Санкт-Петербургский государственный университет

Математическое обеспечение и администрирование информационных систем

Морозко Иван Дмитриевич

Фаззинг высокопроизводительного сетевого стека OpenOnload

Отчёт по учебной практике

Производственное задание

Научный руководитель:
ассистент кафедры ИАС Смирнов К. К.

Консультант:
Генеральный директор ООО “ОКТЕТ Лабз” Ушаков К. С.

Санкт-Петербург
2022

Оглавление

Введение

Предприятия и отрасли все больше и больше полагаются на приложения, работающие в режиме реального времени, такие как голосовые вызовы, видео-встречи, торговля акциями и т. д. Подобные приложения для эффективной работы должны обеспечивать низкую задержку в передаче данных. Одним из механизмов, используемым в данной области, является обход ядра (kernel bypass) [30]. Идея состоит в том, чтобы при обработке определённых данных обойти процессы, проходящие в ядре и использовать чувствительное к задержкам оборудование непосредственно из пользовательского процесса.

В частности, для уменьшения времени обработки сетевых пакетов операционной системой, существуют программные решения, называемые сетевыми стеками пользовательского пространства [27]. Примерами могут служить такие технологии как Netmap [40], DPDK [13] и OpenOnload [8]. Разумеется, ошибки в таком программном обеспечении недопустимы. Так, например, аварийно завершившееся приложение для торговли акциями из-за ошибки в сетевом стеке может принести значительные убытки.

Данная учебная практика выполняется на базе компании ОКТЕТ Лабз[1], которая занимается разработкой программного обеспечения для валидации программно-аппаратных систем. В рамках сотрудничества с компанией Xilinx[2], ОКТЕТ Лабз участвует в разработке и тестировании высокопроизводительного сетевого стека OpenOnload. В процессе работы уже было создано большое количество функциональных тестов. Тем не менее имеет смысл привлечь другие способы поиска ошибок.

Как показывают исследования [11, 50], разработчики зачастую тратят больше времени на тестирование и верификацию программ, чем на их непосредственную реализацию. Поэтому для повышения качества тестирования продукта в условиях наличия большого количества те-

[1] http://oktetlabs.ru

[2] https://www.xilinx.com

стов ценятся методы, не требующие больших затрат на реализацию. Одним из таких методов является фаззинг — процесс многократного выполнения программы со специально сгенерированными входными данными [31].

1 Постановка задачи

Целью данной учебной практики является внедрение дополнительной технологии тестирования — фаззинга, для проверки надежности обработки пакетов сетевым стеком OpenOnload. Для её достижения были поставлены следующие задачи:

Осенний семестр

1. Оценить различные опции и методы, использующиеся в фаззерах.
2. Выбрать набор опций и методов фаззера, подходящих для тестирования OpenOnload.
3. Оценить существующие фаззеры и выбрать подходящий.

Весенний семестр

1. Провести фаззинг OpenOnload, используя выбранное решение.

2 Обзор

В данной главе приведен краткий обзор механизма работы OpenOnload и обоснование выбора технологий, используемых для достижения итоговой цели — фаззинга сетевого стека.

2.1 Принцип работы OpenOnload

OpenOnload предоставляет разделяемую библиотеку уровня пользователя — `libonload.so` и набор модулей ядра. Данная библиотека реализует сетевой стек, практически не отличающийся от реализации Linux [28]. Запустив целевое приложение с переменной окружения `LD_PRELOAD`[3], `libonload.so` будет перехватывать системные вызовы Socket API (`socket(), send(), epoll_wait()...`) [42] и другие системные вызовы, такие как `fork(), exec()` и `close()`. Таким образом, для запуска OpenOnload нет необходимости в изменении приложения.

До недавнего времени единственным ограничением была необходимость использования сетевых карт, использующих аппаратный интерфейс `ef_vi` [52], поддерживаемый сетевыми адаптерами Xilinx. Однако на текущий момент ведётся разработка поддержки `AF_XDP`[4], позволяющая ускорить работу приложений на любых сетевых картах/операционных системах, поддерживающих соответствующую технологию [8].

2.2 Плюсы фаззинга

- Удобство

  В условиях наличия большого числа существующих тестов, для нахождения неочевидных путей выполнения `PUT`[5] автоматическая генерация входных данных в процессе фаззинга удобнее, чем стандартное написание тестов.

[3] Список дополнительных, определённых пользователем разделяемых библиотек, загружаемых раньше всех остальных. Используется для для выборочного переопределения функций из других разделяемых библиотек [53].

[4] Тип сокета, использующий технологию eXpress Data Path (XDP) для ускорения обработки пакетов [60].

[5] Program Under Test.

- Скорость

  В процессе фаззинга приложению передаётся последовательность входных данных, которая может покрывать какой-то граничный случай. Затем, исходя их результатов тестирования, последовательность искажается, с целью покрыть новый блок кода. Результаты таких работ как [5, 47] показывают, что относительно стандартного написания тестов фаззинг намного эффективнее и быстрее проверяет все граничные случаи, в которых может возникнуть ошибка[6]. Более подробная информация об искажении входных данных и понятии покрытия предоставлена в параграфе 2.3.

- Актуальность

  В последние годы фаззинг стал популярен из-за увеличения вычислительной мощности и создания новых алгоритмов, что привело к созданию фаззеров, обнаруживших множество критических ошибок и уязвимостей в производственном программном обеспечении. Однако не все приложения одинаково легко поддаются фаззингу. В частности, сетевые приложения. Передача данных по сети значительно медленнее, чем чтение из файла, и, вместо обработки одного начального множества данных, часто требуется набор пакетов, соответствующий взаимодействию между клиентом и сервером (и наоборот), с отслеживанием состояния `PUT`. Поэтому в последние годы фаззинг сетевых приложений является одной из актуальных тем[7]

2.3 Классификация фаззеров

Существует множество работ, посвященных исследованиям и классификации различных методов фаззинга. В рамках обзора существующих методов используются результаты работ [10, 15, 16, 18, 49].

Фаззеры могут быть классифицированы по различным параметрам,

[6] К примеру, деление на ноль или системный вызов с нестандартными данными.

[7] В течение 2019–2022 годов было опубликовано 17 работ, посвященных фаззингу сетевых приложений [57].

таким как механизм генерации входных данных, механизм изменения входных данных, основываясь на результатах предыдущей итерации теста, стратегия исследования PUT и т. д. В данном разделе описывается основная классификация фаззеров.

2.3.1 Классификация по степени сбора информации

При классификации фаззеров по степени их зависимости от исходного кода PUT, степени анализа PUT и количества собранной информации во время выполнения программы их можно разделить на три вида: blackbox, whitebox и greybox. Примером информации, получаемой фаззерами, может служить покрытие, загруженность процессора, количество использованной оперативной памяти и т. д.

- Blackbox

  Blackbox фаззеры [22–24] выполняют тестирование, не используя никакую информацию о PUT, кроме информации о вводе и выводе. При таком методе тестирования фаззер постепенно изменяет начальные данные, пытаясь обнаружить ошибку в программе, ничего не зная о происходящих проверках внутри программы. Единственная информация, которой фаззер обладает для изменения данных — информация о выходных данных.

  Эффективность blackbox фаззинга всецело зависит от степени точности описания начального набора входных данных, принимаемых PUT, используемого алгоритма изменения данных и сложности самой программы.

- Whitebox

  Противоположностью blackbox фаззеров являются whitebox фаззеры [9, 20, 46]. В таком методе фаззер собирает информацию о внутреннем устройстве PUT и получает дополнительную информацию во время самого тестирования. Используя методы, такие как символьное выполнение или taint analysis, whitebox фаззеры могут эффективно достигать определенных участков

программы для их последующего тестирования и таким образом находить ошибки в глубокой логике программы.

