FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Aspect-Oriented Programming for Javascript using the Lara Language

Ricardo Sá Loureiro Ferreira da Silva

Mestrado Integrado em Engenharia Informática e Computação

Supervisors: *Dr. João Bispo* and *Dr. Tiago Carvalho*

July 23, 2019

Aspect-Oriented Programming for Javascript using the Lara Language

Ricardo Sá Loureiro Ferreira da Silva

Mestrado Integrado em Engenharia Informática e Computação

July 23, 2019

Abstract

Aspect-Oriented Programming (AOP) is a programming paradigm focused on improving modularity through the separation of concerns. The LARA framework is a set of Java libraries that can be used to easily build AOP tools for arbitrary programming languages. The main goal of this dissertation project was to investigate if the LARA approach, which is based on static source-to-source compilation, was adequate for developing tools for a dynamic language such as JavaScript.

In order to test this we developed Jackdaw - an AOP tool for JavaScript which is built upon the LARA framework. The objective was for Jackdaw to present a sufficient number of features in order to satisfy its use cases, namely the implementation of an obfuscation module and related features. The research component of this work analyses state-of-the-art AOP tools, libraries and frameworks for the JavaScript programming language, and presents a methodology for comparison, in order to classify the strengths and weaknesses of each of these tools. Additionally, we compare Jackdaw against the existing tools. Through use of the LARA framework it was possible to develop Jackdaw - a new AOP tool for JavaScript which was capable of satisfying the mentioned use cases, while holding up against other existing tools.

Resumo

Programação orientada a aspetos (AOP) é um paradigma de programação focado em melhorar a modularidade através da separação de "concerns". A LARA framework é um conjunto de bibliotecas de Java que podem ser utilizadas para facilmente construir ferramentas AOP para linguagens de programação arbitrárias. O objetivo principal deste projeto de dissertação foi investigar se a abordagem LARA, que é baseada em compilação estática source-to-source, se adequa a uma linguagem dinâmica como Javascript.

Para testar esta hipótese, desenvolvemos o Jackdaw - uma ferramenta AOP para Javascript construída em cima da framework LARA. O objectivo da ferrementa é apresentar um numero suficiente de capacidades de forma a poder validar os casos de uso selecionados, nomeadamente a implementação de um modulo de ofuscação e "features" relacionadas. A componente de investigação deste trabalho analisa as mais recentes ferramentas e frameworks de AOP para a linguagem de programação Javascript. Esta análise, onde incluímos o Jackdaw, usa uma metodologia para a comparação de ferramentas, de forma a poder classificar os pontos fortes e fracos de cada uma delas.

Através do uso da LARA framework foi possível desenvolver o Jackdaw - uma nova ferramenta AOP para Javascript que foi capaz de satisfazer os casos de uso mencionados e ser competitiva em relação às ferramentas existentes.

Contents

List of Figures

LIST OF FIGURES

List of Tables

LIST OF TABLES

Abbreviations

AOP	Aspect Oriented Programming
LARAI	LARA Interpreter
AST	Abstract Syntax Tree
CFF	Control Flow Flattening
DSL	Domain specific language

Chapter 1

Introduction

This chapter introduces the concept and technological context of this dissertation project. It also states the main objectives of the thesis, the motivation for this project and the validation strategies that are going to be used. Throughout this work we will reference certain words and terminologies that will be briefly explained here.

An **aspect** is code that specifies a concern separately from a program that contains the business logic. In the context of the LARA framework, aspects are also referred as **strategies**.

A **weaver** is piece of software that applies aspects written in an AOP language to a given program. In our case, the weaver will be the LARA-based source-to-source compiler for JavaScript.

The **weaving process** is the application of an aspect to a given program[KLM+97]. In our case it corresponds to the execution of the JavaScript source-to-source compiler when an aspect/strategy is applied to a given program.

1.1 Context

Aspect-oriented programming (AOP) is a programming paradigm focused on improving modularity through separation of concerns [KLM+97]. To achieve this, AOP proposes that certain concerns should be specified in modules (usually called *aspects*) separately from the source code where they would usually appear.

The LARA framework is a framework which uses LARA, a Domain Specific Language (DSL) which is agnostic to the target language of the weaver. This means that a developer can develop AOP aspects for several different target languages using with the same aspect language, and sometimes even reuse the same aspects between languages. Additionally, the LARA Framework contains tools (e.g., a Weaver Generator) which reduces the required effort to add support for new programming languages.

This dissertation investigates if it is possible to use the LARA framework to develop an AOP tool for the JavaScript programming language. This dissertation was done in the laboratory of Computational Systems of the Faculty of Engineering of the University of Porto (FEUP), in the Special-Purpose Computing Systems, languages and tools research group (SPeCS). The LARA

framework, upon which the engineering prototype of the dissertation is built, originated and is currently maintained by this research group.

1.2 Objectives and motivation

JavaScript is currently the *de facto* browser programming language, and one the of the most popular programming languages in the world [1]. Given this, we consider that there is a valid interest in making the advantages and features of AOP available to JavaScript developers. These features include modularity through separation of concerns, automatic logging, automatic code transformations, among others - which can lead to an overall increase in productivity when developing a software system.

One way to add support for AOP to a language is by using a *source-to-source compiler*. Typically, a compiler is a piece of software which accepts as input a program written in a given programming language, and outputs an equivalent program in another programming language, usually machine code. A source-to-source compiler outputs a program written in the same language as the input program, and can be useful to apply custom code transformations. LARA is an aspect-oriented language, developed in Faculdade de Engenharia da Universidade do Porto (FEUP), which allows the development of aspects that can be applied to different programming languages [CCC+12].

The main objective of this dissertation project was to investigate if the LARA approach, which is based on static source-to-source compilation, is adequate for developing an AOP tool for a dynamic language such as JavaScript.

In order to test this idea we developed an Aspect-Oriented weaver for Javascript, called *Jackdaw*, that is based on the LARA framework. We performed a revision of the current state-of-the-art of JavaScript weavers by doing a qualitative comparison of the already existing tools, and tried to understand what could be done to improve upon them. In order to validate Jackdaw, we tested it against a set of existing LARA strategies that are currently supported by other LARA-based weavers that target languages such as MATLAB [BPN+13] or Java [CC18]. Additionally, we developed strategies for an obfuscation use case which required non-trivial features such as variable renaming, which were tested with several benchmarks from the known Jetstream 2 [jet] collection.

1.3 Contributions

We consider that this thesis has the following contributions:

- Review of state-of-the-art tools for applying aspect oriented programming to JavaScript and application of a previous analysis framework to more recent tools.
- Confirmation that the LARA framework can be used to build tools for dynamic languages such as Javascript that implement non-trivial source-to-source transformations.

[1] https://github.blog/2018-11-15-state-of-the-octoverse-top-programming-languages/

- Development and release of Jackdaw, an open-source LARA-based AOP tool for Javascript that uses a static source-to-source approach[2].
- Research and implementation of several obfuscation-related algorithms and their inclusion in Jackdaw.
- Website where users can try an online demo of Jackdaw[3].

1.4 Summary

On this chapter we introduced the concept and technological context of this dissertation project, which is based on AOP and source-to-source compilation. We also stated the main objectives of the thesis - to investigate if the LARA approach is adequate for developing an AOP tool for a dynamic language such as JavaScript, and use it to attempt to improve the current state-of-the-art. We presented the motivation for this project, and the validation strategies that are going to be used.

[2] https://github.com/tansvanio/jsweaver
[3] http://specs.fe.up.pt/tools/jackdaw/

Chapter 2

Existing Work

This chapter describes the basic concepts of aspect-oriented programming and introduces the LARA Framework. It also lists the most relevant currently existing AOP tools and frameworks for Javascript along with their correspondent analysis and comparison. This was particularly useful in order to determine where certain tools succeed and where others fail, in order to tailor Jackdaw to achieve the highest degree of success in all the components which make an AOP tool.

2.1 Aspect-Oriented Programming

Aspect-oriented programming is a programming paradigm that is based on the idea we can specify certain concerns (properties or areas of interest) of a system separately from the business logic, and then relying on mechanisms in the underlying AOP environment to weave or compose them together into a coherent program [EFB01]. One of the main advantages of AOP is that it allows us to achieve modularity through the separation of concerns. In order to separate concerns, AOP introduces to us the concept of *Aspects*, which are "mechanisms beyond subroutines and inheritance for localizing the expression of a crosscutting concern" [EFB01]. After being defined, a particular *Aspect* will then contain several joint points, which are the instructions in which the aspect code will interact with the entire environment, allowing us to perform tasks like: manipulating code, function calls, logging, unit testing, etc.

