СПОСОБ ОБНАРУЖЕНИЯ ПРОГРАММНЫХ ДЕФЕКТОВ В JAVASCRIPT-ИНТЕРПРЕТАТОРАХ МЕТОДОМ ФАЗЗИНГ-ТЕСТИРОВАНИЯ

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

DOI: 10.21681/2311-3456-2024-2-74-80

Цель исследования: *увеличение скорости обнаружения путей и общего их количества в коде JavaScript-интерпретаторов при выполнении фаззинг-тестирования.*

Метод исследования: *в данном исследовании совмещено использование методов машинного обучения для повышения эффективности генерации входного корпуса, а также простых методов мутации для ускорения выявления дефектов в тестируемом коде JavaScript-интерпретаторов.*

Результат исследования: *фаззинг-тестирование сложного программного обеспечения, такого как JavaScript-интерпретатор, принимающего на вход сложноструктурированные данные, а именно JavaScript код, является актуальной и трудоемкой задачей. Существующие фаззеры при проведении мутаций разрушают синтаксические конструкции языка JavaScript, а также семантику, закодированную во входном корпусе. В работе приведены актуальные задачи фаззинг-тестирования JavaScript-интерпретаторов. Авторами предложен способ обнаружения программных дефектов JavaScript-интерпретаторов совмещающих предварительную генерацию входного корпуса с помощью методов машинного обучения и последующего мутационного фаззинг-тестирования с обратной связью по покрытию кода, который позволяет повысить качество и скорость выявления программных дефектов.*

Научная и практическая значимость: *состоит в разработке нового способа поиска программных дефектов JavaScript-интерпретаторов, совмещающего методы генерации входного корпуса с помощью методов машинного обучения и последующего мутационного фаззинг-тестирования.*

Ключевые слова: *JavaScript-интерпретатор, уязвимости программного обеспечения, генерация входного корпуса, методы мутации данных, фаззинг-тестирование.*

THE METHOD FOR DETECTING SOFTWARE DEFECTS IN JAVASCRIPT ENGINES USING FUZZING

Kozachok A. V.[4], Erokhina N. S.[5], Nikolaev D. A.[6]

Purpose of the work: *is to increase the speed of detecting paths and their total number in the code of JavaScript engines when performing fuzzing testing.*

Research method: *this study combines the use of machine learning methods to increase the efficiency of generating the input corpus, as well as simple mutation methods to speed up the identification of defects in the tested code of JavaScript engines.*

Results of the research: *fuzzing of complex software, such as a JavaScript engine, which accepts complex structured data as input, namely JavaScript code, is a relevant and time-consuming task. Existing fuzzers, when carrying out mutations, destroy the syntactic structures of the JavaScript language, as well as the semantics encoded in the input corpus. The paper presents current problems of fuzzing of JavaScript engines. The authors proposed the method for detecting software defects in JavaScript engines by combining preliminary generation*

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 Ерохина Наталья Сергеевна, Академия ФСО России, г. Орел, Россия. E-mail: ens@secdev.space, https://orcid.org/0000-0002-4878-0865
3 Николаев Дмитрий Александрович, Академия ФСО России, г. Орел, Россия. E-mail: mriddi@bk.ru, https://orcid.org/0000-0001-9334-6948

4 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
5 Natalya S. Erokhina, Academy of the Federal Guard Service of the Russian Federation, Orel, Russia. E mail:osipova_nc@mail.ru, https://orcid.org/0000-0002-4878-0865
6 Dmitry A. Nikolaev, Academy of the Federal Guard Service of the Russian Federation, Orel, Russia. E mail:mriddi@bk.ru, https://orcid.org/0000-0001-9334-6948

of the input corpus using machine learning methods and subsequent mutation fuzzing with feedback on code coverage, which allows increasing the quality and speed of identifying software defects.

Scientific and practical significance: *it consists in the development of the new method for searching for software defects in JavaScript engines, combining methods of generating an input corpus using machine learning methods and subsequent mutation fuzzing.*

Keywords: *JavaScript engine, software vulnerabilities, input corpus generation, data mutation methods, fuzzing.*

Введение

Одним из актуальных современных направлений в фаззинг-тестировании является тестирование JavaScript-интерпретаторов. В отличие от большинства других сред исполнения JavaScript-интерпретатор должен безопасно обрабатывать ненадежный код. Распространенные уязвимости, такие как переполнение буфера или использование памяти после освобождения, редко встречаются в интерпретаторах, их заменили сложносоставные и специфичные для предметной области уязвимости [1]. Такие уязвимости возможно выявить с помощью фаззинг-тестирования, однако, тестирование JavaScript-интерпретаторов имеет множество особенностей.

