Федеральное государственное бюджетное образовательное учреждение высшего образования «Московский государственный университет имени М.В.Ломоносова»

На правах рукописи

Сигалов Даниил Алексеевич

Методы выявления поверхности атаки веб-приложений при помощи анализа клиентского JavaScript-кода

Специальность 2.3.5 — «Математическое и программное обеспечение вычислительных систем, комплексов и компьютерных сетей»

Диссертация на соискание учёной степени кандидата технических наук

Научный руководитель:
кандидат физико-математических наук
Гамаюнов Денис Юрьевич

Москва — 2024

Оглавление

Стр.

Стр.

Введение

В настоящее время операционная деятельность организаций, как коммерческих компаний, так и государственных учреждений, существенно зависит от интернета и интернет-приложений. Банковские услуги, интернет-магазины, средства организации рабочего процесса, социальные сети, сервисы отправки показаний счётчиков воды и электричества — в значительной степени реализуются с помощью веб-приложений. При этом, по данным компании Verizon (одна из крупнейших телекоммуникационных компаний США), до 80% инцидентов безопасности в 2023 году, приведших к утечке данных пользователей или коммерческой тайны, были связаны с атаками на уязвимые веб-приложения [1]. На этом фоне чрезвычайно важным является поддержание безопасности веб-приложений, для чего необходимо своевременное обнаружение и устранение уязвимостей в них. Крайне опасными являются, например, уязвимости таких классов, как “SQL-инъекция”, “небезопасная загрузка файлов”, “Server-side template injection” и “Shell injection”. SQL-инъекции могут позволить атакующему получить полный доступ к базе данных приложения, остальные перечисленные уязвимости могут дать возможность выполнить произвольный код на атакуемом сервере. Эксплуатация этих уязвимостей может повлечь кражу пользовательских данных, потерю данных, сбои в работе сервисов.

В данной работе рассматривается задача автоматического обнаружения серверных уязвимостей в веб-приложениях в модели “черного ящика”, когда для анализа доступна только клиентская часть веб-приложения и интерфейсные точки взаимодействия на серверной стороне веб-приложения. Поиск уязвимостей в этой модели востребован, поскольку такие условия близки к условиям, в которых действует реальный атакующий. Кроме того, предоставление исходного кода приложения для анализа может быть нежелательно из соображений конфиденциальности. В модели “черного ящика” процесс поиска серверных уязвимостей обычно состоит из трех этапов [2]:

1. Выявление поверхности атаки — поиск *серверных входных точек* (их также называют доступными функциями веб-приложения, точками входа в приложение или API-вызовами).
2. Вызов найденных функций приложения с различными значениями параметров.

3. Анализ результатов выполнения действий.

Данная работа посвящена автоматизации выявления поверхности атаки. Одним из главных источников информации о серверных входных точках является клиентская часть веб-приложения — то есть его пользовательский интерфейс.

Выявление набора серверных входных точек на основе клиентской части веб-приложения сводят к выявлению набора запросов, которые могут быть отправлены клиентской стороной на сервер. Множество доступных функций сервера полностью определяется кодом серверной части приложения и его конфигурацией. При анализе веб-приложения как черного ящика эта информация недоступна, поэтому восстановить это множество полно и точно не представляется возможным, но можно решать задачу построения аппроксимирующего множества. Выдвигается гипотеза о том, что отправляемые с клиентской части запросы соответствуют серверным входным точкам (код серверной и клиентской части согласован).

Выявление набора отправляемых на сервер запросов тривиально для статических интерфейсов (построенных с использованием только языка HTML). Семантика элементов HTML-разметки и их реакция на действия пользователя всегда фиксированы. Однако для динамических интерфейсов (построенных с использованием JavaScript-кода) задача намного сложнее. JavaScript — это высокоуровневый язык программирования, полный по Тьюрингу. На HTML-странице JavaScript-код может инициировать отправку запроса на сервер, URL-адрес и другие части которого сформированы самим кодом. По статистике [3], JavaScript используется на 98,8% веб-сайтов.

Исследования, посвящённые автоматическому выявлению набора серверных входных точек посредством анализа клиентской части веб-приложения, включая анализ клиентского JavaScript-кода, ведутся рядом научных групп. Недавними являются публикации исследователей из австралийского отделения Oracle Labs, в которых были представлены средства BackREST [4] и Gelato [5], а также статьи группы из австрийского исследовательского центра SBA Research, в которых описаны инструменты XIEv [6] и CHIEv [7]. Работы по этой теме публиковались совместной группой из Университета Британской Колумбии и Делфтского технического университета [8]. Все предложенные на данный момент методы используют динамический анализ, а именно — *динамический краулинг*, автоматизированное взаимодействие с элементами интерфейса на веб-странице

с помощью управляемого браузера (симуляция пользовательских действий) и запись всех отправляемых со страницы запросов на сервер.

У этого метода есть ограничения. В некоторых случаях пользовательский интерфейс слишком сложен для того, чтобы осуществить все возможные действия с ним. Перебор всех комбинаций возможных действий может требовать слишком большого количества времени. Более того, существуют случаи, когда JavaScript-код, отправляющий запрос к определённой серверной входной точке, вообще невозможно вызвать действиями в интерфейсе — по сути, это недостижимый код. Такой код, тем не менее, представляет интерес при поиске уязвимостей: часть серверного кода, к которому делается такое обращение, может работать.

Таким образом, актуальной является задача разработки метода выявления серверных входных точек, основанного на *статическом* анализе клиентского JavaScript-кода.

Целью данной работы является разработка метода автоматического выявления поверхности атаки веб-приложений с динамической клиентской частью, реализованной на языке программирования JavaScript.

Для достижения поставленной цели необходимо было решить следующие **задачи**:

1. Провести исследование особенностей реального JavaScript-кода, влияющих на возможность его анализа с целью обнаружения информации о серверных входных точках. На основе результатов проведённого исследования сформулировать требования к инструментам построения поверхности атаки на основе статического анализа клиентского JavaScript-кода, а также составить эталонный набор веб-страниц (бенчмарк), позволяющий оценивать эффективность методов извлечения информации о серверных входных точках из клиентского кода.
2. Разработать методику поиска уязвимостей в режиме “чёрного ящика”, позволяющую обрабатывать как легко доступные из пользовательского интерфейса серверные входные точки, так и труднодоступные.
3. Разработать и реализовать специализированный метод статического анализа клиентского кода веб-приложения для обнаружения серверных входных точек, учитывающий выявленные в ходе проведённого исследования особенности реального кода. Провести эксперименты с разработанной реализацией на реальных данных.

Научная новизна: В работе получены следующие результаты, обладающие научной новизной.

1. Проведено исследование реального кода JavaScript-приложений и выделены его наиболее существенные особенности с точки зрения статического анализа для поиска входных точек: использование упаковщиков модулей, использование непрямого объявления класса, обращение к полю объекта по вычисленному имени и другие. Учет выявленных особенностей позволил сузить задачу статического анализа и предложить более эффективный метод его выполнения.
2. Предложена методика поиска уязвимостей веб-приложений в модели “чёрного ящика”. Методика отличается от существующих повышением покрытия серверных входных точек, обращения к которым сложно вызвать через взаимодействие с пользовательским интерфейсом, за счет применения статического анализа и автоматизированного перебора имён параметров запроса, а также применением анализа кода клиентской стороны веб-приложения для обнаружения уязвимостей.
3. Предложен метод обнаружения информации о серверных входных точках для задачи анализа защищенности приложений методом “черного ящика” с помощью статического анализа клиентского кода. Для типичной страницы реального веб-приложения разработанный алгоритм анализа позволяет решить задачу обнаружения входных точек за несколько минут, что делает его пригодным для практического использования. Это достигается за счёт ограничения числа проходов по коду и добавления встроенной поддержки популярных JavaScript-библиотек, отправляющих запросы на сервер (позволяет не анализировать собственный код библиотек). Разработанный алгоритм анализирует весь код страницы, включая недостижимый код.

Практическая значимость работы состоит в возможности применения её результатов в анализе защищенности реальных систем: метод позволяет более полно выявлять элементы поверхности атаки — анализируемые на наличие уязвимостей серверные входные точки, тем самым увеличивая количество обнаруживаемых уязвимостей. Реализация предложенного метода была внедрена как часть системы автоматического поиска уязвимостей в веб-приложениях на основе обработки больших данных, разработанной в Центре компетенций НТИ по технологиям хранения и анализа больших данных МГУ (ЦХАБД МГУ) [9].

Также разработанный метод внедрён в систему автоматизированного поиска уязвимостей веб-приложений SolidPoint DAST, разрабатываемую компанией ООО «СолидСофт». Акты о внедрении приведены в приложении А к настоящей работе.

Методология и методы исследования.

В работе использовались методы статического анализа программ и теории компиляции.

Основные положения, выносимые на защиту:

1. Требования к инструментам построения поверхности атаки на основе статического анализа клиентского JavaScript-кода, которые сформулированы на основе результатов исследования особенностей реального JavaScript-кода, влияющих на возможность его анализа для поиска серверных входных точек. Эталонный набор веб-страниц (бенчмарк), также созданный в результате этих исследований, который может быть использован для автоматизированной оценки эффективности методов извлечения информации о серверных входных точках.
2. Методика обнаружения уязвимостей в веб-приложениях, которая повышает полноту выявления серверных входных точек за счет статического анализа клиентской части приложения. Методика апробирована в реальных системах автоматизированного анализа защищенности приложений.
3. Специализированный метод анализа клиентского кода веб-приложения для обнаружения серверных входных точек и последующего поиска уязвимостей в них. Метод разработан с учётом особенностей реального кода, отправляющего запросы на сервер, которые были выявлены в результате исследования. Проведена апробация реализованного метода на составленном эталонном наборе страниц, а также на наборе приложений, использовавшихся в других работах. Кроме того, проведены эксперименты на сайтах в сети Интернет, в результате которых обнаружены реальные уязвимости.

Апробация работы. Основные результаты работы докладывались на:

– Конференции Positive Hack Days 11, Москва, 18-19 мая 2022 г.
– Конференции OFFZONE 2022, Москва, 25-26 августа 2022 г.
– Конференции Ломоносовские чтения — 2023, Москва, 4-14 апреля 2023 г.
– Семинаре Workshop on Attacks and Software Protection 2023, проведённом в рамках конференции ESORICS 2023, Гаага, Нидерланды, 29 сентября 2023 г.

– Конференции Positive Hack Days Fest 2, Москва, 23-26 мая 2024 г.

Личный вклад. Все представленные в диссертации результаты получены лично автором.

Публикации. Основные результаты по теме диссертации изложены в 6 печатных изданиях, 5 из которых [13—17] изданы в журналах, рекомендованных ВАК, 4 — в периодических научных журналах, индексируемых Web of Science и Scopus [13; 16—18]. Зарегистрирована 1 программа для ЭВМ [9].

В работах [13; 16—18] все результаты были получены лично автором.

В работе [14] автором был предложен алгоритм извлечения кода из JavaScript-комментариев, кроме того, автору принадлежит разработка и реализация базового алгоритма анализа, с которым был интегрирован модуль извлечения кода из комментариев, а также методическое обеспечение экспериментов.

В работе [15] автору принадлежит постановка задачи, методология исследования, а также проведение части экспериментов.

Объем и структура работы. Диссертация состоит из введения, 4 глав, заключения и 1 приложения. Полный объём диссертации составляет 133 страницы, включая 6 рисунков и 3 таблицы. Список литературы содержит 89 наименований.

Глава 1. Задача выявления поверхности атаки веб-приложения при поиске уязвимостей методом “чёрного ящика”

Необходимым первым шагом процесса поиска серверных уязвимостей методом “чёрного ящика” является выявление того, каким образом данные могут быть поданы на вход серверной части. Взаимодействие с сервером происходит по протоколу HTTP. В данной работе рассматривается наиболее частый способ передачи данных на сервер — передача в HTTP-запросах. Взаимодействие с сервером по протоколу WebSocket выходит за рамки данной работы.

Как и другие виды программ, веб-приложения принимают данные в определённом формате, который определяется требованиями протокола HTTP и правилами, специфичными для каждого веб-приложения. *Серверной операцией* будем называть часть кода сервера, выполняемую при обработке определённого запроса. Говоря, что сервер *выполняет серверную операцию*, будем иметь ввиду, что сервер выполняет соответствующую часть своего кода. Как правило, сервер поддерживает несколько различных операций, выполнение которых вызывается HTTP-запросами. Будем говорить, что запрос *вызывает серверную операцию*, если при его получении и обработке сервер выполняет эту серверную операцию. Соответственно, если 2 разных запроса таковы, что при их получении сервер выполняет один и тот же код, они вызывают одну и ту же операцию. Если код, выполняемый при обработке 2 разных запросов, отличается, то эти запросы вызывают разные серверные операции. *Серверная входная точка*, соответствующая серверной операции — это шаблон HTTP-запроса, описывающий множество всех запросов, вызывающих эту серверную операцию.

То, какая именно операция вызывается, определяется некоторой частью запроса. У запроса, обращающегося к определённой операции, будем называть *фиксированной* ту его часть, которая была учтена внутренними правилами маршрутизации при выборе именно этой операции (при изменении этой части запрос перестал бы вызывать выполнение этой операции). Часто к фиксированной части относится URL-путь и метод запроса. Кроме того, в запросе могут присутствовать значения параметров — их совокупность будем называть *вариативной* частью запроса. Запросы с различными наборами значений параметров (наборами данных в вариативной части) будут вызывать выполнение той же самой серверной операции при условии совпадения фиксированной части. К вариативной части нередко

относятся значения query-параметров, тело запроса, значения заголовков. Серверная входная точка специфицирует то, какие части HTTP-запросов, относящихся к этой входной точке, фиксированы, а какие могут варьироваться. Обращаясь к различным серверным входным точкам, можно вызывать выполнение различных операций.

На листинге 1.1 приведены два запроса, относящихся к одной серверной входной точке, но имеющие различающиеся данные в вариативной части. На листинге 1.2 приведён пример запроса к другой серверной входной точке — его фиксированная часть отличается от фиксированной части первых двух запросов. Фиксированная часть на этих листингах обозначена чёрным цветом, а вариативная — синим.

Листинг 1.1: Два запроса, обращающихся к одной и той же серверной входной точке, отвечающей за вход в личный кабинет (вариативная часть выделена синим)

```
POST /user/login HTTP/1.1
Host: website.com
Content-Type: application/x-www-form-urlencoded

login=alex&password=secret123
```

```
POST /user/login HTTP/1.1
Host: website.com
Content-Type: application/x-www-form-urlencoded

login=peter&password=hunter2
```

Листинг 1.2: Запрос, обращающийся к серверной входной точке, отвечающей за поиск по сайту (вариативная часть выделена синим)

```
POST /site/search HTTP/1.1
Host: website.com
Content-Type: application/x-www-form-urlencoded

query=kittens
```

Имея полный набор серверных входных точек, можно выполнить все серверные операции, выполнение которых можно инициировать HTTP-запросом к

серверу. Таким образом, набор серверных входных точек полностью определяет пространство атаки для исследуемого веб-приложения.

1.1 Задача выявления серверных входных точек

Как уже было сказано, выявить набор серверных входных точек полно и точно при анализе веб-приложения как черного ящика невозможно. Для этого потребовался бы доступ к серверному коду, а также к серверной конфигурации. Но можно решать задачу построения аппроксимирующего множества.

Различные операции реализуются разным серверным кодом, при этом, при поиске уязвимостей необходимо обеспечивать как можно большее покрытие кода. Поэтому при поиске серверных входных точек наиболее важным критерием качества является *полнота*.

Проводить поиск серверных входных точек можно следующими способами:

– Извлечение информации о них из клиентской части приложения.
– Выявление серверного ПО (набора используемых программ и их версий) по сигнатурам и использование заранее известной информации об этом ПО.
– Активный управляемый перебор запросов к серверной стороне приложения (фаззинг) и анализ ответов для обнаружения входных точек, недоступных из интерфейса — примером такого перебора является поиск URL-путей по словарю, так называемый дирбастинг.

Дирбастинг или фаззинг в общем случае не позволяют найти имена параметров HTTP-запроса и возможные диапазоны их значений [19]. Эти методы, как и выявление серверного ПО по сигнатурам, показывают низкое качество работы при анализе нестандартного ПО, которое было сделано специально для анализируемого приложения и не было известно ранее. Извлечение информации о серверных входных точках из клиентской части веб-приложения необходимо для получения приемлемого качества поиска уязвимостей в режиме “чёрного ящика”, что отмечается авторами работ, посвящённых автоматическому выявлению поверхности атаки веб-приложений [7; 20; 21].

Выявление набора серверных входных точек на основе клиентской части веб-приложения сводят к выявлению набора запросов, которые могут быть от-

правлены клиентской стороной на сервер. Выдвигается гипотеза о том, что код серверной и клиентской части согласован, и отправляемые с клиентской части запросы соответствуют серверным входным точкам. При этом, как уже было отмечено, разные запросы, отправляемые клиентской частью, могут, в общем случае, соответствовать одной и той же серверной входной точке.

Как уже упоминалось, выявления набора отправляемых на сервер запросов решается тривиально для статических интерфейсов (построенных на чистом HTML), поскольку семантика элементов HTML-разметки и их реакция на действия пользователя всегда фиксированы. Однако, задача намного сложнее для динамических интерфейсов (построенных с использованием JavaScript-кода). На странице JavaScript-код может инициировать отправку запроса на сервер, URL-адрес и другие части которого сформированы самим кодом. При этом, использование JavaScript-кода на страницах веб-сайтов, по данным статистики, чрезвычайно распространено: на данный момент его используют 98.8% сайтов [3]. Именно поиску запросов, отправляемых JavaScript-кодом, уделено основное внимание в настоящей работе.

Предельным случаем динамического интерфейса в настоящее время являются так называемые *single-page applications* (SPA), в которых при первом обращении к веб-приложению один раз загружается HTML страница с JavaScript программой, и в дальнейшем все запросы с клиента порождаются только в результате выполнения этой программы.

Практически все средства автоматизированного поиска уязвимостей для выявления отправляемых клиентским JavaScript-кодом HTTP-запросов используют технику, известную как *динамический краулинг* — автоматизированное взаимодействие с элементами интерфейса на веб-странице с помощью управляемого браузера (симуляция пользовательских действий) и запись всех отправляемых при этом со страницы на сервер запросов [7; 8; 20]. У этого метода есть ограничения. В некоторых случаях пользовательский интерфейс слишком сложен для того, чтобы осуществить все возможные действия с ним. Перебор всех комбинаций возможных действий может требовать слишком большого количества времени. Более того, существуют случаи, когда JavaScript-код, отправляющий запрос к определённой серверной входной точке, вообще невозможно вызвать действиями в интерфейсе — по сути, это недостижимый код. Такой код, тем не менее, представляет интерес при поиске уязвимостей: часть серверного

кода, к которому делается такое обращение, может работать. Вот некоторые примеры таких ситуаций:

1. **Входные точки, относящиеся к авторизованной зоне сайта.** Примером является административная панель управления веб-приложением. Анонимный пользователь не увидит ссылку на административную панель или форму смены пароля администратора в своем интерфейсе (по крайней мере, пока не пройдет аутентификацию как администратор). При этом, возможно такое, что клиентский JavaScript-код, соответствующий административной панели, будет доступен анонимному пользователю [19]. Следует отметить, что такие входные точки могут содержать уязвимости. В случае ошибки проверки авторизации взаимодействие с ними может быть возможно даже без наличия корректного авторизационного токена.
2. **Функции веб-приложения, которые в данный момент удаляют или, напротив, ещё не до конца добавили.** Это может быть старый вариант реализации поиска по сайту, который уже убран из интерфейса, но который всё ещё поддерживается сервером — к примеру, потому что соответствующее обновление серверного кода не было сделано. Реальным примером, найденным в ходе исследования, является плагин для системы управления содержимым WordPress под названием “Ajax Search Lite” [22]. После установки и активации этого плагина его серверные входные точки начинают работать. Соответствующий JavaScript-код, обращающийся к этим входным точкам, также будет добавлен на каждую страницу веб-приложения. При этом, поисковую строку нужно добавлять на страницы вручную — если этого не сделать, то никакого способа обратиться к функциональности поиска из интерфейса не будет. По данным реестра плагинов WordPress, этот плагин установлен на более чем 70000 сайтов.

Есть и другие примеры: отладочные функции, доступные только разработчикам, отсутствие на сайте данных, необходимых для функционирования некоторых частей интерфейса (например, кнопка “оценить комментарий” не появляется на страницах форума, где ещё нет ни одного комментария).

Вследствие описанных ограничений динамического краулинга актуальным представляется разработка метода извлечения информации о серверных входных точках из клиентского кода, основанном на *статическом* анализе. При статиче-

ском анализе задача сводится к определению возможных аргументов обращений к программным интерфейсам отправки запросов на сервер. Статические методы способны анализировать весь код программы, независимо от того, достижим ли он, и насколько сложно добиться достижения того или иного участка кода при реальном запуске. Таким образом, статический анализ смог бы выявлять в коде обращения к входным точкам, вызов которых посредством действий в интерфейсе сложен или невозможен. Кроме того, при использовании статического анализа возможна разработка алгоритма, который будет эффективно обрабатывать большие объёмы сложного кода. В частности, при наличии в коде большого количества ветвлений появляется большое число потенциальных путей выполнения. Динамическому анализу для достижения полноты пришлось бы выполнять код множество раз, проходя по каждому из этих путей по отдельности, в то же время, при применении методов статического анализа возможно анализировать множество путей одновременно. Следует отметить, что применимость статического анализатора в реальной практике разработки будет зависеть от его времени работы. Время работы автоматизированного средства поиска уязвимостей на одном веб-приложении не должно превышать нескольких часов. Это связано с тем, что системы поиска уязвимостей используются в составе цикла автоматического тестирования приложения, выполняемого при внесении изменений разработчиками (цикла “непрерывной интеграции”). Кроме того, анализаторы сканируют целиком вычислительные сети крупных организаций, в которых количество приложений может доходить до нескольких тысяч, и необходимо, чтобы сканирование завершалось быстрее, чем разработчики выпустят новую версию приложений. Современные веб-приложения содержат десятки, часто — до сотен страниц, на каждой из которых, вообще говоря, свой JavaScript-код. Поэтому для реальной применимости время анализа одной веб-страницы должно быть около нескольких минут.

1.2 Обзор существующих методов

Извлечение информации о серверных входных точках из клиентского кода веб-приложения сводится к анализу этого кода. Как правило, методы анализа кода разделяют на динамические и статические.

1.2.1 Динамический анализ

Динамический краулинг

Как уже упоминалось выше, по сути, все нетривиальные методы определения набора серверных входных точек на основе клиентского JavaScript-кода, применяемые в средствах поиска уязвимостей, основаны на динамическом краулинге. Который, по своей природе, является одним из вариантов динамического анализа. Популярность этого подхода естественна, так как код на языке JavaScript крайне сложен для статического анализа.

Существует целый ряд работ, предлагающих применять динамический краулинг для решения рассматриваемой задачи. В 2012 году появились работы, предлагающие для решения задачи краулеры Crawljax [8] и Enemy of the State [23], первый из которых поддерживается разработчиками до сих пор. Далее появилось ещё несколько работ, предлагающих инструменты того же класса: FEEDEX [24], jÄk [25] и XIEv [6]. Кроме того, в работах [26; 27] были предложены не просто динамические краулеры, а полноценные сканеры для поиска уязвимостей, содержавшие в себе динамические краулеры как одну из встроенных частей.

Наиболее современными работами в этой области являются работы, предлагающие средства “Black Widow” [28], CHIEv [7] и Gelato [5].

Все эти упомянутые средства используют динамический краулинг в соответствии с его определением, данным выше: во всех них производится автоматизированное взаимодействие с элементами интерфейса приложения с помощью управляемого веб-браузера. Это взаимодействие симулирует пользовательские действия. Запросы, отправляемые на сервер во время этого взаимодействия, отслеживаются и записываются. Для всех упомянутых средств ключевой метрикой качества является покрытие серверного кода. Большинство из них строит модель пользовательского интерфейса приложения, состоящую из состояний интерфейса, и, как правило, основанную на наблюдаемых состояниях клиентской части (т. е. наборе видимых элементов, состоянии DOM-дерева). В процессе анализа средства, как правило, преследуют цель посетить все возможные состояния.

Динамический краулер, встроенный в сканер Htcap [27], особенно интересен тем, что он производит *изолированный* анализ каждой из страниц приложения: если очередное действие в интерфейсе приводит к переходу на другую страницу (в простейшем случае это может быть клик по ссылке), Htcap отменяет этот переход, помещая URL-адрес, на который делался переход, в очередь. Далее анализ оригинальной страницы продолжается до тех пор, пока на ней не будут произведены все запланированные действия. После выполнения всех запланированных действий Htcap берёт из очереди следующий URL-адрес, открывает страницу, на которую он ведёт, также изолированно анализирует её. И так далее.

Средство Gelato применяет статический анализ для построения графа вызовов для JavaScript-кода страницы. Однако, этот граф применяется для того, чтобы подсказать краулеру наиболее оптимальную стратегию взаимодействия с элементами и, помимо построения графа вызовов, весь остальной алгоритм Gelato в целом основан на автоматизированном взаимодействии с пользовательским интерфейсом, то есть на динамическом краулинге.

Средство BackREST [4] является дальнейшим развитием Gelato, однако, в этом средстве применяется инструментация серверного кода, которая используется для улучшения метода извлечения информации о серверных входных точках. Таким образом, работа, предлагающая BackREST выходит за рамки модели “чёрного ящика”.

Все упомянутые в этом разделе методы выявляют отправляемые клиентской частью на сервер запросы посредством взаимодействий с пользовательским интерфейсом и отслеживания запросов, отправленных вследствие этих взаимодействий. Ограничением таких подходов является то, что они упускают запросы, которые сложно или невозможно вызвать действиями с элементами пользовательского интерфейса.

Анализ, меняющий семантику выполняемого кода

Все методы динамического анализа, упомянутые в предыдущем подразделе, выполняют анализируемый код “честно”: код выполняется таким же образом, как выполнялся бы в реальной среде выполнения. Даже если какие-то из описанных методов применяют инструментацию кода или среды выполнения, эта

инструментация используется только для сбор информации, и не влияет на ход выполнения программ. В то же время, существует несколько работ, в которых предлагается применить для анализа клиентского JavaScript-кода технику, известную как “*forced execution*” [29; 30]. В обеих работах инструментация среды выполнения (точнее, инструментация браузера Chrome) используется для изменения семантики, заложенной в интерпретатор JavaScript-кода. Применяемая инструментация позволяет описанным в работах анализаторам вызывать функции, вызовы которых в коде отсутствуют, пропускать вызовы функций, вызов которых нежелателен, произвольным образом менять выбираемую ветвь операторов ветвления (заставляя управление зайти в ветви, которые иначе не были бы посещены), синтезировать новые переменные, которые не были созданы анализируемым кодом.

Подобные анализаторы способны покрывать “мёртвый код”, что, как было сказано выше, проблемно для динамического краулинга. По мнению автора настоящей работы, применение техники “forced execution” для решения задачи извлечения информации о серверных входных точках из клиентского JavaScript-кода является одним из перспективных направлений для развития исследований. Тем не менее, реализация подобного анализатора нетривиальна. Код описанного в работе [29] анализатора JSForce не опубликован. Код средства, реализованного в работе [30], опубликован, но он в своём текущем виде не решает задачу обнаружения отправляемых на сервер запросов, добавление в него такой функциональности может быть затруднительным. Как упоминается в работе [25], модификация JavaScript-интерпретатора современного браузера это трудоёмкий процесс, кроме того, полученный в результате анализатор будет жёстко привязан к конкретной версии определённого интерпретатора. Стабильность работы таких модифицированных интерпретаторов, как правило, достаточно низка — интерпретатор JavaScript-кода не рассчитан на подобные изменения, и после их внесения повышается вероятность ошибок его работы, приводящих к аварийному завершению.

Помимо того, что существуют сложности с реализацией описываемой техники, не для всех случаев понятно, как она могла бы помочь в решении задачи обнаружения отправляемых на сервер запросов из недостижимого кода. К примеру, при анализе кода класса, который не используется в программе, такому анализатору нужно было бы как-то определить правильный порядок вызова ме-

тодов класса и, более того, вызывать их с правильным объектом `this` (который должен быть общим между ними).

1.2.2 Статический анализ

В ходе обзора научной литературы не было найдено работ, где статический анализ кода был бы предложен для обнаружения серверных входных точек в составе сканера безопасности веб-приложений. Современные сканеры безопасности, применяемые в индустрии, также не обладают механизмом выявления серверных входных точек, основанном на полноценном статическом анализаторе клиентского JavaScript-кода. Единственный метод извлечения информации об отправляемых на сервер запросов из клиентского JavaScript-кода, используемый в существующих сканерах, который можно отнести к статическому анализу, это поиск строк, похожих на URL-адреса, с помощью регулярных выражений. Очевидно, этот метод не даёт удовлетворительного результата в большинстве случаев. И сработает только в простейшем случае, когда у URL-адреса есть специфический, легко узнаваемый префикс, запрос отправляется с методом `GET`, все параметры запроса константны и явно указаны в URL-адресе, отсутствует тело запроса и дополнительные HTTP-заголовки.

Одной из вероятных причин отсутствия решения, основанного на полноценном статическом анализе, является то, что современный JavaScript-код плохо поддаётся статическому анализу — это отмечается исследователями в существующих публикациях [10; 11]. Это связано как с особенностями языка, так и с существующими практиками написания кода на нём. Код на JavaScript обладает высокой динамичностью. К примеру, косвенные вызовы в JavaScript-коде встречаются намного чаще чем, скажем, в C, Java или Go.

На данный момент к наиболее известным в литературе статическим анализатором JavaScript-кода относятся WALA [31], SAFE [32] и TAJS [33]. Все эти анализаторы анализируют код консервативно, сохраняя свойства корректности. Это делает их неприменимыми для анализа реальных страниц современных веб-приложений в сети Интернет [11; 12; 34; 35]. Даже чрезвычайно распространённая библиотека jQuery (присутствующая на страницах 77% сайтов в сети

Интернет [36]) не поддаётся анализу этими средствами, что отмечается самими авторами [12].

Кроме того, ни один из этих инструментов в своём текущем виде не определяет отправляемые из кода на сервер запросы и не анализирует мёртвый код.

Методами, наиболее схожими с предлагаемым настоящей работе методом, являются методы, описанные в работах [37] и [38]. В обеих работах описываются алгоритмы извлечения информации об отправляемых на сервер запросах из клиентского JavaScript-кода с помощью статического анализа. Однако, применяются эти алгоритмы не для составления списка серверных входных точек для дальнейшего поиска в них уязвимостей, а для других целей. К сожалению, реализации этих алгоритмов не являются публично доступными. Кроме того, авторы [37] не приводят в работе описания разработанного алгоритма, а лишь отмечают, что он основан на методе статического анализа k-CFA. Алгоритм, описанный в работе [38], работает на уровне отдельных файлов, каждый из которых анализируется независимо. Он не отслеживает потоки данных между разными файлами. В реальном коде части запроса очень часто задаются в разных файлах. Частой является ситуация, когда префикс URL-адреса задан в конфигурационном объекте в одном файле, а отправка запроса делается в другом. Поэтому, следствием такого ограничения является то, что часто не будут найдены критически важные части запросов. Ещё одним недостатком метода [38] является то, что в нём используется метод построение графа вызовов, обладающий свойством высокой масштабируемости, но при этом весьма низкой точностью, который отождествляет все методы объектов с одинаковым названием и количеством аргументов. Из этого можно сделать вывод что, хотя точность метода [38] достаточна для решаемой в той работе задачи (проверки того, соответствуют ли спецификациям запросы, отправляемые из кода), эта точность слишком низка для применения для определения серверных входных точек для последующего использования сканером уязвимостей.

