DOI: 10.15514/ISPRAS-2023-35(5)-4

Метод мутации сложноструктурированных входных данных при фаззинг-тестировании JavaScript интерпретаторов

Н.С. Ерохина, ORCID: 0000-0002-4878-0865 <*ens@secdev.space*>

Академия ФСО России,
302034, Россия, г. Орёл, ул. Приборостроительная, д. 35.

Аннотация. Фаззинг-тестирование JavaScript интерпретаторов является одним из наиболее сложных направлений в тестировании веб-браузера, ввиду сложности генерации его входных данных. Интерпретаторы обрабатывают JavaScript код на веб-странице и требуют постоянной поддержки новых стандартов языка и усложнения своей архитектуры. Наиболее распространенные сегодня фаззеры не способны эффективно мутировать сложноструктурированные входные данные при фаззинг-тестировании. Генерация JavaScript кода с нуля не позволяет инкапсулировать необходимую семантику, а текущие мутаторы быстро разрушают синтаксис и семантику языка входных данных. В данной статье представлена новая стратегия мутации, сохраняющая синтаксис и семантику входных данных за счет модификации AST-деревьев фрагментов JavaScript кода. Данный метод позволяет эффективно генерировать разнообразные и корректные входные данные, которые могут привести к выявлению ошибок и уязвимостей в интерпретаторах JavaScript. Данный метод может быть использован для повышения безопасности веб-браузеров и обеспечения надежности интерпретации JavaScript кода.

Ключевые слова: мутации сложноструктурированных данных; интерпретатор JavaScript; уязвимости программного обеспечения; фаззинг-тестирование.

Для цитирования: Ерохина Н. С. Метод мутации сложноструктурированных входных данных при фаззинг-тестировании JavaScript интерпретаторов. Труды ИСП РАН, том 35, вып. 5, 2023 г., стр. 55–66. DOI: 10.15514/ISPRAS–2023–35(5)–4.

Method for Mutation of Complexly Structured Input Data during Fuzzing of Javascript Engines

N.S. Erokhina, ORCID: 0000-0002-4878-0865 <*ens@secdev.space*>

Academy of the Federal Guard Service of Russian Federation,
35, Priborostroitelnaya st., Orel, 302034, Russia.

Abstract. Fuzzing of JavaScript engines is one of the most difficult areas in web-browser testing due to the complexity of input data generating. JavaScript engines process JavaScript code on a web page and require constant support for new language standards and increasing complexity in their architecture. The most common fuzzers today are not able to effectively mutate complexly structured input data during fuzzing. Generating JavaScript code from scratch does not allow encapsulating the necessary semantics, and current mutators quickly destroy the syntax and semantics of the input data language. This article presents a new mutation strategy that preserves the syntax and semantics of the input data by modifying the AST of JavaScript code fragments. This method allows you to efficiently generate diverse and correct input data, which can lead to the identification of errors and vulnerabilities in JavaScript engines. This method can be used to improve the security of web browsers and ensure reliable interpretation of JavaScript code.

Keywords: mutations of complex structured data; JavaScript engine; software vulnerabilities; fuzzing.

For citation: Erokhina N.S. Method for mutation of complexly structured input data during fuzzing of JavaScript engines. *Trudy ISP RAN/Proc. ISP RAS*, vol. 35, issue 5, 2023. pp. 55-66 (in Russian). DOI: 10.15514/ISPRAS-2023-35(5)-4.

1. Введение

Высокая сложность современного программного обеспечения, обусловленная большим объемом исходного кода, иногда достигающим нескольких миллионов строк, а также наличие уязвимостей и ошибок в нем, является фундаментальной проблемой, вызванной текущим состоянием развития информационных технологий. Веб-браузер является ярким примером, демонстрирующим данный факт. На рис. 1 представлена поверхность атаки веб-браузера.

