МЕТОД ГЕНЕРАЦИИ СЕМАНТИЧЕСКИ КОРРЕКТНОГО КОДА ДЛЯ ФАЗЗИНГ-ТЕСТИРОВАНИЯ ИНТЕРПРЕТАТОРОВ JAVASCRIPT

Козачок А.В.[1], Спирин А.А.[2], Ерохина Н.С.[3]

Цель работы: *разработка метода генерации входных данных для фаззинг-тестирования интерпретаторов JavaScript и его оценка.*

Метод исследования: *изучение закономерностей генерации данных и процента покрытия кода с целью его повышения. Предложенный метод позволяет генерировать входные данные для выявления большего количества уязвимостей при последующем фаззинг-тестировании, за счет повышения процента покрытия кода.*

Результаты исследования: *интерпретатор JavaScript является наиболее уязвимым блоком архитектуры веб-браузера, как следствие возникает необходимость постоянного наращивания объемов анализа/тестирования его исходного кода. Фаззинг-тестирование интерпретатора веб-браузера на основе сложноструктурированных входных данных, например, программного кода JavaScript, является актуальной задачей. В работе приведены уязвимости современных веб-браузеров, а также ключевые проблемы, возникающие при тестировании интерпретаторов JavaScript. Наиболее существенными проблемами являются: отсутствие общедоступных синтаксически и семантически корректных входных данных для фаззинг-тестирования, проблема преодоления внутренних механизмов фильтрации входных данных, выбор рационального алгоритма мутации данных, а также проблема повышения степени покрытия тестируемого кода. Авторами предложен метод генерации входных данных для фаззинг-тестирования интерпретаторов JavaScript, который позволяет повысить качество и скорость фаззинг-тестирования.*

Научная и практическая значимость *результатов исследования заключаются в разработке нового метода генерации входных данных для фаззинг-тестирования интерпретаторов JavaScript веб-браузеров, на основе применения нейросетевых языковых моделей, повышающий покрытие исходного кода.*

Ключевые слова: *веб-браузер, интерпретатор JavaScript, покрытие кода, уязвимости программного обеспечения, информационная безопасность.*

DOI: 10.21681/2311-3456-2023-5-80-88

Введение

В 2023 году согласно отчёту аналитического агентства Meltwater[4] в мире насчитывается 5,16 миллиарда пользователей сети Интернет, что составляет 64,4% мирового населения. По сравнению с 2022 годом количество интернет-пользователей выросло на 1,9%. 92,3% и 65,6% пользователей сети интернет используют мобильные устройства и персональные компьютеры и планшеты соответственно. На каждом из этих устройств работает веб-браузер или аналогичная программа, способная обрабатывать и отображать контент веб-сайтов. Веб-браузеры совершенствуются и становятся все более сложными, осуществляя обработку не только открытого текста и HTML, но и изображений, видео и других форматов данных.

Наибольшую угрозу безопасности веб-браузера представляют интерпретаторы JavaScript (англ. JavaScript engines). Каждый интерпретатор подобен языковому модулю, который позволяет приложению поддерживать определенное подмножество стандартов языка JavaScript. Развитие технологий приводит к постоянному усложнению структуры интерпретаторов JavaScript и увеличению их исходного кода. Данный

4 https://www.meltwater.com/en/global-digital-trends.

1 Козачок Александр Васильевич, доктор технических наук., доцент, Академия ФСО России, г. Орел, Россия, E-mail: a.kozachok@academ.msk.rsnet.ru, https://orcid.org/0000-0002-6501-2008

2 Спирин Андрей Андреевич, кандидат технических наук, Академия ФСО России, г. Орел, Россия, E-mail: spirin_aa@bk.ru, https://orcid.org/ 0000-0002-7231-5728

3 Ерохина Наталья Сергеевна, сотрудник, Академия ФСО России, г. Орел, Россия, E-mail: ens@secdev.space, https://orcid.org/0000-0002-4878-0865

факт негативно влияет на безопасность, что, в свою очередь, активизирует деятельность авторов вредоносных программ.

В последнее время большинство обнаруживаемых ошибок в программном обеспечении (ПО), связанных с удаленным выполнением кода и повышением привилегий, обнаруживаются при помощи фаззинг-тестирования [1]. Вследствие имеющихся ограничений фаззеров, осуществляющих тестирование интерпретаторов JavaScript веб-браузеров, фаззинг-тестирование может быть недостаточно эффективным. Одним из путей повышения эффективности данного процесса является совместное использование алгоритмов машинного обучения и анализа покрытия кода при тестировании с целью преодоления ключевых проблем существующих методов фаззинг-тестирования.

Безопасность интерпретаторов JavaScript веб-браузеров

Среди многих компонентов веб-браузеров интерпретаторы JavaScript представляет особый интерес для злоумышленников, поскольку их полная по Тьюрингу природа позволяет злоумышленникам создавать сложный код, содержащий уязвимости. В частности, интерпретаторы JavaScript оказались в центре внимания исследователей безопасности по разным причинам: во-первых, из соображений производительности они часто реализуются на небезопасных для памяти языках, что влечет за собой уязвимости, приводящие к повреждению памяти. Во-вторых, продолжающаяся гонка за производительностью и постоянное усложнение структуры увеличивает вероятность ошибок при разработке.

