Deobfuscating JavaScript

M. Sc. Computer Science thesis

Jan Frederick Walther

Universität Bremen

30.05.2022

E R K L Ä R U N G

Ich versichere, die Masterarbeit oder den von mir zu verantwortenden Teil einer Gruppenarbeit*) ohne fremde Hilfe angefertigt zu haben. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen sind, sind als solche kenntlich gemacht.

*) Bei einer Gruppenarbeit muss die individuelle Leistung deutlich abgrenzbar und bewertbar sein und den Anforderungen entsprechen.

Bremen, den ____________________

____
(Unterschrift)

1 Intro

Since its inception in 1996, JavaScript, or as this thesis will call it, ECMAScript, has established itself as the premier choice for running code within a web browser. With the introduction of platforms like Node.js, ECMAScript has also been adopted outside of the browser. It is also a target for other programming languages like Kotlin, TypeScript, or Dart. This ubiquitous presence allowed ECMAScript to be present on many hardware platforms. Smartphones and desktop computer users retrieve and execute ECMAScript mostly unbeknownst to them while accessing various websites.

It has become common practice to apply multiple kinds of lossless compression to their ECMAScript files, as is indicated by over fifteen million weekly downloads of UglifyJS[5]. This tool minifies ECMAScript files, which includes renaming all variables and functions with random, short, and unique names to save space without modifying the semantics of a program. In addition to that, it also removes superfluous whitespace. A curious user may undo the last transformation of the source code by using a formatting tool, the former not so much.

In addition to benign compression, a more nefarious use case for software makes ECMAScript harder to read and comprehend for humans. This process is called obfuscation. Malicious software authors sometimes employ it to hide their intent from the user. These use cases can range from browser privacy violations to malicious software deployed on the user's computer using an ECMAScript interpreter native to the operating system.

A simple example of visualizing this process is considering a simple program that only outputs the number one thousand in decimals. The commonly used way in ECMAScript would be `console.log(1000)`. The function output to the console is insignificant and will be omitted for this example. A valid ECMAScript program may represent the number itself in numerous ways. Replacing the integer with `500 + 500` would yield the same result as replacing it with a hexadecimal representation (`0x3E8`). Both ways of representing the numeric value 1000 keep the program's semantics intact while modifying its syntax. Transformations like these may be applied automatically on a program-wide scale, significantly hampering the source code's readability.

While classical approaches may remove structural obfuscation, to some extent, they may not undo the automated renaming employed by benign and malicious agents.

This thesis introduces a deobfuscation program that combines the classical approach to deobfuscation with machine learning to increase the readability of obfuscated programs. This deobfuscation will hopefully allow casual users and reverse engineers more insight into the code running on their internet-connected devices and gauge an obfuscated application's intent.

The classical approach includes techniques already in use in optimizing compilers. The machine learning component will attempt to restore the identifiers replaced in the obfuscation or minification. It acquired this ability from training on a massive dataset of open-source ECMAScript projects. This technique allows for a partial restoration of information erased in the obfuscation process.

This thesis will begin by analyzing the state-of-the-art in ECMAScript deobfuscation in chapter 2.

The thesis will then move on to the foundational knowledge around ECMAScript and programming languages in chapter 3 to allow even a casual observer of the ECMAScript language to understand the topics discussed.

It will then move on to analyzing the adversaries within this space by looking at a commonly used obfuscation software and a hand-picked selection of obfuscated malware samples in chapter 4.

With the adversaries in mind, it will move on to the implementation in chapter 5. It will include the reversal of the obfuscation discussed in the previous chapter and explain the design choices taken within the deobfuscation software.

The thesis will then include an evaluation chapter 6, which is based upon a case study of a deobfuscated program and a survey.

Ideas for continuing the ideas and implementations introduced in this thesis are presented in chapter 7.

This thesis will discuss the results in chapter 8.

2 State of the art

This section will attempt to capture ECMAScript deobfuscation's state of the art. The existing solutions can be sorted into two categories. The first category contains tools that attempt to utilize machine learning to deobfuscate code. The second category instead utilizes classical optimizations taken from compiler theory instead.

2.1 JSNice

JSNice was released as part of a research project called "Predicting Program Properties from"Big Code" " by Raychev et al. in 2015[29].

JSNice does not focus on deobfuscating expressions but instead intends to infer names and types of variables using machine learning. ECMAScript does not support type annotations at the time of writing, so the inferred types are added as comments above the declaration instead.

It is based on Google's Closure Compiler, which will be discussed later in this chapter, and uses the same comment syntax for type annotations.

To train the underlying model, "10,517 JavaScript projects [were downloaded] from GitHub"[29, p. 120]. The subsequent evaluation of the trained model was done on 50 ECMAScript repositories from BitBucket sorted by the number of commits. The authors chose BitBucket to reduce the probability of overlap between training and evaluation data. As part of their due diligence, they ran additional checks to ensure that the repositories from GitHub did not include those downloaded from BitBucket.

The evaluation was done by first applying UglifyJS[5] on all test files, erasing all identifiers, and replacing them with short, automatically generated identifiers to reduce the file size. Existing type annotations, extraneous whitespace, and comments are removed. These type annotations are not a part of the official specification but rather a community-driven effort to ease debugging and maintainability. Multiple different formats exist but are limited to exist in comments due to the syntactical limitations of the language. Several configurations of JSNice were then used to evaluate the precision. These configurations included a system with access to all training data, one with access to 10% of the training data, and one with access to1% of the training data. Another configuration that keeps all identifiers and annotates every type with the most common type, `string`, was added as a baseline.