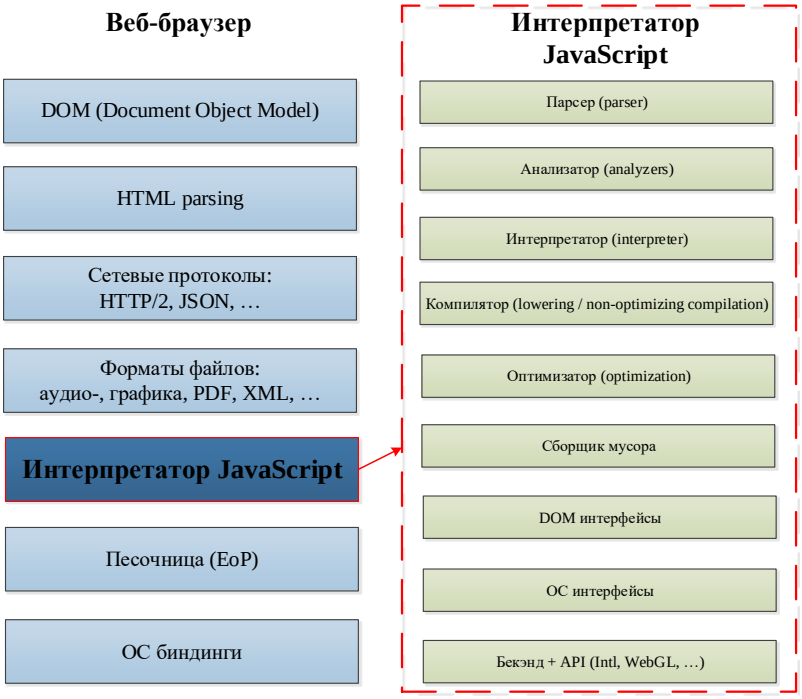

Рис.1. Поверхность атаки веб-браузера.
Fig. 1. Web-browser attack surface.

Среди всей архитектуры веб-браузера, поиск уязвимостей в JavaScript интерпретаторах является наиболее сложной, потому востребованной задачей в тестировании веб-браузеров. Появление в 90-х годах 20 века динамических веб-приложений стало отправной точной популярности языка JavaScript. По данным рейтинга W3Techs[1] на октябрь 2023 года 98,8% всех веб-сайтов используют язык JavaScript. Любой современный веб-браузер поддерживает язык динамической разработки JavaScript без использования дополнительного программного обеспечения (ПО) с помощью встроенного JavaScript интерпретатора. Стремительный рост числа новых интернет-технологий требует постоянного усложнения и наращивания кодовой базы интерпретатора веб-браузера. Данный факт часто приводит к ошибкам, что негативно сказывается на безопасности веб-браузера в целом и активизирует деятельность авторов вредоносных программ. Согласно Национальной базе данных уязвимостей (NVD2), 43% всех

[1] https://w3techs.com/technologies/details/cp-javascript.
[2] https://nvd.nist.gov/.

уязвимостей, обнаруженных в веб-браузерах Microsoft Edge и Google Chrome, были уязвимостями интерпретатора JavaScript.

2. Актуальные проблемы фаззинг-тестирования интерпретаторов JavaScript

Современные руководящие документы по безопасной разработке предлагают широкий спектр методов экспертного, статического и динамического анализа[3]. Наиболее распространенные из них: межпроцедурный контекстно-чувствительный анализ, анализ бинарного кода, а также байт-кода, формальная верификация, отслеживание помеченных данных и сканирование уязвимостей, а также направленный анализ участков кода и моделирование атак. Каждый из них имеет свои достоинства и недостатки, однако фаззинг-тестирование применительно к задаче выявления уязвимостей сложного программного обеспечения, такого как JavaScript-интерпретаторы имеет ряд преимуществ:

- высокая степень автоматизации, что позволяет проводить тестирование большого объема кода за обозримое время;
- может быть использован для тестирования любой системы, которая имеет входные данные;
- возможность обнаружения новых и неизвестных ранее уязвимостей;
- автоматизированное тестирование может выявить те ошибки, которые не удалось найти методом ручного тестирования, за счёт большего покрытия кода;
- автоматизированное тестирование позволяет человеку участвовать лишь на этапе работы с результатами;
- позволяет собрать общее представление о защищенности тестируемого кода.

