Faculdade de Engenharia da Universidade do Porto

RAVEN: A Node.js Static Metadata Extracting Solution for JavaScript Applications

Carlos Maria Antunes Matias

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Prof. João Manuel Paiva Cardoso

July 25, 2016

RAVEN: A Node.js Static Metadata Extracting Solution for JavaScript Applications

Carlos Maria Antunes Matias

Mestrado Integrado em Engenharia Informática e Computação

July 25, 2016

Abstract

Metadata provide useful information about any type of digital resource. Examples of metadata are author and date of creation of a file. By extracting additional metadata from source code files, through static analysis, one can collect additional information, besides the already existent, and gather a better understanding of the resources and compare those resources with similar ones.

Static code analysis consists in examining code files without the need of executing them. This type of analysis allows the creation of a representation of the code which can be used for obtaining more metadata, in the form of software metrics (e.g. lines of code and code complexity). Software metrics are the result of measurements performed over software.

The aim of this dissertation is to develop a metadata extraction solution for JavaScript applications, by leveraging the *Node.js* environment, and by statically analysing JavaScript code. This analysis results in a group of software metrics that, in conjunction with other data such as libraries in use and JavaScript constructions used, produce a valuable tool for the company that proposed this dissertation and allow the comparison of files regarding their complexity/quality.

The solution is used to study the effect of obfuscation techniques upon the software metrics and to reason about the general complexity of code which relies on a specific library.

Resumo

Metadados são um tipo de dados que se encontram em qualquer tipo de recurso digital e que fornecem informações pertinentes sobre estes, como data de criação e autor. Ao analisar estáticamente ficheiros de código, é possível extrair metadados adicionais, para além daqueles já existentes, possibilitando uma melhor compreensão sobre os recursos analisados e a comparação com outros recursos da mesma espécie.

Análisar estaticamente um ficheiro de código consiste em examiná-lo sem ter que o executar. Esta análise permite obter uma representação do código, a qual pode ser utilizada para obter mais metdadados, na forma de métricas de software. Métricas de *software* são o resultado de medições efetuadas sobre *software*, sendo exemplos o número de linhas de código de um ficheiro ou a sua complexidade/qualidade.

O objetivo desta dissertação prende-se com a criação de uma solução de extração de metadados de aplicações JavaScript, através da plataforma Node.js, e que analisa estáticamente código JavaScript. Desta análise surgem um conjunto de métricas de software, que, em conjunto com outros dados como bibliotecas em uso e construções de JavaScript utilizadas, permitem obter uma ferramenta que traduz valor para a empresa proponente e comparar ficheiros quanto à sua complexidade.

A solução é usada para estudar o efeito de técnicas de ofuscação sobre métricas de *software* e analisar o impacto que uma biblioteca poderá ter na complexidade de determinado código.

Agradecimentos

Ao meu Pai e irmãs, o meu obrigado por estarem do meu lado e terem sido capazes de aguentar a mais tremenda das fases e que permitiram, sem sombra de dúvida, a realização deste projecto.

À minha avó, tia e aos meus padrinhos, por terem um valor inestimável e serem sempre um porto seguro.

À minha namorada, por ser o farol nos tempos em que a motivação quebra e por me encorajar a fazer sempre mais, à luz da sua dedicação.

Aos meus amigos, por serem um dos pilares da minha vida.

Ao meu orientador e aos supervisores, que sempre bem me aconselharam e dos quais possuo o maior respeito.

Aos colegas e docentes da faculdade de engenharia, que moldaram o meu percurso de vida e que para além de engenheiro fizeram de mim um melhor ser humano.

Carlos Matias

“Strive not to be a success,
but rather to be of value.”

Albert Einstein

Contents

List of Figures

LIST OF FIGURES

List of Tables

Abbreviations

API	Application Programming Interface
AST	Abstract Syntax Tree
CBO	Coupling Between Objects
CC	McCabe's Cyclomatic Complexity
CFG	Control Flow Graph
CMT	Number of Comments
CSS	Cascading Style Sheets
DIT	Depth of Inheritance Tree
DOM	Document Object Model
ES2015	ECMAScript 2015 Specification
GQM	Goal-Question-Metric Paradigm
GUI	Graphical User Interface
HTML	HyperText Markup Language
HV	Halstead's Volume
IDE	Integrated Development Environment
IR	Intermediate Representation
JSAI	JavaScript Abstract Interpreter
JSON	JavaScript Object Notation
LCOM	Lack of Cohesion in Methods
LOC	Number of Lines of Code
MI	Maintainability Index
MVC	Model-View-Controller
NOC	Number of Children
ODM	Object Document Mapper
OO	Object-Oriented
ORM	Object Relation Mapper
RFC	Response for a Class
STMT	Number of Statements
TAJS	Type Analysis for JavaScript
WMC	Weighted Methods per Class

Chapter 1

Introduction

The following work presents the dissertation entitled *RAVEN: A Node.js Static Metadata Extracting Solution for JavaScript Applications*. This dissertation was proposed by *Jscrambler, S.A.* [1], a software company, based in Porto, specialized in the protection of web applications.

This chapter includes a contextualization of the work, where the subjects of metadata and JavaScript programming are addressed. After the contextualization, the motivation and objectives of the dissertation are presented, followed by a brief overview on how the dissertation is structured.

1.1 Context

Metadata is commonly described as "data which describes other data" [GR04] and serves several purposes such as facilitating the search of resources, organizing file systems and providing digital resource identification. The word metadata can carry different meanings: It can determine that an information set is understandable by a machine or that it represents a description of a resource or of a group of resources [GR04].

According to Guenther and Radebaugh [GR04], there are three distinct types of metadata: descriptive, structural and administrative. The first type holds information about the resource like name and author. The second indicates how a specific resource is structured. An example of this type is the order of the pages that compose a document. The latter includes the rights of access to a resource and preservation data. Preservation data encompasses information such as creation and modification dates and the file extension.

Metadata resources are usually produced upon the resource creation and can be a part of the resource itself or gathered separately. Their existence is key in ensuring the accessibility of a resource remains intact [GR04].

Source code files, as any other type of digital material, contain metadata resources, which can be extended through static analysis (i.e. by analysing the file contents without executing the code

[1] `https://jscrambler.com`

[Wög05]). This type of analysis enables the construction of structures such as the Abstract Syntax Tree (AST), which describes the composition of the code under analysis [Wil97]. Metadata can be retrieved from this characterization of the code files and others (for instance, Control Flow Graphs (CFG) [All70]) in the form of software metrics, which are measurements, calculated to evaluate and provide an understanding of software [LC87]. Examples of commonly used software metrics are lines of code (LOC) and cyclomatic complexity [McC76] [GCP12].

Although not directly related to metrics, information such as dependencies, libraries and the ECMAScript standard (standardized specification of scripting languages, where JavaScript is included [ecm]) in use are also relevant metadata resources within the scope of the dissertation work.

JavaScript is a high-level dynamic programming language that supports object-oriented (OO), imperative and functional programming styles [Cro08]. It is one of the three core web programming languages, alongside HTML and CSS [Fla11]. Since its beginning, JavaScript is used on the client-side spectrum, where it is responsible by providing behaviour to web pages. It has since then evolved to be used as a full stack programming language (i.e. to be used on the client-side and server-side of web applications), mainly due to the *node.js*[2] [nodd] server-side environment .

Within the scope of this dissertation, JavaScript presents an opportunity but also several challenges. An opportunity is presented because of JavaScript's popularity. Since many organizations are adopting JavaScript as their primary programming language [JI12], being proficient in coding and understanding JavaScript can be a valuable asset.

On the other side, a challenge is presented due to the fact that the language has several quirks, some of which are solely related to the way some features are implemented, or the lack of them, and other which can pose a challenge when statically analysing JavaScript code [S^+14] [MS13]. Also, since JavaScript is not, by nature, an OO programming language [Cro08], it restricts the set of software metrics which can be retrieved. This is due to the fact that some metrics can only be calculated when analysing fully-fledged OO programming languages. These OO metrics are related to features which only OO languages possess such as classes, polymorphism and inheritance [CK94]. These challenges are further discussed on Sections 2.1 and 2.2 of the following chapter.

1.2 Motivation and Objectives

This dissertation serves two different purposes:

The first one is directly connected to the *Jscrambler's* business needs. *Jscrambler* performs services upon source code files, which can be submitted to the company's servers either by using their website or via an Application Programming Interface (API). By having a large quantity of metadata present for each submission, it is possible to gather information such as what frameworks are being used by the customers or what ECMAScript specification constructions are more widely used (a more detailed explanation about this subject is given on Section 3.2.2). This information

[2]`https://nodejs.org`

is a valuable asset as it can direct the *Jscrambler's* future work. Knowing which libraries are more popular or which ECMAScript standard is in use can direct efforts to adapt *Jscrambler's* services to the specific characteristics of the code, thus improving the quality of the service, for example. Having this information collected automatically saves time and resources that would otherwise be spent in surveying the customers or having someone manually reviewing the code.

The second one is related to the scientific scope of the dissertation. By studying and characterizing software metrics around JavaScript, which translate complexity and/or quality, and analysing different JavaScript code samples, it is possible to achieve two distinct objectives.

The first objective is to reason about how distinct, in terms of software metrics, two versions of the same software/code are, where one is the original version and another is the result of performing one of *Jscrambler's* services upon the original. These services range from a series of transformations made to the source code in order to obfuscate it, to anti-tampering measures and code minification (removal of whitespaces and new line characters). The differences between both version can lead to optimizations and further development of *Jscrambler's* obfuscation algorithm.

The second objective is to study how similar JavaScript libraries impact the software metrics of the code built upon them. By gathering data about projects built with different tools it is possible to infer which libraries yield code more complex. Although the interests of many developers usually favor the support, documentation and feature set of a library and over performance benefits and metrics, interests usually reserved for academic research [GA13], being able to chose a library which has a less impact on the code developed can present a benefit in terms of code organization, readability and performance.

With this in mind, the following are the main research questions to be addressed by this dissertation:

1. How do the retrieved software metrics behave when analysing code files in their original state and with different levels of *Jscrambler's* protection services? Can one identify the same code sample before and after applying the service through the metrics?

2. Making use of *Jscrambler's* dataset, when comparing the metadata of projects which use similar frameworks or libraries, can one conclude that there is an inherent complexity associated to the usage of a library by analysing the software metrics retrieved?

The main objective of this dissertation, besides addressing the previously mentioned questions, is the creation of a metadata extraction solution for JavaScript applications by leveraging the *node.js* environment [nodd]. The metadata retrieved from the extraction process should be anonymous, that is, there should not be any trace back to the original file from where it was extracted (business requirement). A high-level representation of the built solution is presented on Figure 1.1.

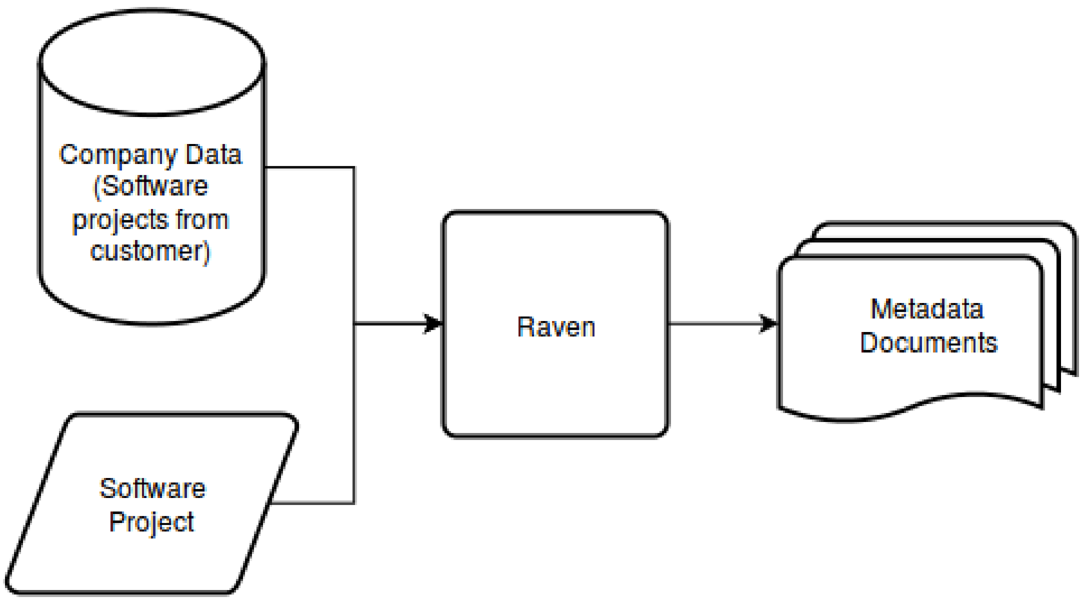


Figure 1.1: High-level representation of the Raven solution.

1.3 Structure of the Dissertation

This report consists of four additional chapters. Chapter 2 addresses the state of the art of the topics that lie within the dissertation's scope, those being static analysis and software metrics. Chapter 3 presents an in depth perspective of the architecture, components and steps taken to build the solution. Chapter 4 presents the detailed experimental results followed by the discussion and conclusions gathered from them. The final chapter, Chapter 5, presents the concluding remarks of the dissertation and future work perspectives.

Chapter 2

Literature Review

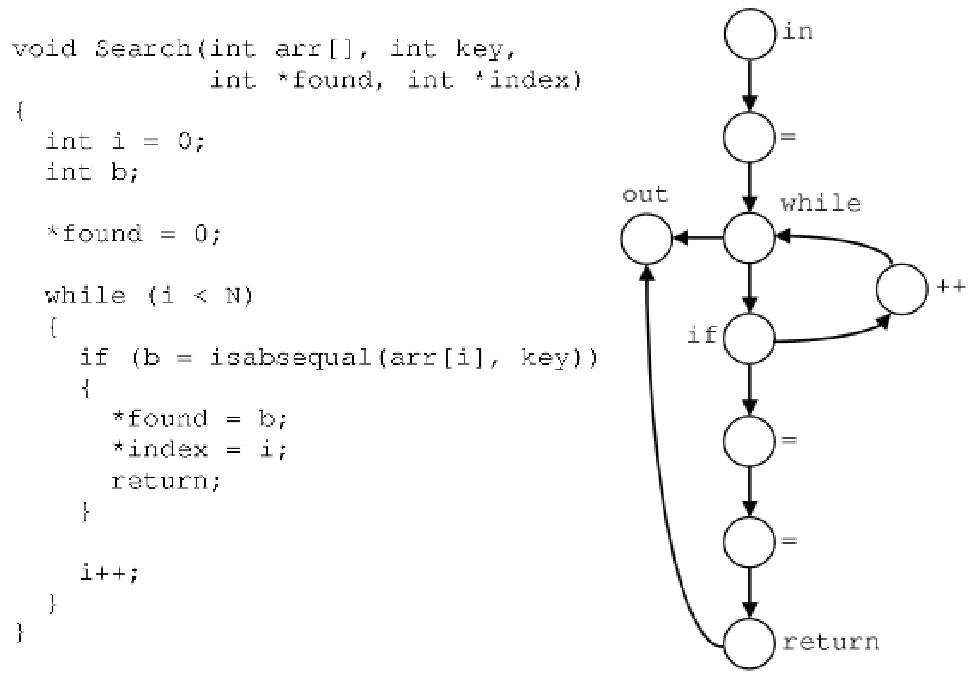
Considering the scope of this dissertation, the literature review addresses following topics: static analysis and software metrics. Static analysis represents the mean of obtaining a representation of code resources, which arises the possibility of extracting software metrics from them.