The researchers then compared the name predictions to the original identifier names before applying UglifyJS. This evaluation yielded an accuracy of 63.4% for the system that had access to all the training data[29, p. 121]. The system with access to 10% of the training data correctly predicted the identifiers with a chance of 54.5%[29, p. 121]. The system with access to only 1% of the training data yielded an accuracy of 41.2%[29, p. 121]. Comparing these results to the baseline indicates an improvement over no prediction, which only yielded an accuracy of 25.3%[29, p. 120].

The precision of the prediction of type annotations indicated that it was less dependent on large datasets. The difference between the system that had access to all training data (81.6%) and the one that had access to only 1% of the training data (77.9%) was a mere 3.7%[29, p. 121]. The research did not analyze the efficacy of JSNice on programs that employed additional obfuscation techniques.

It has to be noted that UglifyJS will not rename global variables, which may inflate the precision[29, p. 121].

2.2 JSNaughty

JSNaughty combines JSNice and another solution for recovering identifiers called "AUTONYM". Autonym utilizes statistical machine translation (SMT), a technique used in natural language processing. "SMT is a data-driven approach to machine translation, based on statistical models estimated from (large) bi-lingual text corpora"[38, p. 684]. The training is done similarly to JSNice as both employ UglifyJS to generate obfuscated source code from code taken from openly available repositories.

The researchers observed "that the static analysis based JSnice[sic] and our simpler, SMT (Moses) based Autonym perform comparably. The precision values for JSnice are more dispersed: the interquartile range is 0–67%, with a median of 25%; in contrast, Autonym has higher median precision, 30%, but the distribution is more concentrated (IQR 10–60%)"[38, p. 690].

2.3 Google Closure Compiler

Google developed the closure compiler as a part of their Closure suite of tools and libraries to build "powerful and efficient JavaScript"[19]. It applies classical compiler techniques to optimize and minimize ECMAScript code. The project's source code is freely available and may be used as a source for analyzing the techniques it employs. However, what separates it from the previous entries in this chapter is that it is not meant to transform code to make it easier for humans to understand but rather to make it smaller and more efficient for ECMAScript interpreters. It may, therefore, also apply optimizations to the code, making it harder for humans to understand.

Since the resulting code is meant to be executed, it may not perform some optimizations that could improve the readability at the cost of generating invalid or inefficient code. This thesis aims not to make code run faster but to improve its readability. Therefore, it may assume more properties about the code that may result in readable but not necessarily executable code.

Optimizations are grouped into different levels. These levels are "whitespace-only", "simple", and "advanced". The following paragraphs will highlight some techniques the closure compiler applies at different optimization levels[18].

Optimizations at the whitespace-only level only affect the whitespace within the input program. It is removed to save space as it is a lossless way of compressing an ECMAScript program. This optimization may be undone using a formatting tool. A later chapter will analyze why this transformation may not be side effect free in some cases.

The compiler will rename local variables at the “simple” level using a similar method to the previously discussed UglifyJS. This optimization is harmful to the readability of the generated code and may not be disabled without modifying the source code of the Closure compiler directly.

At the advanced level, the compiler will optimize the code more aggressively, including more aggressive renaming, inlining, and dead code removal. These optimizations warn that the generated code may be invalid if some assumptions the compiler makes do not apply to the code.

2.4 SAFE-DEOBS

SAFE-DEOBS is a project developed by Adrian Herrera using the SAFE framework. It is designed to use existing techniques used by optimizing compilers to deobfuscate ECMAScript programs with a particular focus on ECMAScript-based malware.

It operates only on the abstract syntax tree as the SAFE framework does not support transforming back from higher levels of intermediate representation[14, p. 3]. However, information gathered from the control flow graph is still used for optimization purposes.

It implements several techniques that are implemented[14, p. 3] within this thesis as well. The first of these techniques is constant folding, which is discussed in section 5.2. Constant propagation, implemented under the name inlining within this thesis, is also supported. Another technique implemented by both SAFE-DEOBS and this thesis is dead-branch removal. Function inlining, called “constant evaluation” in this thesis, is also implemented. It also implements variable renaming but uses a different approach than this thesis.

The results of the deobfuscation were analyzed using escomplex, which employs traditional code complexity metrics such as physical lines of code, number of functions, cyclomatic complexity, and Halstead length. The evaluation of this thesis will do the same analysis to compare its results to SAFE-DEOBS.

3 Technical Background

This chapter will begin with the foundational knowledge required for working with formal languages. It will then introduce the basic concepts of ECMAScript and explain its complicated history. Afterward, it will introduce the basic concepts of software obfuscation to prepare the reader for the subsequent chapters.

3.1 Programming languages

Humans have built specialized languages to communicate with other humans more efficiently because all participants share knowledge. Well-known examples are “artificial languages, like the logical propositions of Aristotle or the music sheet notation of Guittone d’Arezzo” [2, p. 5]. Konrad Zuse laid the foundation for this development in the space of computer science in 1945 with “Plankalkül”, his extensions of David Hilbert’s “Aussagenkalkül” (propositional calculus) and “Prädikatenkalkül” (predicate calculus)[22, p. 203].

Plankalkül, however, except for “brief excerpts”, wasn’t published until 1972[22, p. 203]. Konrad Zuse designed the language “with the aim to provide a uniform language of formulae adapted to all kinds of calculating problems” [41, p. 1].

The term “formal language” will be used to differentiate the “artificial” languages introduced in the preceding paragraphs from the languages that have naturally developed for inter-human communication. The following chapter will briefly introduce the basics, especially the mathematical notation needed to interact with formal languages and this thesis.

3.2 Formal languages

The following definitions are taken in abridged and sometimes verbatim from Peter Linz’s 2001 work “An Introduction to Formal Languages and Automata” [24, pp. 14–26]. Basic knowledge of set theory and the related notation is considered a prerequisite for this thesis.

