ФАЗЗИНГ В КИБЕРБЕЗОПАСНОСТИ – ОБЗОР

Владимир Костандян
Научно-исследовательский центр по цифровизации и интеллектуальной робототехнике (ЦЦИР), Политехнический институт Брагансы
Отдел информационной безопасности и разработки программного обеспечения (ИБПО), Национальный Политехнический Университет Армении
Тиаго Педроса, доктор философии
Научно-исследовательский центр по цифровизации и интеллектуальной робототехнике (ЦЦИР), Политехнический институт Брагансы
Педро Жуан Родригес, доктор философии
Научно-исследовательский центр по цифровизации и интеллектуальной робототехнике (ЦЦИР),
Политехнический институт Брагансы

Геворг Маргаров, доктор философии
Отдел информационной безопасности и разработки программного обеспечения (ИБПО),
Национальный Политехнический Университет Армении

Ключевые слова: фаззеры*, тестирование безопасности программного обеспечения, информационная безопасность, кибербезопасность*
Конспект: Широко признано, что программное обеспечение следует тестировать, предпочтительно на этапах разработки и после выпуска. Программное обеспечение может быть простым исполняемым файлом, приложением, службой или даже операционной системой. В настоящее время требования к тестированиям возросли и охватывают практически все технологические области. Тестирование можно рассматривать как способ проверки соответствуют ли функциональность, результаты и поведение программного обеспечения ожидаемым. Тестирование программного обеспечения также помогает выявить ошибки, пробелы или отсутствующие требования в отличие от фактических требований. Тестирование может быть сделано вручную или с помощью автоматизированных инструментов. Ручные тесты медленные, дорогие и требуют много знаний в этой области. Автоматизированные тесты быстрее, дороже, а также до тестирования должны быть сделаны конфигурация, настройка и много работы. Использование подхода фаззинга позволяет обнаруживать новые слабые места в системе защиты, которые не основываются на предыдущих знаниях, сигнатурах или индикаторов компромисса (IoC), как подход традиционных инструментов оценки уязвимости. Эта статья посвящена обзору фаззеров и тому, как исследователи кибербезопасности могут использовать их для проведения тестов уязвимости полуавтоматического программного обеспечения. Кроме того, наиболее используемые фаззеры проанализированы и классифицированы по типу тестов, с целью помочь исследователям выбрать правильный фаззер. Это исследование будет продолжено путем создания виртуального испытательного стенда, состоящего из систем с уязвимыми программными обеспечениями, которые можно будет протестировать комбинированным образом с использованием проанализированных фаззеров и сравнить результаты с традиционными инструментами оценки уязвимости.

1. ВВЕДЕНИЕ

Приложения тестируются с использованием различных методов и технологий, чтобы увеличить охват тестирования. Разные методы имеют одинаковую корневую логику: имеют некоторые входные данные и пытаются сделать так, чтобы приложение повредилось или неправильно работало. При рассмотрении рисков кибербезопасности последствия могут быть огромными - утечка данных, потеря или изменение ценной информации, повышение привилегий и т. д. Фаззинг- это автоматизированный процесс поиска ошибок в программном обеспечении путем подачи различных перестановок данных в целевую программу до тех пор, пока одна из этих перестановок не обнаружит ошибку, изменение на выходе или даже изменение поведения при выполнении программного обеспечения [30]. Это старый процесс, который начинает более широко использоваться различными участниками кибербезопасности, от хакеров, которые искали уязвимости для эксплуатации, до защитников, которые пытаются найти и исправить ошибки. И в эпоху, когда любой может раскрутить мощные вычислительные ресурсы для атаки приложения жертвы с ненужными данными в поисках ошибки, это стало важной подачей инструмента в нарастающем использовании уязвимости нулевого дня [37]. Эта статья посвящена обзору фаззеров и тому, как исследователи кибербезопасности могут использовать их для автоматизации тестов, связанных с анализом уязвимостей программного обеспечения. Оставшаяся статья организована следующим образом: в Разделе 2 дается обзор, объясняющий методы тестирования безопасности и быстрые фазы, типы и методы, и возобновляется использование фаззинга в кибербезопасности, в разделе 3 представлена характеристика фаззеров для кибербезопасности, где наиболее часто используемые фаззеры присутствуют по способу использования, может быть сделано специалистами по кибербезопасности, от локального тестирования, удаленного тестирования и других специализированных приложений, до сетевых протоколов или веб-браузеров и методов тестирования. Наконец, Раздел 4 подводит итог заключению статьи и будущей работе.

2. ПРЕДВАРИТЕЛЬНАЯ ИНФОРМАЦИЯ

Разработка программного обеспечения направлена на удовлетворение функциональных требований. Функциональное тестирование программного обеспечения обычно

используется для оценки соответствия требованиям и выявления тех, которые не соответствуют этим требованиям.

Тем не менее, другие факторы, такие как производительность (способность поддерживать количество активных пользователей, работающих одновременно) и надежность (способность удовлетворять требованиям в течение определенного периода времени без прерывания или отказа), важны для пользователей, особенно в критически важные приложения, например, которые развернуты в аэрокосмическом, военном, медицинском и финансовом секторах.