2.1.1 Possible approaches in AOP

Regarding the implementation of aspect-oriented programming to an already existing target programming language, we can consider the utilization of two different approaches. The first approach is for the aspect-code to be an extension of the source-code, which allows to move concerns which were previously dealt with inside the business logic into an aspect, effectively moving source code to aspect code [FDNT15]. This approach values a seamless integration between aspect and business logic, and provides access and support of target language features in aspect code [PCB+18]. Examples of this approach is AspectJ [KHH+01] and AspectC++ [SGSP02], which

extend Java and C++ with Aspect-Oriented Programming concepts, respectively. One of the possible advantages of this approach is a possible smaller learning curve, since the user is not required to learn a new language syntax to program the aspect code. The more considerable drawbacks of this approach will be covered later in this chapter when we present the concept ***Invasiveness***.

The second approach is designing an aspect language which is agnostic to the target language. This potentially requires more complex engineering in order to integrate the two different languages for the aspect and business logic, although AOP tools that use the first approach can also be noteworthy in their complexity [Lad09]. Tools that use this second approach will physically separate the aspect code from the target language code. Later in this chapter we will discuss how the LARA framework takes great advantages in using this approach in order to be able to support different target languages [PCB+18].

2.1.2 Domain specific languages

Following the definition proposed by [vDKV00], a Domain-Specific Language (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain.

An advantage of DSLs is that they have the potential to express a solution to the domain problem more concisely and in a clearer way. This can also diminish errors or inefficient idioms, and make code generation more efficient. On the downside, DSLs introduce the overhead of having to learn a new language, and usually are not as integrated in the compilation tool-flow as native libraries or compiler-supported extensions.

DSLs for source-to-source compilers can be generic or specific. LARA [CCC+16] is an example of a DSL which can be used to implement generic source code analysis and transformations. On the other hand, CHiLL [CCH08] is a declarative language that only allows to specify sequences of (predefined) transformations and constrains that are to be applied to existing source code.

On the next section we will introduce the LARA framework and describe how the LARA DSL and its properties make up the foundation of Jackdaw.

2.2 LARA Framework

The LARA language [CCC+12] is a DSL for source-to-source transformations and analysis, inspired by AOP concepts [PCB+18] which was developed at the Faculty of Engineering of the University of Porto and allows for the development of aspects that can be applied to different programming languages. Unlike most AOP approaches, LARA was designed to work independently of the target language and is capable of specifying code insertions and transformations for any target programming language. This was achieved by separating the LARA language from the language specification of the target language, which consists of a specification of relevant points of interest, along with their attributes and actions [PCB+18].

On Figure 2.1 we present a simple example of LARA code. This code declares a LARA aspect which selects all the function calls inside functions and proceeds to insert a new comment in the line prior to each call.

```
aspectdef LaraAspect
    select function.call end
    apply
        $call.insert before "// Call" + $call.name + " at line: " + $call.line;
    end
end
```

Figure 2.1: LARA code example.

The LARA Framework has been previously used to develop a weaver for MATLAB [BPN+13], which is a dynamic language like JavaScript. However, that work mainly focused on the compilation of MATLAB code to C/OpenCL [BRC14, RBC17], and did not fully explore the source-to-source capabilities of LARA over a dynamic language, which is one of the objectives of this thesis.

2.2.1 LARAI

The LARA interpreter (LARAI) was developed in order to improve the integration of the LARA language with different weaving environments. LARAI executes the LARA aspects, while the Weaver makes the connection between the queries and actions in LARA, and the target source-code [PCB+18].

2.2.2 Weaver Generator

Developing and maintaining a Weaver Engine from the ground up requires a high amount of effort. However, one of the main advantages of extending language support with the LARA framework is that it ships with a Weaver generator. The Weaver generator is a piece of software that, provided with a language specification, generates a skeleton Weaver engine. With the skeleton Weaver, the developer can focus on developing the specific actions and attributes that will be exposed by the weaver, while the generator takes care of all the required infrastructure code to interface with LARAI [PCB+18]. Using the Weaver for this work allows for a much faster progress in the building of the new tool and more importantly, it allows for quick prototyping and an incremental build up of strategies.

2.3 AOP frameworks for Javascript

Currently there are several Aspect-oriented plugins and frameworks for the Javascript language. Some of these plugins are experimental proof-of-concepts while others are fully built products that

are quite usable in a production environment. In this work, the scope of analysis and comparison to other frameworks will be limited to the ones we considered to be the most relevant.

2.3.1 AspectScript

AspectScript [TLT10] has been presented as an AOP extension for Javascript that incorporates high-level programming and support for the dynamic aspects of JavaScript as part of its native design. One of the main motives for the creation and development of AspectScript was that previous existing work did not contain any AOP extension for JavaScript which fully supported some of the features that define JavaScript - dynamic prototype-based programming and higher-order functions. AspectScript implements point-cuts and advices as normal JavaScript functions which allows the use of normal high-order programming to define AOP aspects. Additionally, AspectScript implements a series of state-of-the-art AOP features like dynamic aspect deployment with scoping strategies, and user-defined quantified events. On Figure 2.2 we can see an example of the AspectScript syntax. This particular piece of code is reported to do the following:

1. Computes current aspects to determine what set of aspects may apply.
2. Evaluates the point-cuts of these aspects against the current join point.
3. Chains together the advices of aspects that matched the current join point and then applies them.

It was not possible to experiment with AspectScript due to the hosting link being down on the website of the PLEIAD laboratory of the Computer Science Department (DCC) of the University of Chile (Faculty of Engineering).

```
function weave(jp){
    var currentAspects = union(globalAspects,aspectsIn(ctxObj),aspectsIn(ctxFun));
    var advices = match(jp,currentAspects);
    return chainAndApply(advices);
}
```

Figure 2.2: AspectScript code example [HHL15]

2.3.2 AOJS

AOJS [OKM+11] has been presented as an JavaScript framework which can provide AOP support with the full separation of aspects and normal JavaScript modules. The development of AOJS has been motivated by the fact that previous existing AOP frameworks did not achieve the full separation of aspects from normal code, requiring instead modifications in the target code in order to achieve aspect weaving. AOJS allows for the specification of function execution, variable

substitution and file initialization as the join-points of its aspects. Like mentioned before, AOJS guarantees the complete separation of aspects and normal code by adapting its architecture for aspect weaving. On Figure 2.3 we can see an example of the AOJS syntax. For this example, it is reported that AOJS weaves aspects into the location specified by the initialization point cuts, and uses a code template for code replacement. The weaving process proceeds in the following manner:

1. Executes <before>.
2. Executes <target>, and stores the return value of the <target> expression into temporal variable retvalue.
3. Executes <after>.
4. Returns the value stored in retvalue.

```
(function(){
    <before>
        var _returnvalue = <target>;
    <after>
        return _returnvalue;
})();
```

Figure 2.3: AOJS code example [HHL15]

2.3.3 AspectJS

AspectJS [asp] is a programming library that allows the user to implement method-call interception via proxy functions in Javascript. Additionally, this library provides support for method calling and system state validation (locally or remotely) and remote method-call instrumentation. AspectJS was designed and developed by Richard Vaughan, and is distributed by Dodeca Technologies Ltd. The example on Figure 2.4 we can see an example of the AspectJS syntax. The code is reported to execute as follows: the AspectJS function "addPrefix" is used to add the function "prefixFunction" to run before "myFunc()". When myFunc is called, prefixFunction will be executed first. Code experimentation with AspectJS has not been possible at the current date due to it being a paid product.

2.3.4 aspectjs

aspectjs [For17] (in lowercase) is a simple proxy-based-AOP library developed by Philip Ford which was implemented as a Node.js package. It works either with standalone functions or with object methods. Additionally, it can also be used on client-side JavaScript with Browserify. One

```
function prefixFunction(){
    // Code
}
function myFunc(){
    // Code
}
AJS.addPrefix(this,"myFunc",prefixFunc);
myFunc();
```

Figure 2.4: AspectJS code example [HHL15]

of the main advantages of this library is that proxy functions are extremely easy to use and understand, the downside being that it requires the user to mix AOP code with source code. On Figure 2.5 we present a code example of this package. This code declares an object with several properties and a method, following by a use of the aspectjs library to add a new method to this object prior to the original method.

```
const before = require('aspectjs').before;
let addAdvice = require("aspectjs").addAdvice;

let advised, adviser, result;
advised = {
   add: function(increment){this.left += increment; },
   id: 'test',
   left: 32,
   top: 43
};
adviser = {
   override: function(increment){ advised.left = increment; }
};

before(advised, "add").add(adviser, "override");
advised.add(2);
console.log(advised)
// Prints: { add: [Function: f], id: 'test', left: 4, top: 43 }
```

Figure 2.5: aspectjs code example for Node.js.

2.3.5 aspect.js

aspect.js [asp15] (not to be mistaken with aspectjs or AspectJS) is a library for aspect-oriented programming with JavaScript which takes advantage of ECMAScript 2016 decorators syntax. However, it requires AOP logic to be mixed with business logic in the Javascript code, much like our previous example. aspect.js was first presented in AngularConnect, the official European Angular conference of 2015.