This chapter describes the state of the art in static analysis by introducing the topic, contextualizing it in terms of JavaScript and its challenges [MS13] [JMT09] and presenting the methods and evaluation authors applied to their static analysis tools. The chapter also presents the state of the art in software metrics by describing the core metrics that translate code complexity and quality while also addressing the state of JavaScript in regards to software metrics.

2.1 Static Analysis

Static analysis is the process of analyzing computer programs without executing them. For someone who develops software applications, static analysis allows for the optimization of code and is responsible for enforcing code correctness. Static analysis is present in most integrated development environments (IDE) [Wög05]. There are several methods available for statically analyzing a program. According to Wogerer [Wög05], three of the most common are data flow analysis, abstract interpretation and symbolic analysis.

Data flow analysis collects information about the flow of data across a specific program. This analysis does not require a specification of the semantics in use as they are implicitly defined in the algorithm. Data flow analysis divides the program into blocks of consecutive instructions and constructs a control flow graph (CFG) [All70], which depicts the control flow of a program across its basic blocks (blocks of sequential instructions). The analysis of the data flow allows the detection of dead code (nodes in the CFG without an entry edge, which are not the start of the program) and of duplicated computations (by verifying if some variable is defined in more than one block) [Wög05]. Both of these use cases are examples of how data flow analysis allows for the detection of errors in the source code.



```
void Search(int arr[], int key,
            int *found, int *index)
{
  int i = 0;
  int b;

  *found = 0;

  while (i < N)
  {
    if (b = isabsequal(arr[i], key))
    {
      *found = b;
      *index = i;
      return;
    }

    i++;
  }
}
```

Figure 2.1: Example of a CFG of a *Search* function [Gol10].

Abstract interpretation relies on creating an abstract semantics which is a superset of the program's concrete semantics. By doing this, the interpretation ensures every state of the program is represented by an interval of values, which requires a redefinition of boolean and arithmetic expressions. Usually, abstract interpretation also demands for the creation of an abstract value domain. An example of a domain is shown of Figure 2.2. Abstract interpretation facilitates the detection of semantic errors, such as division by zero or variable overflow [Wög05].

Finally, symbolic analysis deduces mathematical expressions about the program's expressions and is used when the values of the program are not constant. Its main use involves program optimization [Wög05].

2.1.1 Challenges of Analyzing JavaScript

Static analysis of JavaScript applications is no easy task since, like other scripting languages, it has a weak, dynamic typing discipline, responsible for silent type conversions that resolves mismatches (i.e. converting between variable types, such as *integer* to *boolean* without any warning). This is a powerful mechanism when developing applications since it grants a new level of freedom in the code development process and for the inputs the developed application allows. Nonetheless, it presents a challenge on the field of static analysis since the analyser may need to keep track of the type of any object/value holds [JMT09] [KDK$^+$14] [MLF13].

The tracking of the flow of data on JavaScript is also non-trivial since it supports objects, first-class functions (passing functions as arguments to other functions) and native exceptions [JMT09].

$$n \in \mathit{Num} \quad b \in \mathit{Bool} \quad str \in \mathit{String} \quad x \in \mathit{Variable} \quad \ell \in \mathit{Label}$$

$$\begin{array}{rcl}
s \in \mathit{Stmt} & ::= & \vec{s_i} \mid \textsf{if}\ e\ s_1\ s_2 \mid \textsf{while}\ e\ s \mid x := e \mid e_1.e_2 := e_3 \\
& \mid & x := e_1(e_2, e_3) \mid x := \textsf{toobj}\ e \mid x := \textsf{del}\ e_1.e_2 \\
& \mid & x := \textsf{newfun}\ m\ n \mid x := \textsf{new}\ e_1(e_2) \mid \textsf{throw}\ e \\
& \mid & \textsf{try-catch-fin}\ s_1\ x\ s_2\ s_3 \mid \ell\ s \mid \textsf{jump}\ \ell\ e \mid \textsf{for}\ x\ e\ s \\
e \in \mathit{Exp} & ::= & n \mid b \mid str \mid \textsf{undef} \mid \textsf{null} \mid x \mid m \mid e_1 \oplus e_2 \mid \odot e \\
d \in \mathit{Decl} & ::= & \textsf{decl}\ \overrightarrow{x_i = e_i}\ \textsf{in}\ s \\
m \in \mathit{Meth} & ::= & (\texttt{self}, \texttt{args}) \Rightarrow d \mid (\texttt{self}, \texttt{args}) \Rightarrow s \\
\oplus \in \mathit{BinOp} & ::= & + \mid - \mid \times \mid \div \mid \% \mid \ll \mid \gg \mid \ggg \mid < \\
& \mid & \leq \mid \& \mid \text{'}|\text{'} \mid \veebar \mid \textsf{and} \mid \textsf{or} \mid +\!\!+ \mid \prec \mid \preceq \\
& \mid & \approx \mid \equiv \mid . \mid \textsf{instanceof} \mid \textsf{in} \\
\odot \in \mathit{UnOp} & ::= & - \mid \sim \mid \neg \mid \textsf{typeof} \mid \textsf{isprim} \mid \textsf{tobool} \\
& \mid & \textsf{tostr} \mid \textsf{tonum}
\end{array}$$

Figure 2.2: Abstract syntax of *notJS*, an intermediate representation that serves as the basis for the abstract interpretation of Kashyap et al.'s work [KDK+14].

Jensen et al. [JMT09] present the following list of challenges when statically analyzing JavaScript:

- JavaScript's *prototype* objects are used to model inheritance since all the predefined operations are accessed through the *prototype*. It is then mandatory that the objects are modeled precisely within the analysis because there are no class files which define the objects.

- Objects in JavaScript are mapped from strings to values and properties can be added, removed and updated during execution. Also, the name of the properties can be dynamically computed.

- When accessing a non-existing property of an object, the application may return the value *undefined*. There is however a difference between a property having a value *undefined* or the value being *null*.

- There are free value conversions between types, with little regulation. Some of the conversions can present a challenge since they're not intuitive (for example, when converting between *boolean* and numeric values, the *boolean* value before and after conversion can be different).

- JavaScript distinguishes primitive values from wrapped primitive values. Although these types of values can appear to have the same behaviour, they can act differently under certain circumstances.
- Object properties can contain attributes, like *ReadOnly*. These properties can not be changed during program execution but must be taken into account if the analysis to be made is to be sound and precise.
- Function creation and calling can be done with a variable number of parameters. Each function receives an *arguments* parameter which behaves like an array but it is not one.
- Function objects can be constructors, methods or first-class functions. This versatility leads to a different behaviour according to the type of function object.
- The *eval* function can interpret strings as pieces of code.

2.1.2 JavaScript Static Analysis

As previously mentioned, the main use of static analysis is to detect errors and optimize code. Complying with this usage, although using different means, is Kashyap et al.'s [KDK+14] and Jensen et al.'s [JMT09] work.

Jensen et al's [JMT09] work was, to the best of our knowledge, the first to soundly analyze JavaScript considering the previous described characteristics. Their analysis, entitled type analysis for JavaScript (TAJS), is context sensitive, that is, it considers the function calling context when analyzing the target of a function call and performs *points-to* analysis [BS09]. In simple terms, *points-to* analysis allows the inference of object properties upon their access. The authors refer that context sensitivity is a valuable feature as it presents a mechanism for dealing with JavaScript's dynamism [JMT09].

Jensen et al.'s [JMT09] analysis represents the JavaScript program as a flow graph. In this graph, the first instruction of the global code of the program represents the program entry node. The types of nodes the program contains are: operations for declaring and modifying a variable and its properties, conditional statements, function calling and also throw and catch (for exceptions). In regards to edges, the authors distinguish between ordinary edges, exception edges, function *call* and *return* edges. The first type corresponds to intra-procedural control flow, the others correspond to the programming constructions as stated by their names. All the nodes that may raise exceptions contain an exception edge.

An abstract representation is created resorting to a lattice of abstract states (a lattice is a partially ordered set, where every two elements have an unique upper bound and an unique lower bound). There are lattices representative of every JavaScript native type, those being *undefined*, *null*, *boolean*, *numeric*, *string* and the object properties (being strings, but needing a separation from the primitive type). The abstraction level grows as object properties are modeled, then abstract states (which are a partial map from object labels to abstract objects) along with an abstract stack, ending on the final form of the analysis lattice, that assigns a set of abstract states to each node in the flow graph [JMT09].

The authors further developed their work by extending their analysis to include the document object model (DOM) and the browser application programming interface (API). The reason behind this extension is the fact that, in some web applications, the majority of the JavaScript code is running on the client-side , where it is bound to events. Therefore, it is not only reasonable but necessary to model the event system, which has dynamic properties, that include the registration and removal of different events, the event capture and trigger mechanism and also the properties that trigger the event. The modelling of the event system was done by creating flow graphs that portrayed the way each event behaved. By doing this, the authors managed to maintain their analysis objectives in the presence of a more dynamic environment [JMM11].

Not making use of flow graphs but also intending to soundly analyze JavaScript is Kashyap et al's tool, JavaScript Abstract Interpreter (JSAI) [KDK$^+$14]. JSAI's main objectives were formalizing a specification of the JavaScript semantics and to provide a customizable sensitivity setting on their analysis by testing their semantics against commercial applications, for soundness purposes.

The design of JSAI has three main components: an intermediate representation (IR)(Figure 2.2) with its semantics, an abstract semantics for the specified IR (where the configurable sensitivity lies) and the design of new abstract domains for the analysis. The authors rely on formal specification to avoid JavaScript's peculiar behaviours. Unlike other tools, JSAI's intermediate representation is based on an AST rather than on a CFG. Reasons for this decision were that some of the core JavaScript distinctive features like high-order functions, implicit exceptions and type conversions harden the task of creating and interpreting a CFG [KDK$^+$14].

With the IR's abstract syntax, the design of the formal concrete semantics was guided by the desire of converting the semantics directly to a testable form, in order to test it versus real JavaScript applications and also with the configurable sensitivity in mind. The semantics implementation turns state definitions into data structures and transitions rules into functions that transform a state into the next one [KDK$^+$14].

Further work was developed within the scope of statically analysing JavaScript. This work included the analysis of JavaScript in the presence of frameworks [MLF13] and extracting features from web applications where JavaScript is included [MS13].

In summary, Madsen et al.'s analysis [BS09] works by combining *points-to* analysis and use analysis . Points-to analysis allows for the gathering of an understanding of the flow from an actual parameter to a formal parameters, the first type being the value a variable holds when entering a function and the second one being the declaration (name) of the variable. An example of this is when a function is called and one of the arguments is present in a statement other than the input, flow can be inferred. Use analysis allows for the opposite to be done, as in that flowing obliges a compliance from formals to actuals, from the objects. This concludes in a more precise determination of object properties, which strengthens the analysis. The analyzer constructs a call-graph at runtime because of JavaScript's high-order functions, which means the *points-to* analysis and call graph relations are mutually dependent [MLF13].

Maras et al.'s [MS13] algorithm performs static and dynamic analysis to capture an event trace of the application, in order to determine which features a combination of HTML, CSS and JavaScript code translate (in this case, features mean actions performed, such as responding to a click). The algorithm, in regards to JavaScript interpretation, proceeds as following: JavaScript nodes are created and a JavaScript expression is evaluated (the program searches for nodes when creating them, so no duplicate nodes are added). Depending on the control flow, control dependencies are created. Tests are made to the expression to check if it is included in a loop/branch statement, in a catch statement or if it is a function statement. Next, the events are handled. The following checks involves verifying if the expression being considered is accessing identifiers or reading an array object. Following this, the dynamic creation of code is analysed, checking for JavaScript, CSS or HTML code creation from the JavaScript code. Lastly, the algorithm checks if the expression is sending or responding a request.

In order to identify the features retrieved, Maras et al.'s [MS13] algorithm proceeds to a graph marking process. In this process, there is a selector chosen and the dependency graph is transversed in order to identify which pieces of code are responsible for what.

To tackle the use of external libraries, in concurrence with Madsen et al.'s [MLF13], the authors used method stubs in order to simulate the desired behaviour of a piece of code when the library code was not available within the scope of the application being analysed [MS13].

Both Jensen et al's [MLF13] and Kashyap et al's [KDK$^+$14] made use of JavaScript benchmarks to validate their analysis, such as the Sunspider [1] and Google V8 [2] [goo] suites. Kashyap et al. [KDK$^+$14] also tested his analysis against real JavaScript applications, as well as Madsen et al. [MLF13]. Maras et al. [MS13] considered another way of evaluating their tool by comparing code before and after their feature extracting mechanism.

In regards to the static analysis of JavaScript, Table 2.1 displays an overview of the reviewed literature. The table presents, for each work reviewed, the mains goals, means used to satisfy those goals and evaluation methods used.

2.2 Software Metrics

The enhancement of a process can only occur if it is possible to measure some of its characteristics. With the evolution of software development and the increasing need to improve the software development processes, the demand for more and better software metrics arose . The necessity of

[1]https://webkit.org/perf/sunspider/sunspider.html
[2]https://developers.google.com/octane/

Table 2.1: Static Analysis Overview

Reference	**Goals**	**Means**	**Evaluation**
Jensen et al. (TAJS) [JMM11]	• Develop a sound static analyser for JavaScript which supplies error detection and auto-completion mechanisms.	• Flow graph construction. • Abstract representation via an analysis lattice.	Google Benchmark Suite [goo].
Maras et al. [MS13]	• Develop a method to identify and extract code and resources that implement features.	• Dependency graph construction. • Graph marking.	Three set of experiments comparing the developed tool's extracted code and the original code.
Madsen et al. [MLF13]	• Provide a tool for API surface discovery. • Analyse JavaScript code in the presence of frameworks and libraries.	• Call graph construction through pointer analysis and use analysis. • Automatic stub creation.	25 JavaScript applications from the Windows 8 store [win]
Kashyap et al. (JSAI) [KDK+14]	• Provide a formally specified static analysis platform for JavaScript.	• Abstract interpretation through the definition of an intermediate representation and an abstract semantics.	Benchmark suites, Mozzila Firefox Addons and Opensource JavaScript frameworks [KDK+14].

metrics is even higher when considering new technologies which do not have established practices [CK94].

Measurement activities should have clear objectives. One must indicate what is going to be measured and what attribute will come out of that measurement. To support this, one can make use of the Goal-Question-Metric paradigm (GQM) [BR88], which indicates that, before yielding any metrics, one must define the goals to be reached and following that, questions to be answered, which drive the metric investigation.

Software metrics are a polemic subject and usually susceptible to criticism such as lack of theoretical backing (appropriate mathematical properties), absence of measurement properties and being technology dependent [CK94]. Software measurement, like any kind of measurement, must heed to the science of measurement if it is to be accepted and valid. Complexity measurement sometimes fails to do so, by presenting a single numeric value which characterizes distinct views of complexity [Fen94]. This results in a metric translating attributes such as difficulty of programming/testing/understanding/maintaining, which are susceptible to different kinds of interpretation and are not theoretically supported [Wey88]. This concludes that, for transmitting a clearer notion of code complexity, one must use not a single metric but a set of metrics.

2.2.1 Evaluating Software Complexity

It is natural, when understanding and developing software, to determine which characteristics of the applications affect its cost and therefore its maintainability. Software complexity is one of the measures responsible for doing so, by translating how difficult a program is to comprehend. A software portrayed as complex should have objective and reliable metrics supporting it, and not only an intuitive notion derived from its inspection [LC87].