3.2.1 Definitions

A formal language is defined over an underlying alphabet, often denoted by either $\mathcal{A}$ or the Greek letter sigma (Σ), which is a non-empty set of symbols. This thesis will use the latter notation. In a natural language, like the English language, the counterpart would be the Latin alphabet consisting of 26 letters in a lower and an upper case variant.

From the symbols defined by the alphabet, finite sequences of the symbols may be formed. These sequences of symbols are called strings and are pretty similar to the concept of strings from a multitude of high-level programming languages.

Given the alphabet $\Sigma = \{a, b\}$ the string $"aa"$ is a string on Σ or more formally: $"aa" \in \Sigma$.

Strings may be concatenated $w = ab$, with $a \in \Sigma$ and $b \in \Sigma$ and the concatenation will also be a string on Σ. This example can also introduce the concepts of prefixes and suffixes. a would be called a prefix of w, and b would be called a suffix.

The length of a string is the number of symbols contained within and is denoted by $|a|$ where $a \in \Sigma$.

An alphabet always contains the empty string, which is usually, depending on the source, denoted by either ϵ or λ. For this thesis, it will be denoted as ϵ. The length of the empty string is 0 ($|\epsilon| = 0$). A string may be exponentiated w^n with $w \in \Sigma$ which means that w is concatenated with itself n-times. The special case of $n = 0$ is defined as $w^0 = \epsilon$.

Exponentiation is also defined for alphabets where Σ^n is the set of all strings obtained by concatenating n elements of Σ. The infinite set of all possible strings that can be obtained by concatenating zero or more symbols from Σ is denoted as Σ^*. Since $\epsilon \in \Sigma$ by the definition of alphabets it follows that $\{\epsilon\} \subset \Sigma^*$. The set of all obtainable strings except for the empty string is denoted as $\Sigma^+ = \Sigma^* - \{\epsilon\}$.

3.2.2 Formal grammars

However, a string of the alphabet Σ is not necessarily a word of the language L, which is defined over Σ. Formal grammars define which strings are elements (or words) of the language L. A grammar in formal languages is very similar to its natural counterpart, with the main difference being that the former is well defined while the latter is malleable and imprecise.

Peter Linz defines formal grammars as a quadruple $G = (V, T, S, P)$ with:

- V being a finite set of objects called variables (sometimes called non-terminals)
- T being a finite set of objects called terminal symbols
- $S \in V$ being the special start variable symbol
- P being a finite set of productions

With $V \neq \emptyset, T \neq \emptyset$ and $T \cap V = \emptyset$. The set of productions P, sometimes called production rules, is a set of rules that specify how non-terminal symbols in a string may be transformed into another symbol. Transformations of this nature are denoted as $x \to y$, where $x \in (V \cup T)^+$ and $y \in (V \cup T)^*$. Given the strings $w = uxv$ and $z = uyv$ with $u, v, y \in (V \cup T)$ and $x \in V$ the rule above would be applied as $w \Rightarrow z$. $w \Rightarrow z$ is said as w derives z or z is derived from w.

Productions may be applied as often as desired and applicable. $w_1 \Rightarrow w_2 \Rightarrow \cdots \Rightarrow w_n$ can be said as w_1 derives w_n. This can be contracted to $w_1 \overset{*}{\Rightarrow} w_n$ to indicate that w_n can be derived from w_1 in an undefined, and possibly zero, number of steps.

The set of all strings, starting from the start symbol S, that can be obtained by applying the production rules, in any order, is the language *defined* or *generated* by the grammar. So with $G = (V, T, S, P)$ the set $L(G) = \{w \in T^* : S \overset{*}{\Rightarrow} w\}$ is the language *generated* by G.

3.2.3 Chomsky hierarchy

Noam Chomsky introduced a hierarchy of formal languages[8] to further reason about formal languages and their generating grammars. The so-called Chomsky hierarchy divides formal languages into four types based on the production rules they define. Chomsky also defines the closure properties of each type of formal grammar.

The following sections will briefly glance over type 0 and type 1 languages as they are of little importance to this thesis and elaborate further on the remaining types of languages. Please note that all types within the hierarchy are subsets of higher types. This fact is visualized in figure 3.1. Therefore, all regular languages are a subset of context-free languages. All context-free languages are subsets of context-sensitive languages, and all context-sensitive languages are part of the recursively enumerable languages[24, p. 295]. Languages outside this hierarchy are proven to exist but are nonessential to this thesis[21, p. 372].

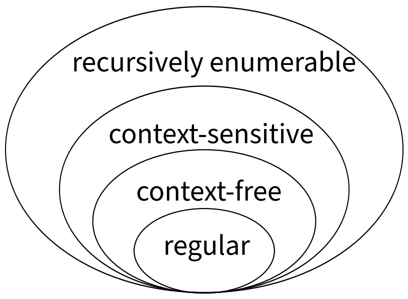

Figure 3.1: A graphical representation of the sets of languages included in the Chomsky hierarchy. **Source:** J. Finkelstein, CC BY-SA 3.0, via Wikimedia Commons

Even though the hierarchy is used as a standard material in automata theory and formal languages, it was designed with natural languages in mind. Nevertheless, it lays the foundation for understanding how a computer reads formal languages.

3.2.3.1 Type 3: Regular

Languages are considered regular if and only if there is a deterministic finite automaton that accepts them. *Regular expressions* may be used to express regular languages. ECMAScript, among other programming languages, supports regular expressions, albeit in an extended version. The symbols of such a regular expression that can represent a regular language are the symbols of the underlying alphabet and three operators ($\Sigma \cup \{+, \cdot, *\}$).

