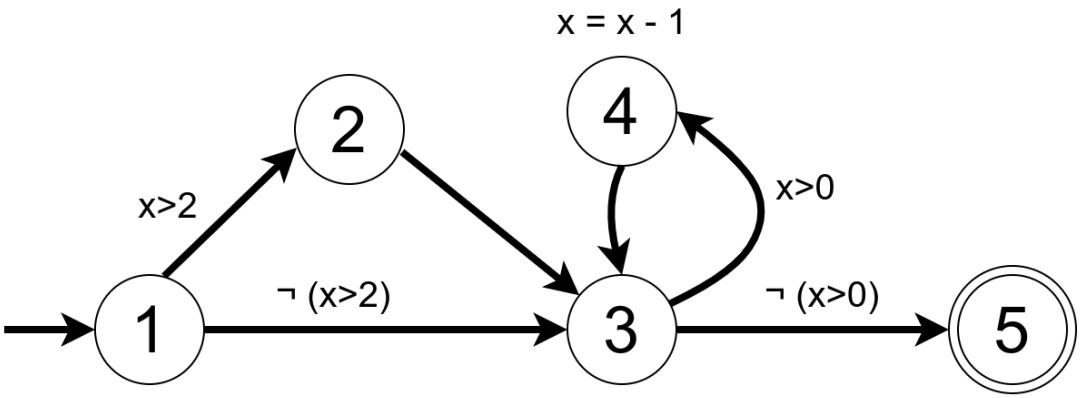

Рисунок 1 – Граф, соответствующий программе

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.

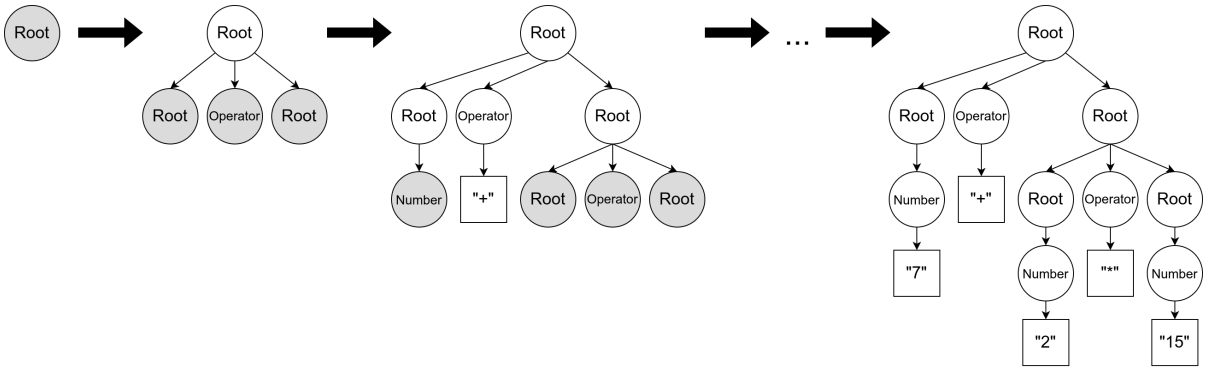

Рисунок 2 – Генерация дерева случайным применением продукций

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”.

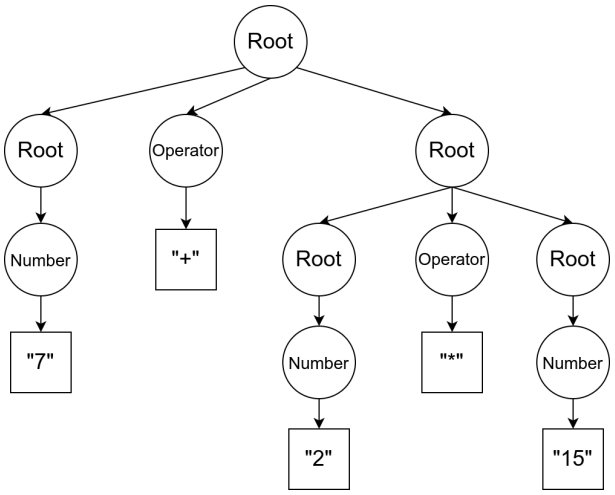

Рисунок 3 – Дерево, построенное по заданной грамматике

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 внедрения в результирующее синтаксическое дерево блоков, содержащих двоичные данные. 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 образца сохраним помимо дерева набор байтовых модификаций, представляющих собой структуру, включающую позицию применения той или иной мутации, её тип, а также дополнительные данные, если требуется. Например, если мы в результирующем образце данных хотим заменить некоторый байт на $\texttt{0xFF}_{16} = 255_{10}$, сохраним для этого изменения структуру вида $\langle k,\ Replace,\ 255\rangle$, где k - позиция байта, который мы хотим заменить. Для формирования результата мы сначала свернём синтаксическое дерево как обычно, а затем применим все байтовые изменения. Также при таком подходе становится возможным сохранение этих изменений при копировании участков из одного дерева в другое – достаточно при копировании вычислить, какому диапазону индексов в свёрнутом векторе соответствует поддерево, и скопировать вместе с ним и все байтовые мутации, приходящиеся на этот диапазон.

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 – 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. 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. 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.

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. — С. 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. — С. 787—802.