Software complexity has several uses in the software development scope, those being: defining requirements for the software to build, verifying the application developed against functional requirements and arranging trade-offs between maintenance and development budgets [LC87].

Li and Cheung [LC87] consider the following as the most popular and widely accepted complexity metrics: Halstead's metrics [Zus05], counts of statements (also known as logical lines of code) , lines of code and comments (usually abbreviated to LO*), McCabe's cyclomatic complexity [McC76] and the Knot measurement, a control flow measure.

Halstead's set of metrics is based on the number of operators and operands present in the code. Operators include basic arithmetic and comparison operators (+, +=, etc.), keywords (*while*, *if*, etc.) and names of subroutines/functions. Operands are the aggregation of all variables and constants. A distinction is made between total number of distinct operators (*n1*), total number of distinct operands (*n2*), total number of operators (*N1*) and total number of distinct operands (*N2*).

- *n1*: Number of distinct operators.
- *n2*: Number of distinct operands.
- *N1*: Total number of operators.
- *N2*: Total number of operands.
- *n1**: Number of potential (minimum) operators.
- *n2**: Number of potential (minimum) operands.

A series of other metrics are derived from these variables [LC87]:

$$Vocabulary\ of\ a\ program\ (n) = n1 + n2 \tag{2.1}$$

$$Program\ length\ (N) = N1 + N2 \tag{2.2}$$

$$Calculated\ program\ length(N*) = n1\ Log_2 n1 + n2\ Log_2 n2 \tag{2.3}$$

$$Program\ Volume\ (V) = N\ Log_2 n \tag{2.4}$$

$$Program\ potential\ (minimum)\ Volume\ (V*) = (2+n2*) Log_2 (2+n2*) \tag{2.5}$$

$$Program\ Level\ (L) = \frac{V*}{V} \tag{2.6}$$

$$Program\ Difficulty\ (D) = \frac{1}{L} \tag{2.7}$$

$$Program\ Level\ estimation\ (L*) = \frac{2}{n1}\ \frac{2}{N2} \tag{2.8}$$

$$Program\ Difficulty\ estimation\ (D*) = \frac{1}{L*} \tag{2.9}$$

$$Programming\ Effort\ (E) = \frac{V}{L} = \frac{n1\ N2\ NLog_2 n}{2\ n2} \tag{2.10}$$

The program volume (V) describes the size of the implementation, an estimation of the number of bits needed to encode the program. The programming effort (E) is often perceived as being a measurement of the mental activity required to conceive the algorithm.

McCabe's cyclomatic complexity is a standard metric in many applications. Its value is obtained from a CFG by subtracting the number of nodes to the number of edges and adding the double of the number of connected components in the program (these components being the number of subgraphs which are not connected between each other). In a strongly connected graph (in a graph where each vertex is reachable from any other vertex), the cyclomatic complexity is the number of independent circuits [LC87].

The knot metric measures the number of interconnected control structures. It's also possible to distinguish between the number of verified knots and the number of possible knots [LC87].

Different authors reviewed the metrics presented in regards to their downfalls and situations were the metrics cannot correctly identify how complex the code is.

The Halstead family of metrics presented a plausible degree of consistence with very little downfalls. The program's length equation is dependent of the size of the program and that the number of distinct operands plays a great part in dominating the approximation of the values [LC87]. However, Halstead's metric failed to comply with a property defined by Weyuker [Wey88], where the author states that *"for any two programs, their concatenation yields a equal or higher complexity value than any of the two programs alone"*. This failure presents a serious drawback on the Halstead set of metrics as it is hard to imagine that a part of a program can be more complex than the whole program.

McCabe's cyclomatic complexity has a fair degree of correlation with other control organization metrics and serves as the connection between volume metrics and control organization metrics [LC87]. Nonetheless, from Weyuker's review [Wey88], it can be noticed that McCabe's metric is not sensitive enough to yield a suitable range of complexity values (i.e. it produces the same value too many times). The author also determined that Halstead's and McCabe's metrics are not sensible to nested loops.

Traditional volume metrics such as number of statements (logical lines of code), number of lines and comment count were also investigated. Li and Cheung [LC87] came to the conclusion that the best volume metrics are the statement count and the number of lines without comments count, mostly because they cannot be easily inflated and because they present a high degree of correlation with other consistent metrics.

A metric not reviewed in either of the already cited sources is the maintainability index, which aims to give a perception on how maintainable certain code is. The authors that proposed this metric combined a series of already accepted metrics (lines of code (LOC), McCabe's cyclomatic complexity (CC) and Halstead's volume metric (HV)) and performed a regression analysis to obtain a formula for calculating maintainability [OH92]. One can obtain the maintainability index of a program by calculating the following:

$$MI = \quad 171 - 5.2 * ln(HV) - 0.25 * CC - 16.2 * ln(LOC) \tag{2.11}$$

Although it is certainly useful to know how maintainable a system is, having a formula with coefficients derived from regression analysis arises some concerns regarding its validity [min].

Besides the general software metrics described until now, there are some metrics which only apply to certain programing paradigms. This is the case of metrics associated with object-oriented (OO) design, which were the subject of study of Chidamber and Kemerer [CK94]. Designing a software system by the OO paradigm relies on adapting real world objects and relationships to classes of objects and their interactions. Following the concerns stated by Weyuker [Wey88], the authors sought out to identify a set of metrics which would be theoretically backed (as presented in their work in an extensive chapter defining properties and formulas which support them) and endorsed by empirical evidence. The metrics proposed by their work are the following:

- Weighted methods per class (WMC): Sum of the complexity measurement of each method of a class. The authors do not define the complexity metric in use to allow a more broad approach to their proposed metrics. The WMC metric provides a general idea on how hard (in terms of time) the class was to develop, the probable impact on child classes (since child classes inherit all the methods of the parent) and the reuse potential, since classes with a large WMC are usually application specific and therefore less reusable.

- Depth of inheritance tree (DIT): Considering a class, its DIT is the maximum length from the root parent class to the farthest child node. The bigger the DIT of a class the harder it is to predict its behaviours and the more complex the class design is.
- Number of children (NOC): The NOC metric represents the number of direct descendants some class has. A big NOC value usually means one of two things, either the class has a great reuse potential, therefore the number of children, or the class is incorrectly abstracted being the number of children a sign of erroneous subclassing. Also, the NOC value is generally an indicator on how relevant for the global system design some class is.
- Coupling between objects (CBO): The coupling metric of a class is the number of classes on which the class acts upon (coupled). A disproportionate amount of coupling is indicative of a monolithic design and lack of reusability.
- Response for a class (RFC): The response for a class is the collection of methods which can be executed in response to a received message. If several methods can respond to the same message then the class gets harder to test and debug. It can be an indicator on how complex a class is.
- Lack of cohesion in methods (LCOM): Two class methods are similar if they operate upon the same instance variables. The LCOM metric is achieved by subtracting the count of the methods of a class to the number of methods which possess a similar value of 0, that is, methods which operate on strictly different instance variables. A class is more cohesive the more similar methods it possesses, since this similarity suggests the design promotes encapsulation, which is usually desirable. A lack of cohesion usually leads to errors and implies that a class should be probably divided into subclasses.

Table 2.2 displays an overview of the software metrics reviewed. Each entry of the table displays the name of the metric, the meaning of it and observations about it, subsequent to the literature review.

2.2.2 JavaScript and Metrics

JavaScript, by not being a native fully-fledged OO programming language, misses out on inherently supporting several metrics (without proper adaptation) for OO systems, as the ones detailed in [CK94] and [BBM96]. The metrics described on these sources can characterize metrics related to inheritance and similarity between classes, which are native OO aspects. However, all the metrics described in the previous section, which do not apply to the OO paradigm, can be calculated from JavaScript code.

Riaz et al. [RMT09] presented a review of software maintainability metrics for the purpose of predicting how maintainable a system is, since it can directly impact the costs of a software project.

These metrics are usually associated with either quality or complexity since, as Fenton [Fen94] states, maintainability is an high level external attribute, that is, an attribute that is computed from the collection of other attributes that can be directly measured.

The review the authors made gathered 15 different studies and attempts to answer questions that revolve around identifying software maintainability metrics and their validity. Other research questions involved the determination of when should these metrics be gathered but that is out of the scope of this investigation. Many of the studies investigated had algorithms determining how maintainable a software would be, but four of the studies, that were proposed by the same group of authors, included an assessment model based on software metrics, those being Halstead's programming effort and program volume, McCabe's complexity, number of lines of code and comments. Given the subjectiveness of software complexity, the authors of these studies also included a possible subjective review of the software [RMT09]. This set of metrics overlaps with the one Graziotin [GA13] refers as being one that correctly characterizes applications, including ones built using JavaScript.

2.3 Summary

This chapter addressed the state of the art of the two main topics of the dissertation: Static analysis and software metrics, in general and focused on JavaScript applications.

On the subject of static analysis, three types of static analysis were introduced: Data flow analysis, abstract interpretation and symbolic analysis. Afterwards, the static analysis of JavaScript applications was discussed where several authors described different successful methods of performing static analysis [JMT09] [KDK$^+$14]. Furthermore, other authors were able to extend this analysis in the presence of frameworks or in the web environment [MS13] [MLF13] [JMM11].

Regarding software metrics, a set of metrics were introduced and detailed, as being the most commonly accepted software metrics since no complexity metric alone is viewed as being the most correct [Wey88]. This set of metrics comprises the Halstead family of metrics [LC87], McCabe's cyclomatic complexity [McC76], the maintainability index [RMT09] and a series of counts regarding lines of code. All of the mentioned metrics can be applied to JavaScript [RMT09].

Table 2.2: Software Metrics Overview

Metric	**Information**	**Observations**
Lines of Code (LOC)	Number of lines in the program.	Susceptible to inflation. Cannot provide meaningful conclusions by itself.
Statements (LLOC/STMT)	Number of statements in the program.	Same as LOC.
Comments (CMT)	Number of comments in the program.	Same as LOC.
McCabe's cyclomatic complexity (CC) [McC76]	Number of independent paths a program has. Computed from the program's CFG	Low sensitivity, especially when combining programs. Does not account for nested loops.
Cyclomatic complexity density (CC/LOC) [GK91]	Expresses CC as a percentage of the logical lines of code.	Same as CC.
Halstead's Metrics [Zus05]	Derivation of program information such as vocabulary, volume, difficulty and effort from the number of operators and operands.	Same as CC.
Maintainability Index (MI) [RMT09]	Combination of LOC, CC and Halstead's Volume. Depicts how easy a program is to maintain.	Formula is based on regression testing and averages values which can mask outliers. Formula has not been update since its proposal.
Weighted methods per class (WMC) [CK94]	Sum of the complexity measurement of each method of a class	Can only be applied to OO systems. Does not apply a specific complexity measure.
Depth of inheritance tree (DIT) [CK94]	Maximum length from the root parent class to the farthest child.	Can only be applied to OO systems.
Number of children (NOC) [CK94]	Number of direct descendants some class has.	Can only be applied to OO systems. Needs a global perspective of the system in order to be correctly interpreted.
Coupling between objects (CBO) [CK94]	Number of classes on which a class acts upon.	Can only be applied to OO systems.
Response for a class (RFC) [CK94]	Set of class methods which can be executed in response to a received message.	Can only be applied to OO systems. A high RFC does not always mean some class is complex.
Lack of cohesion in methods (LCOM) [CK94]	Number of class methods whose similarity is null minus the total count of class methods.	Can only be applied to OO systems.

Chapter 3

Developing Raven

3.1 Solution Architecture

The first iteration of the solution was a monolithic application which was responsible for the whole process from analysing the files to saving them to the database. The application was restructured in order to increase the modularity and reusability of the code. Two modules were created from the restructuring process, one responsible for the analysis of the projects and another, responsible for compiling the results and interacting with the databases.

The final architecture of the *raven* solution is as follows: One module, from here on out referred to as *raven-analyser* is responsible for analysing a file or a directory and output the analysis information. The other module, the *raven-interface*, is responsible for the interaction with the databases (one is a *MongoDB* [mona] database, where the analysis results are stored, and another is a *PostgreSQL* [pos], database, which holds the company's data) and for the processing chain of directories. A flow diagram representative of the solution is presented in Figure 3.1

The *raven-analyser* module is composed by 29 JavaScript files, with a combined size of 87kB and comprising 2,933 lines of code. The *raven-interface* module is constituted by 5 JavaScript files, with a combined size of 19kB and with a total of 544 lines of code.

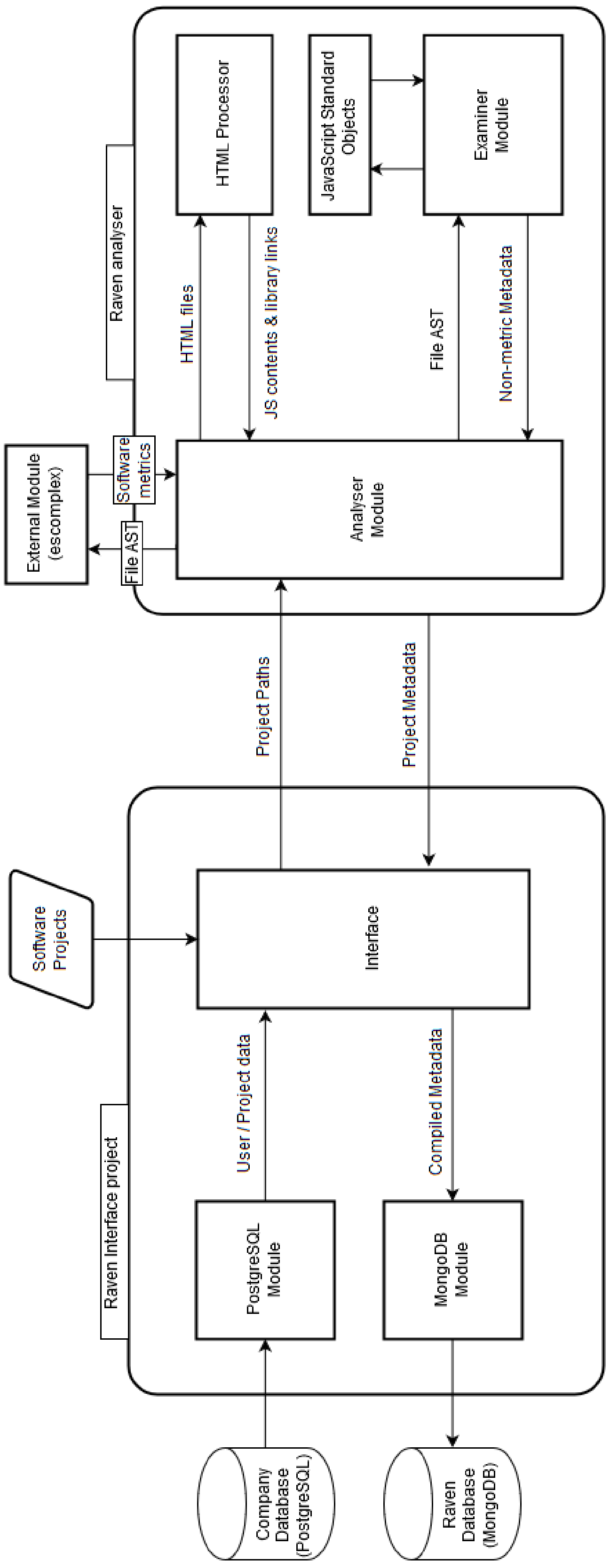


Figure 3.1: Flow diagram of the Raven solution.