The plus operator is defined as union operator so given the alphabet $\Sigma = \{a, b\}$ and the regular expression $r = a + b$ the language generated by r would be $L(r) = \{\epsilon, a, b\}$[24, p. 72]. This behavior differs from the behavior of regular expressions as they are known from programming languages such as ECMAScript or Perl. It signifies repetition, but instead of zero or more, it represents one or more repetitions. A vertical bar (|) is usually used as a union operator instead.

The star operator, also known as Kleene star, Kleene closure or star closure[21, p. 87], denotes the zero or more occurrences of the preceding symbol(s). The language generated by the alphabet $\Sigma = \{a\}$ and the regular expression $r = a*$ would be $L(r) = \{\epsilon, a, aa, aaa, \ldots\}$. This behavior is also used in regular expressions, implemented in programming languages like ECMAScript or Perl.

The dot operator signals concatenation and may be omitted[24, p. 74]. So the language generated by the alphabet $\Sigma = \{a, b\}$ and the regular expression $r = a \cdot b$ would be $L(r) = \{\epsilon, ab\}$.

Noam Chomsky defines constraints on the production rules of each type of grammar in “On Certain Formal Properties of Grammars”[9]. For regular languages he defines those constraints that all production rules may only be of the form $A \to a$ or $A \to aB$ with $a \in T; A, B \in V$ and $a \neq \epsilon$[9, p. 149].

3.2.3.2 Type 2: Context-free

Context-free grammars are the foundation of modern programming languages[3, p. 40]. They are primarily represented in the Backus-Naur-Form or the extended variant, which section 3.2.4 will visit and explain in-depth[3, p. 40].

Parsing, the process of finding a sequence of derivations by which an input $w \in L(G)$ is derived[24, p. 136], is proven to always work on context-free grammars. It is furthermore proven that the derivation for any $w \in L(G)$ can be found “in a number of steps proportional to $|w|^3$”[24, p. 139].

Furthermore, software capable of automatically generating code to parse context-free languages has been developed. It may generate the code for a multitude of popular programming languages. Those parser generators usually take an extended version of the Backus-Naur form from section 3.2.4.

Context free languages are constrained to production rules of the form $A \to \alpha$ with $\alpha \in V \cup T \cup \{\epsilon\}$[9, p. 150].

3.2.3.3 Type 1: Context-sensitive

Context-sensitive languages are constrained to production rules of the form $\alpha A \beta \to \alpha \gamma \beta$, where $\alpha, \beta, \gamma \in (V \cup T)^+; A \in V$ and $\gamma \neq \epsilon$[9, p. 142].

An alternative definition can be found in Peter Linz’s “An Introduction to Formal Languages and Automata”[24]. Context-sensitive grammars after Linz are defined as a grammar G, where all production rules are of the form $\alpha \to \beta$ where $\alpha, \beta \in (V \cup T)^+$ and $|x| \leq |y|$[24, p. 289]. This definition observes that grammars of those types can also be called non-contracting as “the length of successive sentential forms can never decrease.” [24, p. 290].

3.2.3.4 Type 0: Recursively enumerable

Recursively enumerable languages have no constraints [9, p. 143] and are equivalent in complexity to a Turing machine[24, pp. 284–286]. They are of little practical use in programming languages as even trivial properties tend to be undecidable due to the nature of Turing machines[24, p. 309]. The only restriction applied to them is that the left-hand side of the production rules may not be empty as per the definition of production rules[24, p. 283].

3.2.4 Backus–Naur form

The Backus-Naur form (BNF) is a way to describe context-free grammars in an easily understood format. It is known for widespread use when describing the grammar of a programming language[24, p. 146]. Non-Terminals are placed on the left-hand side of the assignment operator (`::=`). The right-hand side represents the production rules for the preceding non-terminal. As defined above, they may contain terminal symbols as well as non-terminals. Terminal symbols are rendered as-is, while non-terminal (or variable) symbols are rendered in angle brackets instead. Alternatives are denoted by a vertical line character (|).

The following is an example of a formal grammar described in BNF. It describes a formal grammar for displaying basic arithmetic expressions on $\mathbb{N}_0$ with the following operators:

- $+$ for addition
- $-$ for subtraction
- $*$ for multiplication
- $/$ for division

The grammar enforces the precedence of multiplicative operations over additive by splitting the operators into two different non-terminal symbols. Infinite leading zeroes for the `Constant` productions are included in this grammar for brevity.

```
Expression             ::= <Term> | <Term> <AdditiveOperator> <Term>
AdditiveOperator       ::= + | -
MultiplicativeOperator ::= * | /
Term                   ::= <Factor> | <Factor> <MultiplicativeOperator>
     <Factor>
Factor                 ::= <Constant> |  (  <Expression>  )
Constant               ::= <Digit> | <Digit> <Constant>
Digit                  ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0
```

The use of the BNF becomes obvious immediately once one considers the underlying production rules of the represented grammar. The first few production rules of the above grammar are listed below to illustrate this fact.

- $S \rightarrow Expression$
- $Expression \rightarrow Term$
- $Expression \rightarrow Term\ AdditiveOperator\ Term$
- $AdditiveOperator \rightarrow "+"$

- $AdditiveOperator \rightarrow "-"$
- $MultiplicativeOperator \rightarrow "*"$
- $MultiplicativeOperator \rightarrow "/"$
- $\cdots$

It becomes apparent that the Backus-Naur form provides a more concise way of representing context-free grammars. Multiple extensions to the Backus-Naur form as defined by John Backus and refined by Peter Naur have been accepted, and some have even been standardized.

The most important extension is the aptly named extended Backus–Naur form (EBNF). One feature added to this extension is the optional operator, designated by square brackets. For example the line `Expression ::= <Term> | <Term> <AdditiveOperator> <Term>` could be written as `Expression ::= Term [AdditiveOperator Term]`, making the representation even more concise.