Фаззинг становится все более популярным методом проверки функциональности программного обеспечения и поиска уязвимостей безопасности. Он успешно применяется для тестирования различных приложений, начиная от механизмов рендеринга и процессоров изображений и заканчивая компиляторами и интерпретаторами.

Программная реализация, которой проводится фаззинг-тестирование, называется фаззером. В различных источниках классификация фаззеров может отличаться, чаще всего она производится по следующим параметрам: по информации о целевой программе, которая будет подвергнута фаззингу; по наличию обратной связи с тестируемой программой; по операциям, которые будут совершаться над входными данными, а по методам генерации данных [1]. В области тестирования JavaScript интерпретаторов проведено множество исследований, среди них: генеративные фаззеры создают новые тестовые случаи с нуля на основе заранее определенной грамматики, такие как jsfunfuzz[4] и Fuzzilli [2], или путем их создания из синтезируемых блоков кода, разобранных на большой корпус [3]. Мутационные фаззеры генерируют новые тестовые примеры на начальных входных данных для тестирования. Например, LangFuzz [4] разбивает программы в большом корпусе на небольшие фрагменты кода, рекомбинирует их с исходным вводом и генерирует новые тестовые примеры; Skyfire [5], Nautilus [6] и Superion [7] мутируют каждую программу индивидуально с помощью сегментов, полученных от других программ в корпусе, или с их правилом мутации.

По количеству информации о целевой программе фаззеры подразделяются на: фаззеры «белого ящика», фаззеры «серого ящика» (например, [2[2], 6-7]) а также фаззеры «черного

[3] Методика выявления уязвимостей и недекларированных возможностей в программном обеспечении (утверждена приказом ФСТЭК России в декабре 2020 года).

[4] https://github.com/MozillaSecurity/funfuzz.

ящика» (например, [3-5]). Среди распространенных фаззеров JavaScript интерпретаторов нет представителей «белого ящика» ввиду сложности и объемности кода интерпретаторов.

В последние годы фаззинг «серого ящика» на основе покрытия зарекомендовал себя как один из наиболее эффективных методов поиска ошибок безопасности на практике. Фаззинг-тестирование методом «серого ящика» предполагает внедрение инструментов в тестируемую программу для получения обратной связи для управления ходом тестирования. Главное преимущество метода в том, что фаззер получает информацию не только о выводе и аварийном завершении программы, но и о ходе исполнения программы. В общем случае информация может быть любой, но обычно используют такую метрику, как покрытие кода программы.

AFL – это канонический пример фаззера, который в 2013 году дал толчок в массовому использованию фаззинга с обратной связью, и обнаруживший тысячи громких уязвимостей. Его базовая идея заключается в сборе покрытия ветвей при каждом исполнении, а цель – максимизация покрытия. Сейчас первоначальный AFL уже не используется, но от него образовалось множество проектов. Самый популярный и быстроразвивающийся из них – это AFLPlusPlus (AFL++) [8]. Он вбирает в себя новые техники и постоянно расширяет возможности исследователей. AFL++ известен своей высокой производительностью и эффективностью в обнаружении уязвимостей и ошибок в программном обеспечении. AFL++ обрабатывает целевую программу в два этапа: инструментирование целевой программы и фаззинг инструментированной программы. На этапе инструментирования в точки ветвления тестируемой программы внедряются маяки для считывания покрытия ветвей вместе с количеством попаданий в них.

Однако, применение фаззинга с обратной связью к интерпретаторам JavaScript нетривиально. AFL++ эффективен при работе с бинарными данными, а не с текстовыми и тем более не с жестко структурированными входными данными, такими как JavaScript код. AFL приходится тратить много времени на борьбу с корректностью синтаксиса, находя при этом только ошибки синтаксического анализа. Универсальные алгоритмы обрезки и мутации данных, встроенные в фаззер AFL++, работают на битовом уровне представления JavaScript кода, что разрушают тонкую семантику или условия, закодированные в коде. Из чего следует, что большая часть предложенных некорректных мутированных входных данных с высокой вероятностью будет мешать обнаруживать новые пути в коде. Тестовые примеры, сгенерированные текущими методами, могут ссылаться на переменные, которые не определены в текущем контексте выполнения. Вследствие чего, программа может завершиться из-за синтаксической или семантической ошибки уже во время выполнения. На рис. 2 представлена разница операций замены и вставки значений встроенным в AFL++ способом и на основе AST.