Платой за такую эффективность являются большие затраты времени на анализ кода, сбор информации во время тестирования и применение этой информации во время тестирования для изменения входных данных [7].

- `Greybox`

  `Greybox` фаззеры [1, 35] частично похожи на `whitebox`. Они тоже получают информацию о программе и используют её для изменения входных данных. Однако между ними есть существенное различие — `greybox` фаззеры используют только часть информации, а именно информацию времени выполнения. Обычно в современных фаззерах такой информацией выступает покрытие [6], получаемое с помощью инструментирования. Хотя информация о покрытии позволяет сильно увеличить шансы того, что следующая тестовая итерация с изменёнными данными покроет новый блок кода, всё же это не гарантируется, в отличии от `whitebox` фаззера.

  `Greybox` фаззеры сочетают в себе достаточную осведомлённость о внутреннем устройстве `PUT` для достижения высокой эффективности в поиске ошибок, при этом сохраняя быстродействие [20].

2.3.2 Классификация по методу генерации данных

Последовательность данных, отправляемая фаззером `PUT`, должна быть достаточно правильной, то есть подходить под некие базовые критерии программы[8], но в то же время быть достаточно искажённой чтобы спровоцировать ошибку в глубокой логике `PUT`. После создания последовательности данных фаззер применяет к ним небольшие изменения.

Также фаззеры изменяют входные данные, основываясь на резуль-

[8] К примеру, сетевой пакет должен удовлетворять модели OSI.

татах предыдущих тестовых итераций. Существует множество методик такого изменения: от случайных инвертирований битов [25] до генетических алгоритмов [26].

Двумя основными методами генерации данных для фаззинга являются генерация, основанная на модели и генерация, основанная на мутации.

- Генерация, основанная на модели

  В данном методе генерация входных данных происходит по определённым строгим правилам. Такие правила представляют собой описание входных данных, которые ожидает `PUT`. К примеру, SNOOZE [41] использует предоставленный пользователем `xml` файл с описанием протокола общения пользователя с программой.

- Мутация заранее определённого набора

  Для данного метода генерации необходим набор начальных, заранее определённых входных данных. Такие данные обычно представляют собой файлы поддерживаемого `PUT` типа. К примеру, `mp3` файл, `pdf` документ или сетевой пакет. Для генерации новых тестовых данных начальные данные разбиваются на небольшие блоки, к которым применяются различные мутации, такие как: инвертирование битов, удаление случайного блока, случайная перестановка блоков, замена случайного блока на случайное значение и подобные [15].

2.3.3 Классификация по стратегии исследования программы

Фаззинг может ставить перед собой различные цели. Несомненно, общей целью любого фаззинга является поиск ошибок, однако ошибки можно искать в различных местах. Поэтому существует разделение по стратегии исследования программы: фаззинг на основе покрытия и направленный фаззинг.

- Фаззинг на основе на покрытия

Данный тип фаззинга нацелен на создание множества наборов входных данных для тестов, покрывающих как можно больше исходного кода.

- Направленный фаззинг

  Напротив, направленный фаззинг нацелен на создание множества набора входных данных, покрывающих конкретную часть программы.

2.4 Выбор подходящего метода

В данном разделе приведены причины, по которым был сделан тот или иной выбор в классификации фаззеров, используемых для фаззера в данной учебной практике.

Выбранные параметры фаззера

- `Greybox`

  `Blackbox` фаззинг используется в случае, когда требуется быстро проверить программу на наличие поверхностных ошибок. Такой подход часто используется для систем, для которых не было ещё проведено тестирование [16]. В случае тестирования OpenOnload, значительная доля тестирования уже была проведена.

  `Whitebox` фаззинг может достичь большого покрытия кода и нахождения не очевидных ошибок, однако на практике такой подход редко используется ввиду больших затрат времени и ресурсов [19].

  `Greybox` фаззинг является хорошей альтернативой вышеперечисленным подходам.

- Мутация заранее определенного набора

  У каждого из способов генерации есть свои преимущества и недостатки. К преимуществам фаззинга с использованием мутаций можно отнести относительно небольшие затраты, необходимо

только получить набор подходящих входных данных. Недостатками такого подхода является неэффективное прохождение различных валидаций при начальной обработке ввода программой, и, как следствие, фаззинг, основанный на мутации, достигает меньшего покрытия чем генерация данных по модели [29].

Однако новые работы в области сетевого фаззинга [33,37] показывают, что для тестирования сетевых приложений мутация гораздо эффективнее, чем генерация. Начальными данными в таком случае выступает реальный трафик передающийся целевому приложению. Таким образом, становится необязательным формально описывать каждый интересующий протокол, достаточно получить соответствующий `pcap`[9] файл.

- Фаззинг на основе покрытия

  Поскольку целью данной учебной практики не являлся поиск ошибок в конкретной части проекта, была выбрана стратегия фаззинга, основывающаяся на покрытии.

  После нахождения подходящего фаззера, было решено не менять использующееся в нём определение покрытия, поскольку исследование [3] показывает, что не существует большой разницы между различными типами покрытия.

2.5 Выбор подходящего фаззера

Как показывают перечисленные ранее обзорные работы (раздел 2.3), в основном сетевые фаззеры представляют собой `blackbox` фаззеры, использующие метод генерации данных, основанный на модели. В данном разделе представлен список известных сетевых фаззеров и основных методов, использующихся в них.

[9] API для перехвата сетевого трафика и одноимённое расширение для файлов, содержащих информацию о пойманном трафике [59].

Сетевые фаззеры

- Sulley [34], Boofuzz [36], Fuzzowski [39], SNOOZE [41]

  `Blackbox` фаззеры, использующие метод генерации данных по модели, предоставленной пользователем.

- Secfuzz [48]

  `Blackbox` фаззер, использующий метод генерации данных по модели. Модель в данном случае строится фаззером автоматически в процессе анализа общения целевого сервера с настоящим клиентом.

- AFLsmart [43], NAUTILUS [44]

  `Greybox` фаззеры, использующие метод генерации данных по модели. Модель в данном случае строится фаззером автоматически в процессе анализа общения целевого сервера с настоящим клиентом.

- MoWF [38], GRT [17]

  `Whitebox` фаззеры, использующие метод генерации данных по модели, предоставленной пользователем.

- AFL [51], Libfuzzer [58]

  `Greybox` фаззеры на основе покрытия, использующие метод мутации заранее определённого набора. Было решено не использовать данные фаззеры ввиду их универсальности, ведь существуют фаззеры, предназначенные конкретно для тестирования сетевых приложений.

- SAVHF [14]

  `Greybox` фаззер на основе покрытия, использующий метод мутации заранее определённого набора. Не подходит, поскольку данный проект — проприетарный.

Фаззер	Открытый код	Для тестирования сетей	Box	Мутация данных	Учёт покрытия
Sulley, Boofuzz, Snooze	+	+	●	-	-
MoWF, GRT	-	-	○	-	+
AFLsmart, NAUTILIUS	+	-	◐	-	+
AFL, Libfuzzer	+	-	◐	+	+
SAVHF	-	-	◐	+	+
AFLnet	+	+	◐	+	+
Nyx-Net	+	+	◐	+	+

Таблица 1: Оценка рассмотренных фаззеров.

- AFLnet [37]

  `Greybox` фаззер на основе покрытия, использующий метод мутации заранее определённого набора и разработанный специально для тестирования сетевых приложений. Подходящий кандидат.

- Nyx-Net [33]

  `Greybox` фаззер на основе покрытия, использующий метод мутации заранее определённого набора и разработанный специально для тестирования сетевых приложений. Подходящий кандидат.

