ПОЛИТЕХ
Санкт-Петербургский
политехнический университет
Петра Великого

На правах рукописи

Ерохина Наталья Сергеевна

Метод генерации семантически корректного кода для фаззинг-тестирования JavaScript-интерпретаторов на основе мутации АСД фрагментов кода

2.3.6. Методы и системы защиты информации, информационная безопасность

АВТОРЕФЕРАТ
диссертации на соискание ученой степени
кандидата технических наук

Орёл
2024

Работа выполнена в федеральном государственном казенном военном образовательном учреждении высшего образования «Академия Федеральной службы охраны Российской Федерации», г. Орёл.

Научный руководитель:
доктор технических наук, доцент Козачок Александр Васильевич.

Официальные оппоненты:
доктор технических наук, старший научный сотрудник Марков Алексей Сергеевич, акционерное общество «Научно-производственное объединение «Эшелон», президент, г. Москва.

кандидат технических наук Самарин Николай Николаевич, Федеральное государственное унитарное предприятие «Научно-исследовательский институт «Квант», начальник научно-исследовательского отделения, г. Москва.

Ведущая организация:
федеральное государственное автономное образовательное учреждение высшего образования «Северо-Кавказский федеральный университет», г. Ставрополь

Защита состоится «11» февраля 2025 г. в 11 часов на заседании диссертационного совета У.2.3.6.12 федерального государственного автономного образовательного учреждения высшего образования «Санкт-Петербургский политехнический университет Петра Великого» (195251, г. Санкт-Петербург, ул. Политехническая, 29, Главный корпус, аудитория 175).

С диссертацией можно ознакомиться в библиотеке и на сайте www.spbstu.ru федерального государственного автономного образовательного учреждения высшего образования «Санкт-Петербургский политехнический университет Петра Великого».

Автореферат разослан «___» _____ 2024 г.

Ученый секретарь
диссертационного совета У.2.3.6.12,
кандидат физико-математических наук, доцент  Шенец Николай Николаевич

ОБЩАЯ ХАРАКТЕРИСТИКА РАБОТЫ

Актуальность работы. В 2024 году согласно отчёту аналитического агентства Meltwater[1] в мире насчитывается 5,35 миллиарда пользователей сети Интернет, что составляет 66,2% мирового населения, 69,4% населения имеют собственный мобильный телефон. На каждом из этих устройств работает веб-браузер или аналогичная программа, способная обрабатывать и отображать контент веб-сайтов. Большинство веб-страниц встраивают или загружают исходный код, написанный на языке программирования JavaScript. В 2024 году JavaScript является одним из самых широко используемых языков программирования в мире. По данным рейтинга W3Techs[2] на октябрь 2024 года 98,9% всех веб-сайтов используют язык JavaScript. Веб-браузеры совершенствуются и становятся все более сложными, осуществляя обработку не только открытого текста и HTML, но и изображений, видео и других форматов данных. Любой современный веб-браузер поддерживает язык динамической разработки JavaScript без использования дополнительного программного обеспечения (ПО) с помощью встроенного JavaScript-интерпретатора (англ. JavaScript engine).

Среди всей архитектуры веб-браузера поиск уязвимостей в JavaScript-интерпретаторах является наиболее сложной и потому востребованной задачей в тестировании веб-браузеров. Согласно Национальной базе данных уязвимостей (англ. National Vulnerability Database, NVD) 43% всех уязвимостей, обнаруженных в веб-браузерах Microsoft Edge и Google Chrome, были уязвимостями интерпретатора JavaScript. Высокая сложность современных JavaScript-интерпретаторов, обусловленная большим объемом исходного кода, иногда достигающим нескольких миллионов строк, а также наличием уязвимостей и ошибок в нем, является фундаментальной проблемой, вызванной текущим состоянием информационных технологий. Стремительное развитие интернет технологий приводит к постоянному усложнению структуры JavaScript-интерпретаторов и увеличению объема их исходного кода. Данный факт негативно влияет на безопасность, что, в свою очередь, активизирует деятельность авторов вредоносных программ.

В наши дни большинство ошибок в программном обеспечении, связанных с удаленным выполнением кода и повышением привилегий, обнаруживаются при помощи фаззинг-тестирования. Вследствие имеющихся особенностей при тестировании JavaScript-интерпретаторов данный процесс может быть недостаточно эффективным.

Степень разработанности темы исследования. Тематика диссертационного исследования достаточно молода, вследствие этого существует множество разных подходов для решения задачи выявления уязвимостей в сложном программном обеспечении. Проблеме анализа безопасности программного обеспечения посвящено множество исследований отечественных и зарубежных ученых, таких как Д.П. Зегжда, П.Д. Зегжда, В.П. Иванников, А.И. Аветисян, А.С. Марков, А.Ю. Тихонов, В.А. Падарян, А.В. Козачок, А.В. Благодаренко, М.О. Шудрак, М.А. Новиков, И.В. Котенко, Г. Вигна, К. Крюгель, Э. Кирда. Обнаружению уязвимостей посвящены работы Я. Н. Алгулиева, И. Л. Алферова, А.А. Захарова, С.Л. Зефирова, Д.О. Карпеева, А.Г. Кащенко, А.А. Кононова, Д.А. Котенко, В.А. Курбатова, Г.А. Кустова, И.Д. Левятова, А.Г. Лысенко, А.В. Львова, А.Н. Назарова, А.Г. Остапенко, Г.А. Остапенко, А.О. Сидорова, М.В. Тимонина, C. Alberts, G. Brændeland, N.E. Fenton, F.V. Jensen, C. Kairab, M. Neil, A. Papoulis, T.R. Peltier, M. Tailor, и др. В работах S. Park, J. Wang, C. Aschermann, S. Lee предложена обработка сложноструктурированных данных на уровне *абстрактного синтаксического дерева (АСД, англ. Abstract Syntax Tree, AST)*. В результате анализа научных работ, посвященных рассматриваемой области, можно сделать вывод о существовании ряда неразрешенных вопросов, касающихся обнаружения уязвимостей безопасности в ПО, обрабатывающем сложноструктурированные данные, таком как JavaScript-интерпретаторы.

[1] https://www.meltwater.com/en/global-digital-trends.

[2] https://w3techs.com/technologies/details/cp-javascript.

Диссертационная работа выполнена в соответствии с одним из основных научных направлений Академии Федеральной службы охраны Российской Федерации.

Объектом исследования являются JavaScript-интерпретаторы.

Предметом исследования являются методы и средства генерации входных данных и фаззинг-тестирования JavaScript-интерпретаторов.

Целью работы является повышение эффективности фаззинг-тестирования JavaScript-интерпретаторов для выявления дефектов безопасности.

Для достижения поставленной цели в работе решались следующие **задачи**:

1. Проведение сравнительного анализа эффективности современных средств фаззинг-тестирования JavaScript-интерпретаторов, выделение существующих недостатков и причин их возникновения, а также определение возможных способов их устранения.
2. Разработка модели генерации входных данных для фаззинг-тестирования JavaScript-интерпретаторов на основе рекуррентной нейронной сети, учитывающей степень покрытия кода, позволяющей сжать входной корпус данных без потери исходного покрытия, что, в свою очередь, позволит повысить эффективность последующего фаззинг-тестирования.
3. Разработка метода мутации сложноструктурированных данных, сохраняющего синтаксис и семантику данных за счет модификации АСД фрагментов JavaScript-кода, который позволит сократить количество тестовых итераций, не открывающих новых путей, и тем самым повысить эффективность процесса обнаружения дефектов безопасности.
4. Разработка и практическая реализация архитектуры системы генерационного и мутационного фаззинг-тестирования JavaScript-интерпретаторов, основанной на разработанных модели генерации и методе мутации, позволяющая повысить эффективность последующего фаззинг-тестирования.
5. Проведение экспериментальных исследований предложенных решений, и оценка их эффективности.

Научная новизна диссертационного исследования состоит в следующем:

1. Разработана модель генерации входных данных, интегрирующая рекуррентные нейронные сети с оценкой покрытия кода, что впервые позволяет выявить синтаксис и семантику исходного сложноструктурированного корпуса данных и автоматически сгенерировать сжатый корректный корпус, повышающий покрытие кода JavaScript-интерпретаторов. В отличие от существующих решений, предложенная модель выявляет основные закономерности из исходных данных и сжимает их без потери в покрытии, что позволяет повысить эффективность последующего фаззинг-тестирования путем максимизации покрытия уникальных путей.
2. Разработан метод мутации JavaScript-кода на основе АСД, который сохраняет синтаксическую и семантическую корректность сложноструктурированных данных. В отличие от традиционных подходов, метод позволяет выполнять целенаправленные модификации кода, сохраняя его структуру и функциональность, что способствует увеличению скорости фаззинг-тестирования.
3. Впервые предложена архитектура системы, сочетающая генерационные и мутационные подходы в фаззинг-тестировании JavaScript-интерпретаторов. Разработанная архитектура включает модель генерации и метод мутации сложноструктурированных данных, что позволяет повысить эффективность выявления дефектов безопасности в JavaScript-интерпретаторах. В отличие от существующих систем, данная архитектура оптимизирует процесс тестирования за счет автоматизации и интеграции моделей машинного обучения для генерации и мутации данных.

Теоретическая значимость результатов работы заключается в развитии новых подходов к фаззинг-тестированию JavaScript-интерпретаторов, объединяющих методы машинного обучения, такие как рекуррентные нейронные сети, и подходы к мутации сложноструктурированных данных на основе АСД. Впервые предложены модель генерации и метод мутации, которые выявляют исходные закономерности данных, а также сохраняют синтаксическую и семантическую корректность мутированных данных, что позволяет повысить эффективность выявления дефектов безопасности при фаззинг-тестировании. Эти результаты расширяют существующую теоретическую базу автоматизированного тестирования и могут служить основой для дальнейших исследований в области разработки методов генерации и мутации данных, направленных на повышение надежности и безопасности программного обеспечения.