1.2.3 Сравнительный анализ методов

Критерии сравнения

Для сравнительного анализа методы были проверены на соответствие требованиям к инструментам построения поверхности атаки на основе статического анализа, составленным в рамках данной работы на основе результатов исследования особенностей реального JavaScript-кода, и сформулированным в главе 2 (в разделе 2.5). Требования состоят из шести пунктов:

1. Поддержка упаковщиков модулей.
2. Поддержка классов (анализ кода, использующего классы).
3. Поддержка строковых операции языка JavaScript, вычисление результатов этих операций.
4. Поддержка анализа кода, содержащего косвенные вызовы и передачу функциональных значений между переменными программы и полями объектов и массивов. Анализатор не обязан полностью точно анализировать косвенные вызовы, но он должен выполнять анализ программ, содержащих такие вызовы, за приемлемое время и при приемлемом потреблении вычислительных ресурсов.
5. Поддержка обращений к полям объектов и массивов вычисленному имени.
6. Поддержка анализа программ, использующих библиотеки, сложные для статического анализа, включая библиотеку jQuery. Анализатор не обязан полностью точно анализировать код таких библиотек или моделировать все их особенности, но он должен выполнять анализ кода, содержащего библиотеки, за приемлемое время и при приемлемом потреблении вычислительных ресурсов.

Помимо этих требований, были также дополнительно использованы следующие критерии:

– Анализ недостижимого кода — как уже говорилось, для повышения покрытия обнаруживаемых серверных входных точек требуется анализировать код, выполнение которого вызвать при нормальном выполнении программы невозможно.

– Отсутствие привязки к конкретной версии браузера — некоторые методы выявления поверхности атаки используют в своём составе реальный веб-браузер. При этом, часть методов переносима на новые версии используемого браузера, другая часть жёстко привязана к конкретной версии. Такая привязка нежелательна, поскольку старая версия браузера не будет поддерживать конструкции, появляющиеся в новых версиях JavaScript, а также новые веб-технологии, кроме того, сама старая версия браузера через какое-то время перестаёт поддерживаться разработчиками, что создаёт технические трудности при применении метода.

Методы, участвующие в сравнении

Средства, использующие для выявления поверхности атаки динамический краулинг, во многом схожи, и не имеют различий с точки зрения используемых критериев сравнения. Поэтому они при сравнении объединены под одним названием “Динамический краулинг”. Это средства Crawljax, Enemy of the State, FEEDEX, jÄk, Arachni, Htcap, XIEv, CHIEv, Black Widow Gelato. Средства JSForce и JSFlowTamper [30] по тем же причинам объединены под названием “Динамический анализ, меняющий семантику”. Также в сравнении участвуют статические анализаторы WALA [31], SAFE [32] и TAJS [33]. Работа [37] не содержит описания разработанного алгоритма, а лишь некоторые его идеи, поэтому не участвует в сравнении. Исходный код средства, описанного в работе [38], недоступен, но в работе сказано, что оно основано на анализаторе WALA, поэтому метод из работы [38] объединён при сравнении с анализатором WALA, поскольку предполагается, что свойства анализатора WALA переносятся и на это средство тоже.

Результаты

Методы, использующие динамический краулинг, способны обрабатывать любой реальный код, в том числе содержащий любые из выявленных в ходе иссле-

дования особенностей. Однако, такие методы неспособны выявлять информацию о серверных входных точках в недостижимом коде.

Методы, использующие динамический анализ с изменением семантики, заложенной в интерпретатор, не имеют такого принципиального ограничения, однако на практике не всегда понятно, как именно применить подобный метод для анализа недостижимого кода. Как упоминалось выше, “сложным” для таких методов примером является анализ кода класса, не используемого в программе. Кроме того, существующие средства, относящиеся к этому виду методов, жёстко привязаны к конкретной версии браузера. Конкретнее, код JSFlowTamper, средства, описанного в работе [30], привязан к версии браузера Chromium, вышедшей в 2020 году. Этой версии уже больше трёх лет, и она не поддерживает ряд конструкций языка, а также стандартных функций языка JavaScript и функций DOM API, добавленных с тех пор. К неподдерживаемым конструкциям языка относятся, например, некоторые варианты операторов `export` и `import`, включая конструкцию `export * as name from "mod"`, а также флаги `d` и `v` литералов регулярных выражений. Среди неподдерживаемых функций есть, в том числе, функция `Promise.withResolvers` и методы `toSorted` и `toReversed` класса `Array`. Отсутствие поддержки этих возможностей проблематично для средства динамического анализа, поскольку попытка анализируемого кода ими воспользоваться приведёт к ошибке и аварийному завершению программы, в результате чего интересующая информация о выполнении программы может быть потеряна. Эта проблема особенно остра в случае неподдерживаемых конструкций языка, поскольку при их наличии попытка загрузки кода в интерпретатор сразу приведёт к ошибке на этапе синтаксического разбора, и весь файл, содержащий неподдерживаемую конструкцию, будет отброшен. Код описанного в работе [29] анализатора JSForce не опубликован.

Статические анализаторы не обладают описанными выше недостатками. Исходный код анализаторов WALA, TAJS и SAFE доступен, и, для изучения их свойств и исследования применимости, с ними в рамках данной работы были проведены эксперименты. Поскольку непосредственно выявление серверных входных точек не поддерживается ни одним из этих анализаторов, в ходе экспериментов с каждым из анализаторов делалась попытка сгенерировать граф вызовов и напечатать значения переменных для ряда тестовых примеров.

При экспериментах с кодом, разделённым на модули с помощью упаковщика модулей Webpack, было выявлено, что TAJS и SAFE не в состоянии

корректным образом определить возвращаемое значение функции импорта модуля (функции `require()`), что приводит к невозможности корректного анализа кода, использующего упаковщики модулей. Точнее, анализатор SAFE при анализе даже максимально упрощённого примера с упаковщиком завершился с ошибкой. TAJS завершился без ошибок, однако, в результирующем графе вызовов отсутствовали рёбра, соответствующие вызовам импортированных функции. Анализатор WALA смог правильно определить возвращаемое значение функции импорта, т. е. установить, какие именно значения импортируются.

Все 3 этих анализатора способны анализировать код, использующий классы, однако, сложности возникают при попытке извлечь информацию из кода с классом, который в коде не используется (аналогично с методами динамического анализа, изменяющими семантику интерпретатора). Код методов такого класса будет недостижим. Хотя в принципе анализ такого кода статическими анализаторами возможен, в интерфейсе анализаторов SAFE и TAJS не было найдено конфигурационных параметров, позволяющих включить анализ недостижимого кода. Для WALA было сделано изменение, в результате которого все функции программы были помечены как точки входа. Однако, даже после этого изменения анализатор не добавил недостающие рёбра между методами неиспользуемого класса в граф вызовов. При экспериментах со поддержкой строковых операций было обнаружено, что WALA не вычисляет результат конкатенации даже для переменных, имеющих однозначно определённое строковое значение. При анализе примера, приведённого на листинге 1.3, WALA будет считать значение переменной `b` неизвестным строковым значением.

Листинг 1.3: Простейший пример, проверяющий поддержку строковых операций

```
1 var a = "test1234";
2 var b = a + "foobar";
```

TAJS правильно определит значение переменной `b` на листинге 1.3, однако, для частично-конкретных строк этот анализатор сохраняет только значение префикса. При анализе примера, приведённого на листинге 1.4, TAJS для переменной `c` охранит префикс `"http://test.com/req?p1="`, однако суффикс `"&p2=123"` будет потерян. Сохранение этой информации важно выявления URL-адресов запросов к серверу.

Листинг 1.4: Пример кода, проверяющий поддержку частично-конкретных строк

```
1 var a = "http://test.com/req?p1=";
2 var b = a + unknownValue();
3 var c = b + "&p2=123"
```

Анализатор SAFE для частично-конкретных запросов не сохраняет никакую информацию, т. е. он при анализе кода на листинге 1.4 будет считать значение переменной с неизвестным строковым значением. Для экспериментов, имеющих цель проверить поддержку современных библиотек, сложных для анализа, была взята библиотека jQuery версии 3.6.0. Ни один из трёх рассматриваемых статических анализаторов не смог проанализировать тестовый пример, содержащий эту библиотеку (кроме самой библиотеки никакого JavaScript-кода в этом тестовом примере не было). Анализатор SAFE быстро завершился на нём с ошибкой, анализатор TAJS проработал 1 час 14 минут и также завершился с ошибкой, анализатор WALA так и не завершился (в ходе эксперимента ожидание завершения продолжалось несколько дней).

Результаты сравнительного анализа приведены в таблице 1.

Таблица 1 — Результаты сравнительного анализа

Метод / Критерий	Динамический краулинг	Динамический анализ, меняющий семантику	WALA	SAFE	TAJS
Поддержка упаковщиков	+	+	+	−	−
Поддержка классов	+	+	+/−	+/−	+/−
Поддержка строковых операции	+	+	−	+/−	+/−
Косвенные вызовы	+	+	+	+	+
Обращения к полям по вычисленному имени	+	+	+	+	+
Поддержка библиотек	+	+	−	−	−
Анализ недостижимого кода	−	+/−	+	+/−	+/−
Отсутствие привязки к конкретной версии браузера	+	−	+	+	+

1.3 Выводы

Ограничением подходов, основанных на динамическом краулинге, является то, что они упускают запросы, которые сложно или невозможно вызвать действиями с элементами пользовательского интерфейса. Для преодоления этого ограничения в настоящей работе было принято решение сделать выбор в пользу метода анализа, который не основан на симуляции пользовательских действий. Существующие средства, использующие динамический анализ с изменением семантики, заложенной в интерпретатор, имеют жёсткую привязку к старой версии браузера. Кроме того, характерной чертой инструментов, при реализации которых вносятся изменения в исходный код существующих браузеров, является низкая стабильность работы.

В научной литературе известно несколько разработок, имеющих цель создать статический анализатор общего назначения для JavaScript-кода. Созданные анализаторы были рассмотрены в ходе обзора, с ними были проделаны эксперименты. Было выявлено, что ни один из рассмотренных статических анализаторов не удовлетворяем всем предъявленным требованиям. Ни один из них не может анализировать страницы современных веб-приложений, в том числе, ни один из них не способен проанализировать страницу, содержащую библиотеку jQuery (библиотеку, встречающуюся на 77% сайтов в сети Интернет). Таким образом видно, что использовать статический анализатор общего назначения для решаемой в настоящей работе задачи по меньшей мере непрактично. Был сделан вывод о целесообразности разработки специализированного метода статического анализа, пожертвовав универсальностью. Для разработки такого метода нужно ограничить задачу, и для этого эффективным вариантом будет исследовать особенности программ на языке JavaScript, характерных именно для клиентской части веб-приложений и для кода, который отправляет запросы на сервер. И сосредоточиться на решении более частных задач статического анализа именно для такого кода. Этому исследованию посвящена следующая глава.

Полученный алгоритм должен отличаться от существующих большей масштабируемостью, он должен за разумное время (порядка нескольких минут) и с адекватным потреблением вычислительных ресурсов обрабатывать код, содержащий современные JavaScript-библиотеки, сложные для анализа (включая jQuery). Кроме того, он должен анализировать весь имеющийся на странице код, включая

недостижимый код. Даже если часть клиентского кода страницы не может выполниться, она может представлять интерес, если несёт информацию о серверной стороне приложения. Описанию разработанного алгоритма посвящена глава 4. Дальнейшим развитием идеи анализа недостижимого кода может быть анализ закомментированного кода — кода, который разработчики решили отключить посредством заключения его в комментарии. Анализу закомментированного кода посвящён раздел 4.4, а раздел 4.7.2 содержит описание экспериментов с реализацией такого анализа.

Глава 2. Исследование особенностей реального JavaScript-кода

Данная глава посвящена исследованию особенностей реального JavaScript-кода, влияющих на возможность его анализа с целью обнаружения серверных входных точек. На защиту выносятся следующие результаты этой главы: требования к инструментам построения поверхности атаки на основе статического анализа клиентского JavaScript-кода, сформулированные на основе результатов исследования, а также эталонный набор веб-страниц (бенчмарк), который может быть использован для автоматизированной оценки эффективности методов извлечения информации о серверных входных точках из клиентского кода.

Как уже было упомянуто, представляет интерес разработка метода выявления серверных входных точек, основанного на статическом анализе клиентского JavaScript-кода. При этом, ни один из описанных в научной литературе статических анализаторов JavaScript-кода не применим для анализа клиентского кода на страницах современных веб-приложений. В связи с этим было принято решение провести исследование реального клиентского JavaScript-кода для выделения его особенностей, которые осложняют статический анализ и которые следует учитывать при разработке алгоритма обнаружения отправляемых на сервер запросов. Результаты такого исследования делают возможной разработку специализированного анализатора, который будет лучше адаптирован для работы с реальным кодом и за счёт этого обладать большей эффективностью.

Язык JavaScript это полноценный язык общего назначения, и программы на JavaScript могут выполняться не только на клиентской части веб-приложения, на JavaScript также может быть реализован сервер. Тот JavaScript-код, который относится к клиентской части, может не отправлять запросы на сервер. Таким образом, результаты проведённого исследования и соответствующим образом устроенный специализированный анализатор будут применимы не для любого существующего JavaScript-кода. Легко будет специально написать программу, которая нарушает основные принятые при разработке анализатора допущения. С другой стороны, создать работающий полностью полно и точно для произвольного кода анализатор в принципе невозможно, поэтому создаваемое решение в любом случае будет работать для какого-то подмножества программ. Поэтому приято решение исследовать то, как устроен реальный код в сети Интернет, чтобы создать алгоритм анализа, который будет работать для как можно большей доли

реально существующих JavaScript-программ, работающих на клиентской стороне веб-приложений.

2.1 Корпус кода для исследования

Для проведения исследования был собран корпус веб-страниц реальных сайтов в сети Интернет.

Для составления набора сайтов, на которых проводится исследование, было решено использовать список Alexa Top 1 Million. Это список наиболее популярных сайтов в сети Интернет. Он активно используется в существующих работах в качестве списка, на сайтах из которого проводились эксперименты, в том числе в работах [35; 39—43]. Что касается размера выборки, то, хотя большой её размер желателен, поскольку увеличение выборки положительно сказывается на её представительности, ограничивающими факторами являются количество доступных вычислительных ресурсов и доступного дискового пространства. В связи с этим при выборе количества сайтов в выборке было решено ориентироваться на размеры выборок, использованных в существующих работах, посвящённых схожим исследованиям. В работе [41] использована выборка сравнительно небольшого размера — 11 сайтов, в работе [35] размер выборки составил 100 сайтов из списка Alexa. В работе [39] авторы вручную экспериментировали со 100 сайтами, а также провели автоматизированные эксперименты на 10000 сайтов. Схожим образом поступили авторы [40] — использовали малую выборку из 500 сайтов и выборку большего размера из 10000 приложений. В статье [42] также взята выборка из 10000 сайтов, а в работе [43] исследование проведено на выборке из 50000 сайтов, взятых их списка Alexa. В рамках настоящей работы решено было провести исследование на 50169 сайтах из списка Alexa.

Для каждом сайте из выборки был запущен статический краулер, построенный на основе библиотеки Colly [44], в результате чего был собран корпус веб-страниц. Для каждой страницы корпус включает в себя HTML-файл с разметкой самой страницы, а также набор подключаемых ею JavaScript-файлов и CSS-файлов. Корпус содержит 3251056 JavaScript-файлов, относящихся к 15785278 веб-страницам.

2.2 Метод исследования

Для проведения исследования применялся следующий метод.

1. Производился случайный отбор страниц из собранного корпуса, каждая из которых вручную анализировалась на предмет наличия клиентского JavaScript-кода, отправляющего запросы на сервер. Всего на этом шаге было отобрано и проанализировано около 150 страниц, все из которых относились к разным веб-сайтам.
2. При обнаружении отправляющего запросы кода вручную выделялись его особенности, которые необходимо поддерживать статическому анализатору для того, чтобы отыскать эти запросы.
3. Для выделенных на предыдущем шаге особенностей производилась оценка их встречаемости на сайтах в сети Интернет. Для этого выполнялся автоматизированный поиск кода, обладающего особенностью, по всем файлам корпуса.

2.3 Выделенные особенности кода

В результате исследования были выделены следующие особенности.

Использование упаковщиков модулей. Современный JavaScript-код нередко организуется разработчиками посредством разбивки на модули. Однако, встроенная поддержка модулей с помощью операторов `import` и `export` была добавлена в язык только в версии ECMAScript 6 (таже известной как ECMAScript 2015) [45]. В популярных веб-браузерах эта поддержка была реализована начиная с 2017-2018 годов [46]. До её добавления разработчики были вынуждены использовать специальные средства для эмуляции разбиения кода на модули — упаковщики модулей (module bundlers). Код каждого модуля помещается упаковщиком в отдельную анонимную функцию, которой через аргументы передаётся функция импорта других модулей (функция `require()`) и объект, куда модуль может поместить экспортируемые значения (объект `exports`). Таким образом, изоляция пространства имён модулей эмулируется с помощью разделения локальных областей видимостей функций. Функция `require()` полу-

чает на вход идентификатор модуля и возвращает объект `exports`. Для анализа такого кода необходимо точно определять, экспортируемые значения какого именно модуля вернёт функция импорта, вызванная с тем или иным аргументом, иначе межмодульный анализ потока данных будет невозможен. Статический анализ кода функции `require` нетривиален. Она извлекает функцию-обёртку модуля из объекта-хранилища, в качестве имени поля используя свой первый аргумент, а потом вызывает извлечённую функцию. Происходит косвенный вызов, в котором вызываемая функция определется аргументом функции `require` и полями объекта-хранилища. Объект `exports` формируется как побочный эффект вызова функции-обёртки модуля. Таким образом, требуется контекстно-чувствительный анализ с поддержкой косвенных вызовов, обращений к полю объекта по вычисленному имени и побочных эффектов вызова функций.

Наиболее распространённым упаковщиками на данный момент являются Webpack и Browserify. Упаковщики остаются самым популярным способом разбиения клиентского JavaScript-кода на модули ряду причин, основными из которых являются сохранение поддержки старых веб-браузеров, не поддерживающих встроенный в язык механизм модулей, и использование старого кода, написанного до добавления в язык механизма поддержки модулей. К другим причинам относится использование устоявшегося технологического стека, включающего в себя применение упаковщика, и инертность при использовании привычных практик разработки.

Стандартной является ситуация, когда в клиентском коде присутствует выделенный модуль, отвечающий за обращения к серверному API (модуль-клиент API). При этом доменное имя сервера и префикс путей всех URL-адресов такой модуль, как правило, считывает из общего конфигурационного объекта, за который отвечает отдельный модуль. Пользователями такого модуля-клиента зачастую являются другие модули, вызывающие его экспортируемые функции. Основная, фиксированная часть URL-пути в таком случае будет находится в модуле-клиенте API, однако имена параметров могут определяться данными, которые пользователи этого модуля передают в вызовы его функций. Поэтому для определения имён параметров требуется, чтобы анализатор находил места использования экспортируемых функций модуля и значения передаваемых в вызовы этих функций аргументов.

Алгоритм анализа должен корректно определять импортируемые модули — возвращаемым значением функции импорта должен считаться объект с экспор-

тируемыми значениями для соответствующего модуля. Если код на странице организован с использованием упаковщика, то отсутствие возможности выявлять связи между модулями часто будет приводить к тому, что никакие обращения со страницы к серверу не будут обнаружены. В ходе эксперимента использование упаковщиков модулей было обнаружено на 67.6% сайтов из выборки.

Использование классов. Код, отправляющий запросы на сервер, бывает написан в объектно-ориентированном стиле — с использованием классов. При анализе методов классов в таком коде анализатору нужно определять, какие данные будут получаться при чтении полей объекта `this`, т. е. объекта-экземпляра класса, чей метод вызывается. А также какие данные записываются в эти поля. Анализатор должен определять, к какому классу относится метод, и находить код вызываемых методов при анализе мест их вызова. Были выявлены как случаи, когда методы классов непосредственно осуществляли отправку запросов на сервер, так и случаи, когда они участвовали в формировании данных запроса, который затем отправлялся другим кодом. Классы в языке JavaScript могут быть объявлены разными способами. Было выделено 2 подвида описываемой особенности, различающихся способом объявления. В ходе эксперимента код, использующий классы, был найден на 91% сайтов из выборки.

Использование непрямого объявления класса. Такой способ объявления был единственным возможным до появления в языке ключевого слова `class`, которое, как и ключевые слова `import` и `export`, было добавлено в версии ECMAScript 6 [47]. По ряду причин код, использующий этот метод объявления, всё ещё распространён на реальных сайтах. Основные причины схожи с причинами использования упаковщиков модулей: это сохранение поддержки старых веб-браузеров, использование старого кода и инертность технологического стека и практик разработки. При использовании непрямого способа объявления в качестве класса в коде объявляется функция. Применение оператора `new` к этой функции приводит к созданию нового объекта, причём использованная функций будет вызвана для этого объекта в качестве конструктора. Для добавления методов и полей, общих между всеми экземплярами, используется специальное свойство-объект `prototype`. То есть методы добавляются в класс с помощью операции присваивания поля объекта (объекта `prototype`). Также встречаются случаи, когда методы добавляются в коде конструктора класса с помощью присваивания полей объекта-экземпляра. На 90.86% сайтов из выборки был обнаружен код с классами, объявленными таким способом, в ходе эксперимента.

Использование прямого объявления класса с помощью ключевого слова `class`. Этот способ объявления был добавлен в язык начиная с версии ECMAScript 6. В язык была добавлена явная конструкция объявления класса с использованием ключевого слова `class`. Основной частью объявления является тело класса, в котором как конструктор, так и остальные методы, перечислены явно. Объявление класса также позволяет перечислить поля класса (поля с данными), хотя это перечисление не является обязательным. Доля веб-сайтов, на страницах которых используется этот способ объявления, изначально была невелика, однако она постоянно растёт. Классы, объявленные таким способом, найдены в ходе эксперимента на 32.2% сайтов.

Использование строковых операций. Код, отправляющий запросы на сервер, активно использует строковые операции для формирования частей запросов — URL-адресов, тел запросов, значений заголовков. Чаще всего это операция конкатенации строк, однако используются и другие поддерживаемые языком строковые операции, такие как взятие подстроки, разбиение по разделителю, замена подстроки и так далее. Для определения того, какой вид имеют запросы, отправляемые на сервер, анализаторам нужно как можно точнее моделировать эти операции — причём как с полностью конкретными данными, так и с частично неизвестными. В результате эксперимента 95.6% сайтов из выборки содержали клиентский JavaScript-код, в котором есть строковые операции.

Передача функциональных значений в программе и косвенные вызовы. Практически все вызовы в JavaScript-коде косвенные. Семантика вызова такова, что то, какая функция вызывается, всегда определяется динамически результатом выражения, задающего вызываемую функцию. Функциональные значения в JavaScript-коде нередко переприсваиваются между переменными (в т. ч. при использовании упаковщиков модулей), передаются в функции в качестве аргументов (функций обратного вызова). При использовании упаковщиков модулей экспортируемые функции переприсваиваются, нередко переприсваиваются несколько раз. Сам по себе механизм импорта и экспорта функций при использовании упаковщиков это тоже передача функциональных значений. Стиль написания кода на JavaScript и устройство стандартных программных интерфейсов стимулируют разработчиков активно использовать передачу функций обратного вызова (“колбеков”). Поддерживать косвенные вызовы необходимо, но при этом остаётся важным, чтобы анализатор отрабатывал за разумное время и при разумных ограничениях по памяти. Анализ кода, активно использующего

эту возможность языка, может не быть полностью точным, но должен осуществляться при приемлемом потреблении вычислительных ресурсов. Доля сайтов, в клиентском коде которых были найдены косвенные вызовы, составила 95.68% по результатам проведённого эксперимента.

Обращение к полю объекта по вычисленному имени. Язык JavaScript допускает обращение к полю объекта по вычисленному имени — то есть, когда имя поля не задано в коде явно, а вычисляется динамически (в качестве имени поля используется строковое значение). Эта возможность языка часто используется в реальном коде, что затрудняет статический анализ кода. Для точного определения того, к какому именно полю происходит обращение, анализатору необходимо определить значение, используемое в качестве имени поля. При использовании упаковщиков модулей функция `require` (функция, осуществляющая импорт модуля) читает поле объекта по имени-значению, пришедшему в эту функцию в качестве аргумента. Также эта возможность языка используется в функциях копирования полей из одного объекта в другой (часто такую функцию называют `extend`), при чтении обработчиков событий из объекта-контейнера обработчиков. Даже когда имя поля фиксировано, встречаются случаи, когда для сокращения размера кода имя поля, используемое несколько раз, записывается в переменную с коротким (часто однобуквенным или двухбуквенным) именем, и эта переменная затем используется в качестве имени поля. Во всех этих случаях отсутствие поддержки обращения к полю объекта по вычисленному имени с определением значения-имени поля приведёт к понижению точности получаемого графа вызовов, что приведёт к меньшей точности определения вида отправляемых из кода запросов. Ещё одним вариантом использования этой возможности языка при отправке запросов является обращение к полю объекта по вычисленному имени встречается в функциях, читающих параметры конфигурации. Нередко доменное имя, префикс URL-адреса или другие части запроса заданы на странице как поля конфигурационного объекта. При этом нередко чтение полей этого объекта (конфигурационных параметров) осуществляется с помощью специальной функции, принимающей имя поля в качестве аргумента. В таких случаях точное определение частей запроса, взятых из параметров конфигурации, потребует контекстно-чувствительного анализа таких вызовов с определением имени считываемого поля объекта. Доля сайтов, на страницах которых была обнаружена операция обращения к полю объекта по вычисленному имени, составила 95.2%.

Использование библиотек, код которых сложен для статического анализа. Как и другие виды программ, клиентский JavaScript-код активно использует библиотеки. Как уже упоминалось, код современных JavaScript-библиотек зачастую сложен для анализа. Что касается, в том числе, библиотеки jQuery — наиболее популярной библиотеки для клиентского JavaScript-кода. Сложность возникает из-за упоминавшихся выше особенностей — передачи функциональных значений и косвенных вызовов, использования строковых операций, обращения к полям объектов по вычисленным именам. Также в коде библиотек (например, Angular) используются классы. Более современные библиотеки (например, Axios) разбиты на несколько модулей, и анализ их кода требует поддержки упаковщиков модулей. Библиотеки в клиентском коде могут использоваться как непосредственно для отправки запросов на сервер, так и для формирования данных, которые затем будут отправлены в составе запроса. Также, библиотека, используемая на странице, может никак не влиять на отправку запросов на сервер — в этом случае уровень точности анализа кода библиотеки не имеет значения, однако, важно, чтобы анализатор обработал анализируемый код за разумное время и при допустимом потреблении ресурсов. Доля сайтов, на страницах которых было обнаружено использование библиотек со сложным кодом, составила 78,6%.

2.4 Эталонный набор веб-страниц

По результатам проведённого исследования был разработан эталонный набор веб-страниц (бенчмарк), позволяющий оценивать эффективность методов извлечения информации о серверных входных точках из клиентского кода. Набор содержит 32 страницы, для каждой из которых в нём есть список входных точек, запросы к которым отправляются JavaScript-кодом страницы. Всего в наборе присутствует 884 входные точки. Страницы в наборе приведены в виде файла с HTML-разметкой, а также набора подключаемых ею JS и CSS файлов. Этот набор может быть использован для автоматизированной оценки эффективности методов извлечения информации о серверных входных точках из клиентского кода. Большая часть страниц в наборе (28 страниц) взяты со случайных веб-сайтов из сети Интернет (взятых из списка Alexa Top 1 Million). Кроме того, в набор

добавлены страницы тестовых приложений Juice Shop и Joomla Store, а также страницы популярных веб-приложений с исходным кодом Mattermost и YDB. На каждой из страниц есть отправляющий запросы к серверу JavaScript-код. Для каждой особенности кода, приведённой в предыдущем разделе, в эталонном наборе есть страницы, для которых поддержка этой особенности необходима для обнаружения отправляемых со страницы запросов.

Созданный эталонный набор веб-страниц опубликован и доступен по адресу `https://github.com/seclab-msu/ajax-page-dataset`.

Список входных точек, запросы к которым отправляются клиентским JavaScript-кодом (эталонная разметка) для каждой страницы представлен в виде JSON-файла. Он содержит массив, элементами которого являются объекты, каждый из которых описывает *шаблон запроса*, обращающегося к одной из входных точек. Формат этих объектов основан на формате описания запросов в составе формата HAR [48]. Формат этих JSON-объектов подразумевает следующие поля:

- `"method"` — строка, метод запроса;
- `"url"` — строка, URL запроса (включая схему);
- `"headers"` — массив объектов, каждый из которых имеет 2 поля: `"name"` и `"value"`, значениями обоих полей являются строки;
- `"postData"` — объект, описывающий тело запроса.

Поле `"headers"` описывает набор HTTP-заголовков запроса: каждый объект описывает заголовок, поле `"name"` содержит имя заголовка, поле `"value"` содержит значение заголовка.

Объект `"postData"` может содержать следующие поля:

- `"mimeType"` — строка, это значение заголовка `Content-Type`, оно имеет формат MIME-типа [49];
- `"text"` — строка, тело запроса;
- `"params"` — массив объектов, каждый из которых имеет 2 поля: `"name"` и `"value"`, значениями обоих полей являются строки.

Массив `"params"` описывает набор параметров, передаваемых в теле запроса. Случае, если поле `"text"` у объекта `"postData"` также присутствует, эта информация будет избыточной — тем не менее, формат HAR предусматривает такое поле, и, если в разметке в эталонном наборе присутствует и поле `"text"` и поле `"params"`, то их значения будут согласованы.

Аналогично, объект с описанием запроса также содержит поле `"queryString"`, содержащий значения параметров в query-части URL-адреса (это

Листинг 2.1: Пример шаблона запроса с параметрами в теле

```
{
    "method": "POST",
    "url": "https://www.hrblog.spotify.com/wp-admin/admin-ajax.php",
    "headers": [
        {
            "name": "Host",
            "value": "www.hrblog.spotify.com"
        },
        {
            "name": "Content-Type",
            "value": "application/x-www-form-urlencoded"
        },
        {
            "name": "Content-Length",
            "value": "42"
        }
    ],
    "queryString": [],
    "postData": {
        "text": "action=pp_del_cover_image&nonce=8a26d566df",
        "mimeType": "application/x-www-form-urlencoded",
        "params": [
            {
                "name": "action",
                "value": "pp_del_cover_image"
            },
            {
                "name": "nonce",
                "value": "8a26d566df"
            }
        ]
    }
}
```

поле также соответствует формату HAR). Все запросы в разметке в эталонном наборе данных содержат поле `"url"`, что делает поле `"queryString"` избыточным, однако, его наличие может быть удобным при использовании разметки.

Все поля объектов-шаблонов запроса, кроме `"method"` и `"url"`, являются опциональными и могут отсутствовать. Если объект `"postData"` присутствует, то все его поля также являются необязательными. Отсутствие того или иного поля в

шаблоне означает, что информация, содержащаяся в нём, не важна с точки зрения обнаружения серверной входной точки, запросы к которой описываются шаблоном. Это означает, что данные в соответствующих частях запроса, скорее всего, никак не будут обработаны сервером, а их отсутствие не приведёт к тому, что запрос не будет распознан сервером как обращающийся к соответствующей серверной входной точке. К примеру, при обработке запросов к серверной входной точке тело игнорируется (данные, переданные в теле, никак не учитываются) — в этом случае поле `"postData"` может отсутствовать в шаблоне. Набор заголовков в шаблоне запроса также не является полным списком заголовков запроса к соответствующей серверной входной точке. В массиве `"headers"` в шаблоне указываются только те заголовки, которые могут повлиять на маршрутизацию запроса — то есть, если в запросе не переданы эти заголовки или переданы со значениями, отличными от указанных в шаблоне, то сервер не распознает запрос как относящийся к соответствующей серверной входной точке.