Оценка рассмотренных фаззеров представлена на таблице 1. Среди двух подходящих фаззеров был сделан выбор в сторону AFLNet. Несмотря на то что Nyx-Net является более новым решением, авторы которого развили предлагаемые AFLnet идеи и предоставили способ фаззинга который, дословно, «Сравнивая по бенчмарку, предоставленным AFLnet — ProFuzzBench [32], Nyx-Net повышает скорость тестирования до 300 раз и увеличивает покрытие до 70%» [33], AFLNet имеет активное сообщество и большее количество документации.

3 Реализация

Этапы фаззинга любого приложения с помощью AFL-подобного фаззера выглядят следующим образом:

1. Инструментирование целевого приложения, то есть сборка проекта одним из компиляторов, предоставляемым AFL: `afl-gcc`, `afl-g++`, `afl-clang-fast`, `afl-clang-fast++` [56].

2. Запуск программы `afl-fuzz`, принимающей целевое приложение в виде бинарного файла, его аргументы, информацию о протоколе и т. д.

   В случае AFLNet, `afl-fuzz`:

   - Запускает целевой сервер в отдельном процессе посредством `fork()` и `execv()`.
   - Выполняет роль клиента, подключающегося к этому серверу.
   - Генерирует входные данные на основе собранной информации и посылает их серверу.
   - Собирает информацию о покрытии.
   - Повторяет процесс.

На рисунке 1 представлена схема взаимодействия сервера и AFLNet в процессе фаззинга.

Для фаззинга OpenOnload необходимо провести инструментирование не только целевого сервера, но и библиотеки `libonload.so`. Сервер в данном случае выполняет роль посредника между фаззером и библиотекой, вызывая различные функции из последней (которые являются простым Socket API). При этом инструментируется только часть OpenOnload, работающая в пространстве пользователя, ввиду невозможности сбора покрытия с модулей ядра.

Поскольку AFLNet запускает сервер и клиент на одном компьютере, трафик направляется через интерфейс-петлю. OpenOnload необходимо

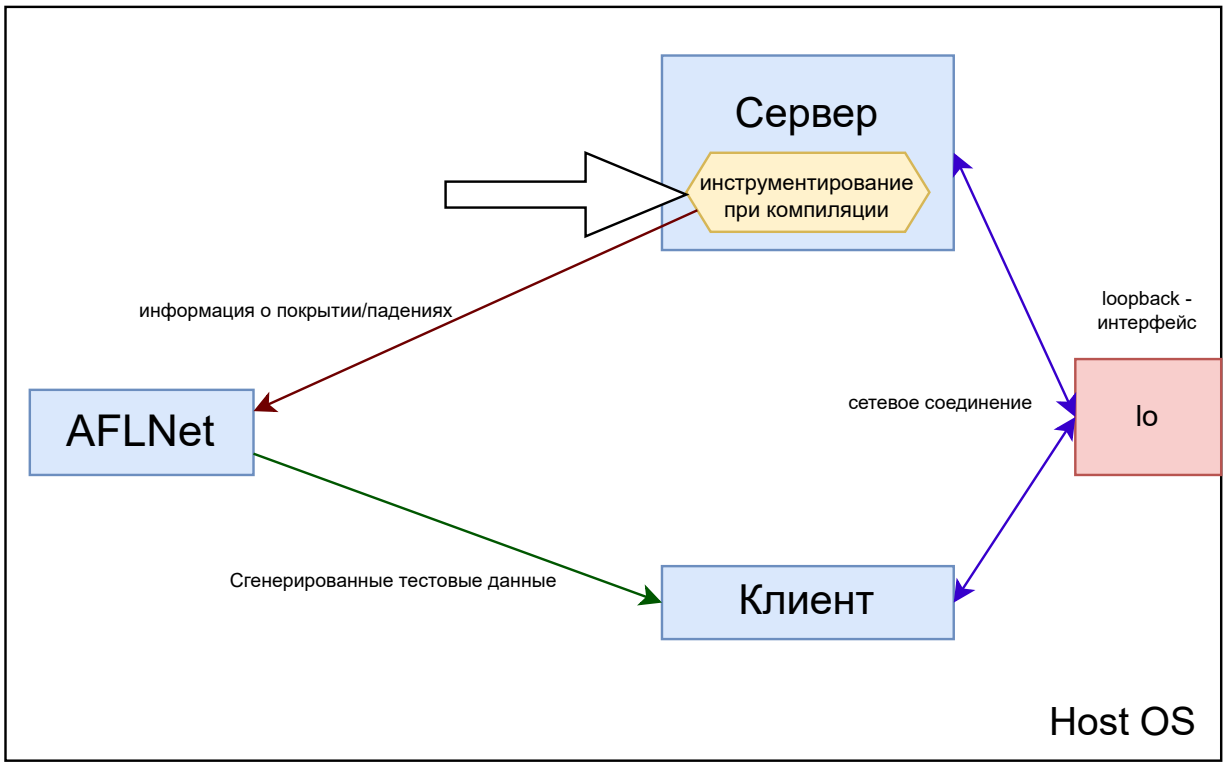

Рис. 1: Взаимодействие целевого сервера с AFLNet. Синие блоки — части одной программы `afl-fuzz`.

указать сетевой интерфейс с которым планируется работа. Стандартный `loopback`-интерфейс для этой цели не поддерживается. Для обхода данного препятствия были использованы виртуальные Ethernet-устройства — `veth` [61]. Поместив пару `veth`-устройств в различные сетевые пространства имён[10], можно избежать участия `loopback` в передаче пакетов.

В AFLNet нет возможности запускать сервер в отдельном сетевом пространстве имён, поэтому в рамках учебной практики был создан `pull request`, добавляющий такую возможность[11].

Итоговая схема взаимодействия AFLNet с целевым сервером в процессе фаззинга OpenOnload предоставлена на рисунке 2.

[10]Сетевое пространство имен в ОС Linux представляет собой копию сетевого стека со своими собственными таблицами маршрутизации, правилами брандмауэра, сетевыми устройствами и т. д. [4]