3.2 *Raven-analyser*

The *raven-analyser* is the core of the metadata extracting solution as it is the module responsible for extracting the information from the source code.

3.2.1 Core Technologies

The following are the main packages used in the *raven-analyser* module. These are available via *npm* (the main package manager for JavaScript) [1].

Espree

There are many JavaScript parsers which translate JavaScript code into a format (generally an Abstract Syntax Tree (AST)) which can be used to collect information about the resource in question. The first parser to be part of the solution was *esprima* [espb]. *Esprima* works for any JavaScript code but lacks the features necessary to parse JSX code [jsx], which is an XML like syntax that *react* [rea], a tool for developing the front-end of applications, uses. JSX usually appears alongside JavaScript code and being unable to parse it would impede the analysis of projects which make use of it.

Espree's [espa] parsing is, by design, compatible with *esprima* and provides the same functionality which is parsing code and representing it in the JavaScript Object Notation (JSON) format. Listing 3.2 holds the JSON representation of an AST, parsed from the code present in Listing 3.1.

```
function add(num1, num2) {
    return num1 + num2;
}
```

Listing 3.1: Function which yields the AST representation of Listing 3.2.

```
{
    "type": "Program",
    "body": [
        {
            "type": "FunctionDeclaration",
            "id": {
                "type": "Identifier",
                "name": "add"
            },
            "params": [
                {
                    "type": "Identifier",
                    "name": "num1"
                },
```

[1] https://www.npmjs.com/

```
                {
                    "type": "Identifier",
                    "name": "num2"
                }
            ],
            "defaults": [],
            "body": {
                "type": "BlockStatement",
                "body": [
                    {
                        "type": "ReturnStatement",
                        "argument": {
                            "type": "BinaryExpression",
                            "operator": "+",
                            "left": {
                                "type": "Identifier",
                                "name": "num1"
                            },
                            "right": {
                                "type": "Identifier",
                                "name": "num2"
                            }
                        }
                    }
                ]
            },
            "generator": false,
            "expression": false
        }
    ],
    "sourceType": "script"
}
```

Listing 3.2: AST representation of the function in Listing 3.1. This representation corresponds to the one yielded by the online parser in the *esprima* website [espb].

Escomplex

Escomplex [esca] is the library responsible for doing all the calculations regarding software metrics. In order to do so it requires an AST representation of the code.

The main metrics calculated by *escomplex* are the following:

- Lines of code: The physical and logical count of lines of code. Physical lines of code are the source code lines excluding comments and the logical lines of code represent the number of executable statements.

- Cyclomatic complexity (CC): As defined by Thomas McCabe [McC76]. Counts the number of distinct cycles in the control flow graph of the program.

- Halstead's Metrics: Indicate the program volume, programming effort, program difficulty, time required to develop the program and probable number of bugs in the code from the number of operators and operands present in the code [Zus05].
- Maintainability index: A logarithmic scale from negative infinity up to 171, calculated according a formula (Equation 2.11) which takes into account the cyclomatic complexity, logical lines of code and Halstead's programming effort. The higher the index, better the maintainability.
- First-order density: The percentage of internal dependencies which are in use in the project. The lower, the better.

Escomplex calculates an aggregate of the metrics for the file or set of files submitted but it also calculates the result individually for each function it encounters.

The metrics calculated by this package cover the set of metrics deemed as being indicative of the general complexity/maintainability of code, while also providing some additional ones. For a more in depth view about software metrics please refer to Chapter 2.

Bluebird

JavaScript code, running in the *node.js* environment [nodd], usually performs tasks in an asynchronous way. This is mostly due to the fact that *node* is single threaded and performing tasks synchronously would severely hurt performance. However, operations which involve resources related to the file-system, such as reading from files, do not block the calling thread and allow the code on the main thread to continue processing. The main thread is then notified when the result of the asynchronous operation is available.

The primary way to deal with asynchronous tasks is by passing a *callback* to the calling function. *Callbacks* have a tendency to become a problem when there are a series of asynchronous operations which need to be done in a sequence. By chaining *callbacks* together, sometimes developers end up with code like the one in Listing 3.3. This is commonly known as the "*callback* hell" where the code grows not only vertically but also horizontally, which hurts the code's readability, maintainability and is considered to be a bad practice [Sim15].

There are many ways to solve this type of problem but one of the most accepted solutions are JavaScript *promises*. *Promises* are an abstraction to deal with an operation which is yet to be completed (asynchronous), to deal with a future value of an operation. *Promises* can be chained thus providing a way of handling the sequential execution of asynchronous operations. Listing 3.4 shows the "*promisification*" of the code depict in 3.3.

```
function isUserTooYoung(id, callback) {
    openDatabase(function(db) {
        getCollection(db, 'users', function(col) {
            find(col, {'id': id},function(result) {
                result.filter(function(user) {
                    callback(user.age < cutoffAge)
                })
            })
        })
    })
}
```

Listing 3.3: Code example of what is usually called in the JavaScript community as *callback hell*: a series of chained *callbacks* which grow the code horizontally. Taken from [cal]

```
function isUserTooYoung(id) {
    return openDatabase(db)
        .then(getCollection)
        .then(find.bind(null, {'id': id}))
        .then(function(user) {
            return user.age < cutoffAge;
        });
}
```

Listing 3.4: Implementation of the code used in Listing 3.3 using JavaScript promises. Taken from [cal].

ECMAScript2015 (ES2015), the new JavaScript standard, introduced native JavaScript *promises*, however, the native JavaScript implementation lacks features when compared to *promise* libraries that have been available before the standard. With this in mind, the *raven-analyser* makes use of the *bluebird* library [blu].

Escope

Before the ES2015 standard, there were only two constructions which created new scopes in JavaScript: *function* and the *catch* block of a *try/catch* statement. Without the knowledge of this behaviour, developers coming from an object oriented (OO) background are caught by surprise when they realize that variables created inside *for* loops, or other types of constructions, belong to the surrounding function or to the *global* scope. Lacking the knowledge of this feature can originate unexpected behaviours such as accessing a wrong *array* element when a *for* loop redeclares the iterator variable.

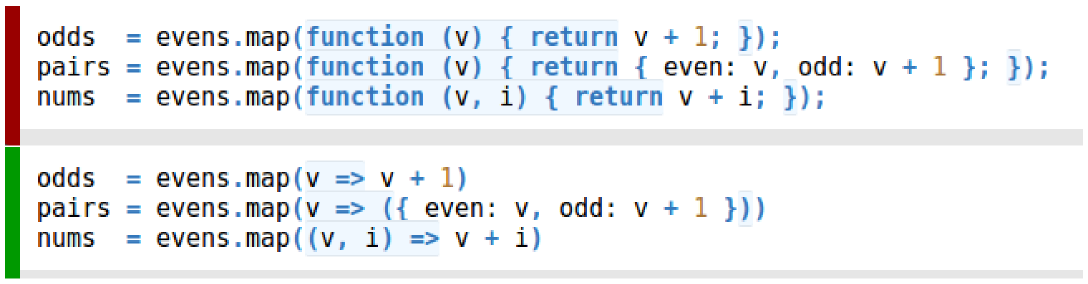
Prior to ES2015, it was a best practice to hoist variables at the top of the program and function declarations [Cro08]. The new standard introduced block scoping for two new types of variables, *let* and *const*. Variables of these types belong to the surrounding block, which is limited by curly braces independently of the construction used [Sim14].

In order to retrieve all the constructions which create new scopes *escope* [escb] was used. The usage of this tool allows the gathering of an understanding about the scope organization of the code and which constructions are used to create it.

3.2.2 ECMAScript2015

In its current state, JavaScript is a continuously evolving language. The creation of *node.js* [nodd] made possible using an event-based language to build scalable applications without the troubles arisen by multithreading [JI12]. The current JavaScript standard in vigor is the ECMAScript2015 [ecm]. This standard introduces new features such as the *for of* loop or *arrow functions* and polishes already existing functionalities such as the importing of frameworks or libraries.

Since the standard is new, some browsers and the *node.js* ecosystem have still to implement all the changes it imposes to the language. In order to start producing code which is compliant with the standard, a tool (transpiler) needs to be used to pre-compile the code into something the environment where it is executed can understand. The tool of choice used to perform this task was *babel* [bab]. Figure 3.2 presents an example of functions written in the standard previous to ES2015 and the arrow functions feature of ES2015.



```
odds  = evens.map(function (v) { return v + 1; });
pairs = evens.map(function (v) { return { even: v, odd: v + 1 }; });
nums  = evens.map(function (v, i) { return v + i; });
```

```
odds  = evens.map(v => v + 1)
pairs = evens.map(v => ({ even: v, odd: v + 1 }))
nums  = evens.map((v, i) => v + i)
```

Figure 3.2: Transpilation example of functions from their regular form to the ES2015 arrow function syntax. Taken from [tra].

The relevance of the new JavaScript standard in the project has two main factors. Firstly, since *Jscrambler* mainly works in a JavaScript environment and provides services upon JavaScript code, it is important to adopt, as soon as possible, the new standard, in order to be proficient in the language as it evolves. And secondly because one of the main objectives of the dissertation is the collection of metadata in order to help guiding *Jscrambler's* software. As will soon be discussed, there are situations where it is relevant to distinguish to which standard some type of data applies.

3.2.3 Architecture

The *raven-analyser* module, responsible for the collection of metadata from code resources, consists of the following four key components, all of which were developed within the dissertation period. The main tools utilized were *escomplex* for software metric retrieval and *escope* for listing the existing scopes.

Examiner

The *examiner* is one of the main components of the *raven-analyser* module. This component is responsible for traversing the AST (or ASTs when processing multiple files) of a code file and extract the following information:

- Tokens: Tokens are considered to be literals (JavaScript native types such as boolean, number, string and regular expressions) and operators (those being of different types such as unary (+), logical (&&), binary, assignment and update.
- Objects: With the help of the Native Objects component, presented further ahead in this section, the *examiner* indexes all the objects constructed, categorizing them by native JavaScript objects, ES2015 objects (objects introduced in the standard) and custom objects. Custom objects are objects of any type which do not belong to the native environment or to the new standard and their origin can either be from the source code or from modules which it makes use of.
- Native Object Methods: Besides identifying native objects, native object methods are also identified. These are the methods which are present in any JavaScript environment since they are part of the native language. Examples are *String.substring()* and *Array.indexOf()*.
- NonTerminal Nodes: After investigating the output of the parsers such as *espree* [espa] and *esprima* [espb] (as presented in Listing 3.2), one can identify which nodes are terminal in the AST (in the case of these parsers *Literal* and *Identifier* nodes are the terminal nodes). All the other nodes represent constructions inherent to the programming language and it can be useful to the *Jscrambler* to compare the frequency of each construction and how they evolved. A distinction is made between constructions in the ES2015 standard and the previous ones.
- Scopes: A JSON object compiling a tree like object of all the existing scopes in the code file is created with the help of the *escope* module [escb].
- Imports/Exports: JavaScript code files can either be a script or a module. Generally scripts are run in a browser environment and modules in the *node.js* environment. JavaScript modules are files usually ran in *node.js* which import libraries and resources and can export classes, objects or functions. In order to analyze which frameworks or libraries some code utilizes it is necessary to compile the imports and exports. The *examiner* compiles an array of imports and exports, their formats (because ES2015 introduced a new way of importing/-exporting resources), and if the resource imported is a *node.js* native library.
- Comments with annotations: Instead of retrieving all the comments existent in the code, which could comprise unnecessary information, only comments with annotations are retrieved. A parser is used to break down, into an organized JSON object, each comment block and the annotations it possesses.

JavaScript Standard Objects

As stated in the previous subsection, the algorithm identifies objects and methods which are native to JavaScript. In order to do so a module was created which organizes all this information. The module contains a list of objects and their native methods, separated by specification, which allows identifying the usage of properties and methods of the native JavaScript implementation. All the information was compiled attending to *Mozzila's Developer Network* [jso] and needs to be updated as the language evolves.

One important aspect to note is that some JavaScript native objects share properties (*prototype* and *length* are common shared properties) which is the effect of inheritance between object types. For example, the *length* property, by default, is inherited from the native implementation of the JavaScript *Object* type. With this in mind, when accessing the property *length* of an *Array* or *String*, the algorithm will identify the property *length* of an *Object* being used, instead of *Array* or *String*, in this case.

The property to object matching could be improved by performing some kind of type inference upon finding statements which operate on certain objects, such as a new object creation or an assignment operation. This possibility was not further explored since it requires a significant amount of time and resources to perform, with no guarantee of it being completely sound, as static type inference in JavaScript needs to cover a great amount of cases [HG12].

HTML Processor

For the algorithm to be able to analyse HTML files there is a need of extracting JavaScript code from them. Usually, JavaScript code is present in HTML in two distinct forms. The first and obvious one is inside *script* tags such as *<script>var a = 1;</script>*. The second is inside some HTML attributes in the form of events (for example, *<button onclick="modifyText()">Click me</button>*).

The functions of the HTML Processor module are:

- Extract code from event scripts: For each HTML element, the processor searches the element's attributes for events (attributes which start by *on* and are followed by the event name, such as *onclick*). Then, it takes the code and wraps a function around it so that the statements inside the events would not interfere with each other. This wrapping is necessary because, when the code is being analysed, the events do not execute in the same scope because if two events would employ the *return* key word, were they not wrapped inside functions, the AST parser would yield an error.

- Extract code from script elements: For each *script* element, the contents are extracted and combined without any modification.

- Extract the *src* contents of script elements: For each *script* element, if it possesses the *src* attribute, the contents of the attribute are gathered to power the framework identification previously mentioned.

Analyser

The final main component is the analyser, which binds everything together and is responsible for the whole process of analysing a project. The workflow of the *analyser* is as follows:

1. A recursive directory read compiles all the files existing in the directory, ignoring all of which are not JavaScript nor HTML files.

2. For each file retrieved the analyser reads the file, retrieving the HTML *src* elements with *url* attributes, if existent, and creates an AST from the file contents.

3. Before moving to the actual analysis, all the *url src* attributes are grouped because, as previously mentioned, one can not know for sure where certain framework was imported in regards to JavaScript script files.

4. Having assembled all the ASTs, the *escomplex* [esca] analysis is called alongside with the *examiner* analysis. At the same time, the *package.json* files, if existent, are parsed and the dependencies retrieved.

5. The final results of each analysis and collection are grouped in a JSON object and returned.

3.2.4 Library Identification

One of the *examiner* module's functions it to identify libraries in use by the code being analysed.

For JavaScript module files it is easy to identify which modules are imported since one only needs to check which libraries are required, usually at the top of the file. When analysing JavaScript script files, the ones which are run in a browser environment, statically identifying the libraries in use can be a challenging task. If the files were to be executed (i.e. dynamically analyzed), one would only need to check which components existed in the *window* object, which represents the browser window, and where all global objects, functions and variables lie [jsw]. Since the *examiner* is limited to static analysis, the following were the steps taken in order to identify frameworks used in JavaScript script code files:

Firstly, when processing either a directory or a single file, the solution compiles all the *src* attributes from the *script* tags existing in HTML files. This is necessary because, unlike in *node.js*, the inclusion of a library can be done in a single file which can propagate to all the code executing in the browser environment.

Secondly, when the *examiner* processes the AST, whenever a new object is created, a member expression (for example *Object.property*) or a literal are employed, the algorithm tries to match a substring of each HTML *src* to the name in use. This matching is necessary since many script elements contain a *src* attribute which is a link to a resource and not just the name as in a *node.js* import. A practical example of this situation is the case when we have a *src* tag like this one *<script src="https://site.com/ajax/15.1.0/react.js"></script>* and a construction in the code like *ReactDOM.render()*. The algorithm would match these constructions as using the *react* framework [rea], based on the fact that the member expression *ReactDOM.render()* uses the word "react" as well as the *src* attribute in the HTML script tag.