Задача интерпретатора JavaScript – анализировать и выполнять код JavaScript. В отличие от большинства других сред, интерпретатор JavaScript, встроенный в веб-браузер, должен безопасно обрабатывать ненадежные сценарии. Кроме того, он разработан с большим акцентом на производительность, чтобы обеспечить интерактивность клиентским веб-приложениям. Как это часто бывает, повышение производительности связано с увеличением сложности кода, что, в свою очередь, приводит к ошибкам программирования, которые иногда являются критическими с точки зрения безопасности. Согласно Национальной базе данных уязвимостей (NVD[5]), 43% всех уязвимостей, обнаруженных в веб-браузерах Microsoft Edge и Google Chrome, были уязвимостями интерпретатора JavaScript [2].

5 https://nvd.nist.gov/.

Хотя дизайн и реализация каждого интерпретатора JavaScript сильно различаются, все они имеют общую архитектуру и два общих свойства: во-первых, они служат стандартизированной средой выполнения для JavaScript кода; во-вторых, обеспечивают JIT-компиляцию для повышения производительности.

В то время как «классические» уязвимости, такие как переполнение буфера или использование динамической памяти после освобождения, редко встречаются в механизмах сценариев, их заменили сложные и специфичные для предметной области уязвимости.

Наиболее часто встречающиеся проблемы в обработчиках сценариев, обнаруженных за последние годы:

- ошибки, связанные с целочисленным переполнением, обычно приводящие к несанкционированному доступу к буферу памяти;
- ошибки из-за неожиданных обратных вызовов при реализации некоторых встроенных функций;
- ошибки использования после освобождения из-за того, что сборщик мусора не находит объект на этапе маркировки;
- уязвимости, возникающие из-за того, что внутренний объект или функция интерпретатора «проникают» в код приложения из-за какой-либо логической проблемы;
- уязвимости, возникающие из-за неправильной оптимизации JIT-компилятора. На сегодняшний день уязвимости были обнаружены, по крайней мере, в реализации устранения проверки границ, анализа выхода и исключении проверки типов [3].

В дополнение к описанным классам уязвимостей существует большое количество различных уязвимостей, которые в настоящее время не могут быть однозначно отнесены к какой-либо категории.

Существующие проблемы при тестировании интерпретаторов JavaScript

Фаззинг – методика тестирования, при которой на вход программы подаются невалидные, непредусмотренные или случайные данные, которые могут привести ее к аварийному завершению или неопределенному поведению. Этот метод автоматического тестирования охватывает множество граничных, а также ложных значений и использует их в качестве входных данных тестируемой программы [4].

Существующие ограничения методов фаззинг-тестирования приводят к тому, что в настоящее время этот процесс недостаточно эффективен. Фаззинг ПО со сложноструктурированными входными данными,

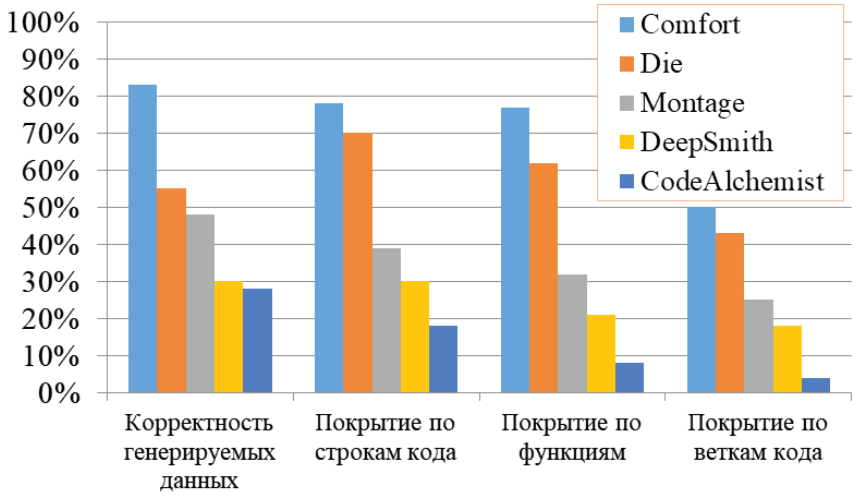

Рис. 1. Оценка существующих фаззеров интерпретаторов JavaScript

такими как программный код, сталкивается с несколькими ключевыми проблемами.

Первая проблема заключается в отсутствии общедоступных синтаксически и семантически корректных входных данных для проведения тестирования. Наличие общедоступного входного корпуса, изначально обеспечивающего высокое покрытие кода тестируемой программы, может значительно повысить эффективность всего процесса фаззинг-тестирования.

