

СОДЕРЖАНИЕ

ВВЕДЕНИЕ	6
ГЛАВА 1. ОБЗОР СУЩЕСТВУЮЩИХ ПОДХОДОВ	8
1.1 Критерии сравнения подходов	8
1.2 Оценка существующих подходов	8
1.2.1 CodeQL	8
1.2.2 Semgrep	9
1.2.3 SonarQube	10
1.2.4 FlowDroid	11
1.3 Резюме	11
ГЛАВА 2. ПОСТАНОВКА ЗАДАЧИ РАЗРАБОТКИ DSL ДЛЯ РАСШИРЕНИЯ БАЗЫ ЗНАНИЙ JSA И АНАЛИЗ ПУТЕЙ РЕШЕНИЯ . .	12
2.1 Постановка задачи	12
2.2 Выбор путей решения	13
2.2.1 Пригодность к использованию программистом	13
2.2.2 Способность описывать внешние библиотеки и фреймворки с точки зрения семантики и потоков данных	14
2.2.3 Возможность для переиспользования кода	15
2.2.4 Совместимость с PT JSA	15
2.3 Абстрагирование от внутренних особенностей PT JSA	16
2.4 Резюме	20
ГЛАВА 3. РАЗРАБОТКА И РЕАЛИЗАЦИЯ ЯЗЫКА И ТРАНСЛЯТОРА .	22
3.1 Описание DSL	22
3.1.1 Объявления верхнего уровня	22
3.1.2 Пакеты	22
3.1.3 Объекты	23
3.1.4 Функции	23
3.1.5 Объявление переменных	25
3.1.6 Выражения	26
3.1.7 Присваивание значение	27
3.1.8 Оператор ветвления	28
3.1.9 Аннотации	29
3.1.10 Совместимость с кодом на C# Script	29
3.1.11 Стандартная библиотека	30
3.1.12 Система типов	33
3.1.13 Резюме	34
3.2 Разработка транслятора	34
3.2.1 Архитектура транслятора	34
3.2.2 Синтаксический анализ	35

3.2.3 Семантический анализ	37
3.2.4 Генерация целевого кода	42
3.3 Особенности языка и транслятора для поддержки Python	45
3.3.1 Резюме	45
3.4 Резюме	45
ГЛАВА 4. АПРОБАЦИЯ ПОЛУЧЕННЫХ ИНСТРУМЕНТОВ	47
4.1 Подход к апробации	47
4.2 Библиотека для отправки HTTP-запросов	47
4.3 Библиотека для работы с базой данных	52
4.4 Фреймворк для обработки HTTP-запросов	56
4.5 Пользовательская библиотека для получения информации об IP ..	61
4.6 Генерируемая автоматически пользовательская библиотека	63
4.7 Сравнение с существующим способом расширения базы знаний ..	68
4.8 Резюме	69
ЗАКЛЮЧЕНИЕ	70
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ	72

ВВЕДЕНИЕ

Программная инженерия на протяжении всего своего существования стремилась автоматизировать каждый из аспектов разработки и сопровождения программного обеспечения. Так появились компиляторы, IDE, анализаторы кода, ассистенты для его написания и многие другие классы инструментов. Многие из них значительно улучшают качество жизни программистов, а значит, делают их продуктивнее.

Одним из ключевых классов инструментов являются анализаторы кода. Они разделяются на статические, которые производят анализ без фактического запуска программы, а также динамические, которые анализируют запущенную программу. Каждый из этих подходов, как и гибридные, нашли своё применение. К примеру, статический анализ кода применяется в компиляторах, а одной из сфер применения динамического анализа является поиск ошибок в ПО.

Анализаторы кода позволяют определять некоторые свойства программ. К ним относятся: оценка качества кода и проверка соблюдения стиля, обнаружение дефектов и неиспользуемого кода, анализ производительности и энергоэффективности. Для каждого из типов анализа применяются различные подходы. Некоторые из них работают с кодом как с обычным текстом, другим же требуется специальное представление программы для большей эффективности.

Одним из подходов, применяемых в статическом анализе, является анализ потоков данных (*Data Flow Analysis*) [1]. Он позволяет собирать информацию о возможных значениях и путях передачи данных в программе. Как и у многих других автоматических методов анализа кода, у него есть множество технических проблем. Одной из них является проблема взрыва количества путей [2]. Современные программы имеют большие размеры. В большинстве из них представлено большое количество ветвлений, циклов и рекурсий. Так как каждая из этих операций потенциально приводит к увеличению числа состояний в два раза, их общее число растёт экспоненциально. Эту проблему называют проблемой взрыва количества путей.

Существует большое количество подходов для частичного решения проблемы взрыва количества путей. К ним относятся использование суммаризаций, объединяющих несколько состояний в одно, применение эффективных алгоритмов для хранения и операций с состояниями, а также аппроксимацией свойств известного анализируемого кода. К такому коду может относиться код стандартной библиотеки анализируемого языка, код

наиболее популярных библиотек и фреймворков. Информация, которую инструмент анализа кода имеет о таких компонентах, называют базой знаний.

Одной из областей применения анализа потока данных является поиск уязвимостей в исходном коде программ (*static automatic security testing, SAST*). Для этого анализаторы отслеживают потоки данных от точек входа (*sources*) до стоков (*sinks*). В качестве точек входа обычно считают обработчики HTTP-запросов, входные аргументы консольных приложений. Стоками называют базы данных, ответ пользователю, записи на диске. Существуют также фильтрующие функции, которые проводят валидацию данных из точек входа, предотвращая попадания загрязнённых (*tainted*) данных в стоки. Примером такой валидации могут служить функции, экранирующие HTML-теги, проверяющие пользовательские данные при помощи регулярных выражений и другие преобразования.

Одним из ведущих SAST анализаторов, использующих анализ потоков данных, является коммерческий анализатор JSA компании Positive Technologies. Для увеличения точности, в нём применяется символьное исполнение, с помощью которого анализатор отсеивает большое количество ложно-положительных срабатываний. Он поддерживает анализ таких языков как C#, Java, JavaScript, TypeScript, Python, PHP и Go. Для каждого из поддерживаемых языков реализован отдельный модуль, называемый языковым провайдером. Языковые провайдеры анализируют исходный код приложений, строят символьное представление, по которому происходит анализ потоков данных между источниками и стоками.

PT JSA использует базы знаний об аппроксимациях библиотек и фреймворков на языке сценариев C# Script — расширении языка C#. Такие сценарии имеют доступ ко внутренней инфраструктуре анализатора JSA, что даёт неограниченные возможности, однако, требует высоких навыков разработки у программистов. Применение более простого подхода к расширению базы знаний анализатора кода JSA позволило бы упростить этот процесс, что повлекло бы к улучшению таких важных характеристик, как скорость анализа и его точность, у пользователей.

В первой главе приведён обзор существующих подходов к расширению функциональности анализаторов кода. Во втором разделе производится постановка задачи и анализ путей её решения. В третьем разделе описаны синтаксис и семантика прототипа DSL и его транслятора в язык C# Script. В четвёртом разделе происходит апробация полученного языка и его транслятора, а также приводятся примеры описаний библиотек на нём. В заключении полученные результаты анализируются, на основании чего приводятся идеи для дальнейшего развития проекта.

ГЛАВА 1. ОБЗОР СУЩЕСТВУЮЩИХ ПОДХОДОВ

Данная глава содержит критерии сравнения существующих подходов к решению проблемы расширения набора правил для статических анализаторов кода. Затем, рассматриваются наиболее актуальные существующие подходы.

1.1 Критерии сравнения подходов

Существует большое число подходов, призванных решить задачу расширения базы знаний SAST анализатора кода. В рамках данной работы, их можно классифицировать по следующим признакам:

- Возможность интеграции в JSA компании Positive Technologies
- Тип описываемых правил
 - Правила, моделирующие поведение внешних зависимостей кода
 - Правила, моделирующие потоки данных
- Способ описания правил
 - Кодом на анализируемом языке
 - Кодом на специальном языке
 - Кодом на другом языке общего назначения
 - Другой структурированный текстовый формат
- Абстракция анализа
 - Потоки данных
 - Исходный код программы
 - Специальное представление
- Поддержка мета-программирования (декораторы, аннотации, определение семантики на основе соглашений об именовании)

1.2 Оценка существующих подходов

Данный раздел содержит оценку подходов, реализованных в существующих продуктах или представленных в академических работах. Каждый из подходов оценивается с помощью критериев, приведённых в Разделе 1.1.

1.2.1 CodeQL

CodeQL [3] — одноимённый язык и инструмент для статического анализа кода. Процесс анализа состоит из трёх этапов: построение базы данных, соответствующей анализируемому коду, запуск запросов к ней и отображение результатов в пользовательском интерфейсе. В основе инструмента находится подход, который рассматривает анализ кода как

запрос к БД. Таким образом, инструмент позволяет обнаруживать дефекты ПО, такие как уязвимости и ошибки.

CodeQL поддерживает различные языки программирования, такие как C/C++, C#, Go, Java, Kotlin, Python. Для каждого из языков строится AST, которое затем преобразуется в специальное реляционное представление. В него уже входят правила для базовых ошибок и уязвимостей. Имеется возможность для создания как запросов, так и моделей, описывающих потоки данных в библиотеках и фреймворках. Для этого используется язык yaml. Модели позволяют указать анализатору на поведение потоков данных в библиотеке. К примеру, можно указать, что загрязнённые данные распространяются от аргументов к возвращаемому значению метода, порождаются им или поглощаются им. Таким образом, можно указывать источники, стоки и фильтрующие функции в домене потоков данных. В языке запросов CodeQL также есть поддержка аннотаций и декораторов, что позволяет создавать запросы с учётом этих языковых конструкций.

Несмотря на то что для этой технологии существует расширение для редактора кода VS Code, код правил, описанный на языке YAML, сложно читать и поддерживать. Также, модели позволяют описывать только потоки данных, но не имеют возможности для описания семантики библиотеки. У языка CodeQL отсутствует возможность быть интегрированным в другой анализатор.

1.2.2 Semgrep

[4] реализует легковесный семантический сбор информации о коде. Описание запросов для получения информации происходит на специальном предметно-определяемом языке. Он поддерживает описание синтаксических конструкций целевого языка на уровне AST с использованием оператора ..., обозначающего произвольное поддереву, а также мета-переменные, позволяющие давать имена таким поддеревьям. Поддержаны также примитивные литералы (строки, числа, булевы значения), идентификаторы, операторы и все основные синтаксические конструкции. Инструмент поддерживает большое число языков программирования, такие как Java, Kotlin, C#, Python. Используется внутреннее представление, в которое переводится код на каждом из анализируемых языков.

Одной из ключевых особенностей инструмента является скорость его работы. Для этого анализ осуществляется с использованием простых подходов: сопоставления с шаблоном, получения информации о типах на основе синтаксиса и примитивного анализа потоков данных. На основе

последнего построен taint-анализ, для которого можно описать источники, стоки и фильтрующие функции в виде правил на языке YAML.

Этот инструмент позволяет описывать большое количество простых ошибок и уязвимостей, таких как использование вывода в консоль в качестве логирования, отключение проверки сертификатов или простые SQL инъекции. Расширение возможностей анализатора предполагает написание правил для пользовательского кода, а не для описания моделей его зависимостей, что, вкупе с неточными методами анализа, приводит к или к низкой полноте анализа, или к высокому количеству ложно-положительных срабатываний. У языка правил semgrep отсутствует возможность интеграции в другие анализаторы. Стоит отметить, что так как правила анализа представляют мета-информацию и код, напоминающий анализируемый, то их удобно читать и поддерживать.

1.2.3 SonarQube

[5] предоставляет различные техники анализа для поиска ошибок и уязвимостей в коде приложений. Поддерживается большой список языков программирования, в частности, Java, C#, Kotlin, Python.

Система предоставляет четыре способа расширения функциональных возможностей: с помощью создания расширения сервера, с помощью правил XPath, с помощью определения дополнительных точек входа, стоков и фильтрующих функций, а также с помощью внешних утилит, результаты анализа которых можно импортировать [6]. Правила XPath поддерживаются только для Flex, PL/SQL, PL/I и XML. Расширения правил для taint-анализа происходит путём описания сигнатур соответствующих функций с помощью кода на JSON. Для некоторых языков поддержано определение важных для taint-анализа сигнатур функций, однако, этот механизм нельзя расширить пользовательскими правилами.

Недостатками данного инструмента можно считать низкую гибкость расширений taint-анализа. Несмотря на то что он предоставляет большое количество типов метода с точки зрения анализа (источник данных, фильтрующая функция, функция проверки, функция передачи без изменений и сток), они не позволяют учитывать многие важные особенности кода библиотек и фреймворков. К примеру, нет возможности описать преобразования значений, что приводит к понижению точности анализа. Также стоит отметить отсутствие возможности задания своих типов уязвимостей. Интеграция языка SonarQube в анализатор кода PT JSA также невозможна ввиду принципиальных различий в используемых подходах.

1.2.4 FlowDroid

В [7] представлен инструмент для анализа безопасности приложений для ОС Android. Он реализует подход taint-анализа, для чего реализован механизм для описания источников, стоков и фильтрующих функций на языке XML.

Подходы, применяемые в FlowDroid для описания библиотечного кода, имеют типичные для такой разметки недостатки: их сложно читать и поддерживать, а любое нетривиальное описание метода на этом языке получается очень объёмным. По этим причинам, реализовывать такой подход, как и интегрировать уже существующие правила, в PT JSA не имеет смысла.

1.3 Резюме

Данная глава обобщает существующие подходы к решению проблемы расширения возможностей SAST-анализаторов кода. Они имеют множество недостатков, основными из которых являются:

- Невыразительный или сложный язык для написания расширений базы знаний;
- Отсутствует возможность для описания потоков данных с источниками, стоками и фильтрующими функциями;
- Расширения, если и позволяют описывать потоки данных, то не описывают их преобразования;
- Отсутствует возможность интеграции в анализатор кода JSA компании Positive Technologies.