На сегодняшний день существует множество вариаций методов фаззинг-тестирования, каждый из них имеет свои достоинства и недостатки и лучше справляются с той или иной конкретной задачей. Некоторые методы могут быть быстрыми (случайный фаззинг и символьное исполнение), но не обнаруживать сложные ошибки, в то время как другие – гибридный и мутационный фаззинг – могут быть более эффективными в обнаружении разных типов ошибок. Наиболее успешный метод фаззинга – фаззинг с обратной связью или управляемый фаззинг, являющийся расширением базового алгоритма фаззинга, в котором используется некоторая информация об отклике тестируемой программы на сгенерированный входной файл [2]. Для фаззинга с обратной связью требуется метрика, позволяющая определить, когда входной файл вызвал «интересное» поведение и, таким образом, должен получить положительную обратную связь.

AFLPlusPlus (AFL++)[7] – это один из самых эффективных современных фаззеров с обратной связью, который быстро развивается, постоянно дорабатывается и имеет подробную документацию [3]. Он является развитием оригинального фаззера AFL (англ. American Fuzzy Lop), который в 2013 году дал толчок к массовому использованию фаззинга с обратной связью. Его базовая идея заключается в сборе покрытия ветвей при каждом исполнении, а цель – максимизация покрытия. Преимущество использования AFL++ заключается в его способности находить сложные и редко встречаемые ошибки, которые могут остаться незамеченными при обычных методах тестирования. Это достигается благодаря уникальной технике мутации, которая позволяет создавать вариации входных данных и эффективно исследовать различные пути выполнения программы. Более того, AFL++ предоставляет информацию о покрытии кода, что позволяет определить, какие части программы были протестированы, а какие требуют дальнейшего исследования.

Современные фаззеры с обратной связью эффективно тестируют программное обеспечение, которое обрабатывает компактные и неструктурированные входные данные (например, изображения). Однако, когда они используются для программ, обрабатывающих сложноструктурированные входные данные (например, программный код на языке JavaScript), которые следуют определенной грамматике, возникает множество проблем. Такие программы часто обрабатывают входные данные поэтапно, т. е. синтаксический анализ, семантическая проверка и затем уже выполнение [4]. Универсальные стратегии мутации данных, встроенные в фаззер AFL++, производится в их битовом представлении, что разрушают синтаксис и семантику JavaScript кода, поэтому большая часть предложенных мутированных входных данных, с высокой вероятностью будет мешать обнаруживать новые пути в коде.

Основные особенности при фаззинг-тестировании интерпретаторов:

1. Постоянно нарастающий и усложняющийся код современных интерпретаторов.
2. Отсутствие общедоступных синтаксически и семантически корректных входных данных для проведения тестирования.
3. Проблема синтаксической корректности и преодоления внутренней проверки входных данных тестируемой программой.
4. Отсутствие эффективной методологии мутации сложноструктурированных входных данных, таких как JavaScript-код.
5. Необходимость достижения максимального покрытия тестируемого кода интерпретаторов, часто составляющего более 500 тысяч строк.

7 https://github.com/AFLplusplus/AFLplusplus.