Второй является проблема преодоления внутренних механизмов фильтрации входных данных. Существующие фаззеры при генерации тестовых данных, как правило, разрушают тонкую семантику или условия, закодированные во входном корпусе, однако повышают процент покрытия кода программы. Такие тестовые данные отбрасываются тестируемой программой еще перед синтаксическим анализом и обработкой. Проверка выполняется с целью защиты программы от сбоев, вызванных некорректными входными данными. При фаззинге программного кода важным критерием при генерации или мутации входных файлов является сохранение синтаксической и семантической корректности. Эффективный фаззер должен полностью собирать тонкие условия, закодированные в высококачественном входном корпусе, таком как известные PoC-файлы, подтверждающие эксплуатацию уязвимостей [5] или модульные тесты интерпретаторов JavaScript.

Третья проблема заключается в поиске оптимального алгоритма изменения исходных данных. Стратегия генерации на основе мутаций широко используется современными фаззерами. Тем не менее, ключевой проблемой являются ответы на вопросы: как изменять и генерировать тестовые примеры, охватывающие больше программных путей как быстрее обнаруживать ошибки [6]. В частности, при выполнении мутации необходимо ответить на два вопроса: как и что мутировать. Одна мутация в нескольких ключевых позициях повлияет на поток управления выполнением. Кроме того, еще одна ключевая проблема заключается в том, как фаззеры изменяют ключевые позиции, то есть, как определить значение, которое могло бы направить тестирование на новые, еще не исследованные трассы в программе. Стратегии мутации должны быть направлены на создание высококачественных тестовых случаев, а не просто на увеличение охвата кода, чтобы можно было найти значимые, трудно обнаруживаемые ошибки. Отсутствие обратной связи по коду приводит к потере качества фаззинг-тестирования, а изменение стратегии мутации может значительно повысить эффективность фаззинга.

Четвертая проблема – это проблема повышения степени покрытия тестируемого кода. Повышение охвата кода означает повышение охвата состояний выполнения программы и повышение качества тестирования. Известно, что большее покрытие приводит к повышению вероятности обнаружения дефектов. Это подтверждается отчетом Миллера[6], который показал, что увеличение покрытия кода на 1% увеличивает процент обнаруженных ошибок на 0,92%. Однако большинство тестовых примеров охватывают только

6 C. Miller, Fuzz by number: More data about fuzzing than you ever wanted to know // in Proceedings of the CanSecWest – 2008.

некоторое, ограниченное число путей, в то время как большая часть кода не достигается. Однако существующие методы обычно фокусируются на покрытии кода, а не на уязвимом коде. Эти методы направлены на то, чтобы охватить как можно больше путей, а не исследовать пути, которые с большей вероятностью будут уязвимы. При выборе начальных значений для тестирования существующие фаззеры обычно обрабатывают все начальные входные данные одинаково, игнорируя тот факт, что пути, реализуемые различными начальными входными данными, не одинаково уязвимы. Это приводит к трате времени на тестирование безопасных, а не уязвимых путей, что снижает эффективность обнаружения уязвимостей. Как сообщается в [7], распределение ошибок в программах часто бывает несбалансированным, т. е. примерно 80% ошибок находятся примерно в 20% программного кода. В результате существующие фаззеры тратят много времени на тестирование «неуязвимых» путей, тем самым снижая эффективность фаззинга.

Метод генерации входных данных для фаззинга интерпретаторов

Метод генерации входных данных для фаззинг-тестирования интерпретаторов JavaScript веб-браузеров отличается использованием нейросетевой языковой модели, а также возможностью отслеживания информации о покрытии исходного кода.

Фаззинг на основе покрытия – широко используемый метод обнаружения ошибок и уязвимостей безопасности в программном обеспечении. Основная идея заключается в улучшении генерации будущих тестовых данных путем сбора обратной связи о текущем образце. С целью повышения эффективности подходы управляемого фаззинга требуют использования разумных мутаций, которые сохраняют особенности существующих образцов, слегка изменяя их семантику. С практической точки зрения это один из самых эффективных на сегодня типов фаззеров. На этой основе работают AFL++ [8], libFuzzer [9]. Несколько исследователей работали над разработкой различных стратегий мутации, основанных на различном поведении программы (например, фокусировке на редких ветвях, контексте вызова и т. д.) [10, 11]. Однако поведение программы резко меняется не только в разных программах, но и в разных частях одной и той же программы. Таким образом, поиск общей надежной стратегии мутации все еще остается важной открытой проблемой.

Анализ покрытия кода – действенный способ повышения эффективности фаззинга, однако нельзя отталкиваться только от него. Добившись высокой степени покрытия кода, все равно возможно пропустить критически важные участки [12].

Применение управляемого фаззинга к интерпретаторам JavaScript нетривиально, поскольку оно требует определения разумных изменений в программном коде. Вследствие чего, результаты фаззинга интерпретаторов сильно уступают результатам, достигнутым в других областях.

Эффективность этого подхода может быть повышена за счет исходного корпуса, который реализует более крупные части целевой программы, предоставляя более широкие границы для поиска входных данных.

Применение методов машинного обучения во многих исследованиях в области кибербезопасности как для обнаружения уязвимостей [13-14]; так и в фаззинг-тестировании [15-18] демонстрирует впечатляющие результаты. Преимуществом нейронных сетей является возможность обработки больших объемов данных с целью выявления закономерностей, что может быть применено для генерации сложноструктурированных данных для фаззинга.