Для исправления этих недостатков было принято разработать новый инструмент.

ГЛАВА 2. ПОСТАНОВКА ЗАДАЧИ РАЗРАБОТКИ DSL ДЛЯ РАСШИРЕНИЯ БАЗЫ ЗНАНИЙ JSA И АНАЛИЗ ПУТЕЙ РЕШЕНИЯ

2.1 Постановка задачи

Пример типичного кода обработчика запросов в web-сервере с использованием библиотеки Flask и языка программирования Python приведён в Листинге 2.1. В нём обработчик запросов *handler* получает от клиента значение параметра *username*, инициализирует подключение к базе данных и отправляет запрос на получение пользователя с заданным именем из БД. Так как запрос получается простой конкатенацией, а проверка *userid* в коде нет, этот код уязвим к атаке «внедрение SQL-кода». К примеру, если клиент отправит запрос с аргументом *username* вида «' OR '1' = '1"», то при конкатенации строки запроса подставится дополнительное условие для оператора *SELECT*, которое позволит получить список всех пользователей.

```
1  app = Flask(__name__)
2
3  @app.route('/search_user', methods=['GET'])
4  def handler():
5      username = request.args.get('username')
6      conn = psycopg2.connect(**DB_PARAMS)
7      cursor = conn.cursor()
8
9      query = f"SELECT * FROM users WHERE username =
      '{username}'"
10     cursor.execute(query)
11
12     results = cursor.fetch()
13
14     cursor.close()
15     conn.close()
16
17     return jsonify({'user': results})
```

Листинг 2.1 — Пример обработчика HTTP-запросов с уязвимостью

Код в Листинге 2.1 позволяет продемонстрировать taint-анализ. С точки зрения анализа потока данных, источником загрязнённой информации является поле *args* объекта *request*. Это ассоциативный массив, содержащий

параметры HTTP-запроса, которые передал клиент. Из него извлекается значение с именем *username*. Затем, эти данные подставляются в SQL запрос на строке 9. В зависимости от семантики интерполяции строк, на это можно смотреть по-разному. В Python интерполируемое значение никак не экранируется. Таким образом, метка загрязнённости данных сохраняется для всего выражения переменной *query*. Затем, она передаётся в функцию *execute*, которая отправляет этот запрос в базу данных. Библиотека для операций с БД не может определить, намеренно запрос был написан так или нет. По этому, запрос передаётся базе данных. Функция *execute* с точки зрения taint-анализа рассматривается как сток данных, однако, анализ можно продолжить и определить, что вызов функции *fetch* на 12 строке возвращает данные из БД, полученные уязвимым запросом, а затем его результат передаётся пользователю.

Автоматически невозможно определить семантику вызовов функции с точки зрения taint-анализа. Информация о них содержится в базе знаний анализатора. Процесс расширения базы знаний сложен из-за необходимости компетенций не только программиста, но и аналитика информационной безопасности. В PT JSA уже существует механизм расширения с помощью C# Script [8]. Он позволяет использовать внутренний API анализатора, что даёт большие возможности, но делает процесс сложным: предполагается понимание внутренних механизмов анализатора программистом. В рамках Работы предлагается необходимо упростить процесс написания кода расширений за счёт разработки предметно-ориентированного языка (*DSL*, *domain specific language*), который будет простым и выразительным.

Таким образом, необходимо разработать синтаксис и семантику прототипа DSL и интегрировать его в анализатор JSA. Он должен соответствовать следующим требованиям:

- Возможность написания расширений для PT JSA;
- Способность описывать источники и стоки загрязнённых потоков данных, а также их фильтрацию;
- Абстрагирование от внутренних особенностей анализатора.

2.2 Выбор путей решения

В данном разделе рассматриваются пути реализации языка, а также его интеграции с анализатором JSA компании Positive Technologies.

2.2.1 Пригодность к использованию программистом

Существует множество способов описания компонентов. У каждого из них есть преимущества и недостатки, приведённые в Таблице 1.

Таблица 1 — Сравнение способов описания компонентов ПО

Характеристика, подход	YAML, JSON, XML	Графически	Существующий язык моделирования	Язык об-щего на-значения	Существующий механизм расширения JSA	Новый DSL
Наличие инструментов разработки	Существуют, не предназначены для этого	Существуют, не предназначены для этого	Существуют	Существуют	Существуют	Необходимо реализовать
Гибкость в описании	Низкая	Низкая	Высокая	Высокая	Высокая	Высокая
Сложность обучения	Средняя	Средняя	Высокая	Низкая	Высокая	Низкая
Сложность интеграции с JSA	Средняя	Высокая	Высокая	Высокая	—	Низкая
Прочее	Необходима доработка	Необходима доработка	Необходима доработка	Необходимо ввести доп. ограничения	Большой объём кода	Высокая гибкость в принятии решений

Последний способ представляет наибольший интерес. В этом случае разрабатываемый язык должен быть пригодным для разработчика средней квалификации. Это накладывает ограничение на концепции, которые могут быть использованы. Предлагается использовать следующие идеи:

- Статическая типизация:
 - Номинативная типизация;
 - Типовые параметры;
 - Простой вывод типов.
- Использование идей из популярных языков:
 - Объекты;
 - Функции или методы;
 - Переменные;
 - Операции должны иметь очевидный синтаксис.

2.2.2 Способность описывать внешние библиотеки и фреймворки с точки зрения семантики и потоков данных

Так как статический анализатор кода РТ JSA применяет техники символьного исполнения, абстрактной интерпретации и анализа потока

данных, необходимы языковые возможности для описания не только потоков загрязнённых данных, но и поведения описываемых компонентов.

Необходимо учесть возможности для:

- Описания сигнатур сущностей (методов, объектов, глобальных переменных и т.д.) для возможности сопоставления с описываемыми;
- Указания типов для полей, аргументов методов, возвращаемых значений, переменных для предоставления этой информации об анализируемом коде для языков с динамической типизацией;
- Описания семантики с помощью арифметических операций, операций над строками, вызовов функций, ветвлений, сохранения и чтения из полей объектов;
- Работы с потоками загрязнённых данных (создание, продвижение, стоки и удаление метки загрязнённости).

2.2.3 Возможность для переиспользования кода

В современной разработке программного обеспечения вопрос переиспользования кода чрезвычайно важен. Механизмы повторного использования кода могут включать:

- Возможность организации модулей;
- Объекты, методы, функции;
- Наследование;
- Метaprogramмирование (декораторы, аннотации, макросы, генерация кода).

Переиспользование кода открывает возможности для создания стандартной библиотеки языка. Так, можно перенести часть функциональных возможностей в неё, а не разрабатывать для неё новый синтаксис и семантику.

2.2.4 Совместимость с PT JSA

Так как основной целью проекта является расширение функциональных возможностей анализатора кода PT JSA, обеспечение первоклассной интеграции инструментов является важной задачей. Существует несколько способов её решения:

- Поддержка DSL со стороны JSA.

Так как поддерживаемые в PT JSA языки имеют особенности, для каждого из них разработан так называемый языковой модуль, который отвечает за статический анализ кода. Таким образом, необходимо поддержать DSL в каждом из модулей, что может оказаться затруднительным.

- Трансляция DSL в поддерживаемый JSA язык.

Этот подход позволит разработать DSL без модификации PT JSA. Однако, для реализации этого подхода потребуется реализация и поддержка транслятора для каждого из поддерживаемых языков (на момент написания работы их 7).

- Трансляция DSL в язык C# Script с использованием существующего API расширений.

Данный подход предлагает преимущества предыдущего, а именно, отсутствие модификации PT JSA. И также он ликвидирует его недостаток с необходимостью поддержки языка в каждом из модулей. Ещё одним преимуществом данного подхода является совместимость с расширениями базы знаний на C# Script. Можно организовать двустороннее использование кода между DSL и старым кодом, реализовывать особенно сложные случаи описания компонентов на C# Script, а также разработать на нём часть стандартной библиотеки.

Последний подход представляет наибольший интерес ввиду своей гибкости, а также возможности интеграции с уже существующей базой знаний.

2.3 Абстрагирование от внутренних особенностей PT JSA

Низкоуровневый API уже доступен из расширений базы знаний на языке C# Script и используется для внутренних нужд. Низкий уровень абстрагирования от деталей реализации анализатора делает практически невозможным его использование программистами, далёкими от них. В Листинге 2.2 приведён фрагмент примера описываемой библиотеки на языке Python. Листинг 2.3 содержит пример использования этой библиотеки.

```

1  class Request:
2      def get_param(self, paramname: str) -> None:
3          # код получения значения параметра HTTP-запроса по
4          # имени; источник загрязнённых данных
5          pass
6
7  class Response:
8      def send(self, text: str) -> None: # сток
9          # код отправки строки пользователю в качестве ответа;
10         # сток данных
11         pass

```

```

12
13 def handle(
14     url: str,
15     handler: Callable[[Request, Response], None]
16 ) -> None:
17     # связывание кода обработчика запроса с
18     # соответствующим URL
19     pass
20
21 def escape_html(text: str) -> str: # фильтрующая функция
22     # возвращает строку text с экранированием HTML-символов;
23     # фильтрующая функция
24     pass

```

Листинг 2.2 — Фрагмент пример кода библиотеки.

```

1 from customModule import handle, escape_html
2 import sqlite3
3
4 con = sqlite3.connect("foo@bar.baz")
5 def vulnerable_handler(request, response):
6     param = request.get_param("baz")
7     response.send(param) # межсайтовый скриптинг (XSS)
8
9 def filtered_handler(request, response):
10     param = request.get_param("baz")
11     # нет уязвимости из-за фильтрации
12     response.send(escape_html(param))
13
14 def conditional_handler(request, response):
15     condition = request.get_param("is_admin")
16     result = None
17     if condition == "secret_key":
18         query = request.get_param("sql")
19         cursor = con.cursor().execute(query) # SQL инъекция
20         result = cursor.fetchone()
21     response.send(result)
22
23 handle("/vulnerable", vulnerable_handler)

```

```
24 handle("/filtered", filtered_handler)
25 handle("/admin", conditional_handler)
```

Листинг 2.3 — Пример пользовательского кода.

Примитивный фреймворк, описанный в Листинге 2.2, содержит в себе источник (функция *get_param* класса *Request*), фильтрующую функцию (*escape_html*) и сток (*send* класса *Response*). Семантика этих функций отображена в расширении базы знаний, приведённом в Листинге 2.4 на языке C# Script.

```

1  var requestclass = PythonTypes.CreateClass("Request")
2      .WithMethod("get_param", (location, functionCall) =>
3      {
4          var nameExpression = functionCall.Arguments[1];
5          var taintOrigin = QueryOrigin(nameExpression);
6          return ProcessorApi.CreateTaintedString(taintOrigin);
7      }
8
9  var responseClass = PythonTypes.CreateClass("Response")
10     .WithMethod("send", (location, functionCall) =>
11     {
12         if (functionCall.Arguments.Length > 0) {
13             Detector.Detect(
14                 location,
15                 functionCall.Arguments[1],
16                 VulnerabilityType.CrossSiteScripting,
17                 PvoArgumentGrammar.HtmlText);
18         }
19         return SemanticsApi.None;
20     }
21
22 CreateModuleFunction("handle", (location, functionCall) =>
23     {
24         var urlExpression = functionCall.Arguments[0];
25         var handlerFunc = functionCall.Arguments[1];
26         var request = requestClass.CreateInstance(location);
27         var response = responseClass.CreateInstance(location);
28
29         ProcessFrameworkEntryPoint(
30             location,
31             urlExpression,
32             HttpMethodType.All,
33             forkedApi =>
34                 forkedApi.Interpreter.InvokeFunction(
35                     location,
36                     handlerFunc,
37                     request,

```



```

38         response));
39
40     return SemanticsApi.None;
41 }
42 );
43
44 CreateModuleFunction(
45     "escapeHtml",
46     (location, functionCall) =>
47     {
48         return functionCall
49             .Arguments[0]
50             .AddFilteredPvoGrammar(PvoArgumentGrammar.HtmlText);
51     }
52 );

```

Листинг 2.4 — Пример расширения JSA для поддержки библиотеки.

Можно заметить, что даже для такого простого примера необходимо оперировать большим количеством низкоуровневых операций, таких как обработчик метода (последний аргумент функции *WithMethod*), грамматики (синтаксический тип данных). Количество аргументов вызова функции также необходимо проверять вручную. Значение *Location* передаётся во многие операции. Это необходимо для точности результатов анализа. Таким образом, проектируемый DSL должен предоставлять абстракции, которые скрывают описанные сложности, что приведёт к упрощению обучения этому языку, а также ускорению написания кода на нём. За счёт уменьшения объёма кода, снизится и количество ошибок.

2.4 Резюме

В этой главе описаны основные свойства, которыми должен обладать разрабатываемый способ расширения базы знаний поддержки библиотек и фреймворков анализатора PT JSA. Ими стали:

- Расширение задаётся с помощью кода на новом предметно-ориентированном языке с простыми и очевидными синтаксисом и семантикой;
- Язык должен быть расширяемым с помощью изменений в стандартной библиотеке;
- В нём должны быть представлены возможности для описания потоков данных и семантики у внешних компонентов;

- Возможность выделения кода в пакеты;

Для ускорения разработки прототипа, были принято решение ограничить возможность разработки расширений только для языка Python с возможностью последующей доработки для поддержки других языков, поддерживаемых PT JSA.

ГЛАВА 3. РАЗРАБОТКА И РЕАЛИЗАЦИЯ ЯЗЫКА И ТРАНСЛЯТОРА

3.1 Описание DSL

Данный раздел содержит описание синтаксиса и семантики разрабатываемого языка. Каждый из подразделов посвящён конкретной языковой возможности и приводит, при необходимости, синтаксическое, семантическое описание, а также правила типизации.