[11]https://github.com/aflnet/aflnet/pull/70

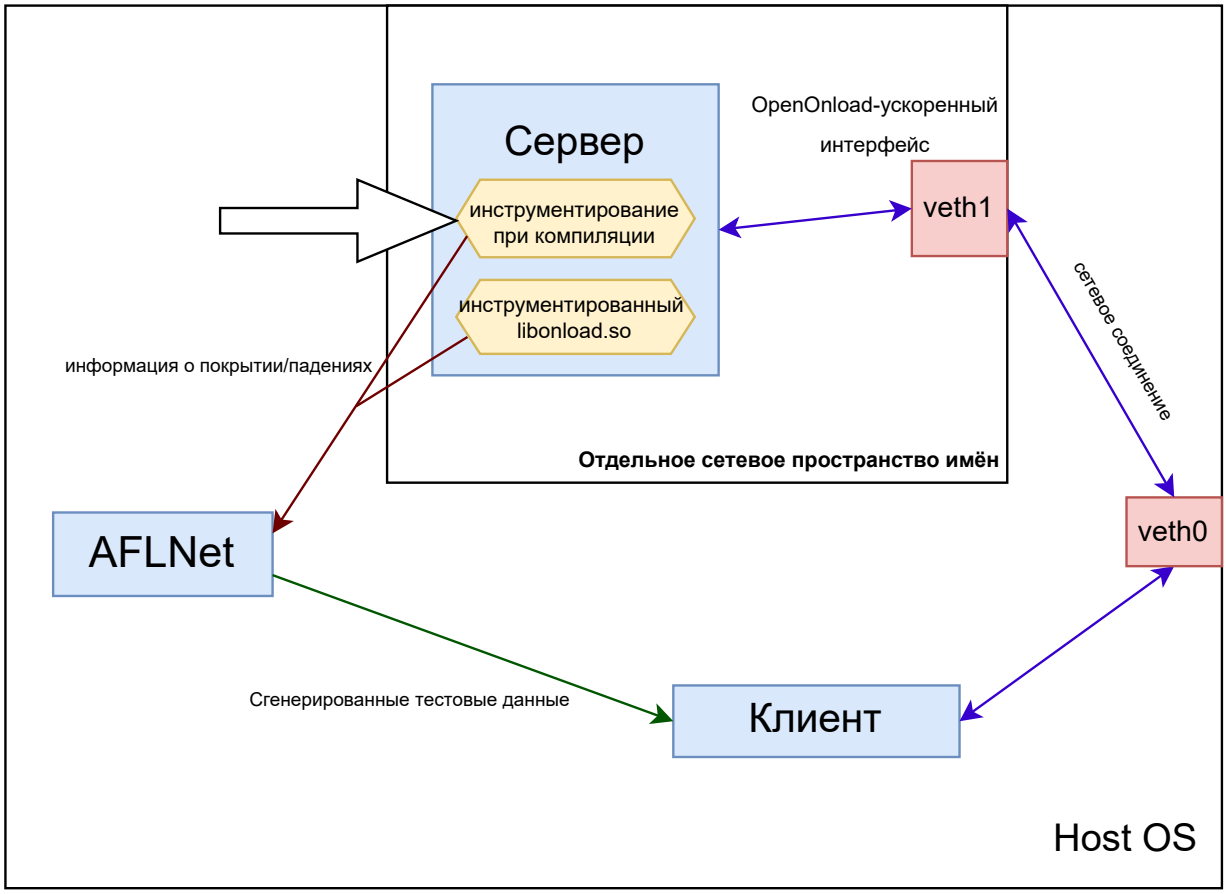

Рис. 2: Архитектура фаззинга OpenOnload через AFLNet. Целевой сервер вызывает функции из `libonload.so`, покрытие собирается и с библиотеки, и с самого сервера.

3.1 Оптимизации

Время на инициализацию OpenOnload при запуске нового процесса довольно велико. Инициализация простого TCP-сервера с OpenOnload, использовавшегося в тестировании (раздел 4.1), занимала порядка 0.10–0.17 секунд. Поскольку AFLNet подразумевает перезапуск сервера на каждой новой тестовой итерации, данное ограничение сильно сказывается на скорости и эффективности фаззинга.

С целью уменьшить время запуска были использованы следующие оптимизации:

- AFL Persistent Mode [21]

  Механизм, позволяющий AFL проводить тестовые итерации несколько раз в одном процессе, вместо вызова `fork()` для каждой новой итерации, что значительно ускоряет процесс фаззинга. Для использования данного метода, код целевой программы должен иметь вид как на листинге 1.

```
while (__AFL_LOOP(1000)) {

  /* Чтение входных данных */
  /* Вызов функции из целевой библиотеки */
  /* Сброс состояния */

}

/* Выход из программы */
```

Листинг 1: Структура целевой программы, использующей `persistent mode`.

  При использовании `persistent mode` желательно чтобы состояние целевого приложения можно было полностью сбросить из кода внутри цикла `__AFL_LOOP`. Иначе предыдущие запуски будут влиять на будущие, и стабильность[12] фаззинга уменьшиться. Но, поскольку данный подход позволяет не инициализировать OpenOnload заново для каждой тестовой итерации, происходит значительное увеличение числа запусков в секунду (подробнее в

[12]Метрика AFL, оценивающая способность программы выполнять одинаковые действия на одних и тех же входных данных [54].

разделе 4.1). Таким образом, во время тестирования было принято решение проводить фаззинг не только стандартным путём, но и воспользоваться этим методом.

`Persistent mode` работает только при компиляции в так называемом режиме `LLVM` с помощью компиляторов `afl-clang-fast/afl-clang-fast++`. OpenOnload, к сожалению, не поддерживает компиляцию через `clang`.

Было найдено решение в виде компилятора `afl-gcc-fast`, предоставляемого проектом AFL++ [2] — ответвлением от AFL. Данный компилятор позволяет использовать `persistent mode`, не завися от внутренних компонентов `LLVM` [45].

- Общий стек OpenOnload

  Приложение начинает свою работу с OpenOnload тогда, когда оно создаёт сокет системным вызовом `socket()`. В этот момент OpenOnload создаёт так называемый стек — абстракцию, позволяющую сокету взаимодействовать с сетевой картой. Время жизни стека не зависит от времени жизни приложения, и разным процессам могут соответствовать разные стеки (иногда больше одного) [8].

  Поскольку создание стека — медленный процесс, было решено использовать один стек для всех запусков. Достигается это путём выставления переменной окружения `EF_NAME`, что заставляет процесс использовать стек с заданным именем. Теперь достаточно создать стек путём запуска какого-либо приложения с OpenOnload в фоновом режиме [13] и указать соответствующий `EF_NAME` для AFLNet.

[13] В данном случае использовался

```
1 EF_NAME=sample_name onload nc -l -p 6666 &
```

4 Тестирование

Тесты проводились с помощью ProFuzzBench [32] — бенчмарк для фаззинга сетевых протоколов, включающий в себя инструменты для автоматизации фаззинга сетевых приложений. Система, на которой проводился запуск фаззинга:
`Debian 11, Linux 5.16.12,`
`AMD Ryzen 7 2700 Eight-Core Processor, 16GB DDR4 RAM.`

4.1 Простой TCP сервер

С целью удостовериться что AFLNet действительно учитывает покрытие с `libonload.so`, было принято решение провести тестирование на простом TCP сервере[14]. Поскольку количество возможных путей, отслеживаемых AFLNet в случае такой программы мало[15] запустив фаззинг сервера с `libonload.so` ожидается значительное увеличение этого числа. Был произведен фаззинг TCP сервера в `persistent mode` в течение 60 минут в двух вариантах: с подгруженной библиотекой, и без неё. Результаты работы фаззинга TCP сервера с OpenOnload можно рассмотреть на рисунке 3, без OpenOnload — на рисунке 4.

Суммарную оценку найденного покрытия можно рассмотреть на рисунке 5. Для построения графиков использовались инструменты, предоставленные ProFuzzBench.

[14] https://gist.github.com/ol-imorozko/5d88d9e2bd74cbfc4abf8a66b34252d6
[15] Инструментирование через `afl-gcc-fast` даёт 18 возможных путей.