This type of identification can yield false positives since there can be a construction which is aligned with some other part of the *url* in the *src* attribute. The website part of the *url* is removed by the algorithm, so the *url* displayed in the previous paragraph would become just "ajax/15.1.0/react.js". Taking this in account, if a construction was found with the name *ajax* then the algorithm would identify the code as utilizing *ajax*, which in that case would be a false positive. Also, some libraries do not use constructions with the library's name but rather some other identifier (such as *$* for many DOM utility libraries). For this type of scenario, framework indicators need to be previously compiled before performing the analysis (as in names, identifiers or constructions the library uses).

In addition to the compilation of the *src* attributes and the imports found, the algorithm searches the project directory for any *package.json* files. These type of files are usually present in *node.js* projects and compile a list of dependencies (frameworks/libraries) on which the project depends on to properly function.

3.3 *Raven-interface*

The *raven-interface* is the module responsible for combining the *raven-analyser* with the infrastructure necessary to process the *Jscrambler's* data and saving the analysis results. All of the *raven-interface* components were developed within the dissertation period.

3.3.1 Core Technologies

The main JavaScript libraries which the *raven-interface* uses are the *bluebird promise* library [blu], described in the previous section, and the two following database libraries:

Mongoose

In order to facilitate the interactions with the *MongoDB* database [mona], *mongoose* [monb] was used. *Mongoose* allows the management of database connections, the creation of *schemas* (which are the outline of the documents saved on the database) and a series of functions for inserting, updating and deleting documents.

Bookshelf

Providing the same utilities as *mongoose*, but to retrieve data from the company's *PostgreSQL* database [pos] and working upon relational data, not documents, *bookshelf* [boo] was used. Besides the features indicated as being part of *mongoose*, *bookshelf* comes packaged with a query builder which gives more power to the user when working with stand-alone queries.

3.3.2 Architecture

The *raven-interface* is a command-line interface application with two distinct modes of action.

The first mode is the regular one, where no company data needs to be retrieved and the operations being invoked are only the *raven-analyser* analysis and the insertion of the result to the *MongoDB* database.

The second mode attends to the structure of the *Jscrambler's* database and resources and executes a series of operations before calling the *raven-analyser* analysis.

The following is the workflow of the algorithm when analysing a directory which contain the projects of a *Jscrambler* user.

1. Concurrent query to the PostgreSQL database retrieving the user information alongside with reading the directory with the user's projects.

2. Since *node.js*' functions which read directories only return relative paths there is a processing step which computes absolute paths.

3. For each user project, in sequence, and according to the executing parameters, call the *raven-analyser* analysis on either the original code or the obfuscated version and save the result of the analysis to the *MongoDB* database with the user data and the transformations executed to transform the code if the folder processed was the obfuscated one.

A visual representation of the whole flow of *raven-interface*, also including the flow of the *raven-analyser*, is presented on Figure 3.3.

3.4 Development Process

The development process was an iterative one, as it usually is for any software development project where the software being developed is non-critical.

The first two to four weeks of development were dedicated to the investigation of the tools being used and to learn more about JavaScript and its characteristics. The learning process of JavaScript was motivated by the fact that, besides the peculiarities of JavaScript, stated in Chapter 2 and in the above sections, there are many others which confuse developers, especially those coming from a object-oriented (OO) background. To address this knowledge deficiency, the learning process was based upon Kyle Simpson's *You Don't Know JavaScript* book series [Sim14] [S+15]

[S+14] [Sim15]. This book series provided not only a thorough understanding about good and bad practices in JavaScript and which characteristics should be approached carefully (such as an use of the *this* keyword as in OO languages) but also shed some light in aspects such as JavaScript native types and *promises* which were of great use in the project.

The following stages of development were dedicated to creating a solution prototype which would calculate the software metrics, via *escomplex* [esca], and save the results to the *MongoDB* database [mona]. Once the prototype was finished the requirements for the *examiner* component were elicited. Some time was dedicated to the study of the ES2015 standard and to the refactor of the code from *callbacks* to *promise* based.

Once the *examiner* was coming to its final stages of development, the decision of separating the analysis module and creating a separate project to interact with the databases and *Jscrambler* data was made which caused another refactor of the codebase.

The last requirement to be fulfilled in regards to the *examiner* was the identification of libraries/frameworks, since the objectives of the literature reviewed, when considering the analysis of JavaScript with the use of frameworks and libraries, did not completely align with the objectives of this dissertation.

After this phase, it was time to develop the *raven-interface*. This part of the development process was relatively quick when comparing to the *raven-analyser* since the interactions with the *MongoDB* database were already made from the first prototype of the solution. The module suffered from several versions since the first version had problems regarding concurrency (discussed in the following section) but it eventually came to a point where a sequential directory analysis would allow to process *Jscrambler's* data.

At this stage the solution was close to final and there were no significant development stages beyond it, only correction of bugs and addition of minor features caused by running the tool against real company data.

3.5 Overcoming Challenges

The following were the main challenges faced in the development process:

The first real challenge felt when developing the solution was the usage of *promises*. *Promises* are a concept that is not so easy to grasp and can pose a serious challenge when not presented with the right information [Sim15]. Many *promise* tutorials fail to go further beyond the basic concept which is that *promises* are a good alternative to *callbacks* and can be used in succession/are composable. Besides this, *promise* chains always need to return some value for the chain not to be broken and are not easy to debug. If an argument is missing the whole program can silently fail and the *catch* clause will not throw an error because no error was thrown along the chain.

After extensive reading and discussing the subject of *promises* with *Jscrambler's* employees, a better understanding of the concept was grasped which culminated in a much cleaner, understandable and better performing software.

In the midst of testing and working with *promises*, the solution presented was not in a satisfactory state as it would concurrently analyse multiple directories causing for the algorithm to quickly run out of memory. To avoid this issue a series of refactoring sessions were done in order to ensure that the solution would analyse each project sequentially, minimizing the possibilities of running out of memory. The process was complicated as it once again asked for further investigation about *promises*. The challenge was overcome when realizing that *promise* code begins executing the moment the *promise* is created and not only when reaching a piece of code which awaits for the *promise* completion. This means that when an array of *promises* is constructed, the asynchronous code is already being executed, so looping the array, in sequence, waiting for the *promise* completion, does not enforce the synchronous execution of the promises. The *promises* need to be created and handled inside a sequential construction to ensure synchronous execution.

The final major challenged faced was to process all of *Jscrambler's* data. The company possesses a large amount of data which is impossible to copy/store on a single PC. For each user, data needs to be retrieved across the network, saved, analysed and then disposed of. Running the program from the command line would pose a problem since non-core operating system functions would be used to copy files around and also the program would fail when any directory failed to be processed. This challenge was overcome by utilizing a bash script, made to handle this process, which to ensures any processing error would not disallow the remaining directories to be handled.

3.6 Summary

This chapter presented a general and an in depth view of the main components of the Raven solution those being the *raven-analyser* and the *raven-interface*. The *raven-analyser* is the analysis module responsible for calculating the software metrics relative to the code being inspected, via the *escomplex* library [esca], and to retrieve the information deemed useful by *Jscrambler*, made possible by the *examiner* component. To make sure everything interacted smoothly with *Jscrambler's* data, another module, the *raven-interface* was created, which is responsible for interacting with the databases and to process all the company data.

The development process was an iterative one where the main steps were a dedicated learning phase, emphasizing on JavaScript, a prototype creation alongside the elicitation of the requirements for the *examiner* component, the coming together of the *raven-analyser* and the many iterations to reach a final version of the *raven-interface*.

Many challenges were faced when developing the solution but the main ones were concerning *promises*, refactoring processes and processing large quantities of data.

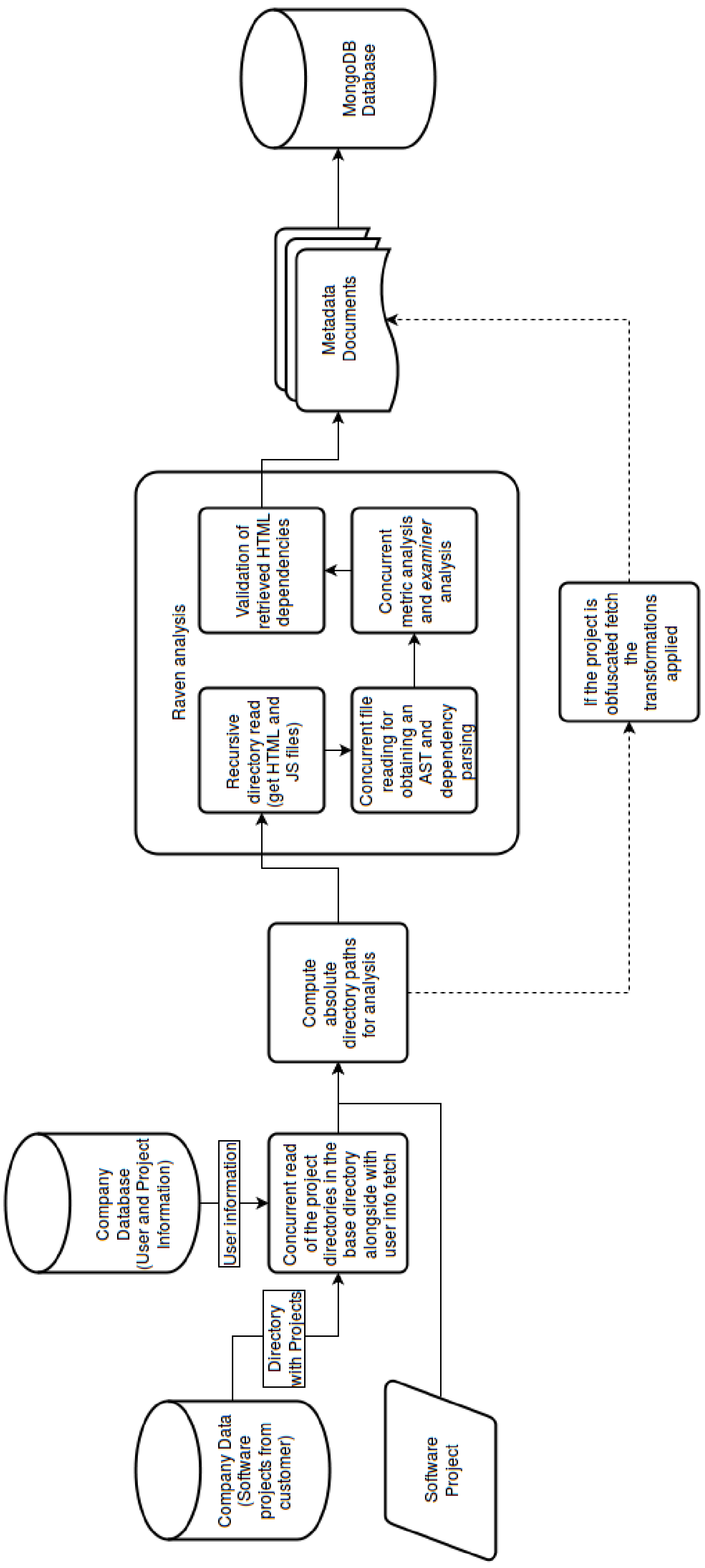

Company Data (Software projects from customer)
Directory with Projects
Company Database (User and Project Information)
User information
Concurrent read of the project directories in the base directory alongside with user info fetch
Software Project
Compute absolute directory paths for analysis
Raven analysis
Recursive directory read (get HTML and JS files)
Concurrent file reading for obtaining an AST and dependency parsing
Concurrent metric analysis and *examiner* analysis
Validation of retrieved HTML dependencies
If the project is obfuscated fetch the transformations applied
Metadata Documents
MongoDB Database

Figure 3.3: Workflow of the Raven solution.

Chapter 4

Experimental Results

In regards to the dissertation's objectives, the performed experiments, detailed in this chapter, looked to address the following research questions:

1. How does the retrieved metadata behave when analysing code files in their original state and after performing *Jscrambler's* services? Are there any software metrics which endure through the changes made to the code?

2. Taking *Jscrambler's* data into account, when comparing the metadata of projects which use similar frameworks or libraries, can conclusions be made in regards to the average complexity of a project that uses a certain a library, by analysing the metrics retrieved? That is, can it be said that a project using certain library is, for example, two times more complex than another using other library for the same purpose?

The following sections present the experiments realized in order to address the research questions and a discussion of the results obtained. First, an explanation on how the results were gathered is presented followed by the results themselves. The consecutive section focus on discussing the results, guided by the research questions.

4.1 Gathering results

4.1.1 Picking the dataset

To test the metrics resiliency to the services *Jscrambler* offers, a controlled dataset was compiled, in order to study the impact of different operations on the same set of data. This dataset included four JavaScript libraries for front-end development, four of the most popular *npm* packages [1] and a JavaScript benchmark suite.

[1] https://www.npmjs.com/browse/star

The front-end libraries chosen are usually executed in a browser environment and provide DOM (document object model) manipulation functions, animations and facilitate event-based scripting. The libraries used were *jquery* [jqu], *dojo* [doj], *mootools* [moo] and *prototype* [pro]. *Jquery* and *dojo* were analysed considering the production version of the code (minified) and the development one.

The *npm* packages utilised were *express* [expa], *cheerio* [che], *commander* [com] and *q* [q]. *Express* is a framework for building JavaScript applications and is usually used to build APIs (Application Programming Interfaces) or applications based on the MVC (Model View Controller) pattern. *Cheerio* is an implementation of the features delivered by *jquery* for the server environment (i.e. for back-end development instead of front-end). *Commander* is a library for developing command-line interfaces. And finally, *Q* is a promise library with features closely related to *bluebird*, as described in Section 3.2.

The benchmark used was the new form of Google's V8 benchmark suite [goo] *Octane*.

4.1.2 Gathering controlled results

For each test set, the following transformations were applied. These transformation originate from a set of predefined templates, available in the *Jscrambler's* web application.

The templates used define a set of specific transformations and each transformation could be applied in a stand-alone fashion. The following were the templates applied:

- *Obfuscation* - Applies the following transformations: *remove comments*, *whitespace removal*, *rename local variables*, *duplicate literals*, *function reordering*, *dot notation transformations* and *function outlining*.

- *Self-defending* - Applies the following transformations: *remove comments* and *Jscrambler's* proprietary *self-defending* transformation.

- *Minification* - Applies the following transformations: *remove comments*, *whitespace removal* and *rename local variables*

- *Compression* - Applies all the transformations of the *minifcation* template as well as a *dictionary compression* transformation.

It is important to note that there are many obfuscation algorithms available for JavaScript besides the one *Jscrambler* provides. The goal of the software metric comparison is to address *Jscrambler's* obfuscation process and not others available.

The *compression* and *minification* datasets were submitted to the *Jscrambler's* website [2] and then downloaded. The *self-defending* and *obfuscation* datasets were retrieved by using the company's API. The transformation processes used *Jscrambler's* software in version 3.8.

The following is an explanation of the results of each transformation. For a more in depth overview on the results of each transformation with code examples consult Appendix A or *JSCrambler's* documentation [3].