Практическая значимость результатов работы состоит в разработке и внедрении системы, сочетающей генерацию и мутацию входных данных для фаззинг-тестирования JavaScript-интерпретаторов, что позволяет существенно повысить эффективность фаззинг-тестирования для выявления дефектов безопасности. Разработанная модель генерации входных данных с использованием рекуррентных нейронных сетей и метод мутации на основе АСД позволяют интегрировать решения в существующие инструменты тестирования, улучшая их работу за счет повышения покрытия кода и сохранения корректности данных. Разработанные модель генерации и метод мутации в архитектуре системы генерационного и мутационного фаззинг-тестирования JavaScript-интерпретаторов обеспечивают повышение показателей эффективности фаззинг-тестирования по времени в 2,7 раза, по числу тестовых итераций – в 4,4 раза и повышает общую производительность выявления дефектов безопасности в 3,6 раза, что имеет высокую практическую ценность для организаций, занимающихся разработкой и исследованием программного обеспечения, и может быть применено в реальных промышленных и исследовательских задачах.

Методология и методы исследования. Для решения поставленных задач в диссертационной работе использовались методы системного анализа, машинного обучения, элементы теории компиляции, теории графов, теории вероятностей и математической статистики, методы компьютерной алгебры, методы анализа и виртуализации программного кода.

Положения, выносимые на защиту:

1. Модель генерации входных данных для фаззинг-тестирования JavaScript-интерпретаторов на основе рекуррентной нейронной сети, учитывающая степень покрытия кода.
2. Метод мутации сложноструктурированных данных, сохраняющий синтаксис и семантику данных за счет модификации АСД фрагментов JavaScript-кода.
3. Архитектура системы генерационного и мутационного фаззинг-тестирования JavaScript-интерпретаторов.

Внедрение результатов работы. Результаты работы внедрены в проектную деятельность ООО «Системы защиты информации», в учебный процесс РТУ МИРЭА при проведении практических и лабораторных занятий по дисциплине «Технологии машинного обучения в кибербезопасности».

Степень достоверности научных положений подтверждается представленным анализом научных работ по предмету исследования, обоснованностью выводов, результатами экспериментальных исследований и примерами их практического использования, апробацией основных результатов работы в научных изданиях и докладах на конференциях.

Соответствие специальности научных работников. Научные результаты соответствуют следующим пунктам паспорта специальности научных работников 2.3.6. Методы и системы защиты информации, информационная безопасность:

п. 5. Методы, модели и средства (комплексы средств) противодействия угрозам нарушения информационной безопасности в открытых компьютерных сетях, включая Интернет;

п. 15. Принципы и решения (технические, математические, организационные и др.) по созданию новых и совершенствованию существующих средств защиты информации и обеспечения информационной безопасности;

п. 17. Методы, модели и средства разработки безопасного программного обеспечения, выявления в нем дефектов безопасности, противодействия скрытым каналам передачи данных и выявления уязвимостей в компьютерных системах и сетях.

Апробация работы. Основные результаты работы были представлены на следующих конференциях: Международная научно-техническая конференция «Безопасные информационные технологии» (Москва, 2021, 2023), Всероссийская научно-практическая конференция «Методы и технологические средства обеспечения безопасности информации» (Санкт-Петербург, 2022, 2023, 2024), Открытая конференция Института системного программирования им. В.П. Иванникова Российской академии наук (Москва, 2023), Национальная научно-практическая конференция «Фундаментальные, поисковые, прикладные исследования и инновационные проекты» (Москва, 2023), Всероссийская межведомственная научная конференция «Актуальные направления развития систем охраны специальной связи и информации для нужд органов государственной власти РФ» (Орёл, 2021, 2023).

Публикации по теме диссертации. Результаты диссертации опубликованы в 13 работах, в том числе в 5 публикациях в рецензируемых журналах из перечня ВАК РФ, а также в 2 свидетельствах о регистрации программы для ЭВМ.

Структура и объем диссертации. Диссертация состоит из введения, трех глав, заключения, списка литературы из 122 наименований и 4 приложений. Общий объем работы составляет 131 страницу, в том числе 37 рисунков и 9 таблиц.

ОСНОВНОЕ СОДЕРЖАНИЕ РАБОТЫ

Во **введении** приведено обоснование актуальности темы исследования, сформулирована цель работы, а также выполнена постановка задач, необходимых для ее достижения. Выделены положения, выносимые на защиту, сформулирована научная новизна, теоретическая и практическая значимость.

В первой главе представлены результаты анализа структуры и функциональности JavaScript-интерпретаторов. Поверхность атаки JavaScript-интерпретатора в архитектуре веб-браузера представлена на рисунке 1. В случае со сложным ПО, таким как JavaScript-интерпретатор, поверхностей атаки достаточно много, некоторые из них будут вложенными, некоторые – могут пересекаться между собой, что является важным аспектом для обнаружения уязвимостей и дальнейшей их эксплуатации.

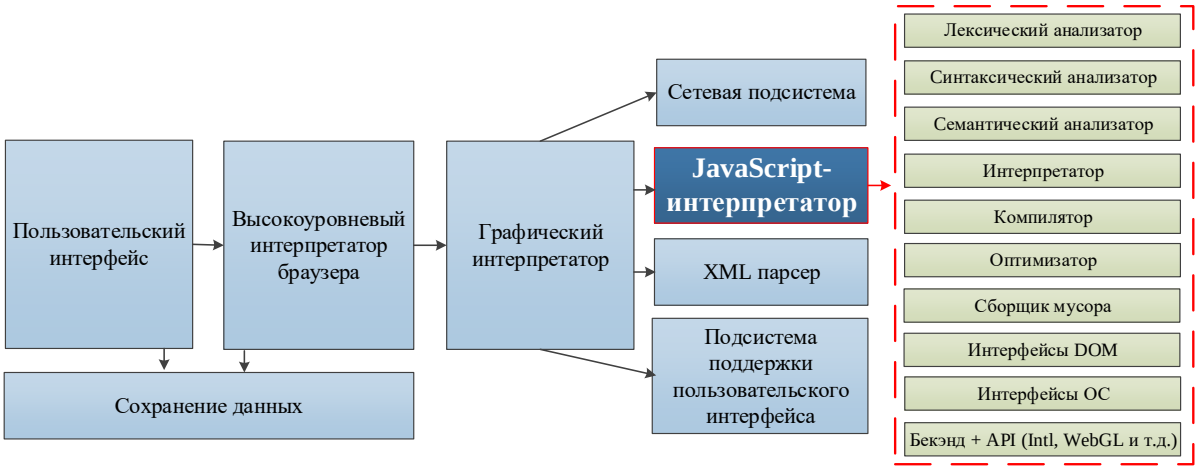

Рисунок 1 – Поверхность атаки JavaScript-интерпретатора веб-браузера

Полная по Тьюрингу природа JavaScript-интерпретаторов позволяет злоумышленникам внедрять сложный код, обнаруживающий уязвимости. В частности, JavaScript-интерпретаторы оказались в центре внимания исследователей безопасности по разным причинам: во-первых, потенциал обнаружения уязвимости в долгосрочной перспективе в них выше. Во-вторых, атака на JavaScript-интерпретатор гораздо более сложная, потому что технологически не так легко найти новую уязвимость. В-третьих, из соображений производительности они часто реализуются на небезопасных для памяти языках, что влечет за собой уязвимости, приводящие к повреждению памяти. В-четвертых, ввиду возросшей сложности их кода, увеличивается вероятность ошибок при разработке. Отчасти это связано с продолжающейся гонкой за производительностью среди поставщиков браузеров. Классические уязвимости, такие как переполнение буфера или использование после освобождения, редко встречаются в JavaScript-интерпретаторах, их заменили сложные и специфичные для предметной области уязвимости, систематизация наиболее распространенных из них представлена в первой главе.

Проанализированы руководящие документы, содержащие требования по безопасной разработке программного обеспечения:

– ГОСТ Р 56939-2016 «Защита информации. Разработка безопасного программного обеспечения. Общие требования».

– Требования по безопасности информации, устанавливающие уровни доверия к средствам технической защиты информации и средствам обеспечения безопасности информационных технологий (утверждены приказом ФСТЭК России № 76 от 02.06.2020 года).

– Методика выявления уязвимостей и недекларированных возможностей (НДВ) в программном обеспечении (утверждена приказом ФСТЭК России в декабре 2020 года).

Исследованы методы выявления уязвимостей и НДВ в ПО и приведена их классификация по трем группам: методы экспертного, динамического и статического анализа. Среди множества доступных сегодня методов обнаружения уязвимостей ПО, фаззинг наиболее распространен из-за его концептуальной простоты, низкого барьера для развертывания и огромного количества эмпирических свидетельств обнаружения уязвимостей реального программного обеспечения. Применительно к задаче выявления уязвимостей сложного ПО, такого как JavaScript-интерпретаторы, фаззинг-тестирование имеет ряд преимуществ:

– высокая степень автоматизации, что позволяет проводить тестирование большого объема кода за обозримое время;

– фаззинг может быть использован для тестирования любой системы, которая имеет любые 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 обработки результатов;

– существенно меньшее количество ложных срабатываний, относительно статических анализаторов;

– позволяет собрать общее представление об уровне защищенности тестируемого кода.

В главе представлена классификация фаззеров и проведен анализ существующих подходов в фаззинг-тестировании. Проведенный анализ позволил выделить ключевые особенности фаззинг-тестирования JavaScript-интерпретаторов:

– постоянно нарастающий и усложняющийся код современных интерпретаторов;

– отсутствие общедоступных синтаксически и семантически корректных, сжатых корпусов входных данных;

– отсутствие эффективной методологии мутации сложноструктурированных входных данных;

– низкая синтаксическая корректность и сложность преодоления внутренней проверки входных данных тестируемой программой при проведении манипуляций над данными;

– необходимость достижения максимального покрытия тестируемого кода интерпретаторов, часто составляющего более 500 тысяч строк.