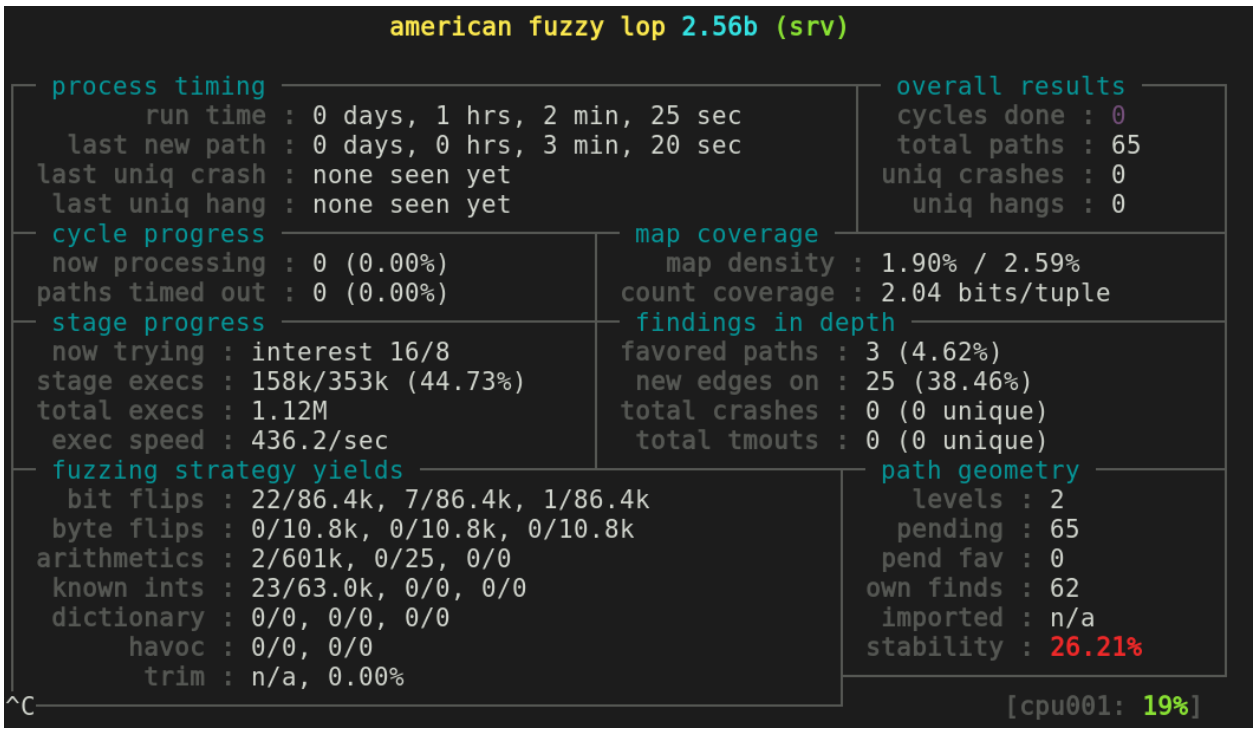

Рис. 3: Фаззинг TCP сервера с OpenOnload. Число путей — 65, стабильность — 26.21%

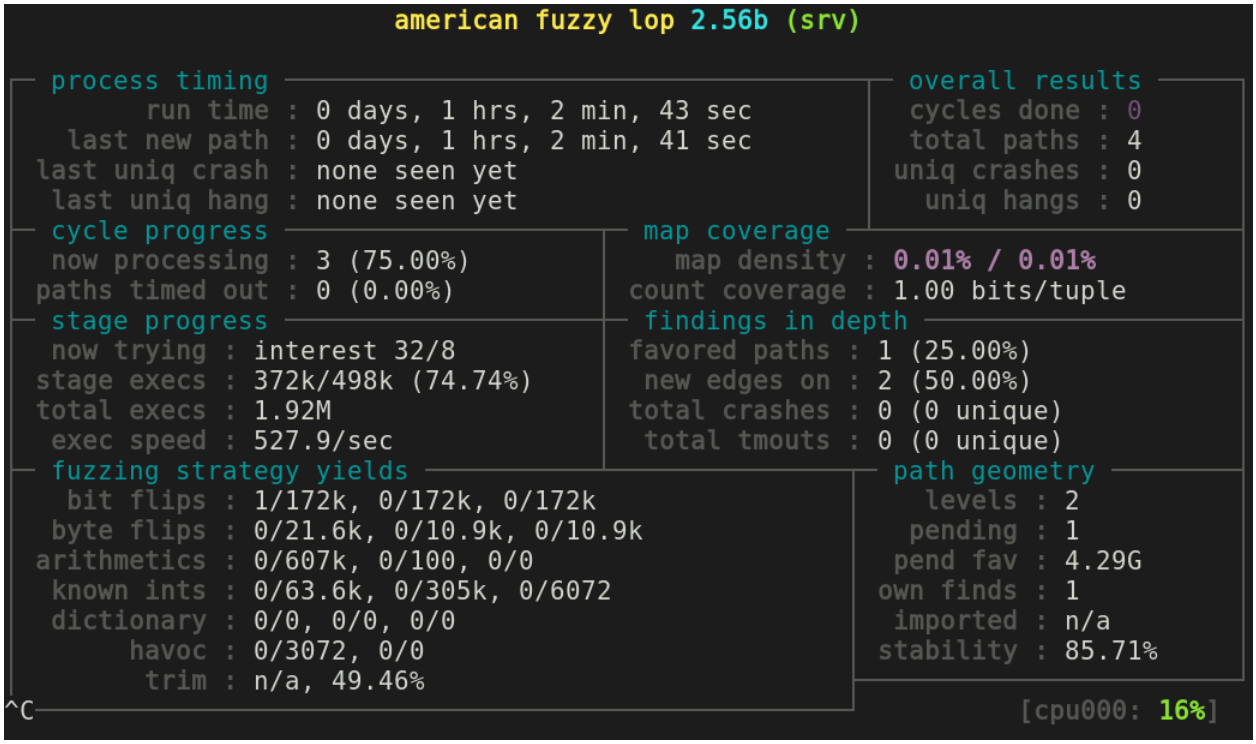

Рис. 4: Фаззинг TCP сервера без OpenOnload. Число путей — 4, стабильность — 85.71%. Число тестовых итераций в секунду больше на ≈ 90.