Помимо возможности отсутствия полей и заголовков, для объектов-шаблонов запросов подразумеваются следующие правила:

– пустое значение параметра, находящегося в query-части запроса или теле, означает, что значение этого параметра может быть любым (то есть значение этого параметра не влияет на маршрутизацию запроса);
– если компонента пути URL является строкой `"UNKNOWN"`, это также означает, что на месте этой компоненты пути может стоять любое значение (то есть в этом месте в пути передаётся параметр, значение которого не влияет на маршрутизацию).

Листинги 2.1 и 2.2 содержат примеры запросов из эталонной разметки.

Листинг 2.2: Пример шаблона запроса с параметрами в query-части URL

```
{
    "method": "GET",
    "url": "http://mattermost.stands/api/v4/users/tokens?page=&per_page=",
    "headers": [],
    "queryString": [
        {
            "name": "page",
            "value": ""
        },
        {
            "name": "per_page",
            "value": ""
        }
    ]
}
```

2.5 Выводы и требования к инструментам

Общим выводом по итогам исследования стало то, что для определения отправляемых запросов посредством статического анализа необходим межпроцедурный алгоритм анализа, способный определять возможные наборы аргументов вызываемых функций и возвращаемые значения. При этом алгоритм должен по возможности как можно точнее различать разные вызовы одной и той же функции (обладать хорошей контекстной чувствительностью), что требуется для повышения точности получаемых результатов анализа. Встречаемость сайтов, в коде которых используются строковые операции и обращения к полям объекта по вычисленным именам ожидаемо оказалась высокой — эти операции присутствуют практически в любой клиентской JavaScript-программе. Также, как упоминалось выше, для применимости в составе реальных систем автоматизированного поиска уязвимостей время анализа одной веб-страницы должно быть порядка нескольких минут. Это связано с современными требованиями к предельному времени работы систем поиска уязвимостей на одном приложении и тем, что у одного приложения, как правило, есть множество страниц, которые будут обрабатываться анализатором.

Алгоритм анализа должен отвечать следующим требованиям:

1. Поддерживать анализ кода, поставляемого в виде набора модулей, запакованных упаковщиком модулей (module bundler).
2. Поддерживать анализ кода, использующего классы. В том числе:
    а) объявленные посредством непрямого объявления класса,
    б) объявленные посредством прямого объявления класса (с помощью ключевого слова `class`).
3. Поддерживать строковые операции языка JavaScript и вычислять результаты этих операций.
4. Поддерживать анализ кода, содержащего косвенные вызовы и передачу функциональных значений между переменными программы и полями объектов и массивов. Анализатор может не всегда точно анализировать косвенные вызовы, но он должен выполнять анализ программ, содержащих такие вызовы, за приемлемое время и при приемлемом потреблении вычислительных ресурсов.
5. Поддерживать операцию обращения к полю объекта по вычисленному имени, вычисление значения, используемого в качестве имени поля. В случае, когда может быть вычислено точное значение или набор значений, используемых в качестве имени поля, определять, к каким именно полям объекта производится обращение.
6. Поддерживать анализ программ, использующих популярные библиотеки, в том числе сложные для статического анализа, включая библиотеку jQuery.

Глава 3. Методика поиска уязвимостей с использованием статического анализа клиентского JavaScript-кода

Данная глава содержит описание методики поиска уязвимостей веб-приложений в модели “чёрного ящика” с использованием статического анализа клиентского JavaScript-кода, которая выносится на защиту. Методика апробирована в реальной системе автоматизированного поиска уязвимостей веб-приложений SolidPoint, разрабатываемой компанией ООО «СолидСофт», а также в системе автоматического поиска уязвимостей в веб-приложениях на основе обработки больших данных, разработанной в ходе научно-исследовательской работы Центра компетенций НТИ по технологиям хранения и анализа больших данных МГУ (ЦХАБД МГУ) [9].

Методика основана на открытой методике тестирования веб-приложений с точки зрения безопасности OWASP Web Security Testing Guide версии 4.2 (WSTG), четвёртом разделе [50].

Традиционно при поиске уязвимостей выделяются этапы:

1. Сбора информации об анализируемом приложении. Этому этапу соответствует пункт 4.1 методики WSTG.
2. Непосредственно проверки приложения на наличие уязвимостей с использованием собранной информации. Этому этапу соответствуют пункты 4.2-4.12 методики WSTG.

3.1 Сбор информации об анализируемом приложении

Для сбора информации об анализируемом приложении предлагаемая методика предполагает выполнение шагов 4.1.1-4.1.10 методики WSTG, а также поиск серверных входных точек, осуществляемый посредством следующих шагов:

1. Осуществить поиск всех веб-страниц приложения посредством статического краулинга.
2. Для каждой найденной страницы выполнить поиск серверных входных точек, к которым обращается эта страница, следующими способами:

   а) Выполнить поиск отправляемых запросов, инициируемых элементами HTML-разметки, с помощью синтаксического разбора HTML-разметки страницы и анализа её элементов.
   б) Выполнить поиск отправляемых запросов статическим анализом, удовлетворяющим требованиям раздела 2.5. Такой анализ предложен в данной работе в главе 4.
3. Выполнить поиск серверных входных точек с помощью запуска динамического краулера, который должен посетить все страницы, найденные на шаге 1.
4. Выполнить поиск серверных входных точек с помощью активного управляемого перебора запросов к серверной стороне приложения и анализа ответов.
   а) Перебор URL-путей по словарю, так называемый дирбастинг. Такой перебор может быть выполнен с помощью средств OWASP Dirbuster, dirsearch, Gobuster.
   б) Перебор имён параметров запросов и других частей запроса. Примером средства с открытым исходным кодом для этой задачи является ffuf.
5. Определить используемое серверное ПО (набор используемых программ и их версий) по сигнатурам и произвести поиск по открытым источникам или заранее подготовленной базе данных информации об известных серверных входных точках для выявленного ПО.

3.2 Поиск уязвимостей

Все серверные входные точки, обнаруженные на этапе сбора информации об анализируемом приложении, должны быть проверены на предмет наличия уязвимостей, включая такие виды недостатков, как SQL-injection, Reflected XSS, XXE, Server-side template injection. Для этого должны быть выполнены шаги 4.2-4.12 методики WSTG. Следует отдельно отметить, что для достижения полноты при поиске уязвимостей в обработке параметров, передаваемых в URL-адресе, следует в том числе пробовать подставлять атакующие вектора в компоненты пути URL-адреса, так как компоненты пути также могут быть параметрами запроса.

Кроме того, предлагаемая методика предписывает поиск уязвимостей в клиентском JavaScript-коде посредством статического и динамического анализа JavaScript-кода. Должна быть проведена проверка на наличие уязвимостей DOM-based XSS, Client-side prototype pollution, а также ошибок в клиентском коде, приводящим к утечкам данных. Существует ряд открытых инструментов, осуществляющих такой анализ (к примеру, DOMDig, esflow, TaintFlow), в научных статьях описаны методы организации анализа [51—55].

3.3 Примеры применения методики для анализа приложений методом “чёрного ящика”

Описанная выше методика была применена с использованием реализации метода, описанного в главе 4 для поиска уязвимостей на реальных веб-приложениях, входящих в программы вознаграждения за обнаружение уязвимостей (программы Bug Bounty). Результатом стало обнаружение ряда реальных уязвимостей. Всего в различные программы было отправлено 20 отчётов об уязвимостях, 11 из которых были уязвимостями типа Reflected XSS, остальные относились к классам SQL injection и “небезопасная десериализация данных”. В том числе, была обнаружена уязвимость типа SQL injection на главном сайте компании IBM `www.ibm.com`. Отчёт об этой уязвимости опубликован и доступен по ссылке `https://hackerone.com/reports/1459147`. Снимок экрана с отчётом приведёт на рис. 3.1.

Кроме того, была обнаружена уязвимость типа Reflected XSS на главном сайте Amazon `www.amazon.com`. Отчёт об этой уязвимости был отправлен компании Amazon на платформе HackerOne. Отчёт был принят компанией Amazon, однако не был опубликован по решению этой компании. Анонимизированный снимок экрана, демонстрирующий эксплуатацию уязвимости, приведён на рис. 3.2.

Обнаруженная уязвимость типа “небезопасная десериализация данных” позволяла, в случае успешной эксплуатации, получить возможность исполнять произвольный код на атакуемом сервере. Компания, на сайте которой была обнаружена эта уязвимость, приняла отправленный ей отчёт на платформе HackerOne, однако на данный момент так и не ответила на запрос о его опубликовании.

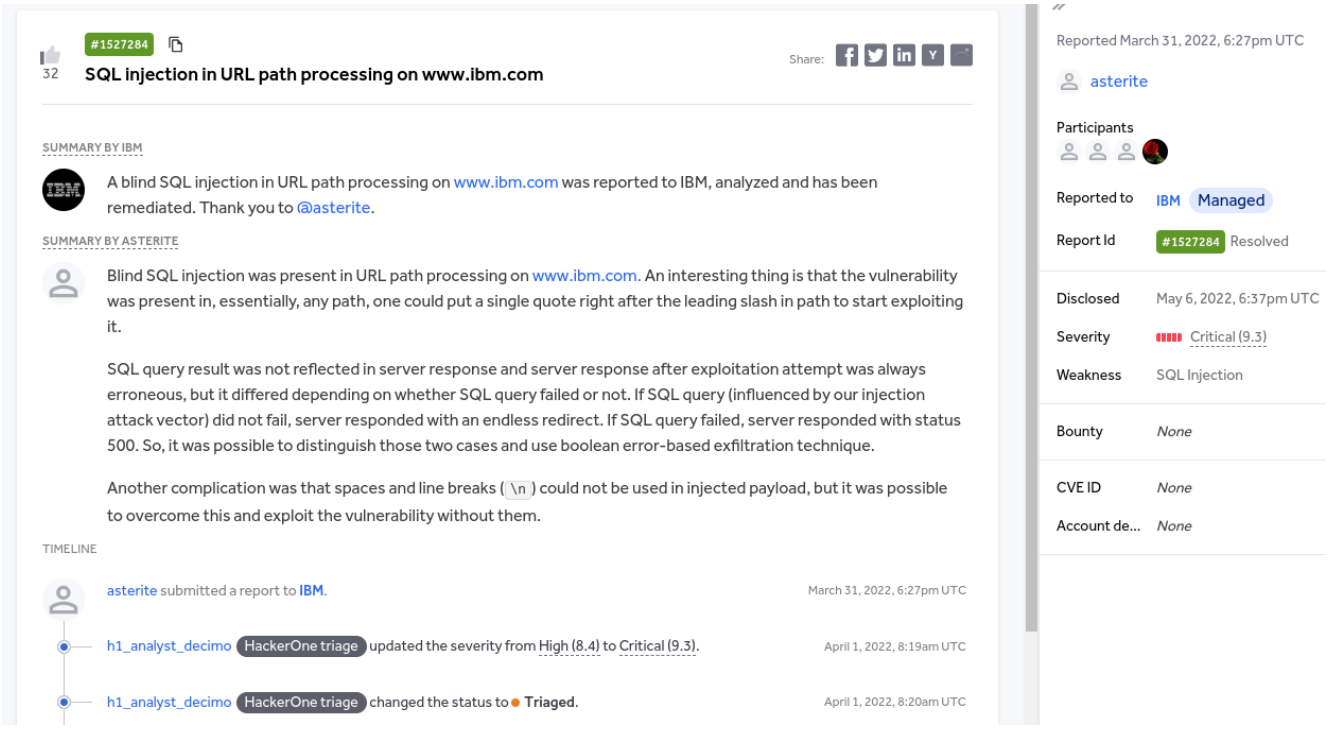

Рисунок 3.1 — Отчёт об уязвимости SQL injection на сайте www.ibm.com

При анализе обнаруженных уязвимостей были выделены следующие факторы, из-за которых, предположительно, уязвимости не были найдены предыдущими исследователями безопасности, но были найдены с помощью предлагаемой методики.

1. Запросы, требующие сложной валидации: были найдены случаи, когда для данных, отправляемых в запросе, делалось множество проверок перед отправкой, и, если проверки не проходили, отправка запроса не делалась. В одном из случаев в запросе отправлялось 11 полей формы, для каждого из которых делалась проверка. Динамическому краулеру потребовалось бы подобрать такие данные для этих полей, чтобы все проверки одновременно прошли, при этом для статического анализатора успешное прохождение проверок необязательно.
2. В некоторых случаях страница содержала вызов функции, отправляющей запрос, но только при выполнении некоторого условия, которое в реальности на странице не выполнялось. Были найдены следующие варианты этой ситуации.
   - Приложение поддерживало несколько вариантов авторизации, один из которых был отключен — а искомый запрос на странице отправлялся именно при использовании этого, отключенного, метода.

- Отсутствие на странице DOM-элемента, обработчиком события для которого была отправляющая запрос функция. В одном из обнаруженных случаев искомый запрос был предназначен для подгрузки дополнительных комментариев в ситуации, когда все имеющиеся комментарии к статье не помещались на страницу. Однако, на всех страницах приложения комментариев было не слишком много, и они помещались и так, поэтому кнопка подгрузки не появлялась. При этом у пользователя, от имени которого производился поиск уязвимостей, было недостаточно прав чтобы оставить свой комментарий.

3. Неиспользуемый код — были найдены случаи, когда код, отправляющий запрос на сервер, вообще не использовался нигде на странице. Страница не содержала вызовов функции, отправляющей запрос на сервер. Зачастую так получается за счёт того, что отправка запроса находится в общем коде, который используется на нескольких разных страницах.
4. Уязвимость в path-компоненте пути: зачастую, для экономии времени компоненты пути не тестируются на наличие уязвимостей для экономии времени, поэтому такие уязвимости могут оставаться ненайденными.

В первых трёх случаях применение статического анализа клиентского JavaScript-кода позволило обнаружить уязвимую входную точку, которую сложно выявить существующими методами.

3.4 Выводы

Полученный опыт применения разработанной методики позволяет сделать вывод о том, что расширение методики таким шагом, как использование статического анализа клиентского JavaScript-кода для обнаружения серверных входных точек, позволяет обнаружить новые входные точки, не найденные другими методами и, в конечном итоге, обнаружить новые уязвимости. Эксплуатация значительной части найденных уязвимостей была достаточно простой, и обнаружение этих уязвимостей не представляло сложности помимо сложности обнаружения соответствующей серверной входной точки. Обнаружение серверной входной точки представляло сложность для методов, использующих

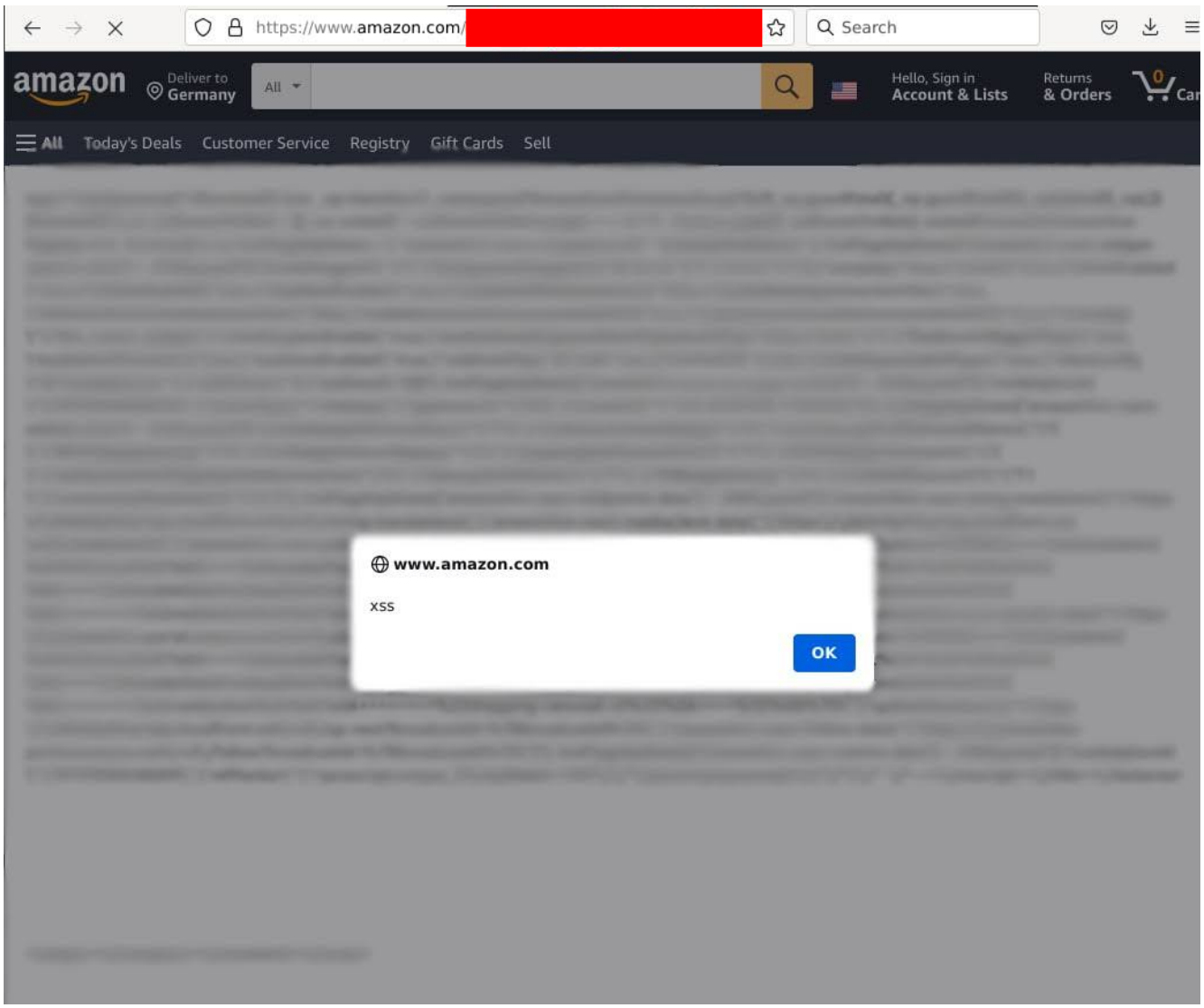

Рисунок 3.2 — Эксплуатация уязвимости Reflected XSS на сайте www.amazon.com

автоматизированное взаимодействие со страницей, а также в случае беглого ручного взаимодействия. Дополнительным обстоятельством, указывающим на то, что уязвимости удалось найти именно благодаря особенностям применяемой методики, является то, что приложения, относящиеся к Bug Bounty программам, постоянно анализируются на предмет наличия уязвимостей большим количеством исследователей безопасности. При этом основную массу приложений исследователи проверяют с помощью автоматических средств поиска уязвимостей. Тщательный ручной анализ устройства клиентской части приложения позволил бы обнаружить упомянутые уязвимые входные точки, однако этот способ анализа требует больших трудозатрат. Поэтому типичной является ситуация, когда тщательному ручному анализу подвергается клиентская часть не всех относящихся к Bug Bounty программе приложений, а некоторых избранных.

Глава 4. Метод анализа клиентского кода веб-приложения для обнаружения серверных входных точек

Данная глава посвящена описанию предлагаемого специализированного метода анализа клиентского кода веб-приложения для обнаружения серверных входных точек для последующего поиска в них уязвимостей, описанию апробации реализованного метода с использованием составленного эталонного набора страниц и на наборе приложений, использовавшихся для сравнения в предыдущих работах, а также описанию проведённых экспериментов с реализацией метода на сайтах в сети Интернет, в результате которых были обнаружены реальные уязвимости.

Предложенный метод позволяет, имея на входе веб-страницу (данную в виде URL-адреса) получить набор спецификаций HTTP-запросов, отправляемых на сервер JavaScript-кодом этой страницы. Метод удовлетворяем всем требованиям, предъявляемым к инструментам построения поверхности атаки, сформулированным в разделе 2.5. Метод заключается в поиске в JavaScript-коде страницы обращений к программным интерфейсам отправки запросов на сервер (практически всегда они имеют вид вызовов функций) и определении аргументов, передаваемых в эти обращения, а также последующем определении для каждого найденного обращения вида HTTP-запроса, который был бы отправлен этим обращением с соответствующими аргументами. Обращения к программным интерфейсам отправки запросов на сервер будем далее называть *AJAX-вызовами*. Для обнаружения в JavaScript-коде страницы AJAX-вызовов и определения их аргументов в работе предложен специализированный алгоритм статического анализа клиентского JavaScript-кода, учитывающий особенности реального кода, выявленные в ходе исследования, описанного в главе 2. Общая схема работы метода приведена на Рисунке 4.1. Красной пунктирной линией выделены те компоненты, которые были разработаны в рамках настоящей диссертационной работы.

Для применения предложенного алгоритма необходимо провести предварительный шаг в виде сбора JavaScript-кода, относящегося к анализируемой странице, синтаксического разбора этого кода и получения информации о его лексических областях видимости. Для сбора кода предлагается открыть анализируемую страницу в управляемом браузере Headless Chrome и использовать

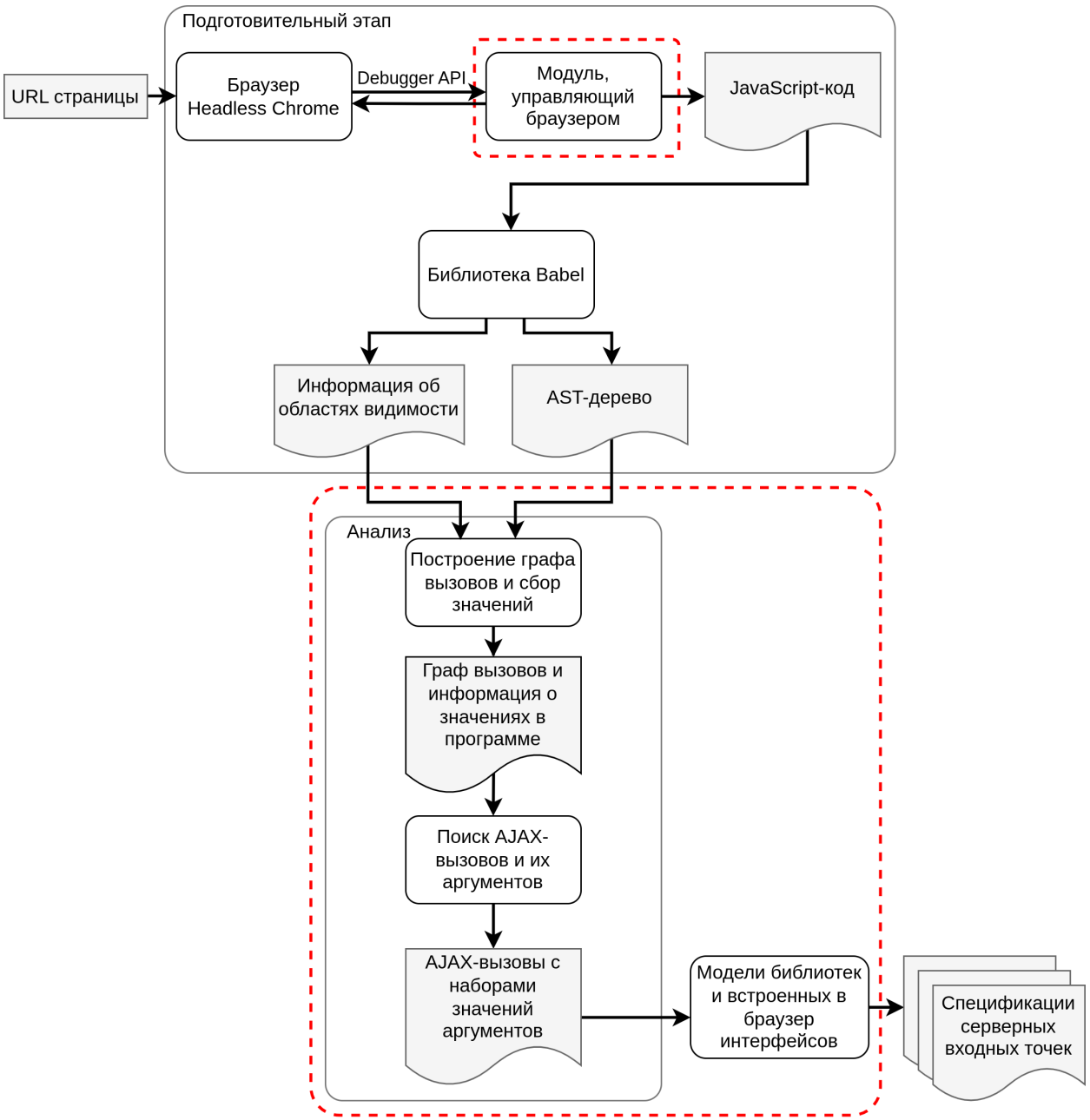

Рисунок 4.1 — Общая схема работы метода

отладочный программный интерфейс браузера для получения всех фрагментов JavaScript, которые попадают в JavaScript-интерпретатор браузера. Применение отладочного программного интерфейса браузера для анализа клиентского JavaScript-кода было предложено и реализовано в нашей работе [17]. Полученные фрагменты кода конкатенируются в один большой фрагмент в том порядке, в котором они были загружены в интерпретатор. Для синтаксического разбора используется парсер JavaScript, предоставляемый библиотекой Babel [56], который строит AST-дерево. Информация о лексических областях видимости, позволяющая различать в программе одноимённые переменные, также получается с помощью библиотеки Babel.

Предлагаемый алгоритм анализа принимает на вход AST-дерево JavaScript-кода и информацию о лексических областях видимости. Результатом работы является набор AJAX-вызовов, каждым элементом которого является пара из названия программного интерфейса отправки запросов и массива значений передаваемых в вызов аргументов. Алгоритм состоит из двух основных шагов:

1. Построение графа вызовов и сбор возможных значений переменных, полей объектов и массивов.
2. Поиск обращений к программным интерфейсам отправки запросов на сервер и аргументов, передаваемых в эти обращения.

Исходный код разработанного анализатора был опубликован, он доступен по ссылке `https://github.com/seclab-msu/relwarc`.

4.1 Построение графа вызовов и сбор возможных значений

Построение графа вызовов и сбор возможных значений переменных, полей объектов и массивов выполняются совместно и делаются с помощью итеративного алгоритма (см. алгоритм 1). Каждая итерация — это рекурсивный обход AST-дерева программы в глубину с обработкой встречаемых вершин для сбора информации о программе. Анализатор поддерживает *абстрактное состояние* программы, содержащее собранную информацию о программе. Итерации выполняются до тех пор, пока абстрактное состояние не перестанет меняться (то есть состояние по итогу итерации не совпадёт с состоянием после предыдущей итерации). Кроме того, анализатор поддерживает ограничение на максимальное количество итераций, при достижении которого анализ прекращаются. Алгоритм не обладает чувствительностью к путям, все ветви условных операторов и циклов посещаются безусловно.

Алгоритм анализа принимает AST-дерево JavaScript-кода в формате Babel AST [57] — именно в этом формате AST-дерево выдаёт библиотека Babel. Обозначим как $\mathcal{V}_{AST}$ множество всех вершин анализируемого AST-дерева. Кроме того, введём обозначения для некоторых AST-вершин более конкретных типов:

– $\mathcal{V}_{expr} \subset \mathcal{V}_{AST}$ — множество всех вершин AST-дерева, соответствующих каким-либо выражениям [58];

– $\mathcal{V}_{call} \subset \mathcal{V}_{AST}$ — множество всех вершин AST-дерева, соответствующих вызовам функций (такие вершины имеют тип `CallExpression`);
– $\mathcal{V}_{func} \subset \mathcal{V}_{AST}$ — множество всех вершин AST-дерева, соответствующих в коде функциям, то есть объявлениям функций (вершинам с типом `FunctionDeclaration`) и функциональным выражениям (вершинам с типом `FunctionExpression` или `ArrowFunctionExpression`);
– $\mathcal{V}_{cls} \subset \mathcal{V}_{AST}$ — множество всех вершин AST-дерева, соответствующих в коде классам, то есть объявлениям классов (вершинам с типом `ClassDeclaration`) и выражениям-классам (вершинам с типом `ClassExpression`).

Перед началом основных итераций алгоритма выполняется предварительный проход по AST-дереву программы с целью сбора информации о модулях, помещённых в код с помощью упаковщиков модулей — за этот шаг в псевдокоде отвечает функция $GatherBundledModuleInfo(AST)$.

Цикл **forall** *vertex* v ***traversing*** *AST* ***in DFS order*** производит обход вершин AST-дерева в глубину. При обходе алгоритм анализа пропускает код ряда библиотек. Перед обработкой вершины делается проверка, не является ли она корнем поддерева AST, являющегося AST-деревом кода библиотеки. За эту проверку отвечает функция $isLibrary(v)$. Проверка делается посредством сравнения с синтаксическими сигнатурами: в анализатор встроен ряд сигнатур популярных библиотек, по которым опознаются их AST-деревья, в том числе сигнатура библиотеки jQuery. Если для вершины v зафиксировано совпадение с сигнатурой, то пропускается эта вершина, а также все её дочерние вершины (всё поддерево AST, корнем которого является v).

Алгоритм анализа поддерживает значения следующих типов (объединение всех поддерживаемых типов будем называть $Value$):

– Значения *примитивных* типов JavaScript: **`undefined`**, `number`, `string`, `boolean`, а также значение **`null`**.
– $Unknown$ — *неизвестные* значения, существует всего 2 уникальных неизвестных значения: $Unknown$ и $FromArg$.
– $Value[]$ — *массивы*, элементами которых могут быть значения любых поддерживаемых типов.
– *Объекты* $\{string \mapsto Value\}$, представляющие собой ассоциативные массивы, ключами в которых являются строки, а значениями — значения любых поддерживаемых типов.

– $ValueSet$ — *множество* возможных значений (любых поддерживаемых типов). $ValueSet$ используется для выражения того, что в определённом месте может быть одно из нескольких возможных значений.
– $FunctionValue$ — *функциональное значение*. Каждой функции в программе (т. е. объявлению функции или функциональному выражению) сопоставлено значение $FunctionValue$, причём только одно. Кроме того, есть специальное функциональное значение **require**, которое не ассоциировано с какой-либо функцией в анализируемом коде, оно используется как часть механизма поддержки упаковщиков модулей.
– *Специальные объекты* — анализатором поддерживается несколько типов специальных объектов. Они устроены аналогично обычным объектам, описанным выше, т. е. являются ассоциативными массивами с ключами-строками и значениями $Value$, но для некоторых операций с ними анализатором используется особая семантика. Для краткости изложения полный список типов специальных объектов не приводится, перечислим лишь часть из них:
  – $Instance$ — экземпляры классов. Для каждого класса анализатором поддерживается единственный такой объект.
  – $ClassObject$ — объект-значение класса. В JavaScript возможна передача классов как значений, $ClassObject$ используется для моделирования этого.
  – $ModuleObject$ — объект, моделирующий объект `module` в системе CommonJS [59]. Экспортируемое значение `exports`, возвращаемое функцией **require**, доступно как свойство `module.exports` у этого объекта.
  – $WindowGlobal$ — объект, моделирующий глобальный объект `window` [60]. Это особый объект: все глобальные переменные являются его ключами.