Анализ исследований, посвященных фаззинг-тестированию JavaScript-интерпретаторов показал, что большинство работ сталкиваются с нехваткой качественного входного корпуса, а также с отсутствием эффективной методологии мутации сложноструктурированных входных данных.

Была осуществлена формальная постановка научной задачи, заключающаяся в повышении эффективности фаззинг-тестирования JavaScript-интерпретаторов для выявлении дефектов безопасности за счет внедрения модели генерации входных данных для фаззинг-тестирования JavaScript-интерпретаторов и метода мутации сложноструктурированных данных в едином контуре тестирования.

В фаззинг-тестировании тестируемая программа представляется как граф вида $K = (V, E)$, где V – множество вершин графа, E – множество дуг. В рамках анализа бинарного кода вершины графа представляются в виде базовых блоков кода, а дуги обозначают переходы между базовыми блоками. Для оценки покрытия кода важным параметром является граф потока управления (ГПУ, англ. Control Flow Graph, CFG), представленный в виде следующей четверки: $CFG = (V, E, s, e), s \in V, e \in V$, где V –

множество вершин, представляющих собой базовый блок на участке анализируемого кода; $E = V \times V$ – множество дуг; s – входная вершина; e – выходная вершина; $E(G)$ – множество дуг графа G; $start(G)$ – входная вершина графа G; $end(G)$ – выходная вершина графа G. Необходимо отметить, что ГПУ имеет лишь один вход $start(P)$ и один выход $end(P)$. *Путь* в графе представляется как упорядоченная последовательность пройденных вершин в ГПУ. По определению ГПУ является ориентированным графом с циклами, что критично для определения его покрытия.

Покрытие кода может оцениваться по различным параметрам: строкам кода, базовым блокам или уникальным путям в ГПУ, а также по покрытым ветвям, функциям и операторам. Полное покрытие по строкам, ветвям, функциям и операторам не гарантирует полного доверия к протестированному ПО и отсутствия уязвимостей в нем. Определим множество всех тестов для программы как $T = \{t_1, t_2, ..., t_n\}$, где n – количество тестовых итераций. Для оценки полноты покрытия кода тестовыми данными определим множество всех вершин во всех возможных путях ГПУ программы как M. Затем определим путь в коде для одной тестовой итерации t_i как множество $c_i = \{V_i'\}$, где V_i' – множество пройденных вершин в ГПУ; *уникальным путем* назовем тот, в котором $\exists v_i : c_i' = \{V_i'\} \cup \{v_i\}$, где c_i' является приращением множества пройденных вершин V_i'. Тогда общее покрытие кода в рамках тестирования, с точки зрения числа уникальных путей в коде, запишется как множество $P = \{c_1', c_2', ..., c_m'\}$, где m – число выявленных уникальных путей в коде.

Показатель эффективности фаззинг-тестирования JavaScript-интерпретаторов можно раскрыть как количественное измерение способности обнаружения новых путей в коде интерпретатора. Он может рассчитываться как отношение количества уникальных путей, выявленных в процессе тестирования, к числу выполненных тестовых итераций или к затраченному времени на тестирование. Тем самым демонстрируя какое количество ресурсов (времени и количества тестовых итераций) необходимо затратить для достижения результата, а именно выявления большего числа уникальных путей в тестируемом коде.

Таким образом, формулы для вычисления эффективности по запускам $E_{зап}$ и по времени $E_{врем}$ фаззинг-тестирования могут быть выражены как:

$$E_{зап} = \frac{M}{N}, \quad (1)$$

$$E_{врем} = \frac{M}{T}, \quad (2)$$

где M – общее количество уникальных путей, обнаруженных в коде JavaScript-интерпретатора за время тестирования,

N – общее количество тестовых итераций, произведенных в процессе фаззинг-тестирования,

T – общее время, затраченное в процессе фаззинг-тестирования.

Задача достижения завершенности тестирования и подтверждения отсутствия уязвимостей является неразрешимой для JavaScript-интерпретаторов. С точки зрения информационной безопасности, лишь покрывая 100% всех уникальных путей в ГПУ тестируемой программы, можно говорить об отсутствии уязвимостей в ней. Пути, отличимые хотя бы количеством итераций цикла – различные уникальные пути, поэтому общее количество путей в ГПУ является счетным множеством. Таким образом, необходимо решить задачу вычисления всех возможных уникальных путей в ориентированном графе с

циклами. Данная задача является NP-полной, ввиду того, что нет возможности решения данной задачи полиномиальными алгоритмами.

Таким образом, научная задача в формальном виде может быть определена следующим образом:

$$GEN(T_{RT}) = (T', P') : |T'| < |T_{RT}|, P_{RT} \subseteq P', \qquad (3)$$

где $GEN()-$ функция генерации, сжимающая исходное множество регрессионных тестов, сохраняя исходное покрытие и преувеличивая его.

$$MUT(T') = c_i' : P'' = P' \cup \{c'\} \mid E > E_0, \qquad (4)$$

где $MUT()$ – функция мутации, за счет простых и быстрых мутаций изменяет тестовые входы, сгенерированные функцией генерации, сохраняя при этом синтаксис и семантику входа, тем самым повышая эффективность тестирования;

E_0 – эффективность фаззинг-тестирования без использования разработанных методов.

Далее в главе сделан вывод об актуальности задачи диссертационного исследования, обоснован выбор целевых и вспомогательных методов тестирования.

Во второй главе представлена *модель генерации входных данных для фаззинг-тестирования JavaScript-интерпретаторов на основе рекуррентной нейронной сети (РНС), учитывающая степень покрытия кода*.

В главе исследовано влияние уровня представления данных: исходного кода, абстрактного синтаксического дерева и байт-кода на сохранение синтаксической и семантической корректности сложноструктурированных данных.

Многими исследованиями наглядно продемонстрирована эффективность представления и дальнейшей обработки сложноструктурированных данных в виде абстрактного синтаксического дерева (АСД) – конечного, помеченного, ориентированного дерева, в котором внутренние вершины сопоставлены с операторами языка программирования, а листья – с соответствующими операндами. Преобразование JavaScript-файла в АСД происходит в соответствии со спецификацией ECMAScript, что гарантирует корректность и единство данного представления.

В отличие от других уровней АСД фактически моделируют тестовые входные данные как объекты с именованными свойствами, четко соблюдая структуру JavaScript-кода. Таким образом, представление на уровне АСД обеспечивают подходящую степень детализации для фаззера, а также позволяет сохранять синтаксис и семантику языка JavaScript.

Введено определение АСД фрагмента, необходимого для дальнейшей работы. ***АСД фрагмент*** дерева $T = (N, E, n_0)$ – это поддерево $T_i = (N_i, E_i, n_i)$, где

- $n_i \in N, C(n_i) \neq \varnothing$;
- $N_i = \{n_i\} \cup C(n_i)$;
- $E_i = \{(n_i; n') \mid n' \in C(n_i)\}$.

где N – множество узлов, E – множество ребер, n_0 – корневой узел, $C(n_i)$ – множество непосредственных потомков n_i, n_i – узел в T_i.

Используя АСД фрагменты, инкапсулирующие структурные связи, АСД кодируется в последовательности фрагментов. Данная процедура однозначно приводит любой JavaScript-код в удобный формат в виде последовательности фрагментов единичной длины, что позволяет удобно оперировать ими в дальнейшем, не нарушая синтаксис и семантику

языка JavaScript (Рис. 2). Таким образом, появляется возможность фиксировать глобальные отношения композиции между АСД фрагментами для дальнейшей их обработки.

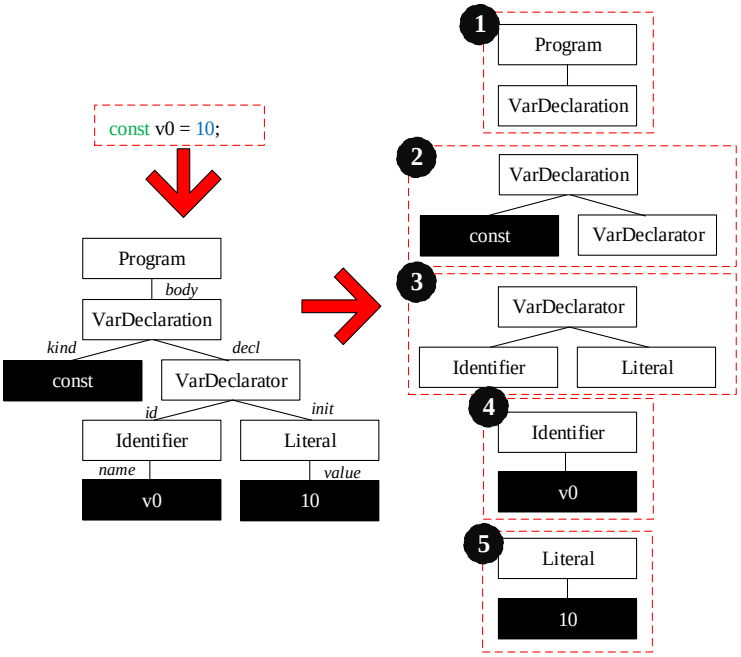

Рисунок 2 – Процесс фрагментации АСД

Также в главе проанализировано влияние исходной семантики входного корпуса на эффективность всего процесса фаззинг-тестирования. Одна из основных задач при фаззинг-тестировании ПО со сложноструктурированными входными данными, такими как JavaScript-код – собрать синтаксически и семантически корректные исходные данные.