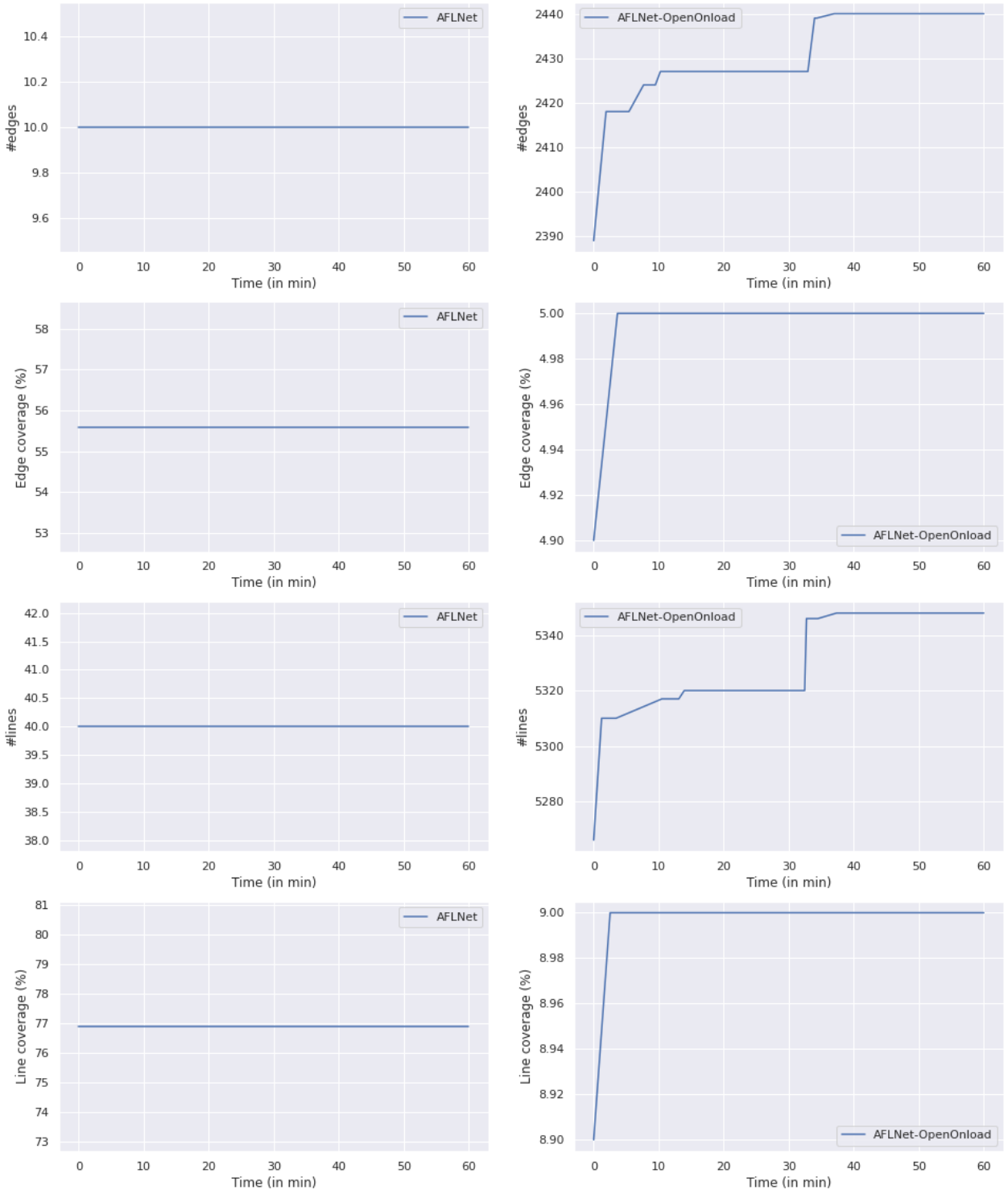

Рис. 5: Оценка найденного AFLNet покрытия в TCP сервере с/без OpenOnload. При фаззинге сервера с OpenOnload, общее число путей значительно больше, чем число путей для обычного запуска. Таким образом, можно сделать вывод что фаззинг действительно собирает покрытие с `libonload.so`.

Так же можно заметить что фаззинг сервера с OpenOnload обладает гораздо меньшей стабильностью. Две основные причины такого поведения:

- Природа сетевого соединения

  Невозможно на каждой тестовой итерации передавать данные с одинаковыми задержками между ними, и ожидать что целевой сервер будет их обрабатывать с одинаковой скоростью. В отличие от приложений, принимающих данные одним блоком, сетевым приложениям данные поступают порциями, поэтому одни и те же данные могут обрабатываться по разному из-за различных задержек.

- `Persistent mode`

  Как было сказано ранее (раздел 3.1), `persistent mode` может быть использован для ускорения фаззинга, но такой режим может приводить к снижению стабильности. В данном случае, стек OpenOnload не освобождался и создавался заново.

4.2 Live555

Следующей целью для фаззинга стал RTSP-сервер[16], основанный на Live555 [12] — наборе библиотек C++ с открытым исходным кодом, разработанных для потоковой передачи мультимедиа. В этом случае было принято решение восстанавливать состояние OpenOnload на каждой тестовой итерации с целью увеличить стабильность.

Планировалось провести фаззинг RTSP-сервера в течение двух дней. Однако, спустя 76 минут после начала фаззинга, была обнаружена ошибка «Unable to handle page fault» в одном из модулей ядра Linux, предоставляемых OpenOnload. Проводить дальнейший процесс фаззинга не предоставлялось возможным из-за ошибки в ядре операционной системы. Результаты фаззинга можно оценить на рисунке 6.

[16] https://github.com/rgaufman/live555/blob/master/testProgs/testOnDemandRTSPServer.cpp

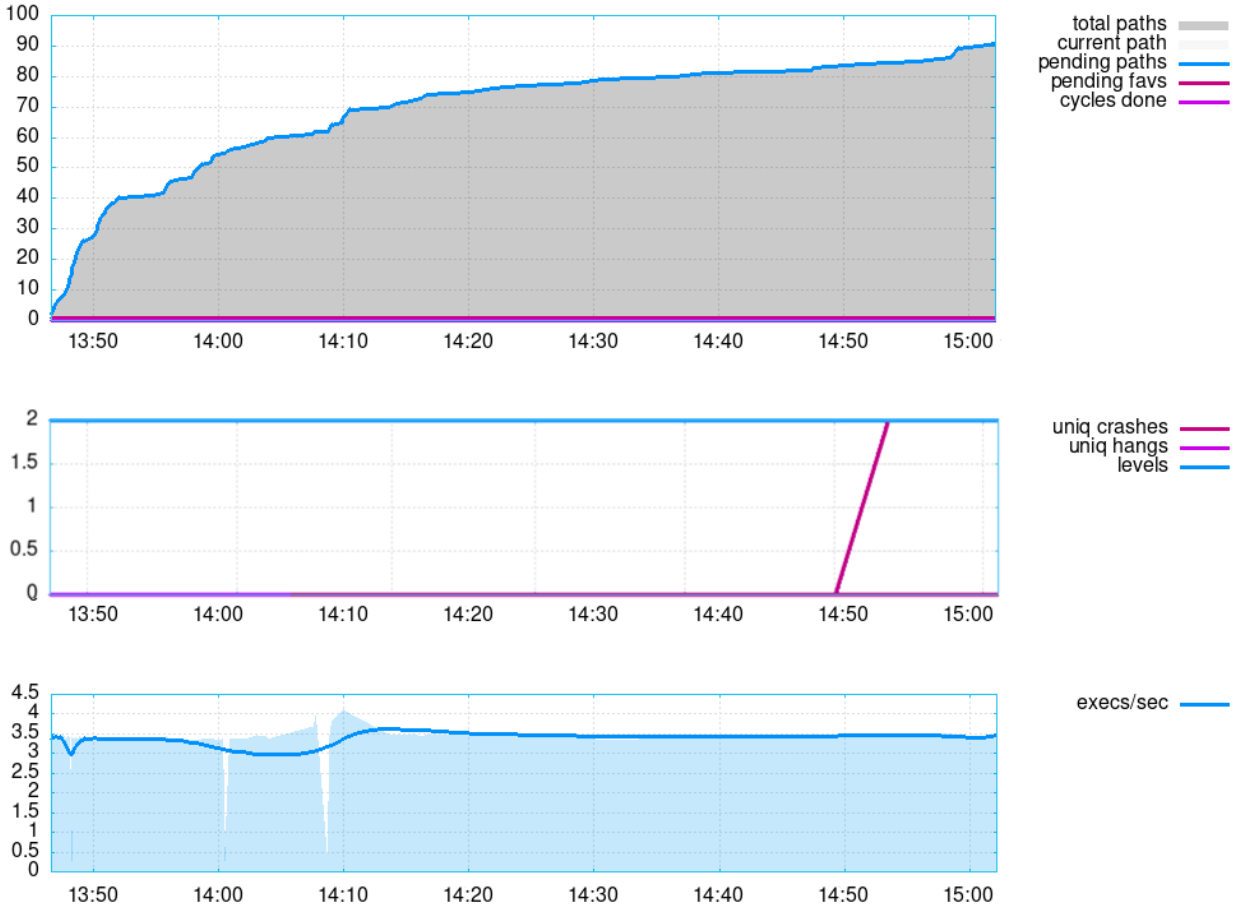

Рис. 6: Фаззинг RTSP сервера с OpenOnload. Графики построены с помощью программы `afl-plot` [55]. Продемонстрировать скринот `afl-fuzz` не предоставляется возможным ввиду ошибки в ядре во время фаззинга.

.

На данный момент ведётся работа по воспроизведению найденной в процессе фаззинга ошибки.

5 Заключение

В результате проделанной работы были выполнены следующие задачи:

- Оценены различные опции и методы, использующиеся в фаззерах.
- Выбран набор различных опции и методов для фаззера, подходящих в данной ситуации.
- Оценены существующие фаззеры.
- Выбран подходящий фаззер.
- Проведён фаззинг OpenOnload.