On Figure 2.6 we present a code example of aspect.js working with typescript. On this code we declare two classes and a AOP logger aspect. This aspect sets up a logging method before the getting or setting of articles in the ArticleCollection class.

```
import {beforeMethod, Wove, Metadata} from 'aspect-dot-js';

class LoggerAspect {
  @beforeMethod({
    classNamePattern: /^Article/,
    methodNamePattern: /^(get|set)/
  })
  invokeBeforeMethod(meta: Metadata) {
    // meta.woveMetadata == { bar: 42 }
    console.log(`Inside of the logger. Called ${meta.className}.${meta.method.name} with args
        : ${meta.method.args.join(', ')}.`);
  }
}

class Article {
  id: number;
  title: string;
  content: string;
}

@Wove({ bar: 42 })
class ArticleCollection {
  articles: Article[] = [];
  getArticle(id: number) {
    console.log(`Getting article with id: ${id}.`);
    return this.articles.filter(a => {
      return a.id === id;
    }).pop();
  }
  setArticle(article: Article) {
    console.log(`Setting article with id: ${article.id}.`);
    this.articles.push(article);
  }
}

new ArticleCollection().getArticle(1);

// Result:
// Inside of the logger. Called ArticleCollection.getArticle with args: 1.
// Getting article with id: 1.
```

Figure 2.6: aspect.js code example for typescript

2.4 Comparison and Evaluation criteria

This section relies on the comparison model proposed by [HHL15]. Following this model, we compared and evaluated the previously mentioned frameworks according to three criteria: Invasiveness, Briefness and Maturity. The main objective of this comparison is to understand why some frameworks fail in some of these criteria and why others succeed. Jackdaw aims to to use

the advantages offered by the LARA framework to gain a competitive advantage against other tools in these criteria.

2.4.1 Invasiveness

While there are many frameworks that manage to successfully implement AOP by mixing it with source code, it should be noted that AOP was designed to be a supplement and extension to conventional programming. The original program should not be modified due to the presence of AOP in the development stack. Additionally, if we take into account code maintenance and update, joining aspect programming with the rest of the system will cause secondary development [HHL15]. Thus, an AOP tool should strive to make aspect code separate from the source code.

2.4.2 Briefness

Usually, Aspect-oriented programming is harder to grasp for new developers as opposed to more popular paradigms, like Object-Oriented programming. Thus, an AOP tool should allow the developer to define aspects with a clear and simple syntax as much as it can.

2.4.3 Maturity

An AOP tool for Javascript should be able to control all the unique features of Javascript and particularly the dynamic aspect of Javascript, as well as providing comprehensive generic AOP functionality.

2.4.4 Comparison

Using the previously defined grading components it is possible to perform a quick evaluation and comparison between the presented tools (see Table 2.1).

Regarding Invasiveness, AOJS is the only tool that manages to achieve full separation of AOP code from source Javascript code, and thus is the only tool to have a positive score in this component.

Regarding briefness, there are two tools that clearly standout in this components - aspectjs and AspectJS. This is due to their proxy-function based operations which allow for very simple AOP directives. Use of ECMAScript 2016 by aspect.js has been given a negative review, since it proposes a somewhat messy syntax which is easy to confuse with the normal JavaScript code. AOJS relies on XML to achieve a somewhat brief syntax and AspectScript achieves the same with its method-based AOP operations.

Regarding maturity, we considered AspectScript and aspect.js to be the most mature tools, since they cover the basic fundamental features of AOP. The proxy-based nature of AspectJS and aspectjs end up limiting their range of features. AOJS is unable to take advantage of high-order programming due to the fact that it uses XML for its syntax [HHL15].

Table 2.1: Table showing comparison between tools.

Component / Tool	AspectScript	AOJS	AspectJS	aspect.js	aspectjs
Invasiveness	-	+	-	-	-
Briefness	+	+	+	-	++
Maturity	+	-	-	+	-

Making a preliminary evaluation of the LARA-based JavaScript Weaver, we consider that in regard to Invasiveness, we expect the tool to do very well, due to the fact that LARA has its own syntax that is kept separate from the source code. This is also one of the characteristics that allows it to be multi-target framework. Regarding Briefness, we consider that LARA has a syntax that, while not minimalistic, is quite easy to comprehend. Additionally, work on previous LARA-weavers has shown that is it possible to write complex LARA aspects that provide accessible interfaces[ABBC18]. Finally, the LARA framework provides several out-of-the-box tools that significantly helps a new weaver to have access to features associated with more mature tools, such as unit testing and documentation generators. The framework also allows to build the weaver in an incremental way, which makes it relatively easy to add new features as needed.

2.5 Summary

On this chapter we described the main technological background for the new AOP tool for JavaScript. Additionally, it presented an introduction to Aspect-oriented programming, the LARA framework, and a set of relevant AOP tools. It introduced the comparison criteria which will be used to measure the developed tool against the state-of-the-art, and a comparison of the state-of-the-art tools, based on investigation and experimentation.

Chapter 3

Work Plan

This chapter presents the work plan used for this dissertation project, as well as how its different development phases were distributed throughout the allowed timetable.

3.1 Development Phases

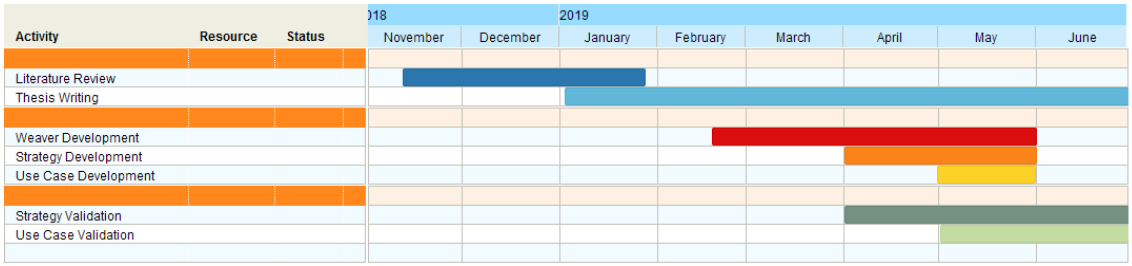
The objective of this dissertation project was to determine if the LARA framework was adequate for developing tools for a dynamic language such as JavaScript. In order to test this, we developed Jackdaw - a working AOP tool for Javascript using the LARA framework - and validated it with a set of use cases. In order to do this in an organized fashion, we used an incremental approach where work on certain software areas began as soon as the needed background implementation was done.

The development of Jackdaw and document writing of this dissertation project consisted of several phases:

1. Literature review - research about current existing AOP tools for JavaScript and methodologies for evaluating them.
2. Thesis writing - writing the dissertation document itself.
3. Weaver development - obtaining a Javascript AST to be used by the weaver, created from parsed JavaScript code, and incremental development of join points, attributes and actions for the weaver.
4. Strategy support - implementing support for already existing LARA strategies.
5. Use case support - implementation of custom strategies in order to perform code obfuscation.
6. Strategy validation - validation of already existing LARA.
7. Use case validation - use case validation by checking if requirements are fulfilled.

The previously mentioned phases can be seen distributed in a time schedule on Figure 3.1. The purpose of this work plan was to provide a basis for time management and allocation of priorities during the development of the project. The writing of the dissertation document took place alongside project coding and development. The first few weeks were spent developing the weaver for JavaScript, *Jackdaw*. After Jackdaw supported a sufficient number of joint-points, the development of strategy support began. Likewise, when a basic set of supported strategies were implemented, the development of use case support began, namely features like variable renaming and obfuscation. Any remaining needed joint-points or strategies were developed in this last stretch of time. Validation of implemented strategies and use-cases were done in parallel with their development.

Figure 3.1: Figure presenting the current work plan



3.2 Summary

On this chapter we presented a reviewed version of the work plan of this work, which describes the different phases of this dissertation project and how they were scheduled.

Chapter 4

Development

In this chapter we will cover what we consider to be the most important aspects of the Jackdaw development process. We present the technologies used during development, and explain their role and how they were used. We will describe the internal mechanisms of the Jackdaw tool and how it uses several different constructions in order to properly modify the incoming AST tree from the parsed Javascript code. We also discuss the implemented obfuscation algorithms and the challenges we encountered, and how they were solved. Finally, this chapter concludes by presenting a set of Jackdaw's main features, outlining their functionality and making a retrospective analysis on the grading criteria introduced on chapter 2.

4.1 Used Technologies

In order to develop Jackdaw and meet the proposed challenges of this work, we assembled a wide range of technologies. The following list describes the technology stack and a short description regarding how each one was used.