- *Remove comments*: Removes all the comments for the source code.
- *Whitespace removal*: Removes all the *whitespace* and *newline* characters from the source code.
- *Rename local variables*: Replaces all identifiers which are not in the *global* scope of the code for ones without any meaning. Identifiers which are not in the global scope are the ones inside function declarations and catch blocks of code.
- *Duplicate literals*: Replaces all duplicate literals used in the code, such as static strings, for identifiers which are used to replace the duplicate usage of the literal.
- *Function reordering*: Reorders all the function in the source code, in no particular order, but considering the declaration hoisting, that is, no function shall be used before it is declared.
- *Dot notation transformation*: Transforms all the usages of the *dot notation* into the *array subscript notation*.
- *Function outlining*: Transforms statements into new function declarations. These new functions wrap around the statements altering the flow of the code, difficulting its comprehensiveness.
- *Self-defending*: Obfuscates functions and objects by concealing their logic and frustrates attempts of debugging the code. Proprietary to the company so no further information is provided on the documentation page.
- *Compression*: Performs a lossless compression by applying the LZ77 algorithm [WP02]. The algorithm allows the compression of data by replacing repeated occurrences of data with a reference for its previous occurrence in the data stream.

Table 4.1 presents the size of each element of the dataset in its original form and after each transformation.

[2] `https://jscrambler.com`
[3] `https://docs.jscrambler.com/transformations`

Table 4.1: Controlled dataset sizes by transformation applied. Sizes are in kB.

Element	**Original**	**Obfuscation**	**Self-defending**	**Minification**	**Compression**
dojo 1.11.2 prod	164.8	200.9	367.7	135.0	98.5
dojo 1.11.2 dev	630.6	200.9	416.8	134.9	98.6
jquery 3.0.0 prod	86.3	119.4	198.2	87.7	64.8
jquery 3.0.0 dev	263.3	131.4	297.3	92.4	67.3
moo tools 1.6.0	162.0	143.5	727.0	93.2	62.0
prototype 1.7.3	199.8	159.4	794.0	103.2	67.0
cheerio 0.20.0	111.6	97.6	905.1	85.7	79.3
commander 2.9.0	44.8	34.7	378.7	29.7	26.6
express 4.14.0	207.1	170.3	1100	153.6	147.3
q 1.4.1	123.6	80.4	129.2	77.1	70.7
octane-master	10500	14100	39500	8200	3700

After this process, the *raven-analysis* was run for each transformed dataset and for the original version, which originated the results displayed on Tables 4.2 to 4.6, all of which were rounded to two decimal places.

Each of the software metrics present in the tables represent the average value per-function (in the case o the maintainability it is per module, that is, per code file), and not an absolute value. By comparing average values per-function, comparisons can be made between datasets of different sizes. If the comparison relied on absolute values, the results would be largely influenced by the size of the dataset. It is easy to understand that, for a large codebase, a non-normalized count of lines of code (LOC) would be significantly higher than the one of a smaller codebase.

To facilitate the comparison between the different transformations, Figures 4.1 to 4.4 present column charts of each metric, contemplating original values and all the transformations, for *express*, *Q* and the development version of *jquery*.

4.1.3 Gathering company data

A total of 54,227 projects were analysed. These projects are stored by the company and originated from their API client. The main objective of this data gathering process, in terms of addressing the dissertation objectives, is to identify frameworks/libraries in use and possibly withdraw conclusions from the software metrics of the projects using them.

2,978 projects identified depended on *node.js* [nodd] libraries. Since only the newer version of the company's software deals with code compliant with the new standard (ECMAScript2015 or ES2015), it is safe to assume that all these projects were built to perform in the *node.js* environment. This is because ES2015 will allow the usage of *import* and *export* keywords in the web environment [S$^+$15] and is by analysing the *imports* that *raven* identifies the libraries in use.

Of all the 2,978 projects, 267 distinct libraries were identified. This value is apparently low as one would expect more diversity out of almost 3,000 projects. The cause of this lack of diversity

Table 4.2: Metadata results for the original versions of the dataset.

Element	**Logical LOC**	**Cyclomatic Complexity**	**Halstead's effort**	**Maint. Index**
dojo 1.11.2 prod	6.98	2.90	3952.20	110.95
dojo 1.11.2 dev	6.99	2.90	4099.27	110.82
jquery 3.0.0 prod	4.92	3.09	5848.37	115.27
jquery 3.0.0 dev	7.84	3.60	5863.63	107.67
moo tools 1.6.0	6.21	2.59	2961.43	113.87
prototype 1.7.3	6.54	2.71	3002.51	112.96
cheerio 0.20.0	6.02	2.81	3214.66	118.29
commander 2.9.0	6.73	2.89	3409.28	112.04
express 4.14.0	6.60	2.58	2445.36	116.65
q 1.4.1	4.14	1.36	1102.08	124.40
octane-master	19.72	2.20	19460.88	111.75

is the fact that a user can submit the same projects, with minor alterations, several times, resulting in an inflation of the number of projects and a low diversity of libraries used. This means that, when considering the comparison between the metrics of the projects, similar metrics of a series of projects of the same user, which use the same libraries, need to be normalized, in order to count as a single project. The software metrics from projects from the same user, submitted in the same day, were normalized and considered to be a single set of metrics.

Another concern, when regarding the comparison of libraries, is the fact that the comparison must be made, not only between libraries which are used for the same purpose, but also with libraries meaningful enough to affect the code's metrics. For example, a JavaScript application framework, such as *express* [expa], has a much more meaningful impact in the code's structure and organization, because it enforces a certain type of constructions to be used, which are then translated in the value of the metrics, than a library such as the *node.js* filesystem library, which is normally used only at specific times in the program and has, in general, less impact on the code's structure. Also, most software projects have multiple dependencies, which means that the metrics retrieved are not influenced by a single dependency but rather by their aggregation, which strengthens the argument of the library needing to be meaningful enough.

With these concerns in mind, and after the revision of the *node.js* libraries found in the analysed projects, the following list of libraries susceptible to be compared was compiled. This list attends to the fact that the libraries used need to provide a lot of functionality in order to influence the code's metrics:

- Frameworks to model the *view* component of an application: *react* [rea] and *backbone* [bac].
- Login/registration libraries: *passport* [pas] and *express-session* [expb].
- Relational databases: *mysql* [noda] and *sqlite3* [nodc].
- Non-relational database tools: *mongoose* [monb] and *nedb* [ned].

Table 4.3: Metadata results for the obfuscated versions of the dataset.

Element	LogicalLOC	Cyclomatic Complexity	Halstead's effort	Maint. Index
dojo 1.11.2 prod	4.33	1.81	1842.96	121.42
dojo 1.11.2 dev	4.33	1.81	1841.33	121.39
jquery 3.0.0 prod	4.06	2.00	3051.44	120.69
jquery 3.0.0 dev	4.60	1.93	2093.92	119.96
moo tools 1.6.0	4.68	1.79	2075.58	119.73
prototype 1.7.3	4.51	1.80	1868.57	120.69
cheerio 0.20.0	5.52	1.93	2021.34	118.66
commander 2.9.0	4.95	1.93	2614.95	118.03
express 4.14.0	6.00	1.80	1804.48	117.68
q 1.4.1	3.99	1.23	757.35	126.24
octane-master	8.57	1.36	3850.97	120.45

- Asynchronous code handling: *q* [q] and *async* [asy].
- Templating: *handlebars* [han], *mustache* [mus] and *swig* [swi].
- General frameworks/application: *soap* [nodb], *express* [expa] and *connect* [con].

The analysis result considering each group of libraries is displayed on Table 4.7. The total and normalized numbers of projects are displayed although the metrics following only correspond to the normalized results.

4.2 Discussion

4.2.1 Controlled dataset

The following section presents the discussion of the results obtained for the controlled dataset.

Data sizes

It is important, before discussing the transformations applied, the effects of the transformations applied on the size of the projects (Table 4.1):

In terms of *obfuscation*, all the front-end libraries yielded smaller sizes for the obfuscation transformation except the production versions of *jquery* [jqu] and *dojo* [doj]. Most of the *npm* packages did not suffer a significant impact, except for *Q* [q] and the *octane* [goo] benchmarks was heavily affected by the transformation.

This type of behaviour is as expected because, although the *duplicate literals*, *dot notation* and *function outlining* transformations generate more code than the one previously existent, comments and whitespaces are removed. The outlying of the production versions of *jquery* and *dojo* can be explained by the fact that they are minified and do not contain comments or whitespaces, so their

Table 4.4: Metadata results for the self-defended versions of the dataset.

Element	Logical LOC	Cyclomatic Complexity	Halstead's effort	Maint. Index
dojo 1.11.2 prod	9.72	4.05	28145.40	98.79
dojo 1.11.2 dev	10.89	4.52	30709.93	96.63
jquery 3.0.0 prod	10.89	4.52	31078.67	96.59
jquery 3.0.0 dev	10.89	4.52	30226.58	96.68
moo tools 1.6.0	9.54	3.97	28171.43	99.10
prototype 1.7.3	9.41	3.92	27951.50	99.35
cheerio 0.20.0	9.07	3.79	25886.58	102.95
commander 2.9.0	9.27	3.86	27633.02	99.65
express 4.14.0	9.16	3.83	26276.71	102.70
q 1.4.1	10.89	4.53	30840.42	96.61
octane-master	9.19	3.86	26913.35	102.00

size increased. Regarding the *octane* benchmarks, it was predictable that the size would increase since the code being tested is used to test performance and not for development purposes, therefore not containing a large amount of comments and more constructions susceptible to be bloated.

Upon inspection of the self-defended code files it is observed that there are a series of *eval* statements being used on a very large string. Since the algorithm being used to encode the code into the strings is proprietary, it is hard to predict the result, in terms of size, of applying the transformation, so sound conclusions can not be made as why some data samples increased more than others. This being said, all the samples displayed an increase in size. The least amount of increase was of observed on the *jquery* development version, which was of about 30 kB. As expected, the largest increase was of the *octane* benchmarks.

As expected, all minified versions of the dataset were smaller than the original except for the *jquery* production library, which was already minified. *Jscrambler's* process of variable renaming probably generated literals of bigger size than the process to minify the original version of *jquery*.

All data samples presented a smaller size value when compressed, as expected.

Original metrics

Table 4.2 presents the metrics gathered from the original versions of the data samples.

It can be verified that for the *front-end* libraries and *npm* projects, the values of the logical LOC is between 4 and 8, which indicates that on average, each function executes 6 statements. Having modular functions is a JavaScript best practice [Cro08], which means each function is responsible for executing only one task, so a low average value was expected from production grade tools. For *octane* [goo] a higher logical LOC count was expected, since, as already stated, it focus on heavily testing JavaScript.

The cyclomatic complexity (CC) lies between 1 and 4 for all data samples, which indicates that each function has, on average, between 2 or 3 control flow paths, which again is a strong indicative of modular code.

Table 4.5: Metadata results for the minified versions of the dataset.

Element	Logical LOC	Cyclomatic Complexity	Halstead's effort	Maint. Index
dojo 1.11.2 prod	6.98	2.90	3893.52	111.00
dojo 1.11.2 dev	6.99	2.90	3894.38	111.00
jquery 3.0.0 prod	4.92	3.09	5848.37	115.27
jquery 3.0.0 dev	7.84	3.60	5689.04	107.77
moo tools 1.6.0	6.21	2.59	2845.18	114.00
prototype 1.7.3	6.54	2.71	2832.61	113.16
cheerio 0.20.0	6.02	2.81	3109.26	118.38
commander 2.9.0	6.73	2.89	3320.91	112.13
express 4.14.0	6.60	2.58	2314.00	116.83
q 1.4.1	4.14	1.36	997.98	124.67
octane-master	19.72	2.20	19214.94	111.83

Halstead's effort (HE) (2.10) behaviour is more difficult to predict as it relies on a number of different factors. The main contributors for a bigger HE value are the number of distinct operators, number of distinct operands and total number of operators. These counts are not only affected by the purpose of the library but also the code style and programming standards.

The *Q* [q] package, by providing abstractions to deal with asynchronous code and not having to perform many computations, presents a lower HE value than most of the other data samples. *Octane*, as expected, presents a much larger average HE per function than any other sample. It is also important to notice that there is a minor difference between the HE value between the production and development versions of *jquery* and *dojo*. Since HE is based on the counts of operators, and since minification does not affect its number, they present similar values.

The maintainability index (MI) (Equation 2.11) ranges between 107 and 125 for the whole dataset. Since the MI is mostly affected by the LOC count, the similarity between the LOC values proves to influence the MI values.

Obfuscation

Upon comparing the original metrics (Table 4.2) with the obfuscated ones (Table 4.3), the following is observed:

- The logical LOC count is on average 2 lines less for each data sample.
- The average CC now lies close to 2 instead of 3.
- The HE is, on average, half the original value.
- The MI of each sample increased by 8 points on average.

There was no single metric which remained unchanged after the combination of transformations which comprise the obfuscation process.

Table 4.6: Metadata results for the compressed versions of the dataset.

Element	Logical LOC	Cyclomatic Complexity	Halstead's effort	Maint. Index
dojo 1.11.2 prod	11.5	3.5	7148.07	100.79
dojo 1.11.2 dev	11.5	3.5	7148.07	100.79
jquery 3.0.0 prod	11.5	3.5	7148.07	100.79
jquery 3.0.0 dev	11.5	3.5	7148.07	100.79
moo tools 1.6.0	11.5	3.5	7148.07	100.79
prototype 1.7.3	11.5	3.5	7148.07	100.79
cheerio 0.20.0	8.51	2.98	5011.33	112.14
commander 2.9.0	11.5	3.5	7148.07	100.79
express 4.14.0	9.71	3.13	5535.94	107.86
q 1.4.1	8.05	2.35	3878.64	112.55
octane-master	11.34	3.46	7039.78	101.13

The decrease of the logical LOC count and the average CC can be explained by the *function outlining* transformation which, by encapsulating statements inside functions, reduces the mean value of both the logical LOC and CC.

The decrease of HE is explained by two of the transformations. First, the *duplicate literals* transformation attributes a variable for each literal which is repeated in the code. This decreases the total number of operands, which is a main contributor for an increase in effort (2.10). And second, the fact that the *dot notation* transformation is applied, creates a number of variables in order to mask the access to a property. By creating new variables, the number of distinct operands increases. This increase has an inverse effect on the HE value.

It is possible to conclude that the *duplicate literal* transformation had a great impact on the *octane* benchmarks since the HE value was reduced by about 15000 units. This means that the benchmarks probably have the same literal being retyped frequently.

As stated before, the major factor of the MI index is the LOC count. By decreasing the LOC count but also the program volume (2.5), due to the *duplicate literal* transformation, the MI was increased on average.

Self-defense

The results displayed on Table 4.4 are respective to the *self-defending* transformation, which, as already stated, is proprietary. This transformation removes comments, so that should also be taken into consideration.

Upon observing the *self-defended* code it is noted that the code relies on a series of *eval* constructions, inside loops, executed on very large strings, which are presumably encoding the contents of the code. This pattern is reproduced across each file which means that the metrics results gathered are from the structure used to reproduce the code and not the code itself. This ultimately leads to a very high degree of similarity between the metrics, which is observed. The only metric which could vary from this is HE, since the string which encodes the code could

Table 4.7: Metadata results for the set of libraries identified in the *Jscrambler's* dataset. The total and normalized numbers of projects are displayed although the metrics following only correspond to the normalized results.