С этой целью функциональное тестирование может быть дополнено другими формами тестирования программного обеспечения, включая модульное тестирование, интеграцию, регрессию, производительность и безопасность на различных этапах жизненного цикла разработки программного обеспечения, причем все они направлены на выявление дефектов программного обеспечения, так что они могут быть исправлены [38].

2.1.Тестирование безопасности

Идея тестирования безопасности заключается в выявлении ошибок и избавлении от нежелательного и неожиданного поведения системы. Тестирование должно подтвердить, что система работает в соответствии с требованиями (спецификациями), а тестирование безопасности должно подтвердить надежность этой системы. Из-за огромного количества услуг, предоставляемых в Интернете, в жизненно важных системах происходит все больше нарушений безопасности [22].

На высоком уровне методы тестирования безопасности можно разделить на три типа: белый ящик, серый ящик, черный ящик, в зависимости от знаний и информации о тестируемой системе [30].

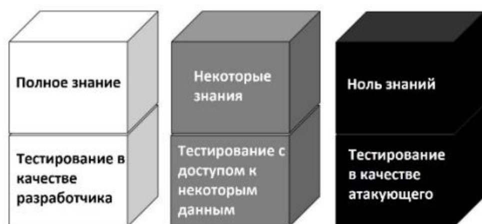

Рисунок 1: Методы тестирования

Белый ящик: пример тестирования белого ящика - обзор исходного кода. Проверка исходного кода может быть сделана вручную, строка за строкой, но это не практично для более широкой программы. Требуется помощь автоматизированных инструментов,

чтобы просмотреть весь код, чтобы найти подозрительный код, который может содержать ошибки.

Серый ящик: тестирование серого ящика располагается непосредственно между тестированием белого и черного ящика. Это делается, когда тестер обладает более глубокими знаниями о системе, чем обычный черный ящик, но не имеет доступа к исходному коду или не имеет цели проведения теста белого ящика. Например, тестирование с использованием метода «серого ящика» может включать в себя подход «черного ящика» с дополнительной обратной инженерией двоичного файла.

Черный ящик: тестирование черного ящика выполняется, когда у тестировщика есть готовая к использованию программа, и они могут сделать входные данные для приложения и затем наблюдать результаты. Нет знаний о внутренней работе черного ящика. Есть только вход и выход, нет исходного кода и нет дополнительной информации о работе цели.

На рисунке 1 показаны методы тестирования в зависимости от знаний о цели. Методы тестирования «серого» и «черного ящика» по сравнению с «белым ящиком» требуют больше времени и ресурсов. Они также сталкиваются с некоторыми проблемами, касающимися моделирования памяти и доступа к коду. Подводя итоги, мы говорим, что если целевая система прозрачна, процесс тестирования будет проще, и обратное тоже верно.

Существуют различные методы обнаружения уязвимостей для приложений тестирования безопасности. Методы используются для обнаружения общих ошибок и ошибок системы, службы, приложения или любого типа тестируемого программного обеспечения. Методы обнаружения уязвимостей включают следующие типы: статический анализ, динамический анализ, символьное выполнение и фаззинг.

Основная идея статического анализа - это интерактивная пошаговая проверка кода. Оно может быть ручным, полуавтоматическим и для некоторых испытаний полностью автоматизированным. Статически наиболее распространенный метод проверки и изучения программного кода выполняется без запуска самого программного обеспечения [18]. Процесс заключается в проверке кода (или его части) против соответствующей документации и лучших практик и шаблонов кодирования.

Хотя для ручного анализа кода люди работают очень медленно, и по этой причине задействованы автоматизированные инструменты, помогающие повысить производительность и добиться более быстрого тестирования. Ошибки, обнаруженные на более ранней стадии выпуска, дешевле исправить. Тем не менее, метод дает некоторые ложноположительные результаты в отношении того, что анализ выполняется

с использованием некоторого сопоставления с образцом, лексического анализа, проверки моделей и аналогичных методов [24, 15].

Хотя для ручного анализа кода люди работают очень медленно, и по этой причине задействованы автоматизированные инструменты, помогающие повысить производительность и добиться более быстрого тестирования. Ошибки, обнаруженные на более ранней стадии выпуска, дешевле исправить.

Однако метод дает некоторые ложноположительные результаты в том смысле, что анализ выполняется с использованием некоторого сопоставления с образцом, лексического анализа, проверки моделей и аналогичных техник [24, 15].

Общий подход в технике динамического анализа, в противоположность статическому анализу, это выполнение целевого приложения для проверки состояний исполнения, основанных на поведении приложения во время выполнения [18]. Одним из главных преимуществ динамического анализа является обеспечение возможности анализов, когда исходный код программы недоступен. Однако обнаружение логических ошибок и покрытие кода с использованием метода динамического анализа сложно реализовать, и оно имеет низкую скорость и высокие требования [24].

Другой метод анализа программного обеспечения - это символическое выполнение, при котором выполнение программы моделируется с использованием символов вместо фактических значений для входных данных.