Another extension is the repetition operator, as designated by braces. For example, the line that defines the rules for the $Constant$ non-terminal could instead be written using the repetition operator as `Constant ::= {Digit}`. EBNF further eliminates the requirement for angle brackets around non-terminals by requiring terminals to be within quotation marks instead. The extended form also eases incorporating special characters from BNF (<, >, `::=`, |) into a formal grammar.

3.2.5 Ambiguities

As mentioned in the section 3.2.3.2, context-free grammars are used as the foundation of most modern programming languages. The programming language C++ includes in the 2020 iteration of its standard specification that the language generated by the context-free grammar included in the standard is a superset of the valid C++ programs[20, §A.1]. The C++ programming language is but one example where programming languages diverge from the theoretical construct of a formal language.

An easy-to-understand example of this behavior in C++ is the ambiguity between variables and types. Suppose the construct `A * B;` were to be parsed. With the C++ programming language in mind, it becomes evident that its meaning depends entirely on whether `A` is a type or `A` and `B` are declared as variables. In the first case, it would be a declaration of a pointer of the type `A` with the identifier `B`, and in the latter case, it would be a multiplication with a discarded result. Both are valid statements and need to be resolved. Eli Bendersky describes how different C and C++ parser implementations solve this problem. He describes that the usual way of dealing with this ambiguity is to amend an automatically generated C or C++ parser (what he calls the “lexer hack”) with a bidirectional data flow between the lexer and the symbol table[6].

3.3 Compiling & Interpreting

After introducing the theoretical foundations of formal languages, we will cover how a given text in a formal language is translated into a semantically equivalent program. The structure of this chapter will mirror the chapter “The structure of a compiler” from Alfred Aho et al.’s “Compilers: principles, techniques, & tools”[3]. Due to the age of the source mentioned above, this thesis will add updates and alternatives as necessary.

Aho et al. begin by splitting the processes of a compiler into two phases. The first one is analysis, and the second one is synthesis. “The analysis part breaks up the source program into constituent pieces and imposes a grammatical structure on them” [3, p. 4].

The second phase, synthesis, is responsible for turning the intermediate representation produced by the first phase. However, the exact details for this phase are not particularly relevant for this thesis as the target is not creating an executable program. The focus lies on generating deobfuscated source code instead.

Aho et al. begin the first phase with the “Lexical Analyzer” (sometimes also referred to as Lexing)[3, p. 5]. They then proceed to the syntactical analysis and finally move on to the semantic analysis.

This chapter will also serve as an overview of how real-world implementations of programming languages and their parsers differ from their theoretical origins.

3.3.1 Lexical analysis

Modern compilers usually do not process the input character stream directly but instead work on tokens. A token is a container of a token type and an optional attribute reference. While Aho et al. describe the token attribute as a reference to the symbol table, modern compilers usually represent tokens in an object-oriented manner. As such, the attributes are fields of the token object. The token’s location in the source code is often included as an attribute of tokens to ease handling analysis errors.

An example of tokens would be identifiers, which are used for many constructs in programming languages. Some use-cases are to name methods, variables, and classes within certain programming languages, keywords, or literal values.

Turning the character stream of the input source code into a stream of tokens is done by using lexemes. A *lexeme* is defined as a “sequence of characters in the source program that matches the pattern for a token and is identified by the lexical analyzer as an instance of that token” [3, p. 111]. A pattern is “a description of the form that the lexemes of a token may take” [3, p. 111]. This description may take the form of a regular expression in some cases[3, p. 117].

Usually, those regular expressions are extended versions of the regular expressions introduced in section 3.2.3.1. The extensions introduced by Aho et al. include the introduction of the postfix one-or-more operator “˙”, which is represented by a postfix $+$ in ECMAScript[3, p. 124]. The second extension is the postfix zero-or-one operator, which Aho et al. and ECMAScript denote with a $?$[3, p. 124]. The final and most complex extension introduced by Aho et al. is character classes. They allow the grouping of a set of characters to be reused. An example of this would be `[A-Z]`, which is the character class that describes all uppercase ASCII characters. The regular expression for an identifier could, using these extensions, be defined as `[A-Za-z_][A-Za-z0-9_]*`. This regular expression would allow all identifiers beginning with an upper- or lowercase letter or an underscore followed by an arbitrary number of letters, underscores, or digits.

Aho et al. define five categories that the tokens usually found in a programming language can be sorted into. The following section will introduce these five categories with a short paragraph and examples from the ECMAScript specification.

- **Keywords**: Keywords like `if` and `else`, have the keyword itself as patterns. These tokens usually do not require any additional attributes beyond their location in the character stream, which is used for diagnostics and error messages. A sample lexeme for a keyword token would be "if" for the keyword `if`.
- **Operators**: Every operator, binary and unary, is assigned a token in the operator category. Additional attributes beyond the location are usually unnecessary in this category. Example lexemes would be `<=`, `-`, `!`, or `==`.
- **Identifiers**: The Identifiers category contains all tokens used as identifiers. These identifiers usually contain an attribute that contains the raw value of the identifier from the character stream. An example of this would be `main` as a function identifier for the main function.
- **Constants**: Constants, which are sometimes called literals, are constant values such as strings which are usually denoted by being contained within quotation marks, numeric literals such as floating-point numbers or integers in decimal, hexadecimal, or octal representation. The tokens in this category usually contain the raw value from the character stream as an additional attribute. In most cases, different representations are implemented as the same token. A parsing software may add an attribute for the numeric representation. Depending on the parser software, the information may also be discarded. This loss of information could make the source code harder to understand when transformed into a human-readable format. Example lexemes for these would be `0x3F`, `42`, `"Hello World!"`, and `1.33`.
- **Punctuation**: This category contains tokens for all the punctuation used "such as left and right parentheses, comma, and semicolon" [3, p. 112]. Additional attributes beyond the location in the character stream are usually not needed.