Абстрактное состояние программы представляет собой четвёрку

$$
\begin{aligned}
\mathcal{A} = &\langle \boldsymbol{M}, \boldsymbol{F}, \boldsymbol{C}, \boldsymbol{CG} \rangle, \text{где} \\
&\boldsymbol{M} : Vars \to Value, \\
&\boldsymbol{F} : \langle Args, R \rangle, \\
&\qquad Args : FunctionValue \times \mathbb{N} \to 2^{Value}, \\
&\qquad R : FunctionValue \to 2^{Value}, \\
&\boldsymbol{C} : \{\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_n\}, \\
&\qquad \mathbf{c}_i = \langle Instance_i, ClassObject_i, Methods_i \rangle, \\
&\qquad\qquad Methods_i \subset string \times FunctionValue, \\
&\boldsymbol{CG} : \mathcal{V}_{call} \to 2^{FunctionValue}.
\end{aligned}
$$

Абстрактное состояние состоит из *состояния памяти* $\boldsymbol{M}$, *информации о функциях* $\boldsymbol{F}$, *информации о классах* $\boldsymbol{C}$ и *графа вызовов* $\boldsymbol{CG}$.

Состояния памяти — это отображение, ставящее значения в соответствие переменным. Оно описывает состояние переменных. Под переменными в данном случае понимаются локальные и глобальные переменные, а также аргументы функций. Множество $Vars$ это множество всех переменных программы.

Информация о функциях $\boldsymbol{F}$ это пара из двух отображений: информации об аргументах $Args$ и информации о возвращаемых значениях R. Отображение $Args$ паре из функционального значения f и номера i ставит в соответствие множество из возможных значений, которые могли быть переданы в вызов f в качестве i-ого аргумента. Отображение R функции f ставит в соответствие множество возможных значений, которые могли быть возвращены из этой функции.

Информация о классах $\boldsymbol{C}$ представляет собой набор описаний классов $\mathbf{c}_i$. Каждое описание класса $\mathbf{c}_i$ это тройка из значения экземпляра класса $Instance_i$, значения объекта класса $ClassObject_i$, а также набора методов $Methods_i$. Каждому классу сопоставлен один объект-экземпляр и один объект-значение класса. Набор методов $Methods_i$ содержит пары из имени и функционального значения метода.

Граф вызовов $\boldsymbol{CG}$ сопоставляет месту вызова множество функций, которые могут быть вызваны в этом месте.

В ходе рекурсивного обхода, выполняемого на каждой итерации анализа, анализатор обрабатывает вершины AST, соответствующие инструкциям, которые влияют на абстрактное состояние. За эту обработку в алгоритме 1 отвечает

функция $ApplyEffect : \mathcal{V}_{AST} \times State \to State$. Эта функция при необходимости вычисляет выражения, которыми заданы входные данные инструкции, используя старое значение состояния, после чего применяет к состоянию эффект от инструкции, которой соответствует обрабатываемая вершина.

Алгоритм 1: Алгоритм построения графа вызовов и сбора значений

Входные данные: $PageURL$, AST
Результат: $\boldsymbol{M}, \boldsymbol{F}, \boldsymbol{C}, \boldsymbol{CG}$
$GatherBundledModuleInfo(AST)$
$\mathcal{A} \leftarrow \langle SeedInitialMem(PageURL), \varnothing, \varnothing, \varnothing \rangle$
$\mathcal{A}_{old} \leftarrow nil$
$i \leftarrow 0$
while $\mathcal{A} \neq \mathcal{A}_{old}$ ***and*** $i < MaxIter$ **do**
    $\mathcal{A}_{old} \leftarrow \mathcal{A}$
    $\mathcal{A} \leftarrow AnalysisPass(AST, \mathcal{A})$
    $i \leftarrow i + 1$
$\langle \boldsymbol{M}, \boldsymbol{F}, \boldsymbol{C}, \boldsymbol{CG} \rangle \leftarrow \mathcal{A}$

---

function $AnalysisPass(AST, \mathcal{A})$
    forall *vertex* v ***traversing*** AST ***in DFS order*** **do**
        if $isLibrary(v)$ **then**
            skip subtree and continue
        $\mathcal{A} \leftarrow ApplyEffect(v, \mathcal{A})$
    return $\mathcal{A}$

Сбор информации об упакованных модулях

Перед началом основных итераций алгоритма сбор информации об упакованных модулях, обозначаемый в псевдокоде функцией $GatherBundledModuleInfo(AST)$. В ходе этого шага в коде с помощью синтаксических сигнатур выделяются наборы модулей, собранные с помощью упаковщика модулей (module bundles). Внутри каждого набора выделяются

отдельные модули. Упакованные модули размещаются внутри наборов в виде анонимных функций (функциональных выражений), причём каждому из них соответствует идентификатор, уникальный внутри набора, используемый для импорта модуля. Для каждого найденного модуля создаётся уникальный для него объект типа $ModuleObject$. Внутри кода каждого модуля выделяются переменные, указывающие на объекты `module` и `exports`, а также на функцию `require`. Для упрощения изложения будем считать, что в коде эти переменные имеют в коде каждого модуля именно такие названия — “module”, “exports” и “require”. Заметим, что у каждого модуля будут свои такие локальные переменные.

Информация о модулях не меняется в ходе анализа, поэтому она не является частью абстрактного состояния. Однако, она может использоваться в ходе анализа. В алгоритме анализа используется несколько функций, работающих с информацией о модулях.

Функция $ismodule(v) : \mathcal{V}_{func} \to bool$ принимает на вход вершину AST, соответствующую функции, и возвращает бинарный признак того, была ли эта функция распознана, как один из упакованных модулей.

Функция $getmoduleobj(v) : \mathcal{V}_{AST} \to ModuleObject$ принимает на вход вершину AST и, если она находится внутри упакованного модуля (внутри анонимной функции, являющейся модулем), возвращает объект $ModuleObject$, соответствущий этому модулю. Если v непосредственно сама является вершиной функции-модуля, $getmoduleobj$ также возвращает объект этого модуля. Если вершина v не относится к упакованному модуля (не является корневой вершиной функции модуля и не находится внутри модуля), использование $getmoduleobj$ приводит к ошибке и аварийному завершению алгоритма.

Функция $exportsvar(ob) : ModuleObject \to Vars$ принимает на вход объект $ModuleObject$ и возвращает локальную переменную $exports$ в теле этого модуля (то есть внутри анонимной функции, в виде которой модуль помещён упаковщиком на страницу).

Функции $havemodule$, $modulenode$ и $getexports$ принимают на вход значение $Value$ и интерпретируют его как идентификатор модуля. Идентификаторы модулей могут иметь тип `string` или `number`: в наборах модулей, собранных с помощью упаковщиков, встречаются как числовые, так и строковые идентификаторы.

Функция $havemodule : Value \to bool$ возвращает бинарный признак того, существует ли такой модуль. Если переданное ей значение это строка или число

и в процессе сбора информации об упакованных модулях был найден модуль с таким идентификатором, эта функция возвращает $true$. В противном случае она возвращает $false$.

Функция $resolvemodule : Value \to ModuleObject$ возвращает объект $ModuleObject$ модуля. Если переданное ей значение это строка или число и в процессе сбора информации об упакованных модулях был найден модуль с таким идентификатором, эта функция возвращает его $ModuleObject$. В противном случае применение этой функции приводит к ошибке и аварийному завершению алгоритма.

Функция $getexports : Value \to Value$ возвращает экспортируемое значение модуля `exports`, или $Unknown$, если модуля с таким идентификатором не существует:

$$getexports(id) = \begin{cases} exportsvar(resolvemodule(id)), & havemodule(id) = true \\ Unknown, & \text{иначе} \end{cases}$$

4.1.1 Обработка AST-вершин

Опишем то, как обрабатываются вершины AST на этом этапе анализа, то есть действие функции $ApplyEffect$. Каждый вызов функции $ApplyEffect(v,\mathcal{A})$ обрабатывает одну вершину AST — переданную в качестве первого аргумента вершину v. Выражения, которыми заданы входные данные инструкции, которой соответствует вершина v, вычисляются, при их вычислении используется абстрактное состояние $\mathcal{A}$ (переданное вторым аргументом). После этого функция $ApplyEffect$ формирует новую версию абстрактного состояния, к которой применён эффект от инструкции, которой соответствует v, и возвращает это новое абстрактное состояние.

Для вычисления значений выражений алгоритм анализа использует функцию $evalExpr(E,\mathcal{A}) : \mathcal{V}_{expr} \times State \to Value$, которая принимает на вход выражение $E \in \mathcal{V}_{expr}$ и абстрактное состояние $\mathcal{A}$, и вычисляет значение выражения E. Подробнее устройство функции $evalExpr$ описано в следующем разделе (разделе 4.1.2).

Функция $ApplyEffect$ обрабатывает следующие вершины:

– Объявления переменных — вершины с типом `VariableDeclarator`.
– Присваивания — вершины с типом `AssignmentExpression`. Это могут быть как присваивания переменных, так и присваивания полей объектов или массивов.
– Функции ($\mathcal{V}_{func}$) — вершины с типами `FunctionDeclaration`, `FunctionExpression` и `ArrowFunctionExpression`.
– Классы ($\mathcal{V}_{cls}$) — вершины с типами `ClassExpression` и `ClassDeclaration`.
– Вызовы функций — вершины с типом `CallExpression`.
– Инструкции инстанциирования классов (применения оператора **new**) — вершины с типом `NewExpression`.
– Инструкции возврата из функции — вершины с типом `ReturnStatement`.

Вершины всех остальных типов, не упомянутые в этом списке, не обрабатываются никак, то есть для них $ApplyEffect$ возвращает входное состояние без изменений.

Теперь опишем подробнее то, как обрабатывается каждая вершина на этом этапе анализа. Для этого, прежде всего, введём несколько вспомогательных функций и обозначений.

Вспомогательные функции и обозначения

Обозначим как $\mathcal{A}_{in}$ состояние, пришедшее на вход функции $ApplyEffect$, а как $\mathcal{A}_{out}$ — результирующее состояние, которое будет возвращено этой функцией.

Функция $toset$ приводит значение к типу $ValueSet$, возвращая $ValueSet$ из одного элемента, если аргумент ещё не был множеством:

$$toset(val) = \begin{cases} val, & \text{если } v \text{ это } ValueSet \\ ValueSet\{val\}, & \text{иначе} \end{cases}$$

Функция $join$ объединяет 2 значения в $ValueSet$:

$$join(val,val') = toset(val) \cup toset(val')$$

Функция $inbranch(v)$ возвращает $true$, если вершина v находится внутри ветви условного оператора **`if`** или **`switch`**, и значение $false$ иначе. При этом, если v содержится в теле функции, то условные операторы, находящиеся вне тела этой функции (возможно, содержащие само определение функции) не учитываются.

Для краткости будем использовать нотацию $[\![E]\!]_{\mathcal{A}}$ для обозначения применения функции $evalExpr$, то есть для вычисления выражений:

$$[\![E]\!]_{\mathcal{A}} = evalExpr(E,\mathcal{A}).$$

Для отображения X нотацией $X[k \mapsto val]$ будем обозначать отображение, которое во всём совпадает с X, кроме того, что ключу k оно сопоставляет значение val. Нотацию $X[k_1 \mapsto val_1, k_2 \mapsto val_2, \ldots]$ будем использовать как сокращённый вариант нотации $X[k_1 \mapsto val_1][k_2 \mapsto val_2]\ldots$.

Поскольку объекты и массивы могут содержать значения различных типов (в том числе также объекты и массивы), абстрактное состояние $\mathcal{A}$ в том числе неявно описывает состояние кучи. Нотацией $\mathcal{A} \uplus \{val[k] \mapsto val'\}$ будем обозначать абстрактное состояние, которое во всём совпадает с состоянием $\mathcal{A}$, кроме того, что в нём у объекта или массива val поле с ключом k имеет значение val. Такая нотация имеет смысл только в случаях, если:

– val это объект, и k это значение типа `string`,

– val это массив, и k это значение типа `number`.

Функция $globalvar(s) : string \to Vars$ принимает на вход строку и возвращает глобальную переменную с именем, заданным переданной строкой. Функция $localvar(s,v) : string \times \mathcal{V}_{func} \to Vars$ принимает на вход строку и AST-вершину, соответствующую функции, и возвращает локальную переменную с именем, заданным переданной строкой, находящейся в области видимости внутри этой функции.

Функция $fval(v) : \mathcal{V}_{func} \to FunctionValue$ для вершины AST, соответствующей функции в коде, возвращает функциональное значение этой функции (как уже было сказано выше, каждой функции алгоритм сопоставляет единственное значение $FunctionValue$).

Функция $fval_{parent}(v) : \mathcal{V}_{AST} \to FunctionValue$ делает то же, что и $fval$, но работает не для вершины самой функции, а для вершины, находящейся внутри её тела. То есть $fval_{parent}$ находит функцию, в теле которой находится вершина v

и возвращает результат вызова $fval$ от этой функции. Если вершина v не содержится в теле какой-либо функции, применение $fval_{parent}(v)$ приводит к ошибке и аварийному завершению алгоритма.

Функция $clsval(v,\boldsymbol{C}) : \mathcal{V}_{cls} \times \boldsymbol{C} \to \langle ClassObject, \boldsymbol{C} \rangle$ для вершины AST, соответствующей классу в коде, добавляет запись об этом классе в $\boldsymbol{C}$, если она ещё не была добавлена туда, и возвращает пару из $ClassObject$ этого класса, и дополненной информации о классах $\boldsymbol{C}$. Каждому классу в коде соответствует ровно одна запись в $\boldsymbol{C}$. Если запись о классе, вершина которого была передана в $clsval$, уже присутствует в $\boldsymbol{C}$, то функция возвращает пару из $ClassObject$ этого класса и неизменённой информации $\boldsymbol{C}$.

Объявления переменных

За объявления переменных отвечают вершины с типом `VariableDeclarator`. Объявление может содержать инициализацию — выражение, значением которого будет инициализирована объявленная переменная. Такое объявление имеет в коде вид **`var`** $a = E$. При обработке такого присваивания значение переменной a в $\boldsymbol{M}$ будет заменено на результат вычисления выражения E:

$$\frac{\mathcal{A}_{in} \vdash \langle \boldsymbol{M}, \boldsymbol{F}, \boldsymbol{C}, \boldsymbol{CG} \rangle \qquad v \vdash \textbf{var}\ a = E \qquad val = [\![E]\!]_{\mathcal{A}_{in}}}{\mathcal{A}_{out} \vdash \langle \boldsymbol{M}[a \mapsto val], \boldsymbol{F}, \boldsymbol{C}, \boldsymbol{CG} \rangle}$$

Объявление переменной может не содержать инициализации — в этом случае переменная инициализируется значением **`undefined`**.

$$\frac{\mathcal{A}_{in} \vdash \langle \boldsymbol{M}, \boldsymbol{F}, \boldsymbol{C}, \boldsymbol{CG} \rangle \qquad v \vdash \textbf{var}\ a}{\mathcal{A}_{out} \vdash \langle \boldsymbol{M}[a \mapsto \textbf{undefined}], \boldsymbol{F}, \boldsymbol{C}, \boldsymbol{CG} \rangle}$$

В нотации выше приведены объявления с ключевым словом **`var`**. В JavaScript также есть объявления с ключевыми словами **`let`** и **`const`** — они обрабатываются таким же образом.

Присваивания

Присваивания — это выражения типа `AssignmentExpression`. В рамках данной работы будем считать, что в левой части такого выражения может стоять либо идентификатор (вершина типа `Identifier`), либо обращение к полю объекта (вершина типа `MemberExpression`). В первом случае это будет присваивание переменной (с именем, определяемым идентификатором), во втором — присваиванием поля объекта.

В современных версиях JavaScript в левой части могут стоять более сложные выражения — так называемое деструктурирующее присваивание [61]. Однако, по сути, деструктурирующее присваивание является вариантом присваивания переменных, “синтаксическим сахаром”, и такой код может быть несложным образом преобразован, чтобы выполнять те же действия с помощью обычных присваиваний переменных. Для краткости изложения будем считать, что программа не содержит деструктурирующих присваиваний.

Присваивания переменных

При обработке присваивания переменной $a = E$ прежде всего будет вычислено присваиваемое значение, т. е. значение выражения E. Далее присваивание обрабатывается по-разному в зависимости от того, находится ли это присваивание внутри ветви условного оператора. Если присваивание не содержится в ветви условного оператора, выполняется т.н. *сильное обновление* — присваиваемое значение заменит в $\boldsymbol{M}$ предыдущее значение переменной.

$$\frac{\mathcal{A}_{in} \vdash \langle \boldsymbol{M}, \boldsymbol{F}, \boldsymbol{C}, \boldsymbol{CG} \rangle \qquad v \vdash a = E \qquad inbranch(v) = false \qquad val = [\![E]\!]_{\mathcal{A}_{in}}}{\mathcal{A}_{out} \vdash \langle \boldsymbol{M}[a \mapsto val], \boldsymbol{F}, \boldsymbol{C}, \boldsymbol{CG} \rangle}$$

Если же присваивание находится внутри ветви условного оператора, и в $\boldsymbol{M}$ уже содержалось какое-то значение для переменной a, то выполняется *слабое обновление* — значением для a в $\boldsymbol{M}$ станет множество $ValueSet$, содержащее новое присваиваемое значение, а также старое значение (или значения, если для a в $\boldsymbol{M}$ уже был записан $ValueSet$ из нескольких элементов):

$$\frac{\mathcal{A}_{in} \vdash \langle \boldsymbol{M}, \boldsymbol{F}, \boldsymbol{C}, \boldsymbol{CG} \rangle \qquad v \vdash a = E \qquad inbranch(v) = true \qquad val = [\![E]\!]_{\mathcal{A}_{in}}}{\mathcal{A}_{out} \vdash \langle \boldsymbol{M}[a \mapsto join(\boldsymbol{M}(a), val)], \boldsymbol{F}, \boldsymbol{C}, \boldsymbol{CG} \rangle}$$

Присваивания полей объектов и массивов

Присваивание поля объекта может быть в одной из двух форм: присваивание фиксированного поля, когда имя поля явно написано в коде программы (в этом случае используется синтаксис с точкой `ob.field = E`) и присваивание поля с вычисленным именем, когда имя поля дано в виде какого-то выражения и используется синтаксис с квадратными скобками (`ob[prop] = E`).

Определим вспомогательную функцию $setprop$:

$$setprop(ob, k, val, \mathcal{A}) : Value \times Value \times Value \times State \to State$$

Действие функции $setprop$ опишем с помощью псевдокода — см. алгоритм 2. Как видно из псевдокода функции, в случае, если на месте объекта ob оказывается $ValueSet$, присваивание будет выполнено для каждого из элементов этого множества. Функция $pickkey$, используемая функцией $setprop$, возвращает элемент множества, который подходит в качестве ключа в ob, и $Unknown$ иначе. Если подходящих элементов несколько, детерминированным образом выбирается один из элементов.

$$pickkey(k,ob) = \begin{cases} min(k_{num}), & \text{если } ob \textit{ это массив} \wedge \\ & k_{num} = \{el_i \in k \mid el_i \text{ это } \texttt{number}\} \wedge k_{num} \neq \varnothing \\ max_{lex}(k_{str}), & \text{если } ob \textit{ это } \texttt{object} \wedge \\ & k_{str} = \{el_i \in k \mid el_i \text{ это } \texttt{string}\} \wedge k_{str} \neq \varnothing \\ Unknown, & \text{иначе} \end{cases}$$

Функция $max_{lex}(elems)$ возвращает последний элемент списка, полученного в результате лексикографической сортировки элементов множества $elems$.

Для случая, когда ob это специальный объект (имеющий тип $Instance$, $ModuleObject$ или $WindowGlobal$) функция $setprop$ использует ещё одну вспомогательную функцию $setprop_{special}$, имеющую те же типы аргументов и также

Алгоритм 2: Алгоритм присвоения свойства объекта

function $setprop(ob, k, val, \mathcal{A})$
- **if** ob *это* $ValueSet$ **then**
  - **forall** $el \in ob$ **do**
    - **if** el *это объект* **then**
      - $\mathcal{A} \leftarrow setprop(el, k, val, \mathcal{A})$
  - **return** $\mathcal{A}$
- **if** k *это* $ValueSet$ **then**
  - $k \leftarrow pickkey(k,ob)$
- **if** *тип* k *это не* `string` *и не* `number` **then**
  - **return** $\mathcal{A}$
- **if** *тип* ob *это* $Instance$ *или* $ModuleObject$ *или* $WindowGlobal$ **then**
  - **return** $setprop_{special}(ob,k,val,\mathcal{A})$
- **return** $\mathcal{A} \uplus \{ob[k] \mapsto val\}$

возвращающую абстрактное состояние. Её действие также опишем с помощью псевдокода — см. алгоритм 3.

Как можно видеть из псевдокода, в случае, если объект ob имеет тип $Instance$, присваиваемое значение поля не заменяет предыдущее значение этого поля. Вместо этого новым значение поля всегда становится множество $ValueSet$, в которое добавляется как старое, так и новое значение. Кроме того, если присваиваемое значение это функция, то она будет добавлена в набор методов класса, чьим экземпляром является ob, как метод с именем k — за это отвечает функция $addmethod_{ob}$, вносящая соответствующее изменение в информацию о классах $\boldsymbol{C}$.

$$addmethod_{cls}(i,k,val,\boldsymbol{C}) = \{\mathbf{c}_1, \ldots, \mathbf{c}'_i, \ldots, \mathbf{c}_n\}, \text{где}$$
$$\mathbf{c}_i = \langle Instance_i, ClassObject_i, Methods_i \rangle$$
$$\mathbf{c}'_i = \langle Instance_i, ClassObject_i, Methods_i \cup \langle k, val \rangle \rangle$$

$$addmethod_{ob}(ob,k,val,\boldsymbol{C}) = addmethod_{cls}(i,k,val,\boldsymbol{C}), \text{где}$$
$$i \text{ такое, что } ClassObject_i = ob$$
$$\mathbf{c}_i = \langle Instance_i, ClassObject_i, Methods_i \rangle$$

Алгоритм 3: Алгоритм присвоения свойства для специальных объектов

function $setprop_{special}$(*ob*, k, val, $\mathcal{A}$)
- $\langle \boldsymbol{M}, \boldsymbol{F}, \boldsymbol{C}, \boldsymbol{CG} \rangle \leftarrow \mathcal{A}$
- **if** *ob это* $Instance$ **then**
  - **if** val *это* $FunctionValue$ **then**
    - $\boldsymbol{C} \leftarrow addmethod(ob,k,val,\boldsymbol{C})$
  - **if** *ob содержит свойство* k **then**
    - **return** $\langle \boldsymbol{M}, \boldsymbol{F}, \boldsymbol{C}, \boldsymbol{CG} \rangle \uplus \{ob[k] \mapsto join(ob[k],val)\}$
  - **else**
    - **return** $\langle \boldsymbol{M}, \boldsymbol{F}, \boldsymbol{C}, \boldsymbol{CG} \rangle \uplus \{ob[k] \mapsto toset(val)\}$
- **else if** *ob это* $ModuleObject$ ***and*** $k =$ `"exports"` **then**
  - **return** $\langle \boldsymbol{M}[exportsvar(ob) \mapsto val], \boldsymbol{F}, \boldsymbol{C}, \boldsymbol{CG} \rangle \uplus \{ob[\texttt{"exports"}] \mapsto val\}$
- **else if** *ob это* $WindowGlobal$ **then**
  - **return** $\langle \boldsymbol{M}[globalvar(k) \mapsto val], \boldsymbol{F}, \boldsymbol{C}, \boldsymbol{CG} \rangle$

Если объект ob имеет тип $ModuleObject$, а k это строка `"exports"`, то, помимо обновления соответствующего поля объекта, значение val присваивается в состоянии памяти $\boldsymbol{M}$ переменной `exports` в коде модуля, которому соответствует объект ob. Если объект ob это $WindowGlobal$, то, вместо изменения поля объекта значение val присваивается в состоянии памяти $\boldsymbol{M}$ глобальной переменной с именем k.

В начале обработки присваивания проверяется, не имеет ли оно вид E_1`.prototype.`P `=` E_2, где E_1 и E_2 — какие-то выражения, а P — явно заданное, фиксированное имя поля. То есть, задан ли объект, чьё свойство присваивается, как, в свою очередь, обращение к свойству `prototype` какого-то объекта. Если присваивание имеет такой вид, оно будет обработано специальным образом, как задание метода класса.

Рассмотрим сначала более частый случай, когда присваивание имеет вид, отличный от E_1`.prototype.`P `=` E_2.

Если v это присваивание фиксированного поля $E_1.k = E_2$, то вершина v обрабатывается следующим образом:

$$\frac{v \vdash E_1.k = E_2 \qquad ob = [\![E_1]\!]_{\mathcal{A}_{in}} \qquad val = [\![E_2]\!]_{\mathcal{A}_{in}}}{\mathcal{A}_{out} \vdash setprop(ob,k,val,\mathcal{A}_{in})}$$

В случае, если имя поля дано в виде выражения, то есть присваивание имеет вид $E_1[E_2] = E_3$, то вершина v обрабатывается следующим образом:

$$\frac{v \vdash E_1[E_2] = E_3 \qquad ob = [\![E_1]\!]_{\mathcal{A}_{in}} \qquad k = [\![E_2]\!]_{\mathcal{A}_{in}} \qquad val = [\![E_3]\!]_{\mathcal{A}_{in}}}{\mathcal{A}_{out} \vdash setprop(ob,k,val,\mathcal{A}_{in})}$$

Теперь рассмотрим случай, когда присваивание имеет вид E_1`.prototype.`P `=` E_2.

$$\frac{\begin{array}{c}\mathcal{A}_{in} \vdash \langle \boldsymbol{M}, \boldsymbol{F}, \boldsymbol{C}, \boldsymbol{CG} \rangle \qquad v \vdash E_1\texttt{.prototype.}P\ \texttt{=}\ E_2 \qquad ctor = [\![E_1]\!]_{\mathcal{A}_{in}} \\ val = [\![E_2]\!]_{\mathcal{A}_{in}} \qquad cls \text{ это } FunctionValue \qquad val \text{ это } FunctionValue\end{array}}{\mathcal{A}_{out} \vdash \langle \boldsymbol{M}, \boldsymbol{F}, addmethod_{ctor}(ctor,P,val,\boldsymbol{C}), \boldsymbol{CG} \rangle}$$

Функция $addmethod_{ctor}(ctor,P,val,\boldsymbol{C})$, используемая при обработке этого типа присваивания поля, проверяет, есть ли в $\boldsymbol{C}$ класс с методом с именем `constructor` и значением $ctor$, и, если такой класс найдётся, добавляет к его набору методов метод с именем P и значением val.

$$addmethod_{ctor}(f,k,val,\boldsymbol{C}) = \begin{cases} addmethod_{cls}(i,k,val,\boldsymbol{C}), & \text{если } \exists i : isctor(f,\mathbf{c}_i) \\ \boldsymbol{C}, & \text{иначе} \end{cases}$$

$$isctor(f,\mathbf{c}) = \langle \texttt{"constructor"}, f \rangle \in Methods, \text{ где}$$
$$\mathbf{c} = \langle Instance, ClassObject, Methods \rangle$$

Функциональные выражения

Функциональные выражения это вершины с типом `FunctionExpression` или `ArrowFunctionExpression`. Их обработка различается в зависимости от того, была ли функция распознана как один из упакованных модулей. Если функциональное выражение не было распознано как модуль, и было анонимным, то при его обработке в абстрактное состояние вносятся следующие изменения:

– в $\boldsymbol{M}$ добавляются значения формальных аргументов функции — за эти значения принимаются значения, сохранённые в $Args$,
– создаётся класс с конструктором-объявляемой функцией — это изменение вносится, если такой класс не был уже добавлен ранее.

$$\frac{\begin{array}{c}\mathcal{A}_{in} \vdash \langle \boldsymbol{M}, \boldsymbol{F}, \boldsymbol{C}, \boldsymbol{CG} \rangle \quad v \vdash function(a_1, \ldots, a_n)\ \{...\} \\ f = fval(v) \quad ismodule(v) = false \quad \langle Args, R \rangle = \boldsymbol{F} \\ \alpha_1 = Args(\langle f, 1 \rangle), \ldots, \alpha_n = Args(\langle f, n \rangle)\end{array}}{\mathcal{A}_{out} \vdash \langle \boldsymbol{M}[a_1 \mapsto \alpha_1, \ldots, a_n \mapsto \alpha_n], \boldsymbol{F}, addfunccls(f, \boldsymbol{C}), \boldsymbol{CG} \rangle}$$

$$addfunccls(f, \boldsymbol{C}) = \begin{cases} \boldsymbol{C}, & \text{если } \exists i : isctor(f, \mathbf{c}_i) \\ \boldsymbol{C}.append(mkcls(f, \boldsymbol{C})), & \text{иначе} \end{cases}$$

$$mkcls(f, \boldsymbol{C}) = \langle freshInstance(), freshClassObject(), \langle \texttt{"constructor"}, f \rangle \rangle$$

Здесь функции $freshInstance$ и $freshClassObject()$ создают, соответственно, новые объекты типа $Instance$ и $ClassObject$, не относящиеся ни к какому из уже существующих в $\boldsymbol{C}$ классов. Таким образом, анализатор добавляет в $\boldsymbol{C}$ запись о классе с конструктором-объявленной функцией. Следует отметить, что большинство таких записей в $\boldsymbol{C}$ в дальнейшем не будут использованы, поскольку большинство описанных в коде функций не используются в нём, как классы. Тем не менее, в теории, любая написанная в JavaScript-коде функция может быть использована как класс посредством применения оператора **`new`**.

Функциональное выражение, созданное с помощью ключевого слова **`function`** (ему соответствует AST-вершина типа `FunctionExpression`), может быть именованным — после ключевого слова **`function`** может быть указан идентификатор-имя функции. В этом случае указанное имя будет доступно *внутри* созданной функции, и будет указывать на эту функцию. Поэтому для такого функционального выражения дополнительно делается соответствующая запись в $\boldsymbol{M}$.

$$\frac{\begin{array}{c}\mathcal{A}_{in} \vdash \langle \boldsymbol{M}, \boldsymbol{F}, \boldsymbol{C}, \boldsymbol{CG} \rangle \quad v \vdash function\ F(a_1, \ldots, a_n)\ \{...\} \\ f = fval(v) \quad ismodule(v) = false \quad \langle Args, R \rangle = \boldsymbol{F} \\ \alpha_1 = Args(\langle f, 1 \rangle), \ldots, \alpha_n = Args(\langle f, n \rangle)\end{array}}{\mathcal{A}_{out} \vdash \langle \boldsymbol{M}[F \mapsto f, a_1 \mapsto \alpha_1, \ldots, a_n \mapsto \alpha_n], \boldsymbol{F}, addfunccls(f, \boldsymbol{C}), \boldsymbol{CG} \rangle}$$

В случае, если функциональное выражение было распознано, как модуль, то при его обработке в $\boldsymbol{M}$ добавляются значения `module`, `exports` и `require`.

$$\frac{\begin{array}{c}\mathcal{A}_{in} \vdash \langle \boldsymbol{M}, \boldsymbol{F}, \boldsymbol{C}, \boldsymbol{CG} \rangle \quad v \vdash function(...)\ \{...\} \quad m = getmoduleobj(v) \\ ismodule(v) = true \qquad mvar = localvar(\texttt{"module"}, v) \\ evar = localvar(\texttt{"exports"}, v) \qquad rvar = localvar(\texttt{"require"}, v)\end{array}}{\mathcal{A}_{out} \vdash \langle \boldsymbol{M}[mvar \mapsto m, evar \mapsto m.exports, rvar \mapsto \textbf{require}], \boldsymbol{F}, \boldsymbol{C}, \boldsymbol{CG} \rangle}$$

Использованное здесь значение **require** — это специальное функциональное значение (имеющее тип $FunctionValue$), моделирующее функцию `require` в системе CommonJS [62]. Это значение создаётся анализатором автоматически перед началом работы, оно единственно, и является общим для всех модулей.