Используя эти символы, вывод программы описывается как логическое или математическое выражение [18]. Выполнение выполняется со всеми возможными входными значениями, что приводит к созданию потокового графа, который определяет точку принятия решения для каждого потока для анализа поведения программы. Для больших программ возможные пути растут, в то же время создавая проблему взрыва пути [9].

2.2. ФАЗЗИНГ

Одним из хорошо известных методов автоматизации тестирования является фаззинг.

Основная идея фаззинга заключается в заполнении входных данных программы данными для обнаружения сбоев или уязвимостей в программном коде. Ключ должен доставить эти данные через различные интерфейсы связи в программе. Эффективный фаззер должен генерировать полуадекватные входные данные, которые «достаточно действительны», так как они не отклоняются напрямую синтаксическим анализатором, но создают непредвиденное поведение глубже в программе и являются «достаточно

недействительными», чтобы выявлять тупиковые ситуации, которые не были должным образом решены.

Например, приложение считает возраст пользователя по заданной дате рождения. Когда пользователь вводит данные, приложение считает возраст. Но что произойдет, если пользователь вместо 31 декабря передает 32 или 0? Если приложение не было реализовано безопасно, программа может аварийно завершить работу и привести к проблемам. Фаззинг - это искусство автоматического поиска ошибок, и его роль состоит в том, чтобы находить недостатки реализации программного обеспечения и пытаться их идентифицировать. Самое раннее упоминание о фаззинге было в 1989 году связано с профессором Бартоном Миллером и его классом, который разработал фаззер для тестирования приложений UNIX [31].

Фаззинг был обнаружен почти случайно, когда Бартон испытывал электромагнитные помехи при использовании компьютерного терминала во время сильного шторма. Это приводило к тому, что строки случайных символов вставлялись в командную строку по мере ввода пользователем, что приводило к аварийному завершению работы ряда приложений. Неспособность многих приложений надежно обрабатывать этот случайно искаженный ввод привела Миллера и других к тому, чтобы разработать формальный режим тестирования, и они разработали инструменты специально для проверки устойчивости приложения к случайному вводу [31].

Подводя итог, можно сказать, что использование фаззеров для тестирования не является полностью автоматизированным подходом, по крайней мере, на данный момент, так как он необходим для настройки некоторых параметров, которые контролируют генерацию входных данных, подлежащих тестированию, и некоторое взаимодействие необходимо для анализа результатов.

По словам Майкла Саттона, «не имеет значения, что вы размышляете или какой подход вы выбираете, всегда применяются одни и те же базовые фазы» [30].

Процесс фаззинга обычно делится на шесть основных этапов, не зависящих от цели и формата ввода, как это показано на рисунке 2. Этапы: идентификация цели, идентификация ввода, генерация данных, выполнение нечетких данных, мониторинг исключений и определение возможности использования.

Рисунок 2: Базовая модель фаззера. Адаптировано из [30]

Первый шаг - это идентификация цели, процесс, когда принимается решение о системе тестирования, если это стек протоколов, приложение, формат файла или веб-браузер и т. д. Имея эту информацию, он должен распознавать входной вектор программы, включая заголовки, переменные окружения и т. д., что является этапом идентификации входа.

Принимая во внимание тип и размер входных данных, третий шаг - это генерация входных данных для полу-допустимых входных данных, которые выявят тупиковые ситуации, которые не были должным образом рассмотрены. Теперь сгенерированные пакеты входных данных должны быть переданы целевому приложению, которое является этапом «отправки входных данных на целевой». Принимая во внимание входные данные, пришло время записать поведение цели, завершив предпоследний этап: мониторинг исключений.

Если заметили некоторые ошибки, сбои, повреждения или другие неожиданные действия, то это - долгожданный результат. Этот этап называется определением эксплуатируемости, проверкой, может ли обнаруженная ошибка использоваться в будущем. Обычно это полуручной процесс, требующий специальных знаний [30].

2.3.Фаззинг в кибербезопасности

Ошибки, допущенные людьми во время разработки, могут привести к сбоям программного обеспечения. Некоторые ошибки незначительны, но некоторые довольно опасны и требуют постоянной оценки и мониторинга. Поэтому жизненно важно обеспечить эффективность и безопасность разработанного программного обеспечения.

Тестирование помогает узнать больше о стабильности, надежности, безопасности и многих других приложениях. Это также снижает общий уровень рисков, предотвращает сбои программных приложений, проверяет эффективную работу и функциональность программного продукта, делает программное обеспечение более безопасным, подтверждает, что пользовательская среда соответствует необходимым нормативным стандартам. Тестирование приложения, системы, сети или программного обеспечения помогает улучшить качество продукта, внеся в него уверенность.

Резкое увеличение масштабных кибератак за последние несколько лет, многие из которых приводят к значительным финансовым и репутационным потерям, а также к рекордным штрафам, подчеркивает то, что делается недостаточно, и нынешние методы кибербезопасности неэффективны.

В большинстве случаев технология фаззинга используется для обнаружения уязвимостей в программных приложениях, выполненных до выпуска рабочей версии приложения, чтобы обеспечить качественное поведение приложения во время выполнения в непредсказуемых сценариях. Так что это способ повысить уровень безопасности приложения.