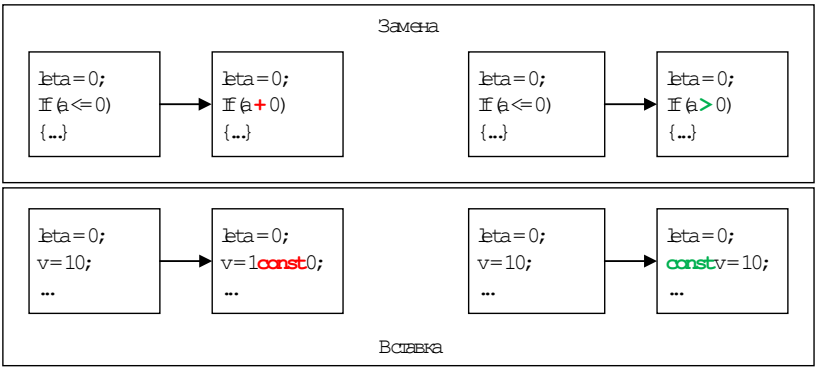

Рис. 2. Пример мутации AFL++ (а) и мутации на основе AST-дерева (б).
Fig. 2. Example of AFL++ mutation (a) and mutations based on AST (b).

Не менее важно также ответить на вопросы: как генерировать тестовые примеры, охватывающие больше программных путей, и как определить мутацию, которая направит тестирование на новые, еще не исследованные пути в программе. Повышение покрытия кода означает повышение охвата состояний выполнения программы и повышение качества тестирования. Известно, что увеличение покрытия кода на 1% увеличивает процент обнаруженных ошибок на 0,92%[5]. Однако, большинство тестовых примеров охватывают только некоторое, ограниченное число путей, в то время как большая часть кода не достигается. Распределение ошибок в программах чаще не сбалансированно, то есть, например, 80% ошибок могут находиться в 20% программного кода, что приводит к тому, что многие фаззеры тратят много времени на тестирование неуязвимых путей.

Большинству фаззеров, основанных на покрытии, включая AFL++, требуется набор входных данных, которые они используют в качестве основы для запуска процесса фаззинга. Корпус хорошего качества имеет решающее значение для производительности и эффективности фаззера. Получить такой высококачественный корпус непросто: если язык, принимаемый целевым приложением, широко используется, одним из подходов является сканирование общедоступных примеров из Интернета или из общедоступных репозиториев кода. Однако эти примеры, скорее всего, будут смещаться в сторону очень часто используемых частей грамматики, в которых используются хорошо протестированные части целевого ПО. Естественно, исследователи безопасности часто хотят протестировать функции, которые редко используются или были представлены совсем недавно и, которые с большей вероятностью приведут к ошибкам в целевом ПО. Собрать корпус для тестирования этих функций явно сложнее, а написание примеров вручную обходится очень дорого. Вследствие вышеизложенных проблем, результаты фаззинг-тестирования интерпретаторов сильно уступают результатам, достигнутым в других областях [9].

Дополнительной сложностью является выбор данных, инкапсулирующих в себе необходимую семантику, позволяющую более эффективно выявлять уязвимости в тестируемом коде. В случае с JavaScript интерпретаторами подходящими данными являются файлы регрессионных тестов браузеров, написанные специалистами вручную и нацеленные на проверку функциональности веб-браузеров. В исследовании [10] была выявлена корреляция между файлами регрессионных тестов и фрагментами кода, вызывающими уязвимости, и наглядно продемонстрирована эффективность их использования.

Эффективность фаззинг-тестирования JavaScript интерпретаторов может быть повышена за счет использования новых алгоритмов обрезки и мутации, которые нацелены на сохранение синтаксиса JavaScript языка и семантики регрессионных тестов. Проведенный анализ литературы позволяет утверждать, что, на сегодняшний день, разработка алгоритмов эффективной обрезки и мутации сложноструктурированного кода JavaScript-интерпретаторов является достаточно сложным и востребованным процессом, с точки зрения информационной безопасности [6-11].