Машинное обучение не может работать напрямую с кодом, ему необходимо подавать на вход числа, поэтому для работы с языковыми моделями необходима процедура токенизации – преобразование кода в последовательность чисел. В общем случае, на вход нейронной сети подаются не фрагменты кода или его структурные единицы, а токены – результат разбиения строки кода на непересекающиеся подстроки. Процедура токенизации для работы с нейросетями широко изучалась при решении задач завершения кода [19-20]. Однако генерация исполняемого теста является более сложной задачей, чем проблема завершения кода, которая предсказывает ограниченное количество семантически корректных лексических токенов.

В работе [2] предложена идея работы обучения языковых моделей не токенами, а AST-фрагментами и их последовательностями.

Фрагмент $T = (N, E, n| \quad |0)$ – это поддерево $T = E_i, n_i)$, где

$n_i \in NC(n_i) \neq 0;$

$N_i = \{n_i\} \cup C(n_i);$

$E_i = \{(n_i; n') \vee n' = C(n_i)\}.$

где: N – множество узлов, E – множество ребер, n_0 – корневой узел, $C(n_i)$ – непосредственный потомок n_i, где n_i – узел в T.

Данный способ сохраняет семантику в обучающем наборе, разбивая узлы AST-дерева на фрагменты, ко-

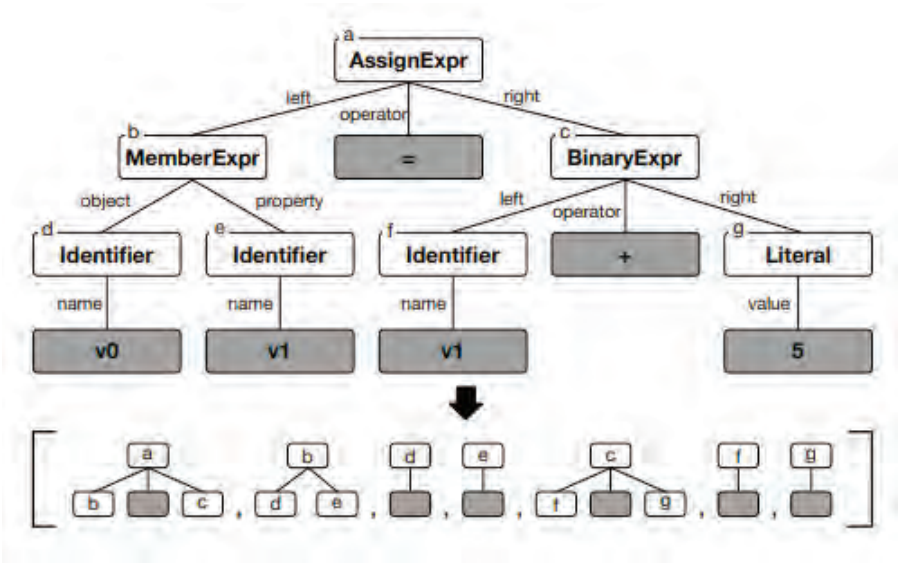

Рис. 2. Процесс фрагментации AST-дерева

торые используются в качестве лексикона для генерации кода JS. Процесс фрагментации AST-дерева представлен на рисунке Рис. 2. Процесс фрагментации AST-дерева. Используя фрагменты, инкапсулирующие структурные связи AST-деревьев, AST-дерево кодируется в последовательности фрагментов. Таким образом, возможно фиксировать глобальные отношения композиции между фрагментами кода для выбора следующего фрагмента.

Также были выявлены следующие закономерности:

1. Уязвимости интерпретаторов JavaScript часто возникают из-за того, что один и тот же js-файл может быть повторно исправлен в случае наличия нескольких ошибок;

2. Более 95% AST-фрагментов синтаксически перекрываются между регрессионными тестами интерпретаторов и PoC-фрагментами кода, запускающими CVE.

Данные факты подразумевают, что вероятность обнаружения новой уязвимости безопасности путем сборки фрагментов кода из существующих наборов регрессионных тестов, гораздо выше. Таким образом, возможно сгенерировать новый, обеспечивающий хорошее покрытие тестируемого кода, набор входных данных для проведения на нем дальнейшего фаззинг-тестирования.

Процесс фаззинг-тестирования интерпретатора предложено разделить на два больших этапа:

— генерацию качественных входных данных для фаззинга;
— непосредственно сам процесс фаззинг-тестирования.

Данное разделение способствует повышению покрытия тестируемого кода тестами, а также ускорению процесса фаззинг-тестирования. Так как генерация базы входных данных является длительным процессом, более эффективно с точки зрения временных затрат проделать этот этап один раз до начала тестирования. Предварительно сгенерированный набор входных данных для фаззинга может использоваться далее при тестировании различных интерпретаторов и за счет дальнейших мутаций сгенерированных файлов выполнять тестирование на наличие уязвимостей.

Данный подход позволяет:

— ускорить процесс фаззинг-тестирования;
— удалить избыточность из входных данных;
— обеспечить большее покрытие тестируемого кода.

Для реализации метода генерации входных данных для фаззинг-тестирования интерпретаторов JavaScript веб-браузеров, отличающегося использованием нейросетевых языковых моделей, а также управляемого информацией о покрытии исходного кода была разработана следующая архитектура (Рис. 3).