Element	Total Projects	Normalized Projects	Logical LOC	Cyclomatic Complexity	Halstead's effort	Maint. Index
react	58	13	23.88	3.23	106788.09	86.86
backbone	35	4	6.13	2.66	8689.60	113.08
passport	107	35	6.01	2.09	3655.69	123.32
express-session	7	2	3.81	1.19	612.68	127.65
mysql	7	2	9.88	2.26	5545.17	110.55
sqlite3	12	1	13.82	3.95	12133.56	96.01
mongoose	2	1	9.85	1.96	2703.95	110.70
nedb	237	33	6.75	2.10	3983.91	111.97
q	12	2	3.65	1.73	389.52	133.47
async	878	85	31.45	2.15	33205.01	116.47
handlebars	6	2	10.08	3.70	42799.73	96.81
mustache	63	2	19.31	2.57	6208.52	108.81
swig	229	31	6.59	2.08	4026.23	111.91
soap	546	17	131.58	2.32	150521.31	111.48
express	755	48	50.97	2.15	55808.96	113.17
connect	1	1	3.56	1.09	407.85	130.24

increase or decrease in size depending on the magnitude of the data sample, but no correlation could be found between the sizes of the data samples and the HE value for the *self-defense*.

Minification

The minification process removes all whitespaces and comments and replaces the names of the variables for meaningless names. As it would be expected, this has no impact in the software metrics, being the impact felt only on the size of the file.

Compression

The compression algorithm presented curious results. Each of the *front-end* libraries used as sample scored exactly the same value on every metric. This is due to the fact the the LZ77 compression algorithm [WP02] creates a vocabulary from the code files. This vocabulary has a fixed size so the compression resulting always yields the same values for the number of operands and operators, which results in the HE being the same. The compressed files present a similar structure to the ones resulting from the *self-defending* transformations, since *eval* statements alongside loops are used to decompress the file contents. Being the structure equal for every file, the metrics also are. The discrepancy viewed in the *npm* projects and the *octane* benchmarks is due to the fact that the compression algorithm has a size threshold in order to operate so very reduced files will not be compressed hence the discrepancy.

4.2.2 Company dataset

As stated in the previous subsection 54,227 projects were analysed. Of those, only 2 was the analysis able to identify libraries present in web applications. On the other hand, 2978 projects had *node.js* libraries identified. Chapter 3 details the framework identification process. For the identification of libraries in applications designed to be executed by a browser, the *raven-analyser* gathers all the included *urls* from *script* elements in the HTML files. These *urls* then serve as the basis for matching the usage of library constructions, by matching keywords.

It is a common practice to use JavaScript outside of HTML files by not having the code inside *script* tags. Being the JavaScript code separated from the HTML files which include it, case that represents the majority of use cases of the company's tool, the *raven-analyser* is unable to identify libraries being used in JavaScript files ran in the browser, resulting in the low number of web applications projects where libraries were identified. A brief paragraph with ideas for improving this process is presented in Section 5.2.

From observing Table 4.7 it is possible to conclude that only the *soap* [nodb] and *express* [expa] frameworks are susceptible to be compared in terms of software metrics. All other groups of libraries do not present a sufficient number of projects analysed in order to soundly gather conclusions in regards to software metrics.

From comparing *express* and *soap* it is not possible to conclude which one enforces higher complexity. Although *express* applications present a significant high degree of logical LOC, this is easily attributed to the fact the applications which rely on express can be big monolithic functions. Their cyclomatic complexity, which could be the main indicator on which is the most complex, is similar. *Express* also presents a high number of the same operands being used, hence the high HE value. But although the high HE value for *express*, both frameworks present the same MI, which is again a strong argument on why the results gathered do not mean one is more complex than the other.

With this being said, in terms of quality, the *soap* framework should be more human readable as it presents a lower degree of sequential logic, as the logical LOC value displays.

4.3 Summary

This chapter firstly presented the process of gathering the analysis results for *raven* to analyse. The first dataset compiled consisted in 9 distinct samples, 4 being *front-end* JavaScript libraries, other 4 *npm* packages and the latter being a JavaScript benchmark suite.

Secondly, the discussion of the results were made. The sizes of the data samples used generally increased for the *obfuscation* and *self-defending* transformations and decreased for the *compression* and *minification* transformations, which were applied by using the *JScrabler's* software.

No software metric endured the *obfuscation* transformation although the obfuscation results obtained were predictable by the conjunction of applied transformations. The *self-defending* and *compression* transformations both rely on encoding the code in a large string and then decoding it using the *eval* statement. These transformations completely mask the code's metric footprint.

Since minification only removes and renames properties of the code, the software metrics gathered remained the same.

The data gathered from analysing *Jscrambler's* projects did not yield sufficient results in order to take significant conclusions from them, mostly because there was not a great amount of data to compare but also because comparisons need to be made between libraries being used for the same purpose.

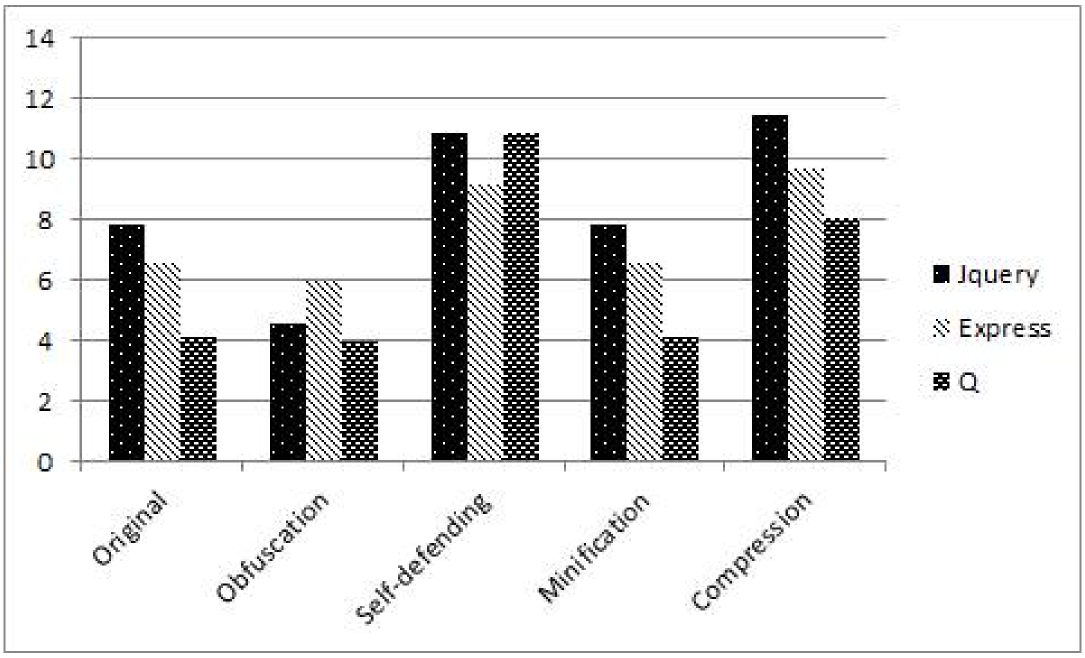


Figure 4.1: Chart representative of the Logical Lines of Code for each transformation of *express*, *Q* and the development version of *jquery*.

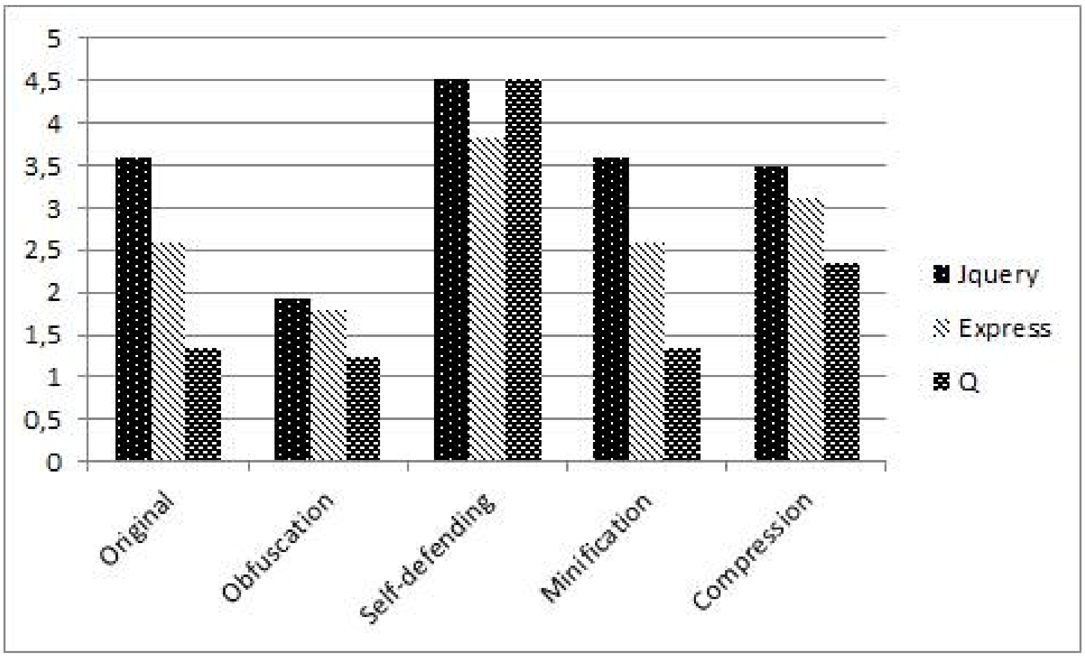


Figure 4.2: Chart representative of the Cyclomatic Complexity for each transformation of *express*, *Q* and the development version of *jquery*.

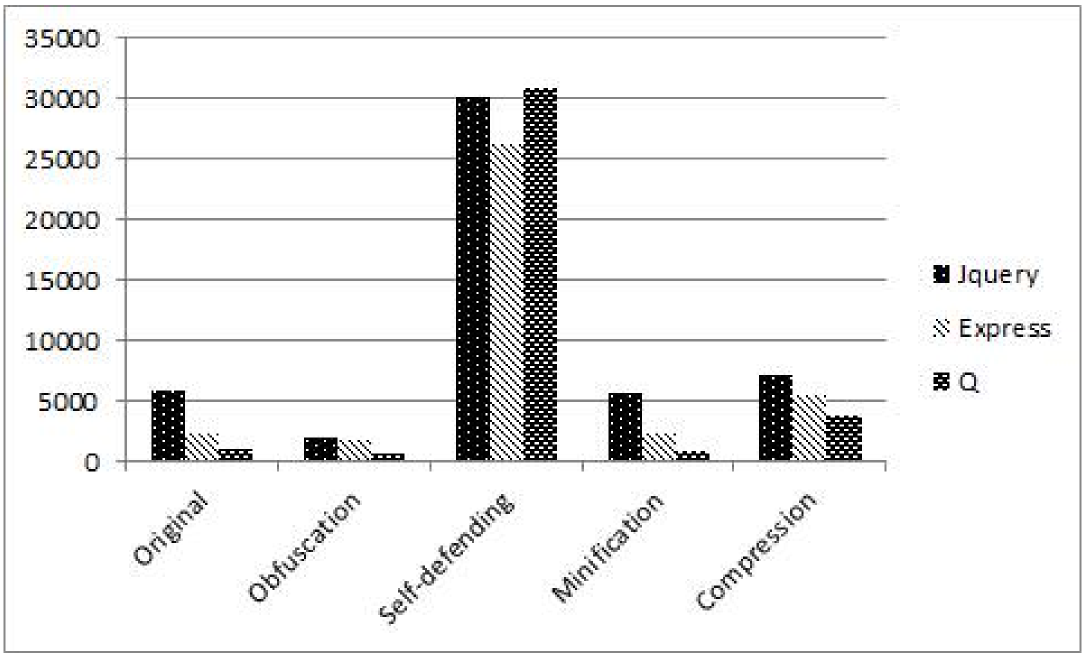


Figure 4.3: Chart representative of Halstead's effort for each transformation of *express*, *Q* and the development version of *jquery*.

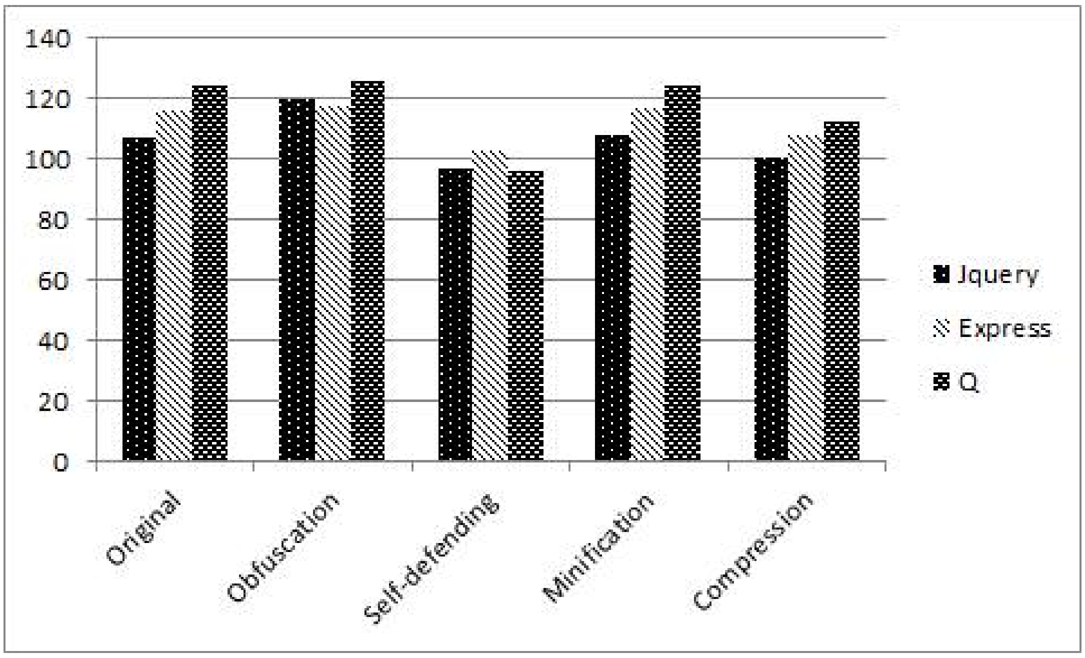


Figure 4.4: Chart representative of the Maintainability Index for each transformation of *express*, *Q* and the development version of *jquery*.

Chapter 5

Conclusion

Static analysis of code resources allows the construction of data structures which represent their form in a way that can be programmatically interpreted. These structures allow the execution of analyses which can yield information about them. The information retrieved can be in the form of software metrics. There is not a single metric that comprises a global understanding on how complex or good (in terms of quality) some piece of code is so one must rely in a set of metrics to properly distinguish code resources and gather an understanding about their aspects [Fen94].

5.1 Contributions

The tool developed in the scope of this dissertation proved to be capable of successfully extracting metadata from several JavaScript applications and compiling information from different sources, thus being an asset for future use by *Jscrambler*.

Having retrieved a set of metrics from a controlled dataset it be can concluded that software metrics retrieved can be a good indicator on how the transformations impact the code and how *Jscrambler's* obfuscation is enough to mask the metric footprint of code resources. Although one can compare the metrics before and after obfuscation and infer what has been performed, the general software metrics do not endure the process. The metric footprints of code resources are completely masked when *Jscrambler's* proprietary *self-defending* or *compression* is applied.

The identification of libraries in a large dataset of JavaScript applications is not an easy task unless the applications are designed for the *node.js* environment [nodd]. Upon analysing a series of different projects and comparing the software metrics of similar frameworks being used, sound conclusions could not be made since the data sample was not big enough and because the software metrics are mostly influenced by the programmer than by the tool being employed.