  Была найдена ошибка в модуле ядра OpenOnload. Так же был создан `pull request`, добавляющий в AFLNet возможность запуска сервера в отдельном сетевом пространстве имён[17].

В дальнейших планах для развития проекта:

- Воспользоваться фаззером Nyx-Net

  Благодаря запуску целевого приложения в виртуальной машине, после ошибки в ядре можно восстановить состояние всей системы, и продолжать процесс фаззинга. Так же, Nyx-Net позволяет проводить фаззинг не только пакетов, но и API приложений [33]. Таким образом, можно будет провести фаззинг `ioctl()` вызовов, посредством которых `libonload.so` взаимодействует с модулями ядра OpenOnload.

В связи с тем, что фаззинг проводился на версии OpenOnload, находящейся сейчас в разработке, код закрыт.

[17] https://github.com/aflnet/aflnet/pull/70

Список литературы

[1] Fioraldi Andrea, Maier Dominik, Eißfeldt Heiko, and Heuse Marc. AFL++ : Combining Incremental Steps of Fuzzing Research // 14th USENIX Workshop on Offensive Technologies (WOOT 20). — USENIX Association. — 2020. — Aug. — Access mode: https://www.usenix.org/conference/woot20/presentation/fioraldi (online; accessed: 2021-12-09).

[2] Fioraldi Andrea, Maier Dominik, Eißfeldt Heiko, and Heuse Marc. AFL++ : Combining Incremental Steps of Fuzzing Research // 14th USENIX Workshop on Offensive Technologies (WOOT 20). — USENIX Association. — 2020. — Aug. — Access mode: https://www.usenix.org/conference/woot20/presentation/fioraldi.

[3] Wang Jinghan, Duan Yue, Song Wei, Yin Heng, and Song Chengyu. Be Sensitive and Collaborative: Analyzing Impact of Coverage Metrics in Greybox Fuzzing // 22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019). — Chaoyang District, Beijing : USENIX Association. — 2019. — Sep. — P. 1–15. — Access mode: https://www.usenix.org/conference/raid2019/presentation/wang (online; accessed: 2021-12-09).

[4] Biederman Eric W. ip-netns(8) — Linux manual page. — Access mode: https://man7.org/linux/man-pages/man8/ip-netns.8.html (online; accessed: 2022-05-18).

[5] Böhme Marcel, Pham Van-Thuan, and Roychoudhury Abhik. Coverage-Based Greybox Fuzzing as Markov Chain // Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. — New York, NY, USA : Association for Computing Machinery. — 2016. — CCS '16. — P. 1032–1043. — Access mode: https://doi.org/10.1145/2976749.2978428 (online; accessed: 2021-12-09).

[6] Böhme Marcel, Pham Van-Thuan, and Roychoudhury Abhik.

Coverage-Based Greybox Fuzzing as Markov Chain // Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. — New York, NY, USA : Association for Computing Machinery. — 2016. — CCS '16. — P. 1032–1043. — Access mode: https://doi.org/10.1145/2976749.2978428 (online; accessed: 2021-12-09).

[7] Böhme Marcel and Paul Soumya. A Probabilistic Analysis of the Efficiency of Automated Software Testing // IEEE Transactions on Software Engineering. — 2016. — Vol. 42, no. 4. — P. 345–360.

[8] CNS Xilinx. OpenOnload. — Access mode: https://github.com/Xilinx-CNS/onload (online; accessed: 2022-05-18).

[9] Cadar Cristian, Dunbar Daniel, and Engler Dawson. 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. — USA : USENIX Association. — 2008. — OSDI'08. — P. 209–224.

[10] Fell James. A Review of Fuzzing Tools and Methods // Technical report, Technical Report. — 2017.

[11] Fine S. and Ziv A. Coverage directed test generation for functional verification using Bayesian networks // Proceedings 2003. Design Automation Conference (IEEE Cat. No.03CH37451). — 2003. — P. 286–291.

[12] Finlayson Ross. Internet Streaming Media, Wireless, and Multicast technology, services, & standards. — Access mode: http://www.live555.com/ (online; accessed: 2022-05-18).

[13] Foundation Linux. Data Plane Development Kit (DPDK). — 2015. — Access mode: http://www.dpdk.org.

[14] Xu Hang, Qin Ganyu, Zhu Junhu, Liu Zimian, and Liu Zhiqiang. Framework for State-Aware Virtual Hardware Fuzzing // Wireless

Communications and Mobile Computing. — 2021. — May. — Vol. 2021. — P. 1–14.

[15] Manès Valentin J. M., Han HyungSeok, Han Choongwoo, Cha Sang Kil, Egele Manuel, Schwartz Edward J., and Woo Maverick. Fuzzing: Art, Science, and Engineering // CoRR. — 2018. — Vol. abs/1812.00140. — arXiv : 1812.00140.

[16] Liang Hongliang, Pei Xiaoxiao, Jia Xiaodong, Shen Wuwei, and Zhang Jian. Fuzzing: State of the Art // IEEE Transactions on Reliability. — 2018. — Vol. 67, no. 3. — P. 1199–1218.

[17] Ma Lei, Artho Cyrille, Zhang Cheng, Sato Hiroyuki, Gmeiner Johannes, and Ramler Rudolf. GRT: Program-Analysis-Guided Random Testing (T) // 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE). — 2015. — P. 212–223.

[18] Godefroid Patrice. Fuzzing: Hack, Art, and Science // Commun. ACM. — 2020. — Jan. — Vol. 63, no. 2. — P. 70–76. — Access mode: https://doi.org/10.1145/3363824 (online; accessed: 2021-12-09).

[19] Godefroid Patrice, Levin Michael Y., and Molnar David. SAGE: Whitebox Fuzzing for Security Testing: SAGE Has Had a Remarkable Impact at Microsoft. // Queue. — 2012. — Jan. — Vol. 10, no. 1. — P. 20–27. — Access mode: https://doi.org/10.1145/2090147.2094081 (online; accessed: 2021-12-09).

[20] Godefroid Patrice, Levin Michael Y., and Molnar David A. Automated Whitebox Fuzz Testing // Network Distributed Security Symposium (NDSS) / Internet Society. — 2008. — Access mode: http://www.truststc.org/pubs/499.html (online; accessed: 2021-12-09).

[21] Google. Fast LLVM-based instrumentation for afl-fuzz. — Access mode: https://github.com/aflnet/aflnet/tree/master/llvm_mode (online; accessed: 2022-05-18).

[22] Helin Aki. Radamsa. — 2021. — Access mode: https://gitlab.com/akihe/radamsa (online; accessed: 2021-12-09).

[23] Hocevar Sam. zzuf. — 2021. — Access mode: https://github.com/samhocevar/zzuf (online; accessed: 2021-12-09).

[24] Holler Christian, Herzig Kim, and Zeller Andreas. Fuzzing with Code Fragments // 21st USENIX Security Symposium (USENIX Security 12). — Bellevue, WA : USENIX Association. — 2012. — Aug. — P. 445–458. — Access mode: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/holler (online; accessed: 2021-12-09).

[25] Householder Allen D. CERT BFF - Basic Fuzzing Framework. — 2020. — Access mode: https://vuls.cert.org/confluence/display/tools/CERT+BFF+-+Basic+Fuzzing+Framework (online; accessed: 2021-12-09).

[26] Jääskelä Esa. Genetic algorithm in code coverage guided fuzz testing // Dept. Comput. Sci. Eng., Univ. Oulu. — 2016.