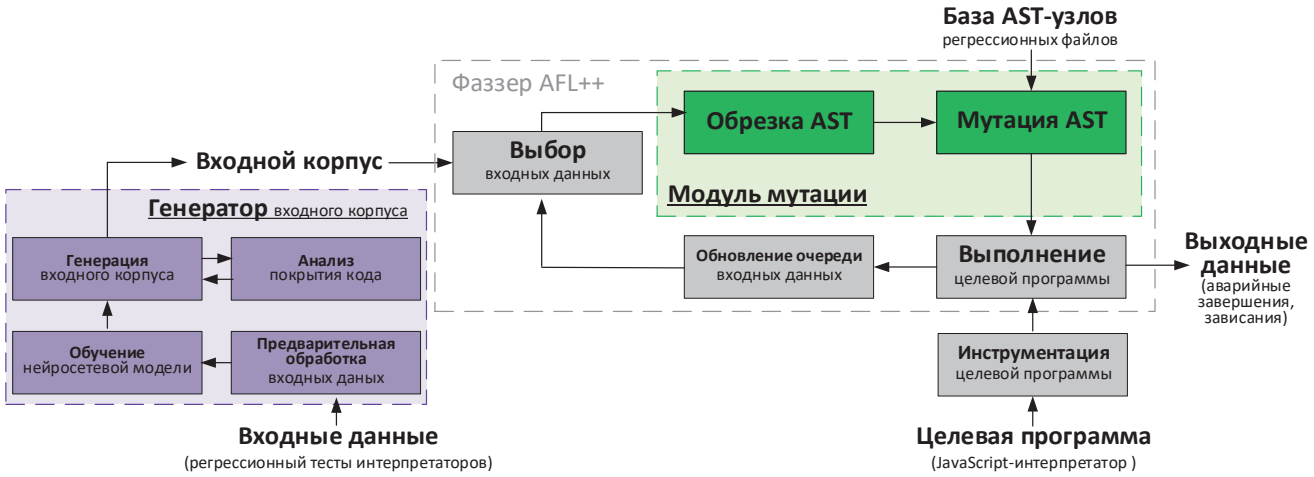

Рис. 1. Архитектура разработанного способа

С целью преодоления вышеизложенных особенностей были разработаны два метода:

1) метод генерации входных данных для фаззинг-тестирования JavaScript-интерпретаторов веб-браузеров, отличающийся использованием нейросетевых языковых моделей, а также управляемый информацией о покрытии исходного кода;
2) метод мутации сложноструктурированных входных данных при фаззинг-тестировании JavaScript-интерпретаторов.

Способ обнаружения программных дефектов JavaScript-интерпретаторов методом фаззинг-тестирования

Прототип подсистемы фаззинг-тестирования на основе внедрения методов предварительной генерации входного корпуса и мутации сложноструктурированных входных данных при фаззинг-тестировании JavaScript-интерпретаторов был реализован на языках программирования Python и JavaScript.

Процесс фаззинг-тестирования JavaScript-интерпретаторов предложено разделить на два больших этапа:

1) генерацию минимального качественного входного корпуса данных из набора регрессионных тестов;
2) быстрый мутационный фаззинг.

На рис. 1 представлена архитектура подсистемы фаззинг-тестирования, разработанная на базе фаззера AFL++. Данная архитектура реализует предлагаемый способ обнаружения программных дефектов и способствует увеличению скорости обнаружения новых путей в коде, а также их общего количества, тем самым, ускоряя процесс фаззинг-тестирования и повышая его эффективность.

Этап генерации входного корпуса

Ввиду того, что ручной сбор входного корпуса является трудоемким и длительным процессом, более эффективно полностью автоматизировать данный процесс.

Методы машинного обучения (МО) глубоко проникли во многие сферы деятельности, включая методы обнаружения уязвимостей [5-6], и фаззинг [7-9]. Также нейронные сети уже активно используются исследователями JavaScript-интерпретаторов [10–13]. В результате обработки большого массива данных нейронная сеть может эффективно выявлять закономерности и обучаться генерировать новый массив, основываясь на выявленной семантике входного корпуса [14]. Данный факт мотивирует применять методы МО для генерации сложноструктурированных данных [15]. Выявляя закономерности в JavaScript-коде, нейронная сеть может генерировать входные данные для фаззинга JavaScript-интерпретаторов.

Для повышения эффективности процесса генерации входного корпуса JavaScript-файлов предложено использовать нейросетевую модель, а для обратной связи – покрытие тестируемого кода.

В работе [10] предложена идея обучения нейросетевой модели последовательностями AST-фрагментов, которые можно использовать в качестве лексикона для работы с нейросетевой моделью. Такой вариант представления позволяет фиксировать глобальные отношения композиции между фрагментами кода для выбора следующего фрагмента и генерации данных.

Более подробно метод генерации входных данных для фаззинг-тестирования JavaScript-интерпретаторов (рис. 2) представлен в работе [16], получено свидетельство о регистрации программы для ЭВМ [17].