Правила описания синтаксиса представлены в форме, похожей на BNF. В них в угловых скобках указаны имена правил, знак ? указывает на опциональность термина, * указывает произвольное количество повторений термина, + указывает, что терм может повторяться больше одного раза, а | используется как разделитель между правилами в случае, когда применимо любое из списка. Токены, указанные без угловых скобок или в двойных кавычках, обозначают себя.

3.1.1 Объявления верхнего уровня

Объявления верхнего уровня описывают структуру файла с кодом на разрабатываемом языке. Он состоит из опционального указания имени пакета, за которым идёт произвольное число верхоуровневых объявлений. Соответствующие синтаксические правила приведены на Рисунке 3.1.

```
<file> ::=  
    <packageDecl>? <topLevelDecl>*;  
  
<topLevelDecl> ::=  
    ...
```

Рисунок 3.1 — Синтаксические правила структуры файла

Правило *<file>* описывает структуру всего файла с кодом на разрабатываемом DSL. Правило *<topLevelDecl>* содержит в себе все определения для верхоуровневых сущностей. Они будут рассмотрены далее.

3.1.2 Пакеты

Пакеты (ключевое слово *package*) позволяют определить имя пакета в описываемой библиотеке. Они могут быть использованы для импортирования (ключевое слово *import*). Соответствующие синтаксические правила приведены на Рисунке 3.2.

```

<packageDecl> ::=
    package <string> ";";

<topLevelDecl> ::=
    ...
    | import <string> ";";
    ...

```

Рисунок 3.2 — Синтаксические правила объявления и подключения пакетов

Объявление пакета является опциональным. Если оно не указано, оно будет установлено на основе имени файла. Это поведение было оставлено для сохранения совместимости со старым механизмом модулей в расширениях PT JSA. Из синтаксических правил на Рисунке 3.2 следует, что не допускается больше одного такого объявления, а также то что оно должно быть первым.

3.1.3 Объекты

Объекты используются для моделирования таких сущностей библиотек как классы, объекты и структуры. Синтаксические правила приведены на Рисунке 3.3.

```

<topLevelDecl> ::=
    ...
    | <objectDecl>
    ...

<objectDecl> ::=
    <annotation>* 'object' <ID> '{' <objectBody> '}';

<objectBody> ::=
    <objectBodyStatement>*;

<objectBodyStatement> ::=
    ...

```

Рисунок 3.3 — Синтаксические правила для функций

Объекты являются контейнерами для функций и полей, которые будут рассмотрены далее.

3.1.4 Функции

Функции в DSL позволяют описывать сигнатуру и поведение функций из описываемого кода. Их синтаксические правила представлены на Рисунке 3.4.

```

<topLevelDecl> ::=
    ...
    | <funcDecl>
    ...

<funcDecl> ::=
    <annotation>*
    'intrinsic'? 'func' <ID> generic? '(' <args> ')' (':' <ID>)?
    ('{' <statementsBlock> '}' )?;

<generic> ::=
    '<' <ID> (',' <ID>)* ','? '>';

<args> ::=
    <arg>? (',' <arg>)* ','?;

<arg> ::=
    <ID> (':' <ID>) ('=' <expression>)?;

<statementsBlock> ::=
    <statement>*;

<statement> ::=
    ...
    | return <expression>?
    ...

```

Рисунок 3.4 — Синтаксические правила для функций

Определение функции содержит её имя, список аргументов (возможно, пустой), опциональный возвращаемый тип и тело. В случае отсутствия указания возвращаемого типа, предполагается неявный тип *none*, обозначающий, что функция не возвращает значение. Поддерживается указание значений по-умолчанию у аргументов.

Модификатор *intrinsic* позволяет описывать внутренние функции. Они не содержат тела, а также могут объявлять типовые параметры (*generics*). Это необходимо для:

- специальной поддержки некоторых функций в трансляторе;
- возможности реализации функций на языке C# Scripting.

Типовые параметры позволяют писать обобщённый код с учётом неизвестных при определении функции типов. В настоящее время их можно объявить только у внутренних функций, использовать у их аргументов, а также передать конкретный тип при вызове.

Механизм внутренних функций описан подробнее в разделе 3.1.10.

Возврат значения из функции осуществляется оператором `return`. Допускается наличие нескольких операторов возврата в одной функции. Пример такой функции приведён в Листинге 3.1

```
1 func minOf(a: int, b: int): int {
2     if (a < b) {
3         return a;
4     }
5     return b;
6 }
```

DSL

Листинг 3.1 — Пример функции с возвращением результата

3.1.5 Объявление переменных

Синтаксис для определения новой переменной приведён на Рисунке 3.5.

```
<topLevelDecl> ::=
    ...
    | <varDecl> ';'
    ...

<objectBodyStatement> ::=
    ...
    | <varDecl> ';'
    ...

<statement> ::=
    ...
    | <varDecl> ';'
    ...

<varDecl> ::=
    'var' ID (':' ID)? ('=' expression);
```

Рисунок 3.5 — Синтаксические правила для объявления переменной

Из этих правил следует, что объявления переменных могут быть на верхнем уровне, на уровне объекта (поля) и внутри тел функций. Допускается отсутствие указания типа. В этом случае оно будет выведено на основании типа инициализирующего выражения. Допускается отсутствие этого выражения, в этом случае переменная будет непроинициализирована при объявлении. Отсутствие типа и значения при инициализации не допускается. Обработка чтения значения из этой переменной зависит от анализатора.

Запрещается переопределять переменные (*variables shadowing*), неизменяемые переменные отсутствуют для упрощения языка и транслятора.

3.1.6 Выражения

Разрабатываемый DSL поддерживает арифметические, строковые и булевы выражения. Также поддерживается вызов функций, instantiation экземпляров класса, использование переменных и чтение значений полей. Синтаксические правила для выражений приведены на Рисунке 3.6.

```
<expression> ::=
    <functionCall>
    | '(' <expression> ')'
    | <expression> ('*' | '/') <expression>
    | <expression> '%' <expression>
    | <expression> ('+' | '-') <expression>
    | ('-' | '+') <expression>
    | <expression> ('==' | '!=' | '<=' | '<' | '>=' | '>') <expression>
    | <expressionAtomic>
    ;

<expressionAtomic> ::=
    <primitiveLiteral>
    | <newExpression>
    | <variableExpression>
    | <qualifiedExpression>
    ;

<primitiveLiteral> ::=
    <intLiteral>
    | <floatLiteral>
    | <stringLiteral>
    | <boolLiteral>

newExpression ::=
    'new' <ID> '(' <expression>? (',' <expression>)+ ','? ')';

variableExpression ::=
    <ID>;

qualifiedAccess    ::=
    (<ID> '.')* <ID>;
```

Рисунок 3.6 — Синтаксические правила для выражений

В Листинге 3.2 приведен пример использования выражений.

```

1 func buildClient(host: string, port: int): HttpClient { DSL
2     var hostWithSchema = "https://" + host;
3     return new HttpClient(
4         /*port*/ minOf(1024, port),
5         /*host*/ hostWithSchema,
6     )
7 }

```

Листинг 3.2 — Пример использования выражений

Некоторые правила типизации приведены на Рисунке 3.7. Для этого используется способ, похожий на представленный в [9].

$$\frac{\Gamma \vdash v_1 : T \quad \Gamma \vdash v_2 : T}{\Gamma \vdash v_1 \oplus v_2 : T, \text{ где } \oplus \in \{*, /, \%, +, -\}} \text{T-BinExpr}$$

$$\frac{\Gamma \vdash v : \text{Bool}}{\Gamma \vdash !v : \text{Bool}} \text{T-Not}$$

$$\frac{\Gamma \vdash v_1 : T \quad \Gamma \vdash v_2 : T}{\Gamma \vdash v_1 R v_2 : \text{Bool}, \text{ где } R \in \{==, !=, >, <, >=, <= \}} \text{T-Rel}$$

$$\frac{v : T \in \Gamma}{\Gamma \vdash v : T} \text{T-Var}$$

$$\frac{\Gamma \vdash f : (T_1, \dots, T_n) \rightarrow T \quad \Gamma \vdash v_1 : T_1 \quad \dots \quad \Gamma \vdash v_n : T_n}{\Gamma \vdash (f(v_1, \dots, v_n)) : T} \text{T-FuncApp}$$

Рисунок 3.7 — Некоторые правила типизации выражений

3.1.7 Присваивание значение

Присваивание значений осуществляется с помощью оператора `=`. Оно позволяет присваивать значения для полей и переменных. Синтаксические правила приведены на Рисунке 3.8.

```

<statement> ::=
    ...
    | <assignment>
    ...

<assignment> ::=
    <expression> '=' <expression>;

```

Рисунок 3.8 — Синтаксические правила оператора присваивания значения

Можно заметить, что в правиле `<assignment>` слева от токена `'='` указано правило `<expression>`. Таким образом, синтаксически верно присваивание нового значения совершенно любым выражениям, даже таким, как instantiation нового объекта, вызов функции. Обработка таких случаев как ошибок осуществляется на этапе семантического анализа в процессе трансляции. Такое решение позволяет значительно облегчить код грамматики языка.

В Листинге 3.3 приведен пример использования операторов присваивания.

```
1 object HttpClient {
2     var port: int;
3     var base_url: string;
4
5     func __init__(port: int, base_url: string) {
6         self.port = port;
7         self.base_url = base_url;
8     }
9 }
```

DSL

Листинг 3.3 — Пример использования операций присваивания

3.1.8 Оператор ветвления

Оператор ветвления позволяет описать нелинейное поведение описываемого кода. Синтаксис ветвлений приведён на Рисунке 3.9. У ветвления, как и во многих популярных языках программирования, есть условие, тип которого должен быть *Bool*, основное тело, а также альтернативная ветка, которая также может содержать условное ветвление.

Для упрощения языка в рамках прототипа было принято решение отказаться от концепции использование условных ветвлений в качестве выражений.

```

<statement> ::=
    ...
    | <ifStatement>
    ...

<ifStatement> ::=
    'if' '(' <expression> ')'
    '{' <statementsBlock> '}'
    (
        ('else' <ifStatement>)
        | 'else' '{' <statementsBlock> '}'
    )?

```

Рисунок 3.9 — Синтаксические правила для ветвления

3.1.9 Аннотации

Аннотации позволяют изменять поведение транслятора для некоторых сущностей в коде. Их можно применять к объявлению объекта или функции. Синтаксические правила представлены на Рисунке 3.10.

```

<annotation> ::=
    '@' <ID> '(' <expressionList> ')';

<expressionList> ::=
    <expression> (',' <expression>)* ',';

```

Рисунок 3.10 — Синтаксическое правило для аннотации

Пример использования аннотации приведён на Рисунке 3.4. Сейчас реализована поддержка только для одной аннотации `@GeneratedName`. Оно позволяет задать имя переменной, которая будет использована в сгенерированном коде. Это необходимо для обеспечения возможности частичного написания кода расширения на низкоуровневом коде на языке C# Script. Дело в том, что при трансляции применяется намеренное искажение имён (*name mangling*) для избежания коллизий. Подробнее эта концепция была описана в [10].

```

1 @GeneratedName
2 object Obj {
3     ...
4 }

```

DSL

Листинг 3.4 — Пример использования аннотации `@GeneratedName`

3.1.10 Совместимость с кодом на C# Script

В силу того, что уже есть набор расширений, написанных на языке C# Script, а также ограничений текущей реализации DSL, важным аспектом разрабатываемых технологий является совместимость между языком и C# Script. На уровне языка, она обеспечивается двумя механизмами: *intrinsic-*

функциями и аннотацией `@GeneratedName` (см. раздел 3.1.9). Вторая часть поддержки реализована в трансляторе. При обнаружении им файлов с расширением `.jsa` (файлы старых расширений) с именем, совпадающим с именем файла на DSL, содержимое специальным образом будет добавлено в результирующий файл трансляции. В нём можно использовать имена сущностей из DSL с аннотацией `@GeneratedName`.

Внутренние (`intrinsic`) функции позволяют описать сигнатуру на DSL, а её реализацию на языке C# Script. Пример сигнатуры функции приведён в Листинге 3.5, а её реализации в Листинге 3.6. Она позволяет делать приведение типов. Так как с точки зрения расширений на языке C# Script семантических типов не существует, они заменяются на тип символьного выражения `SymbolicExpression`. Таким образом, код, использующий функцию `As` в DSL остаётся типизированным, а в реализации функция превращается в функцию идентичности. По соглашению, во все функции на целевом языке добавляется аргумент типа `Location`, который повсеместно используется в PT JSA. Это значение автоматически передаётся транслятором. В дальнейшем планируется добавить специальные аннотации, которые позволят изменить это поведение.

```
1 intrinsic func As<TResult>(expression: any): TResult DSL
```

Листинг 3.5 — Пример сигнатуры внутренней функции

```
1 SymbolicExpression As(C#  
2     Location location,  
3     SymbolicExpression expression)  
4 {  
5     return expression;  
6 }
```

Листинг 3.6 — Пример реализации внутренней функции

3.1.11 Стандартная библиотека

Стандартная библиотека языка содержит операции для работы с `taint`-данными. В настоящее время она содержит только `intrinsic`-функции, приведённые в Листинге 3.7.