Объявления функций

Если v это объявление функции, то есть вершина с типом `FunctionDeclaration`:

$$\frac{\begin{array}{c}\mathcal{A}_{in} \vdash \langle \boldsymbol{M}, \boldsymbol{F}, \boldsymbol{C}, \boldsymbol{CG} \rangle \quad v \vdash function\ F(a_1, \ldots, a_n)\ \{...\} \\ f = fval(v) \quad ismodule(v) = false \quad \langle Args, R \rangle = \boldsymbol{F} \\ \alpha_1 = Args(\langle f, 1 \rangle), \ldots, \alpha_n = Args(\langle f, n \rangle)\end{array}}{\mathcal{A}_{out} \vdash \langle \boldsymbol{M}[F \mapsto f, a_1 \mapsto \alpha_1, \ldots, a_n \mapsto \alpha_n], \boldsymbol{F}, addfunccls(f, \boldsymbol{C}), \boldsymbol{CG} \rangle}$$

Встретив объявление функции, анализатор добавляет запись, соответствующую имени этой функции, в $\boldsymbol{M}$ (объявление, по сути, создаёт переменную, указывающую на значение этой функции), а также добавляет в $\boldsymbol{C}$ запись о классе с конструктором-объявленной функцией (аналогично тому, как это делается для функционального выражения).

Объявления классов и классы-выражения

Объявление класса это вершина типа `ClassDeclaration`. Оно создаёт новый класс и добавляет его в область видимости, в которой находится объявление.

$$\frac{\mathcal{A}_{in} \vdash \langle \boldsymbol{M}, \boldsymbol{F}, \boldsymbol{C}, \boldsymbol{CG} \rangle \quad v \vdash class\ C\ \{...\} \quad \langle ob, \boldsymbol{C}' \rangle = clsval(\boldsymbol{C})}{\mathcal{A}_{out} \vdash \langle \boldsymbol{M}[C \mapsto ob], \boldsymbol{F}, \boldsymbol{C}', \boldsymbol{CG} \rangle}$$

Информация о классе добавляется в $\boldsymbol{C}$, в $\boldsymbol{M}$ добавляется переменная с именем, совпадающим с именем класса, значение которой это $ClassObject$ этого класса.

Классы-выражения это вершины типа `ClassExpression`, как и функциональные выражения, они могут быть как именованными, так и не именованными. Обработка именованных классов-выражений полностью аналогична обработке объявлений классов, за исключением того, что переменная, значение которой будет соответствующим $ClassObject$, добавляемая в $\boldsymbol{M}$, будет относиться к области видимости внутри класса, а не к области видимости, в которой находится класс-выражение. Соответственно, для анонимного (не именованного) класса-выражения запись в $\boldsymbol{M}$ не делается.

Вызовы функций

Если v это вызов функции, то есть вершина типа `CallExpression` ($v \in \mathcal{V}_{call}$):

$$\frac{\begin{array}{c}\mathcal{A}_{in} \vdash \langle \boldsymbol{M}, \boldsymbol{F}, \boldsymbol{C}, \boldsymbol{CG} \rangle \qquad v \vdash E_f(E_1,E_2,\ldots,E_n) \qquad \langle Args, R \rangle = \boldsymbol{F} \\ f = [\![E_f]\!]_{\mathcal{A}_{in}} \qquad a_1 = [\![E_1]\!]_{\mathcal{A}_{in}} \quad a_2 = [\![E_2]\!]_{\mathcal{A}_{in}} \quad \ldots \quad a_n = [\![E_n]\!]_{\mathcal{A}_{in}} \\ fvals = \{f_1, f_2, ..., f_n\} = \{f_i \in toset(f) \mid f_i \text{ это } FunctionValue\} \\ \{a'_{i,j}\} = \{Args(\langle f_i, j \rangle) \cup toset(a_j)\} \\ Args' = Args[\langle f_1, 1 \rangle \mapsto a'_{1,1}, \langle f_1, 2 \rangle \mapsto a'_{1,2}, \ldots, \langle f_2, 1 \rangle \mapsto a'_{2,1}, \ldots]\end{array}}{\mathcal{A}_{out} \vdash \langle \boldsymbol{M}, \langle Args', R \rangle, \boldsymbol{C}, \boldsymbol{CG}[v \mapsto \boldsymbol{CG}(v) \cup fvals] \rangle}$$

При обработке вызова функции изменяется граф вызовов $\boldsymbol{CG}$ и информация о функциях $\boldsymbol{F}$. Все функциональные значения, в которые вычисляется выражение E_f, которое стоит на месте вызываемой функции, добавляются в граф вызовов как функции, которые могут быть вызваны в месте вызова v. Кроме того, для каждого из этих значений в $\boldsymbol{F}$ добавляется информация о возможных значениях аргументов.

Инструкции инстанциирования классов

При обработке мест инстанциирования классов (то есть мест применения оператора **new**), за которые в AST-дереве отвечают вершины типа `NewExpression` , выполняется сохранение в $Args$ набора значений передаваемых в конструктор аргументов. На месте значения, задающего инстанциируемый класс, может стоять либо функция (в этом случае это будет соответствовать непрямому объявлению класса), либо $ClassObject$ (это будет соответствовать прямому объявлению класса). Во втором случае, для сохранения значений аргументов в $Args$ нужно получить конструктор класса, которому соответствует этот $ClassObject$. Введём функцию $getctor$, которая принимает на вход $ClassObject$ и возвращает функцию-метод соответствующего ему класса с именем `constructor`, если такой метод существует в наборе методов этого класса. Если метода с таким именем у этого класса нет, то $getctor$ возвращает $Unknown$. Также, введём функцию $toctor$ следующего вида:

$$toctor(val) = \begin{cases} val, & val \text{ это } FunctionValue, \\ getctor(val) & val \text{ это } ClassObject, \\ Unknown, & \text{иначе}. \end{cases}$$

Наконец, введём функцию $toctorset$, которая принимает на вход $ValueSet$ и возвращает новый $ValueSet$, полученный в результате применения функции $toctor$ ко всем элементам входного множества.

$$toctorset(values) = \{toctor(v_1), toctor(v_2), \ldots, toctor(v_n) \mid v_i \in values\}$$

Обработка вершин типа `NewExpression` выглядит следующим образом:

$$\frac{\begin{array}{c} \mathcal{A}_{in} \vdash \langle \boldsymbol{M}, \boldsymbol{F}, \boldsymbol{C}, \boldsymbol{CG} \rangle \quad v \vdash \textbf{new } E_f(E_1,E_2,\ldots,E_n) \quad \langle Args, R \rangle = \boldsymbol{F} \\ f = [\![E_f]\!]_{\mathcal{A}_{in}} \qquad a_1 = [\![E_1]\!]_{\mathcal{A}_{in}} \quad a_2 = [\![E_2]\!]_{\mathcal{A}_{in}} \quad \ldots \quad a_n = [\![E_n]\!]_{\mathcal{A}_{in}} \\ \{f_1, f_2, ..., f_n\} = \{f_i \in toctorset(toset(f)) \mid f_i \text{ это } FunctionValue\} \\ \{a'_{i,j}\} = \{Args(\langle f_i, j \rangle) \cup toset(a_j)\} \\ Args' = Args[\langle f_1, 1 \rangle \mapsto a'_{1,1}, \langle f_1, 2 \rangle \mapsto a'_{1,2}, \ldots, \langle f_2, 1 \rangle \mapsto a'_{2,1}, \ldots] \end{array}}{\mathcal{A}_{out} \vdash \langle \boldsymbol{M}, \langle Args', R \rangle, \boldsymbol{C}, \boldsymbol{CG} \rangle}$$

Возврат значения из функции

Инструкции возврата значения из функции соответствует вершина с типом `ReturnStatement`.

$$\frac{\mathcal{A}_{in} \vdash \langle \boldsymbol{M}, \boldsymbol{F}, \boldsymbol{C}, \boldsymbol{CG} \rangle \qquad v \vdash return\ E \qquad \langle Args, R \rangle = \boldsymbol{F} \qquad f = fval_{parent}(v)}{\mathcal{A}_{out} \vdash \langle \boldsymbol{M}, \langle Args, R[f \mapsto R(f) \cup toset([\![E]\!]_{\mathcal{A}_{in}})] \rangle, \boldsymbol{C}, \boldsymbol{CG} \rangle}$$

4.1.2 Алгоритм вычисления выражений

Опишем алгоритм вычисления выражений, то есть функцию $evalExpr(v, \mathcal{A})$, для обозначения применения которой также используется нотация $[\![v]\!]_{\mathcal{A}}$. Работа этой функции зависит от типа вершины, поданной ей на вход.

Литералы и переменные

Получив на вход литерал примитивного типа — строковой, числовой, булевский или литерал **`undefined`**, функция возвращает соответствующее значение. Аналогично, получив на вход литерал **`null`**, функция возвращает значение **`null`**.

$$[\![v : Literal]\!]_{\mathcal{A}} = v.value$$

Получив на вход идентификатор с именем $name$ (то есть имя переменной), $evalExpr$ вернёт соответствующее значение из состояния памяти.

$$[\![v : Identifier]\!]_{\mathcal{A}} = \boldsymbol{M}(v.name)$$

При обработке литералов массивов выражения, которыми задаются значения элементов, вычисляются с помощью рекурсивных вызовов $evalExpr$, после чего конструируется новый массив, содержащий вычисленные элементы.

$$[\![v : [E_1, E_2, \ldots, E_n]]\!]_{\mathcal{A}} = [[\![E_1]\!]_{\mathcal{A}}, [\![E_2]\!]_{\mathcal{A}}, \ldots, [\![E_n]\!]_{\mathcal{A}}]$$

Аналогично делается обработка объектных литералов: вычисляются значения полей объекта, затем конструируется новый объект, содержащий их.

$$[\![v : \{k_1 : E_1, k_2 : E_2, \ldots, k_n : E_n\}]\!]_{\mathcal{A}} = \{k_1 : [\![E_1]\!]_{\mathcal{A}}, k_2 : [\![E_2]\!]_{\mathcal{A}}, \ldots, k_n : [\![E_n]\!]_{\mathcal{A}}\}$$

Одноместные и двуместные операции

Функция вычисления выражений $evalExpr$ поддерживает 2 одноместные префиксные операции (унарные операции). Это отрицание числа и логическое отрицание: {`-`, `!`}.

Введём вспомогательную функцию $unop$ следующего вида:

$$unop(\triangle, arg, \mathcal{A}) = \begin{cases} Unknown, & arg = Unknown \\ \bigcup\limits_{el \in arg} toset(unop(\triangle, el, \mathcal{A})), & arg \text{ это } ValueSet, \\ \triangle\ arg, & \text{иначе}. \end{cases}$$

Здесь нотация $\triangle\ arg$ означает применение к значению arg соответствующей унарной операции (отрицания числа или логического отрицания). Операция применяется в соответствии с семантикой языка JavaScript [63]. Если тип операнда не соответствует выполняемой операции, то производится приведение типа к подходящему (к числовому или булевому типу).

Вычисление значения унарного выражения (вершины типа `UnaryExpression`) выглядит следующим образом:

$$[\![v : \triangle\, E]\!]_{\mathcal{A}} = \begin{cases} unop(\triangle, [\![E]\!]_{\mathcal{A}}, \mathcal{A}), & \triangle \in \{\texttt{-}, \texttt{!}\}, \\ Unknown, & \text{иначе}. \end{cases}$$

Если унарная операция не поддерживается, результатом вычисления выражения является $Unknown$.

Функция вычисления выражений $evalExpr$ поддерживает 3 двуместные (бинарные) операции: операцию сложения `+`, а также проверку на тождественное равенство `===` и тождественное неравенство `!==`.

Для обработки бинарных выражений также выражений также введём вспомогательную функцию:

$$binop(\circ,x,y,\mathcal{A}) = \begin{cases} \bigcup\limits_{el_x \in x} toset(binop(\circ,el_x,y,\mathcal{A})), & x \text{ это } ValueSet, \\ \bigcup\limits_{el_y \in y} toset(binop(\circ,x,el_y,\mathcal{A})), & y \text{ это } ValueSet, \\ x \circ y, & \text{иначе}. \end{cases}$$

Здесь нотация $x \circ y$ означает применение к значениям x и y соответствующей бинарной операции.

Операция === это операция проверки на строгое равенство, она выполняется в соответствии с семантикой соответствующей операции языка JavaScript [64]. Это означает, что для операндов различающихся типов результат проверки всегда равен **`false`**. Для операндов одинакового типа результатом является **`true`** только в случае, если они имеют строго совпадающие значения, иначе результатом является **`false`**. Для объектов и массивов это означает, что для равенства с точки зрения этой операции сравниваемые значения должны быть одним и тем же объектом (или массивом). Различные объекты, имеющие одинаковый набор ключей и значений, не будут считаться равными с точки зрения операции ===. Это же касается значений, не входящих в систему типов JavaScript, но используемых анализатором: $Unknown$, $FromArg$, $FunctionValue$ и специальных объектов. Если левый операнд это $Unknown$, то результатом операции === будет считаться **`true`** если и только если правый операнд также имеет значение $Unknown$. Для всех остальных значений операция === выполняет проверку на то, что левый и правый операнд это одно и то же значение. Результатом операции `!==` является отрицание результата применения операции ===.

При вычислении результата операции + выполняется приведение операндов к примитивным типам языка JavaScript. При этом значение $FromArg$ приводится к строке `"FROM_ARG"`, значение $Unknown$ приводится к строке `"UNKNOWN"`. Все остальные значения, не входящие в систему типов JavaScript, но используемые анализатором (включая $FunctionValue$ и специальные объекты) также приводятся к строке `"UNKNOWN"`. Объекты приводятся к строке `"[object Object]"`. Массивы приводятся к строке, получаемой, как результаты приведения к строке каждого элемента массива, соединённые запятой. При приведении к строке элементов массива, имеющих непримитивные типы, используются такие же правила, как только что были описаны для приведения к примитивным типам операндов операции +. Элементы массива, имеющие примитивные типы, приводятся к строке в соответствии со стандартными правилами языка JavaScript [65]. После

приведения операндов к примитивным типам результат операции + вычисляется в соответствии с семантикой этой операции в языке JavaScript [66].

Вычисление значения бинарного выражения (вершины, имеющий тип `BinaryExpression`) выглядит следующим образом:

$$[\![v:\ E_1 \circ E_2]\!]_{\mathcal{A}} = \begin{cases} binop(\circ, [\![E_1]\!]_{\mathcal{A}}, [\![E_2]\!]_{\mathcal{A}}, \mathcal{A}), & \circ \in \{\texttt{+}, \texttt{===}, \texttt{!==}\}, \\ Unknown, & \textit{иначе}. \end{cases}$$

Аналогично обработке унарных операций, результатом вычисления выражения с неподдерживаемой бинарной операцией является $Unknown$.

Чтение поля объекта или массива

Для описания того, как обрабатывается чтение поля объекта или массива, введём несколько вспомогательных функций. Функция $getglobal$ возвращает глобальную переменную с именем k, если она существует:

$$getglobal(k, \boldsymbol{M}) = \begin{cases} \boldsymbol{M}(globalvar(k)), & \textit{в } \boldsymbol{M} \textit{ есть ключ } globalvar(k), \\ \texttt{undefined}, & \textit{иначе}. \end{cases}$$

Функция $getprop_{instance}(o, k)$ проверяет, есть ли у класса, к которому относится o, метод с именем k, и если он есть, возвращает этот метод. В противном случае $getprop_{instance}$ предпринимает попытку считать свойство объекта o с именем k. Если такое свойство существует, $getprop_{instance}$ возвращает его значение, иначе возвращается `undefined`.

Введём также функцию $getprop$ следующего вида:

$$getprop(o,k,\mathcal{A}) = \begin{cases} Unknown, & o = Unknown \vee o = null \vee \\ & o = \texttt{undefined} \vee k = Unknown \\ \bigcup\limits_{el_o \in o} toset(getprop(el_o,k,\mathcal{A})), & o \text{ это } ValueSet, \\ \bigcup\limits_{el_k \in k} toset(getprop(o,el_k,\mathcal{A})), & k \text{ это } ValueSet, \\ getglobal(k,\boldsymbol{M}), & o \text{ это } WindowGlobal, \\ getprop_{instance}(o,k), & o \text{ это } Instance, \\ o[k], & (o \text{ это объект или массив}) \wedge \\ & o \text{ содержит поле } k, \\ \texttt{undefined}, & \text{иначе}. \end{cases}$$

Здесь нотация $o[k]$ означает чтение поля объекта или массива o к ключом (или индексом) k.

Как и при присвоении свойства объекта, при чтении свойства имя свойства может быть задано двумя способами. Читаться может фиксированное поля, имя которого явно задано в коде (при этом используется синтаксис с точкой), либо имя поля может задаваться выражением (в этом случае используется синтаксис с квадратными скобками). Как и в случае с присвоением свойства объекта, обработка этих случаев схожая, отличие лишь в том, что в одном из случаев имя свойства нужно вычислять, а в другом нет.

Чтение свойства с фиксированным именем обрабатывается следующим образом:

$$[\![v : \ E.k]\!]_{\mathcal{A}} = getprop([\![E]\!]_{\mathcal{A}},k,\mathcal{A})$$

Чтение свойства с вычисляемым именем обрабатывается следующим образом:

$$[\![v : \ E_1[E_2]]\!]_{\mathcal{A}} = getprop([\![E_1]\!]_{\mathcal{A}},[\![E_2]\!]_{\mathcal{A}},\mathcal{A})$$

Вызовы функций

Алгоритм анализа поддерживает ряд встроенных функций языка JavaScript. К таким функциям относятся:

– функции URL-кодирования строк `escape`, `encodeURIComponent`, `encodeURI`,
– функции преобразования строк в числа `parseInt` и `parseFloat`,
– функция `Math.random()` — её возвращаемым значением считается константа, одинаковая при всех запусках алгоритма,
– функция сериализации объекта `JSON.stringify`,
– методы класса `String`.

При обработке вызова проверяется, является ли вызываемая функция одной из поддерживаемых встроенных функций. Проверка осуществляется синтаксически, по виду выражения, задающего вызываемую функцию. За эту проверку отвечает функция $isBuiltinCall$. Если вызов распознан как вызов поддерживаемой встроенной функции, используется механизм поддержки встроенных функций, за применение которого отвечает функция $callBuiltin$. Она вычисляет аргументы вызываемой функции с помощью обращения к $evalExpr$. Для реализации поддержки встроенных функций используются обращения к функциям реального интерпретатора JavaScript.

Ещё один случай, обрабатываемый специальным образом, это вызовы функции **require** — их возвращаемым значением будет считаться объект `exports` модуля, идентификатор которого задаётся значением первого аргумента вызова. Для его поиска используется функция $getexports$, описанная выше.

Во всех остальных случаях для определения результата вызова функции (то есть определения возвращаемого значения вызова) используются значения, сохранённые в информации о возвращаемых значениях R (являющейся частью информации о функциях $\boldsymbol{F}$ в состоянии $\mathcal{A}$). Точнее, результатом вызова считается $ValueSet$, множеством элементов которого является объединение множеств возвращаемых значений из R, взятых по всем функциям, которые могут вызываться в месте вызова v согласно графу вызовов $\boldsymbol{CG}$.

Пусть n это количество аргументов, передаваемое вызываемой функции, $E_1, \ldots, E_n$ — сами передаваемые аргументы, E_f — выражение, которое в обрабатываемом вызове задаёт вызываемую функцию. Обработка вершины $v \in \mathcal{V}_{call}$ функцией $evalExpr$ выглядит следующим образом:

$$[\![v : E_f(E_1,\ldots,E_n)]\!]_{\mathcal{A}} = \begin{cases} callBuiltin(v), & isBuiltinCall(E_f), \\ \boldsymbol{M}(getexports([\![E_1]\!]_{\mathcal{A}})), & \textbf{require} \in \boldsymbol{CG}(v) \wedge n > 0, \\ Unknown, & \textbf{require} \in \boldsymbol{CG}(v) \wedge n = 0, \\ \bigcup_{f \in \boldsymbol{CG}(v)} R(f), & \textit{иначе}. \end{cases}$$

Места инстанциирования классов

Как и в случае с вызовами функций, для некоторых классов у алгоритма анализа есть встроенная обработка. Как и вызываемые функции, классы, чья обработка встроена в алгоритм анализа распознаются в месте инстанциирования по виду выражения, задающего инстанциируемый класс. Точнее, для всех таких классов ожидается, что они будут заданы в месте инстанциирования в виде идентификаторов, и распознавание делается по имени этого идентификатора (то есть просто по имени класса). За проверку того, является ли инстанциируемый класс одним из классов, для которых есть встроенная поддержка, отвечает функция $isBuiltinCls$, принимающая на вход AST-вершину. В набор классов, для которых есть встроенная поддержка, входит класс $Object$ (результатом его инстанциирования является новый пустой объект, не содержащий свойств) и $Array$ (результатом его инстанциирования является новый пустой массив). Для применения механизма встроенной поддержки классов используется функция $newBuiltin$.

Также, введём несколько вспомогательных функций.

Функция $getinstance_{ob}(ob,\boldsymbol{C}) : ClassObject \times \boldsymbol{C} \to Instance$ принимает на вход $ClassObject$ и $\boldsymbol{C}$ и возвращает соответствующий объект $Instance$.

$$\begin{aligned} getinstance_{ob}(ob,\boldsymbol{C}) = Instance_i,\ &\textit{где} \\ &i \text{ такое, что } ClassObject_i = ob \\ &\mathbf{c}_i = \langle Instance_i, ClassObject_i, Methods_i \rangle \end{aligned}$$

Функция $getinstance_{ctor}(f,\boldsymbol{C}) : FunctionValue \times \boldsymbol{C} \to Value$ принимает на вход функциональное значение f и $\boldsymbol{C}$ и, если существует класс с конструктором f, возвращает его $Instance$. Если такого класса не существует, функция возвращает $Unknown$.

$$getinstance_{ctor}(f,\boldsymbol{C}) = \begin{cases} Instance_i, & \exists i : isctor(f,\mathbf{c}_i) \\ Unknown, & \textit{иначе}. \end{cases}$$

Кроме того, введём функцию $getinstance$ следующего вида:

$$getinstance(cls,\boldsymbol{C}) = \begin{cases} \bigcup\limits_{el \in cls} toset(getinstance(el,\boldsymbol{C})), & cls \textit{ это } ValueSet, \\ getinstance_{ob}(cls,\boldsymbol{C}), & cls \textit{ это } ClassObject, \\ getinstance_{ctor}(cls,\boldsymbol{C}), & cls \textit{ это } FunctionValue, \\ Unknown, & \textit{иначе}. \end{cases}$$

Места инстанциирования классов (то есть места применения оператора **new**), за которые отвечают вершины типа `NewExpression`, обрабатываются следующим образом:

$$[\![v : \textbf{new } E(\ldots)]\!]_{\mathcal{A}} = \begin{cases} newBuiltin(v), & isBuiltinCls(E), \\ getinstance([\![E]\!]_{\mathcal{A}},\boldsymbol{C}), & \textit{иначе}. \end{cases}$$

Тернарный оператор

Тернарный оператор записывается в виде `condition ? expr1 : expr2`, он используется для выбора одного из результатов двух выражений в зависимости от условия. За него отвечает AST-вершина с типом `ConditionalExpression`. В случае, если значение условия неизвестно, результатом выражения будет считаться $ValueSet$ из объединения результатов вычисления обеих ветвей. Если одно из значений выбираемых выражений это $Unknown$, а другое нет, то выбирается то значение, которое не совпадает с $Unknown$. Иначе для выбора значений используется истинность значения-условия.

$$[\![v : E_c \ ? \ E_1 : E_2]\!]_{\mathcal{A}} = \begin{cases} join([\![E_1]\!]_{\mathcal{A}},[\![E_2]\!]_{\mathcal{A}}), & [\![E_c]\!]_{\mathcal{A}} = Unknown, \\ [\![E_1]\!]_{\mathcal{A}}, & [\![E_2]\!]_{\mathcal{A}} = Unknown \vee \\ & truthy([\![E_c]\!]_{\mathcal{A}}) \wedge [\![E_1]\!]_{\mathcal{A}} \neq Unknown, \\ [\![E_2]\!]_{\mathcal{A}}, & \textit{иначе}. \end{cases}$$

Здесь функция $truthy(val) : Value \to bool$ возвращает бинарный признак того, является ли значение val конкретным значением системы типов JavaScript, приведение которого к типу `boolean` дало бы результат **`true`**.

Ключевое слово `this`

Для определения значения, в которое вычисляется ключевое слово **`this`** (ему соответствует вершина AST типа `ThisExpression`), берётся функция, в теле которой находится это ключевое слово, и проверяется, есть ли в $\boldsymbol{C}$ класс, методом которого является эта функция. Если обрабатываемое в данный момент ключевое слово **`this`** находится вне какой-либо функции, или такого класса в $\boldsymbol{C}$ не нашлось, то значением, в которое вычисляется это ключевое слово, считается $Unknown$.

Введём вспомогательную функцию $getinstance_{method}$, которая для функционального значения f возвращает $Instance$ класса из $\boldsymbol{C}$, у которого есть метод с таким функциональным значением, и $Unknown$, если такого класса нет в $\boldsymbol{C}$:

$$getinstance_{method}(f,\boldsymbol{C}) = \begin{cases} Instance_i, & \exists i : \langle \ldots, f \rangle \in Methods_i \\ & \mathbf{c}_i = \langle Instance_i, \ldots, Methods_i \rangle, \\ Unknown, & \textit{иначе}. \end{cases}$$

Также, пусть функция $infunc(v) : \mathcal{V}_{AST} \to bool$ возвращает бинарный признак того, находится ли вершина v в какой-либо функции. Тогда обработка ключевого слова **`this`** функцией вычисления выражений $evalExpr$ будет иметь следующий вид:

$$[\![v : \textbf{this}]\!]_{\mathcal{A}} = \begin{cases} getinstance_{method}(fval_{parent}(v),\boldsymbol{C}), & infunc(v), \\ Unknown, & \textit{иначе}. \end{cases}$$

Функциональные выражения и классы-выражения

Для функциональных выражений (вершин типа `FunctionExpression`) и классов-выражений (вершин типа `ClassExpression`) результатом $evalExpr$ бу-

дет соответствующее им значение: $FunctionValue$ для функций и $ClassObject$ для классов.

$$[\![v : function \ldots (\ldots)\{\ldots\}]\!]_{\mathcal{A}} = fval(v)$$

$$[\![v : class \ldots \{\ldots\}]\!]_{\mathcal{A}} = ob, \text{ где } \langle \ldots, ob \rangle = clsval(v, \boldsymbol{C})$$

4.2 Поиск AJAX-вызовов и их аргументов

В данном разделе описывается алгоритм поиска обращений к программным интерфейсам отправки запросов на сервер и аргументов, передаваемых в эти обращения (см. алгоритм 4). На этом этапе используется информация, собранная на предыдущем этапе, точнее, используется абстрактное состояние $\langle \boldsymbol{M}, \boldsymbol{F}, \boldsymbol{C}, \boldsymbol{CG} \rangle$, являющееся результатом работы алгоритма, описанного в предыдущем разделе.

Результатом работы алгоритма поиска AJAX-вызовов и их аргументов является набор $\mathcal{C}$, состоящий из пар $\langle p, argValues \rangle$, где:

– p это строка, название сработавшей сигнатуры AJAX-вызова (например, `"$.ajax"` или `"fetch"`);
– $argValues : Value[]$ это массив значений аргументов.

В алгоритме поиск AJAX-вызовов и их аргументов используется ещё одна функция, выполняющая обход AST-дерева программы: $FindRequestsAnalysisPass$. Обход делается схоже с тем, как работает функция $AnalysisPass$ из предыдущего раздела, однако, помимо обработки вершин функцией $ApplyEffect$, делается ещё несколько вещей:

1. При входе в вершину функции при рекурсивном обходе AST ко множеству возможных значений её формальных аргументов в $\boldsymbol{M}$ добавляется $FromArg$. За это отвечает функция $AddFromArg$.
2. При обработке некоторых вершин (вызовов функций и присваиваний) дополнительно делается проверка, не является ли выполняемая в данной вершине операция AJAX-вызовом. За эту проверку отвечает функция $MatchPattern$, проверка делается на строке 16 алгоритма 4. Если операция распознана как AJAX-вызов, её аргументы вычисляются, и в набор

результатов работы алгоритма $\mathcal{C}$ добавляется запись с названием операции и массивом вычисленных значений формальных аргументов. За обработку найденного вызова отвечает функция $ProcessAJAXCall$.

3. После одного такого прохода по AST-дереву программы анализатор дополнительно производит *анализ цепочек вызова, ведущих к AJAX-вызовам*. За этот анализ цепочек вызовов отвечают функции $HandleFromArg$, $BuildCallChains$ и $EnterCall$.

Практически все AJAX-вызовы имеют вид вызовов функций, поэтому рассмотрим подробнее именно этот случай. Для вызова производится проверка, не является ли он вызовом функции, отправляющей запросы на сервер (*AJAX-функции*), следующими способами:

– По имени функции в месте вызова в коде (по синтаксическим сигнатурам). В анализатор встроен набор шаблонов, описывающих то, как обычно называются отправляющие запрос на сервер функции. К примеру, в этот шаблон входят такие имена, как `fetch`, `$.ajax` и `$http.post`.
– По значению функции, взятому для данной вершины из графа вызовов $\boldsymbol{CG}$. Для этого анализатором поддерживаются специальных функциональные значения, помеченные, как значений AJAX-функций. Они создаются для функций, чьё тело было опознано по синтаксической сигнатуре.