В главе приведены утверждения, подтверждающие эффективность выбора регрессионных тестов (РТ) в качестве входных данных для фаззинга JavaScript-интерпретаторов:

– более 95% АСД фрагментов синтаксически перекрываются между файлами РТ JavaScript-интерпретаторов и PoC-фрагментами (англ. Proof-of-concept, PoC) кода, запускающими уязвимости из базы общеизвестных уязвимостей и рисков информационной безопасности (англ. Common Vulnerabilities and Exposures, CVE);

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

– разработчики склонны совершать типичные и повторяющиеся ошибки в разных реализациях JavaScript-интерпретаторов.

Одной из основных причин возникновения ошибок в коде является неполное или некорректное исправление прошлых ошибок. Между тем, такие же или похожие на распространенные ошибки часто возникают в ходе эволюции программного обеспечения. В соответствии с существующими исследованиями по обнаружению уязвимостей JavaScript-интерпретаторов было выявлено более 95% синтаксического перекрытия между фрагментами регрессионных тестов интерпретаторов и фрагментами кода, запускающими уязвимости. Данные факты указывают на то, что вероятность обнаружения новой уязвимости с использованием семантики регрессионных тестов, гораздо выше. Рисунок 3 иллюстрирует схожесть фрагментов кода PoC-файла, запускающего CVE-2017-11911 и регрессионного теста для интерпретатора ChakraCore.

```
function f0() {
    'use asm';
    const v0 = 1.0;
    function f1() {
        var v1 = v0;
        var v0 = 0;
    }
    return f1;
}
```

```
function f0() {
    'use asm';
    const v0 = Math.fround(1);
    function f1() {
        var v1 = v0;
        var v2 = Math.fround(4);
    }
    return {f1: f1}
}
```

Рисунок 3 – Сравнение PoC-файла, запускающего CVE-2017-11911 и файла регрессионного теста для интерпретатора ChakraCore

Таким образом, подтверждается, что вероятность обнаружения новой уязвимости путем сборки фрагментов кода из существующих наборов РТ, гораздо выше. Однако, для проведения эффективного фаззинг-тестирования входной корпус должен не только обеспечивать хорошее покрытие кода, но и отвечать условию отсутствия избыточности. Таким образом, предлагается модель генерации входных данных для фаззинг-тестирования JavaScript-интерпретаторов на основе рекуррентной нейронной сети, обеспечивающая большее покрытие кода интерпретатора, и соблюдающая условие отсутствия избыточности.

Применение методов машинного обучения (МО) во многих исследованиях в фаззинг-тестировании демонстрирует впечатляющие результаты. Преимуществом нейронных сетей является возможность обработки больших объемов данных с целью выявления закономерностей, что может быть применено для генерации сложноструктурированных данных при фаззинге JavaScript-интерпретаторов.

Модель генерации сложноструктурированных данных для фаззинг-тестирования JavaScript-интерпретаторов на основе РНС представлена на рисунке 4. Данная модель включает в себя РНС LSTM, с помощью которой генерируется новый корпус входных данных на основе базы РТ для JavaScript-интерпретаторов, обеспечивающий высокое начальное покрытие кода для последующего фаззинг-тестирования.

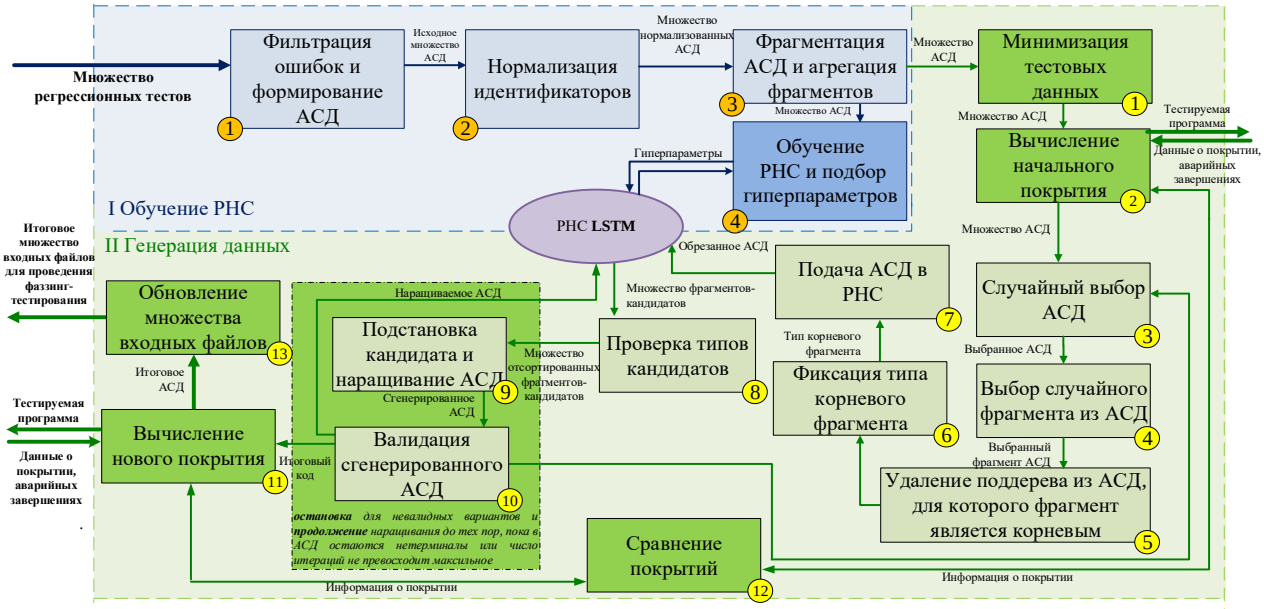

Рисунок 4 – Модель генерации входных данных на основе РНС

Разработанная модель, за счет использования РНС LSTM, выявляет закономерности, заложенные во входном корпусе РТ, и за счет этого генерирует новый корпус входных данных, обеспечивающий высокое начальное покрытие кода.

Данная модель описывает два этапа генерации данных:

– обучение РНС;
– генерация данных с помощью обученной РНС.

В ходе проведения анализа существующих решений была выбрана идея обучения РНС не токенами, а последовательностями АСД фрагментов, которые можно использовать в качестве лексикона для работы с РНС. Такой вариант представления позволяет фиксировать глобальные отношения композиции между фрагментами кода для выбора следующего фрагмента и генерации данных.

Первый этап состоит из следующих шагов:

шаг 1. Фильтрация ошибок и формирование АСД – фильтрация ошибок нацелена на сокращения времени процесса генерации за счет исключения ошибочных данных.

шаг 2. Нормализация идентификаторов – на данном этапе все имена функций и переменных приводятся к общему виду с целью дедупликации имен функций и переменных для исключения повторов одинаковых фрагментов. Встроенные функции и объекты интерпретаторов, зависящие от языка, исключаются из этапа нормализации, поскольку их нормализация влияет на семантику исходного АСД.

шаг 3. Фрагментация АСД и агрегация фрагментов – нацелены на выделение семантики из корпуса регрессионных тестов и повышение скорости вычислений, также на данном шаге происходит удаление редковстречающихся фрагментов АСД.

шаг 4. Подбор гиперпараметров и обучение РНС.

Результатом первого этапа являются:

– обученная РНС;
– сформированное множество последовательностей АСД-фрагментов.

Второй этап включает в себя следующие шаги:

шаг 1. первичная минимизация входных данных – направлена на удаление избыточности в данных для ускорения процесса генерации;

шаг 2. вычисление начального покрытия кода входными данными – покрытие считается эталонным, до тех пор, пока не будет получено превышающее его значение, в таком случае оно обновляется;

шаг 3. случайный выбор АСД из множества, сформированного в 1 фазе при обучении РНС;

шаг 4. выбор случайного фрагмента из АСД;

шаг 5. удаление поддерева из АСД, для которого выбранный фрагмент является корневым;

шаг 6. фиксация типа корневого фрагмента – предназначен для сохранения синтаксиса и семантики исходного АСД;

шаг 7. подача на вход языковой модели обрезанного АСД;

шаг 8. получение множества фрагментов-кандидатов от РНС и отсев некорректных кандидатов по типу – число фрагментов, генерируемых РНС является параметром конфигурации;

шаг 9. подстановка фрагмента-кандидата в АСД;

шаг 10. валидация сгенерированного АСД – остановка наращивания в случае, если АСД не валидно (переход на шаг 3), а также если в АСД не осталось нетерминалов, либо достигнуто максимальное число итераций, которое является параметром конфигурации (переход на шаг 11), продолжение наращивания в противном случае (переход на шаг 6);

шаг 11. вычисление нового покрытия кода каждым из оставшихся корректных АСД;

шаг 12. анализ прироста покрытия кода;

шаг 13. обновление базы входных файлов теми, что вызвали прирост эталонного покрытия кода и обновление эталонного значения покрытия.

Далее происходит выбор нового АСД и повтор шагов 3-13 или завершение этапа в зависимости от достижения цели.

В качестве исходных данных для 2 этапа используется база начальных данных для генерации, сформированная на 1 этапе. Из нее случайно выбирается одно АСД, соответствующее некоторому файлу РТ. Начальное покрытие запоминается. Из выбранного АСД выбирается случайный фрагмент и удаляется все поддерево, для которого этот фрагмент является корневым. Далее тип корневого фрагмента фиксируется.

Обрезанное АСД подается в РНС в виде последовательности его фрагментов, которая выдает список фрагментов-кандидатов. Далее производится дополнительная проверка типа фрагмента, чтобы отбросить те, которые не соответствуют зафиксированному типу. Оставшиеся кандидаты фиксируются и получившиеся новые АСД вновь подаются в НМ параллельно и наращиваются до тех пор, пока в дереве не останется нетерминальных символов.