1	<code>package "Standard";</code>	DSL
2		
3	<code>intrinsic func Detect(</code>	
4	<code> expression: any,</code>	
5	<code> vulnerabilityType: string,</code>	
6	<code> grammar: string)</code>	
7		
8	<code>intrinsic func WithTaintMark<T>(</code>	
9	<code> expression: T,</code>	
10	<code> taintOrigin: string): T</code>	
11		
12	<code>intrinsic func CreateTaintedDataOfTType<T>(</code>	
13	<code> taintOrigin: string): T</code>	
14		
15	<code>intrinsic func CreateDataOfTType<T>(): T</code>	
16		
17	<code>intrinsic func WithoutVulnerability<T>(</code>	
18	<code> expression: T,</code>	
19	<code> vulnerabilityType: string): T</code>	
20		
21	<code>intrinsic func GetTaintOrigin(</code>	
22	<code> expression: any): string</code>	
23		
24	<code>intrinsic func As<TResult>(</code>	
25	<code> expression: any): TResult</code>	

Листинг 3.7 — Код стандартной библиотеки на разрабатываемом языке

Рассмотрим функцию `WithTaintMark`. Она позволяет получить копию выражения, но с добавленной `taint`-меткой, указывающей её место происхождения (`origin`). Оно используется в РТ JSA для обнаружения уязвимостей, а также генерации кода для эксплуатации уязвимости. Обычно, им является код для HTTP-запроса. Эта функция позволяет описывать данные, порождённые источником (*sink*). Реализация функции `WithTaintMark` приведена в Листинге 3.8.

```

1 TExpr WithTaintMark<TExpr>(
2     Location location,
3     TExpr? expression,
4     string origin) where TExpr : SymbolicExpression
5 {
6     if (expression == null) {
7         return null;
8     }
9     var taintOrigin = new TaintOrigin(origin);
10    return expression.With(taintOrigin);
11 }

```

Листинг 3.8 — Реализация функции WithTaintMark

В Таблице 2 приведено описание всех функций стандартной библиотеки

Таблица 2 — Перечень функций стандартной библиотеки

Функция	Аргументы	Описание
Detect	<i>expression</i> : <i>any</i> — выражение для выявления загрязнённых данных; <i>vulnerabilityType</i> : <i>string</i> — тип уязвимости; <i>grammar</i> : <i>string</i> — синтаксический признак данных.	Запускает обнаружение загрязнённых данных в выражении. В случае, если они будут обнаружены, анализатор вернёт уязвимость соответствующего типа. Грамматика определяет синтаксический тип данных, такой как URL, HTML текст, SQL.
WithTaintMark<T>	<i>expression</i> : <i>any</i> — выражение для добавления taint-метки; <i>taintOrigin</i> — тип источника данных.	Добавляет taint-метку к копии выражения.
CreateTaintedDataOfType<T>	<i>T</i> — тип создаваемого объекта; <i>taintOrigin</i> — тип источника данных.	Создаёт непроинициализированный объект нужного типа без вызова конструктора и с taint-меткой.
CreateDataOfType<T>	<i>T</i> — тип создаваемого объекта.	Создаёт объект нужного типа без вызова конструктора без taint-метки.
WithoutVulnerability<T>	<i>expression</i> : <i>T</i> — выражение; <i>vulnerabilityType</i> : <i>string</i> — тип уязвимости для фильтрации.	Добавляет отметку об отфильтровывании уязвимости.

Функция	Аргументы	Описание
GetTaintOrigin	<i>expression: any</i> — выражение.	Возвращает источник taint-данных в виде строки.
As<T>	<i>expression: any</i> — выражение.	Приводит небезопасно выражение к типу <i>T</i> .

3.1.12 Система типов

Система типов разрабатываемого языка проста. Представлено несколько встроенных типов, а также произвольное количество пользовательских типов, порождённых объектами из описаний. В свою очередь, встроенные типы разделяются на основные и на типы, добавленные специально для поддержки определенного языкового модуля PT JSA. Таблица 3 содержит перечень всех встроенных типов. Отношение типизации, используемое для них, представлено на Рисунке 3.11.

Таблица 3 — Встроенные типы

Имя типа	Описание
Основные типы	
<i>any</i>	Высший (<i>top</i>) тип. Является базовым для любого другого типа
<i>int</i>	Тип целого знакового числа. В текущей реализации 64 разряда
<i>string</i>	Строковый тип
<i>bool</i>	Логический тип
<i>float</i>	Тип числа с плавающей точкой. В текущей реализации 64 разряда, IEEE 754
<i>none</i>	Возвращаемый тип функции, не возвращающей ничего, единичный (<i>unit</i>) тип
Типы для поддержки Python	
<i>bytes</i>	Тип строки байт
<i>dict</i>	Тип ассоциативного массива
<i>list</i>	Тип списка

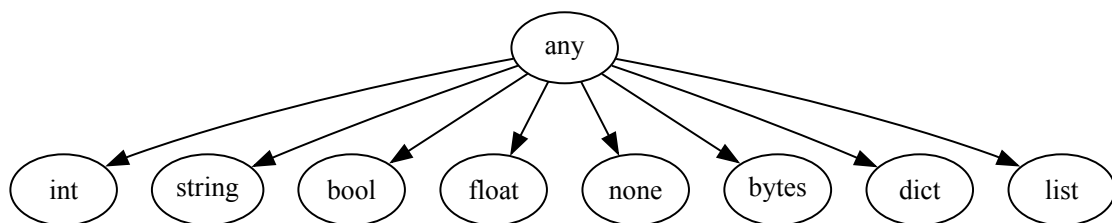


Рисунок 3.11 — Иерархия встроенных типов

Пользовательские типы являются подтипом *any* и не могут расширить правила типизации. В случаях, когда в описываемой библиотеке используется подтипирование, можно воспользоваться, по сути, структурной типизацией объектов, которая применяется в PT JSA. Таким образом, необходимо продублировать часть кода, а также использовать приведение типов.

В дальнейшем планируется реализовать механизм подтипирования для объектов.

3.1.13 Резюме

В данном разделе приведено описание разработанного DSL. Он содержит описания его синтаксиса, семантики и системы типов. Для основных возможностей приведены примеры в виде кода на нём. Подробно описана стандартная библиотека и механизм её расширения.

3.2 Разработка транслятора

Данный раздел содержит описание основных технических решений, которые были приняты при разработке транслятора с разработанного DSL в код расширений базы знаний анализатора кода PT JSA на языке C# Script. Полный код может быть найден в публичном репозитории¹

3.2.1 Архитектура транслятора

Упрощённая логическая схема проекта приведена на Рисунке 3.12. *Модуль интерфейса командной строки* позволяет конфигурировать транслятор, передавая исходные файлы на DSL, C# Script, а также указывая путь до директории с результирующим кодом. *Ядро транслятора* выступает посредником между другими модулями. *Модуль построения внутреннего представления (IR)* строит внутреннее представление с помощью лексера и парсера, которые генерируются автоматически при помощи antlr. *Модуль семантического анализа* ответственен за трансформацию внутреннего представления, а также за проверку типов и прочие проверки. *Транслятор IR->CGIR* конвертирует внутреннее представление, полученное на этапе семантического анализа в специальное представление, пригодное для генерации кода (CGIR). *Модуль синтеза* генерирует код на языке C# Script из CGIR.

¹Исходный код транслятора: <http://github.com/vldf/Master-thesis-DSL/>

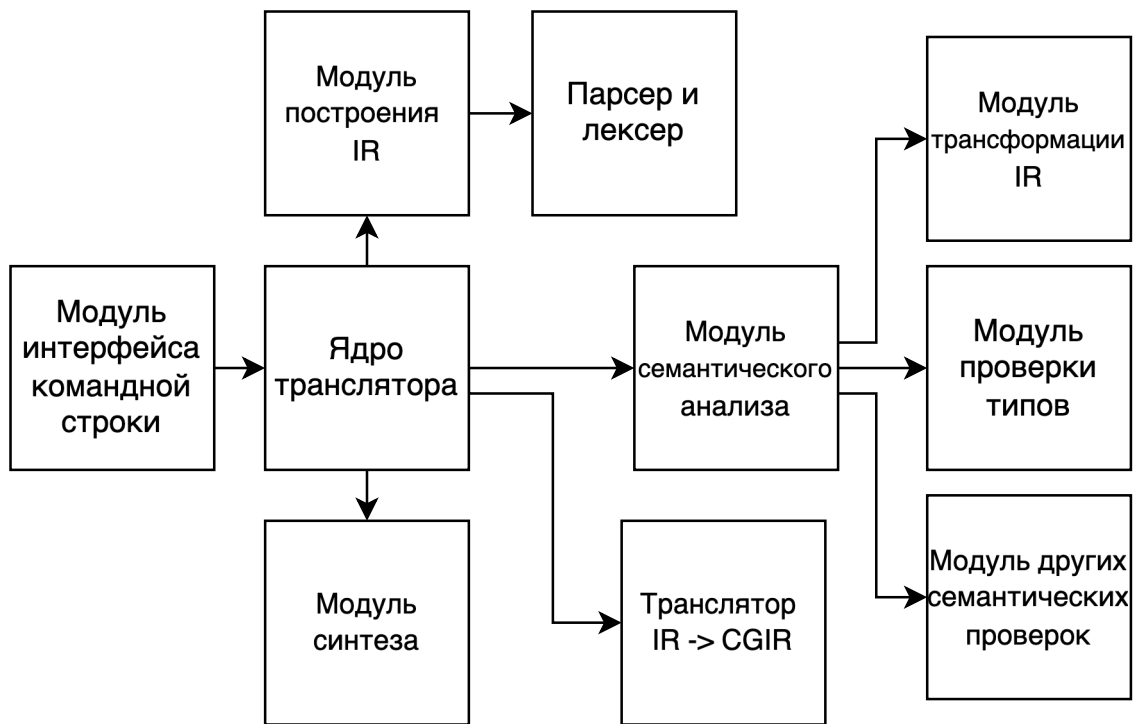


Рисунок 3.12 — Логическая архитектурная схема

На Рисунке 3.13 приведена верхоуровневая диаграмма потока данных, показывающая их преобразования в процессе трансляции.



Рисунок 3.13 — Верхоуровневая диаграмма потока данных при трансляции


3.2.2 Синтаксический анализ

Модуль построения внутреннего представления использует парсер и лексер для получения конкретного синтаксического дерева (*CST*, *concrete syntax tree*). Они генерируются автоматически при помощи инструмента antlr [11]. В качестве входных данных он принимает описания синтаксиса DSL на специализированном языке и порождает код лексера и парсера.

Стадия синтаксического анализа транслирует конкретное синтаксическое дерево во внутреннее представление, используемое анализатором. Это необходимо так как затруднительно расширить *CST*.

Для трансляции применяется два класса, реализующие шаблон «посетитель». Они наследуются от класса-посетителя, сгенерированного antlr. Первый необходим для обхода и конвертации представления выражений. Второй — для объявлений и определений, его фрагмент приведён в Листинге 3.9. Он создаёт экземпляр класса *IfStatementAstNode*,

который содержит информацию об условии ветвления и альтернативную ветку.

```
1  public override IStatementAstNode VisitIfStatement( 
2      JSADSLParser.IfStatementContext context)
3  {
4      var cond = VisitExpression(context.cond);
5      var mainBlockContext = new IrContext(irContext);
6      var mainBlockVisitor = new BaseBuilderVisitor(
7          mainBlockContext);
8
9      var mainBlock = mainBlockVisitor.VisitStatementsBlock(
10         context.mainBlock);
11     IStatementAstNode? elseBranch = null;
12
13     if (context.elseIfStatement != null)
14     {
15         var newContext = new IrContext(irContext);
16         var newVisitor = new BaseBuilderVisitor(newContext);
17
18         elseBranch = newVisitor.VisitIfStatement(
19             context.elseIfStatement);
20     } else if (context.elseBlock != null)
21     {
22         var newContext = new IrContext(irContext);
23         var newVisitor = new BaseBuilderVisitor(newContext);
24
25         elseBranch = newVisitor.VisitStatementsBlock(
26             context.elseBlock);
27     }
28
29     var ifStatementAstNode = new IfStatementAstNode(
30         cond, mainBlock, elseBranch);
31
32     mainBlock.Parent = ifStatementAstNode;
33     if (elseBranch != null)
34     {
35         elseBranch.Parent = ifStatementAstNode;
```

```

36      }
37
38      return ifStatementAstNode;
39  }

```

Листинг 3.9 — Фрагмент класса-посетителя для обработки ветвлений

3.2.3 Семантический анализ

Стадия семантического анализа преобразует внутреннее представление, полученное на этапе синтаксического анализа.

Семантический анализ использует классы, описывающие внутреннее представление. Основные из них приведены в Таблице 4. В ней колонка «тип» описывает роль соответствующего узла. Узел может быть объявлением, выражением, операцией или иметь другой тип. Колонка «Владение контекстом» обозначает, задаёт ли соответствующий узел свой контекст.

Контексты используются при разрешении ссылок и позволяют (при отсутствии синтаксических ограничений) создавать новые определения. Контексты образуют иерархию: существует корневой контекст файла, в котором может быть несколько дочерних контекстов для объектов, в каждом из которых может быть произвольное число контекстов функций и так далее. У корневых контекстов есть ссылки на контексты файлов, импортируемые в текущем. Таким образом, сущности в файле могут ссылаться на другие, определённые в подключенных к нему.

Таблица 4 — Основные классы внутреннего представления

Имя	Тип	Владение контекстом	Описание
<i>FileAstNode</i>	-	Да	Хранит имя пакета, а также список объявлений верхнего уровня.
<i>FunctionAstNode</i>	Объявление	Да	Хранит имя, список аргументов, тело и прочую информацию о функции.
<i>IntrinsicFunctionAstNode</i>	Объявление	Нет	Хранит имя и список аргументов, а также типовые параметры для внутренней функции.
<i>ObjectAstNode</i>	Объявление	Да	Хранит имя, объявление полей и функций объекта.
<i>VarDeclAstNode</i>	Объявление	Нет	Хранит имя, ссылку на тип и инициализирующее значение переменной.