Функция $AddFromArg$ проходит по формальным аргументам обрабатываемой функции и добавляет к их значениям в $\boldsymbol{M}$ значение $FromArg$. То есть значения в $\boldsymbol{M}$ приводятся к типу $ValueSet$, и в него добавляется значение $FromArg$:

$$\frac{\begin{array}{cc}\mathcal{A}_{in} \vdash \langle \boldsymbol{M}, \boldsymbol{F}, \boldsymbol{C}, \boldsymbol{CG} \rangle & v \vdash function \ldots (a_1, \ldots, a_n)\ \{...\} \\ \alpha_1 = toset(\boldsymbol{M}(a_1)) \cup \{FromArg\} & \ldots \quad \alpha_n = toset(\boldsymbol{M}(a_n)) \cup \{FromArg\}\end{array}}{\mathcal{A}_{out} \vdash \langle \boldsymbol{M}[a_1 \mapsto \alpha_1, \ldots, a_n \mapsto \alpha_n], \boldsymbol{F}, \boldsymbol{C}, \boldsymbol{CG} \rangle}$$

Алгоритм 4: Алгоритм поиска AJAX-вызовов и их аргументов

Входные данные: AST, $\boldsymbol{M}$, $\boldsymbol{F}$, $\boldsymbol{C}$, $\boldsymbol{CG}$

Результат: $\mathcal{C}$ — набор найденных AJAX-вызовов и их аргументов

$\mathcal{A} \leftarrow \langle \boldsymbol{M}, \boldsymbol{F}, \boldsymbol{C}, \boldsymbol{CG} \rangle$
$\langle \mathcal{C}, \mathcal{Q}, \mathcal{A} \rangle \leftarrow FindRequestsAnalysisPass(AST, \mathcal{A}, [], [])$
while $|\mathcal{Q}| > 0$ **do**
    $\langle callerAST, chain \rangle \leftarrow$ *вытолкнуть первый элемент из* $\mathcal{Q}$
    $\langle \mathcal{C}', \mathcal{Q}, \mathcal{A} \rangle \leftarrow FindRequestsAnalysisPass(callerAST, \mathcal{A}, chain, \mathcal{Q})$
    $\mathcal{C} \leftarrow \mathcal{C} \cup \mathcal{C}'$

---

function $FindRequestsAnalysisPass(AST, \mathcal{A}, chain, \mathcal{Q})$
    $\mathcal{C} \leftarrow \varnothing$
    forall *vertex* v ***traversing*** AST ***in DFS order*** **do**
        if $isLibrary(v)$ **then**
            skip subtree and continue
        $\mathcal{A} \leftarrow ApplyEffect(v, \mathcal{A})$
        if $isFunction(v)$ **then**
            $\mathcal{A} \leftarrow AddFromArg(v, \mathcal{A})$
        if $\exists p \in AJAXPatternSet : MatchPattern(v, p, \mathcal{A})$ **then**
            $\langle \mathcal{C}', \mathcal{Q} \rangle \leftarrow ProcessAJAXCall(v, \mathcal{Q}, \mathcal{A})$
            $\mathcal{C} \leftarrow \mathcal{C} \cup \mathcal{C}'$
        if $|chain| > 0$ ***and*** $v \in \mathcal{V}_{call}$ ***and*** $chain[0].site = v$ **then**
            $\langle \mathcal{C}', \mathcal{Q}' \rangle \leftarrow EnterCall(chain, \mathcal{Q}, \mathcal{A})$
            $\mathcal{C} \leftarrow \mathcal{C} \cup \mathcal{C}'$
            $\mathcal{Q} \leftarrow \mathcal{Q}.concat(\mathcal{Q}')$
    return $\langle \mathcal{C}, \mathcal{Q}, \mathcal{A} \rangle$

---

function $ProcessAJAXCall(callSite, \mathcal{Q}, \mathcal{A})$
    $argValues \leftarrow [[\![E_1]\!]_{\mathcal{A}}, \ldots, [\![E_n]\!]_{\mathcal{A}}]$, *где* $[E_1, \ldots, E_n] = call.args$
    $\langle argValues, hadFromArg \rangle \leftarrow HandleFromArg(argValues)$
    $Result \leftarrow \{\langle p, argValues \rangle\}$
    if $hadFromArg$ **then**
        $\mathcal{Q} \leftarrow BuildCallChains(callSite, AST, chain, \mathcal{Q}, \mathcal{A})$
    return $\langle Result, \mathcal{Q} \rangle$

4.2.1 Анализ цепочек вызова, ведущих к AJAX-вызовам

После одного прохода по AST-дереву программы анализатор дополнительно производит нализ цепочек вызова, ведущих к AJAX-вызовам. Цепочки вызовов строятся для AJAX-вызовов, для которых обнаружено, что аргументы вызова зависят от формальных аргументов функции, содержащей этот AJAX-вызов. Будем называть эту содержащую AJAX-вызов функцию $caller$. Такую ситуацию можно определить по факту наличия значения $FromArg$ среди значений аргументов AJAX-вызова. Проверку аргументов на наличие среди них значения $FromArg$ осуществляет функция $HandleFromArg : Value[] \to \langle Value[], bool \rangle$. Эта функция принимает на вход массив значений аргументов и возвращает пару из следующих элементов:

– массива значений аргументов, в котором значения $FromArg$ заменены на $Unknown$,
– бинарного признака того, было ли среди аргументов найдено хотя бы одно значение $FromArg$ (другими словами, была ли произведена хотя бы одна замена).

Поиск и замена значения $FromArg$ производится с рекурсивным обходом в глубину структур данных (объектов и массивов), для $ValueSet$ также делается их поэлементный обход. Кроме того, функция $HandleFromArg$ проверяет все строки на наличие подстроки `"FROM_ARG"`. Как было упомянуто выше, значение $FromArg$ приводится к этой строке. В случае, если строка с такой подстрокой обнаружена, она будет заменена на `"UNKNOWN"`, а вторым элементом в паре значений, возвращаемой функцией $HandleFromArg$, будет $true$, то есть функция сигнализирует о том, что аргументы AJAX-вызова зависели от формальных аргументов вызывающей функции. В случае, если значение $FromArg$ было найдено среди значений аргументов, в графе вызовов $\boldsymbol{CG}$ будет выполнен поиск мест вызова функции $caller$.

Далее происходит анализ всех найденных мест вызова $caller$ — для каждого найденного места вызова выполняется ещё один проход $FindRequestsAnalysisPass$, однако, начиная не от корня AST-дерева всей программы, а от корня AST-дерева тела функции, содержащей найденное место вызова $caller$. Когда обход AST дойдёт до вызова $caller$, он перейдёт на начало её кода (на корень AST-дерева её кода) с подстановкой вычисленных фактический

Алгоритм 5: Функция построения цепочек вызова $BuildCallChains$

```
1  function BuildCallChains(v, chain, 𝒬, 𝒜)
2      if |chain| ⩾ MaxChain then
3          return 𝒬
4      caller ← fval_parent(callSite)
5      callerCallSites ← {v ∈ 𝒱_call | caller ∈ CG(v)}
6      forall site ∈ callerCallSites do
7          callInfo ← ⟨site, caller.ast⟩
8          chain ← [callInfo].concat(chain)
9          caller′ ← fval_parent(site)
10         if not alreadyProcessed(⟨caller′.ast, chain⟩) then
11             𝒬 ← 𝒬.append(⟨caller′.ast, chain⟩)
12     return 𝒬
```

значений аргументов, эмулируя её вызов. После чего обход AST дойдёт до интересующего AJAX-вызова, но уже с более точными значениями аргументов. Может случиться, что аргументы вызова $caller$ в свою очередь зависели от аргументов вызова функции, содержащей вызов $caller$ (назовём эту функцию $caller'$). Такая ситуация также может быть зафиксирована по наличию значения $FromArg$ среди аргументов AJAX-вызова. В этом случае будет произведён поиск мест вызова $caller'$, а затем — анализ цепочек уже из двух вызовов, и так далее. Алгоритм анализа предполагает ограничение на максимальную длину цепочек вызовов. Это настраиваемый параметр алгоритма, в текущей версии анализируются цепочки не длиннее пяти вызовов.

Этот механизм реализуется следующим образом. Будем называть *описанием вызова* пару $callInfo = \langle site, callee \rangle$ из места вызова $site \in \mathcal{V}_{call}$ и AST-вершины функции, вызываемой в этом месте $callee \in \mathcal{V}_{func}$. *Цепочкой вызова* будем называть упорядоченный список из описаний вызова $callChain = [callInfo_1, \ldots, callInfo_n]$. Алгоритм анализа поддерживает очередь цепочек вызова $\mathcal{Q}$ — упорядоченный список из пар $\langle firstFuncAST, callChain \rangle$. Первый элемент пары $firstFuncAST \in \mathcal{V}_{func}$ это функция, с тела которой нужно начать анализ цепочки, эта функция содержит первый вызов цепочки. Второй элемент пары $callChain$ это цепочка вызовов.

Если при обработке AJAX-вызова функция $HandleFromArg$ выявила, что среди аргументов есть значение $FromArg$, то принимается решение о том, что требуется анализ цепочек вызовов. Новые цепочки вызовов строятся функцией $BuildCallChains$ и добавляются в очередь $\mathcal{Q}$ (это делается на строке 30 алгоритма 4). Функция $BuildCallChains$ приведена в виде псевдокода (см. алгоритм 5). Она берёт функцию, содержащую AJAX-вызов, и с помощью графа вызовов $\boldsymbol{CG}$ находит все места, где она вызывается. Для каждого места $site$ из этих найденных мест вызова создаётся новая цепочка вызова, получаемая, как информация об этом вызове (т. е. пара $\langle site, caller.ast \rangle$), приписанная к началу уже имевшейся цепочки вызовов, анализ которой производился в тот момент. При первом вызове функции $FindRequestsAnalysisPass$ никакая цепочка не анализируется (“уже имеющаяся” цепочка пуста). Каждая созданная цепочка $chain$ добавляется в очередь $\mathcal{Q}$ в составе пары $\langle caller', chain \rangle$, где $caller'$ это AST-дерево функции, содержащей место вызова $site$. Функция $BuildCallChains$ не будет добавлять новый элемент в очередь $\mathcal{Q}$, если уже имеющаяся цепочка достигла максимальной длины, а также если такая пара $\langle caller', chain \rangle$ уже добавлялась в $\mathcal{Q}$ ранее.

После первого вызова $FindRequestsAnalysisPass$ алгоритм поиска AJAX-вызовов и их аргументов работает в цикле, извлекая из очереди $\mathcal{Q}$ по одному элементы, и обрабатывая их. Цикл продолжается до тех пор, пока очередь $\mathcal{Q}$ не опустеет (этот цикл находится на строке 3 алгоритма 4). Обработка элемента $\langle callerAST, chain \rangle$ на очередной итерации цикла заключается в вызове $FindRequestsAnalysisPass$, в который в качестве первого аргумента (AST-дерева, обход которого требуется сделать) передаётся $callerAST$, а в качестве третьего аргумента передаётся $chain$.

Функция $FindRequestsAnalysisPass$ поддерживает *анализ цепочек вызова*: в качестве третьего аргумента $chain$ она принимает цепочку вызова и, в ходе своей работы, переходит по вызовам этой цепочки. Точнее, если переданная цепочка $chain$ не пуста и при обходе AST встречается вызов функции $v \in \mathcal{V}_{call}$, делается проверка, не совпадает ли v с местом вызова $site$ из первого элемента $chain$ (эта проверка делается на строке 19 алгоритма 4). Если v совпадает с местом вызова из первого элемента цепочки, анализ переходит на начало функции $callee$ из первого элемента цепочки, эмулируя её вызов. За это отвечает функция $EnterCall$ (см. алгоритм 6). Она работает следующим образом. Фактические аргументы, которые передаются в вызов, вычисляются, и заносятся в $\boldsymbol{M}$ как

значения формальных аргументов вызываемой функции $callee$ (за это отвечает функция $setArgs$).

Алгоритм 6: Функция эмуляции вызова $EnterCall$

function $EnterCall(chain, \mathcal{Q}, \mathcal{A})$
- $\langle site, callee\rangle \leftarrow chain[0]$
- $rest \leftarrow chain[1\colon\ldots]$
- $\mathcal{A}' \leftarrow setArgs(callee.args, site, \mathcal{A})$
- $\langle \mathcal{C}', \mathcal{Q}'\rangle \leftarrow FindRequestsAnalysisPass(callee.body, \mathcal{A}', rest, \mathcal{Q})$
- **return** $\langle \mathcal{C}', \mathcal{Q}'\rangle$

function $setArgs(formalArgs, callSite, \mathcal{A})$
- $\langle \boldsymbol{M}, \boldsymbol{F}, \boldsymbol{C}, \boldsymbol{CG}\rangle \leftarrow \mathcal{A}$
- $[a_1, \ldots, a_n] \leftarrow formalArgs$
- $\{E_1, \ldots, E_n\} \leftarrow callSite.args$ // $callSite$ имеет вид $F(E_1, \ldots, E_n)$
- $\boldsymbol{M}' \leftarrow \boldsymbol{M}[a_1 \mapsto [\![E_1]\!]_{\mathcal{A}}, \ldots, a_n \mapsto [\![E_n]\!]_{\mathcal{A}}]$
- **return** $\langle \boldsymbol{M}', \boldsymbol{F}, \boldsymbol{C}, \boldsymbol{CG}\rangle$

После этого вызывается функция $FindRequestsAnalysisPass$, которая в качестве AST-дерева, обход которого она будет осуществлять, получает тело $callee$, а в качестве $chain$ ей передаётся оставшаяся часть анализируемой цепочки вызовов (у которой удалён первый элемент). Отметим, что функции $FindRequestsAnalysisPass$ передаётся в качестве первого аргумента именно вершина, соответствующая телу $callee$ (содержащая её код), но не вершина самой функции — поэтому задание значений её формальных аргументов, осуществляемое функциями $ApplyEffect$ и $AddFromArg$, осуществляться не будет — вместо этого в качестве значений её формальных аргументов в $\boldsymbol{M}$ будут оставаться заданные значения фактических аргументов, переданные в месте вызова. В случае, если оставшаяся часть цепочки не будет пуста, вызов $FindRequestsAnalysisPass$ может в дальнейшем, в свою очередь, сделать ещё один или несколько таких входов в вызовы функций. После завершения вызова $FindRequestsAnalysisPass$ функция $EnterCall$ также завершит работу, возвратив результат этого вызова.

Следует отметить, что описанный в данном разделе подход к межпроцедурному анализу — не единственный возможный. Альтернативой построению цепочек вызова и “прохождению” по ним может быть построение резюме

функций, более точных по сравнению с используемым в настоящей работе отображением $Args$. То есть сбор для каждой функции более точной информации, позволяющей определить возможные значения аргументов для различных контекстов вызова. Подходы, основанные на резюме функций (их также называют “function summary” или аннотациями функций), активно применяются в существующих работах, в том числе для задач компьютерной безопасности и поиска дефектов в программах [67—70]. Создание варианта алгоритма, использующей этот подход, и исследование свойств этого варианта является перспективным направлением дальнейшей работы.

4.3 Формирование спецификаций HTTP-запросов

Найденные вызовы со значениями аргументов преобразуются в набор спецификаций HTTP-запросов, отправляемых на сервер JavaScript-кодом страницы.

В качестве первого шага этого преобразования в наборе найденных вызовов $\mathcal{C}$ те элементы, у которых аргументы содержат множества $ValueSet$, заменяются на элементы с комбинациями множеств. В случаях, когда значения аргументов это объекты или массивы, множества могут содержаться внутри значения аргумента на определённой глубине: в качестве поля объекта или массива. Одно значение аргумента, в общем случае, может содержать несколько множеств. Все такие вложенные множества также учитываются при порождении комбинаций, для чего производится рекурсивный обход объектов и массивов в глубину. Также учитываются множества, которые могут присутствовать в разных аргументах. Пусть элементы массива аргументов $argValues$, входящего в пару $\langle p, argValues\rangle \in \mathcal{C}$, содержат множества $ValueSet$ (непосредственно или внутри объектов/массивов). Тогда вычисляется декартово произведение всех множеств, входящих в $argValues$, и для каждого кортежа декартова произведения порождается массив $argValues_i$, у которого на месте каждого из множеств подставлен соответствующий элемент из кортежа. Такие пары $\langle p, argValues_1\rangle, \ldots, \langle p, argValues_n\rangle$, соответствующие комбинациям множеств, записываются в $\mathcal{C}$ вместо $\langle p, argValues\rangle$.

Поскольку количество элементов, порождаемых комбинациями, может быть очень велико, вводится лимит на количество комбинаций, которые могут

быть порождены для одной пары $\langle p, argValues \rangle$, выданной алгоритмом из предыдущего раздела. Этот лимит является настраиваемым параметром алгоритма, в текущей версии реализации алгоритма используется лимит в 50 элементов.

После удаления элементов с множествами $ValueSet$ путём замен на комбинации элементы $\mathcal{C}$ преобразуются в спецификации HTTP-запросов. То, какие HTTP-запросы будут отправлены тем или иным вызовом с некоторыми аргументами, определяется с помощью набора эвристических *моделей функций, отправляющих запросы на сервер*. Они моделируют работу поддерживаемых библиотек и встроенных в браузер механизмов в части отправки запросов на сервер. То есть, для каждой AJAX-функции, для которой у анализатора есть *сигнатура вызова*, вручную написан код, который на основе аргументов этого вызова и исходного URL-адреса страницы определяет вид отправляемого запроса.

Формат, в котором анализатор выдаёт найденные спецификации HTTP-запросов, схож с форматом описания шаблонов запросов, описанным в разделе 2.4: каждая спецификация это JSON-объект, формат которого основан на формате описания запросов в HAR [48].

У него такой же набор полей, как и у объектов-шаблонов в эталонной разметке: `"method"`, `"url"`, `"queryString"`, `"headers"`, `"postData"`. Типы их значение тоже такие же. Поле `"postData"` может отсутствовать в объекте, выдаваемом анализатором, если у запроса отсутствует тело. Листинг 4.1 содержит пример спецификации запроса, выдаваемой анализатором.

При формировании итогового набора спецификаций, которые будут выданы анализатором, производится простая дедупликация: если несколько спецификаций полностью совпадают, с учётом всех имён и значений полей, будет оставлен только один экземпляр такой спецификации, остальные будут удалены.

Листинг 4.1: Пример спецификации запроса, выдаваемой анализатором

```
{
    "method": "POST",
    "url": "http://test.com/toys/search.aspx?sort=1",
    "headers": [{
            "name": "Host",
            "value": "test.com"
        },
        {
            "name": "Content-Type",
            "value": "application/x-www-form-urlencoded"
        },
        {
            "name": "Content-Length",
            "value": "10"
        }
    ],
    "queryString": [{
        "name": "sort",
        "value": "1"
    }],
    "postData": {
        "text": "query=lego",
        "mimeType": "application/x-www-form-urlencoded",
        "params": [{
            "name": "query",
            "value": "lego"
        }]
    }
}
```

4.4 Анализ закомментированного клиентского кода

Данный раздел посвящён улучшению алгоритма за счёт анализа закомментированного клиентского кода. Такой код в клиентской части веб-приложения может содержать полезную информацию о серверных точках входа, недоступную в “живом” коде клиентской части. В языке JavaScript, как и в любом другом современном языке программирования, есть возможность оставлять комментарии. Как правило, комментарии используют для написания пояснений к исходному тек-

сту программы, но это не всегда так. Нередко программисты неиспользуемую или устаревшую со временем часть кода вместо удаления прячут внутри блока комментариев. Поскольку клиентский код и серверный код исполняются изолированно друг от друга, то возможна такая ситуация, что часть клиентского кода окажется закомментированной, при этом с сервера соответствующая данному коду функциональная возможность удалена не будет.

```
var isGuest =true;

function goBackOnly()
{
    googleActionEvent('Eski Sürüm','Eskiye Dön', 'V1 e geçti');
    setCookie('backtothefuture', '2', 365);
    window.location.reload();

    //$.ajax({
    //    type: "POST",
    //    url: "/data/ReturnBackDeleteCookie",
    //    success: function (response) {
    //        if(response)
    //        {
    //            window.location.reload();
    //        }
    //    }
    //});
}

function readResponse(form, response) {
    form.find('.uyari').hide();
    if (response.HasError) {
        if (response.Message) {
            form.find('.uyari.kirmizi').html('<strong>Hata! </strong>' + response.Message).
```

Рисунок 4.2 — Пример закомментированного кода с сайта www.donanimhaber.com

Следует отметить, что анализ комментариев является достаточно сложным направлением по причине того, что сами комментарии в большей степени представляют собой текстовые блоки на естественном языке, не имеющие обязательного формата помимо синтаксических разделителей. Из этого можно сделать вывод, что все алгоритмические решения данной задачи будут носить эвристический характер. На рис. 4.2 и 4.3 представлены реальные примеры закомментированных вызовов, встречающиеся в сети Интернет.

Алгоритм извлечения закомментированного кода (алгоритм 7) получает на вход текст JavaScript-кода страницы. На первом этапе происходит удаление пустых строк (за это отвечает функция $RemoveBlankLines$). Они не влияют на синтаксическую корректность, но некоторые разработчики добавляют их для повышения читаемости кода. Поскольку на одном из следующих этапов необходимо будет объединить близлежащие однострочные комментарии в блочные, то пустые строки, встречающиеся между ними, могут повлиять на синтаксическую

```
    } else {
        alert('Не добавлено изображение или фото оттиска штампа!');
    }
}

/* function carouselAjax(type, count) {
    $.post(
            "/ajax.php",
            {
                method: 'carouselajax',
                type: type,
                count: count
            },
                function(data) {
                    // alert(data);
                    $('#' + type + '_carousel .slidered').append(data);

                    $("#ooo_carousel").jCarouselLite('reload', {
                        animation: 'slow'
                    });
            }
    );
} */
```

Рисунок 4.3 — Пример закомментированного кода с сайта master-stampov.ru

корректность уже объединенных блоков. Для наглядности приведен реальный пример (рис. 4.4), который показывает, что без удаления пустых строк после объединения будет получено несколько некорректных блоков с JavaScript-кодом вместо одного корректного.

```
// $(window).on('scroll', function () {
//  if ($('.load-more').offset().top <= $(window).scrollTop() + $(window).height()) {
//     console.log('bottom');
//     $.ajax({
//       type: "GET",
//       url: 'http://www.tmpo.co/ajax?tipe=beritaterkini&limit=10&start=10',
//       success: function (result) {
//         $(".divContent").append(result);

//         sIndex = sIndex + offSet;
//         isPreviousEventComplete = true;

//         if (result == '') //When data is not available
//             isDataAvailable = false;

//         $(".LoaderImage").css("display", "none");
//       },
//       error: function (error) {
//          alert(error);
//       }
//     });
//  }
//  });
```

Рисунок 4.4 — Пример кода с разделением однострочных комментариев пустыми строками.

Алгоритм 7: Алгоритм извлечения закомментированного кода

```
function ExtractCommentedCode(Scripts)
    ScriptsAST ← []
    forall script s ∈ Scripts do
        s ← RemoveBlankLines(s)
        mergedComments ← MergeComments(s)
        forall blockComment ∈ mergedComments do
            ScriptsAST.append(ParseComment(blockComment))
    return ScriptsAST
function ParseComment(BlockComment)
    while true do
        ast ← ParseCode(BlockComment)
        if ast = nil then
            BlockComment ← DeleteBadSymbol(BlockComment)
        else
            return ast
```

На втором этапе происходит синтаксический разбор кода страницы при помощи парсера JavaScript, предоставляемого библиотекой Babel [56]. Результатом синтаксического разбора является AST-дерево. Из полученного дерева извлекается массив закомментированных строк и блоков. После чего этот массив подается на вход функции $MergeComments$, которая объединяет находящиеся на соседних строках комментарии в блоки. После слияния комментариев начинается этап парсинга. Аналогично второму этапу, при помощи парсера Babel проводится синтаксический разбор каждого из полученных блоков. Если парсер сообщает об ошибке, то необходимо удалить вызывающий ее символ и повторить этап заново. Если по результатам некоторого числа итераций удается представить комментарий в виде AST-дерева, он будет добавлен в область для последующего анализа.

4.5 Особенности реализации метода

Данный раздел содержит особенности реализации метода, описанного в этой главе, а также некоторые эвристики, оказавшихся полезными для практического применения анализатора.

Анализатор реализован на языке TypeScript [71]. Программы на языке TypeScript транслируются в программы на JavaScript. Таким образом, скомпилированная версия анализатора представляет собой набор JavaScript-файлов. Она выполняется в среде NodeJS [72]. Реализация однопоточная, то есть при анализе одной веб-страницы используется только одно вычислительное ядро процессора. Тем не менее, выигрыш от наличия нескольких вычислительных ядер может быть получен за счёт анализа нескольких страниц параллельно (с независимым запуском нескольких процессов анализатора).

Тот факт, что анализатор, в конечном итоге, работает в реальном JavaScript интерпретаторе, используется для того, чтобы точно моделировать результаты операций с конкретными значениями примитивных типов JavaScript (такие, как приведение к строке или логическое отрицание) — они выполняются в реальном интерпретаторе V8 (интерпретаторе, используемом в среде выполнения NodeJS).

4.5.1 Использование управляемого браузера

Как уже было сказано в начале настоящей главы, для получения клиентского JavaScript-кода веб-страницы используется управляемый браузер Headless Chrome. Взаимодействие с браузером происходит посредством протокола CDP (Chrome DevTools Protocol) [73]. Для работы с этим протоколом реализация анализатора использует библиотеку Puppeteer [74].

Использование реального браузера позволяет осуществить загрузку страницы (включая подгрузку и выполнение клиентского JavaScript-кода) в условиях, приближенных к реальным. Осуществить это другими способами сложно из-за высокой сложности устройства веб-браузеров и клиентской части веб-приложений, а также многочисленных отклонений их реализаций от стандартов.

Протокол CDP предоставляет ряд функций, включая возможность перейти на заданный URL-адрес, подписаться на различные события, происходящие с веб-страницей. Как и в работе [17], активно используется программный интерфейс отладчика JavaScript-кода страницы (Debugger API). Доступ к Debugger API браузера Headless Chrome также может быть получен через протокол CDP. Через Debugger API анализатор подписывается на событие `scriptParsed`, оповещающее о том, что в интерпретатор JavaScript попал новый JavaScript-код. Это позволяет получать все скрипты страницы именно в том порядке, в котором они попадали в интерпретатор, вне зависимости от того, каким образом они туда попадали. Таким образом, анализатор может получить как код, который присутствовал на странице изначально или был загружен “традиционным” способом, с помощью тега `<script>` с атрибутом `src`, так и динамически загруженные или сформированные скрипты, которые могли быть переданы в интерпретатор через функции `eval` или `Function`. Это позволяет повысить полноту корпуса анализируемого кода, а также повысить точность анализа за счёт восстановления корректного порядка выполнения кода на странице.

Для получения кода таким образом нужно открыть страницу в управляемом браузере и дождаться её загрузки (получая выполняемые в ходе загрузки страницы скрипты). Однако, возникает вопрос — что именно означает “дождаться загрузки”, как зафиксировать момент, что страница загружена полностью? Хотя существует несколько событий жизненного цикла загрузки, сигнализирующих о различных этапах её загрузки, таких, как `DOMContentLoaded` [75] и `load` [76], ответить на этот вопрос точно не представляется возможным. Ничто не мешает JavaScript-коду страницы подгрузить ещё какие-то ресурсы в произвольный момент времени в будущем, уже после прихода события `load`, которое должно сигнализировать о “полной” загрузке страницы. Поэтому все решения задачи фиксации момента загрузки страницы будут носить эвристический характер, не являясь полностью точными (сама библиотека Puppeteer предоставляет несколько готовых вариантов таких эвристик). В реализации, созданной в рамках настоящей работы, используется следующий алгоритм определения момента окончания загрузки страницы:

1. Анализатор дожидается события `load`.
2. Анализатор дожидается момента, когда в течение 250 миллисекунд не будет ни одного запроса, ожидающего ответа и ни одного изменения в DOM-дереве страницы.

Для получения оповещений об изменениях в DOM-дереве страницы используется интерфейс MutationObserver [77], анализатор инициализирует его на странице и использует через Debugger API. Для получения количества запросов, ожидающих ответа, анализатор подписывается на событие отправки запроса со страницы и на событие получения ответа — такая возможность также предоставляется протоколом CDP. В общем случае, ожидание описанным методом может затянуться на неопределённо долгое время, поэтому введено максимально допустимое время ожидания, по прошествии которого безусловно считается, что страница загрузилась. В текущей реализации максимальное время ожидания составляет 2 минуты.

Большая часть кода, находящегося на странице, попадает в интерпретатор и начинает выполняться при загрузке страницы автоматически. Однако, это верно не для всего кода на странице. Атрибуты, имена которых начинаются на `on` (`onclick`, `onerror` и так далее) содержат код обработчиков событий [78]. Тем не менее, значения этих атрибутов не попадают в интерпретатор и не выполняются сразу при загрузке страницы — вместо этого их синтаксический разбор и выполнение производится в “ленивом” режиме. Как правило, они попадают в интерпретатор только при наступлении соответствующих событий. Однако, в ходе экспериментов было обнаружено, что спровоцировать попадание кода такого обработчика в интерпретатор можно, обратившись к соответствующей функции из JavaScript-кода. Дело в том, что любой такой обработчик можно считать в виде функционального значения из клиентского JavaScript-кода, обратившись к одноимённому свойству объекта DOM-элемента. То есть, если у DOM-элемента `el` есть атрибут с именем `onclick`, то при обращении к полю объекта `el.onclick` произойдёт синтаксический разбор и трансляция кода, записанного в значении атрибута. При этом отладчик сгенерирует событие `scriptParsed`, что позволит анализатору получить код из этого атрибута. Поэтому при загрузке страницы анализатор, используя интерфейс отладчика, генерирует обращения к свойствам DOM-элементов, соответствующим обработчикам событий (с именами, начинающимися на `on`), что позволяет получить код из всех обработчиков событий.

Хотя основное внимание в настоящей работе уделено получению информации об отправляемых запросах с помощью статического анализа, при реализации в анализатор также добавлены механизмы получения информации об отправляемых запросах из других источников:

– реальные наблюдаемые запросы, сделанные в ходе загрузки страницы;
– запросы, порождаемые HTML-разметкой.

Оба способа реализуются достаточно просто благодаря тому, что страница загружается в управляемом браузере Headless Chrome. Как уже упоминалось, протокол CDP позволяет получить информацию о всех отправляемых запросах. Debugger API позволяет выполнять JavaScript-код на странице, что позволяет обращаться к DOM API — благодаря чему нет необходимости осуществлять синтаксический разбор HTML-разметки, можно работать с программными объектами, созданными в результате разбора страницы браузером.

4.5.2 Реализация статического анализатора

При реализации алгоритма статического анализа, описанного в предыдущих разделах, было введено несколько ограничений, призванных сократить как потребление памяти, так и время работы. Уже упоминалось ограничение на количество комбинаций множеств, порождаемых на финальном этапе анализа, при формировании спецификаций отправляемых запросов — в текущей версии используется ограничение в 50 комбинаций. Кроме того, введено ограничение на количество элементов в множестве $ValueSet$: одно множество $ValueSet$ может содержать не больше 32 элементов. Также введено ограничение на длину строк, получаемую в результате вычисления выражений: строка может содержать не больше 10000 символов. Ограничение $MaxIter$ на число итераций анализа на первом этапе алгоритма (на число вызовов $AnalysisPass$) было установлено в максимум 6 итераций. Все описанные ограничения были установлены в такие значения по результатам экспериментов с реальным клиентским JavaScript-кодом.

После реализации основного алгоритма анализа, описанного в предыдущих разделах настоящей главы, и экспериментов с ним, было решено добавить ещё несколько доработок, направленных на повышение точности анализа. При поиске возвращаемого значения функции, помимо обращения к отображению R, хранящему глобально все возможные возвращаемые значения для каждой функции, было решено сделать механизм, который бы эмулировал вызовы функций с небольшой глубиной. При использовании этого механизма анализатор вычисляет фактические аргументы функции и переходит на начало её тела, делая проход по нему с использованием значений фактических аргументов, и собирая возможные возвращаемые значения при проходе через инструкции `return`. После заверше-

ния прохода возвращаемым значением функции считается $ValueSet$ из значений, собранных по инструкциям `return`. Если при таком проходе по функции анализатор обнаруживает место, где используется возвращаемое значение другой функции, он, в свою очередь, заходит и в ту функцию, если лимит глубины ещё не был достигнут. Если же лимит глубины был достигнут, возвращаемое значение берётся из R. В текущей реализации лимит глубины для этого механизма равен 2.

Отметим, что другим вариантом повышения точности определения возвращаемых значений было бы использование уже упоминавшегося метода резюме — использование более точной, по сравнению с используемым отображением R, собираемой информации о возвращаемых значениях функций, как это сделано в работах [68; 70]. Исследование такого способа уточнения определения возвращаемых значений представляется интересным направлением для дальнейшей работы.

Ещё одной доработкой, призванной повысить точность анализа, является различие объектов-экземпляров классов с точностью до мест аллокации. В базовой версии алгоритма, описанной выше, для каждого класса используется один объект-экземпляр, общий для всех мест аллокации. Информация о значениях полей экземпляров “смешивается” в этом общем объекте. В доработанной версии алгоритма для каждого места инстанциирования (то есть применения оператора `new`) создаётся отдельный экземпляр класса. Механизм эмуляции вызовов, упомянутый выше, доработан так, что он, помимо вызовов функций, работает для мест применения оператора `new`: анализатор “входит” в конструктор класса, чтобы проанализировать его действие на конкретный экземпляр, соответствующий этому месту инстанциирования. Кроме того, для каждого класса поддерживается “экземпляр по умолчанию”, используемый тогда, когда рекурсивный обход AST ($AnalysisPass$) посещает тело метода. “Экземпляр по умолчанию” особенно полезен для классов, которые вообще не инстанциируются в коде.

4.6 Апробация реализованного метода

Была проведена апробация реализованного метода с использованием составленного эталонного набора страниц, а также на наборе приложений, использовавшихся для сравнения в предыдущих работах.