В остальных случаях тестирование проводится после того, как у вас есть готовое и работающее приложение. Из-за большого размера используемого кода часто нецелесообразно выполнять ручные проверки на наличие критических ошибок и уязвимостей в программном обеспечении. Тестирование приложения только человеческим мозгом очень медленно. Методы, используемые до сих пор для исправления функциональных ошибок, в частности, еще не обнаруженных уязвимостей, очень дороги [8, 5].

Это может сделать любой, у кого есть доступ к приложению. Доступ к исходному коду не требуется. По сравнению с другими методологиями обнаружения уязвимостей требуется очень мало опыта (по крайней мере, для выявления основных дефектов) [11].

Различные люди и организации по всему миру проводят аудит программного обеспечения. Они могут сделать это для обеспечения качества или в качестве хакеров, которые хотят найти ошибки для получения прибыли. Подводя итоги работы, стало ясно, что нечеткое тестирование используется для выявления уязвимостей в безопасности приложений, и это дает возможность для поиска ошибок. Кроме того, это один из лучших методов для автоматического тестирования.

Помимо предоставления различных преимуществ тестировщикам, разработчикам и конечным пользователям, фаззинг также имеет ряд недостатков. Хотя на данный момент реверс-инжиниринг, контрольно-проверочный код, анализ загрязнения (метод,

который проверяет регистры и области памяти, которые могут быть изменены) [7, 16] и другие методы включены и улучшены для частичной поддержки проблемы покрытия кода, потому что «черный ящик» часто обеспечивает низкий охват кода на практике [11, 6, 39].

В общем, процесс генерации тестовых случаев для фаззинга черного ящика для охвата большого числа путей в целевой программе является трудоемким. Не существует единого уникального подхода к тестированию, который может обеспечить 100-процентное покрытие для каждого случая, входных данных и пути в цели. Лучший подход фаззинга - это комбинация фаззеров. Для фаззинга, для его эффективного выполнения требуется значительное количество времени. Кроме того, сложно анализировать сбои при удаленном тестировании черного ящика практически без информации с целевого сервера или приложения. В других случаях, когда целевая программа не дает сбоя или сообщает об ошибках, мы не можем быть уверены, что ошибки нет. Кроме того, недостатки управления доступом очень трудно найти с помощью фаззинга, учитывая тот факт, что фаззер обычно ищет сбои и неожиданное поведение. Выявить многоэтапные уязвимости - еще один сложный процесс для фаззера [30].

3. ХАРАКТЕРИСТИКА ФАЗЗЕРОВ КИБЕРБЕЗОПАСНОСТИ

В этом разделе описаны большинство используемых фаззеров и их известные области применения, в то же время учитывая, что мы говорим только о некоторых фаззерах, поскольку невозможно охватить все фаззеры.

Существуют различные фаззеры для разных программных систем. В этом разделе описаны некоторые из наиболее популярных и эффективных фаззеров, классифицированных по области тестируемого приложения, с учетом эффективности фаззера.

Фаззеры для различных областей применения делятся на две категории: дистанционный и локальный фаззинг. Дистанционный означает, что программное обеспечение прослушивает сетевые интерфейсы, и тест будет выполнен с дистанционного компьютера. Локальный означает, что выполнение программного обеспечения, подлежащего проверке, и фаззера выполняется на одном и том же компьютере. Кроме того, фаззеры могут быть классифицированы с учетом входных данных: на основе поколения и мутации. Фаззинг на основе генерации обычно нацелен на один тип ввода, используя предопределенную грамматику для генерации ввода. Они также известны как умные фаззеры. Фуззеры на основе мутаций, как правило, проще в настройке и использовании, поскольку понимание формата / структуры данных

является необязательным. Примером мутации может быть добавление случайных битов или переключение битов.

В следующих разделах проводится анализ фаззеров для локального и дистанционного тестирования. Краткое описание характеристики представлено в таблице 1.

3.1.Локальное

Таблица 1: Характеристика фаззеров

Тип	Фаззер	Покрытие	Методы тестирования	Лицензирование	Платформа
Локальный	AFL libFuzzer	Инструментарий, руководствуемый генетическим алгоритмомй	Черный Черный	Открытый исходный код	Unix/ Windows Unix/ Windows
	Eclipser	Направленное покрытие, поколение мутаций	Серый	Открытый исходный код	Unix
	IOCTL	Конколическое тестирование	Черный	Открытый исходный код	Windows
	Charm	Драйвер ядра	Серый	Открытый исходный код	Android
Дистанционный	BFuzz jsfunfuzz domato Peach	Основано на поколении Основано на грамматикедля JavaScript Общая генерация грамматики	Серый Серый Белый Черный	Открытый исходный код Открытый исходный код Открытый исходный код Открытый исходный код	Unix Unix/ Windows Unix/ Windows Unix/ Windows