3. Метод мутаций сложноструктурированных данных

Многими исследованиями наглядно продемонстрирована эффективность представления и дальнейшей обработки сложноструктурированных данных в виде абстрактного синтаксического дерева [7-10]. Абстрактное синтаксическое дерево или AST (от англ. Abstract syntax tree) – это конечное, помеченное, ориентированное дерево, в котором внутренние вершины сопоставлены с операторами языка программирования, а листья – с соответствующими операндами [12]. Каждый входной JavaScript файл преобразуется в AST-дерево в соответствии со спецификацией языка ECMAScript.

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

В отличие от токенов, используемых при мутациях на основе словаря, AST фактически моделируют тестовые входные данные как объекты с именованными свойствами, четко соблюдая структуру кода. Таким образом, мутация на уровне AST обеспечивают подходящую степень детализации для фаззера, а также позволяет сохранять синтаксис и семантику JavaScript языка.

Стратегия обрезка реализуется путем итеративного удаления каждого поддерева в AST входных данных и наблюдения за различиями в покрытии. Алгоритм пытается обрезать AST-дерево, так чтобы сохранить исходное покрытие. Обрезанные входные данные имеют преимущество более короткого времени выполнения и меньшего набора потенциальных мутаций во время дальнейшей обработки. Обрезка поддеревьев направлена на то, чтобы сделать их как можно короче, при этом вызывая новые пути в коде. На рис. 3 показан пример разработанной стратегии обрезки входных данных на языке JavaScript, где полная строка (выделена перечеркиванием) обрезается без внесения каких-либо различий в покрытии. С помощью встроенной стратегии обрезки AFL++ практически невозможно отсечь такое полное утверждение.

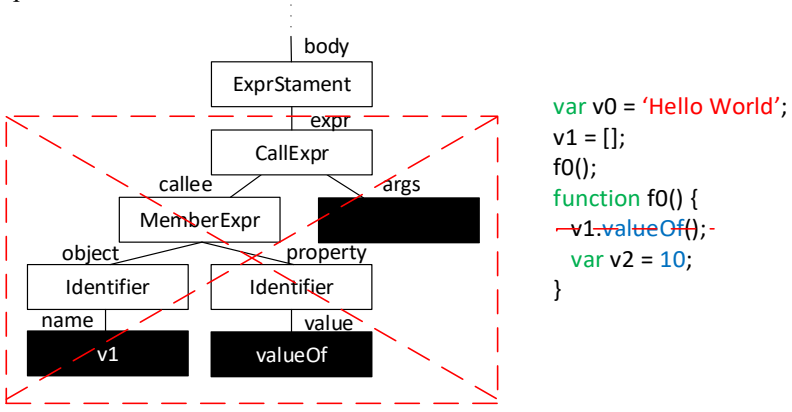

Рис. 3. Пример обрезки файла на основе AST-дерева .
Fig. 3. Example of file trimming based on AST.

Процедура разработанной стратегии представлена на рис. 4 псевдокодом алгоритма. Случайно выбранный входной файл преобразуется в AST-дерево. Далее алгоритм проходит по каждому узлу в дереве и пытается обрезать поддерево, для которого данный узел является корневым. Если обрезка проходит успешно и итоговое покрытие тестируемого кода тестом не ухудшается – обрезанный входной файл добавляется в очередь фаззера. Минимизация входных данных направлена на удаление избыточности в данных, мешающей фаззеру наиболее эффективно производить мутации.

После того, как AST-дерево было обрезано, используется несколько методов мутации для создания новых входных данных для фаззера (рис. 5). Разработанная стратегия мутации AST-дерева JavaScript кода включает в себя последовательное применение алгоритмов: мутаций узла AST (рис. 6), случайных мутаций, мутаций объединения (рис. 7), а также AFL++ мутаций. Данная стратегия изменяет AST входных данных JavaScript, сохраняя с высокой вероятностью структуру кода, влияющую на общие потоки управления, и исходные типы используемых переменных.