Имя	Тип	Владение контекстом	Описание
<i>BinaryExpressionAstNode</i>	Выражение	Нет	Хранит операцию и операнды бинарного выражения.
<i>UnaryExpressionAstNode</i>	Выражение	Нет	Хранит операцию и операнд унарного выражения.
<i>FunctionCallAstNode</i>	Выражение, операция	Нет	Хранит ссылку на вызываемую функцию и список выражений-аргументов.
<i>IntrinsicFuncInvocationAstNode</i>	Выражение, операция	Нет	Описывает обычный вызов функции на целевом языке и хранит её имя, ресивер и набор аргументов.
<i>IntLiteralAstNode</i> , <i>FloatLiteralAstNode</i> , <i>StringLiteralAstNode</i> , <i>BoolLiteralAstNode</i> ,	Выражение	Нет	Хранят значение литерала соответствующего типа.
<i>NewAstNode</i>	Выражение	Нет	Хранит ссылку на тип инстанцируемого объекта и список аргументов-выражений.
<i>VarExpressionAstNode</i>	Выражение	Нет	Хранит ссылку на переменную для выражения.
<i>AssignmentAstNode</i>	Операция	Нет	Хранит ресивер и новое значение.
<i>IfStatementAstNode</i>	Операция	Нет	Хранит условие, блок главной ветки и альтернативную операцию или ветку.
<i>StatementsBlockAstNode</i>	-	Да	Блок-контейнер для операций.
<i>ReturnStatement</i>	Операция	Нет	Хранит опциональное возвращаемое из функции значение.
<i>ImportAstNode</i>	Операция	Нет	Подключает другой пакет к текущему файлу.

Основным этапом является этап трансформаций. В ней итеративно применяется набор правил переписывания внутреннего представления. Примерами являются упрощение синтаксического сахара [12] и частичная замена синтаксических конструкций на вызовы низкоуровневого API JSA. Основные трансформации:

- **Трансформация инициализирующего метода**

Модуль PT JSA, ответственный за анализ кода на Python, учитывает конструкторы классов, которые должны называться `__init__`. Для обеспечения возможности описания конструкторов, а также инициализирующих значений полей объектов, транслятор использует этот механизм. Для полей объектов с инициализирующими значениями генерируется код для их вычисления и присвоения, который вставляется в функцию с именем `__init__`, объявленную в объекте. Если она отсутствует, то функция с таким именем будет синтезирована этой стадией

- **Трансформация ссылок**

Для обеспечения возможности предварительного определения (*forward declaration*), в трансляторе применяется механизм ссылок. Каждая из ссылок привязана к своему контексту и по ней можно получить ссылаемый объект, если он доступен из соответствующего контекста. На этом трансформации фактического изменения IR не происходит, однако, оно обходится и для каждой ссылки происходит её запечатывание: если её разрешение происходит успешно, то разрешённая сущность сохраняется в специальное поле внутри ссылки и каждое следующее разрешение будет возвращать одно и то же значение. Если разрешение не удалось, то это говорит о некорректной программе и возбуждается ошибка.

- **Трансформатор ветвлений**

Данное переписывание изменяет внутреннее представление, заменяя узлы ветвлений на использование соответствующего низкоуровневого API анализатора. Использование ветвлений языка C# Script невозможно, так как значение условия является символьным значением, конкретное значение которого определяется анализатором в момент детектирования уязвимостей.

JSA предоставляет большой набор методов для поддержки ветвлений. Основными из них являются *TryEnterBranch*, принимающий условие попадания в ветку, а также *LeaveBranch*. Они оба принимают специальный идентификатор ветки и сигнализируют о входе и выходе из неё анализатору. При входе в ветку PT JSA автоматически управляет символьным состоянием, добавляя соответствующее условию символьное выражение в текущие ограничения.

Таким образом, после этой трансформации код становится линейным с синтаксической точки зрения, но не с семантической

- **Трансформация операций возврата значения из функции**

В JSA каждый из обработчиков описываемой функции возвращает символьное значение в специальной обёртке. Возврат осуществляется при помощи оператора `return` языка C# Script. Для возможности раннего возврата из ветвления анализатор предоставляет низкоуровневый API. Он использует так называемые кондиционалы, которые можно представить как пары (*текущие ограничения из условий ветвлений; возвращаемое значение*). API позволяет автоматически получать текущие ограничения, передавая только возвращаемое значение. Так как сам кондиционал является символьным выражением, анализатор поддерживает его возврат из функции-обработчика, что и происходит после применения этого переписывания.

- **Трансформация вызовов функций**

Вызов обычных функций происходит при помощи API анализатора и так как соответствующий код требует указания обработчика вызываемой функции, его генерация происходит на этапе кодогенерации.

Отдельно обрабатываются вызовы внутренних функций. Так как они не являются частью описываемой библиотеки, их вызов происходит напрямую, как вызов обычной функции языка C# Script. Для этого трансформация заменяет узел внутреннего представления *FunctionCallAstNode* на *IntrinsicFunctionInvocationAstNode*, что позволяет учесть это различие в поведении при кодогенерации. Дополнительно, по соглашению, это переписывание добавляет аргумент типа `Location`.

Семантические проверки — ключевой этап разработки любого транпилятора. Они позволяют отлавливать ошибки, которые программист допустил при написании кода. Такие дефекты могут быть связаны с типизацией. Этому классу ошибок уделено отдельное внимание. Однако, в коде могут присутствовать и другие. К примеру, использование ещё не объявленной переменной и вызов несуществующей функции.

Для их обнаружения, а также для проверки и вывода отсутствующих типов реализована отдельная подсистема. Она включает в себя абстрактный класс *AbstractCheckerBase* с единственным методом *Check()*, а также его наследника класс-посетитель *AbstractChecker<TR>*, который позволяет обходить всё внутреннее представление программы.

В Листинге 3.10 представлен фрагмент кода, проверяющего типы. Метод *CheckBinArithmeticExpression* вызывается для каждого узла бинарного выражения. Аргументы *left* и *right* содержат левую и правую его часть. В начале метод определяет типы операндов. В случае если возвращаемое значение *CheckExpression* является нулевым указателем, весь метод также

возвращает его. Возвращение нулевого указателя показывает, что на анализируемом выражении произошла ошибка, а значит, его дальнейшую обработку можно пропустить. Затем, метод проверяет, что типы операндов совпадают. Так как в языке нет неявного приведения типов, это поведение корректно. Далее, метод проверяет, что типы операндов являются числовыми. Это следует из семантики языка. Если все проверки выполнены успешно, возвращается тип всего выражения, равный типу любого из аргумента.

```
1 private AstType? CheckBinArithmeticExpression(  
2     IExpressionAstNode left,  
3     IExpressionAstNode right)  
4 {  
5     var leftType = CheckExpression(left);  
6     if (leftType == null) {  
7         return null;  
8     }  
9  
10    var rightType = CheckExpression(right);  
11    if (rightType == null) {  
12        return null;  
13    }  
14  
15    if (!CheckTypes(leftType, rightType))  
16    {  
17        errorManager.Report(  
18            Error.TypeMismatch(leftType, rightType));  
19        return null;  
20    }  
21  
22    if (!_numericTypes.Contains(leftType))  
23    {  
24        errorManager.Report(  
25            Error.NumericTypeExpected(leftType));  
26    }  
27  
28    if (!_numericTypes.Contains(rightType))  
29    {  
30        errorManager.Report(  

```

```

31         Error.NumericTypeExpected(leftType));
32     }
33
34     return rightType;
35 }

```

Листинг 3.10 — Фрагмент проверки типов бинарного выражения

Аналогичным образом происходит проверка всех остальных типов. Отдельно учитываются случаи, когда тип не указан в коде. В этом случае происходит вывод типов с последующим его сохранением. Таким образом решается задача расстановки отсутствующих типов для последующего анализа и кодогенерации.

Так как предполагается будущее расширение DSL для описания библиотек на различных языках, в которых семантика различается, представлены специальные проверки, зависящие от конкретного языка. Для Python реализована проверка сигнатуры функции `__init__`, если она представлена. Проверяется что её возвращаемый тип либо отсутствует, либо совпадает с именем объекта, в котором она написана. Это совпадает с семантикой Python. Проверяется отсутствие оператора `return` в этом конструкторе.

3.2.4 Генерация целевого кода

Процесс генерации кода начинается с конвертации внутреннего представления *IR* в специализированное внутреннее представление *CGIR*. Это представление было реализовано специально для проекта за неимением существующих для генерации кода на C# Script. Представлено большое количество библиотек для генерации обычного кода на C#, однако, C# Script является его надмножеством. К примеру, он допускает декларации из тел методов на верхнем уровне, что активно используется для расширения PT JSA. В отличие от *IR*, *CGIR* не является сильно-типизированным что упрощает процесс конвертации. Также, оно приближено к синтаксису и семантики целевого языка.

Для перевода *IR* в *CGIR* используется шаблон посетитель. Внутреннее представление семантического анализа обходится и порождается новое, пригодное для генерации. Фрагмент кода для генерации ветвлений приведён в Листинге 3.11. В нём сначала генерируется *CGIR* для выражения условия. Затем, происходит размещение нового контейнера для операций в текущий контекст. Это сделано для упрощения порождения *CGIR*. Каждая из функций конвертера просто помещает очередной оператор в текущий контейнер.

После помещения нового контейнера для основной ветки, происходит генерация её тела, а затем, в зависимости от типа альтернативной ветки. Если это другой оператор ветвления, то происходит рекурсивный вызов, результат которого сохраняется в новое представление. В случае если там располагается блок кода, то происходит его кодогенерация. В самом конце, метод возвращает результат конвертации.

```
1  private CgIfElseStatement AsIfStatement(C#
2      IfStatementAstNode node)
3      {
4          var cond = _expressionsEmitter.EmitExpression(
5              node.Cond);
6          var ifStatement = new CgIfElseStatement(cond);
7
8          ctx.PushContainer(ifStatement.MainBody);
9          EmitStatementBlockAstNode(node.MainBlock);
10         ctx.PopContainer();
11
12         if (ifStatement.Elseif != null)
13         {
14             switch (node.ElseStatement)
15             {
16                 case IfStatementAstNode elseIf:
17                     ifStatement.Elseif = AsIfStatement(elseIf);
18                     break;
19                 case StatementsBlockAstNode elseBlock:
20                     ifStatement.ElseBody = new
21                         CgStatementsContainer();
22                     ctx.PushContainer(ifStatement.ElseBody);
23                     EmitStatementBlockAstNode(elseBlock);
24                     ctx.PopContainer();
25                     break;
26                 default:
27                     throw new ArgumentOutOfRangeException(
28                         nameof(node.ElseStatement));
29             }
30
31             return ifStatement;
```


Листинг 3.11 — Фрагмент конвертации ветвления в CGIR

После конвертации происходит порождение кода на языке C# Script по *CGIR*. Это происходит путём обхода этого представления. Фрагмент генерации текста приведён в Листинге 3.12. Метод *SynthNewExpression* предназначен для генерации оператора *new*. Стоит заметить, что этот оператор совершенно не совпадает по семантике с одноимённым на DSL, однако, он активно используется в целевом коде. Методы *Append* и *AppendSpace* являются вспомогательными и, по своей сути, добавляют соответствующие символы во внутренний *StringBuilder*. Поддерживается автоматическое расстановка отступов блока кода при необходимости, что немного улучшает читаемость генерируемого кода.

```
1  private void SynthNewExpression(C#
2      CgNewExpression cgNewExpression)
3  {
4      Append("new");
5      AppendSpace();
6      Append(cgNewExpression.TypeName);
7      Append("(");
8      var argsDropLast = cgNewExpression.Args.SkipLast(1);
9      foreach (var cgExpression in argsDropLast)
10     {
11         SynthExpression(cgExpression);
12         Append(", ");
13     }
14
15     var lastArg = cgNewExpression.Args.LastOrDefault();
16     if (lastArg != null)
17     {
18         SynthExpression(lastArg);
19     }
20
21     Append(")");
22 }
```

Листинг 3.12 — Фрагмент генерации целевого кода

После генерации целевого кода, работа транслятора завершается. Этот процесс проходят все файлы на DSL, подаваемые на вход.

3.3 Особенности языка и транслятора для поддержки Python

Как было установлено ранее, в качестве демонстрации было принято решение ограничиться поддержкой написания расширений только для модуля PT JSA для анализа кода на Python. Предполагается дальнейшее развитие проекта, в том числе, для поддержки других языков. Этим продиктованы некоторые технические решения, основным из которых является отделение основной семантики DSL от семантики модуля анализа.

Одним из примеров такого разделения является описание конструкторов объектов. Так как поведение конструкторов существенно различается у популярных языков программирования, было решено повторить эту семантику без дополнительного синтаксиса DSL. Так, для объявления конструктора необходимо добавить функцию с именем `__init__` и аргументами, которые принимает конструктор описываемого объекта. Это полностью повторяет аналогичную семантику языка Python.

Другой особенностью, которая была поддержана, но уже в трансляторе, является специализированный низкоуровневый API, предоставляемый PT JSA. В частности, этот API позволяет использовать нетипизированные объекты. Они могут быть порождены, к примеру, функцией `CreateDataOfType<any>`. Тип `any` является специальным и указывает, что либо тип объекта неизвестен, либо он неважен.

Для реализации будущей интеграции с другими языковыми модулями, были заложены архитектурные основы проекта. Код, специфичный для Python, был вынесен в отдельные классы и инкапсулирован в них. В прочем, для этой цели предполагается значительная переработка кода, написанного в большей степени для демонстрации идеи.

3.3.1 Резюме

В данном разделе приведены основные архитектурные и технические решения, стоящие за разработанным транслятором с DSL в C# Script. Подробно описаны внутренние структуры данных, такие как IR и CG IR, и их преобразования. Рассмотрены все основные стадии работы транслятора.

3.4 Резюме

В данной Главе описан разработанный прототипа DSL с точки зрения его синтаксиса и семантики. Используются различные формализмы, такие как грамматические правила и правила типизации. Приводятся и словесные описания для всех языковых возможностей. Затем, происходит описание разработанного прототипа его транслятора в низкоуровневый код

на C# Script. Демонстрируется его архитектура, а также детали реализации синтаксического и семантического анализа, генерации кода. В заключение приводятся различные решения как в DSL, так и в трансляторе, необходимые для поддержки Python.

ГЛАВА 4. АПРОБАЦИЯ ПОЛУЧЕННЫХ ИНСТРУМЕНТОВ

Данная глава содержит результаты апробации DSL и его транслятора в низкоуровневый код расширений PT JSA.