5.2 Future Work

The *Raven* solution proved to be capable of fulfilling the objectives it was proposed to. As any other type of software there are improvements which, if implemented, would be of great value for the solution. Following are some ideas for future development.

5.2.1 Optimizing storage space

When analysing large projects the size of the metadata retrieved sometimes exceeds the size allowed for a single document in *MongoDB* [mona]. To solve this issue one of two measures can be taken: Increase the storage space, by implementing a *MongoDB* solution using *GridFS* [1], which allows saving large documents since they are chunked and then saved alongside metadata which allows for the reconstruction of the whole document; or reevaluate the importance of some types of data being saved, specially the results of the *examiner* component of the *raven-analyser* (detailed in Chapter 3).

5.2.2 Web Interface

Using database graphical-user-interface (GUI) clients suffices for easily navigating through the analysed results, since most matured database solutions provide one. However, for a user which is unfamiliar with the tool, navigating and organizing the results would benefit from a dedicated user interface, which would automatically connect to the data sources needed and could allow the user the selection of which types of data he wants to retrieve from the code being analysed.

5.2.3 Improving library identification

The library identification process on the subject of statically analysing JavaScript module files (i.e. files executed in the *node.js* environment) relies on identifying the imports made as it is necessary to, in each file, import the frameworks/libraries in use. This process is reliable and could only be improved by checking if the the project being analysed possesses a *pacakage.json* file, which was accomplished.

The process of identifying frameworks for JavaScript script files (i.e. files ran in the browser), in a static fashion, presents some complications and can be improved in several ways. Here are some suggestions which could strengthen the process and make it more reliable:

- Gather framework indicators, through machine learning, which reveal the usage of certain frameworks. The solution would gather data from publicly available repositories and package managers and compile a series of categorized indicators which would then be used for the identification process. For example, when analysing HTML files, if HTML elements

[1] https://docs.mongodb.com/manual/core/gridfs/

were found with attributes named as *ng-**, where * is a set of keywords, and a *script* element included a link containing the word *angular* then it is almost certain that the code being analysed uses *angular*[2].

- Allow for the dynamic analysis of HTML files (this would imply an alteration to *Jscrambler's* terms of service). By executing HTML files one can inspect the *window* variable for libraries in use which would greatly improve the identification process.

- The easiest alternative would be to let the JavaScript community mature. Since the upbringing of *react* [rea] there's been an increased use of *npm* [quo] in the web environment. With the new standard (ECMAScript2015 or ES2015), the import/export notation is to be adopted in the web, which would facilitate the identification process.

5.2.4 Implementing new software metrics

The new ECMAScript standard (ECMAScript2015 or ES2015) introduces the syntax necessary to create classes in JavaScript, a highly anticipated and requested feature. Although most JavaScript implementations rely on the language native *object* type to model classes, if the feature is successful in gaining some traction then the object-oriented metrics referred in Chapter 2 can be made part of the solution, by an extension of the *escomplex* library [esca].

By implementing new metrics and testing them against *Jscrambler's* services, specially the obfuscation process, one can guide the way the services are developed. If some metric is resilient enough to a set of transformations then it presents a risk and the services development should address its masking in order to minimize the risk of reverse-engineering.

5.2.5 Guide the company's obfuscation process

By having the software metrics retrieved from the original code, the company's obfuscation can be enhanced in two ways, if deemed suitable. Firstly, an interval can be established as to how much the metrics can change after the process. The user might want the number of new variables created to remain within certain bounds, for example. Secondly, there recommendations can be provided based on the metrics retrieved. If the cyclomatic complexity of a code is high then the tool can suggest the application of control flow masking measures.

[2]`https://angularjs.org/`

References

[All70] Frances E Allen. Control flow analysis. In *ACM Sigplan Notices*, volume 5, pages 1–19. ACM, 1970.

[asy] Async - https://github.com/caolan/async.

[bab] Babel - https://babeljs.io/.

[bac] BackboneJS - http://backbonejs.org/.

[BBM96] Victor R. Basili, Lionel C. Briand, and Walcélio L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22(10):751–761, 1996.

[blu] Bluebird - http://bluebirdjs.com.

[boo] BookshelfJs - http://bookshelfjs.org/.

[BR88] Victor R Basili and H Dieter Rombach. The tame project: Towards improvement-oriented software environments. *Software Engineering, IEEE Transactions on*, 14(6):758–773, 1988.

[BS09] Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. *ACM SIGPLAN Notices*, 44(10):243, 2009.

[cal] Difference between callbacks and promises - https://www.quora.com/Whats-the-difference-between-a-promise-and-a-callback-in-Javascript.

[che] Cheerio - https://github.com/cheeriojs/cheerio.

[CK94] Shyam R Chidamber and Chris F Kemerer. A metrics suite for object oriented design. *Software Engineering, IEEE Transactions on*, 20(6):476–493, 1994.

[com] Commander - https://github.com/tj/commander.js.

[con] ConnectJS - http://connect-js.com/.

[Cro08] Douglas Crockford. *JavaScript: The Good Parts: The Good Parts.* " O'Reilly Media, Inc.", 2008.

[doj] Dojo - https://dojotoolkit.org/.

[ecm] ECMAScript2015 - http://www.ecma-international.org/ecma-262/6.0/.

[esca] Escomplex - https://github.com/jared-stilwell/escomplex.

[escb] Escope - https://github.com/estools/escope.

[espa] Espree - https://github.com/eslint/espree.

[espb] Esprima - http://esprima.org/.

[expa] Express - http://expressjs.com/.

[expb] Express-session - https://github.com/expressjs/session.

[Fen94] Norman Fenton. Software measurement: A necessary scientific basis. *Software Engineering, IEEE Transactions on*, 20(3):199–206, 1994.

[Fla11] David Flanagan. *JavaScript: The definitive guide: Activate your web pages.* " O'Reilly Media, Inc.", 2011.

[GA13] Daniel Graziotin and Pekka Abrahamsson. Making sense out of a jungle of javascript frameworks. In *Product-Focused Software Process Improvement*, pages 334–337. Springer, 2013.

[GCP12] Andreas B Gizas, Sotiris P Christodoulou, and Theodore S Papatheodorou. Comparative Evaluation of JavaScript Frameworks. *www 2012 Companion*, (Cc):513–514, 2012.

[GK91] Geoffrey K Gill and Chris F Kemerer. Cyclomatic complexity density and software maintenance productivity. *Software Engineering, IEEE Transactions on*, 17(12):1284–1288, 1991.

[Gol10] Robert Gold. Control flow graphs and code coverage. *International Journal of Applied Mathematics and Computer Science*, 20(4):739–749, 2010.

[goo] Google Octane Benchmark Suite - https://developers.google.com/octane/.

[GR04] Rebecca Guenther and Jacqueline Radebaugh. Understanding metadata. *National Information Standard Organization (NISO) Press, Bethesda, USA*, 2004.

[han] HandlebarsJS - http://handlebarsjs.com/.

[HG12] Brian Hackett and Shu-yu Guo. Fast and Precise Hybrid Type Inference for JavaScript. 2012.

[JI12] Enter Javascript and Technical Inspirations. JavaScript: Designing a Language in 10 Days. (February):7–8, 2012.

[JMM11] Simon Holm Jensen, Magnus Madsen, and Anders Møller. Modeling the HTML DOM and Browser API in Static Analysis of JavaScript Web Applications. *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, pages 59–69, 2011.

[JMT09] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for JavaScript. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 5673 LNCS(274):238–255, 2009.

[jqu] Jquery - https://jquery.com/.

[jso] MDN JS Objects - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_objects.

[jsw] JavaScript Window - https://developer.mozilla.org/en-US/docs/Web/API/Window.

[jsx] JSX - https://facebook.github.io/react/docs/jsx-in-depth.html.

[KDK$^+$14] Vineeth Kashyap, Kyle Dewey, Ethan A Kuefner, John Wagner, Kevin Gibbons, John Sarracino, Ben Wiedermann, and Ben Hardekopf. Jsai: A static analysis platform for javascript. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 121–132. ACM, 2014.

[LC87] H.F. Li and W.K. Cheung. An Empirical Study of Software Metrics. *IEEE Transactions on Software Engineering*, SE-13(6):697–708, 1987.

[McC76] Thomas J McCabe. A complexity measure. *Software Engineering, IEEE Transactions on*, (4):308–320, 1976.

[min] Maintainability Index - http://avandeursen.com/2014/08/29/think-twice-before-using-the-maintainability-index/.

[MLF13] Magnus Madsen, Benjamin Livshits, and Michael Fanning. Practical static analysis of javascript applications in the presence of frameworks and libraries. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 499–509. ACM, 2013.

[mona] MongoDB - https://www.mongodb.org/.

[monb] Mongoose - http://mongoosejs.com/.

[moo] MooTools - http://mootools.net/.

[MS13] Josip Maras and Maja Stula. Identifying Code of Individual Features in Client-Side Web Applications. 39(12):1680–1697, 2013.

[mus] Mustache - https://github.com/mustache/mustache.github.com.

[ned] NeDB - https://github.com/louischatriot/nedb.

[noda] MySQL - https://github.com/mysqljs/mysql.

[nodb] Node SOAP - https://github.com/vpulim/node-soap.

[nodc] Node SQLite3 - https://github.com/mapbox/node-sqlite3.

[nodd] Node.js - https://nodejs.org/en/.

[OH92] Paul Oman and Jack Hagemeister. Metrics for assessing a software system's maintainability. In *Software Maintenance, 1992. Proceerdings., Conference on*, pages 337–344. IEEE, 1992.

[pas] PassportJS - http://passportjs.org/.

[pos] PostgreSQ - https://www.postgresql.org/.

[pro] PrototypeJS - http://prototypejs.org/.

[q] Q - https://github.com/kriskowal/q.

[quo] Quora - why use bower when there is npm.

[rea] React - https://facebook.github.io/react/.

[RMT09] Mehwish Riaz, Emilia Mendes, and Ewan Tempero. A systematic review of software maintainability prediction and metrics. *2009 3rd International Symposium on Empirical Software Engineering and Measurement, ESEM 2009*, pages 367–377, 2009.

[S$^+$14] Kyle Simpson et al. *You Don't Know JS: this & Object Prototypes*. " O'Reilly Media, Inc.", 2014.

[S$^+$15] Kyle Simpson et al. *You Don't Know JS: Types & Grammar*. " O'Reilly Media, Inc.", 2015.

[Sim14] Kyle Simpson. *You Don't Know JS: Scope & Closures*. " O'Reilly Media, Inc.", 2014.

[Sim15] Kyle Simpson. *You Don't Know JS: Async & Performance*. " O'Reilly Media, Inc.", 2015.

[swi] Swig - http://paularmstrong.github.io/swig/.

[tra] Transpilation Examples - http://es6-features.org.

[Wey88] Elaine J Weyuker. Evaluating software complexity measures. *Software Engineering, IEEE Transactions on*, 14(9):1357–1365, 1988.

[Wil97] David S. Wile. Abstract syntax from concrete syntax. *Proceedings - International Conference on Software Engineering*, pages 472–480, 1997.

[win] Windows Store - https://www.microsoft.com/en-us/store/apps.

[Wög05] W Wögerer. A survey of static program analysis techniques. *Vienna University of Technology*, pages 1–16, 2005.

[WP02] Francis G Wolff and Chris Papachristou. Multiscan-based test compression and hardware decompression using lz77. In *Test Conference, 2002. Proceedings. International*, pages 331–339. IEEE, 2002.

[Zus05] Horst Zuse. *Resolving the Mysteries of the Halstead Measures*. Technische Universität Berlin, Fakultät IV-Elektrotechnik und Informatik, 2005.

Appendix A

JScrambler Transformations

In this appendix, the transformations used from *JScrambler's* website [1] are described as per the help page [2] found on the site.

The following is the list of transformations used in the project. The list does not comprise all the transformations the company offers but only the ones which were used to obtain experimental results for the project.

- Comment Removal
- Dictionary Compression
- Dot notation obfuscation
- Duplicate literals elimination
- Function outlining
- Function reordering
- Rename (local)
- Self-defending
- Whitespace removal

A.1 Description

A.1.1 Comment Removal

Removes all the comments in the code since they are unnecessary for its execution.

[1] `https://jscrambler.com`
[2] `https://jscrambler.com/en/help`

A.1.2 Dictionary Compression

Uses a lossless data compression algorithm (LZ77 [WP02]). The algorithm creates a dictionary of the JavaScript source code which is then used to replace duplicates with a reference to the existing match, thus achieving the compression.

A.1.3 Dot notation obfuscation

Transforms JavaScript's dot notation into array subscript notation.

An example of this transformation can be found on Listing A.1.

```
//source code
navigator.plugins.length

//transformed code
var a = navigator, b = 'plugins', c = 'length'; a[b][c];
```

Listing A.1: An example of the dot notation obfuscation transformation.

A.1.4 Duplicate literals elimination

Replaces duplicate literals by a variable, which will replace repeated usages. When minifying JavaScript, the transformation only occurs if the replacements yield a smaller code size.

The variable declaration is added to the first private scope accessible by all the usages of the variables. If a private scope is not found then it is added to the global one. To avoid polluting the global scope the variables are added to an object literal.

An example of this transformation can be found on Listing A.2.

```
//source code
variable1 = "http://jscrambler.com";
function1("http://jscrambler.com");

//minified code
var a = "http://jscrambler.com";
variable1 = a;
function1(a);
```

Listing A.2: An example of the duplicate literals elimination transformation.

A.1.5 Function outlining

Creates new function declarations from a single or group of statements. The resulting code has an altered structure and is, therefore, harder to read/follow.

An example of this transformation can be found on Listing A.3.

```
//source code
function doesSomething() {
   // more code
   if(predicate) {
      // more code
      variable = statement;
      // more code
   }
   // more code
}


//minified code
O = {
   'functionOutline1' : function (argumentN) {
      variable = argumentN;
   },

   'functionOutline2' : function(argumentN) {
      return argumentN;
   }
}

function doesSomething() {
   // more code
   if(O.functionOutline2(predicate)) {
      // more code
      O.functionOutline1(statement);
      // more code
   }
   // more code
}
```

Listing A.3: An example of the function outline transformation.

A.1.6 Function reordering

Reorders the declaration of functions in a random fashion but taking into account the declaration hoisting.

A.1.7 Local rename

Renaming transformations are responsible for replacing identifiers present in the code for ones without meaning. By making identifiers smaller and removing their meaning it is harder to reason about the code without affecting the way the program is executed.

Local names are those which are private, that is, those that are hidden from the global scope namespace. Any name which can be publicly called is not replaced by this transformation.

An example of this transformation can be found on Listing A.4

```
//source code
function doesSomething(argument1) {
    var variable1 = "alert";
    window.alert(argument1+variable1);
}

//minified code
function doesSomething(a) {
    var b = "alert";
    window.alert(a+b);
}
```

Listing A.4: An example of the local renaming transformation.

A.1.8 Self-defending

Obfuscates functions and objects by concealing their logic. Frustrates code tampering attempts by using anti-tampering and anti-debugging techniques.

Trying to tamper the code will break its functionality and using JavaScript debuggers will trigger defenses to hinder the analysis.

A.1.9 Whitespace removal

Removes all white spaces and newlines from the code.

An example of this transformation can be found on Listing A.5

```
//source code
function doesSomething(argument) {
    if (argument > 2) {
       return argument
   } else {
       return 0
   }
}

//minified code:
function doesSomething(argument){if(argument>2){return argument}else{return 0}}
```

Listing A.5: An example of the whitespace removal transformation.

Index