- **Java 11** - The LARA framework source code is written in Java and the preferred way to develop a new LARA-based tool is to write the tool also in Java. Java development was done using Eclipse, which allowed for a very quick setup of the project after importing the LARA framework dependencies using a Maven repository [1].
- **Nashorn Engine** - Nashorn is a JavaScript engine developed in the Java programming language by Oracle which comes included in Java 8[nas]. The Nashorn engine is used by Jackdaw in order to parse files and generate Javascript files using packages written in Javascript, such as Esprima and Escodegen.
- **Gson** - Gson is a Java library that can be used to convert Java Objects into their JSON representation. It can also be used to convert a JSON string to an equivalent Java object. Gson can work with arbitrary Java classes, including compiled classes for which we do not

[1] The lara-framework can be found here: https://mvnrepository.com/artifact/pt.up.fe.specs/lara-framework

have the source-code [gso17]. This was the library chosen for dealing with JSON and it is extensively used by Jackdaw to serialize and deserialize Javascript code nodes.

- **Esprima** - Esprima is a high performance, standard-compliant ECMAScript parser written in ECMAScript (also popularly known as Javascript) and it can be used to perform lexical analysis (tokenization) and syntactic analysis (parsing) of a JavaScript program [esp]. The ability to parse Javascript code into an AST structure was an obvious need early on. The lack of readily available Javascript parsers written in Java, and the time it would take to write a custom parser, led us to consider using the Java 8 Nashorn Javascript engine to execute a package that would let us parse Javascript source code into an abstract syntax tree. Esprima was the perfect choice for this task since it allows us to quickly parse Javascript files with several option flags to include extra information in the AST, such as the inclusion of comments, location of the code corresponding to each node, etc.

- **Escodegen** - Escodegen (escodegen) is an ECMAScript code generator from Mozilla's Parser API AST [esc]. After Jackdaw parses all the source code and applies the transformations to the AST, it needs to convert it back into Javascript code. In order to do this we decided to use Escodegen to generate the output code from the modified JSON tree structure. In addition to the advantage of not having to write a custom code generator, Escodegen already contains several flags which allow the user to define how the generated code should be formatted.

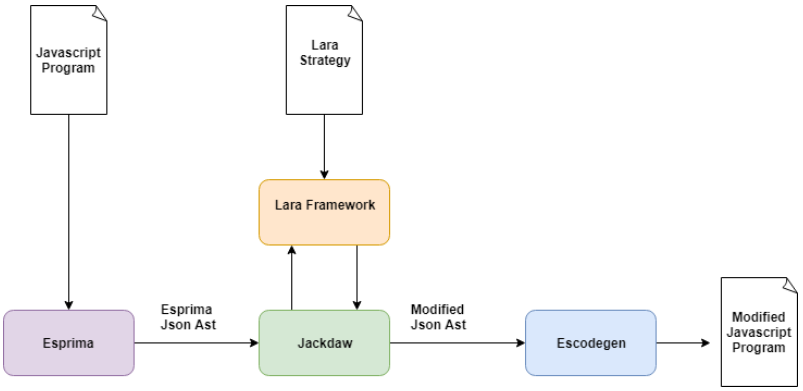
4.2 Jackdaw

Figure 4.1 describes the structure of the tool Jackdaw. It uses the JavaScript parser Esprima to tokenize and parse the code into an abstract syntax tree (AST), as a JSON object. After some thought and planning, we decided to directly use the AST produced by Esprima, instead of producing a new AST from it. The Jackdaw weaver interacts with the AST and applies the appropriate actions according to the requests of the LARA framework, which executes the LARA code. In the final step, Jackdaw uses the JavaScript library Escodegen [esc] to generate code from JSON AST and produce a modified Javascript program as result.

4.2.1 Jackdaw AST

The first task in the development of Jackdaw was the Jackdaw AST module. The objective of this module is to parse JavaScript source files into an Abstract Syntax tree, and back to code. In order to do this we used Esprima and Escodegen, a JavaScript parser and generator respectively. However, since both of these packages are written in Javascript, we had to use the Nashorn engine in order to successfully execute Javascript code from within a Java runtime.

Figure 4.1: Jackdaw flowchart



4.2.2 AST Structure

The result from parsing a Javascript source is a JSON object with all the codes structures and hierarchies defined by Esprima. Initially, we planned to convert this JSON object into an intermediate node structure which contained only relevant information, before converting them into Jackdaw join points. However, with some investigation we were able to conclude that it was possible to skip this step and use the JSON object to directly create the Jackdaw join points. Additionally, this approach has the added advantage of being able to use existing tools such as Escodegen to generate the output code. If we used an intermediate node structure we would have to either implement our own Javascript generator, or a generator for Esprima-compatible JSON, in order to convert this structure back into Javascript code. Any added or computed properties that might be useful to have added to the original JSON structure yielded by Esprima can instead be computed on join point side (much like virtual properties), leaving the original JSON structure intact.

4.2.3 Jackdaw Join points

The join points supported by Jackdaw represent Javascript language structures and statements present in the source code, such as declarations, while statements, if statements, try-catch statements, etc. It is possible to use the LARA syntax to query and capture these join points in order to do further operations. Each join point is mapped to a Java class, which internally queries the corresponding JSON node and provide methods that return the attributes and perform the actions specified in the weaver language specification.

4.2.4 Jackdaw Query Engine

Early in the development of Jackdaw we noticed that there would be a need to perform certain searching tasks from within a language node or join point. Examples include getting the parent of a join point, getting all child nodes, getting all descendant nodes of a certain type, etc. In order to provide this functionality we developed the Jackdaw Query Engine, which was developed as a static class that provides several search functions.

4.2.5 Join point Parent Mapping

A very common property needed for AOP actions or implementing strategies is the ability to obtain the parent node of a join point, e.g., obtain the scope of a given statement. This presented a particular challenge due to the fact that the JSON structure generated by Esprima does not contain any information regarding the parent of a node. In order to address this problem three options were considered.

- Recursively iterate through the JSON structure of each source file and add a parent property to each node. This option was discarded due to the fact that this would lead to a circular JSON structure, which is not supported by the Gson library.

- Recursively iterate through the JSON structure of each source file and add a unique identifier, and a property containing the identifier of its parent. This option was considered more reasonable due to the fact that it no longer leads to a circular JSON structure. However, it still requires the generation of unique identifiers and updating the parents every time the AST is changed.

- Recursively iterating through the JSON structure of each source file and adding a <child, parent> pair to a map. Every time a change is made to the AST, this method is recursively executed from the root of the project, ensuring that an AST changing action never yields a child-to-parent map that contains stale information.

We solved this problem by implementing a class that performs the algorithm described in the third option. This algorithm was further improved by adding a "dirty" flag to the child-parent map. Every time we make a change to the AST, this flag is set to true. When we request the parent of a node, a map rebuilding process is triggered if the state is set to true. This ensures we are always working with an updated map, while avoiding unnecessary rebuilding operations, which improves performance.

4.2.6 Join Point Creator

Every time we use the LARA selector language to query for a particular type of join point, there is a need to search for the corresponding JSON object in the AST and then wrap it around the correct type of join point. The Join point creator module offers this capability by checking the attributes of the input JSON object and trying to wrap it into the correct join point type. If there is no corresponding join point type for the input node, then the Join point creator will wrap it around a generic type of join point, which only contains methods for properties common to all join points. The generic join point type is specially useful for join point attributes which return other join points, since this particular use case may at times require the wrapping of JSON nodes for which join points have not been yet defined.

4.3 Code Obfuscation

We have chosen obfuscation of Javascript source code as a use case to validate Jackdaw. Code obfuscation is a deliberate act to make code unintelligible and hard to comprehend to a human. The process of obfuscating code is done by applying transformations to the source code such that the result is much more difficult to read, while maintaining the same functionality [BS05]. Unlike encrypted code, obfuscated code can be compiled and executed by anyone.

Code obfuscation is particularly relevant for the Javascript that is executed by Internet browsers, since the code is always available to the user. Javascript obfuscation is often used in this situation to help ensure protection against code theft, and to hinder reverse engineering that provide insights about the code original meaning and purpose [dSRS16, QBB08].

In order to successfully achieve the obfuscation of Javascript code, the Jackdaw tool relies on different techniques such has code refactoring and flattened control flow. These techniques will be explained further in this section.

4.3.1 Variable renaming

Renaming a variable declaration is a simple technique we have used in order to make code unintelligible and usually serves as one of the first steps of obfuscation. The renaming of a code structure involves renaming the keyword of an object or variable declaration, following by a propagation of this renaming to all the code instructions that reference it. The ability to rename variables and other code structures has many uses and its utility is not limited to code obfuscation. Figure 4.2 shows a LARA aspect developed for Jackdaw which selects all the declarations in the source files and renames them to a new automatically generated keyword. On figures 4.3 and 4.4 we can observe the input and generated JavaScript code, respectively.

This example shows a repeated declaration of the variable **a**, which is allowed in Javascript by use of the declaration keyword **var**. The underlying mechanisms used in our implementation of the variable renaming action allow us to properly detect the different scopes of these two variables (which happen to have the same name) and properly rename them and their references to two different variable names. This is particularly useful in code obfuscation due to the fact that, depending on the obfuscation techniques used, repeated variable declarations may cause problems in the generated obfuscated JavaScript code. It was easier to address this problem by simply renaming the variables and their references into different names, which is something that would typically also be done in order to obfuscate the actual names of the variables.