В результате работы генератора формируется минимизированный входной корпус данных, который может использоваться далее для повышения эффективности фаззинг-тестирования различных JavaScript-интерпретаторов и, за счет дальнейших мутаций сгенерированного корпуса, выполнять тестирование на наличие дефектов и уязвимостей.

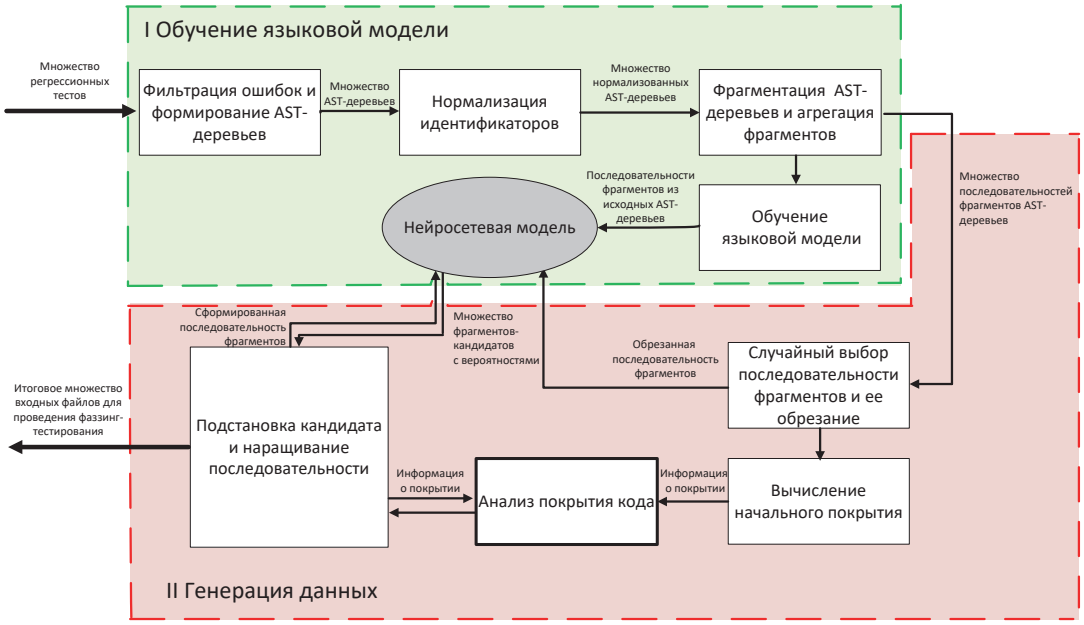

Рис. 2. Архитектура генератора входных данных

Этап мутационного тестирования

Далее для эффективной мутации сложноструктурированных данных необходима их предварительная минимизация, так как это позволяет сократить число потенциальных синтаксических ошибок и повысить эффективность дальнейшего фаззинг-тестирования [18-19]. С данной целью в методе мутации предложена стратегия AST-обрезки. Стратегия AST-обрезки заключается в цикличном удалении каждого узла в AST, и оценка изменения покрытия кода тестируемой программы. Ключевым фактором при обрезке является сохранение синтаксиса исходного фрагмента кода.

После того, как входной файл был минимизирован, к нему применяется одна из следующих простых мутаций: случайная мутация узлов, случайная мутация выражений, случайная мутация литералов, мутация объединения, а также AFL++ мутаций. Данная стратегия изменяет AST представление JavaScript-кода, с высокой вероятностью сохраняя его структуру.

Более подробно метод мутации сложноструктурированных данных при фаззинг-тестировании JavaScript-интерпретаторов представлен в работе [20].

Данный подход позволяет ускорить процесс обнаружения уязвимостей, за счет удаления избыточности из входного корпуса, а также повышения скорости обнаружения путей.

Этап обработки результатов

Одной из наиболее трудоемких частей процесса фаззинга-тестирования является работа, необходимая для определения того, представляет ли конкретное аварийное завершение угрозу безопасности. Конечной целью жизненного цикла фаззинг-тестирования является выявление эксплуатируемых уязвимостей безопасности, которые можно использовать при создании эксплойта для целевого ПО. Незначительное меньшинство всех сбоев будет иметь очевидные последствия, однако, большинство аварийных завершений более неоднозначны. В данном вопросе факт эксплуатируемости уязвимости является наиболее критичным. Задача фаззинга – нахождение аварийных завершений программы, некоторые из которых могут сигнализировать о наличии в ней уязвимости [2].