Далее для всех сформированных АСД производится обратный перевод в JavaScript-код, производится фильтрация ошибочных вариантов и в анализаторе покрытия кода вычисляется покрытие тестируемого кода. Прошедшие валидацию варианты, повышающие исходное покрытие, снова проходят этапы выбора случайного фрагмента, удаление поддерева, фиксации типа корневого фрагмента и подаются в РНС для генерации новых кандидатов. Таким образом, наращивание новых АСД из одного исходного выполняется параллельно до тех пор, пока остаются валидные варианты и продолжается увеличение покрытия кода. Если покрытие кода начинает ухудшаться, то наиболее успешный вариант записывается в исходную базу и из нее выбирается следующее АСД.

Псевдокод алгоритма работы второго этапа генерации представлен на рисунке 5.

```
   Input: the test input to be trimmed input
   Output: the set of trimmed test T
 1 Function Trimming(input)
 2 T = ∅
 3 E = ∅ // Множество ошибок
 4 tree ← ParseToAst(input)
 5 coverage ← RunEngine(tree) // Вычиление начального
     покрытия
 6 seq ← TraverseAst(tree)
 7 count ← CountNodes(tree) // Вычиление числа узлов дерева
 8 step = count
 9 while step > 1 do
10 |   removable ← seq[step] // Выбор узла для обрезки
11 |   trimmed, E ← RemoveNode(tree, removable) // Обрезка
   |    дерева
12 |   if E == ∅ then
13 |   |   coverage_new ← RunEngine(trimmed)
14 |   |   if coverage_new ≥ coverage then
15 |   |   |   T = T ∪ {trimmed} // Добавление экземпляра в
   |   |   |    последовательность
16 |   |   end
17 |   end
18 |   step = step − 1
19 end
20 return T
```

Рисунок 5 – Псевдокод алгоритма этапа генерации данных

Таким образом, происходит увеличение исходного покрытия кода для базы входных данных для фаззинга. С целью сокращения избыточности входных данных применятся минимизатор тестовых данных, который удаляет избыточный код, не влияющий на покрытие.

Данный подход позволяет за счет использования РНС, а также обратной связи по покрытию исходного кода, увеличивать качество исходных данных для фаззинга, что решает одну из актуальных проблем при обработке сложноструктурированных входных данных.

В рамках данного исследования в качестве исходных данных была собрана база РТ различных JavaScript-интерпретаторов размером 49475 файлов, была выбрана РНС долгой краткосрочной памяти (англ. Long Short Term Memory, LSTM), JavaScript-интерпретаторы V8, JavaScriptCore, ChakraCore и JerryScript произведено вычисление суммарного покрытия их кода исходными РТ (Табл. 1).

Таблица 1 – Результаты вычисления исходного покрытия кода JavaScript-интерпретаторов

JavaScript-интерпретатор	Исходное покрытие, %		Входной корпус
	По строкам	По функциям	
V8	40,4% (257122 / 636765)	35,5% (153595 / 432185)	49475
JavaScriptCore	51,5% (222199 / 431062)	51,1% (125741 / 245920)	
ChakraCore	63,6% (171815 / 270360)	35,2% (36619 / 104156)	
JerryScript	60,3% (28063 / 46511)	60,4% (1279 / 2117)	

Для получения актуальной оценки покрытия кода необходимо проводить динамическую бинарную инструментацию каждого линейного блока в ГПУ для каждой тестовой итерации. Под инструментацией понимается процесс внедрения специального кода, отслеживающего факт передачи управления, перед каждым входом в линейный блок в рамках ГПУ. 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, которые выполняются во время тестового прогона. Для сбора информации о покрытии кода используются утилиты gcov и lcov. Представленные выше данные по начальному покрытию кода свидетельствуют о том, что почти половина кода интерпретаторов не затрагивается при использовании базы РТ.

Был реализован прототип генератора, реализующий представленную модель. Результаты работы генератора были сведены в таблицу 2. Результирующее покрытие представлено на графике (Рис. 6). Входные корпуса сокращены более чем на 99% для каждого из JavaScript-интерпретаторов. При этом покрытие кода интерпретаторов было увеличено: для V8 покрытие увеличено на 1,1% по строкам и 4,7% по функциям; для JavaScriptCore – на 13,2% и 19,3%; для ChakraCore – на 12,6% и 16,4%; для JerryScript – на 4,9% и 13,5%. Относительно малое увеличение покрытия кода для интерпретатора V8 в сравнении с другими обосновано его сложностью. Для заметного увеличения покрытия сложного и объемного кода необходима более длительная генерация корпуса.

Таблица 2 – Результаты вычисления итогового покрытия кода JavaScript-интерпретаторов новыми корпусами

JavaScript-интерпретатор	Исходное покрытие, %		Входной корпус
	По строкам	По функциям	
V8	40,8% (260037 / 636765)	37,2% (160785 / 432185)	397
JavaScriptCore	58,3% (251470 / 431062)	61,1% (150012 / 245920)	257

JavaScript-интерпретатор	Исходное покрытие, %		Входной корпус
	По строкам	По функциям	
ChakraCore	71,6% (193486 / 270360)	42,1% (43832 / 104156)	292
JerryScript	63,3% (29435 / 46511)	68,6% (1452 / 2117)	327

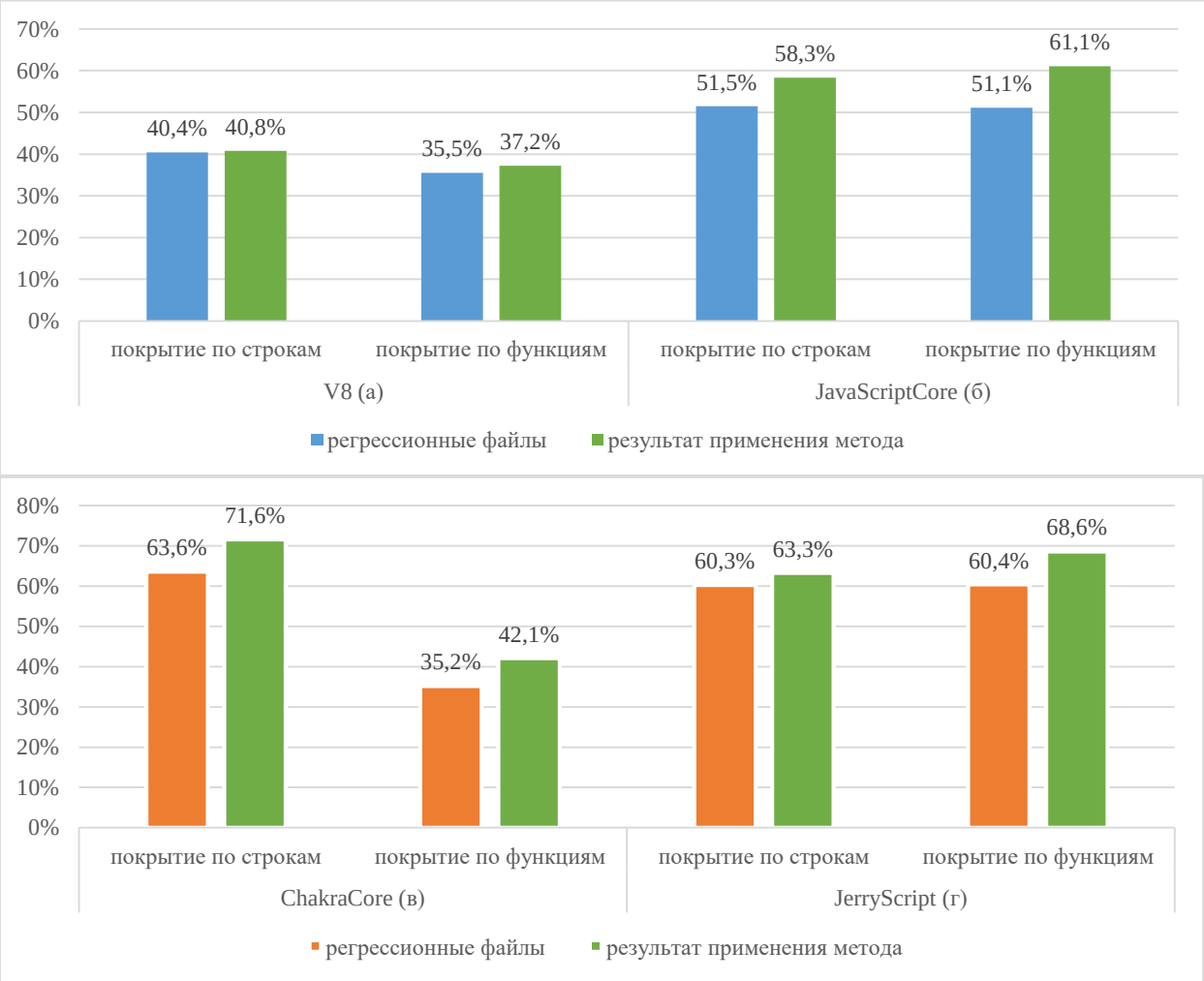

Рисунок 6 – Диаграммы итогового покрытия кода после применения метода генерации для исследованных JavaScript-интерпретаторов

Представленная модель генерации сложноструктурированных данных позволяет выявлять семантику из входного корпуса данных и генерировать новый корпус с как минимум таким же покрытием, но без избыточности в данных. Что является критически важным аспектом для проведения последующего фаззинг-тестирования.

Третья глава посвящена разработке *метода мутации сложноструктурированных данных и архитектуры системы генерационного и мутационного фаззинг-тестирования JavaScript-интерпретаторов* в едином контуре тестирования, позволяющих повысить эффективность фаззинг-тестирования JavaScript-интерпретаторов.

В главе представлен обзор существующей схемы работы одного из наиболее популярных и быстроразвивающихся современных фаззеров с обратной связью – AFLPlusPlus (AFL++).