ECMAScript, in particular, defines the categories "reserved words, identifiers, literals, and punctuators" [12, § 5.1.2].

Lexical analysis can also remove comments from the input stream early. Since they are of no meaning for the rest of the parsing process in most languages, they can be discarded early on. ECMAScript differs as some programs may utilize comments to add improvised type annotations to programs.

3.3.2 Syntactical Analysis

Once the transformation of the stream of characters into a stream of tokens is complete, the syntactical analysis begins. This process can take many forms, most of them outside of the scope of this thesis. The most important types of parsers will be introduced, and their drawbacks will be highlighted accordingly.

The product of all highlighted parsing methods is the so-called *parse tree*. According to Aho et al., a parse tree meets the following four properties[3, pp. 45–46]:

1. The root is labeled by the start symbol.
2. Each leaf is labeled by a terminal or by e.
3. Each interior node is labeled by a nonterminal.
4. If A is the nonterminal labeling some interior node and $X, X_2, \cdots, X_n$ are the labels of the children of that node from left to right, then there must be a production $A \to X_1 X_2 \cdots X_n$. Here, $X_1, X_2, \cdots, X_n$ each stand for a symbol that is either a terminal or a nonterminal. As a special case, if $A \to \epsilon$ is a production, then a node labeled A may have a single child labeled ϵ.

To illustrate this concept we'll return to the example grammar from section 3.2.4 and build a parse tree for the expression $s = 13 + 10 * 5 + 1$. The data structures representing the non-terminal symbols will have to be defined to build a parse tree. The EBN form already dramatically simplifies this task by adding optional values. The structure needed for the `Expression` non-terminal, which is defined as `Expression ::= Term [AdditiveOperator Term]`, can be represented by a structure that contains one field for the right-hand side term, one for the left-hand side, and a field for the additive operator. However, the last two fields are optional, as described in the grammars EBNF. The `AdditiveOperator` non-terminal can be represented as a data structure containing one of the terminal symbols + or −. `Term` can be defined analogously to `Expression`.

The full derivation of the `10` in `10 * 5` would look something like $MultiplicativeExpression$, $ExponentiationExpression, UnaryExpression, UpdateExpression, LeftHandSideExpression$, $OptionalExpression$, $MemberExpression$, $PrimaryExpression$, $Literal$. This nesting is done to enforce operator precedence within the parse tree. However, the operator precedence is established once the expression has been parsed, and the intermediate steps may be purged. Purging all intermediate steps from a parse tree converts the parse tree into the so-called abstract syntax tree.

An example of an abstract syntax tree can be seen in figure 3.2. The $BinaryExpression$ nodes are represented by their operator and $Literal$ nodes by their value.

In their work "Compilers: principles, techniques, & tools", Aho et al. define two groups that most parsers may be sorted into [3, p. 61]. Those two types are *bottom-up* parsers and *top-down* parsers. The names of the types are based on how they construct the resulting parse tree. While a top-down parser will construct the parse tree starting from the root node, a bottom-up parser will do the opposite and begin construction from the leaves of the parse tree structure.

Aho et al. attribute the popularity of top-down parsers to the relative ease of constructing an efficient parser by hand instead of bottom-up parsing[3, p. 61].

Bottom-up parsers "can handle a larger class of grammars and translation schemes, so software tools for generating parsers directly from grammars often use bottom-up methods"[3, p. 61].

- **Recursive Descent** parsers implement a dedicated function for each non-terminal symbol of the grammar. They belong to the group of top-down parsers and are often chosen when a parser has to be implemented manually due to their simplicity. Both V8 and SpiderMonkey, the ECMAScript engines used in popular browsers and other software, employ recursive-descent parsers to parse

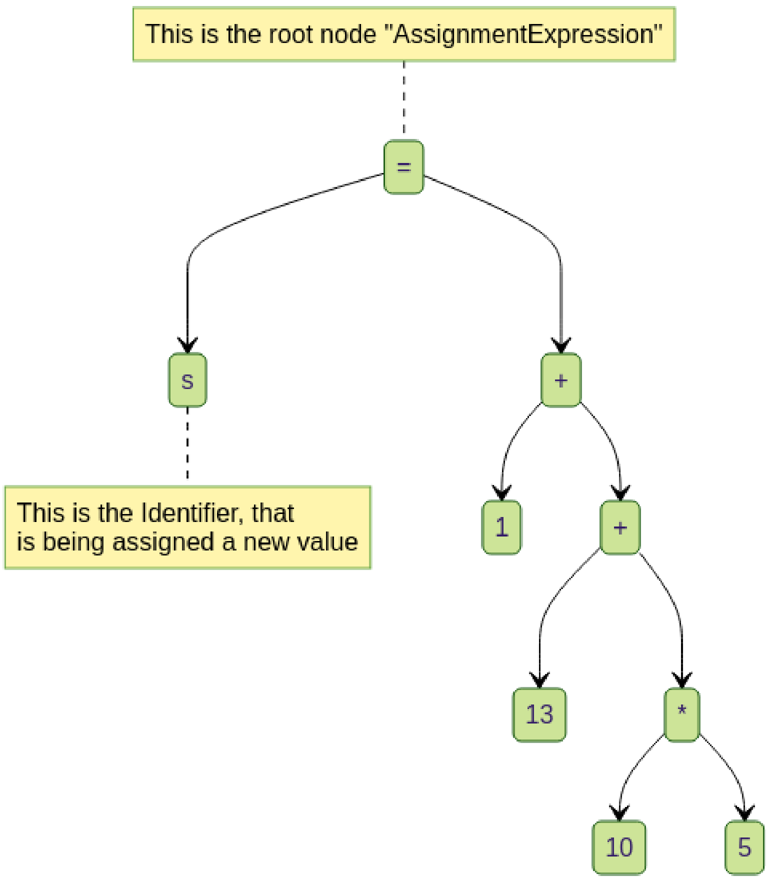
This is the root node "AssignmentExpression"
=
s
This is the Identifier, that is being assigned a new value
+
1
+
13
*
10
5

Figure 3.2: Abstract syntax tree of the expression $s = 13 + 10 * 5 + 1$

ECMAScript.