При проведении фаззинг-тестирования сложного ПО, такого как интерпретаторы, выявляются десятки, а иногда и сотни аварийных завершений. Ручной анализ каждого аварийного завершения требует от специалиста, производящего тестирование, высокой квалификации и больших трудозатрат.

Данная архитектура, основанная на фаззере AFL++ имеет еще одно значительное преимущество: помимо подробной документации по использованию AFL++ имеет множество инструментов, позволяющих эффективно обрабатывать выявленные сбои.

«AFLTriage» – это инструмент для сортировки аварийных входных файлов с помощью отладчика. AFLTriage не классифицирует аварийные завершения по потенциальной возможности использования. Точная классификация уязвимостей зависит от цели и сценария выполнения, и производится специализированными инструментами и экспертами-аналитиками.

«Режим исследования сбоев AFL»[8] (англ. AFL crash exploration mode) – это режим, встроенный в AFL++, который берет найденный сбой и повторяет его, чтобы найти другие варианты того же сбоя. В этом режиме AFL++ определит изменения, которые можно применить к входным данным, генерирующим сбой, чтобы достичь других путей кода, не приводя к другому сбою.

Целью этого режима является создание небольшого набора файлов, который можно быстро изучить, чтобы определить, эксплуатируема ли выявленная уязвимость.

Экспериментальная оценка результатов исследования

С целью сравнения эффективности разработанного способа в данном исследовании используется скорость нахождения новых путей в тестируемом коде интерпретатора, определяемая как количество путей в отношении ко времени выполнения, а также количеству запусков. Данная метрика демонстрирует, как часто открываются новые состояния тестируемой программы. Уязвимости содержатся не только в строках кода тестируемой программы, но и ее состояниях, некоторые из которых могут привести к этой уязвимости. Из чего следует, что количество обнаруженных путей, также важно для оценки эффективности разработанной стратегии фаззинг-тестирования.

Для проведения экспериментов был выбран JavaScript-интерпретатор V8[9], а также было собрано 49475 файлов регрессионных тестов различных интерпретаторов. Исходное покрытие кода интерпретатора V8 файлами регрессионных тестов составило: 42%. В результате работы метода генерации был сгенерирован 81 файл, обеспечивающий покрытие кода: 48%. Что повысило исходное покрытие на 6%, сократив при этом корпус на 99.83%.

Для сравнения эффективности разработанного прототипа был выбран оригинальный фаззер AFL++, а также еще один встроенный модуль мутации для AFL++ Grammar-Mutator[10], основанный на грамматике языка JavaScript. Оба фаззера предлагают различный подход к мутациям входных данных, поэтому при сравнении с ними можно оценить, как предложенные улучшения стратегии повлияли на эффективность подхода.

Фаззинг-тестирование проводилось в течение 24 часов, что достаточно для демонстрации тенденции изменения скорости обнаружения новых путей.

На рис. 3 представлены графики зависимости количества обнаруженных новых путей от времени выполнения в коде интерпретатора V8. В табл. 1 представлены количественные результаты по обнаруженным новым путям в коде через 3 отрезка времени: 1 час, 12 часов и 22 часа. Исходя из таблицы, разработанный способ через 22 часа эксперимента превосходит два других фаззера на 38.85% (AFL++) и 3.87% (Grammar-Mutator).

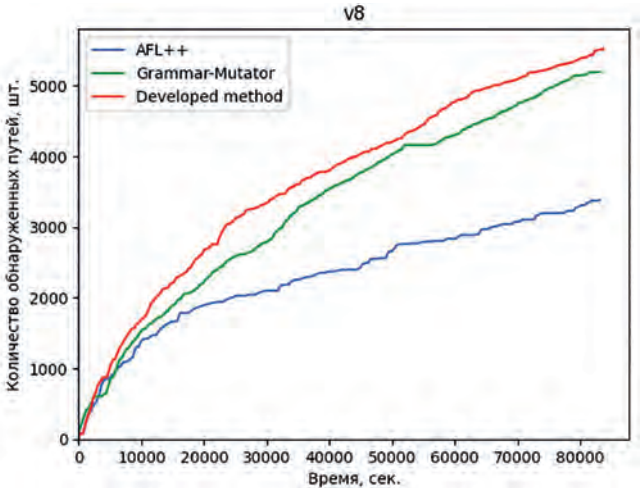

Рис. 3. Графики зависимости количества обнаруженных новых путей от времени выполнения в коде интерпретатора V8