Данная архитектура позволяет генерировать новые входные данные, обеспечивающие высокое начальное покрытие кода, на основе базы регрессионных тестов для интерпретаторов.

Процесс генерации также подразделяется на два этапа:

— обучение нейронной сети;
— генерация входных данных.

1. Первый этап состоит из следующих шагов:

2. Фильтрация ошибок и формирование AST-деревьев;

3. Нормализация идентификаторов;

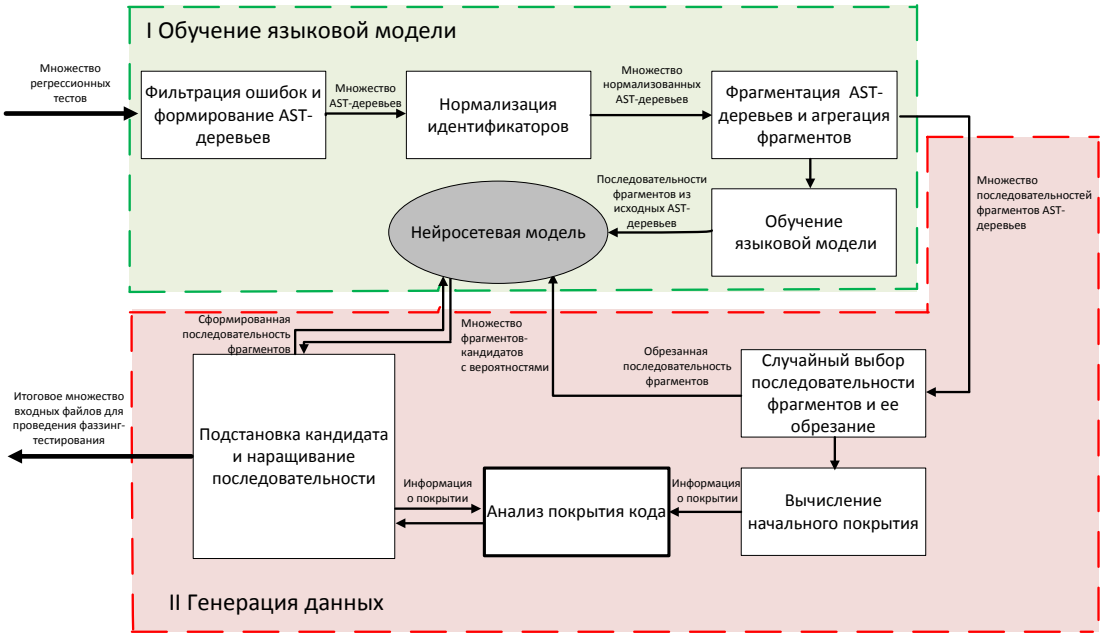

Рис. 3. Архитектура генератора входных данных

4. Фрагментация AST-деревьев и агрегация фрагментов;

Подбор гиперпараметров языковой модели и обучение нейросетевой модели.

Результатом первой фазы генерации являются:

— обученная нейросетевая модель;

— сформированное множество последовательностей фрагментов AST-деревьев.

Второй этап включает в себя следующие шаги:

1. Случайный выбор последовательности фрагментов из множества, сформированного в 1 фазе;
2. Выбор случайного фрагмента из последовательности;
3. Удаление поддерева из AST-дерева, для которого выбранный фрагмент является корневым;
4. Фиксация типа корневого фрагмента;
5. Подача на вход языковой модели обрезанной последовательности фрагментов;
6. Получение множества фрагментов-кандидатов от нейросетевой модели;
7. Отсев некорректных кандидатов по типу;
8. Наращивание последовательности по каждому фрагменту-кандидату;
9. Отсев полученных последовательностей, не прошедших валидацию;
10. Анализ прироста покрытия кода сгенерированными последовательностями;
11. Добавление в итоговое множество последовательностей, повышающих исходное покрытие кода;
12. Выбор новой последовательности из множества и повтор пунктов 3-11.
13. Минимизация полученного множества последовательностей;

В результате работы генератора формируется набор входных данных, с помощью которого можно провести более эффективное последующее фаззинг-тестирование.

Оценка покрытия тестируемого кода интерпретаторов JavaScript сгенерированными входными данными

В рамках данного исследования в качестве исходных данных была собрана база регрессионных тестов различных интерпретаторов JavaScript, была выбрана нейросеть долгой краткосрочной памяти (англ. Long Short Term Memory, LSTM) и JavaScript интерпетатор ChakraCore.

Для оценки покрытия рассматривались две широко используемые метрики: покрытие строк кода, а также покрытие функций. Две метрики соответственно измеряют среднее соотношение строк кода и функций тестируемой программы, а именно интерпретатора ChakraCore, которые выполняются во время тестового прогона. Для сбора информации о покрытии кода используются утилиты gcov [21] и lcov [22].