An issue we have encountered when dealing with Javascript variable renaming is that, due to the dynamic nature of Javascript, the renaming operation may require the propagation of the new name not just in the statements after the declaration, but above the declaration as well. The renaming function implemented in Jackdaw also allows to propagate renaming to the statements about, however, it is not well developed and is deactivated by default.

```
aspectdef RenameDeclarations

  var i = 0;
  select declarator end
  apply
    i++;
    var newName = "new_var_"+i;
    $declarator.refactor(newName);
  end

end
```

Figure 4.2: LARA aspect that renames all declarations.

```
var a = 1
while(a < 5){
  a++;
}
var a = 3
if(a >= 5){
  console.log(a);
}
```

Figure 4.3: Input JavaScript code.

```
var new_var_1 = 1;
while (new_var_1 < 5) {
    new_var_1++;
}
var new_var_2 = 3;
if (new_var_2 >= 5) {
    console.log(new_var_2);
}
```

Figure 4.4: Generated JavaScript code.

4.3.2 Control Flow Flattening

Control Flow Flatenning (CFF) is the main obfuscation technique applied by Jackdaw in order to achieve code obfuscation. CFF is a code transformation whose idea was first described by Wang et al. [WHKD00] and consists in gathering all the code blocks contained in a program, which might be located in different scopes, and placing them next to each other, usually inside a large switch statement within a loop with control flow variables that decide which case gets activated at each iteration of the loop [BS05].

The CFF transformation that Jackdaw implements is mainly an adaptation of the CFF algorithm described by Balakrishnan et al. [BS05], which presents a CFF algorithm for C++. Since the target language of Jackdaw is Javascript, we had to make several changes to the original algorithm, for instance, due to the fact that Javascript does not support *label* and *goto* statements in the same way that C++ does.

The structure that was most dramatically impacted was the *switch* statement, which originally heavily relied on *goto* statements in order to generate its obfuscated counterpart. The solution was to first generate a *switch* statement which assigns a value to a control variable depending on the matching case, and then generate new control flow cases for each case of the original *switch* statement. This strategy effectively replaces the use of *goto* with a more extensive usage of control flow variables.

Figures 4.5 and 4.6 show an example of a Fibonacci sequence algorithm before and after it

is obfuscated by Jackdaw. As we can see, all the declared identifiers were renamed and the code contained in the function was completely flattened to a single *while* with a *switch* statement. As a result, it is quite difficult for the human eye to understand what this simple algorithm does.

```
function fibonacci(num){
  var a = 1, b = 0, temp;

  while (num >= 0){
    temp = a;
    a = a + b;
    b = temp;
    num--;
  }
}
fibonacci(13);
```

Figure 4.5: Fibonacci algorithm using a while cycle.

4.3.3 Opaque Predicates

In order to complement the obfuscation achieved by variable renaming and control flow flattening, we decided to also implement a proof-of-concept opaque predicate obfuscation. Opaque Predicate obfuscation is a low-cost and stealthy control flow obfuscation technique used to add superfluous branches [XMW16]. Predicates are essentially functions that return boolean values which are known to the obfuscator software, in this case Jackdaw. These predicates are then introduced in the original code in such a way that they do not alter the original logic but make it harder for a potential reverse engineering effort to succeed.

Jackdaw allows the user the ability to generate a number of random predicates and applying them to Javascript nodes via selections and aspects. Additionally, it is possible to combine opaque predicates obfuscation with control flow flatenning via an optional flag. If this flag is activated, the control flow algorithm will generate N predicates per file and will randomly assign them to all nodes containing logical expressions (ex: *while, if, do, for, etc*). It is important to note that in this process the predicate functions themselves will also be flattened, making it very difficult for a user to infer what value a logical expression in the code will evaluate to during runtime. Since the Jackdaw obfuscator always knows the return value of these predicates, it is able to apply them in such a way that the original values of the logical expressions are not altered. The predicates themselves are also not distinguishable from the original code which has been flattened. Figure 4.7 shows an example of the simplest type of opaque predicates - a function that was randomly chosen to return false or true.

```
function id_303316e8_69bc_46cc_bd27_41d5c6dcdc0f(id_b9c6d674_a4e0_43a7_814c_a85fe818d315) {
    var id_8be2a0cd_a06c_415e_b1ba_7440e30ca288 = 1, id_819824fa_e1bb_4640_8b59_03cf187a58a6
        = 0, id_4fc078f0_100a_4140_b914_0202e7b4b3d0;
    var id_44d650f0_ac7d_420d_a5c8_01fa3fd87c4b = 'id_edc601e9_12f9_4615_aadc_024e1b62175c';
    while (id_44d650f0_ac7d_420d_a5c8_01fa3fd87c4b != '
        id_8e61087b_738d_4a4e_93ae_2f72946464c4') {
        switch (id_44d650f0_ac7d_420d_a5c8_01fa3fd87c4b) {
        case 'id_edc601e9_12f9_4615_aadc_024e1b62175c':
            id_44d650f0_ac7d_420d_a5c8_01fa3fd87c4b = '
                id_1115e25a_3c77_4b28_b0a1_752c38bc9fb7';
            break;
        case 'id_1115e25a_3c77_4b28_b0a1_752c38bc9fb7':
            if (id_b9c6d674_a4e0_43a7_814c_a85fe818d315 >= 0)
                id_44d650f0_ac7d_420d_a5c8_01fa3fd87c4b = '
                    id_06e88c81_41ea_45d1_ba42_8ef748126ea2';
            else
                id_44d650f0_ac7d_420d_a5c8_01fa3fd87c4b = '
                    id_8e61087b_738d_4a4e_93ae_2f72946464c4';
            break;
        case 'id_06e88c81_41ea_45d1_ba42_8ef748126ea2':
            id_4fc078f0_100a_4140_b914_0202e7b4b3d0 = id_8be2a0cd_a06c_415e_b1ba_7440e30ca288
                ;
            id_8be2a0cd_a06c_415e_b1ba_7440e30ca288 = id_8be2a0cd_a06c_415e_b1ba_7440e30ca288
                 + id_819824fa_e1bb_4640_8b59_03cf187a58a6;
            id_819824fa_e1bb_4640_8b59_03cf187a58a6 = id_4fc078f0_100a_4140_b914_0202e7b4b3d0
                ;
            console.log(id_819824fa_e1bb_4640_8b59_03cf187a58a6);
            id_b9c6d674_a4e0_43a7_814c_a85fe818d315--;
            id_44d650f0_ac7d_420d_a5c8_01fa3fd87c4b = '
                id_1115e25a_3c77_4b28_b0a1_752c38bc9fb7';
            break;
        }
    }
}
id_303316e8_69bc_46cc_bd27_41d5c6dcdc0f(13);
```

Figure 4.6: Fibonacci algorithm obfuscated by jackdaw.

```
function id_8a8e532c_4104_4f5c_9d78_a9d2670ab40d() {
    return true;
}
function id_ced2e1e7_92db_4de7_801b_e5e782d1a66d() {
    return false;
}
function id_9c8433d1_ebdf_4e25_8111_b189cc108c11() {
    return false;
}
function id_b690ceab_7c29_4feb_93b8_f281e5b197da() {
    return true;
}
```

Figure 4.7: An example of four opaque predicates.

4.4 Jackdaw Features

On this section we will describe that features that Jackdaw presents in order to satisfy the use cases which are the goal of this work.

4.4.1 Selection of Language Nodes

One of the most common use cases of AOP is the ability to select a certain language node or set of nodes from the source code and apply an aspect to it, as it can be seen on example 1.

Example 1. Select all function declarations in a file and append automatically generated documentation.

In order to do this Jackdaw implements a wide range of different types of join points with different properties which correspond to Javascript language nodes. Most of these join points can be selected via the LARA query system, while others are meant to be merely obtained as properties of selected join points.

Some of the more important join points the user can select in order to interact with Javascript code are: `project`, `file`, `functionDeclaration`, `classDeclaration`, `declaration`, `forStatement`, `whileStatement`, `switchStatement`, `ifStatement`, `tryStatement` and `expressionStatement`. Although Jackdaw currently supports more join points, these are sufficient to make the tool functional as a proof-of-concept software and to allow common LARA packages to work properly.

4.4.2 Code insertions

Jackdaw allows the user to select a join point and append (or replace) another join point or string to it (see Figure 4.8). Jackdaw also increases the security of this operation by parsing the string inserted by the user and printing out an error if it does not match a valid Javascript syntax.

```
1 aspectdef insertDocumentation
2
3   select function end
4   apply
5     var text = "/* Function: " + $function.name + " */";
6     $function.insert("before", text);
7   end
8 end
```

Figure 4.8: LARA aspect for inserting a comment.