4.1 Подход к апробации

Основной зоной применения PT JSA является анализ кода web-приложений на безопасность. Так как прототип DSL и его транслятора предназначены для описания библиотек и фреймворков на языке Python, проекты, приведённые в этой главе, также написаны на этом языке. Они показывают реальные способы написания кода серверов.

В Главе рассмотрено несколько проектов. Каждый из них показывает пример использования библиотеки или фреймворка наиболее популярными способами, а также код для расширения JSA с поддержкой их. Список компонентов для демонстрации был составлен с учётом их популярности. Стоит отметить, что целью данной Главы не является проверка и демонстрация функциональных возможностей самого анализатора. По этому, примеры содержат простой код, целью которого является демонстрация сценариев использования внешних компонентов. Проекты не содержат намеренно сложный код, который мог бы привести к ложным срабатываниям анализатора.

Для апробации, код на DSL транслируется в низкоуровневый код для расширений, анализатор PT JSA запускается из терминала. Он сохраняет запись в журнал о каждой найденной уязвимости. Для краткости, пояснительная информация содержит только основные значения, по которым можно определить корректность написания расширения.

Исходный и сгенерированный коды проектов располагаются в поддиректории соответствующего проекта в директории Examples².

4.2 Библиотека для отправки HTTP-запросов

Одной из популярных операций в серверных web-приложениях является отправка HTTP-запросов. Она необходима как для интеграции со внешними сервисами, так и для коммуникациями между микросервисами одной системы.

Библиотека urllib3 предоставляет возможности для отправки HTTP-запросов с различными опциями и получения ответов. Листинг 4.1 содержит пример кода обработки запроса с использованием фреймворка Flask.

²<https://github.com/vldF/Master-thesis-DSL/tree/main/Examples/>

```

1  @app.route('/save_profile_image', methods=['GET'])
2  def save_profile_image():
3      image_url = request.args.get('image_url')
4      temp_file = tempfile.TemporaryFile()
5      download_image(image_url, temp_file)
6
7      process_image(temp_file)
8
9      return
10
11 def download_image(url, filename):
12     token = get_system_token()
13     # уязвимость: подделка запросов со стороны сервера
14     response = urllib3.request('GET', url, { 'auth_token':
15         token })
16
17     with open(filename, 'wb') as f:
18         f.write(response.data)
19
20     print(f"Image saved as {filename}")

```

Листинг 4.1 — Пример использования библиотеки urllib3

В Листинге 4.1 описана функция-обработчик HTTP GET запроса по относительному адресу `/save_profile_image`. Можно считать, что она позволяет сохранить фотографию пользователя после её загрузки на сервер. Обработчик принимает аргумент с именем `image_url`, содержащий адрес загруженного изображения и загружает его с использованием системного токена для идентификации и аутентификации. Для упрощения, последующая обработка данных, не относящаяся к демонстрации, была опущена. Так как в коде отсутствует проверка URL фотографии, клиент может совершить атаку, отправив произвольный адрес. Это может привести к:

- Отправке запросов с системным токеном на произвольный сервер;
- Загрузке файла с произвольным содержимым.

Первый пункт приведёт к раскрытию чувствительной информации, в то время как второй позволит атакующему загрузить файл с вредоносным кодом на сервер. Возможна также загрузка большого файла, что приведёт к отказу в обслуживании (*DOS*). Данная уязвимость называется «подделка запросов со стороны сервера» (*Server-side request forgery*), подробнее она описана [13].

Для добавления поддержки библиотеки `urllib3` в JSA был написан код на DSL, фрагмент которого приведён в Листинге 4.2. Он состоит из трёх файлов, которые подписаны комментариями. Полный текст приведён в директории `Examples/urllib3/dsl`³

```
1 // файл urllib3.response.jsadsl DSL
2 package "urllib3.response";
3
4 import "Standard";
5 import "urllib3.connection";
6
7 object HTTPResponse {
8     var status = CreateTaintedDataOfTypes<int>("Second order");
9     var data = CreateTaintedDataOfTypes<bytes>("Second order");
10    var body = CreateTaintedDataOfTypes<string>("Second order");
11    var url = CreateTaintedDataOfTypes<string>("Second order");
12    var request_url = CreateTaintedDataOfTypes<string>("Second order");
13
14    var connection = new HTTPConnection();
15
16    func json(): any {
17        return CreateTaintedDataOfTypes<any>("Second order");
18    }
19
20    func readline(): string {
21        return CreateTaintedDataOfTypes<string>("Second order");
22    }
23
24    func read(): any {
25        return CreateTaintedDataOfTypes<any>("Second order");
26    }
27
28    func fileno(): any {
29        return CreateTaintedDataOfTypes<any>("Second order");
30    }
31    // аналогичное описание ещё трёх функций
```

³<https://github.com/vldF/Master-thesis-DSL/tree/main/Examples/urllib3>

```

32 }
33
34
35 // файл urllib3.connection.jsadsl
36 package "urllib3.connection";
37
38 import "Standard";
39 import "urllib3.response";
40
41 object HTTPConnection {
42     func getresponse(): HTTPResponse {
43         return new HTTPResponse();
44     }
45
46     func request(
47         method: string = "GET",
48         url: string,
49         body: any = none,
50         fields: any = none,
51         headers: any = none,
52         json: any = none
53     ): HTTPResponse {
54         Detect(url, "Server-Side Request Forgery", "HTTP URI");
55         Detect(headers, "Server-Side Request Forgery", "HTTP URI");
56         Detect(headers, "Information Exposure", "HTTP URI");
57
58         return new HTTPResponse();
59     }
60
61
62     func request_chunked(url: string) {
63         Detect(url, "Server-Side Request Forgery", "HTTP URI");
64     }
65 }
66
67
68 // файл urllib3.jsadsl

```

```

69 package "urllib3";
70
71 import "Standard";
72 import "urllib3.response";
73
74 func request(
75     method: string = "GET",
76     url: string,
77     body: any = none,
78     fields: any = none,
79     headers: any = none,
80     json: any = none
81 ): HTTPResponse {
82     Detect(url, "Server-Side Request Forgery", "HTTP URI");
83     Detect(headers, "Server-Side Request Forgery", "HTTP
84     URI");
85     Detect(headers, "Information Exposure", "HTTP URI");
86
87     return new HTTPResponse();
88 }

```

Листинг 4.2 — Код расширения для поддержки urllib3

В Листинге 4.2 объект *HTTPResponse* описывает для PT JSA одноимённый класс библиотеки urllib3. Он содержит несколько полей, содержащие потенциально загрязнённые данные, такие, как *status* и *data*. Листинг содержит объект типа *HTTPConnection*, а также несколько функций. К примеру, функция *json* в urllib3 возвращает тело ответа на запрос, десериализованное из JSON в произвольный объект. С точки зрения taint-анализа, она является источником загрязнённых данных, по этому, в коде её результатом является результат вызова функции *CreateTaintedDataOfType<any>*, которая создаёт такой объект произвольного типа.

Функции *request* объекта *HTTPConnection* и одноимённая функция пакета *urllib3* отправляют HTTP-запрос по указанному URL. Они содержат большое количество аргументов по-умолчанию. Каждая из них запускает операцию определения на аргументах *url* и *headers*. В проверке для URL содержится уязвимость SSRF, в проверке для заголовков SSRF и Information Exposure (утечка чувствительной информации).

Информация об уязвимости, найденной в коде в Листинге 4.1 с расширением из Листинга 4.2 приведена в Таблице 5. В ней видно, что найдена уязвимость указанного в коде на DSL типа на строке 13. Можно заметить, что трасса данных приведена также корректно.

Таблица 5 — Информация об уязвимости

Параметр	Значение
Функция <i>save_profile_image</i>	
Уязвимость «подделка запросов со стороны сервера»	
Уязвимое выражение	<code>response = urllib3.request('GET', url, { '...</code>
Точка входа	<code>def save_profile_image():</code>
Трасса данных	<code>image_url = request.args.get('image_url') download_image(image_url, temp_file) response = urllib3.request(...{ 'auth_token': token })</code>

Этот пример показывает, что разработанный DSL позволяет расширять базу знаний PT JSA элементарными библиотеками и позволяет описывать их с точки зрения анализа потока данных с достаточной для анализа точностью.

4.3 Библиотека для работы с базой данных


В Python существует несколько популярных библиотек для работы с базами данных. В рамках демонстрации была выбрана библиотека `psycopg2`, предоставляющая набор всех базовых операций для взаимодействия с СУБД `postgres`. Выбор библиотеки для взаимодействия с базой данных закономерен: по многим исследованиям ([14,15]), операции с ними часто приводят к уязвимостям.

Листинг 4.3 содержит фрагмент web-приложения, построенного на фреймворке `Flask` и использующего библиотеку `psycopg2`.

```

1  @app.route("unsafe/users/description/<user_id>/",
    methods=["GET"])
2  def get_user_description():
3      user_id = request.args.get('user_id')
4      conn = connect("dbname=test user=postgres")
5      cur = conn.cursor()
6      # уязвимость: внедрение SQL-кода
7      cur.execute("SELECT description FROM table WHERE ID = " +
    user_id)
8      description = cur.fetchone()
9      # уязвимость второго порядка: межсайтовый скриптинг
10     return description

```

 Python

```

11
12 @app.route("unsafe/users/<user_id>/", methods=["GET"])
13 def get_user1():
14     user_id = request.args.get('user_id')
15     conn = connect("dbname=test user=postgres")
16     cur = conn.cursor()
17     # уязвимость: внедрение SQL-кода
18     cur.execute("SELECT * FROM table WHERE ID = " + user_id)
19     user = cur.fetchone()
20     # нет уязвимости
21     return jsonify(user)
22
23 @app.route("safe/users/<user_id>/", methods=["GET"])
24 def get_user2():
25     user_id = request.args.get('user_id')
26     conn = connect("dbname=test user=postgres")
27     cur = conn.cursor()
28     # нет уязвимости
29     cur.execute("SELECT * FROM table WHERE ID = %s",
30                 (user_id, ))
31     user = cur.fetchone()
32     # нет уязвимости
33     return jsonify(user)

```

Листинг 4.3 — Фрагмент кода приложения с использованием psycopg2

При использовании библиотеки psycopg2 используется функция *connect*, устанавливающая подключение к БД и возвращает объект типа *Connection*. Он содержит функцию *cursor*, которая возвращает объект типа *Cursor*, позволяющий обращаться к базе данных. Его функции *execute* и *executemany* позволяют отправить запрос, функции *fetchone*, *fetchmany* и *fetchall* возвращают результат запроса.

В Листинге 4.3 *get_user_description* содержит сразу две уязвимости: внедрение SQL-кода и уязвимость второго порядка типа межсайтовый скриптинг. Уязвимостью второго порядка называется уязвимость, уязвимые данные для которой сначала сохраняются в некоторое состояние (в оперативную память, в базу данных), а затем возвращаются пользователю в результате другого запроса. Фактически, их можно описать фразой «уязвимость из-за уязвимости». В данном примере атакующий мог бы установить описание пользователя на HTML текст с скриптом на javascript

(что само по себе не безопасно). Получение этого описания другим пользователем привело бы к уязвимости. В этой функции одним из способов исправления дефекта может быть экранирование данных, что демонстрируется в функции *get_user1*. В прочем, в *get_user1* всё ещё присутствует дефект типа «внедрение SQL-кода». Её исправление приведено в функции *get_user2*, где используется механизм библиотеки *psycpg2*, экранирующий значения при формировании SQL-запроса по шаблону на строке 29.

Для расширения базы знаний PT JSA был написан код на DSL, приведённый в Листинге 4.4. В нём приведены три файла. Файл *psycpg2.jsadsl* содержит функцию *connect*, которая возвращает объект типа *Connection* вне зависимости от строки подключения к БД. Объект *Connection* содержит функцию *cursor*, возвращающую объект типа *Cursor*, который содержит функции для исполнения запросов и возврата результатов. Таким образом моделируется общая структура библиотеки. Функции *execute* и *executemany* содержат код, запускающий поиск уязвимых данных на их аргументах. Это позволяет, в частности, определять случаи конкатенации к SQL запросам потенциально уязвимых данных. Функции *fetchone*, *fetchmany* и *fetchall* возвращают данные соответствующего типа с taint-меткой и источником вида «Second Order», обозначающих, что это данные, которые могут привести к уязвимости второго порядка.

```
1 // файл psycpg2.jsadsl DSL
2 package "psycpg2";
3 import "psycpg2.Cursor";
4 import "psycpg2.Connection";
5 func connect(connection_string: any): Connection {
6     return new Connection();
7 }
8
9 // файл psycpg2.Connection.jsadsl
10 package "psycpg2.Connection";
11 import "Standard";
12 import "psycpg2.Cursor";
13 object Connection {
14     func cursor(): Cursor {
15         return new Cursor();
16     }
17 }
```

```

18
19 // файл psycpg2.Cursor.jsadsl
20 package "psycpg2.Cursor";
21 import "Standard";
22 object Cursor {
23     func execute(
24         query: string,
25         vars: list = CreateDataOfT<list>()
26     ) {
27         Detect(query, "SQL Injection", "SQL common");
28     }
29     func executemany(
30         query: string,
31         vars: list = CreateDataOfT<list>()
32     ) {
33         Detect(query, "SQL Injection", "SQL common");
34     }
35     func fetchone(): any {
36         return CreateTaintedDataOfT<any>(
37             "Second Order");
38     }
39     func fetchmany(): list {
40         return CreateTaintedDataOfT<list>(
41             "Second Order");
42     }
43     // ...
44 }

```

Листинг 4.4 — Код для добавления поддержки psycpg2 в PT JSA

Уязвимости, которые были найдены анализатором PT JSA в коде, приведённом в Листинге 4.3 с помощью расширения в Листинге 4.4, приведены в Таблице 6. Из неё следует, что все уязвимости в коде, описанные ранее, были обнаружены и были обнаружены только они.