Мутации узла направлены на замену 25% узлов в AST-дереве на инверсные, либо случайно выбранные «интересные» значения из представленного словаря. Случайная мутация выбирает случайный узел дерева и заменяет его случайно сгенерированным новым поддеревом, корнем которого является тот же нетерминал. Мутация объединения объединяет входные данные, которые нашли разные пути, помещая поддерево из одного файла в другой. Для этого выбирается случайный внутренний узел, который становится корнем заменяемого

поддерева. и из дерева в очереди берется случайное поддерево, корнем которого является тот же нетерминал. Мутация AFL выполняет мутации, которые также используются AFL, например, перестановку битов с целью проверки синтаксического анализатора интерпретатора.

```
Input: the test input to be trimmed input
Output: the set of trimmed test T
Function Trimming(input)
T = ∅
E = ∅ // Множество ошибок
tree ← ParseToAst(input)
coverage ← RunEngine(tree) // Вычиление начального
  покрытия
seq ← TraverseAst(tree)
count ← CountNodes(tree) // Вычиление числа узлов дерева
step = count
while step > 1 do
    removable ← seq[step] // Выбор узла для обрезки
    trimmed, E ← RemoveNode(tree, removable) // Обрезка
      дерева
    if E == ∅ then
        coverage_new ← RunEngine(trimmed)
        if coverage_new ≥ coverage then
            T = T ∪ {trimmed} // Добавление экземпляра в
              последовательность
        end
    end
    step = step − 1
end
return T
```

Рис. 4. Псевдокод алгоритма обрезки AST-дерева.
Fig. 4. Pseudocode of the AST trimming algorithm.

```
Input: Мутируемые входные данные input
Output: Мутированное AST mutatedTree
Function Mutation(input)
E = ∅ // Множество ошибок
tree ← ParseToAst(input)
while True do
    mutStrategy = RandomChoice(mutateNodes,
                               mutateLiterals,
                               mutateExpressions,
                               mutateSubtrees)

    mutatedTree, E ← mutStrategy(tree)
    if mutatedTree ≠ None ∧ E == ∅ ∧ mutatedTree ≠ tree then
        mutatedCode ← ParseToCode(mutatedTree)
        return mutatedCode
    end
end
```

Рис. 5. Псевдокод общего алгоритма мутаций AST-дерева JavaScript кода.
Fig. 5. Pseudocode of the main AST mutation algorithm for JavaScript code.

```
Input: Мутируемое AST tree,
       Пул фрагментов регрессионных тестов P
Output: Мутированное AST mutatedTree, Множество ошибок E
Function mutateNodes(tree)
E = ∅
fragSeq ← GetFrags(tree)
replacedIdx = getRandomInt(Length(fragSeq))
trimmedTree, fragType = RemoveNode(fragSeq, replacedIdx)
newFrag = RandomChoice(P(fragType))
mutatedTree ← ReplaceNode(trimmedTree, newFrag)
return mutatedTree, E
```

Рис. 6. Псевдокод алгоритма мутации узла AST JavaScript кода.
Fig. 6. Pseudocode of the AST node mutation algorithm for JavaScript code.

```
Input: Мутируемое AST tree, множество AST для вставки T
Output: Мутированное AST mutatedTree, Множество ошибок E
Function mutateSubtrees(tree)
E = ∅
// Случайный выбор дерева для вставки
sourceTree = RandomChoice(T)
count ← GetFrags(tree) // Вычиление числа узлов дерева
// Выбор места для вставки
replacedIdx ← getRandomInt(count)
for node ∈ tree do
    count + +
    if mutationFrag == True then
        break// Вставка реализована, выход из цикла
    else
        if count == replacedIdx then
            // Получение подходящего узла
            newNode, sourceTree = getNode(node.type)
            // Подготовка к вставке узла
            prepareNodeForInsertion(newNode, sourceTree)
            mutationFrag = True
            break
        end
        mutatedTree = tree ∪ {newNode}
        return mutatedTree, E
    end
end
```

Рис. 7. Псевдокод алгоритма мутации объединения AST JavaScript кода.
Fig. 7. Pseudocode of the union mutation algorithm for JavaScript code AST.