		двигатель для: DOM Поколение / мутация		/ коммерческая	
	Sulley	На основе вводных знаний поколение / мутация	Черный	Открытый исходный код	Unix
	Boofuzz	На основе вводных знаний поколение / мутация	Черный	Открытый исходный код	Unix/ Windows
	Fuzzotron	TCP/UDP фаззинг	Черный	Открытый исходный код	Unix/ Windows
	FFW	Сеть на основе обратной связи	Серый	Открытый исходный код	Unix/ Windows
	TLS-attacker	SSL & TLS Грамматическая мутация	Черный	Открытый исходный код	Unix/ Windows
	TumbleRF	Радиопротокол и драйвер	Черный	Открытый исходный код	Unix
	LL-Fuzzer	NFC на Android	Черный	Открытый исходный код	Ubuntu

Очень известным и используемым фаззером для локального тестирования файлов является американская нечеткая обкорнать (АНО) /American fuzzy lop AFL/, инструментарий, руководствуемый генетическим алгоритмом фаззера. Он использует генетические алгоритмы для эффективного увеличения охвата кода тестовых случаев [42], [43]. Используя инструментарий кода [2], он принимает решения об интересных путях. Этот инструментарий может быть введен либо во время компиляции, либо через модифицированный QEMU [36]. Когда исходный код недоступен, АНО предлагает поддержку быстрого инструментария двоичных файлов черного ящика, который является QEMU [43].

Другой особенностью АНО является использование постоянного режима [41], который позволяет чрезвычайно быстро выполнять фаззинг многих программ с минимальными изменениями кода. Фаззер АНО обрабатывает ошибки в реальных случаях использования, таких как анализ изображений, видео и других мультимедийных файлов, библиотеки

поддержки сжатия и так далее. Инструмент может работать с приложениями, написанными на C, C ++ или Objective-C, скомпилированными с использованием gcc или clang. AHO работают не только в Linux-подобных системах, но также в операционных системах Windows, MacOS X и Solaris. Кроме того, существует множество модифицированных версий AHO, таких как Unicorefuzz [28], которые могут выполнять фаззирование модулей ядра, используя основанные на эмуляции методы фаззинга [29], WinAHO, еще один результат AHO, который хорош для фазирования двоичных файлов Windows.

Тем не менее AHO фаззер не использует синтаксис и оптимизирован для компактных контейнеров, таких как архивы, мультимедийные файлы и т. д.

В качестве эволюционного механизма фаззинга используется libFuzzer, который является интеллектуальным фаззером, ориентированным на покрытие.

Тестирование библиотек с относительно небольшими входными данными, например, инструменты сравнения регулярных выражений, анализаторы текстового или двоичного формата, сжатие или сеть, libFuzzer может работать с хорошими результатами. Единственное условие здесь - это то, что библиотечный код не должен аварийно завершать работу при неверных входах [26]. На основе простых входных данных libFuzzer генерирует случайные мутации. Когда в тестовом коде мутация достигает открытого пути, этот случай сохраняется для дальнейшей адаптации. Когда тест библиотеки кода принимает сложные структурированные данные, libFuzzer менее эффективен. ibFuzzer использует набор образцов входных данных для тестируемого кода, и этот набор в идеале должен заполняться разнообразной коллекцией действительных и недействительных входных данных. Например, в графической библиотеке стартовый регистр может содержать много разных небольших файлов PNG / JPG / GIF.

LibFuzzer, похожий на другие эволюционные фаззеры, работает на MacOS и Linux OS, но в других операционных системах также возможна компиляция [27] с использованием специального флага.

AFL и LibFuzzer не ограничены одним типом ввода и не требуют грамматических определений. К сожалению, синтаксический слепой фаззер без входной грамматики имеет некоторые недостатки. Для сложных типов ввода с хорошо известной мутацией (перемещение или удаление блоков данных, добавление случайных байтов) может привести к неэффективному размытию, отклоняя неверный ввод с ранней стадии синтаксического анализа. Тем не менее, в отношении документации libFuzzer от Google [25] libFuzzer может использовать пользовательский ввод, предоставленный пользователем, и превращаться в грамматический фаззер.

Еще одним инструментом нечеткого тестирования на двоичной основе является Eclipser, который использует недавно обновленную методику генерации тестовых примеров, называемую конколическим тестированием типа «серый ящик» [10]. Он обеспечивает многие преимущества символического размытия, без высоких вычислительных затрат и затрат памяти при использовании только легких инструментов. Eclipser эффективно генерирует контрольные примеры для покрытия пути. Он опирается на стратегию поиска поколений, при которой выполнение одной программы генерирует тестовые случаи путем разрешения каждой условной ветви, обнаруженной во время выполнения. В настоящее время Eclipser может работать на трех широко используемых архитектурах: x86, x86-64 и ARMv7 [10].

Для фаззинга драйверов используется IOCTL, инструмент, предназначенный для фаззинга драйверов ядра Windows. IOCTL - это аббревиатура управления В/В (входа/выхода) /I/O (input/output)/. Это системный вызов для устройств ввода-выхода и других операций, которые не могут быть выражены обычными системными вызовами [23]. В настоящее время IOCTL фаззер может работать на Windows XP, Vista, 7, 2003 server, 2008 server как в x32, так и в x64 архитектурах.