Выявлены его основные недостатки:

−существующие алгоритмы обрезки и мутации данных, встроенные в фаззер AFL++, производятся в их битовом представлении, что разрушают синтаксис и семантику сложноструктурированных данных;

−большая часть данных отбрасывается синтаксическим анализатором;

−большая часть данных после обрезки теряет семантическую ценность, что сильно влияет на эффективность AFL++, поскольку ему нужно тратить больше времени на фаззинг тестовых входных данных, структуры которых разрушены, что сильно ограничивает возможности фаззера в поиске глубоких ошибок.

Метод мутации сложноструктурированных данных, сохраняющий синтаксис и семантику данных за счет модификации АСД фрагментов JavaScript-кода, включает в себя реализацию двух стратегий: обрезки и мутации АСД JavaScript-кода. Для эффективной мутации сложноструктурированных данных необходима их предварительная минимизация за счет обрезки АСД с сохранением покрытия кода, что позволяет сократить число потенциальных мутаций и, тем самым, потенциальных синтаксических ошибок и повысить эффективность дальнейшего фаззинг-тестирования.

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

Схема и псевдокод предложенного метода представлена на рисунках 9-10.

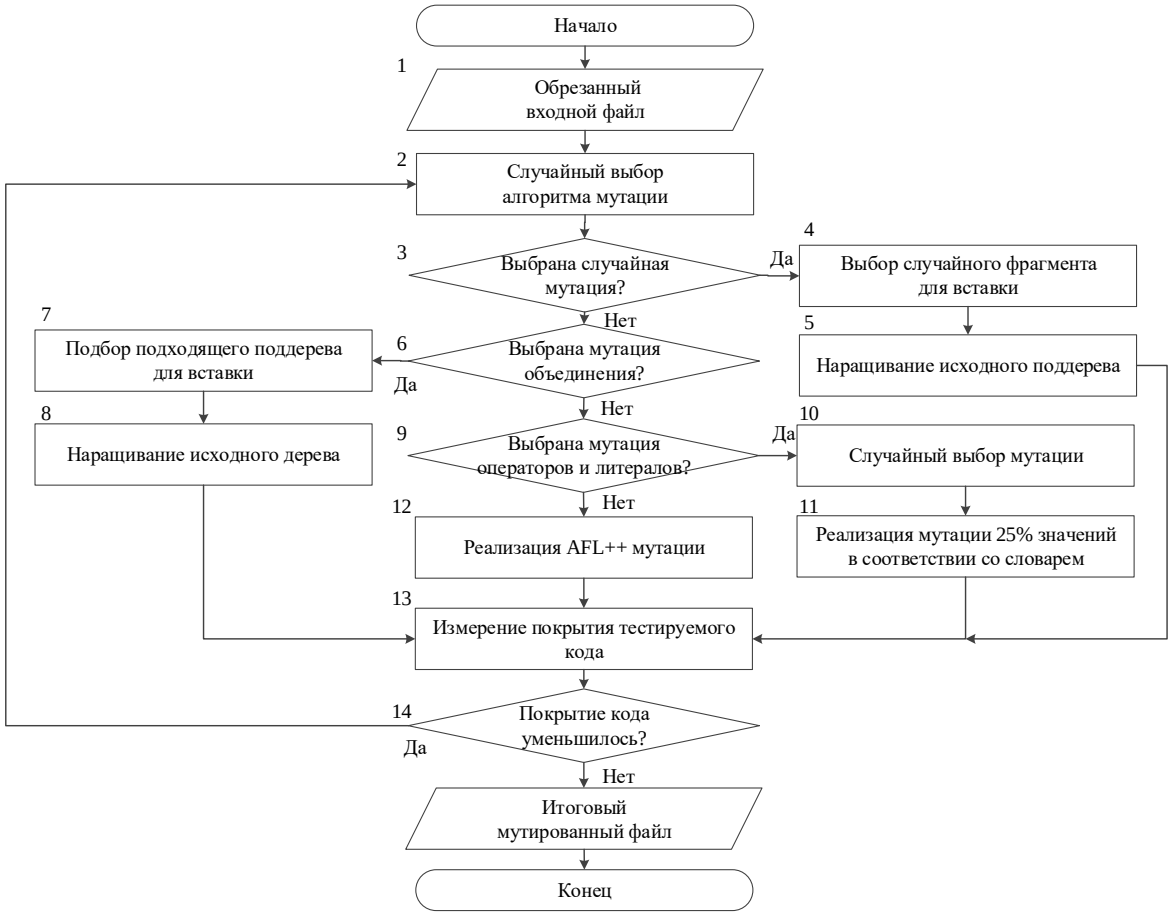

Рисунок 9 – Схема метода мутации АСД

```
Input: Мутируемые входные данные input
Output: Мутированное AST mutatedTree
1  Function Mutation(input)
2  E = ∅ // Множество ошибок
3  tree ← ParseToAst(input)
4  while True do
5  |  mutStrategy = RandomChoice(mutateNodes,
   |                              mutateLiterals,
   |                              mutateExpressions,
   |                              mutateSubtrees)
6  |  mutatedTree, E ← mutStrategy(tree)
7  |  if mutatedTree ≠ None ∧ E == ∅ ∧ mutatedTree ≠ tree then
8  |  |  mutatedCode ← ParseToCode(mutatedTree)
9  |  |  return mutatedCode
10 |  end
11 end
```

Рисунок 10 – Псевдокод алгоритма мутации АСД

Случайная мутация узлов (блоки 4-5) выбирает случайный узел дерева и заменяет его случайно сгенерированным новым поддеревом, корнем которого является тот же нетерминал. Новые узлы АСД выбираются из базы фрагментов АСД, сформированной из корпуса РТ. Псевдокод стратегии мутации узлов JavaScript-кода представлен на рисунке 11.

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

Рисунок 11 – Псевдокод алгоритма мутации АСД узлов

Мутация объединения (блоки 7-8) направлена на объединение АСД, помещая поддерево из одного АСД в другой. Для этого выбирается случайный внутренний узел, который становится корнем заменяемого поддерева и из АСД в очереди берется случайное поддерево, корнем которого является тот же нетерминал. Псевдокод алгоритма мутации объединения JavaScript-кода представлен на рисунке 12.

Мутация объединения происходит по следующим шагам:

шаг 1. Запрос узла с выбранным типом из базы РТ. Поиск в АСД источника узла, который имеет тот же тип, что и мутируемый.

шаг 2. Анализ применимости нового АСД фрагмента, начиная с выбранного узла. Новый узел применим только тогда, когда он не содержит ключевых слов, которые неприменимы в текущем контексте.

шаг 3. Подготовка вставки узла в АСД:

шаг 3.1. Замена идентификаторов переменных, если они не объявлены в мутируемом АСД, во вставляемом фрагменте или не являются объектом среды выполнения.

шаг 3.2. Если во вставляемом фрагменте присутствуют новые имена функций или классов (не объявленные в мутируемом АСД и не являющиеся объектами среды выполнения), то копировать их в глобальный контекст из АСД источника.

```
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
```

Рисунок 12 – Псевдокод алгоритма мутации объединения АСД фрагментов

Мутация литералов и выражений направлена на замену 25% значений узлов в АСД типа «Literal» или типа «Expression» (блоки 10-11 соответственно) на инверсные, либо случайно выбранные «интересные» значения из представленного словаря. Узлы типа «Literal» в JavaScript – это строки, числа, регулярные выражения.

AFL мутация (блок 12) выполняет мутации, которые также используются в AFL++, например, перестановку битов с целью проверки синтаксического анализатора интерпретатора.

Встроенными стратегиями мутаций в AFL++ являются:

- инверсия (побитовая операция НЕ) одного/двух/четырех битов;
- инверсия одного/двух/четырех байтов;
- арифметика – вычитание или добавление небольших целых чисел к 8-/16-/32-битным значениям);
- перезапись значений – установка «интересных» 8-/16-/32-битных значений;
- «хаос» – случайное применение инверсии битов, инверсии байтов, арифметики и перезаписи значений;
- объединение двух случайных тестовых входных данных из очереди, а затем нанесение хаоса.

Архитектура системы генерационного и мутационного фаззинг-тестирования JavaScript-интерпретаторов, разработанная на базе фаззера AFL++, представлена на рисунке 13.

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

–генерационный фаззинг на основе модели генерации входных данных за счет рекуррентной нейронной сети LSTM;

–быстрый мутационный фаззинг.

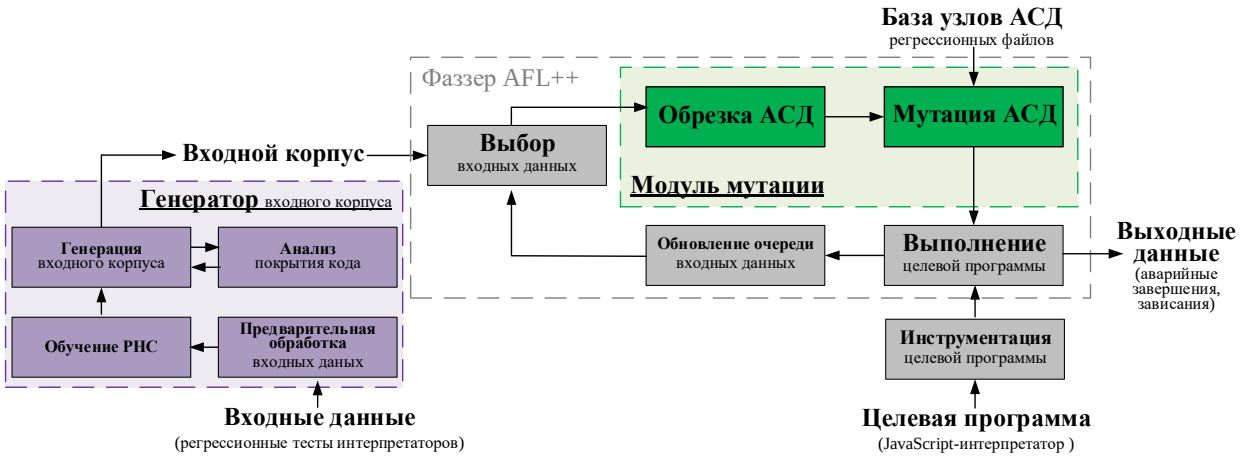

Рисунок 13 – Архитектура системы генерационного и мутационного фаззинг-тестирования JavaScript-интерпретаторов

Архитектура фаззинг-тестирования JavaScript-интерпретаторов была реализована за счет внесения в существующий фаззер (AFL++) дополнительного модуля генерации входных данных для фаззинг-тестирования JavaScript-интерпретаторов и модуля мутации сложноструктурированных данных. Система была реализована на языках программирования Python и JavaScript, получены 2 свидетельства о государственной регистрации программ для ЭВМ. Разработанный программный модуль генерации входных данных для фаззинг-тестирования JavaScript-интерпретаторов может быть внедрен в существующие фаззеры с целью минимизации входных корпусов и повышения скорости тестирования. Разработанный программный модуль, реализующий метод мутации сложноструктурированных данных, может быть внедрен в существующий фаззер AFL++ с целью повышения эффективности мутаций сложноструктурированных входных данных тестирования ПО, обрабатывающего сложноструктурированные входные данные, а также повышения скорости обнаружения новых путей в коде, тем самым повышая эффективность проводимого фаззинг-тестирования.