[27] Kinsella Ray. The brief case for User-space Network Stacks $DPDKandfriends$. — FOSDEM VZW. — 2019. — https://doi.org/10.5446/44581 $Lastaccessed$: $18May2022$.

[28] Konstantin Ushakov. Сетевой стек OpenOnload. В чем и почему он обыгрывает ядро Linux : Oktet Labs. — 2016. — Access mode: https://www.youtube.com/watch?v=HkfAKFtBvG0 (online; accessed: 2022-05-18).

[29] Li Jun, Zhao Bodong, and Zhang Chao. Fuzzing: a survey // Cybersecurity. — 2018. — Dec. — Vol. 1.

[30] Majkowski Marek. Kernel Bypass. — 2015. — Access mode: https://blog.cloudflare.com/kernel-bypass/ (online; accessed: 2022-05-18).

[31] Miller Barton P., Fredriksen Louis, and So Bryan. An Empirical Study of the Reliability of UNIX Utilities // Commun. ACM. — 1990. — Dec. — Vol. 33, no. 12. — P. 32–44. — Access mode: https://doi.org/10.1145/96267.96279 (online; accessed: 2021-12-09).

[32] Natella Roberto and Pham Van-Thuan. ProFuzzBench: A Benchmark for Stateful Protocol Fuzzing // Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis. — 2021.

[33] Schumilo Sergej, Aschermann Cornelius, Jemmett Andrea, Abbasi Ali, and Holz Thorsten. Nyx-Net: Network Fuzzing with Incremental Snapshots // CoRR. — 2021. — Vol. abs/2111.03013. — arXiv : 2111.03013.

[34] OpenRCE. Sulley. — 2021. — Access mode: https://github.com/OpenRCE/sulley (online; accessed: 2021-12-09).

[35] Österlund Sebastian, Razavi Kaveh, Bos Herbert, and Giuffrida Cristiano. ParmeSan: Sanitizer-guided Greybox Fuzzing // 29th USENIX Security Symposium (USENIX Security 20). — USENIX Association. — 2020. — Aug. — P. 2289–2306. — Access mode: https://www.usenix.org/conference/usenixsecurity20/presentation/osterlund (online; accessed: 2021-12-09).

[36] Pereyda Joshua. boofuzz: Network Protocol Fuzzing for Humans. — 2021. — Access mode: https://llvm.org/docs/LibFuzzer.html (online; accessed: 2021-12-09).

[37] Pham Van-Thuan, Böhme Marcel, and Roychoudhury Abhik. AFLNet: A Greybox Fuzzer for Network Protocols // Proceedings of the 13rd IEEE International Conference on Software Testing, Verification and Validation : Testing Tools Track. — 2020.

[38] Pham Van-Thuan, Böhme Marcel, and Roychoudhury Abhik. Model-based whitebox fuzzing for program binaries // 2016

31st IEEE/ACM International Conference on Automated Software Engineering (ASE). — 2016. — P. 543–553.

[39] Plc NCC Group. Fuzzowski. — 2021. — Access mode: https://github.com/nccgroup/fuzzowski (online; accessed: 2021-12-09).

[40] Rizzo Luigi. netmap: A Novel Framework for Fast Packet I/O // 2012 USENIX Annual Technical Conference (USENIX ATC 12). — Boston, MA : USENIX Association. — 2012. — June. — P. 101–112. — Access mode: https://www.usenix.org/conference/atc12/technical-sessions/presentation/rizzo.

[41] Banks Greg, Cova Marco, Felmetsger Viktoria, Almeroth Kevin C., Kemmerer Richard A., and Vigna Giovanni. SNOOZE: Toward a Stateful NetwOrk prOtocol fuzZEr // ISC. — 2006.

[42] Sechrest Stuart. An introductory 4.4.bsd interprocess communication tutorial. — 1993.

[43] Pham Van-Thuan, Böhme Marcel, Santosa Andrew E., Căciulescu Alexandru Răzvan, and Roychoudhury Abhik. Smart Greybox Fuzzing. — 2018. — 1811.09447.

[44] Pham Van-Thuan, Böhme Marcel, Santosa Andrew E., Căciulescu Alexandru Răzvan, and Roychoudhury Abhik. Smart Greybox Fuzzing. — 2018. — 1811.09447.

[45] Szekeres Laszlo. GCC-based instrumentation for afl-fuzz. — Access mode: https://github.com/AFLplusplus/AFLplusplus/blob/stable/instrumentation/README.gcc_plugin.md (online; accessed: 2022-05-18).

[46] Wang Tielei, Wei Tao, Gu Guofei, and Zou Wei. TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection // 2010 IEEE Symposium on Security and Privacy. — 2010. — P. 497–512.

[47] Toffola Luca Della, Staicu Cristian-Alexandru, and Pradel Michael. Saying 'Hi!' is not enough: Mining inputs for effective test generation // 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE). — 2017. — P. 44–49.

[48] Tsankov Petar, Dashti Mohammad, and Basin David. SECFUZZ: Fuzz-testing security protocols // 2012 7th International Workshop on Automation of Software Test, AST 2012 - Proceedings. — 2012. — June.

[49] Wang Pengfei and Zhou Xu. SoK: The Progress, Challenges, and Perspectives of Directed Greybox Fuzzing // CoRR. — 2020. — Vol. abs/2005.11907. — arXiv : 2005.11907.

[50] Wen Hung-Pin (Charles), Wang Li-C., and Cheng Kwang-Ting (Tim). CHAPTER 9 - Functional verification // Electronic Design Automation / ed. by Wang Laung-Terng, Chang Yao-Wen, and Cheng Kwang-Ting (Tim). — Boston : Morgan Kaufmann, 2009. — P. 513–573. — Access mode: https://www.sciencedirect.com/science/article/pii/B9780123743640500163 (online; accessed: 2021-12-09).

[51] Zalewski Michal. american fuzzy lop. — 2021. — Access mode: https://github.com/google/AFL (online; accessed: 2021-12-09).

[52] ef_vi User Guide. — Access mode: https://docs.xilinx.com/v/u/en-US/SF-114063-CD-10_ef_vi_User_Guide (online; accessed: 2022-05-18).

[53] ld.so(8) — Linux manual page. — Access mode: https://man7.org/linux/man-pages/man8/ld.so.8.html (online; accessed: 2022-05-18).

[54] AFL contributors. AFL User Guide. — Access mode: https://afl-1.readthedocs.io/en/latest/user_guide.html (online; accessed: 2022-05-18).

[55] AFL++ contributors. american fuzzy lop++ - Advanced Persistent

Graphing. — Access mode: https://github.com/AFLplusplus/AFLplusplus/blob/stable/afl-plot (online; accessed: 2022-05-18).

[56] AFL contributors. Instrumenting programs for use with AFL. — 2018. — Access mode: https://github.com/google/AFL#3-instrumenting-programs-for-use-with-afl (online; accessed: 2022-05-18).

[57] Fuzzing community. Recent Papers Related To Fuzzing. — 2021. — Access mode: https://wcventure.github.io/FuzzingPaper/ (online; accessed: 2021-12-09).

[58] The LLVM team. libfuzzer. — 2021. — Access mode: https://llvm.org/docs/LibFuzzer.html (online; accessed: 2021-12-09).

[59] The Tcpdump team. libpcap. — 2021. — Access mode: https://github.com/the-tcpdump-group/libpcap (online; accessed: 2021-12-09).

[60] The kernel development community. AF_XDP. — Access mode: https://www.kernel.org/doc/html/latest/networking/af_xdp.html (online; accessed: 2022-05-18).

[61] veth(4) — Linux manual page. — Access mode: https://man7.org/linux/man-pages/man4/veth.4.html (online; accessed: 2022-05-18).