Смартфоны и планшеты являются неотъемлемой частью повседневной жизни. Мобильные устройства, включая разнообразный набор входных и выходных частей, таких как камера, аудио, графический процессор и другие датчики. Будучи встроенными в аппаратное обеспечение, эти системы постоянно связаны с ним. Эта корреляция обеспечивает драйверы устройств, которые работают в ядре операционной системы, и в то же время являются источником многих серьезных уязвимостей [44]. Для фаззинга драйверов в мобильных системах Charm [40] фаззер представляет собой решение, которое облегчает динамический анализ драйверов устройств.

3.3. Дистанционные фаззеры

Хорошо известная основа для фаззинга в браузере - это BFuzz [32]. Он принимает входной файл и проходит несколько тестовых случаев, сгенерированных domato [14]. Механизм генерации в основном не зависит от приложения и, следовательно, может использоваться в других фаззерах (не DOM). Фаззер движка JavaScript - это jsfunfuzz, основанный на грамматике черный фаззер. jsfunfuzz может работать в оболочке JavaScript, и он специально разработан для тестирования движка SpiderMonkey JavaScript [33]. Другим примером JavaScript фаззера является Jsfunfuzz, который может эффективно находить ошибки, связанные с корректностью, а также ошибки, которые вызывают сбои в различных движках. Большая часть кода, за исключением генерации тестовых случаев, написана на Python. С 2007 по 2015 год, используя jsfunfuzz и DOMfuzz, Джесси Рудерман, основатель этих фаззеров [34], обнаружил около 6450 ошибок в Firefox. 790 из них были оценены как критические [35]. DOMFuzz тестирует движки браузера через вызовы

API DOM. Тестовые модули могут работать как на основе мутаций, так и на основе генерации. jsfunfuzz можно запускать как на платформах Linux, так и на Windows.
Domato используется для генерации входных данных для размытия DOM. Будучи частью Google Project Zero [14], который генерирует образец входных данных с нуля, используя набор грамматик, описывающих структуру HTML / CSS, а также различные объекты, свойства и функции JavaScript [17].
Остальная часть фаззера - это основной скрипт, который анализирует аргументы и использует базовый движок для создания шаблонов, особенно для DOM, и набор грамматик для генерации кода HTML, CSS и JavaScript.
Другое подмножество в удаленной категории - фаззеры сетевых протоколов, которые помогают в фаззинге приложений, использующих сетевые протоколы, такие как HTTP, SSH, SMTP и т. д. По своей сути, сетевой фаззер включает в себя отправку полуадекватных пакетов приложений на сервер или клиент. Peach - это кроссплатформенная структура фаззинга, которая использует автоматическое генеративное и мутационное моделирование для генерации тестовых случаев для выявления ошибок. Фаззинг с Peach требует предварительного знания протокола для генерации и изменения входных данных. Сам процесс фаззинга интегрирован с возможностями отчетности и дает исчерпывающие результаты [3]. Peach может выявить недостатки безопасности в широком диапазоне целей тестирования: файлы, сетевые протоколы, драйверы и т. д. [19]. Peach - это сообщество разработчиков с открытым исходным кодом, имеющее ограниченный набор функций.
Для тестирования сетевого протокола известен еще один фаззер по имени Sulley. Он предоставляет несколько расширяемых компонентов для построения описаний протоколов. Контролируя сеть и методично регистрируя, учитывая официальную документацию [4], Sulley классифицирует обнаруженные неисправности. Он может параллельно выполнять фаззеры, увеличивая скорость теста, а также отслеживать тестовый случай, вызывающий сбой. Однако, это идет со многими ошибками и не поддерживается в настоящее время. Одним из успешных форков Sulley является Boofuzz. Помимо того, что boofuzz [21] является преемником уязвимой структуры фаззинга, он стремится к расширяемости.
Кроме того, существует множество сетевых фаззеров, например, Fuzzotron, который является сетевым фаззером, поддерживающим TCP, UDP и многопоточность [13], инфраструктура Mutiny фаззинг, которая воспроизводит PCAP (интерфейс прикладного программирования (ИПП) для захвата сетевого трафика) через мутационный фаззер [20]. Mutiny берет законный образец трафика, мутирует его, и после использования этих мутаций генерирует трафик к хосту назначения. Для выполнения мутаций Mutiny использует Radamsa [1].

Еще одна структура фаззинга сетевых протоколов –Fuzzing For Worms (FFW), которая ориентирована на фаззинг сетевых серверов. По сравнению с альтернативами FFW поддерживает осторожные протоколы, и, как правило, модификация нечеткой цели не требуется [12]. А также, с добавлением аппаратного обеспечения, он имеет фаззинг на основе обратной связи и может фаззить клиентов сети. И это было представлено на конференции по безопасности в Зоне 41 Швейцарии.

Существуют и другие средства фаззинга сетевых протоколов: хорошими примерами фаззинга библиотеки TLS являются TLSAttacker и tlsfuzzer, которые предоставляют различные тестовые наборы для тестирования. Для ВЧ (радиочастотных) систем фаззинга используется TumbleRF, даже предоставляющий API для тестирования протоколов, радиоприемников и драйверов. Приложение NFC Fuzzer на устройствах Android является LL-Fuzzer.