Используя вышеизложенный метод генерации входных данных для фаззинга интерпретаторов, удалось сгенерировать новый набор входных данных,

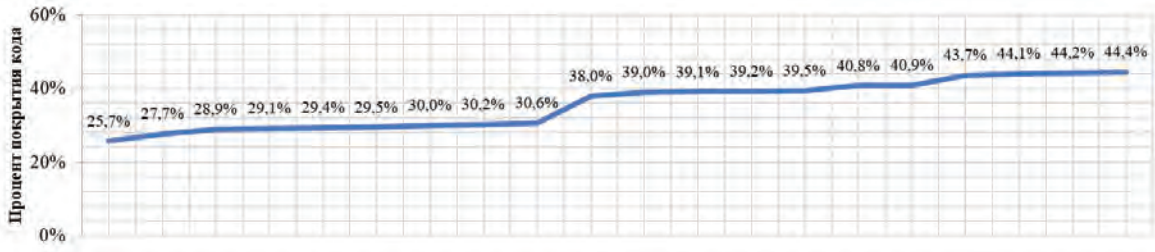

Рис. 4. График роста покрытия при запуске метода генерации

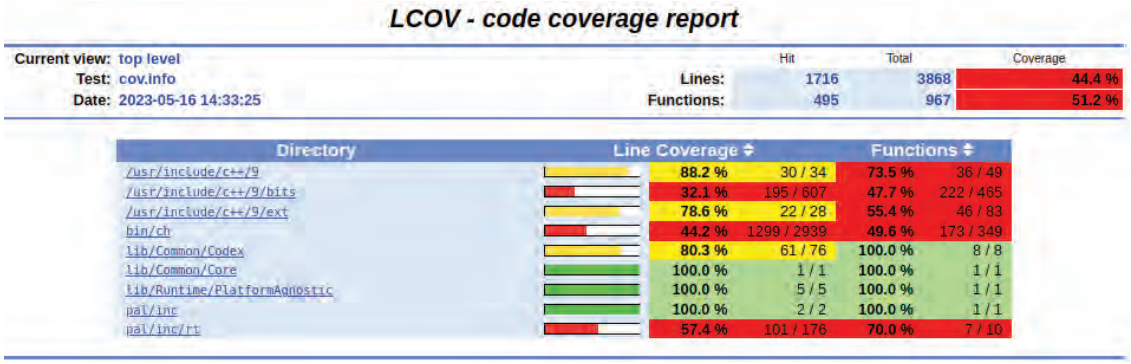

Рис. 5. Отчет утилиты lcov об увеличении покрытия кода интерпретатора

обеспечивающий покрытие 44.4 % по строкам кода и 51.2 % по функциям (Рис. 4).

Данные результаты демонстрируют возможность успешной генерации сложноструктурированных входных данных, таких как JavaScript код, для последующего фаззинг-тестирования. Преимуществом данного метода является ускорение процесса фаззинга, за счет разделения процесса тестирования на два этапа.

Выводы

Фаззинг-тестирование сложного программного обеспечения, такого как интерпретатор языка JavaScript, со сложно структурированными входными данными является актуальной и трудоемкой задачей. В работе приведены актуальные уязвимости веб-браузеров, а также ключевые проблемы, возникающие при тестировании интерпретаторов веб-браузеров. Наиболее актуальными проблемами являются: отсутствие общедоступных синтаксически и семантически корректных входных данных, проблема преодоления внутренних механизмов фильтрации входных данных, выбор рационального алгоритма мутации данных, а также проблема повышения степени покрытия тестируемого кода. Авторами предложен метод генерации входных данных для фаззинг-тестирования интерпретаторов JavaScript, который позволяет повысить качество и скорость последующего фаззинг-тестирования.

Литература

14. Bytes A. et al. Field Fuzz: In Situ Blackbox Fuzzing of Proprietary Industrial Automation Runtimes via the Network //Proceedings of International Symposium on Research in Attacks, Intrusions and Defenses (RAID). – 2023 DOI: 10.1145/3607199.3607226.
15. Lee S. et al. Montage: A neural network language model-guided javaScript engine fuzzer //Proceedings of the 29th USENIX Conference on Security Symposium. – 2020. – С. 2613-2630, DOI: 10.48550/arXiv.2001.04107.
16. Groß S. Fuzzil: Coverage guided fuzzing for JavaScript engines // Department of Informatics, Karlsruhe Institute of Technology, 2018.
17. Козачок А. В. и др. Обзор исследований по применению методов машинного обучения для повышения эффективности фаззинг-тестирования // Вестник Воронежского государственного университета, серия: системный анализ и информационные технологии. – 2021. – №. 4. – С. 83–106., DOI: 10.17308/sait.2021.4/3800.
18. C. Han. js-vuln-db, A collection of JavaScript engine CVEs with PoCs,2019. https://github.com/tunz/js-vuln-db.
19. Huang W. et al. testrnn: Coverage-guided testing on recurrent neural networks //arXiv preprint arXiv:1906.08557. – 2019, DOI: 10.48550/arXiv.1906.08557.