4.4.3 Common Strategies and Packages

There are several LARA packages and aspects that can work in different LARA-based weavers for different target languages. In order to get these packages the developer of a new LARA weaver must implement certain join points with certain properties or actions. With Jackdaw we managed to implement the required features so that we were be able to support some basic LARA packages (i.e., `lara.code.Logger`, `lara.code.Timer` and `weaver.JoinPoints`).

4.4.4 Obfuscation

As it was already discussed with greater detail in previous sections, Jackdaw provides the user with an obfuscator package that is able to obfuscate source code with two different algorithms: control flow flattening and opaque predicates. The user can use a general obfuscation function which selects by default all functions in a file and applies obfuscation to them. Alternatively, the user can use the available lower-level obfuscation methods to create his own custom obfuscation algorithm, choosing parameters such as the number of predicates to use or the type of language structures to apply obfuscation.

4.4.5 Customization of Output Syntax

One of the great advantages of using Escodegen [esc] as a code generator is the fact that it allows to set custom syntax rules for the generated Javascript code. These rules are setup by adding a configuration JSON file which can specified from the Jackdaw IDE. All generated code will have these rules applied to it.

4.5 Evaluation Through Grading Criteria

The proposed evaluation for the Jackdaw tool was to implement common strategies that existing LARA tools already support, and by using a comparison model for evaluation of AOP tools. As mentioned in section 4.4.3, this first objective was successfully achieved.

Regarding the second objective, on chapter 2 we introduced the comparison model proposed by [HHL15] and which will be used to evaluate the potential success of the Jackdaw tool in an AOP environment. We will now provide a specific analysis for Jackdaw regarding three grading criteria: **invasiveness**, **briefness** and **maturity**. Additionally, we will talk about the common basic strategies which were implemented and provide an overall analysis of Jackdaw, mentioning interesting features from an user perspective and also pointing out what sort of further improvements can be done to make Jackdaw competitive with its industry peers.

4.5.1 Invasiveness

As it was predicted, the fact that the Jackdaw tool is based on LARA makes it completely non-invasive due to the fact that LARA is a code generation oriented tool. Since the AOP aspects

written in LARA are kept separate from the source code, the Javascript programs are always kept in its original form, while the new generated modified files are written into a different folder specified by the user.

4.5.2 Briefness

The LARA scripting language uses the Nashorn javascript engine in order to run, which in this case makes it extremely similar to its target language, with the exception of a few syntax keywords and statements which are specifically designed for Aspect-Oriented programming. The overall advantage is that a user does not have to work with two completely different languages in order to modify the original source code and write the AOP aspects. In order to justify the grading which will be later shown, we will offer a comparison with aspect.js, the other technology that received a double plus in our evaluation. Lets take a look at the function shown in Figure 4.9. Lets say that we wanted to a add some code to be executed before the initial statements in the body of this function. One approach in aspect.js, would be the example shown on Figure 4.10. As we can see, we have to declare these new statements inside a function and then add them with the "before" and "add" methods. Now lets take a look at a LARA implementation on Figure 4.11. Since the source and AOP languages are separated, we cannot access the function object anymore, instead we have to query for all functions and their bodies and apply the needed changes when we have the right one. While these two tools have different operational paradigms, we can see that the LARA language is equally clean, effective and non-verbose in accomplishing this task. Additionally, this example could be made even less verbose by implementing specific methods for prefixing instructions to a function.

```
1 function fun1(){
2     console.log("original code.")
3 }
```

Figure 4.9: Example function.

Finally, we can conclude that writing aspects in LARA is accessible for both experienced and non-experienced users in AOP. The fact that the LARA IDE gives immediate access to the Jackdaw API and its documentation to the user is a great help.

4.5.3 Maturity

Maturity is rated by measuring the amount of implemented features and comparing with other existing tools. In the case of Jackdaw, it must also be compared to other LARA-based tools in order to grade its implementation of the LARA API. The main features that Jackdaw provides at the time of writing of this document are:

```
let aspectjs = require("aspectjs");
let before = require('aspectjs').before;
let addAdvice = require("aspectjs").addAdvice;

let wrapper = {
    fun1: fun1
}
function fun2(){
    console.log("prefix code")
}
aspectjs.before(wrapper,"fun1").add(fun2);

wrapper.fun1();
```

Figure 4.10: aspectjs aspect for inserting a prefix function.

```
aspectdef addPrefix

  select function.blockStatement end
  apply
    if($function.name == "fun1"){
      $blockStatement.children[0].insert("before","console.log('prefix code.');");
    }
  end
end
```

Figure 4.11: LARA aspect for inserting a prefix function.

- A wide but incomplete range of selectable Javascript language nodes and properties.
- A sufficient implementation of join point types and properties that enable common LARA packages to work properly.
- A LARA package that provides obfuscation of Javascript using two different popular techniques.
- The ability of customize the syntax of the output Javascript code according to a definable set of rules.

It can be concluded that Jackdaw in its present state was able to meet the proposed goals of this dissertation project and can be considered a usable tool. We can consider as proof of this the fact that it was possible to implement an entire LARA package for obfuscation that implements two obfuscation algorithms using the Jackdaw API and its implemented join points and properties. Finally, while Jackdaw can be considered a usable proof-of-concept tool, it would need to be further developed in order to be able to seriously compete with the current industry standards. The main area of improvement would be completing the scope of join points that can be selected to include the entirety of the Javascript language. Finally, on Table 4.1 we present our evaluation

of Jackdaw while comparing it to the previously analyzed tools. The self-given grading aims to reflect what was previously said in this section about each of the grading components.

Table 4.1: Table showing comparison between tools.

Component / Tool	AspectScript	AOJS	AspectJS	aspect.js	aspectjs	**Jackdaw**
Invasiveness	-	+	-	-	-	++
Briefness	+	+	+	-	++	++
Maturity	+	-	-	+	-	–

4.5.4 Jackdaw Code Metrics

This section provides an evaluation on the amount of code that was written in order to develop Jackdaw. Table 4.2 presents the amount of written logical lines, for each of the main development languages, Java and LARA. It is important to note that the table contains only user-written code. The code automatically generated by the weaver generator contains 2652 lines of code, which is more than the total lines of code manually written, while the LARA framework itself contains more than 24.000 lines of Java code [PCB+18]. This is an indication of the effort saved by using the LARA framework to develop the tool, as opposed to writing the tool from scratch.

Project	Java	LARA
Jackdaw	1286	412
JAST	151	0

Table 4.2: Jackdaw Logical line measurements.

4.6 Summary

On this chapter we covered the most important aspects of the Jackdaw development process. We presented the technologies used in the development of Jackdaw and explained the reasons for their usage and impact in functionality. We described the most important internal mechanisms of Jackdaw and how it uses several different constructions in order to work and properly modify the incoming AST tree from the parsed Javascript code. We described and discussed the implemented obfuscation algorithms and the challenges we encountered while solving them. Finally, we concluded this chapter by presenting a set of Jackdaw's main features, outlining their functionality and making a retrospective analysis on the grading criteria components that were introduced in the state-of-the-art review.

Chapter 5

Experimental Evaluation

This chapter will cover the experimental usage and evaluation that was done in order to validate Jackdaw as a reliable tool. This chapter will begin by introducing some relevant LARA aspects that were developed in order to assist the experimental evaluation process. Next, we will cover some of the issues encountered both on the implementation side and on the performance side. Finally, the chapter presents results regarding Jackdaw usage in obfuscating and running a battery of benchmarks.

5.1 Developed Aspects

We developed several LARA aspects in order to facilitate the process of running and debugging benchmarks. These aspects were also integrated in Jackdaw's obfuscation API, in order to give the user some more functionality and control over the obfuscation process.

5.1.1 Configurable Obfuscations

Jackdaw offers a method which allows the user to control and customize the obfuscation the user wants to apply to a file. This is done by passing a JSON object to the function which contains the configuration. This configuration object contains the minimum and maximum amount of opaque predicates to be generated (the true number we will be a random number in this interval), which functions should have obfuscation applied to them, and which obfuscation techniques (opaque predicates, control flow flatenning, or both). Figure 5.1 shows an aspect that uses the method `applyCustomObfuscation` to apply a custom obfuscation configuration to a file.

5.1.2 Learning Obfuscation Configurations

One of the most useful aspects that Jackdaw provides is the ability to automatically discover an obfuscation configuration for a Javascript source file. Since the current obfuscation method has some limitations and occasionally can fail when obfuscation a code structure, the alternatives would be to either blindly call a global obfuscation function, ignoring the potential problems of

```
1
2 import obfuscation.Obfuscator;
3 aspectdef obfuscateFile
4
5   select file end
6
7   apply
8     var configuration = {predicates:{min:3,max:6}, functions:[{name:"fun1",algorithms:["
          opaque","flat"]}],excluded:[]};
9     applyCustomObfuscation($file,configuration);
10  end
11 end
```

Figure 5.1: LARA aspect uses a custom configuration in order to apply obfuscation to the functions of a file.

obfuscating everything in a file, or manually investigating what functions can and can not be obfuscated.