Подводя итоги классификации и сравнения фаззеров, мы получили представление об известных и наиболее часто используемых фаззерах. Учитывая вышеупомянутые фаззеры, все, кроме Peach, имеют открытый исходный код. Peach предлагает ограниченное сообщество с открытым исходным кодом и полнофункциональный коммерческий продукт с поддержкой.

4. ЗАКЛЮЧЕНИЕ

Из-за большого размера приложений и программ часто нецелесообразно выполнять ручные проверки на наличие ошибок и уязвимостей в программном обеспечении. Человеческий мозг очень медленный для тестирования приложения только вручную.

Фаззинг можно назвать одним из наиболее важных методов тестирования, выполняемых для обеспечения качества и надежности программного продукта. Цель фаззинга - попытаться найти уязвимости, ошибки и проблемы в программном обеспечении, которые могут повлиять на его безопасность и позволить различным угрозам преодолеть систему. Люди, использующие разные виды фаззеров, ищут одно и то же: уязвимости. Этот подход может привести к поиску новых уязвимостей, которые не основаны на предыдущих знаниях, сигнатурах или индикаторах компромисса (IoC) в качестве подхода традиционных инструментов оценки уязвимости.

В этой статье был проведен опрос о фаззерах и о том, как исследователи кибербезопасности могут использовать их для полуавтоматического тестирования уязвимостей программного обеспечения. Кроме того, наиболее часто используемые фаззеры анализировались и классифицировались по типу тестов, чтобы помочь исследователям в области кибербезопасности выбрать наиболее подходящий фаззер для своего анализа.

В будущем будет создан виртуальный испытательный стенд с системами, состоящими из уязвимого программного обеспечения, которые можно комбинированным образом протестировать с использованием проанализированных фаззеров, чтобы сравнить результаты с традиционными инструментами оценки уязвимостей.

5. СПИСОК ЛИТЕРАТУРЫ

[1] AKI HELIN. Aki helin / radamsa - gitlab. https://gitlab.com/akihe/radamsa, 2017.

[2] ALLEN, G., NABRZYSKI, J., SEIDEL, E., AND DONGARRA, J. Computational science ICCS 2009: 9th International Conference Baton Rouge, LA, USA, May 25-27, 2009: proceedings. *Vox Sanguinis*, January 2014 (2009).

[3] AMINI, P. Fuzzing Frameworks. Tech. rep., 2007.

[4] AMINI, P., AND AARON, P. Sulley: Fuzzing Framework. Tech. rep., 2011.

[5] AUSTIN, A., AND WILLIAMS, L. One technique is not enough: A comparison of vulnerability discovery techniques. In *2011 International Symposium on Empirical Software Engineering and Measurement* (Sep. 2011), pp. 97– 106.

[6] BAZZOLI, E., CRISCIONE, C., MAGGI, F., AND ZANERO, S. XSS PEEKER: Dissecting the XSS Exploitation Techniques and Fuzzing Mechanisms of Blackbox Web Application Scanners. 243–258.

[7] BEKRAR, S., BEKRAR, C., GROZ, R., AND MOUNIER, L. A taint based approach for smart fuzzing. *Proceedings - IEEE 5th International Conference on Software Testing, Verification and Validation, ICST 2012* (2012), 818–825.

[8] BOEHM, B. W. Software engineering economics. *IEEE Transactions on Software Engineering SE-10*, 1 (Jan 1984), 4–21.

[9] BOYER, R. S., ELSPAS, B., AND LEVITT, K. N. Select—a formal system for testing and debugging programs by symbolic execution. *SIGPLAN Not. 10*, 6 (Apr. 1975), 234–245.

[10] CHOI, J., JANG, J., HAN, C., AND CHA, S. K. Grey-box concolic testing on binary code. In *Proceedings of the 41st International Conference on Software Engineering* (2019), ICSE '19, IEEE Press, p. 736–747.

[11] CLARKE, T. Fuzzing for software vulnerability discovery. Tech. Rep. February, 2009.

[12] DOBIN RUTISHAUSER. Github - dobin/ffw: A fuzzing framework for network servers. https: //github.com/dobin/ffw, 2017.

[13] DOI, DENANDZ. Github - denandz/fuzzotron: A tcp/udp based network daemon fuzzer. https: //github.com/denandz/fuzzotron/, 2020.

[14] FRATRIC, I. Github - googleprojectzero/domato: Dom fuzzer. https: //github.com/googleprojectzero/domato, 2017.

[15] FREITEZ, W. R. J., MAMMAR, A., AND CAVALLI, A. R. Software vulnerabilities, prevention and detection methods : a review. Tech. Rep. 215995, 2009.

[16] GANESH, V., LEEK, T., AND RINARD, M. Taint-based directed whitebox fuzzing. In *2009 IEEE 31st International Conference on Software Engineering* (May 2009), pp. 474–484.