Таблица 6 — Информация об уязвимостях

Параметр	Значение
Функция <i>get_user_description</i>	
Уязвимость «внедрение SQL-кода»	
Уязвимое выражение	cur.execute("SELECT description FROM table...
Точка входа	@app.route("users/description/<user_id>/",:
Трасса данных	user_id = request.args.get('user_id') cur.execute("SELECT description FROM table...
Уязвимость «межсайтовый скриптинг»	
Уязвимое выражение	return description
Точка входа	@app.route("users/description/<user_id>/", ...
Трасса данных	description = cur.fetchone() return description
Функция <i>get_user1</i>	
Уязвимость «внедрение SQL-кода»	
Уязвимое выражение	cur.execute("SELECT * FROM table...
Точка входа	@app.route("users/<user_id>/", ...
Трасса данных	user_id = request.args.get('user_id') cur.execute("SELECT * FROM table...

Таким образом, разработанный DSL пригоден для добавления в PT JSA поддержки библиотек для работы с базами данных.

4.4 Фреймворк для обработки HTTP-запросов

Для Python предоставлено большое число фреймворков, позволяющих разрабатывать серверные приложения, обрабатывающие HTTP-запросы от клиентов. Одним из популярных является фреймворк Flask. Он позволяет с использованием декоратора *@route* объявить функцию обработчиком запроса. Эта функция может содержать аргументы, которые могут быть получены и альтернативным способом — при помощи поля *request.args*, которое является ассоциативным массивом. Flask также предоставляет большое число утилитарных функций, таких как *jsonify*, которая преобразует аргумент произвольного типа в JSON. Она позволяет экранировать данные в полях сериализуемого объекта, по этому, можно рассматривать её как фильтрующую функцию.

В Листинге 4.5 приведён код HTTP сервера с использованием этого фреймворка. Он состоит из двух обработчиков запросов: функции *auth*, имитирующей идентификацию и аутентификацию по логину и хешу пароля, а также функции *get_current_user*, возвращающей информацию о текущем пользователе. В Листинге представлены и вспомогательные функции: *validate_login_and_password*, проверяющая корректность логина и

хеши пароля записи в БД, а также *get_user_by_token*, получающая из БД запись о пользователе по его токену аутентификации.

```
1  @app.route("/auth/<string:redirect_url>")
2  def auth(redirect_url: str):
3      login = request.args["user_login"]
4      pass_hash = request.args["user_login"]
5      if not validate_login_and_password(login, pass_hash):
6          return "Login failed", 401
7
8      token = get_user_token(login)
9      # уязвимость: подделка запросов со стороны сервера
10     return redirect(f"redirect_url?token={token}")
11
12 def validate_login_and_password(login: str, pass_hash: str) -
    > bool:
13     conn = connect("dbname=test user=postgres")
14     cur = conn.cursor()
15     cur.execute("SELECT * FROM users WHERE login = %s",
16                 (login, ))
17     user = cur.fetchone()
18     return user is not None and user.pass_hash == pass_hash
19
20 @app.route("/user/get_me/<string:token>")
21 def get_current_user(token: str):
22     current_user = get_user_by_token(token)
23     if current_user is None:
24         return f"invalid token: {token}"
25
26     return jsonify(current_user)
27
28 def get_user_by_token(token: str):
29     conn = connect("dbname=test user=postgres")
30     cur = conn.cursor()
31
32     cur.execute("SELECT * FROM user_tokens WHERE token = %s",
33                 (token, ))
34     token_record = cur.fetchone()
35     if token_record is None:
```

```

34         return None
35
36     cur.execute("SELECT * FROM users WHERE login = %s",
37               (token_record.login, ))
38     user = cur.fetchone()
39     return user

```

Листинг 4.5 — Фрагмент кода web-приложения на Flask

Функция-обработчик *auth* в Листинге 4.5 получает от клиента логин, хеш пароля, а также адрес, на который пользователь будет перенаправлен в случае успешной аутентификации. Стоит заметить, что в коде полностью отсутствует проверка адреса для перенаправления. Так как он принимается в запросе от пользователя, атакующий может проставить туда произвольный адрес, на который при удачной аутентификации отправится токен пользователя с помощью параметра *token*. Таким образом, это пример уязвимости типа «открытое перенаправление» (*Open redirect*).

Функция *get_current_user* получает запись о пользователе по его токену. В случае, если она не может найти его, возвращается ошибка, которая содержит сам токен. Так как токен передается клиентом, атакующий может передать намеренно данные, не проходящие проверку и содержащие произвольный код. Так как этот код возвращается достоверным сервером, то у него будет доступ к механизмам браузера, которые используются для хранения настоящего токена. Так, эта информация может быть передана на внешний сервер и злоумышленник получит к ней доступ. Для исправления этой уязвимости можно воспользоваться функцией *escape*, которую предоставляет Flask. Она экранирует HTML теги, что лишает злоумышленника возможность провести такую атаку. Также, можно просто не возвращать некорректный токен.

Flask — сложный для поддержки фреймворк. Текущих возможностей DSL не хватает для достаточной его поддержки. К примеру, во Flask применяются декораторы, которые, с точки зрения семантики python, являются синтаксическим сахаром для композиции функций. Таким образом, для описания декораторов необходима поддержка функциональных значений и типов, которая отсутствует в прототипе. Однако, так как предоставляются возможности для написания части кода расширения с использованием низкоуровневого API на C# Script, эти ограничения можно обойти, что и было сделано для апробации. Так, расширение состоит из двух файлов. Файл *flask.jsadsl* (см. Листинг 4.6), содержащий, помимо прочего, объект *Flask*,

помеченный аннотацией `@GeneratedName("FlaskClassDescriptor")`. Таким образом, объект *Flask* может быть расширен из низкоуровневого API на C# Script. Файл *flaskComplementation.jsa* содержит такой код, с ним можно ознакомиться в файле `Examples/flask/sharp`⁴.

```
1  import "Standard";
2
3  @GeneratedName("FlaskClassDescriptor")
4  object Flask { }
5
6  func url_for(endpoint: any): string {
7      return CreateDataOf<string>();
8  }
9
10 object Request {
11     var args = CreateTaintedDataOf<dict>("Query");
12     var data = CreateTaintedDataOf<dict>("Body");
13     // ...
14     var blueprint = CreateDataOf<string>();
15     var endpoint = CreateDataOf<any>();
16
17     func get_json(): any {
18         return CreateTaintedDataOf<any>("Body");
19     }
20 }
21
22 var request = new Request();
23
24 func escape(data: string): string {
25     var escaped = WithoutVulnerability(data, "Cross-site Scripting");
26     escaped = WithoutVulnerability(data, "Server-Side Template Injection");
27 }
28
29 func jsonify(data: any): any {
```

⁴<https://github.com/vldF/Master-thesis-DSL/blob/main/Examples/flask/sharp/flaskComplementation.jsa>


```

30     return CreateDataOfAnyType<any>();
31 }
32
33 func redirect(url: string, code: int = 302, response: any =
    none): any {
34     Detect(url, "Open redirect", "HTTP URI");
35     Detect(response, "Cross-site Scripting", "HTTP URI");
36
37     return CreateDataOfAnyType<any>();
38 }

```

Листинг 4.6 — Фрагмент кода расширения для поддержки Flask

Листинг 4.6 содержит фрагмент файла расширения. Помимо объекта *Flask*, в нём расположен объект *Request*, содержащий большое количество полей-источников taint-данных (их часть скрыта для краткости). Представлены и поля с данными без метки (*blueprint* и *endpoint*). Обратим внимание на функцию *get_data*. В зависимости от значения аргумента *as_text*, она возвращает либо строку байт, либо обычную строку с taint-метками. Функция *escape*, располагающаяся на глобальном уровне, сообщает анализатору об отфильтровывании данных таким образом, что уязвимость типа «Cross-site Scripting» (межсайтовый скриптинг) больше не может быть обнаружена на соответствующем потоке данных. Функция *jsonify* возвращает данные произвольного типа без taint-метки, моделируя преобразование данных в JSON с экранированием. Функция *redirect* запускает обнаружение уязвимостей типа «Open redirect» (открытый редирект) на адресе перенаправления и «Cross-site Scripting» на аргументе ответа от сервера.

Таблица 7 содержит уязвимости, обнаруженные анализатором JSA на коде из Листинга 4.5 с разработанным расширением.

Таблица 7 — Информация об уязвимостях

Параметр	Значение
Функция <i>auth</i>	
Уязвимость «открытое перенаправление»	
Уязвимое выражение	return redirect(f" {redirect_url} ?token={token}")
Точка входа	@app.route("/auth/<string:redirect_url>")
Трасса данных	@app.route("/auth/<string:redirect_url>") login = request.args["user_login"]« token = get_user_token(login) return redirect(f" {redirect_url} ?token={token}")»
Функция <i>get_current_user</i>	

Параметр	Значение
Уязвимость «межсайтовый скриптинг»	
Уязвимое выражение	<code>return f"invalid token: {token}"</code>
Точка входа	<code>@app.route("/user/get_me/<string:token>")</code>
Трасса данных	<code>@app.route("/user/get_me/<string:token>")</code> <code>return f"invalid token: {token}"</code>

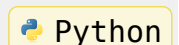
В Таблице 7 видно, что все описанные ранее уязвимости были найдены, а все представленные значения являются корректными. Таким образом, несмотря на ограничения текущей реализации прототипа, благодаря совместимости с C# Script можно описывать и такие сложные фреймворки как Flask.

4.5 Пользовательская библиотека для получения информации об IP

Рассмотренные ранее библиотеки могли бы оказаться в базе знаний PT JSA благодаря его производителю, так как они являются популярными. Однако, существует большое количество компонентов, которые не могут быть добавлены производителем. Например, некоторые из компаний-пользователей PT JSA имеют собственные разработки библиотек. Одним из них может являться компонент для определения информации об IP адресе. Это необходимо, в частности, для отображения данных об активности пользователя. Эта функциональная возможность предоставляется большим количеством сервисов и позволяет обнаруживать подозрительную активность на своём аккаунте. Для демонстрации была разработана библиотека, использующая API сервиса `ipgeolocation.abstractapi.com`. Исходный код клиентов может быть обнаружен в `Examples/abstractapi-ip-geolocation`⁵. Код приложения для демонстрации приведён в Листинге 4.7. Функция `get_user_session_vulner` содержит уязвимость «межсайтовый скриптинг». В функции `get_user_session_safe` происходит экранирование ответа от удалённого сервера. Стоит отметить, что удалённый сервер возвращает единственную строку без HTML-разметки, по этому, экранирование функцией `escape` тут возможно.

```

1 ip_client = IpGeolocationClient(base_url="https://
  ipgeolocation.abstractapi.com/", token="api-token")
2 def get_ip_info(ip: str) -> str:
3     return ip_client.get_info(ip)
```



⁵<https://github.com/vldF/Master-thesis-DSL/tree/main/Examples/abstractapi-ip-geolocation/library>

```

4
5 @app.route('/vulnerable/ip_info/<int:user_ip>',
  methods=['GET'])
6 def get_user_session_vulner(user_ip):
7     ip_info = get_ip_info(user_ip)
8     escaped_ip = escape(user_ip)
9     # уязвимость: межсайтовый скриптинг
10    return f"""
11    <html>
12    <b>IP: {escaped_ip}</b>
13    </br>
14    <b>IP geolocation: {ip_info}</b>
15    </html>
16    """
17
18 @app.route('/safe/ip_info/<int:user_ip>', methods=['GET'])
19 def get_user_session_safe(user_ip):
20     ip_info = get_ip_info(user_ip)
21     escaped_ip_info = escape(ip_info)
22     escaped_ip = escape(user_ip)
23     # уязвимость отсутствует
24     return f"""
25     <html>
26     <b>IP: {escaped_ip}</b>
27     </br>
28     <b>IP geolocation: {escaped_ip_info}</b>
29     </html>
30     """

```

Листинг 4.7 — Код приложения, использующий библиотеку

Для добавления поддержки разработанной библиотеки в PT JSA был написан код, приведённый в Листинге 4.8. Он содержит единственный объект *IpGeolocationClient*, содержащий функции `__init__` метод (выполняющий роль конструктора) и *get_ip_info*. Последняя принимает IP в виде строки и возвращает строку с taint-меткой и происхождением типа *Body*. Они соответствуют ответу от HTTP сервиса, который используется в качестве поставщика информации. Функция *get_ip_info* в данном примере возвращает недостоверные данные больше для демонстрационных целей. В прочем, некоторые требования к обеспечению безопасности приложений

действительно могут быть установлены так, что такое поведение будет само собой разумеющимся. В самом деле, данные данные получаются со внешнего сервера, находящегося вне зоны контроля. Атакующий может получить к нему доступ и отправить из него недостоверные данные для эксплуатации уязвимости, по этому, данные нужно валидировать.

```

1 import "Standard";
2
3 object IpGeolocationClient {
4     func __init__(base_url: string, token: string) {}
5
6     func get_ip_info(ip: string): string {
7         return CreateTaintedDataOfType<string>("Body");
8     }
9 }

```

DSL

Листинг 4.8 — Код расширения для поддержки библиотеки

В Таблице 8 приведена информация об уязвимостях, обнаруженных в коде 4.7 при помощи разработанного расширения базы знаний анализатора. Видно, что анализатор нашёл единственную уязвимость, описанную выше, а также что представленные свойства этой уязвимости корректны.

Таблица 8 — Информация об уязвимостях

Параметр	Значение
Функция <i>get_user_session_vulner</i>	
Уязвимость «межсайтовый скриптинг»	
Уязвимое выражение	IP info: {ip_info}
Точка входа	@app.route('/vulnerable/ip_info/<int:user_ip>'...
Трасса данных	@app.route('/vulnerable/ip_info/<int:user_ip>'... ip_info = get_ip_info(user_ip)... IP info: {ip_info}... return f''''

Таким образом, разработанный DSL позволяет описывать простые пользовательские библиотеки, которые не могут быть внесены в базу знаний анализатора при его разработке, что делает инструмент актуальным для большого количества пользователей.