4.6.1 Апробация на созданном эталонном наборе страниц

В качестве основной метрики качества работы алгоритма на созданном эталонном наборе страниц, описанном в разделе 2.4, было выбрано среднее покрытие. Для его вычисления для каждой страницы, входящей в набор данных считается покрытие (полнота) — то есть доля входящих в эталонную разметку страницы входных точек, которые были найдены анализатором.

$$Cov_i = \frac{|Endpoints_i^{true} \cap Endpoints_i^{found}|}{|Endpoints_i^{true}|}$$

Среднее покрытие это среднее арифметическое значений покрытия всех страниц.

$$Cov = \frac{\sum_{i=1}^{n} Cov_i}{n}$$

Здесь n — количество страниц в выборке, на данный момент это число равно 32.

Для вычисления Cov_i нужно иметь возможность находить множество $Endpoints_i^{true} \cap Endpoints_i^{found}$, то есть определять, какие из входных точек, найденных анализатором, совпадают со входными точками из эталонной разметки. Опишем алгоритм сопоставления спецификации запроса, выданных анализатором, с шаблоном запроса, из разметки эталонного набора веб-страниц, описанного в разделе 2.4.

1. Сравниваются значения поля `"method"`, при их несовпадении сравнение завершается с вердиктом *ложь*.
2. Сравниваются path-компоненты URL-адреса шаблона и проверяемой спецификации. Для этого path-компонента URL шаблона преобразуется в регулярное выражение, у которого все вхождения строки `"UNKNOWN"` заменены на `[^/]*` (регулярное выражение, допускающее всё, кроме символа `/`), после чего path-компонента URL-адреса спецификации сопоставляется с полученным регулярным выражением. В случае, если сопоставление не было успешным, сравнение завершается с вердиктом *ложь*.
3. Если у URL-адреса шаблона присутствует query-часть URL-адреса, то query-части сравниваются следующим образом:

а) для каждого параметра из query-части URL шаблона ищется параметр с таким же именем в query-части URL спецификации,
б) если параметр с таким именем не найден, то сравнение завершается с вердиктом *ложь*,
в) если параметр с совпадающим именем найден, но значения этих query-параметров сравниваются следующим образом:
1) если значение из шаблона это пустая строка или значение из шаблона совпадает со значением из спецификации, то сравнение значений считается успешным, алгоритм продолжает работу,
2) в противном случае сравнение завершается с вердиктом *ложь*.

4. Если у шаблона есть поле `"headers"`, то есть массив заголовков, то производится сравнение заголовков:
а) для каждого объекта, описывающего заголовок в шаблоне, проверяется, есть ли в массиве `"headers"` проверяемой спецификации объект с такими же значениями полей `"name"` и `"value"`,
б) если такой объект не нашёлся, то сравнение завершается с вердиктом *ложь*.
5. Если у шаблона присутствует поле `"postData"`, то проверяется, есть ли такое поле у проверяемой спецификации. Если у спецификации это поле отсутствует, то сравнение завершается с вердиктом *ложь*, иначе производится сравнение полей `"postData"` шаблона и спецификации.
6. Если сравнение полей `"postData"` шаблона и спецификации выявило несовпадение, то сравнение завершается с вердиктом *ложь*, иначе сравнение завершается с вердиктом *истина*, то есть считается, что спецификация совпадает с шаблоном.

Сравнение полей `"postData"` шаблона и спецификации (то есть объектов, описывающих тело запроса) производится по следующему алгоритму:

1. Если у объекта шаблона есть поле `"mimeType"`, то его значение сравнивается с полем `"mimeType"` объекта спецификации следующим образом:
а) если у объекта спецификации отсутствует поле `"mimeType"`, то сравнение завершается с вердиктом *ложь*,
б) если у значения `"mimeType"` из шаблона или значения из спецификации есть параметр `charset`, то есть строка содержит

подстроку `"; charset="`, то часть строки, включающая символ `;` и всё, что идёт после него, удаляется перед сравнением,

в) строки-значения `"mimeType"` проверяются на совпадение, и, если они не совпадают, сравнение завершается с вердиктом *ложь*.

2. если у объекта шаблона есть поле `"params"`, то проверяется, есть ли такое поле у объекта спецификации. Если у спецификации поле `"params"` отсутствует, то сравнение завершается с вердиктом *ложь*. В противном случае массив `"params"` шаблона и спецификации сравниваются так же, как сравниваются query-части URL, то есть для каждого объекта из `"params"` шаблона ищется совпадающий объект в `"params"` спецификации, при этом пустое значение в спецификации считается совпадающим с любым значением. В случае успешного сопоставления всё сравнение объектов `"postData"` завершается с вердиктом *истина*, в противном случае — с вердиктом *ложь*.
3. если у объекта шаблона поле `"params"` отсутствовало, производится сопоставление полей `"text"`. Оно делается следующим образом:
    а) если и у объекта шаблона и у объекта спецификации поле `"text"` отсутствует, сравнение объектов завершается с вердиктом *истина*,
    б) если у объекта шаблона есть поле `"text"`, а у спецификации его нет, то сравнение объектов завершается с вердиктом *ложь*,
    в) проверяется, имеет ли тело формат JSON — для этого проверяется, входит ли в значение `"mimeType"` шаблона подстрока `"json"`, и, если такая подстрока обнаружена, считается, что тело имеет формат JSON,
    г) если тело *не* имеет формат JSON, то результатом сравнения является результат сравнения значений полей `"text"` как строк (если у спецификации отсутствует поле `"text"`, то сравнение завершается с вердиктом *ложь*),
    д) если тело имеет формат JSON, то делается синтаксический разбор значения полей `"text"` шаблона и спецификации как данных в формате JSON, в случае ошибки разбора сравнение завершается с вердиктом *ложь*; если же разбор произошёл успешно, то результирующие объекты сравниваются следующим образом:

1) объекты сравниваются рекурсивно как структуры данных,
2) наборы ключей объектов должны совпадать,
3) порядок элементов в массивах может не совпадать, но набор элементов должен совпадать,
4) при сравнении значений примитивных типов, если значение из шаблона это `""`, `0` или `false`, то должен совпадать только тип значения из спецификации, но не обязано совпадать само значение (то есть пустая строка считается совпадающей с любой строкой, `0` считается совпадающим с любым числом, `false` считается совпадающим и с `false` и с `true`),

е) если ни на одном из предыдущих шагов сравнение не завершилось с каким-либо вердиктом, оно завершается с вердиктом *истина*, то есть считается, что объекты совпадают.

В код эталонного набора страниц (бенчмарка) входит программа `check.py` на языке Python, которая реализует приведённый алгоритм сравнения, и может быть использована для автоматического подсчёта метрики качества.

Среднее покрытие входных точек предложенным методом на эталонном наборе составило 42,6%, тогда как все протестированные статические анализаторы JavaScript-кода (WALA, SAFE и TAJS) не проанализировали ни одну страницу из набора — они либо завершаются с ошибкой, либо зависают. Причины пропуска входных точек были проанализированы, среди основных — использование при формировании запросов данных, считанных из DOM-модели, а также использование таких возможностей языка JavaScript, как наследование классов и применение встроенных в язык операторов `import` и `export`. Поддержку кода с этими особенностями планируется добавить в дальнейшей работе. При этом максимальное время выполнения на элементе набора составило 5 минут 3 секунды при работе на одном ядре процессора AMD EPYC 7502. Среднее время работы составило 1 минуту 2 секунды.

4.6.2 Эксперимент по сравнению с аналогами

Был проведён эксперимент по сравнению разработанного метода с существующими методами поиска серверных входных точек.

В настоящее время не существует какого-либо одного признанного бенчмарка или набора тестовых приложений в области тестирования безопасности приложений методом черного ящика — в разных статьях используются разные тестовые приложения. При этом есть такие тестовые приложения, которые используются в экспериментах в разных работах. Поэтому для получения результатов сравнения предложенного в этой работе метода с существующими средствами, которые позволяли бы делать обоснованные выводы, был составлен комбинированный набор приложений из наиболее часто встречающихся в научных публикациях. При составлении набора учитывалось наличие на страницах приложений JavaScript-кода, отправляющего запросы на сервер, кроме того, исходный код всех отобранных приложений открыт. Наличие открытого исходного кода позволяет авторам будущих исследований создавать тестовые стенды с использованием этих приложений и использовать их для сравнения. Кроме того, это даёт возможность изучить серверную часть приложения. Были выбраны следующие 5 приложений:

– DVWA [79];
– Juice Shop [80];
– MyBB [81];
– WebGoat [82];
– WIVET [83].

Приложения WIVET и MyBB использовались для сравнения в работах [6; 7; 25], кроме того, приложение WIVET также использовалось в сравнении в работе [5]. Приложение Juice Shop было использовано для сравнения в работах [4; 5]. Приложения DVWA и WebGoat были использованы для сравнения в работах [5; 84], кроме того, приложение DVWA использовалось в работе [85].

Разработанный прототип метода сравнивается со всеми инструментами, с которыми проводилось сравнение в работах [5] и [7], кроме инструментов Skipfish and jÄk. Skipfish для обнаружения серверных входных точек активно использует фаззинг и дирбастинг, вместо того чтобы проводить глубокий анализ клиентской части приложения. Таким образом, этот инструмент относится к другой категории

методов. Инструмент jÄk не удалось запустить из-за сложностей, которые были описаны в работах [4; 6; 7]: jÄk зависит от библиотеки PyQT5 версий 5.3 и 5.4, которые в данный момент не доступны для скачивания. Инструмент активно использует компонент QtWebKit библиотеки PyQT5, этот компонент отсутствует в более новых версиях этой библиотеки, которые в данный момент доступны.

Сравнение с недавними средствами Gelato [5] и CHIEv [7] произвести не удалось по следующим причинам. Исходный код обоих средств не опубликован. Автором настоящей работы была предпринята попытка связаться с авторами этих работ с просьбой предоставить данные, полученные в результате их экспериментов, на основе которых были составлены таблицы с результатами, приведённые в статьях. Авторы CHIEv не ответили на электронные письма. Авторы Gelato предоставили данные, полученные в ходе экспериментов. Однако, сравниться с результатами, приведёнными в статье [5], используя эти данные, не удалось: авторами [5] используется другой способ подсчёта серверных входных точек, нежели в настоящей работе. Запросы с разными значениями параметров, передаваемых в path-компоненте URL, считались авторами [5] как относящиеся к разным серверным входным точкам. В настоящей работе запросы, отличающиеся значениями параметров, передаваемых в path, но совпадающих в остальных частях (с совпадающей фиксированной частью) считаются как относящиеся к одной и той же серверной входной точке. Поскольку возможных значений параметров чрезвычайно много (даже в случае числовых параметров), при способе подсчёта, используемом в работе [5], можно получить сколь угодно большое количество найденных входных точек, что негативно влияет на интерпретируемость результатов.

Таким образом, сравнение производилось со следующими инструментами:

– Arachni [26];
– Crawljax [8];
– Enemy of the State (EoS) [23];
– Htcap [27];
– w3af [86];
– wget [87].

Средствам, участвующим в сравнении, не передавались никакие настройки авторизации (логины, пароли или сессионные идентификаторы для приложений). Приложения DVWA и WebGoat были установлены за прокси-сервером который автоматически добавлял сессионный идентификатор в запросы (cookie), так что

все инструменты сразу автоматически работали в авторизованной зоне. Это было сделано, поскольку все страницы этих приложений требуют авторизации, вне авторизованной зоны у них, по сути, ничего нет. Для приложений JuiceShop, MyBB и WIVET все средства работали без авторизации. Предельное время работы для всех средств было установлено в 6 часов (это совпадает с условиями тестирования, использовавшимися в работах [7] и [5]).

Поскольку метод, разработанный в рамках настоящей работы, работает над индивидуальными страницами, а не приложением целиком, для сравнения прототип был интегрирован со статическим краулером, реализованным с помощью фреймворка Colly[1]. Краулер производил обход приложения, находя все его страницы, затем разработанный прототип анализировал каждую из найденных страниц.

В качестве метрики качества использовалось количество найденных входных точек. Как отмечается в [5], априорная эталонная разметка для этих приложений (всех, кроме WIVET) отсутствует, и получить её весьма затруднительно. Для получения такой разметки можно было бы проанализировать серверный код приложений, однако, это достаточно трудоёмко, и не гарантирует отсутствие ошибок. Поэтому было решено использовать такой же подход, что и в работах [5] и [7]: для всех приложений, кроме WIVET, подсчитывается количество входных точек, найденных каждым из приложений, с помощью изучения сделанных ими на сервер запросов. Затем количества найденных входных точек сравниваются между собой. Определить то, обращаются ли два разных запроса к разным серверным входным точкам, или к одной и той же, нетривиально. Для проведения эксперимента эта работа делалась вручную, производился ручной осмотр запросов и выявление дубликатов (нескольких запросов, обращавшихся к одной и той же серверной входной точке).

Полные данные эксперимента, включая запросы, сделанные сравниваемыми средствами, и определённые на их основе входные точки, были опубликованы[2]. Эти данные могут быть использованы для перепроверки приведённых результатов эксперимента, как и для сравнения в рамках будущих экспериментов.

Результаты эксперимента приведены в таблице 2. Колонка *Всего* содержит количество входных точек в объединении множеств входных точек, найденных

[1] Colly: фреймворк для разработки веб-краулеров https://github.com/gocolly/colly

[2] https://github.com/asterite3/finding-endpoints-with-analysis-experiment-data

всеми инструментами. Колонка *Метод* содержит количество входных точек, найденных разработанным методом.

Таблица 2 — Уникальные входные точки, найденные каждым из инструментов

Название	*Всего*	Arachni	Crawljax	EoS	Htcap	w3af	wget	*Метод*
DVWA	57	36	43	40	**54**	48	1	**54**
JuiceShop	37	9	9	2	13	1	2	**36**
MyBB	110	68	7	65	63	59	52	**77**
WebGoat	78	**78**	10	1	10	1	10	12
WIVET	51	**51**	5	4	42	25	7	41

Как видно в таблице 2, разработанный метод обнаружил больше серверных входных точек, чем краулеры, в двух приложениях: JuiceShop и MyBB. Для одного из приложений, DVWA, количество найденных входных точек оказалось одинаковым у прототипа метода и краулера Htcap. На двух приложениях, WebGoat и WIVET, результаты разработанного прототипа метода оказались хуже. Причина, по которой описанный метод оказался лучше других средств на JuiceShop и MyBB, заключается в том, что JavaScript-кода на страницах этих приложений содержит обращения к серверным входным точкам, которые невозможно вызвать из пользовательского интерфейса, но которые смог обнаружить предлагаемый в данной работе алгоритм анализа.

Приложение JuiceShop относится к классу SPA, и у этого приложения вообще весь его клиентский JavaScript-код присутствует на каждой из его страниц, включая главную страницу. В том числе код, содержащий обращения ко всем серверным входным точкам этого приложения. В то же самое время, часть пользовательского интерфейса, вызывающего обращения к этим входным точкам, не показывается анонимному пользователю (пользователю, который не прошёл аутентификацию). Это касается, в том числе, запроса с методом `POST` и адресом `"/file-upload"`, который был обнаружен разработанным прототипом. Серверная входная точка, к которой обращается этот запрос, уязвима, причём запрос будет обработан сервером (и эксплуатация уязвимости возможна) даже без аутентификации. Что означает, что сканер уязвимостей, снабжённый описанным в настоящей работе анализатором, был бы способен найти и проэксплуатировать эту уязвимость, даже не имея учётных данных от приложения.

Приложение MyBB это форум. Сразу после установки на этом форуме нет ни одного обсуждения и ни одного поста. Отметим, что это те же условия

эксперимента, что и в работе [7], где отмечается, что никакой дополнительной подстройки приложения после установки не осуществлялось. Поскольку содержимое в форуме отсутствует, некоторые действия, такие как отмечание поста как прочитанного, или запрос автора поста, не могут быть осуществлены из пользовательского интерфейса. Соответствующие управляющие элементы, как правило, просто не появляются на страницах. Однако, соответствующие запросы обнаружены разработанным в настоящей работе анализатором. Среди клиентского JavaScript-кода этого приложения можно выделить файл `"general.js"`, содержащий несколько функций, обращающихся к упомянутым входным точкам, а также к серверным входным точкам, предназначенным для модераторов. Следует отметить, что некоторые из этих входных точек всё же были найдены одним из средств, с которыми производилось сравнение. W3af [86] смог обнаружить некоторые из входных точек, обращения к которым делались из мёртвого кода, поскольку этот сканер использует, в том числе, поиск по коду строк, похожих на URL-адреса, с помощью регулярных выражений. Тем не менее, даже для запросов с методом `GET`, которые этот инструмент обнаружил, некоторые параметры запросов не были найдены, поскольку они не присутствовали в URL-адресе изначально, а добавлялись к нему с помощью операции конкатенации строк. Входные точки, запросы к которым должны были отправляться с методом `POST`, этим средством найдены не были.

С точки зрения обращений к серверу клиентский код приложения DVWA достаточно простой, поэтому вполне ожидаемо, что результат разработанного прототипа сравнялся с результатом другого средства, найдя почти все входные точки.

Клиентский код приложения WebGoat имеет особенности, не поддерживаемые методом, описанным в данной работе. К особенностям, с которыми анализатору справиться не удалось, относится упаковщик модулей RequireJS, относящийся к типу “Asynchronous Module Definition”, а также некоторые механизмы JavaScript-фреймворка Backbone. И RequireJS и Backbone достаточно редки в современном Интернете: по статистике, опубликованной сайтом W^{3}Techs, RequireJS встречается на 0.6% веб-сайтов [88], а Backbone встречается на 0.8% веб-сайтов [89]. В соответствии с предложенным в работе подходом, для получения качественного решения в случае WebGoat требуется реализовать поддержку упаковщика RequireJS и библиотеки Backbone в анализаторе — но, так как их распространённость в настоящее время очень низкая, было решено отложить

это на будущее. В то же самое время, действиями в пользовательском интерфейсе можно вызвать обращения к ряду серверных входных точек этого приложения.

Что же до приложения WIVET, как отмечают в работе [7], некоторые из обращений к серверу со страниц этого приложения невозможно вызвать при работе пользователя с настоящим веб-браузером. Кроме того, в приложении присутствует входная точка, обращение к которой делается из Flash-анимации. Обнаружение таких входных точек выходит за рамки настоящей работы. Все серверные входные точки, обращения к которым делаются из клиентского JavaScript-кода, были найдены инструментом, разработанным в рамках данной диссертационной работы.

Для тестовых приложений DVWA, JuiceShop и WebGoat имеется информация о заложенных в них уязвимостях. Для них было оценено покрытие уязвимых входных точек сравниваемыми инструментами. Покрытие приведено в таблице 3.

Таблица 3 — Покрытие уязвимых входных точек

Название	Arachni	Crawljax	EoS	Htcap	w3af	wget	*Метод*
DVWA	28.6%	92.9%	50%	**100%**	71.4%	0%	**100%**
JuiceShop	16.7%	16.7%	0%	41.7%	0%	0%	**100%**
WebGoat	**11.9%**	0%	0%	0%	0%	0%	0%

Для приложений DVWA и JuiceShop разработанный метод обнаружил все уязвимые входные точки. Приложение DVWA содержит 2 обращения к уязвимым входным точкам из JavaScript-кода, оба из которых находятся в файле `"authbypass.js"`. Оба были найдены разработанным прототипом. Запросы к остальным уязвимым входным точкам DVWA инициируются элементами HTML-разметки. Средству Htcap также удалось обнаружить все уязвимые входные точки этого приложения.

Arachni оказался единственным инструментом, которому удалось выявить уязвимые входные точки WebGoat. Однако, покрытие показывает, что и для Arachni навигация в клиентской части WebGoat представляет сложность: из 42 уязвимых входных точек Arachni нашёл только 5.

Приложения MyBB и WIVET не содержат специально заложенных уязвимостей. Приложение MyBB это реальное приложение, информация о его уязвимостях отсутствует (предполагается, что их нет). Приложение WIVET тестовое, но оно предназначено именно для тестирования способности выявлять входные точки. Автор WIVET не заложил в него уязвимую функциональность.

Максимальное время работы разработанного прототипа на одной странице составило 28 секунд (такое время занял анализ страницы JuiceShop). Наибольшее время полного анализа всего приложения составило 5 минут и 2 секунды (такое время занял анализ приложения MyBB).

4.7 Эксперименты с поиском уязвимостей на реальных приложениях

В данном разделе описаны проведённые эксперименты с реализацией метода на сайтах в сети Интернет, в результате которых были обнаружены реальные уязвимости.

4.7.1 Уязвимости типа “Небезопасная десериализация” в запрашиваемых из JavaScript-кода входных точках

Был проведён эксперимент по поиску реальных уязвимостей в серверных входных точках, обнаруживаемых разработанным методом. Эксперимент проведён на примере класса уязвимостей “Небезопасная десериализация” (также известного как “Десериализация недоверенных данных”). Уязвимости данного класса достаточно распространены по сей день, наличие такой уязвимости может нести серьёзную угрозу безопасности сервера. Кроме того, этот класс интересен тем, что гипотезу о наличии уязвимости можно делать на основе обнаружения сериализованного представления объекта в клиентском коде приложения. Точнее, из наличия такого объекта в клиентском коде можно предположить, что объект будет отправлен на сервер и десериализован там. В случае если это так, факт наличия уязвимости будет зависеть от того, может ли атакующий произвольным образом менять отправленный объект, после чего он всё равно будет десериализован. Другими словами, применяются ли механизмы защиты целостности таких объектов. В случае, если найденный в клиентской части объект находился внутри JavaScript-кода, можно предполагать, что он будет отправлен из этого кода. То есть, что он принимается серверной входной точкой, обращения к которой делаются из клиентского JavaScript-кода.

Корпус кода для исследования

Эксперимент проведён на выборке реальных приложений. Были взята та же выборка из 50169 сайтов из списка Alexa, которая была описана в разделе 2.1. Как уже говорилось в разделе 2.1, список Alexa активно использовался в существующих научных работах [35; 39—43], размер выборки был выбран с учётом того, какие выборки использовались в существующих опубликованных исследованиях. Помимо 50169 сайтов Alexa, в выборку было также добавлено 20000 веб-приложений, для которых существуют программы Bug Bounty. Эти приложения представляют интерес, поскольку существование программы Bug Bounty для приложения говорит о том, что его владельцы уделяют внимание его безопасности, кроме того, для таких приложений возможна проверка на наличие уязвимости непосредственно на самом приложении (так как по условиям программ Bug Bounty разрешён поиск уязвимостей во входящих в программу приложениях). Таким образом, исследование проводилось на выборке, в которую входило всего около 70000 приложений.

Метод исследования

В исходном клиентском коде веб-страницы выявляются сериализованные объекты, в том числе закодированные одной или более кодировок. Для выявления на странице сериализованных объектов (в том числе, возможно, закодированных) используется метод, описанный в работе [15]. Предполагается, что, если сериализованный объект находится в клиентском коде, значит он должен быть отправлен на сервер и десериализован. В общем случае это не всегда так, в ходе исследования были обнаружены случаи, когда сериализованные объекты, присутствующие на странице, на самом деле не будут отправлены на сервер или не будут десериализованы после получения сервером. Тем не менее, будем в дальнейшем исходить из предположения, что наличие сериализованного объекта на клиентской стороне говорит о том, что скорее всего он будет отправлен на сервер, и в большей части выявленных классов (о них речь пойдёт ниже) это так. Далее, для JavaScript-кода, в котором использованы сериализованные объекты, составляют-

ся синтаксические шаблоны, которые содержат характерные признаки для кода. Шаблоны составляются вручную, их цель — выявлять аналогичное программное обеспечение. Шаблоны выявляют аналогичные фрагменты JavaScript-кода, в которых используется сериализованный объект (объект, который чаще всего либо отправляется этим кодом на сервер, либо сохраняется в какую-то переменную или структуру данных, откуда он будет считан и отправлен на сервер позднее). В большинстве случаев фрагменты кода, выявляемые шаблонами, представляют собой присваивания или вызовы функций. Характерными признаками, используемыми шаблонами, являются имена переменных, имена классов, имена полей объектов.

Затем, с помощью разработанного инструментального средства, полученные шаблоны выявляются в коде других веб-приложений. Для каждого шаблона выделяется класс приложений, на которых выявлен попадающий под этот шаблон код.

Примеры шаблонов приведены на листингах 4.2 и 4.3. Символ звёздочки (*) в конце идентификатора означает, что в конце идентификатора может идти произвольный суффикс (то есть что шаблон фиксирует не весь идентификатор, а только его начало). Выражение с вертикальными чертами (`a | b | c`) означает, что на его месте может стоять один из нескольких вариантов (разделённых символом `|`). На месте `SERIALIZED_OBJ` должна находиться строка, содержащая в себе сериализованный объект (как правило, закодированный с помощью кодировки base64, hex-кодировки или какой-либо ещё). `OBJ_WITH_SERIALIZED_OBJ_INSIDE` обозначает JavaScript-объект, в значениях полей которого содержится строка с сериализованным объектом (опять же, как правило, закодированным). Сигнатура на листинге 4.2 соответствует классу LoadMoreWordpress. Листинг 4.4 содержит фрагмент кода, соответствующий этому шаблону. А листинг 4.6 содержит более полный пример кода, содержащий этот фрагмент — пример кода приложения, относящегося к классу LoadMoreWordpress. Этот код отправляет на сервер запрос, в одном из параметров которого передаётся сериализованный объект.

Листинг 4.2: Шаблон класса LoadMoreWordPress

```
1 (true_posts* | ajax_query* | loadmore*) = SERIALIZED_OBJ |
    OBJ_WITH_SERIALIZED_OBJ_INSIDE
```

Сигнатура на листинге 4.3 соответствует классу JCCatalogBitrixOnlineShop. Листинг 4.5 содержит фрагмент кода, соответствующий этому шаблону.

Листинг 4.3: Шаблон класса JCCatalogBitrixOnlineShop

```
obbx_* = new JCCatalog*(OBJ_WITH_SERIALIZED_OBJ_INSIDE)
```

Листинг 4.4: Код, соответствующий шаблону класса LoadMoreWordPress

```
var true_posts = 'a:63:{s:14:"posts_per_page";i:10...; s :4:"DESC";}';
```

Листинг 4.5: Код, соответствующий шаблону класса JCCatalogBitrixOnlineShop

```
var obbx_117848907_56410 = new JCCatalogElement({
    'SKU_PROPS':'YTozOntpOjA7czo5OiJWWVNPVEFfTU0iO2k6MTtzOjc6IlNUT1JPT...,
    ...
});
```

Листинг 4.6: Пример кода одного из веб-приложений класса LoadMoreWordPress

```
var ajaxurl = '/wp-admin/admin-ajax.php ';
var true_posts = 'a:63:{s:14:"posts_per_page";i:10...; s :4:"DESC";}';
var current_page = 1;
var max_pages = '3 ';
...
$('#true_loadmore').click(function() {
    $(this).text('Loading ...');
    var data = {
        'action': 'loadmore',
        'query': true_posts,
        'page': current_page
    };
    $.ajax({
        url: ajaxurl,
        data: data,
        type: 'POST',
        success: function(data) { ... }
    });
});
```

Для представителей выделенных классов проверялось, отправляется ли сериализованный объект на сервер и десериализуется ли там. Если объект десериализуется, делалась проверка на наличие уязвимости.

Для валидации того, происходит ли на сервере десериализация объекта, использовались следующие методы:

– Отправка оригинального объекта, найденного на странице, а также некорректного, с точки зрения формата сериализации, объекта.
– Отправка измененного, но синтаксически корректного объекта. Например, добавление нового поля в словарь или изменение строкового литерала незначительным образом. Данная проверка нужна, чтобы отбросить случаи, когда на серверной части приложения сериализованный объект проходит проверку на строгое равенство с константной строкой.
– Отправка сериализованных объектов стандартных классов используемого языка программирования. Например, в случае с PHP, таковым являлся класс DateTime. Отправлялся корректный и некорректный сериализованный объект. Десериализация некорректного PHP-объекта DateTime приводит к фатальной ошибке сервера, что являлось индикатором происходящей на сервере десериализации.

При использовании всех методов выше сравнивались и анализировались HTTP-ответы, полученные на запросы с сериализованным объектом.

Для определения HTTP-запроса, отправляющего сериализованный объект на сервер, использовался метода анализа клиентского кода веб-приложения для обнаружения серверных входных точек, описанный в настоящей главе. Алгоритм анализа применялся к странице, на которой был обнаружен сериализованный объект. Найденный запрос отправлялся на сервер. В случае, если алгоритм не обнаруживал отправляемый HTTP-запрос с сериализованным объектом, а также если итоговый результат проверки был отрицательным (не был установлен факт десериализации объекта на сервере), дополнительно проводилась ручная проверка корректности работы алгоритма.

Проверка на наличие уязвимости выполнялась следующим образом. Для веб–приложений, входящих в программу Bug Bounty, производилась попытка эксплуатации уязвимостей на самих приложениях. Для остальных приложений производился поиск компонентов с открытым исходным кодом, на основе которых они реализованы. В случае нахождения таковых проверялась возможность эксплуатации уязвимости путем ручного анализа кода и атака на стендовые приложения, сделанные на их основе.

Результаты

В ходе исследования более 50% выявленных сериализованных объектов находились именно внутри JavaScript-кода (остальные сериализованные объекты чаще всего находились в атрибутах HTML-тегов).

Было выявлено 23 класса, каждый их которых содержит более одного приложения (ещё 37 приложений не были объединены в классы с другими). Для нескольких классов было установлено, что они “уязвимы”, то есть все приложения, относящиеся к ним, содержат уязвимость типа “Небезопасная десериализация”.

Класс *Settings*: было выявлено, что в данный класс входят приложения, сделанные на основе PHP–фреймворка OpenCart с использованием одного и того же плагина. В данном плагине присутствует недостаток небезопасной сериализации. Было выявлено 17 приложений, относящихся к данному классу.

– Была найдена цепочка гаджетов с использованием классов базового фреймворка, позволяющая выполнить произвольный код на сервере.
– Данная уязвимость не была публично известной, поэтому разработчики продукта были оповещены о наличии данной уязвимости, а также был получен идентификатор уязвимости CVE-2022-24108[3].

Класс *SimpleAjaxManagerWordpress*: было выявлено, что приложения данного класса используют CMS WordPress (является одной из самых распространенных CMS в сети Интернет) и плагин Simple Ajax Manager. Было выявлено 7 приложений, относящихся к данному классу.

– Были найдены две уязвимости: небезопасная десериализация и SQL-инъекция через поля сериализованного объекта.
– Найденная SQL-инъекция не была публично известной уязвимостью, поэтому был отправлен запрос на получение идентификатора CVE в организацию WPScan. Уязвимость не получила нового идентификатора уязвимости, но информация о ней была добавлена в существующую запись о недостатке “Десериализации недоверенных данных”[4].

[3]CVE-2022-24108 (https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-24108)

[4]Wordpress: Simple Ads Manager vulnerability page (https://wpscan.com/vulnerability/38787b49-c19c-49db-925a-69e2c9cf7a43)

Класс *LoadMoreWordpress*: приложения данного класса используют CMS WordPress, но технология, где используется небезопасная десериализация, не является плагином. Код, где выполняется уязвимый запрос, был приведен на одном из форумов для разработчиков. Было выявлено 65 приложений, относящихся к данному классу.

– При помощи эксплуатации небезопасной десериализации получилось добиться выполнения произвольного кода на серверной стороне для всех этих приложений.
– Был отправлен отчет об уязвимости на один из главных агрегаторов Bug Bounty программ HackerOne.

Для всех упомянутых “уязвимых” классов анализ клиентского JavaScript-кода, предложенный в данной работе, успешно определил правильный вид HTTP-запроса, отправляющего сериализованный объект на сервер.