This aspect solves this problem by iterating over all the functions in a file, obfuscating them one-by-one, and attempting to run the file each time a function is obfuscated. If there is any runtime error or a timeout in the execution of the program, the aspect will exclude the last obfuscated function, revert the code back to the previous state, and continue the process until the last function is tested. The method then returns a working configuration which contains all the functions for which there were no obfuscation or runtime problems. The configuration can be obtained as a return value, and it is also saved as a JSON object to the Jackdaw configuration folder.

Figure 5.2 shows an example of an aspect that uses the method `discoverObfuscableFunctions` to test and learn a working configuration for the file to be obfuscated. The previous mentioned method can then be called passing the obtained configuration as an argument.

```
1
2 import obfuscation.Obfuscator;
3 aspectdef getObfuscableFunctionsAndExecute
4
5   select file end
6
7   apply
8     var config = discoverObfuscableFunctions($file);
9     applyCustomObfuscation($file,config);
10  end
11 end
```

Figure 5.2: LARA aspect that uses the discovery method in order to create a working obfuscation configuration for a file.

5.2 Implementation and Performance Issues

During the implementation and testing of Jackdaw we encountered several issues which are worth mentioning, some of which were fixed during the dissertation project. Most of these issues were related to performance.

5.2.1 Obfuscation Issues

Due to time constraints and also the scope of this project, it was not possible to implement custom obfuscation for every single type of Javascript structure. Additionally, we detected some bugs in the list of supported code constructions. Here is a listing and description of these problems:

1. The obfuscation of for-loops with implicit variable declaration may cause runtime errors. This is due to the fact that the renaming of the declared variable works for the loop's code block but this renaming does not propagate to the update step and verification step of the for loop.

2. The same situation applies to the alternative for-of loop syntax for iterating elements of an array or list.

3. Attribution of class properties with the **this** keyword are not being translated to the top of the function body as the CFF algorithm proposes. This is an interesting case because this operation depending on its context and position can be for all effects and purposes a variable declaration or just a simple attribution. The end result is that the order of the declarations will be wrong, thus the likely-hood of a runtime error is high.

5.2.2 Performance Issues

Some performance issues started being detected after testing with very large files, such as the ones we found in the benchmarks. These issues usually led to an unacceptable large consumption of RAM and usually also lead to very large execution times and general unresponsiveness. After careful re-evaluation and debugging, and with information from profiling tools, we found out this problem was connected to the creation of a large number of Nashorn engines. Every time certain tasks were executed (e.g., code insertion), a new engine was created. After detecting that this was the culprit for the lack of performance, we implemented a system of reusing previously called Nashorn engine instances, which fixed the performance issues.

5.2.3 Parsing Performance

Through an initial analysis using debugging and time measuring, it was possible to determine that a very large percentage of execution time was spent parsing the Javascript source files. Parsing is of course delegated to Esprima [esp], which is run within the Nashorn Javascript engine. Our reasoning at the time was that using Esprima was a trade-off which allowed us to spend more

time developing more interesting features within the scope of this dissertation project. However, solving this particular type of issue was important to make Jackdaw usable.

As previously mentioned, problems with parsing performance were also partly related to unnecessary calls done to the Nashorn engine. In order to improve parsing performance, we introduced several optimizations which reduced the number of calls and initializations of the Nashorn engine, which saved results of previous parsings for future aspect executions.

5.3 Performance Analysis

In order to correctly measure the performance impact of both applying obfuscation to files and executing obfuscated files we decided to run a set of benchmarks taken from the JetStream 2 benchmark collection [jet]. Throughout this section we will present the recorded results of applying obfuscation and executing obfuscated code. The components of the machine used to run these benchmarks can be found on Table 5.1.

Component	Model
CPU	AMD Ryzen 5 2600x
RAM	16Gb DDR4 3200 Mhz
OS	Windows 10

Table 5.1: Machine specifications

5.3.1 Performance of Applying Obfuscation

Initially, the time of applying obfuscation was quite large due to redundant calls to the Nashorn engine. After this problem was corrected, it was possible to apply obfuscation algorithms to large Javascript files in a few milliseconds. Table 5.3 shows the time for applying complete obfuscation to the selected benchmarks in the Jetstream 2 benchmark collection [jet]. As we can see, we managed to successfully obfuscate all examples in less than one second. It is important to note that the measured times only includes the actual time spent applying obfuscation constructions, and not time needed to learn the obfuscation configuration, like the presented example on the beginning of this chapter.

Table 5.4 presents the number of obfuscated functions out of the total number of functions for each benchmark. The obfuscation aspect, which was developed as a proof-of-concept, was able to obfuscate a very large number of functions for each benchmark (82% of the functions, on average). The remaining 18% of functions that were not obfuscated, was usually related to the already known issues described on subsection 5.2.1. An obfuscation bug will either cause a runtime error, or fail to assign proper exit variable labels in the control flow flattening algorithm which will cause an infinite loop which is then detected by the learning method via a timeout.

Finally, Table 5.5 presents the increase on the number of lines of the obfuscated files. As expected, we can notice a large increase in the number of lines (sometimes up to more than double

the original size). This is mostly due to the applying of Control Flow Flattening, which transforms small functions into very large while cycles with switch statements that contain many cases inside. We can also see that Jackdaw was able to obfuscate in under a second files with hundreds of lines of code.

Algorithm	Obfuscation	Lines
Fibonacci	0.2s	33
Base64	0.2s	213
Delta-Blue	0.5s	499
Gaussian-Blur	0.1s	168
N-Body	0.2s	132
Navier-Stokes	0.8s	367
Poker	0.2s	299
Raytrace	0.6s	613
Richards	0.3s	313
String-Unpack-Code	0.1s	121
Tagcloud	0.3s	200

Table 5.2: Time measurement of applying obfuscation.

Algorithm	Obfuscation
Fibonacci	0.2s
Base64	0.2s
Delta-Blue	0.5s
Gaussian-Blur	0.1s
N-Body	0.2s
Navier-Stokes	0.8s
Poker	0.2s
Raytrace	0.6s
Richards	0.3s
String-Unpack-Code	0.1s
Tagcloud	0.3s

Table 5.3: Time measurement of applying obfuscation.

5.3.1.1 Impact in Execution time

It is expected that applying obfuscation techniques leads to a degradation of the execution time for the obfuscated programs, which is one of the trade-offs when applying obfuscation to a program. In order to do some evaluation of this effect, we decided to measure some algorithms, comparing the execution time of the original code with its correspondent obfuscated code. Table 5.6 shows the performance of the evaluated benchmarks. Additionally, we calculate the slowdown multiplier related to obfuscation for each benchmark.

Analyzing these results we can notice that we have a wide range of outcomes. Most of times the obfuscated version will be slightly slower than its original counterpart while other times there

Algorithm	Functions	Functions
Fibonacci	1/1	100%
Base64	6/8	75%
Delta-Blue	17/21	81%
Gaussian-Blur	5/6	83%
N-Body	9/11	81%
Navier-Stokes	11/24	46%
Poker	5/6	83%
Raytrace	4/4	100%
Richards	12/12	100%
String-Unpack-Code	4/5	80%
Tagcloud	8/9	89%
Geometric Mean		**82%**

Table 5.4: Total obfuscated functions by the learning method

Algorithm	Lines before	Lines after	Increase Percentage
Fibonacci	33	145	339%
Base64	213	361	69%
Delta-Blue	499	734	47%
Gaussian-Blur	168	359	114%
N-Body	132	327	148%
Navier-Stokes	367	557	52%
Poker	299	395	32%
Raytrace	613	778	27%
Richards	313	556	77%
String-Unpack-Code	121	281	132%
Tagcloud	200	368	84%
Geometric Mean			**78%**

Table 5.5: Obfuscation line increase

is quite a large difference. For instance, Fibonacci is the benchmark that has the biggest slowdown (more than 300%). We suspect this is due to Fibonacci's execution time being mostly bound by computation. If a benchmark execution time is bounded by memory operations, the increased computation required by obfuscation is masked by the time the application is waiting for the memory. The conclusions we make from the analysis of these results is that while it is expected that obfuscation will generally cause a slowdown of runtime execution, ultimately, it depends on the particular code that is being obfuscated.