20. Gouveia I. P., Völp M., Esteves-Verissimo P. Behind the last line of defense: Surviving SoC faults and intrusions //Computers & Security. – 2022. – Т. 123. – С. 102920, DOI: 10.1016/j.cose.2022.102920.
21. Fioraldi A. et al. AFL++ combining incremental steps of fuzzing research //Proceedings of the 14th USENIX Conference on Offensive Technologies. – 2020. – С. 10-10.
22. Chao W. C. et al. Design and Implement Binary Fuzzing Based on Libfuzzer //2018 IEEE Conference on Dependable and Secure Computing (DSC). – IEEE, 2018. – С. 1-2.
23. Peng Chen and Hao Chen. 2018. Angora: Efficient fuzzing by principled search.
24. Österlund S. et al. Parmesan: Sanitizer-guided greybox fuzzing //Proceedings of the 29th USENIX Conference on Security Symposium. – 2020. – С. 2289-2306.
25. Козачок А. В., Николаев Д. А., Ерохина Н. С. Подходы к оценке поверхности атаки и фаззингу веб-браузеров //Вопросы кибербезопасности. – 2022. – №. 3 (49). – С. 32–43, DOI: 10.21681/2311–3456-2022-3-32-43.
26. Lin G. et al. Software vulnerability detection using deep neural networks: a survey //Proceedings of the IEEE. – 2020. – Т. 108. – №. 10. – С. 1825-1848, DOI: 10.1109/JPROC.2020.2993293.
27. Hanif H. et al. The rise of software vulnerability: Taxonomy of software vulnerabilities detection and machine learning approaches // Journal of Network and Computer Applications. – 2021. – Т. 179. – С. 103009, DOI: 10.1016/j.jnca.2021.103009.
28. Chernis B, Verma R. Machine Learning Methods for Software Vulnerability Detection. In: Proceedings of the Fourth ACM International. Workshop on Security and Privacy Analytics ACM. 2018. p. 31–39, DOI: 10.1145/3180445.3180453.
29. Zhu X. et al. Fuzzing: a survey for roadmap // ACM Computing Surveys (CSUR). – 2022. – Т. 54. – №. 11s. – С. 1-36, DOI: 10.1145/3512345.
30. Kaloudi N., Li J. The ai-based cyber threat landscape: A survey //ACM Computing Surveys (CSUR). – 2020. – Т. 53. – №. 1. – С. 1-34, DOI: 10.1145/3372823.
31. She D, Pei K, Epstein D, Yang J, Ray B, Jana S. NEUZZ: Efficient Fuzzing with Neural Program Smoothing; IEEE Symposium on Security & Privacy; 2019 – с. 38, DOI: 10.1109/SP.2019.00052.
32. Allamanis M. et al. A survey of machine learning for big code and naturalness //ACM Computing Surveys (CSUR). – 2018. – Т. 51. – №. 4. – С. 1-37, DOI: 10.1145/3212695.
33. Karampatsis R. M. et al. Big code! = big vocabulary: Open-vocabulary models for source code //Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. – 2020. – С. 1073-1085, DOI: 10.1145/3377811.3380342.
34. Hu Jr Z. A Software Package for Generating Code Coverage Reports with Gcov, 2021.
35. Beyer D., Lemberger T. TestCov: Robust test-suite execution and coverage measurement // 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2019, pp. 1074-1077, DOI: 10.1109/ASE.2019.00105.

METHOD FOR SEMANTICALLY CORRECT CODE GENERATION FOR FUZZING TESTING JAVASCRIPT ENGINES

Kozachok A.V.[7], Spirin A.A.[8], Erokhina N.S.[9]

Purpose of the work *is to develop of a method for input data generation for fuzzing testing of JavaScript engines and its evaluation.*

Research method *studying the patterns of data generation and the percentage of code coverage in order to increase it. The proposed method allows you to generate input data to identify more vulnerabilities during subsequent fuzzing testing, by increasing the percentage of code coverage.*

Results of the research: *the JavaScript engines is the most vulnerable block of the web-browser architecture, as a result, there is a need to constantly increase the volume of analysis/testing of its source code. Fuzzing testing of a web-browser engine based on complexly structured input data, such as JavaScript code, is an urgent task. The paper presents the vulnerabilities of modern web-browsers, as well as key problems that arise when testing JavaScript engines. The most significant problems are: the lack of publicly available syntactically and semantically*

7 Alexander V. Kozachok, Dr.Sc., Associate Professor, Academy of the Federal Guard Service of the Russian Federation, Orel, Russia. E-mail: a.kozachok@academ.msk.rsnet.ru, https://orcid.org/0000-0002-6501-2008

8 Andrey A. Spirin, Ph.D., Academy of the Federal Guard Service of the Russian Federation, Orel, Russia. E-mail: spirin_aa@bk.ru, https://orcid.org/ 0000-0002-7231-5728

9 Natalya S. Erokhina, Academy of the Federal Guard Service of the Russian Federation, Orel, Russia. E-mail: ens@secdev.space, https://orcid.org/0000-0002-4878-0865

correct input data for fuzzing testing, the problem of overcoming internal mechanisms for filtering input data, the choice of a rational data mutation algorithm, and the problem of increasing the degree of coverage of the code under test. The authors propose a method for generating input data for fuzzing testing of JavaScript engines, which improves the quality and speed of fuzzing testing.