Для проведения экспериментов были выбраны наиболее распространенные JavaScript-интерпретаторы V8[3] и JavaScriptCore[4], а также база из 49475 файлов РТ различных интерпретаторов.

Для сравнения эффективности разработанной архитектуры был выбран оригинальный фаззер AFL++, а также существующие встраиваемые модули мутации для AFL++: Grammar-Mutator[5], основанный на грамматике языка JavaScript и Superion-Mutator[6], комбинирующий в себе знания о грамматике и простые АСД мутации. Представленные фаззеры предлагают различные подходы к мутациям входных данных, поэтому при сравнении с ними можно оценить, как предложенные улучшения стратегии повлияли на эффективность подхода.

Результаты тестирования представлены для двух наиболее распространенных фаззеров: V8 и JavaScriptCore. Фаззинг-тестирование проводилось в среднем в течение 7 дней, каждым фаззером было совершено по 2 млн. запусков, что достаточно для демонстрации тенденции изменения скорости обнаружения новых путей.

На рисунке 14 (а)-(б) представлены графики зависимости количества обнаруженных новых путей от времени выполнения в исследованных фаззерах для JavaScript-интерпретаторов V8 и JavaScriptCore соответственно.

---

[3] JavaScript-интерпретатор v8. https://github.com/v8/v8.

[4] JavaScript-интерпретатор JavaScriptCore. https://github.com/WebKit/WebKit.

[5]Shengtuo Hu (h1994st). 2020. Grammar Mutator – AFL++. https://github.com/AFLplusplus/Grammar-Mutator.

[6] Adrian Tiron. 2020. Superion Mutator – AFL++. https://github.com/adrian-rt/superion-mutator.

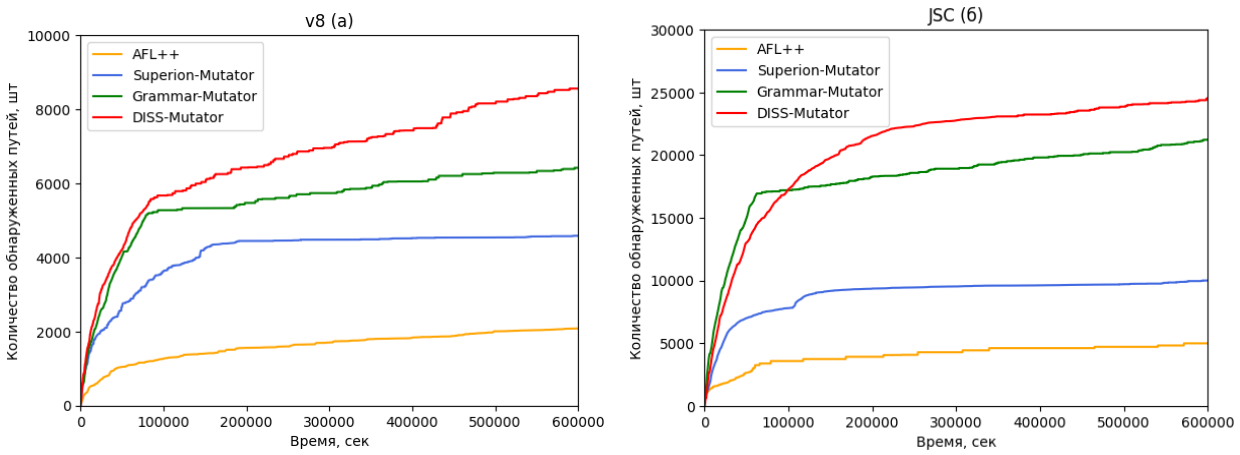

Рисунок 14 – Графики зависимости количества обнаруженных новых путей от времени выполнения в коде интерпретаторов V8 (а) и JavaScriptCore (б)

Графики наглядно демонстрируют превосходство по количеству обнаруженных новых уникальных путей в коде разработанной архитектуры над аналогичными фаззерами. Одной из важных характеристик фаззера является частота единичного запуска. Средняя частота запусков для исследуемых фаззеров составляет: 3,6 зап/сек (AFL++); 7,1 зап/сек (Superion-Mutator); 5,31 зап/сек (Grammar-Mutator) и 3,6 зап/сек (DISS-Mutator). Данный факт мотивирует нас оценить также зависимость количества обнаруженных новых путей от числа единичных запусков фаззера.

На рисунке 15 (а)-(б) представлены графики зависимости количества обнаруженных новых путей от числа итераций в исследованных фаззерах для JavaScript-интерпретаторов V8 и JavaScriptCore соответственно.

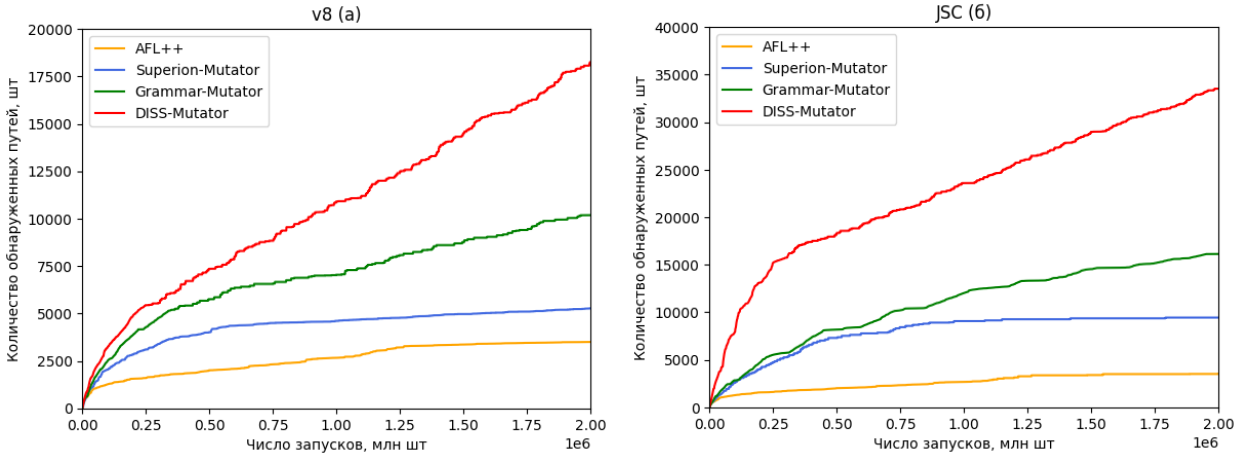

Рисунок 15 – Графики зависимости количества обнаруженных новых путей от числа итераций в коде интерпретаторов V8 (а) и JavaScriptCore (б)

Также в результате проведения экспериментальных исследований разработанной системой фаззинга было обнаружено 589 аварийных завершений протестированных JavaScript-интерпретаторов V8 и JavaScriptCore, что превышает результаты рассмотренных аналогов, а именно: 0, 138 и 196 аварийных завершений обнаруженных фаззером AFL++ без встроенных модулей и с модулями Superion-Mutator и Grammar-Mutator соответственно (Рис.16).

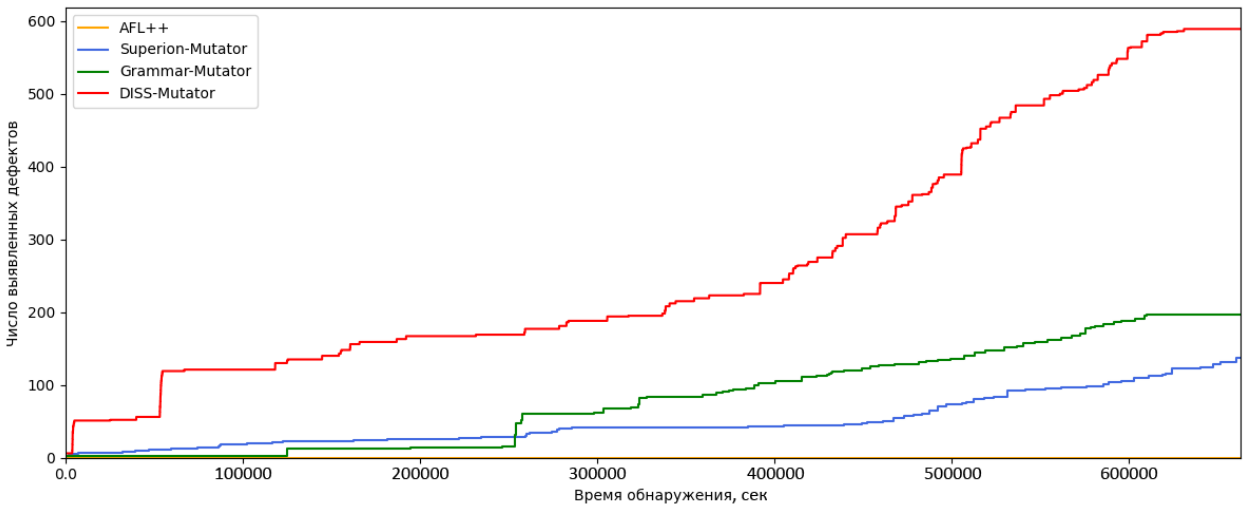

Рисунок 16 – Графики зависимости количества обнаруженных аварийных завершений от времени обнаружения

В таблице 3 представлены количественные результаты сравнения показателя эффективности разработанной системы тестирования и рассмотренных аналогов фаззера AFL++ без встроенных модулей и с модулями Superion-Mutator и Grammar-Mutator соответственно.