4. Оценка эффективности представленного способа

Для проверки эффективности предложенных стратегий было собрано 49475 файлов регрессионных тестов различных интерпретаторов. Для лучшей работы фаззера AFL данные были предварительно проверены на синтаксическую корректность, а также сжаты утилитой afl-cmin для интерпретаторов JavaScriptCore и v8 до 247 и 81 файлов соответственно, на которых в итоге проводилось тестирование.

4.1 Эффективность разработанной стратегии обрезки

В табл. 1 представлено сравнение соотношения тестовых входных данных, которые являются валидными (грамматически допустимыми) после обрезки с использованием встроенной обрезки в AFL и разработанным способом.

В численном отношении разработанная стратегия увеличила коэффициент достоверности грамматики входных данных для v8 и JavaScriptCore с 74,1%, 83,7% до 89.7%, 97.3% что может увеличить вероятность эффективного дальнейшего применения мутации. И тем самым повысить эффективность создания тестовых входных данных, которые могут инициировать новое покрытие.

Таким образом, разработанная стратегия обрезки с учетом грамматики может улучшить коэффициент достоверности грамматики для тестовых входных данных после обрезки, что облегчает дальнейшую мутацию.

Табл. 1. Результаты оценки эффективности предложенной стратегии обрезки
Table 1. Results of evaluating the effectiveness of the proposed trimming strategy

Тестируемый интерпретатор	**Процент валидности**	
	AFL++ (%)	**Обрезка AST** (%)
V8	74.1	89.7
JavaScriptCore	83.7	97.3

4.2 Эффективность разработанной стратегии мутации

Эффективность разработанной стратегии мутации демонстрируется графиками скорости обнаружения новых путей в коде интерпретаторов на рис. 8-9. Что наглядно демонстрирует увеличение скорости при применении встроенного модуля мутации.

Число обнаруженных воспроизводимых уникальных зависаний протестированных интерпретаторов представлено в табл. 2.

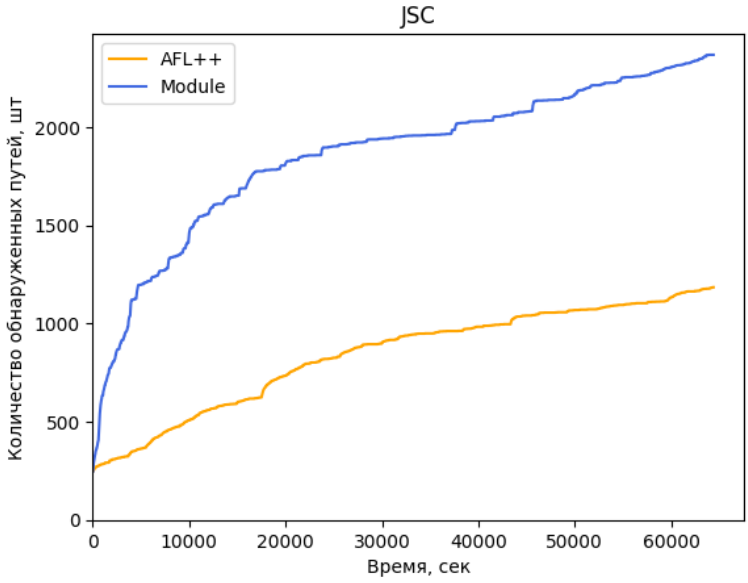

Рис. 8. Графики скорости обнаружения новых путей в коде в интерпретаторе JSC.
Fig. 8. Graphs of the speed of discovering new paths in code in the JSC engine.