4.6 Генерируемая автоматически пользовательская библиотека

Такие инструменты как `protobuf` [16] и `openAPI` позволяют генерировать клиенты и сервера на основании спецификации. Это часто

используется, к примеру, в подходе разработки на основе контракта (contract-first development). Она популярна при разработке web-приложений с микросервисной архитектурой, так как позволяет разрабатывать каждый из сервисов параллельно. В микросервисной архитектуре каждый из компонентов системы выполняет различные функции. Можно представить, что в некотором приложении база данных о пользователях реализована в виде выделенного сервиса. У него есть заранее известный контракт на языке `protobuf`, который позволяет автоматически сгенерировать код клиента для gRPC [17]. Код клиента, сгенерированный gRPC затруднителен для автоматического анализа, так как, в частности, содержит сложную сериализацию в памяти с последующей отправкой сообщения серверу. Ответы проходят другую сложную процедуру — десериализацию. По этому, есть необходимость в добавлении поддержки этого клиента в базу знаний анализатора.

Код контрактов сервиса базы данных о пользователе приведён в Листинге 4.9. Он включает в себя сервис *UserStore*, а также два сообщения — запрос данных о пользователе *UserRequest* и ответ на него *User*.

```
1  syntax = "proto3"; protobuf
2  package userstorage;
3
4  message User {
5      string id = 1;
6      string username = 2;
7  }
8
9  message UserRequest {
10     string id = 1;
11 }
12
13 service UserStore {
14     rpc GetUser(UserRequest) returns (User);
15 }
```

Листинг 4.9 — Контракты базы данных о пользователе

В Листинге 4.10 приведён код приложения, использующего этот сервис. Обработчике запросов *vulnerable_get_user_info* содержится уязвимость «небезопасная прямая ссылка на объект», позволяющая атакующему получить перебором все значения из БД, если он может явно передать идентификатор каждого из них. К примеру, он может получить список всех

идентификаторов пользователей другим запросом, а затем для каждого из них получить всю возможную информацию через *vulnerable_get_user_info*. Для исправления уязвимости стоит запретить явную передачу идентификатора через аргументы запроса и получать его, к примеру, из токена текущего пользователя. Этот подход демонстрируется в *safe_get_user_info*. Функция *_get_user_id* получает ИД текущего пользователя на основании токена из запроса, её реализация опущена.

Обе функции используют канал (*channel*) для коммуникации с сервисом. Эту абстракцию предоставляет gRPC в своей библиотеке. Далее, они получают экземпляр «заглушки» (*stub*) для сервиса UserStore, который содержит функцию *GetUser*. Она принимает объект запроса, отправляет его на сервер, получает ответ и возвращает результат, который, затем, передаётся пользователю.

```
1  @app.route("/unsafe/getCurrentUserInfo/  
   <string:user_id>")  
2  def vulnerable_get_user_info(user_id: str):  
3      with grpc.insecure_channel(SERVER_URL) as channel:  
4          user_storage_stub =  
            user_storage_service.UserStoreStub(channel)  
5          getUserRequest = user_storage_dto.UserRequest(user_id)  
6  
7          # уязвимость: небезопасная прямая ссылка на объект  
8          user = user_storage_stub.GetUser(getUserRequest)  
9  
10         return jsonify(user)  
11  
12  @app.route("/safe/getUserInfo/")  
13  def safe_get_user_info():  
14      current_user_id = _get_user_id()  
15      with grpc.insecure_channel(SERVER_URL) as channel:  
16          user_storage_stub =  
            user_storage_service.UserStoreStub(channel)  
17          # нет уязвимости  
18          user =  
            user_storage_stub.GetUser(user_storage_dto.UserRequest(curr  
19  
20         return jsonify(user)
```

Листинг 4.10 — Код приложения, использующий сервис UserStore

Инструмент для генерации кода gRPC клиента для Python генерирует отдельные файлы для моделей сообщений и кода сервисов. Каждый из них располагается в собственном пакете. По этому, это необходимо повторить и в коде расширения, см. Листинг 4.11. Объект *UserRequest* содержит поле *id* и конструктор для его инициализации. Объект *User* не содержит конструктора, так как он не нужен для моделирования кода. Объект *UserStoreStub* содержит конструктор, принимающий аргумент *channel* так как он используется для инициализации из кода приложения. Функция *GetUser* запускает определение уязвимости типа «небезопасная прямая ссылка на объект» (Insecure Direct Object References).

```

1 // файл userstorage.services_pb2.jsadsl DSL
2 package "userstorage.services_pb2";
3 object UserRequest {
4     var id: string;
5     func __init__(id: string) {
6         self.id = id;
7     }
8 }
9 object User {
10     var id: int;
11     var username: string;
12 }
13
14 // файл userstorage.services_pb2_grpc.jsadsl
15 import "Standard";
16 import "userstorage.services_pb2";
17
18 object UserStoreStub {
19     func __init__(channel: any) { }
20
21     func GetUser(request: UserRequest): User {
22         Detect(
23             request.id,
24             "Insecure Direct Object References",
25             "Arbitrary string data");
26         return CreateDataOfType<User>();
27     }
28 }

```

Листинг 4.11 — Расширение для поддержки UserStore

В Таблице 9 приведён результат анализа кода из Листинга 4.10 с разработанным расширением.

Таблица 9 — Информация об уязвимости

Параметр	Значение
Функция <i>vulnerable_get_user_info</i>	
Уязвимость «небезопасная прямая ссылка на объект»	
Уязвимое выражение	<code>user = user_storage_stub.GetUser(getUserRequest)</code>
Точка входа	<code>@app.route("/vulnerable/getCurrentUserInfo/...</code>

Параметр	Значение
Трасса данных	@app.route("/vulnerable/getCurrentUserInfo/ getUserRequest = user_storage_dto.UserRequest(...) user = user_storage_stub.GetUser(getUserRequest)

По Таблице 9 видно, что описанная выше уязвимость была обнаружена и её свойства корректны. Это показывает, что разработанный DSL может быть использован для добавления поддержки автоматически генерируемых библиотек, а также показывает, как пользователь анализатора может учесть архитектуру конкретного проекта для уточнения результатов анализа.

4.7 Сравнение с существующим способом расширения базы знаний

Важным аспектом для апробации является проверка того, насколько разработанный DSL эффективнее существующего способа расширения базы знаний PT JSA на языке C# Script. Одним из показателей эффективности является объём исходного кода. Для такого сравнения для нескольких библиотеки, представленных в этой Главе были дополнительно реализованы расширения в старом формате. Они эквивалентны рассмотренным ранее. Это показывается абсолютно одинаковыми результатами срабатываний анализатора на тестовых проектах, а также обеспечивается по построению. Исходные коды расширений на C# Script приведены в директории Examples⁶.

Результаты сравнения расширений на DSL и на C# Script приведены в Таблице 10. Для расчётов игнорировались все пустые строки, а также пробельные символы и комментарии.

Таблица 10 — Сравнение объёмов кода расширений на DSL и C# Script

Библиотека	DSL		C# Script	
	Строки	Символы	Строки	Символы
Urllib3	58	1848	328	9936
Psycopg2	33	790	176	5323
Flask	DSL: 71 C# Script: 111	DSL: 2793 C# Script: 3993	487	16642

Таким образом, объём кода расширений на разработанном DSL меньше в 5-6 раз по сравнению с аналогичным на C# Script, что указывает на выразительные способности языка.

⁶<https://github.com/vldF/Master-thesis-DSL/tree/main/Examples/>

4.8 Резюме

В данной главе демонстрируются возможности разработанного DSL на примере пяти проектов. Каждый из них демонстрирует различные аспекты языка: возможность описания источников, стоков и фильтрующих функций, возможность описания поведения кода, сохранение информации в поля объектов, совместное написание кода на DSL и C# Script. Выразительные способности языка подкрепляются численным сравнением объёма кода расширений на новом и существующем механизмах.

Язык предоставляет возможность описания функций и фреймворков с различной степенью детализации поведения и потоков данных. Он позволяет разрабатывать расширения для PT JSA, как было показано в различных проектах. DSL предоставляет абстракции от внутренних особенностей анализатора. Несмотря на то что от пользователя всё ещё требуются компетенции аналитика безопасной разработки приложений, разрабатывать расширения стало существенно проще. Таким образом, требования, поставленные в рамках Работы, были выполнены.

ЗАКЛЮЧЕНИЕ

В ходе Работы был разработан прототип DSL для расширения базы знаний taint-анализатора PT JSA. Для него подробно описан синтаксис и семантика. Предоставлены механизмы переиспользования кода и создания пакетов библиотек. Для разработанного языка реализован транслятор, переводящий его в низкоуровневый код на C# Script, который уже поддерживается анализатором. Таким образом обеспечивается не только совместимость с PT JSA, но и возможность реализации части кода расширения с использованием полного набора API анализатора.

Полученное решение было апробировано на нескольких проектах, за основу которых взяты подходы, используемые в современной корпоративной разработке web-приложений. Для демонстрации возможностей DSL и его транслятора, были реализованы расширения базы знаний JSA для нескольких библиотек, использующих разные подходы и применяемые с разными целями: базы данных, HTTP-сервера, отправка и получение сетевых запросов. Также демонстрируется возможность разработки описаний для компонентов, которые были реализованы пользователем или автоматически им сгенерированы. Это особенно важно, так как, в отличие от популярных библиотек и фреймворков, они не могут быть поддержаны в базе знаний анализатора его производителем. По результатам апробации можно сделать вывод, что разработанные технологии пригодны к использованию.

В ходе апробации прототипов были выявлены некоторые недостатки. Одним из них является отсутствие функциональных значений и типов, что делает затруднительным, к примеру, поддержку мета-программирования с помощью декораторов в языке Python. Также, это не позволяет описывать функции обратного вызова (*callbacks*). В качестве преодоления этого ограничения, предлагается реализовывать необходимую часть расширения на языке C# Script, что даст доступ к низкоуровневому API анализатора.

Другим недостатком является отсутствие алгебраических типов данных, таких как типы пересечения и типы объединения. Они могли бы помочь точнее типизировать возвращаемые значения функций. Так как в Python типизация не является сильной, авторы некоторых библиотек пользуются этим и возвращают значения, тип которых продиктован внутренним состоянием объекта или значениями аргумента функции.

Два недостатка, приведённых выше, должны быть исправлены в будущем. Также, необходимо увеличить число семантических проверок. Для реализации конечного продукта по этому прототипу, необходимо включить поддержку всех других языков, поддерживаемых PT JSA: Ruby,

C#, Go, Java и других. Также, необходимо разработать инструменты программиста, такие как отладчик, LSP сервер, расширения для IDE. Одним из направлений развития могут стать инструменты для автоматической или автоматизированной генерации кода расширений PT JSA. К примеру, их можно получать на основании контрактов protobuf или openAPI.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Khedker U., Sanyal A., Sathe B. Data flow analysis: theory and practice. CRC Press, 2017.
2. Krishnamoorthy S., Hsiao M.S., Lingappan L. Tackling the Path Explosion Problem in Symbolic Execution-Driven Test Generation for Programs // 2010 19th IEEE Asian Test Symposium. 2010. сс. 59–64.
3. Разработчики CodeQL. Документация CodeQL, описание возможностей [электронный ресурс]. 2025. URL: <https://codeql.github.com/docs/> (дата обращения: 05.04.2025).
4. Разработчики Semgrep. Документация Semgrep, описание возможностей [электронный ресурс]. 2025. URL: <https://semgrep.dev/docs/> (дата обращения: 05.04.2025).
5. Разработчики SonarQube. Документация SonarQube, описание возможностей [электронный ресурс]. 2025. URL: <https://docs.sonarsource.com/sonarqube-server/> (дата обращения: 05.04.2025).
6. Разработчики SonarQube. Документация SonarQube, Adding coding rules [электронный ресурс]. 2025. URL: <https://docs.sonarsource.com/sonarqube-server/latest/extension-guide/adding-coding-rules/> (дата обращения: 05.04.2025).
7. Arzt S. Static Data Flow Analysis for Android Applications. Darmstadt, 2017.
8. Mark Michaelis. Essential .NET — C# Scripting [электронный ресурс]. 2016. URL: <https://learn.microsoft.com/en-us/archive/msdn-magazine/2016/january/essential-net-csharp-scripting> (дата обращения: 05.04.2025).
9. Pierce B.C. Types and programming languages. MIT press, 2002.
10. Kefallonitis F. Name mangling demystified // int 0x80, White Paper. 2007.
11. Parr T. The definitive ANTLR 4 reference. The Pragmatic Bookshelf, 2013.
12. Guan Z. и др. Semantics Lifting for Syntactic Sugar // Proceedings of the ACM on Programming Languages. ACM New York, NY, USA, 2024. т. 8, № OOPSLA2. сс. 1336–1361.
13. Zeller W., Felten E.W. Cross-site request forgeries: Exploitation and prevention // The New York Times. 2008. сс. 1–13.
14. Kindy D.A., Pathan A.-S.K. A survey on SQL injection: Vulnerabilities, attacks, and prevention techniques // 2011 IEEE 15th international symposium on consumer electronics (ISCE). 2011. сс. 468–471.

15. Rafique S. и др. Systematic review of web application security vulnerabilities detection methods // Journal of Computer and Communications. Scientific Research Publishing, 2015. т. 3, № 9. сс. 28–40.
16. Разработчики protobuf. Официальный сайт protobuf, описание возможностей [электронный ресурс]. 2025. URL: <https://protobuf.dev/> (дата обращения: 05.04.2025).
17. Разработчики gRPC. Официальный сайт gRPC, описание возможностей [электронный ресурс]. 2025. URL: <https://grpc.io/> (дата обращения: 05.04.2025).