Таблица 3 – Сравнение показателей эффективности фаззинг-тестирования JavaScript-интерпретаторов

Фаззер	Показатель эффективности фаззинга по времени $E_{врем}$, путей/ч		Показатель эффективности фаззинга по запускам $E_{зап}$, путей/10 тыс. запусков		Общее количество обнаруженных аварийных завершений
	V8	JSC	V8	JSC	
AFL++	12,94	30,58	18,4	18,81	0
Superion-Mutator	19,26	61,05	26,32	49,17	138
Grammar-Mutator	38,93	127,89	51,66	81,18	196
DISS-Mutator	**51,79**	**144,41**	**97,56**	**181,4**	**589**

Показатель эффективности фаззинга по времени $E_{врем}$ для интерпретатора V8 был **повышен** относительно фаззера AFL++ без встроенных модулей в 4 раза, с модулем Superion-Mutator в 2,7 раза, с модулем Grammar-Mutator в 1,3 раза; для интерпретатора JSC был **повышен** относительно фаззера AFL++ без встроенных модулей в 4,7 раза, с модулем Superion-Mutator в 2,4 раза, с модулем Grammar-Mutator в 1,1 раза; то есть в среднем **в 2,7 раза**.

Показатель эффективности фаззинга по запускам $E_{зап}$ V8 был **повышен** относительно фаззера AFL++ без встроенных модулей в 5,3 раза, с модулем Superion-Mutator в 3,7 раза, с модулем Grammar-Mutator в 1,9 раза; для интерпретатора JSC был **повышен** относительно фаззера AFL++ без встроенных модулей в 9,6 раза, с модулем Superion-Mutator в 3,7 раза, с модулем Grammar-Mutator в 2,2 раза; то есть в среднем **в 4,4 раза**.

Таким образом, экспериментально подтверждено повышение эффективности фаззинг-тестирования JavaScript-интерпретаторов при применении модели генерации входного корпуса, метода мутации сложноструктурированного кода в архитектуре системы генерационного и мутационного фаззинг-тестирования JavaScript-интерпретаторов.

В заключении приведены основные результаты, полученные в ходе выполнения диссертационной работы.

ОСНОВНЫЕ РЕЗУЛЬТАТЫ РАБОТЫ

Предложен новый подход к повышению эффективности фаззинг-тестирования JavaScript-интерпретаторов с целью выявления дефектов безопасности, который состоит из полученных в ходе выполнения работы результатов:

1. Проведен сравнительный анализ современных средств фаззинг-тестирования JavaScript-интерпретаторов, выявлены существующие недостатки и причины их возникновения, связанные с отсутствием общедоступных неизбыточных корпусов входных данных для фаззинг-тестирования, а также отсутствием методов мутации сложноструктурированных данных, учитывающих синтаксис и семантику данных, таких как JavaScript-код. Определены возможные способы их устранения.
2. Разработана модель генерации входных данных для фаззинг-тестирования JavaScript-интерпретаторов на основе рекуррентной нейронной сети, учитывающая степень покрытия кода.
3. Разработан метод мутации сложноструктурированных данных, сохраняющий синтаксис и семантику данных за счет модификации АСД фрагментов JavaScript-кода.
4. Разработана архитектура системы генерационного и мутационного фаззинг-тестирования JavaScript-интерпретаторов и реализованы программные средства, позволяющие повысить эффективность фаззинг-тестирования JavaScript-интерпретаторов в среднем в 3,5 раза.
5. Проведены экспериментальные исследования предложенных решений и оценка их эффективности. В результате показатель эффективности фаззинга по времени $E_{врем}$ для интерпретаторов V8, JSC был **повышен в 2,7 раза**. Показатель эффективности фаззинга по запускам $E_{зап}$ для интерпретаторов V8, JSC был **повышен в 4,4 раза**.

Перспективы дальнейшей разработки темы диссертационного исследования заключаются в применении методов машинного обучения для определения приоритизации мутаций в сложноструктурированных данных. Также перспективным направлением дальнейших исследований является поиск наиболее подходящих нейронных сетей под задачи, возникающие при фаззинг-тестировании сложноструктурированным кодом.

СПИСОК РАБОТ, ОПУБЛИКОВАННЫХ АВТОРОМ ПО ТЕМЕ ДИССЕРТАЦИИ

Публикации в изданиях из перечня ВАК РФ:

1. Козачок, А.В. Обзор исследований по применению методов машинного обучения для повышения эффективности фаззинг-тестирования / А.В. Козачок, В.И. Козачок, **Н.С. Осипова,** Д.В. Пономарев // Вестник Воронежского государственного университета. Серия: Системный анализ и информационные технологии. – 2021. – № 4. – С. 83-106.
2. Козачок, А.В. Подходы к оценке поверхности атаки и фаззингу веб-браузеров / А.В. Козачок, Д.А. Николаев, **Н.С. Ерохина** // Вопросы кибербезопасности. – 2022. – № 3 (49). – С.32-43.
3. Козачок, А.В. Метод генерации семантически корректного кода для фаззинг-тестирования интерпретаторов JavaScript / А.В. Козачок, А.А. Спирин, **Н.С. Ерохина** // Вопросы кибербезопасности. – 2023. – № 5 (57). – С.80-88.
4. **Ерохина, Н.С.** Метод мутации сложноструктурированных входных данных при фаззинг-тестировании JavaScript интерпретаторов // Труды ИСП РАН – 2023. –№ 5 (35). – С. 55-66.

5. Козачок, А.В. Способ обнаружения программных дефектов в JavaScript-интерпретаторах методом фаззинг-тестирования / А.В. Козачок, Д.А. Николаев, **Н.С. Ерохина** // Вопросы кибербезопасности. – 2024. – №. 2 (60). – С. 74-80.

Свидетельства о регистрации программы для ЭВМ:

1. Программа для ЭВМ 2023664761 Российская Федерация. Программный модуль генерации семантически корректного JavaScript-кода для фаззинг-тестирования JavaScript интерпретаторов веб-браузеров [Текст] / Козачок А.В., **Ерохина Н.С.** – № 2023664761 заявл. 30.06.2023 опубл. 07.07.2023.
2. Программа для ЭВМ 2024613523 Российская Федерация. Программный модуль мутации сложноструктурированных данных для фаззинг-тестирования JavaScript-интерпретаторов [Текст] / **Ерохина Н.С.** – № 2024613523 заявл. 12.02.2024 опубл. 13.02.2024.

Наиболее значимые публикации в других изданиях:

1. Ерохина, Н.С. Применение методов машинного обучения при проведении фаззинг-тестирования / **Н.С. Ерохина** // Безопасные информационные технологии: XI Международная научно-техническая конференция. – 2021. – с. 263-269. – URL: https://elibrary.ru/item.asp?id=46414450 (дата обр. 20.03.2022).
2. Козачок, А.В. О некоторых подходах к оценке поверхности атаки веб-браузеров / А.В. Козачок, **Н.С. Ерохина**, Д.А. Николаев // Методы и технические средства обеспечения безопасности информации. XXXI Всероссийская научно-практическая конференция. – 2022. – с. 72-74. – URL: https://ic.spbstu.ru/userfiles/files/tezisy_mitsobi_2022.pdf#page=72 (дата обр. 26.09.2023).
3. Козачок, А.В. Генерация семантически корректного JavaScript-кода для фаззинг-тестирования движков браузеров / А.В. Козачок, **Н.С. Ерохина** // Методы и технические средства обеспечения безопасности информации. XXXII Всероссийская научно-практическая конференция. – 2023. – с. 78-80. – URL: https://elibrary.ru/item.asp?id=54244668 (дата обр. 02.02.2023).
4. Козачок, А.В. Подходы к повышению эффективности мутаций сложноструктурированных данных при фаззинг-тестировании JavaScript интерпретаторов / А.В. Козачок, **Н.С. Ерохина** // Безопасные информационные технологии: XII Международная научно-техническая конференция. – 2023. – с. 75-79. – URL: https://npo-echelon.ru/doc/BIT_2023.pdf (дата обр. 02.02.2023).
5. Козачок, А.В. Способ фаззинг-тестирования интерпретаторов JavaScript на основе AST мутаций / А.В. Козачок, **Н.С. Ерохина** // Национальная научно-практическая конференция "Фундаментальные, поисковые, прикладные исследования и инновационные проекты" – 2023. – с. 55-59. – URL: https://www.elibrary.ru/download/elibrary_65243418_60376107.pdf (дата обр. 15.01.24).
6. Козачок, А.В. Метод генерации семантически корректного кода для фаззинг-тестирования JavaScript-интерпретаторов на основе мутации АСД фрагментов кода / А.В. Козачок, **Н.С. Ерохина** // Методы и технические средства обеспечения безопасности информации. XXXIII Всероссийская научно-практическая конференция. – 2024. – С. 159-161.