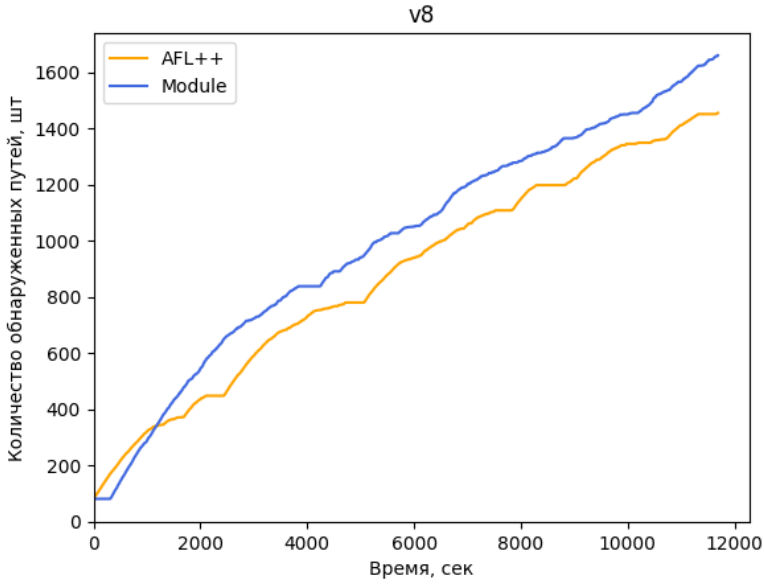

Рис. 9. Графики скорости обнаружения новых путей в коде в интерпретаторе v8.
Fig. 9. Graphs of the rate of discovery of new paths in the code in the v8 engine.

Табл. 2. Сравнение количества обнаруженных зависаний
Table 2. Comparison of number of detected hangs

	JSC	**V8**
AFL++	82	52
Module	500+	322

5. *Заключение*

Проблема неэффективных мутаций является одной из ключевых в фаззинге программного обеспечения, обрабатывающего сложноструктурированные входные данные, такие как программный код. Необходимость сохранения синтаксиса и семантики JavaScript кода требует изменения текущего подхода к обрезке и мутации кода. Мутации на уровне AST-деревьев позволяют производить эффективные мутации, сохраняя требуемую семантику для эффективного обнаружения новых путей в тестируемом коде.

Список литературы / References

[1]. 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, 2021 (4), С. 83-106, DOI: 10.17308/sait.2021.4/3800.

[2]. Groß S. FuzzIL: Coverage guided fuzzing for javascript engines // Department of Informatics, Karlsruhe Institute of Technology, 2018.

[3]. H. Han, D. Oh, and S. K. Cha. Codealchemist: Semantics-aware code generation to find vulnerabilities in javascript engines. In Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA, Feb. 2019

[4]. Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with code fragments. In Proceedings of the 21st USENIX Security Symposium (USENIX Security). 445–458. https://doi.org/10.5555/2362793.2362831.

[5]. Wang J, Chen B, Wei L, Liu Y. Skyfire: Data-Driven Seed Generation for Fuzzing. In: 2017 IEEE Symposium on Security and Privacy (SP). IEEE; 2017 – с. 579–94. DOI: 10.1109/SP.2017.23.

[6]. Aschermann C. et al. NAUTILUS: Fishing for Deep Bugs with Grammars //NDSS. – 2019, DOI: 10.14722/ndss.2019.23xxx.
[7]. 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). 724–735. https: //doi.org/10.1109/ICSE.2019.00081.
[8]. Fioraldi A. et al. AFL++ combining incremental steps of fuzzing research //Proceedings of the 14th USENIX Conference on Offensive Technologies. – 2020. – p. 10.
[9]. 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.
[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, DOI: 10.48550/arXiv.2001.04107.
[11]. Gopinath R., Görz P., Groce A. Mutation analysis: Answering the fuzzing challenge //arXiv preprint arXiv:2201.11303. – 2022, DOI: 10.48550/arXiv.2201.11303.
[12]. Старцев Е. В. Разработка алгоритмов и моделирование динамической типизации в программах для технических систем: дис. – URL: http://www.csu.ru/scientific-departments/PublishingImages/D21229602/startsev_ev/Диссертация (Старцев ЕВ).pdf – С. 73-86, 2015.

Информация об авторах / Information about authors

Наталья Сергеевна ЕРОХИНА является сотрудником Академии ФСО России. Её научные интересы включают информационную безопасность, фаззинг-тестирование, алгоритмы машинного обучения.

Natalya EROKHINA is employee in Academy of the Federal Guard Service of Russian Federation. Her research interests include information security, fuzzing testing, and machine learning algorithms.