Scientific and practical significance: *the results lies in the development of a new method for generating input data for fuzzing testing of JavaScript engines of web-browsers, based on the use of neural network language models, which increases the coverage of the source code.*

Keywords: *web-browser, JavaScript engine, code coverage, software defects, software vulnerabilities, fuzzing testing, information security.*

References

1. Bytes A. et al. FieldFuzz: In Situ Blackbox Fuzzing of Proprietary Industrial Automation Runtimes via the Network //Proceedings of International Symposium on Research in Attacks, Intrusions and Defenses (RAID). – 2023 ¬ DOI: 10.1145/3607199.3607226.
2. Lee S. et al. Montage: A neural network language model-guided javascript engine fuzzer //Proceedings of the 29th USENIX Conference on Security Symposium. – 2020. – S. 2613-2630, DOI: 10.48550/arXiv.2001.04107.
3. Groß S. Fuzzil: Coverage guided fuzzing for javascript engines // Department of Informatics, Karlsruhe Institute of Technology, 2018.
4. Kozachok A. V. i dr. Obzor issledovanij po primeneniju metodov mashinnogo obuchenija dlja povyshenija jeffektivnosti fazzing-testirovanija // Vestnik Voronezhskogo gosudarstvennogo universiteta, serija: sistemnyj analiz i informacionnye tehnologii. – 2021. – №. 4. – S. 83–106., DOI: 10.17308/sait.2021.4/3800.
5. C. Han. js-vuln-db, A collection of JavaScript engine CVEs with PoCs, 2019. https://github.com/tunz/js-vuln-db.
6. Huang W. et al. testrnn: Coverage-guided testing on recurrent neural networks //arXiv preprint arXiv:1906.08557. – 2019, DOI: 10.48550/arXiv.1906.08557.
7. Gouveia I. P., Völp M., Esteves-Verissimo P. Behind the last line of de-fense: Surviving SoC faults and intrusions //Computers & Security. – 2022. – T. 123. – S. 102920, DOI: 10.1016/j.cose.2022.102920.
8. Fioraldi A. et al. AFL++ combining incremental steps of fuzzing research //Proceedings of the 14th USENIX Conference on Offensive Technologies. – 2020. – S. 10-10.
9. Chao W. C. et al. Design and Implement Binary Fuzzing Based on Libfuzz-er //2018 IEEE Conference on Dependable and Secure Computing (DSC). – IEEE, 2018. – S. 1-2.
10. Peng Chen and Hao Chen. 2018. Angora: Efficient fuzzing by principled search.
11. Österlund S. et al. Parmesan: Sanitizer-guided greybox fuzzing //Proceedings of the 29th USENIX Conference on Security Symposium. – 2020. – S. 2289-2306.
12. Kozachok A. V., Nikolaev D. A., Erohina N. S. Podhody k ocenke po-verhnosti ataki i fazzingu veb-brauzerov //Voprosy kiberbezopasnosti. – 2022. – №. 3 (49). – S. 32–43, DOI: 10.21681/2311-3456-2022-3-32-43.
13. Lin G. et al. Software vulnerability detection using deep neural networks: a survey //Proceedings of the IEEE. – 2020. – T. 108. – №. 10. – S. 1825-1848, DOI: 10.1109/JPROC.2020.2993293.
14. Hanif H. et al. The rise of software vulnerability: Taxonomy of software vulnerabilities detection and machine learning approaches // Journal of Network and Com-puter Applications. – 2021. – T. 179. – S. 103009, DOI: 10.1016/j.jnca.2021.103009.
15. Chernis B, Verma R. Machine Learning Methods for Software Vulnerabil-ity Detection. In: Proceedings of the Fourth ACM International. Workshop on Security and Privacy Analytics ACM. 2018. p. 31–39, DOI: 10.1145/3180445.3180453.
16. Zhu X. et al. Fuzzing: a survey for roadmap // ACM Computing Surveys (CSUR). – 2022. – T. 54. – №. 11s. – S. 1-36, DOI: 10.1145/3512345.
17. Kaloudi N., Li J. The ai-based cyber threat landscape: A survey //ACM Computing Surveys (CSUR). – 2020. – T. 53. – №. 1. – S. 1-34, DOI: 10.1145/3372823.
18. She D, Pei K, Epstein D, Yang J, Ray B, Jana S. NEUZZ: Efficient Fuzzing with Neural Program Smoothing; IEEE Symposium on Security & Privacy; 2019 – s. 38, DOI: 10.1109/SP.2019.00052.
19. Allamanis M. et al. A survey of machine learning for big code and natural-ness //ACM Computing Surveys (CSUR). – 2018. – T. 51. – №. 4. – S. 1-37, DOI: 10.1145/3212695.
20. Karampatsis R. M. et al. Big code! = big vocabulary: Open-vocabulary models for source code //Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. – 2020. – S. 1073-1085, DOI: 10.1145/3377811.3380342.
21. Hu Jr Z. A Software Package for Generating Code Coverage Reports with Gcov, 2021.
22. Beyer D., Lemberger T. TestCov: Robust test-suite execution and coverage measurement // 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2019, pp. 1074-1077, DOI: 10.1109/ASE.2019.00105.