- **LL(k)** where *k* specifies the number of tokens of lookahead provided to the parser. A language from this set is usually called LL(k) grammar. It also belongs to the group of top-down parsers. They are limited because they can only parse a subset of context-free languages[13].
- **LR(k)** where *k* specifies the number of tokens of lookahead provided to the parser. They belong to the group of bottom-up parsers, and their main drawback is that manually constructing such a parser is very complicated due to the non-intuitive way they work. Compared to top-down parsing, it allows recognizing more languages at the cost of worse error reporting and complexity[13].

3.3.3 Semantical Analysis

With ECMAScript being a dynamically typed language, it is also not necessary or possible to perform type checking at this point except for trivial cases. Statically typed programming languages will use this step to enforce their typing system.

ECMAScript performs error handling during this stage. The mechanism is called "early error" and is defined per production basis. As an example, the WithStatement defines two early errors. The first is that the production is considered a syntax error if contained within strict mode code. The second is only applied under certain preconditions and specifies that it is considered a syntax error if the expression is a labeled function.

This thesis operates because only valid programs may be supplied as input. This restriction allows it to skip the semantical analysis in most situations.

3.4 JavaScript

JavaScript is a programming language derived from two commercial, proprietary implementations of a web browser-focused scripting language. The original implementation, called JavaScript, was featured in a web browser named *Navigator* by the American company Netscape. Microsoft then decided to re-implement the language under a different name, specifically JScript, in the new version 3.0 of their product *Internet Explorer*[28, p. 45].

This development prompted Netscape to seek standardization of the language under the overview of the non-profit ECMA. The name used to be an acronym for "European Computer Manufacturers Association" but was changed in 1994 to reflect the organization's global influence [15].

Despite the similar name, there is no relation to the Java programming language developed by Sun Microsystems and then later Oracle. However, the name became an issue when the standardization process began because the term *Java* was already trademarked by the Sun Microsystems corporation, which now belongs to the Oracle corporation. In addition to that, the word *JavaScript* was also trademarked by the Sun Microsystems corporation in the United States before the beginning of the standardization process in November 1996[37]. This trademark led to the new name of *ECMAScript* for the standardized version of the language[28, p. 45].

Despite the registered trademarks, the implementation of the ECMAScript standards is referred to as JavaScript, while the standards themselves are referred to as ECMAScript[28, p. 45].

This thesis will refer to the language defined by the ECMAScript standard as ECMAScript from here on out. The standard will be referred to as the ECMAScript standard or the ECMAScript specification.

3.4.1 Paradigms

ECMAScript features, as is usual for scripting languages, a multi-paradigm approach. Its typing system is dynamic and weak, with type annotations not officially supported by the ECMAScript specification. Comment-based type annotations have been adopted by some tools, as mentioned in section 2.1. There are plans to adopt optional, explicit type annotations as part of the language[34].

This thesis will also cover some of the implicit type conversions within ECMAScript.

Features from object-oriented programming can be found in ECMAScript through the recently added support for classes. Abstract classes and inheritance are supported. The type system renders the existence of interfaces meaningless as the boundaries may be broken trivially and even accidentally. ECMAScript, therefore, does not support interfaces. Presumably destined for this feature, the `implements` keyword is a future reserved word. At its core, ECMAScript is a prototyped language, however

Functional aspects of ECMAScript can be found in the associated methods of arrays. ECMAScript provides routines for mapping, filtering, and reducing this way. Other higher-order functions such as currying are supported as well.

Procedural and imperative patterns may be built using the various control-flow statements afforded by ECMAScript.

Event-driven aspects can be found when interacting with the document object model (DOM). The DOM represents the currently displayed website when ECMAScript is used in the context of a browser. It is a tree structure containing the HTML nodes as ECMAScript objects. Each node within this tree may emit events that may be listened to by attaching a function to the event. For example a button may be defined in HTML like this: `<button onclick="clickHandler()">Button</button>`.

This button will be labeled by the text “Button” and will execute the function `clickHandler()` once clicked. Event handlers may also be attached by traversing the DOM and setting the property value of, for example, the `onClick` event.

3.4.2 Structure

Like many other programming languages featuring an imperative paradigm, the ECMAScript abstract syntax tree nodes feature two subclasses. The first one is statements that regulate the intraprocedural control flow or declare names and do not evaluate a value. An example of this in ECMAScript would be the `if` statement. ECMAScript is mentioned explicitly as some programming languages, such as Rust and Python, allow using it as an expression.

The second subclass of the nodes is expressions, which, unlike statements, evaluate to a value. Expressions may be used in the place of statements using the `ExpressionStatement`. Expressions may also influence the intraprocedural control flow. For example, an expression might do this by throwing an exception implicitly. They may also influence the interprocedural control flow using the `CallExpression`, which invokes other functions. However, the control flow will always return to the calling function in the absence of exceptions.

3.4.3 Data types

The ECMAScript standard defines seven built-in data types introduced during the following chapters.

3.4.3.1 Boolean type

The Boolean type defines **`true`** and **`false`** as its only possible values[12, § 6.1.3] analog to other programming languages like Java and C++. When coerced into a numeric data type, **`true`** will be assigned a mathematical value of one. **`false`** will be given the mathematical value zero instead.