[17] GOOGLE PROJECT ZERO. Fuzzing with grammars - the fuzzing book. https://www. fuzzingbook.org/html/Grammars.html, 2020.

[18] IEEE. IEEE Standard Glossary of Software Engineering Terminology. Tech. rep., 1990.

[19] IOACTIVE. Peach fuzzer - peach tech. https://www.peach.tech/products/ peach-fuzzer/, 2020.

[20] JAMES SPADARO, LILITH WYATT. Github cisco-talos/mutiny-fuzzer. https://github.com/Cisco-Talos/mutiny-fuzzer, 2017.

[21] JOSHUA PEREYDA. Github - jtpereyda/boofuzz: A fork and successor of the sulley fuzzing framework. https://github.com/jtpereyda/boofuzz/, 2020.

[22] JUSTIN LAVELLE - GARTNER. Gartner says data and cyber-related risks remain top worries for audit executives. Gartner Newsroom @ https://www.gartner.com/ en/newsroom, 2019.

[23] KERRISK, M. ioctl(2) - linux manual page. http://man7.org/linux/man-pages/ man2/ioctl.2.html, 2017.

[24] LI, J., ZHAO, B., AND ZHANG, C. Fuzzing: a survey. *Cybersecurity 1*, 1 (2018), 1–13.

[25] LIBFUZZER: STRUCTURE-AWARE FUZZING,
2019.

[26] LLVM. a library for coverage-guided fuzz testing, llvm 10 documentation, libfuzzer. http: //llvm.org/docs/LibFuzzer.html, 2020.

[27] LLVM. a library for coverage-guided fuzz testing, llvm 10 documentation, libfuzzer. http://llvm.org/docs/LibFuzzer.
html#developing-libfuzzer, 2020.

[28] MAIER, D. Github - fgsect/unicorefuzz: Fuzzing the kernel using unicornafl and afl++. https://github.com/fgsect/unicorefuzz, 2020.

[29] MAIER, D., RADTKE, B., AND HARREN, B.
Unicorefuzz: On the viability of emulation for kernelspace fuzzing. In *Proceedings of the 13th USENIX Conference on Offensive Technologies* (USA, 2019), WOOT'19, USENIX Association, p. 8.

[30] MICHAEL SUTTON, ADAM GREENE, AND
AMINI, P. [Book]Fuzzing Brute Force Vulnerability Discovery 2007.pdf, 2007.

[31] MILLER, B. P., FREDRIKSEN, L., AND SO, B. An empirical study of the reliability of unix utilities. *Commun. ACM 33*, 12 (Dec. 1990), 32–44.

[32] MISHRA, D. Github - rootup/bfuzz: Fuzzing browsers. https://github.com/RootUp/ BFuzz, 2019.

[33] MOZILLA FUZZING SECURITY. Github mozillasecurity/funfuzz: A collection of fuzzers in a harness for testing the spidermonkey javascript engine. https://github.com/ MozillaSecurity/funfuzz, 2020.

[34] RUDERMAN, J. Releasing jsfunfuzz and domfuzz. http:// www.squarefree.com/2015/07/28/ releasing-jsfunfuzz-and-domfuzz, 2015.

[35] RUDERMAN, J. Bug list mozilla. https:// tinyurl.com/t7eeplx, 2020.

[36] STEPHENS, N., GROSEN, J., SALLS, C., DUTCHER, A., WANG, R., CORBETTA, J., SHOSHITAISHVILI, Y., KRUEGEL, C., UC,
G. V., AND BARBARA, S. Driller: Augmenting Fuzzing Through Selective Symbolic Execution.

[37] STEVE RANGER. Inside the secret digital arms race: Facing the threat of a global cyberwar - techrepublic.
https://www.techrepublic.com/article/ inside-the-secret-digital-arms-race, 2014.

[38] TAKANEN, A., DEMOTT, J., AND MILLER, C. Fuzzing for Software Security Testing and Quality Assurance (Artech House Information Security and Privacy). Tech. rep., 2008.

[39] TALEBI, S. M. S., TAVAKOLI, H., ZHANG, H., ZHANG, Z., SANI, A. A., AND QIAN, Z. Charm: Facilitating dynamic analysis of device drivers of mobile systems. Tech. rep., 2018.

[40] TALEBI, S. M. S., TAVAKOLI, H., ZHANG, H., ZHANG, Z., SANI, A. A., AND QIAN, Z. Charm: Facilitating dynamic analysis of device drivers of mobile systems. Tech. rep., 2018.

[41] ZALEWSKI, M. New in afl: persistent mode. https://lcamtuf.blogspot.com/2015/06/ new-in-afl-persistent-mode.html, 2015.

[42] ZALEWSKI, M. american fuzzy lop. http:// lcamtuf.coredump.cx/afl, 2016.

[43] ZALEWSKI, M. Github - google/afl: american fuzzy lop - a security-oriented fuzzer. https: //github.com/google/AFL, 2016.

[44] ZHANG, H., SHE, D., AND QIAN, Z. Android root and its providers: A double-edged sword. *Proceedings of the ACM Conference on Computer and Communications Security 2015October* (2015), 1093–1104.