Таблица 1

Сравнение результатов эксперимента по времени

Фаззер	Кол-во обнаруж. путей через 1 ч (3600 сек)	Кол-во обнаруж. путей через 12 ч (43200 сек)	Кол-во обнаруж. путей через 22 ч (79200 сек)
AFL++	751	2399	3272
Grammar-Mutator	605	3675	5145
Разработанный способ	839	3973	5352

Однако, сравнение лишь по времени не совсем наглядно демонстрирует результат. Одной из важных характеристик фаззера является скорость запусков. Средняя скорость запусков для исследуемых фаззеров составляет: 3.60 зап/сек (AFL++), 5.31 зап/сек (Grammar-Mutator) и 3.63 зап/сек (My-Mutator). Данный факт мотивирует нас дополнительно оценить зависимость количества обнаруженных новых путей от числа запусков (рис. 4).

В табл. 2 представлены количественные результаты по обнаруженным новым путям в коде через 10, 100 и 230 тыс. запусков. Исходя из таблицы, разработанный способ через 230 тыс. запусков превосходит

8 https://afl-1.readthedocs.io/en/latest/fuzzing.html#crash-triage
9 JavaScript-интерпретатор v8. https://github.com/v8/v8
10 Shengtuo Hu (h1994st). 2020. Grammar Mutator – AFL++. https://github.com/AFLplusplus/Grammar-Mutator

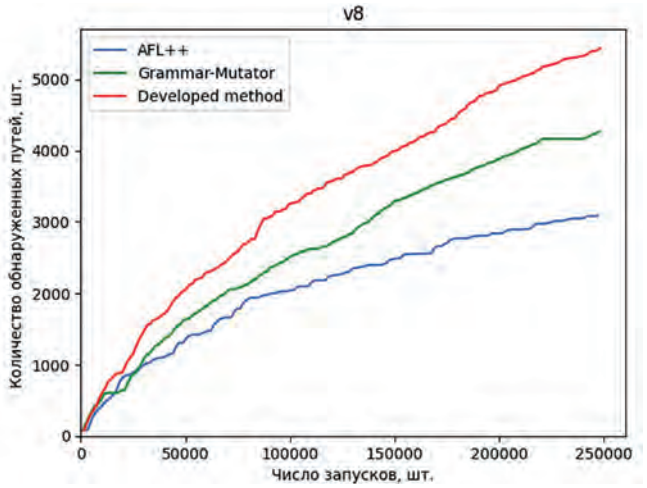

Рис. 4. Графики зависимости количества обнаруженных новых путей от числа запусков в коде интерпретатора V8

Таблица 2

Сравнение результатов эксперимента по запускам

Фаззер	Кол-во обнаруж. путей через 10 тыс. запусков	Кол-во обнаруж. путей через 100 тыс. запусков	Кол-во обнаруж. путей через 230 тыс. запусков
AFL++	412	2043	3010
Grammar-Mutator	514	2513	4163
Разработанный способ	599	3252	5263

два других фаззера на 42.79% (AFL++) и 20.89% (Grammar-Mutator).

Следовательно, экспериментально доказано увеличение скорости обнаружения путей и общего их количества при применении метода генерации входного корпуса, а также встроенного модуля мутации в способе обнаружения уязвимостей.

В ходе экспериментов аварийных завершений в последней версии JavaScript-интерпретатора V8 не выявлено.

Заключение

Фаззинг-тестирование сложного программного обеспечения, такого как JavaScript-интерпретатор, обрабатывающего сложноструктурированные входные данные, а именно программный код на языке JavaScript, является актуальной и трудоемкой задачей. В работе приведены актуальные задачи в фаззинг-тестировании JavaScript-интерпретаторов, а также описаны эффективные методы их решения. Авторами предложен способ обнаружения программных дефектов в JavaScript-интерпретаторах, который позволяет повысить качество и скорость обнаружения новых путей в коде. Разработанный способ по скорости обнаружения путей относительно времени выполнения тестирования превосходит фазеры AFL++ и Grammar-Mutator на 38.85% и 3.87% соответственно; по скорости обнаружения путей относительно числа запусков на 42.79% и 20.89% соответственно.