5.4 Summary

On this chapter we covered several topics regarding the experimental usage, process and evaluation of Jackdaw in order to achieve its validation. We discussed several important LARA aspects that were developed and which were critical in the experimental evaluation process, and pointed

Algorithm	Non-Obfuscated	Obfuscated	**Slowdown**
Fibonacci	1.7s	23.1s	13.5x
Base64	11.8s	12.4s	1.05x
Delta-Blue	1s	1.2s	1.2x
Gaussian-Blur	1.1s	8.1s	7.36x
N-Body	1.1s	1.1s	1x
Navier-Stokes	1.1s	1.6s	1.45x
Poker	0.9s	0.9s	1x
Raytrace	1.1s	1.2s	1.09x
Richards	1.3s	1.4s	1.07x
String-Unpack-Code	1.3s	1.3s	1x
Tagcloud	1.4s	1.5s	1.07x

Table 5.6: Runtime measurement

out how they were added to the Jackdaw obfuscation package. We discussed several issues encountered both in the implementation side and in the performance side. During this analysis, we presented data regarding the obfuscation process and execution of obfuscated benchmarks. Finally, we ended this chapter by discussing the impact of obfuscation in execution time.

Chapter 6

Conclusion

This chapter states what was created during this dissertation project, how we used literature review in order to better aim the goals of this application and how we used the implementation of obfuscation in order to validate the developed tool. Finally, we will talk about the improvements that could be made to Jackdaw in order to make it a more relevant tool when it comes to Aspect Oriented Programming for Javascript.

6.1 Concluding remarks

During the development of this thesis we managed to meet the projects objectives and successfully develop an Aspect-Oriented Programming tool for Javascript through the use of the LARA framework. Our analysis of past work in this area and literature review allowed us to understand and grade an AOP tool for Javascript through a series of criteria. With this in mind, we consider that the developed tool could successfully overcome the proposed challenges, and in certain criteria, such as Invasiveness, is an improvement over the state-of-the-art. Once the Jackdaw tool was sufficiently matured, we started using it in order to develop an obfuscation module. We then integrated this developed module within Jackdaw itself and analyzed the results of obfuscating and executing a collection of known benchmarks, from which we could obfuscate 82% of their functions, on average, always under 1 second.

6.2 Future work

While we believe that Jackdaw is a working functional tool at the present moment, its functionality could be vastly improved by increasing the scope of the Javascript language that Jackdaw supports. Adding more structure support, actions and features would greatly improve its usability. Regarding the obfuscation module, its usability could be improved by solving some currently known bugs which were stated previously, and increasing the number of code structures that can be obfuscated by the Control Flow Flattening algorithm. The random opaque predicates package can be extended

to generate more complex predicates, and more obfuscation algorithms could be added into the package.

References

[ABBC18] Hamid Arabnejad, João Bispo, Jorge G Barbosa, and João MP Cardoso. Autoparclava: An automatic parallelization source-to-source tool for c code applications. In *Proceedings of the 9th Workshop and 7th Workshop on Parallel Programming and RunTime Management Techniques for Manycore Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms*, pages 13–19. ACM, 2018.

[asp] Aspectjs. http://www.aspectjs.com. Accessed: 2019-01-08.

[asp15] aspect.js. https://github.com/mgechev/aspect.js/, 2015.

[BPN+13] J. Bispo, P. Pinto, R. Nobre, T. Carvalho, J. M. P. Cardoso, and P. C. Diniz. The matisse matlab compiler. In *2013 11th IEEE International Conference on Industrial Informatics (INDIN)*, pages 602–608, July 2013.

[BRC14] João Bispo, Luís Reis, and João MP Cardoso. Multi-target c code generation from matlab. In *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, page 95. ACM, 2014.

[BS05] Arini Balakrishnan and Chloe Schulze. Code obfuscation literature survey. *CS701 Construction of compilers, 19*, 2005.

[CC18] Tiago Carvalho and João M. P. Cardoso. An approach based on a dsl + api for programming runtime adaptivity and autotuning concerns. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, SAC '18, pages 1211–1220, New York, NY, USA, 2018. ACM.

[CCC+12] João M.P. Cardoso, Tiago Carvalho, José G.F. Coutinho, Wayne Luk, Ricardo Nobre, Pedro Diniz, and Zlatko Petrov. Lara: An aspect-oriented programming language for embedded systems. In *Proceedings of the 11th Annual International Conference on Aspect-oriented Software Development*, AOSD '12, pages 179–190, New York, NY, USA, 2012. ACM.

[CCC+16] João M. P. Cardoso, José GF Coutinho, Tiago Carvalho, Pedro C Diniz, Zlatko Petrov, Wayne Luk, and Fernando Gonçalves. Performance-driven instrumentation and mapping strategies using the lara aspect-oriented programming approach. *Software: Practice and Experience*, 46(2):251–287, 2016.

[CCH08] Chun Chen, Jacqueline Chame, and Mary Hall. Chill: A framework for composing high-level loop transformations. Technical report, 08-897 (pp. 136–150), U. of Southern California, 2008.

REFERENCES

[dSRS16] Antonio Pedro Freitas Fortuna dos Santos, Rui Miguel Silvares Ribeiro, and Filipe Manuel Gomes Silva. Web application protection, May 12 2016. US Patent App. 14/894,919.

[EFB01] Tzilla Elrad, Robert E. Filman, and Atef Bader. Aspect-oriented programming: Introduction. *Commun. ACM*, 44(10):29–32, October 2001.

[esc] Escodegen. https://github.com/estools/escodegen. Accessed: 2019-02-03.

[esp] Esprima. http://esprima.org. Accessed: 2019-02-03.

[FDNT15] Johan Fabry, Tom Dinkelaker, Jacques Noyé, and Éric Tanter. A taxonomy of domain-specific aspect languages. *ACM Comput. Surv.*, 47(3):40:1–40:44, February 2015.

[For17] Philip Ford. aspectjs. https://github.com/pford68/aspectjs, 2017.

[gso17] Gson. https://github.com/google/gson, 2017.

[HHL15] Wenhao Huang, Chengwan He, and Zheng Li. A comparison of implementations for aspect-oriented javascript. In *2015 International Conference on Computer Science and Intelligent Communication*. Atlantis Press, 2015.

[jet] Jetstream 2. https://browserbench.org/JetStream/. Accessed: 2019-06-15.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G Griswold. An overview of aspectj. In *European Conference on Object-Oriented Programming*, pages 327–354. Springer, 2001.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP'97 — Object-Oriented Programming*, pages 220–242, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[Lad09] Ramnivas Laddad. *Aspectj in action: enterprise AOP with spring applications*. Manning Publications Co., 2009.

[nas] Nashorn. https://www.oracle.com/technetwork/articles/java/jf14-nashorn-2126515.html. Accessed: 2019-04-04.

[OKM+11] Akira Ohashi, Atsuto Kubo, Tomohiko Mizumachi, Kazuki Eguchi, Shinichi Murakami, Ryuichi Takahashi, Hironori Takahashi, Yoshiaki Fukazawa, Hideyuki Kanuka, Toshihiro Kodaka, Rieko Yamamoto, Youichi Nagai, Nobukazu Yoshioka, Fuyuki Ishikawa, and Hisashi Ikari. Aojs: Aspect-oriented programming framework for javascript. *Computer Software*, 28(3):114–131, 2011.

[PCB+18] Pedro Pinto, Tiago Carvalho, João Bispo, Miguel António Ramalho, and João M.P. Cardoso. Aspect composition for multiple target languages using lara. *Computer Languages, Systems & Structures*, 53:1 – 26, 2018.

[QBB08] Jiancheng Qin, Zhongying Bai, and Yuan Bai. Polymorphic algorithm of javascript code protection. In *2008 International Symposium on Computer Science and Computational Technology*, volume 1, pages 451–454. IEEE, 2008.

[RBC17] Luís Reis, João Bispo, and João MP Cardoso. Compiler techniques for efficient matlab to opencl code generation. In *Proceedings of the 5th International Workshop on OpenCL*, page 29. ACM, 2017.

[SGSP02] Olaf Spinczyk, Andreas Gal, and Wolfgang Schröder-Preikschat. Aspectc++: An aspect-oriented extension to the c++ programming language. In *Proceedings of the Fortieth International Conference on Tools Pacific: Objects for Internet, Mobile and Embedded Applications*, CRPIT '02, pages 53–60, Darlinghurst, Australia, Australia, 2002. Australian Computer Society, Inc.

[TLT10] Rodolfo Toledo, Paul Leger, and Éric Tanter. Aspectscript: Expressive aspects for the web. In *Proceedings of the 9th International Conference on Aspect-Oriented Software Development*, AOSD '10, pages 13–24, New York, NY, USA, 2010. ACM.

[vDKV00] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *SIGPLAN Notices*, 35:26–36, 01 2000.

[WHKD00] Chenxi Wang, Jonathan Hill, John Knight, and Jack Davidson. Software tamper resistance: Obstructing static analysis of programs. Technical report, Charlottesville, VA, USA, 2000.

[XMW16] Dongpeng Xu, Jiang Ming, and Dinghao Wu. Generalized dynamic opaque predicates: A new control flow obfuscation method. In Matt Bishop and Anderson C A Nascimento, editors, *Information Security*, pages 323–342, Cham, 2016. Springer International Publishing.