4.7.2 Эксперименты с закомментированным кодом

Был проведён эксперимент с использованием модуля анализа закомментированного кода, представленного в разделе 4.4. Была исследована гипотеза о том, что закомментированный JavaScript-код клиентской части веб-приложения может содержать полезную информацию о серверных точках входа, недоступную в “живом” коде клиентской части. В рамках экспериментов оценивалась распространённость закомментированного JavaScript-кода в реальных веб-приложениях, а также проводился анализ обнаруженных в них серверных точек входа, в том числе на наличие уязвимостей.

Для проведения эксперимента была использована выборка реальных приложений, по своим параметрам аналогичная той, которая использовалась в разделах 2.1 и 4.7.1. Выборка состоит из 50291 сайта, взятого из списка Alexa Top 1 Million. Как уже упоминалось выше, список Alexa часто использовался в существующих работах в качестве списка реальных приложений, на приложениях из которого проводилось исследование [35; 39—43]. При выборе размера выборки было решено ориентироваться на размеры выборок, использованных в существующих работах. Всего проанализировано 580056 страниц, относящихся к веб-сайтам из выборки.

В рамках эксперимента анализатор запускался на локальных копиях страниц в двух режимах: с включенной опцией анализа закомментированного кода и без неё. Результаты каждого из запусков сохранялись независимо друг от друга. Следующим этапом необходимо было выделить запросы, являющиеся уникальными для закомментированного кода. Сама по себе задача, какие запросы считать уникальными, является нетривиальной. Значения параметров могут находиться в path-компоненте URL. В таком случае, пара запросов, отличающихся значением этого параметра, скорее всего, будут обработаны сервером одинаково. С другой стороны, возможно использование параметров, находящихся в query-части URL и при этом отвечающих за маршрутизацию в приложении. По этой причине даже идентичные запросы, имеющие разные значения такого параметра, могут обрабатываться сервером по-разному. В рамках описываемого эксперимента было решено считать одинаковыми запросы, у которых совпадают методы, path-компоненты URL, наборы параметров в query-части URL и/или в теле запроса, значения заголовков Content-Type и Host и значения параметров. Если значения одноименных параметров расходятся, но относятся к числовому типу, то такие запросы также считаются одинаковыми.

Далее было проведено сравнение результатов двух запусков анализатора по следующему алгоритму. Для каждого запроса с пометкой о том, что он был найден в закомментированном коде, проводился поиск аналогичного запроса из результатов запуска анализатора без включенного модуля на той же странице. Если соответствующий запрос не был найден, то закомментированный вызов считался уникальным и попадал в финальную выборку.

Следующим шагом выполнялась проверка существования соответствующих конечных точек на сервере для найденных запросов с использованием описанной далее процедуры. Все закомментированные вызовы дедуплицировались по описанному ранее алгоритму уже между собой, после чего подсчитывалось общее число уникальных вызовов. Затем по каждому из них формировался и отправлялся на сервер запрос. Сразу после этого отправлялся аналогичный запрос на новый адрес, который получался добавлением в компонент path оригинального URL строки, сгенерированной из случайных символов. Полученный таким образом адрес, скорее всего, обрабатывался сервером как несуществующий. Далее сравнивались коды ответов. Если они одинаковы, то с большой вероятностью проверяемая конечная точка уже отключена. Если коды ответов разные, то, скорее всего, с сервера еще не была удалена соответствующая

функциональная возможность — такие запросы будем называть *успешно выявленными*. В свою очередь, по коду ответа можно делать предположения о том, насколько точно был выявлен запрос. Нормальный код ответа (начинающийся на 2 или 3) может говорить о полностью найденном корректном запросе к серверу. Ошибочный код ответа может указывать на наличие параметров, которые анализатором не были найдены или были найдены некорректно. Вне зависимости от кода ответа в дальнейшем найденную конечную точку можно анализировать на предмет уязвимостей.

На последнем этапе был выполнен сбор статистики по категориям точности среди всех найденных уникальных запросов, а также оценено процентное соотношение успешно найденных запросов ко всем. Было выделено три категории точности выявления запросов. Первая категория — полностью известные: в нее входят запросы, у которых анализатор смог определить URL и все параметры с их значениями. Вторая категория включает в себя запросы, у которых полностью известны URL и имена параметров, но какие-то значения параметров оказались не определены. К третьей категории были отнесены все остальные запросы.

Результаты эксперимента

Всего проанализировано 580056 страниц, относящихся к 50291 веб-приложению.

- AJAX-вызовы, встречающиеся только в закомментированном коде, оказались на 9948 страницах (1,72% от общего количества) или в 1396 веб-приложениях (2,78% от общего количества).
- Всего найдено 14395 закомментированных вызовов. Из них оказалось 3923 уникальных (27,25%).
- Соответствующие конечные точки на сервере имели 1536 вызовов (39,15% среди уникальных), относящиеся к 1232 ресурсам (0,21% от общего количества) или 739 различным веб-приложениям (около 1,47% от общего количества).

Большая часть успешно выявленных запросов имела коды ответов, равные 200, 400, 403, 500, а процент ответов с кодом 404 минимален, что свидетельствует в пользу верности выдвинутых предположений для проверки наличия конечной

точки на сервере. Статистика по успешно найденным запросам в зависимости от того, насколько точно анализатору удалось определить вид отправляемый запрос, выглядит следующим образом:

- Среди 1961 полностью известного запроса 728 имеют конечную точку на сервере, что соответствует 37,12%;
- Из 1682 запросов с неизвестными значениями параметров 773 были успешно выявлены, что составляет 45,96%;
- Запросов с неизвестными именами параметров или частью URL было найдено 280, среди которых 35 (12,5%) имеют соответствующую функциональность на сервере.

Уязвимости

Закомментированные запросы с оставленной функциональностью на сервере были исследованы на наличие уязвимостей. Из них 5,6% оказались уязвимы. Самые часто встречающиеся виды уязвимостей – Reflected XSS и SQL Injection. Кроме того, были найдены уязвимости типа PHP code injection и некоторых других типов.

Выводы

Экспериментальное исследование на приложениях из рейтинга Alexa Top 1 Million позволяет сделать вывод, что закомментированные запросы с оставленной функциональностью на сервере действительно встречаются в реальном мире. В проведенном эксперименте они нашлись примерно в 2,78% от исследованных веб-приложений, более того, около 40% из них имеют соответствующую “живую” входную точку входа на сервере. Выборочный анализ найденных входных точек, что подобные запросы нередко оказываются уязвимыми: в “забытых” серверных API были найдены такие уязвимости, как Reflected XSS, SQL Injection, PHP code injection и другие. Разработанный модуль можно использовать в составе скане-

ра веб-безопасности для получения более полной информации о серверной части веб-приложения методом “чёрного ящика”.

Заключение

Основные результаты работы заключаются в следующем.

1. Проведено исследование особенностей реального JavaScript-кода, влияющих на возможность его анализа с целью обнаружения серверных входных точек, на наборе популярных приложений из рейтинга Alexa Top 1 Million. Выделены наиболее существенные особенности, в том числе использование упаковщиков модулей, непрямые объявления классов, обращения к полям объектов по вычисленному имени. С учетом выявленных особенностей поставлена задача разработки специализированного статического анализа и сформулированы требования к нему. На основе исследованных приложений составлен эталонный набор веб-страниц, позволяющий оценивать эффективность методов извлечения информации о серверных входных точках из клиентского кода.
2. Предложена новая методика поиска уязвимостей веб-приложений в модели “чёрного ящика”. Методика отличается от существующих повышением покрытия серверных входных точек, обращения к которым сложно вызвать через взаимодействие с пользовательским интерфейсом, за счет применения статического анализа и автоматизированного перебора имён параметров запроса, а также применением анализа кода клиентской стороны веб-приложения для обнаружения уязвимостей.
3. Разработан и реализован метод анализа клиентского кода веб-приложения для обнаружения серверных входных точек с целью последующего поиска уязвимостей. Метод использует алгоритм статического анализа, разработанный с учётом выявленных особенностей реального кода и способный анализировать весь код страницы, включая недостижимый код. Алгоритм позволяет решать задачу обнаружения входных точек в заданных ограничениях по времени и по использованию вычислительных ресурсов, что достигается за счёт ограничения числа проходов по коду и добавления встроенной поддержки популярных JavaScript-библиотек, что позволяет не анализировать их собственный код.
4. Проведена апробация реализованного метода на наборе приложений, использовавшихся для сравнения в других работах, и на составленном

эталонном набора страниц. На трех приложениях из пяти разработанный метод выявил максимальное количество серверных входных точек (при этом в двух приложениях все проверенные аналоги нашли меньше входных точек). Среднее покрытие входных точек на эталонном наборе составило 42,6%, а максимальное время выполнения на одном приложении — 5 минут 3 секунды при работе на одном ядре процессора AMD EPYC 7502. Проведены эксперименты с сайтами из сети Интернет, в результате которых на них были обнаружены реальные уязвимости.

Перспективными направлениями дальнейшей работы представляются применение метода резюме функций для межпроцедурного анализа (для более точного определения возвращаемых значений и значений аргументов), а также интеграция статического и динамического анализа, которая могла бы позволить использовать при анализе информацию о выполнении программы.

Список литературы

1. Отчёт 2023 Data Breach Investigations Report от компании Verizon [Электронный ресурс]. — URL: https://www.verizon.com/business/resources/reports/dbir/ (дата обр. 01.08.2024).
2. Web application security assessment by fault injection and behavior monitoring / Y.-W. Huang [и др.] // Proceedings of the 12th international conference on World Wide Web. — 2003. — С. 148—159.
3. Usage statistics of client-side programming languages for websites [Электронный ресурс]. — URL: https://w3techs.com/technologies/overview/client_side_language (дата обр. 01.08.2024).
4. Experience: Model-Based, Feedback-Driven, Greybox Web Fuzzing with BackREST / F. Gauthier [и др.] // 36th European Conference on Object-Oriented Programming (ECOOP 2022). Т. 222 / под ред. K. Ali, J. Vitek. — Dagstuhl, Germany : Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. — 29:1—29:30. — (Leibniz International Proceedings in Informatics (LIPIcs)). — URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2022.29.
5. *Hassanshahi, B.* Gelato: Feedback-driven and Guided Security Analysis of Client-side Web Applications / B. Hassanshahi, H. Lee, P. Krishnan // 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER). — 2022. — С. 618—629.
6. *Leithner, M.* XIEv: Dynamic Analysis for Crawling and Modeling of Web Applications / M. Leithner, D. E. Simos // Proceedings of the 35th Annual ACM Symposium on Applied Computing. — Brno, Czech Republic : Association for Computing Machinery, 2020. — С. 2201—2210. — (SAC '20). — URL: https://doi.org/10.1145/3341105.3373885.
7. *Leithner, M.* CHIEv: Concurrent Hybrid Analysis for Crawling and Modeling of Web Applications / M. Leithner, D. E. Simos // SIGAPP Appl. Comput. Rev. — New York, NY, USA, 2021. — Июль. — Т. 21, № 1. — С. 5—23. — URL: https://doi.org/10.1145/3477133.3477134.

8. *Mesbah, A.* Crawling Ajax-Based Web Applications through Dynamic Analysis of User Interface State Changes / A. Mesbah, A. van Deursen, S. Lenselink // ACM Trans. Web. — New York, NY, USA, 2012. — Март. — Т. 6, № 1. — URL: https://doi.org/10.1145/2109205.2109208.

9. *Свидетельство о регистрации прав на ПО, базу данных*. Система автоматического поиска уязвимостей в веб-приложениях на основе обработки больших данных / Д. Ю. Гамаюнов [и др.] ; МГУ имени М.В.Ломоносова. — № 2022610972 ; заявл. 18.01.2022 ; опубл. 18.01.2022, 2022610972 (Рос. Федерация).

10. *Sun, K.* Analysis of JavaScript programs: Challenges and research trends / K. Sun, S. Ryu // ACM Computing Surveys (CSUR). — 2017. — Т. 50, № 4. — С. 1—34.

11. *Ryu, S.* Toward Analysis and Bug Finding in JavaScript Web Applications in the Wild / S. Ryu, J. Park, J. Park // IEEE Software. — 2019. — Т. 36, № 3. — С. 74—82.

12. *Andreasen, E.* Determinacy in Static Analysis for JQuery / E. Andreasen, A. Møller // Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications. — Portland, Oregon, USA : Association for Computing Machinery, 2014. — С. 17—31. — (OOPSLA '14). — URL: https://doi.org/10.1145/2660193.2660214.

13. *Сигалов, Д. А.* Обнаружение серверных точек взаимодействия в веб-приложениях на основе анализа клиентского JavaScript-кода / Д. А. Сигалов, А. А. Хашаев, Д. Ю. Гамаюнов // Прикладная дискретная математика. — 2021. — № 53. — С. 32—54.

14. *Назаров, Д. И.* Поиск информации о принимаемых сервером запросах в закомментированном клиентском коде веб-приложений / Д. И. Назаров, Д. А. Сигалов, Д. Ю. Гамаюнов // Программная инженерия. — 2023. — Т. 14, № 5. — С. 245—253.

15. *Миронов, Д. Д.* Исследование встречаемости небезопасно сериализованных программных объектов в клиентском коде веб-приложений / Д. Д. Миронов, Д. А. Сигалов, М. П. Мальков // Труды Института системного программирования РАН. — 2023. — Т. 35, № 1. — С. 223—236.

16. *Sigalov, D.* Dead or alive: Discovering server HTTP endpoints in both reachable and dead client-side code / D. Sigalov, D. Gamayunov // Journal of Information Security and Applications. — 2024. — Vol. 82. — P. 103746. — URL: https://www.sciencedirect.com/science/article/pii/S2214212624000498.

17. *Сигалов, Д. А.* Использование отладочного API современного веб-обозревателя для обнаружения уязвимостей класса DOM-based XSS / Д. А. Сигалов, А. В. Раздобаров, А. А. Петухов // Прикладная дискретная математика. — 2017. — № 35. — С. 63—75.

18. *Sigalov, D.* Finding Server-Side Endpoints with Static Analysis of Client-Side JavaScript / D. Sigalov, D. Gamayunov // Computer Security. ESORICS 2023 International Workshops. — Springer Nature Switzerland, 2024. — P. 442—458.

19. *Раздобаров, А.* Проблемы обнаружения уязвимостей в современных веб-приложениях / А. Раздобаров, А. Петухов, Д. Гамаюнов // Проблемы информационной безопасности. Компьютерные системы. — 2015. — № 4. — С. 64—69.

20. *Doupé, A.* Why Johnny Can't Pentest: An Analysis of Black-Box Web Vulnerability Scanners / A. Doupé, M. Cova, G. Vigna // Detection of Intrusions and Malware, and Vulnerability Assessment / под ред. C. Kreibich, M. Jahnke. — Berlin, Heidelberg : Springer Berlin Heidelberg, 2010. — С. 111—131.

21. Improving the effectiveness of web application vulnerability scanning / M. Rennhard [et al.] // International Journal on Advances in Internet Technology. — 2019. — July. — Vol. 12, no. 1/2. — P. 12—27. — URL: https://digitalcollection.zhaw.ch/handle/11475/17956.

22. Ajax Search Lite plugin for WordPress (version 4.11.2) [Электронный ресурс]. — URL: https://wordpress.org/plugins/ajax-search-lite/ (дата обр. 01.07.2023).

23. Enemy of the State: A State-Aware Black-Box Web Vulnerability Scanner / A. Doupé [и др.] // Proceedings of the 21st USENIX Conference on Security Symposium. — Bellevue, WA : USENIX Association, 2012. — С. 26. — (Security'12).

24. *Fard, A. M.* Feedback-directed exploration of web applications to derive test models / A. M. Fard, A. Mesbah // 2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE). — 2013. — С. 278—287.

25. jÄk: Using Dynamic Analysis to Crawl and Test Modern Web Applications / G. Pellegrino [и др.] // Research in Attacks, Intrusions, and Defenses / под ред. H. Bos, F. Monrose, G. Blanc. — Cham : Springer International Publishing, 2015. — С. 295—316.

26. Arachni Framework version 1.6.1.3-0.6.1.1 [Электронный ресурс]. — URL: https://github.com/Arachni/arachni (дата обр. 01.08.2024).

27. Htcap 1.1.0 [Электронный ресурс]. — URL: https://github.com/fcavallarin/htcap (дата обр. 01.07.2023).

28. *Eriksson, B.* Black Widow: Blackbox Data-driven Web Scanning / B. Eriksson, G. Pellegrino, A. Sabelfeld // 2021 IEEE Symposium on Security and Privacy (SP). — 2021. — С. 1125—1142.

29. JSForce: A Forced Execution Engine for Malicious JavaScript Detection / X. Hu [и др.] // Security and Privacy in Communication Networks / под ред. X. Lin [и др.]. — Cham : Springer International Publishing, 2018. — С. 704—720.

30. Finding Client-Side Business Flow Tampering Vulnerabilities / I. L. Kim [и др.] // Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. — Seoul, South Korea : Association for Computing Machinery, 2020. — С. 222—233. — (ICSE '20). — URL: https://doi.org/10.1145/3377811.3380355.

31. The T. J. Watson Libraries for Analysis (WALA) [Электронный ресурс]. — URL: https://github.com/wala/WALA (дата обр. 01.08.2024).

32. SAFE: Formal specification and implementation of a scalable analysis framework for ECMAScript / H. Lee [и др.] // FOOL 2012: 19th International Workshop on Foundations of Object-Oriented Languages. — Citeseer. 2012. — С. 96.

33. *Jensen, S. H.* Type Analysis for JavaScript / S. H. Jensen, A. Møller, P. Thiemann // Static Analysis / под ред. J. Palsberg, Z. Su. — Berlin, Heidelberg : Springer Berlin Heidelberg, 2009. — С. 238—255.

34. Practically Tunable Static Analysis Framework for Large-Scale JavaScript Applications (T) / Y. Ko [и др.] // 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE). — 2015. — С. 541—551.

35. An Analysis of the Dynamic Behavior of JavaScript Programs / G. Richards [и др.] // Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation. — Toronto, Ontario, Canada : Association for Computing Machinery, 2010. — С. 1—12. — (PLDI ’10). — URL: https://doi.org/10.1145/1806596.1806598.

36. Usage statistics of client-side programming languages for websites [Электронный ресурс]. — URL: https://w3techs.com/technologies/overview/javascript_library (дата обр. 01.08.2024).

37. *Guha, A.* Using Static Analysis for Ajax Intrusion Detection / A. Guha, S. Krishnamurthi, T. Jim // Proceedings of the 18th International Conference on World Wide Web. — Madrid, Spain : Association for Computing Machinery, 2009. — С. 561—570. — (WWW ’09). — URL: https://doi.org/10.1145/1526709.1526785.

38. Statically Checking Web API Requests in JavaScript / E. Wittern [и др.] // 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE). — 2017. — С. 244—254.

39. The Eval That Men Do / G. Richards [и др.] // ECOOP 2011 – Object-Oriented Programming / под ред. M. Mezini. — Berlin, Heidelberg : Springer Berlin Heidelberg, 2011. — С. 52—78.

40. *Jensen, S. H.* Remedying the eval that men do / S. H. Jensen, P. A. Jonsson, A. Møller // Proceedings of the 2012 International Symposium on Software Testing and Analysis. — Minneapolis, MN, USA : Association for Computing Machinery, 2012. — С. 34—44. — (ISSTA 2012). — URL: https://doi.org/10.1145/2338965.2336758.

41. *Ratanaworabhan, P.* JSMeter: comparing the behavior of JavaScript benchmarks with real web applications / P. Ratanaworabhan, B. Livshits, B. G. Zorn // Proceedings of the 2010 USENIX Conference on Web Application Development. — Boston, MA : USENIX Association, 2010. — С. 3. — (WebApps’10).

42. Time Present and Time Past: Analyzing the Evolution of JavaScript Code in the Wild / D. Mitropoulos [и др.] // 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR). — 2019. — С. 126—137.

43. *Jueckstock, J.* VisibleV8: In-browser Monitoring of JavaScript in the Wild / J. Jueckstock, A. Kapravelos // Proceedings of the Internet Measurement Conference. — Amsterdam, Netherlands : Association for Computing Machinery, 2019. — С. 393—405. — (IMC '19). — URL: https://doi.org/10.1145/3355369.3355599.

44. Colly Crawler Framework for Golang [Электронный ресурс]. — URL: https://github.com/gocolly/colly (дата обр. 25.04.2024).

45. ECMAScript 2015 Language Specification [Электронный ресурс]. — URL: https://262.ecma-international.org/6.0/ (дата обр. 19.08.2024).

46. "import" declaration — MDN Web Docs [Электронный ресурс]. — URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/import (дата обр. 19.08.2024).

47. "class" declaration — MDN Web Docs [Электронный ресурс]. — URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/class (дата обр. 19.08.2024).

48. HTTP Archive (HAR) format [Электронный ресурс]. — URL: https://w3c.github.io/web-performance/specs/HAR/Overview.html (дата обр. 30.04.2022).

49. MIME types — MDN Web Docs [Электронный ресурс]. — URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/MIME_types (дата обр. 03.09.2024).

50. Раздел Web Application Security Testing методики OWASP Web Security Testing Guide версии 4.2 [Электронный ресурс]. — URL: https://owasp.org/www-project-web-security-testing-guide/v42/4-Web_Application_Security_Testing/ (дата обр. 03.07.2024).

51. *Kang, Z.* Probe the Proto: Measuring Client-Side Prototype Pollution Vulnerabilities of One Million Real-world Websites. / Z. Kang, S. Li, Y. Cao // NDSS. — 2022.

52. *Lekies, S.* 25 million flows later: large-scale detection of DOM-based XSS / S. Lekies, B. Stock, M. Johns // Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security. — 2013. — С. 1193—1204.

53. A symbolic execution framework for javascript / P. Saxena [и др.] // 2010 IEEE Symposium on Security and Privacy. — IEEE. 2010. — С. 513—528.

54. An empirical study of information flows in real-world javascript / C.-A. Staicu [и др.] // Proceedings of the 14th ACM SIGSAC Workshop on Programming Languages and Analysis for Security. — 2019. — С. 45—59.

55. Staged information flow for JavaScript / R. Chugh [и др.] // Proceedings of the 30th ACM SIGPLAN conference on programming language design and implementation. — 2009. — С. 50—62.

56. Babel library [Электронный ресурс]. — URL: https://babeljs.io/ (дата обр. 25.04.2024).

57. Babel AST Specification [Электронный ресурс]. — URL: https://github.com/babel/babel/blob/main/packages/babel-parser/ast/spec.md (дата обр. 03.09.2024).

58. Babel AST Specification — Expressions [Электронный ресурс]. — URL: https://github.com/babel/babel/blob/main/packages/babel-parser/ast/spec.md#expressions (дата обр. 03.09.2024).

59. CommonJS modules — The module object — Node.js Documentation [Электронный ресурс]. — URL: https://nodejs.org/api/modules.html#the-module-object (дата обр. 03.09.2024).

60. Destructuring assignment — MDN Web Docs [Электронный ресурс]. — URL: https://developer.mozilla.org/en-US/docs/Web/API/Window (дата обр. 19.08.2024).

61. Destructuring assignment — MDN Web Docs [Электронный ресурс]. — URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment (дата обр. 19.08.2024).

62. CommonJS modules — The require function — Node.js Documentation [Электронный ресурс]. — URL: https://nodejs.org/api/modules.html#requireid (дата обр. 03.09.2024).

63. ECMAScript Language Specification [Электронный ресурс]. — URL: https://ecma-international.org/publications-and-standards/standards/ecma-262/ (дата обр. 07.09.2024).

64. Strict equality — MDN Web Docs [Электронный ресурс]. — URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Strict_equality#description (дата обр. 03.09.2024).

65. ECMAScript Language Specification — ToString [Электронный ресурс]. — URL: https://tc39.es/ecma262/multipage/abstract-operations.html#sec-tostring (дата обр. 07.09.2024).

66. ECMAScript Language Specification — ToString [Электронный ресурс]. — URL: https://tc39.es/ecma262/multipage/ecmascript-language-expressions.html#sec-applystringornumericbinaryoperator (дата обр. 07.09.2024).

67. *Xie, Y.* Static Detection of Security Vulnerabilities in Scripting Languages / Y. Xie, A. Aiken // USENIX Security Symposium. T. 15. — 2006. — С. 179—192.

68. *Dahse, J.* Simulation of Built-in PHP Features for Precise Static Code Analysis / J. Dahse, T. Holz // NDSS. T. 14. — 2014. — С. 23—26.

69. Статический анализатор Svace для поиска дефектов в исходном коде программ / В. П. Иванников [и др.] // Труды Института системного программирования РАН. — 2014. — Т. 26, № 1. — С. 231—250.

70. *Дудина, И. А.* Применение статического символьного выполнения для поиска ошибок доступа к буферу / И. А. Дудина, А. А. Белеванцев // Программирование. — 2017. — № 5. — С. 3—17.

71. Сайт языка программирования TypeScript [Электронный ресурс]. — URL: https://www.typescriptlang.org/ (дата обр. 10.09.2024).

72. Сайт среды выполнения NodeJS [Электронный ресурс]. — URL: https://nodejs.org/ (дата обр. 10.09.2024).

73. Сайт с описанием протокода Chrome DevTools Protocol [Электронный ресурс]. — URL: https://chromedevtools.github.io/devtools-protocol/ (дата обр. 10.09.2024).

74. Сайт библиотеки Puppeteer [Электронный ресурс]. — URL: https://pptr.dev/ (дата обр. 10.09.2024).

75. DOMContentLoaded event — MDN Web Docs [Электронный ресурс]. — URL: https://developer.mozilla.org/en-US/docs/Web/API/Document/DOMContentLoaded_event (дата обр. 03.09.2024).

76. load event — MDN Web Docs [Электронный ресурс]. — URL: https://developer.mozilla.org/en-US/docs/Web/API/Window/load_event (дата обр. 03.09.2024).

77. MutationObserver — MDN Web Docs [Электронный ресурс]. — URL: https://developer.mozilla.org/en-US/docs/Web/API/MutationObserver (дата обр. 03.09.2024).

78. Event handler attributes — MDN Web Docs [Электронный ресурс]. — URL: https://developer.mozilla.org/en-US/docs/Web/HTML/Attributes#event_handler_attributes (дата обр. 03.09.2024).

79. Damn Vulnerable Web Application (DVWA) [Электронный ресурс]. — URL: https://github.com/digininja/DVWA (дата обр. 01.08.2024).

80. Juice Shop 8.3.0 [Электронный ресурс]. — URL: https://github.com/juice-shop/juice-shop/tree/v8.3.0 (дата обр. 01.07.2023).

81. MyBB 1.8.19 [Электронный ресурс]. — URL: https://mybb.com/ (дата обр. 01.08.2024).

82. OWASP WebGoat Project [Электронный ресурс]. — URL: https://owasp.org/www-project-webgoat/ (дата обр. 01.07.2023).

83. Web Input Vector Extractor Teaser [Электронный ресурс]. — URL: https://github.com/bedirhan/wivet (дата обр. 01.07.2023).

84. An empirical comparison of commercial and open-source web vulnerability scanners / R. Amankwah [и др.] // Software: Practice and Experience. — 2020. — Т. 50, № 9. — С. 1842—1857. — eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.2870. — URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2870.

85. *Makino, Y.* Evaluation of web vulnerability scanners / Y. Makino, V. Klyuev // 2015 IEEE 8th International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS). — Warsaw, Poland : IEEE Press, 2015. — С. 399—402. — URL: https://doi.org/10.1109/IDAACS.2015.7340766.

86. W3af [Электронный ресурс]. — URL: http://w3af.org/ (дата обр. 01.07.2023).

87. Wget [Электронный ресурс]. — URL: https://www.gnu.org/software/wget/ (дата обр. 01.07.2023).

88. Статистика распространённости упаковщика RequireJS по данным сайта W3Techs [Электронный ресурс]. — URL: https://w3techs.com/technologies/details/js-requirejs (дата обр. 03.09.2024).

89. Статистика распространённости библиотки Backbone по данным сайта W3Techs [Электронный ресурс]. — URL: https://w3techs.com/technologies/details/js-backbone (дата обр. 03.09.2024).

Список рисунков

Список таблиц

Приложение А

Акты о внедрении результатов диссертационной работы

BIG DATA

МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ имени М.В.ЛОМОНОСОВА (МГУ)

Центр компетенций НТИ по технологиям хранения и анализа больших данных

119192, Москва, Ломоносовский проспект, д. 27,
корп. 1, оф. Е 801-804
тел.: (495) 938-25-72; +7 (915) 140-42-04
info@digital.msu.ru

Исх. от **24.09.2024** № **36-24/248-03**
На № _____ от _____

АКТ
о внедрении результатов диссертационной работы
Сигалова Даниила Алексеевича
на тему «Методы выявления поверхности атаки веб-приложений при помощи анализа клиентского JavaScript-кода», представленной на соискание учёной степени кандидата технических наук по специальности 2.3.5 — «Математическое и программное обеспечение вычислительных систем, комплексов и компьютерных сетей»

Следующие результаты диссертационного исследования Сигалова Даниила Алексеевича на тему «Методы выявления поверхности атаки веб-приложений при помощи анализа клиентского JavaScript-кода»:

1) метод обнаружения информации о серверных входных точках для задачи анализа защищенности приложений методом “черного ящика” с помощью статического анализа клиентского кода;
2) методика поиска уязвимостей веб-приложений в модели “чёрного ящика”,

при разработке была использована «Система автоматического поиска уязвимостей в веб-приложениях на основе обработки больших данных». Разработанный анализатор клиентского кода, осуществляющий поиск входных серверных точек, позволил системе выявлять и анализировать входные точки, не поддающиеся обнаружению другими методами. При этом анализатор применим для работы с реальными приложениями — он способен за

относительно короткое время обрабатывать их клиентскую часть, включая страницы, в большом количестве содержащие сложный JavaScript-код. Результатом этого стало улучшение характеристик разработанной системы в части её способности обнаруживать серверные уязвимости. Анализатор является одним из основных модулей обнаружения серверных входных точек в составе системы поиска уязвимостей.

Заместитель директора ЦХАБД МГУ _____ Тростьянский С.С.

ООО «СолидСофт»
ОГРН 5147746020241, ИНН 7714944046,
КПП 773601001
117312, г. Москва, вн. тер. г.
муниципальный округ Академический,
улица Вавилова, дом 47А, помещение ½
Тел.: +7 (499) 705-76-57
www.solidwall.ru

от 30.09.2024г.

АКТ

о внедрении результатов диссертационной работы

«Методы выявления поверхности атаки веб-приложений при помощи анализа клиентского JavaScript-кода»

Результаты диссертационной работы Сигалова Даниила Алексеевича на тему «Методы выявления поверхности атаки веб-приложений при помощи анализа клиентского JavaScript-кода», представленной на соискание учёной степени кандидата технических наук по специальности 2.3.5 «Математическое и программное обеспечение вычислительных систем, комплексов и компьютерных сетей», были внедрены ООО «СолидСофт» при разработке сканера SolidPoint DAST, осуществляющего автоматизированный поиск уязвимостей в веб-приложениях в режиме «чёрного ящика». Созданное Сигаловым Д. А. средство анализа клиентского JavaScript-кода входит в стандартную сборку сканера, начиная с первых его версий. Оно успешно применено пользователями сканера при анализе реальных приложений, что привело к обнаружению проблем безопасности различной степени критичности. В данный момент анализатор используется для выявления серверной функциональности при каждом сканировании. Благодаря использованию статического анализа JavaScript-кода разработанное Сигаловым Д. А. средство дополняет набор серверных точек взаимодействия, которые могут быть обнаружены более традиционными средствами, такими, как анализ HTML-элементов («статический краулинг») и динамический анализ JavaScript («динамический краулинг»).

ООО «СолидСофт»,
Технический директор
Петухов А.А.