Литература

1. *Groß S. et al. FUZZILLI: Fuzzing for JavaScript JIT Compiler Vulnerabilities // Network and Distributed Systems Security (NDSS) Symposium. – 2023.*
2. *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.*
3. *Fioraldi A. et al. AFL++ combining incremental steps of fuzzing research //Proceedings of the 14th USENIX Conference on Offensive Technologies. – 2020. – с. 10.*
4. *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.*
5. *Xing C. et al. A new scheme of vulnerability analysis in smart contract with machine learning //Wireless Networks. – 2020. – С. 1–10.*
6. *Wang Y. et al. A systematic review of fuzzing based on machine learning techniques //PloS one. – 2020. – Т. 15. – №. 8. – С. e0237749.*
7. *Xue Y. et al. xfuzz: Machine learning guided cross-contract fuzzing //IEEE Transactions on Dependable and Secure Computing. – 2022.*
8. *Kashyap G. S. et al. Using Machine Learning to Quantify the Multimedia Risk Due to Fuzzing //Multimedia Tools and Applications. – 2022. – Т. 81. – №. 25. – С. 36685–36698.*
9. *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.*
10. *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.*
11. *Liu X, Li X, Prajapati R, Wu D. DeepFuzz: Automatic Generation of Syntax Valid C Programs for Fuzz Testing. In: Proceedings of the... AAAI Conference on Artificial Intelligence, 2019, DOI: 10.1609/aaai.v33i01.33011044.*
12. *Ye G. et al. Automated conformance testing for JavaScript engines via deep compiler fuzzing //Proceedings of the 42nd ACM SIGPLAN international conference on programming language design and implementation. – 2021. – С. 435–450, DOI: 10.1145/3453483.3454054*

13. *Ye G. et al. A Generative and Mutational Approach for Synthesizing Bug-Exposing Test Cases to Guide Compiler Fuzzing //Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. – 2023. – С. 1127–1139, DOI: 10.1145/3611643.3616332*
14. *Осипова Н. С. Применение методов машинного обучения при проведении фаззинг-тестирования // Безопасные информационные технологии. Сборник трудов XI. – 2021. – Т. 6. – с. 25.*
15. *Козачок, А. В., Козачок, В. И., Осипова, Н. С., Пономарев, Д. В. Обзор исследований по применению методов машинного обучения для повышения эффективности фаззинг-тестирования // Вестник ВГУ. Серия: Системный анализ и информационные технологии, 2016, №4, с. 83–106. DOI: 10.17308/sait.2021.4/3800*
16. *Козачок А. В., Спирин А. А., Ерохина Н. С. Метод генерации семантически корректного кода для фаззинг-тестирования интерпретаторов JavaScript // Вопросы кибербезопасности. – 2023. – №. 5 (57). – С. 80–88. DOI: 10.21681/2311-3456-2023-5-80-88*
17. *Свидетельство о государственной регистрации программы для ЭВМ № 2023664761 Российская Федерация. Программный модуль генерации семантически корректного Javascript-кода для фаззинг-тестирования Javascript интерпретаторов веб-браузеров: № 2023663536: заявл. 29.06.2023: опубл. 07.07.2023 / Н. С. Ерохина, А. В. Козачок; заявитель Федеральное государственное казенное военное образовательное учреждение высшего образования «Академия Федеральной службы охраны Российской Федерации». – EDN XGDSQY.*
18. *Aschermann C. et al. NAUTILUS: Fishing for Deep Bugs with Grammars // NDSS. – 2019. DOI: 10.14722/ndss.2019.23xxx*
19. *Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superion: grammar-aware greybox fuzzing. In Proceedings of the 41st International Conference on Software Engineering (ICSE). С. 724–735. https: //doi.org/10.1109/ICSE.2019.00081.*
20. *Ерохина Н. С. Метод мутации сложноструктурированных входных данных при фаззинг-тестировании JavaScript интерпретаторов. Труды Института системного программирования РАН, том 35, вып. 5, 2023, С. 55–66. DOI: 10.15514/ISPRAS-2023-35(5)-4*