3.4.3.2 Undefined

The ECMAScript standard 12th edition defines the undefined type as follows[12, § 6.1.1]:

> The Undefined type has exactly one value, called `undefined`. Any variable that has not been assigned a value has the value `undefined`.

3.4.3.3 Null type

The null type, similarly to the `undefined` type, only has one value called **`null`**[12, § 6.1.2]. Semantically undefined represents a variable that has not yet been defined, the value of a nonexistent object property or an empty optional parameter. **`null`** is used when a reference to an object is expected[28, p.11]. Both `undefined` and **`null`** will evaluate to a boolean **`false`** when coerced to the boolean type.

3.4.3.4 String type

> The String type is the set of all ordered sequences of zero or more 16-bit unsigned integer values (“elements”) up to a maximum length of $2^{53} - 1$ elements.[12, § 6.1.4]

3.4.3.5 Number type

The number type represents a single floating-point number in the IEEE754-2019[17] double precision format[12, § 6.1.6.1]. The size of each `Number` instance is 64 bits, where one bit represents the sign, eleven for the exponent, and 52 for the fraction, as can be seen in figure 3.3.

To convert the binary format into a human-readable number, the three binary parts are combined as follows:

If the sign bit is set, the resulting number will be negative. Instead of opting for a twos-complement, the exponent is offset by 1023. Therefore, to get an exponent of zero, the exponent bits have to be set to `0x3FF` in hexadecimal or `1111111111` in binary. The base of the exponentiation is always two. The fraction part is the sum of one and the sum of exponentiating two, with the index of each bit multiplied by minus one. So to get a fraction of 1.5 the first bit of the fraction has to be set: $1 + 2^{-1} = 1.5$. This can be expressed mathematically with the following sum: $1 + \sum_{i=1}^{52} f_i 2^{-i}$ where f represents the bits of the fraction and f_n represents the nth bit of the fraction.

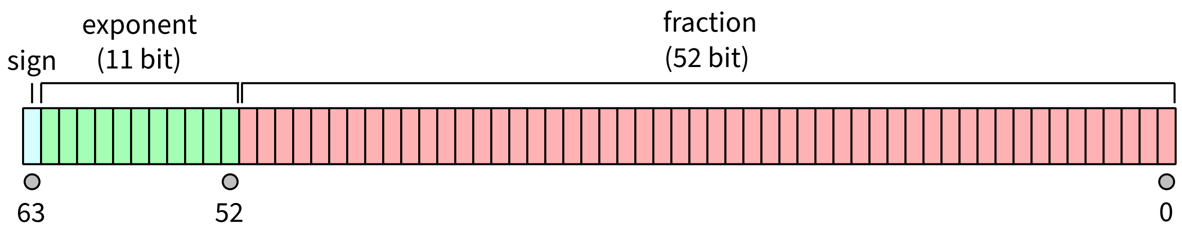

Figure 3.3: The memory format of an IEEE 754 double-precision floating-point value
Source: Codekaizen, CC BY-SA 4.0, via Wikimedia Commons

The ECMAScript standard follows the IEEE754-2019[17] standard except for the special value "Not-a-Number" (**NaN**) where the IEEE754 standard defines $2^{53} - 2$ distinct values. Instead of that, there is only a single `NaN` value. This deviation from the standard only affects the user-facing part, though. The actual layout of the NaN value is implementation-specific and may be observed using either an external tool or the built-in classes `ArrayBuffer` and `SharedArrayBuffer`[12, § 6.1.6.1]. A malicious agent may use this to construct an opaque predicate, which is explained in section 3.5.1.

The ECMAScript specification defines three distinct sets of numbers. The first one is the `Number` values, which are denoted by a subscript $\mathbb{F}$ and represent a number within the limitations of a double-precision IEEE 754 floating-point value. For example, $0.1_{\mathbb{F}} + 0.2_{\mathbb{F}}$, which results in $0.30000000000000004_{\mathbb{F}}$ due to the limitations of IEEE 754.

BigInt values, which are represented by $\mathbb{Z}$, represent positive or negative integers.

Finally, the mathematical value is the value of the previously mentioned numeric types mapped into real numbers' mathematical space. This set is denoted by $\mathbb{R}$.

3.4.3.6 BigInt

The `BigInt` type represents an arbitrary precision integer type. Because of that, no particular values are reserved like they are with `number`s `NaN`, negative zero, or `Infinity`. Operations on `BigInt` values "are designed to return exact mathematically-based answers"[12, § 6.1.6.2]. `BigInt` literal values may be specified by appending the character `n` to a numeric literal. They are implemented as binary strings of arbitrary length and support a sign by considering a sign bit set infinitely far to the left of the binary string. This out-of-band storage of the sign bit becomes essential for some bitwise operations encountered later in this thesis.

There is no maximum size defined for the `BigInt` type values in the ECMAScript specification. Still, technical limits can be reached quite easily by trying to allocate a `BigInt` value more extensive than the available physical or even virtual memory. Exceeding the available memory will result in a `RangeError` thrown in V8 and SpiderMonkey. However, this behavior is implementation-defined and not part of the ECMAScript specification.

3.4.3.7 Objects

The ECMAScript specification defines the Object type as a collection of properties[12, § 6.1.7]. These properties may take one of two forms, either a data property or an accessor property. Both forms associate a key, a `string` or `Symbol` value, with a set of boolean attributes and either an ECMAScript value or one or two accessor functions. The attributes are denoted by being enclosed in square brackets.

The first form, the data properties, associate a key with a value and a set of boolean attributes, similar to how structures work in other programming languages. This value may be of any of the ECMAScript types.

The attributes of data properties consist of the following elements[12, § 6.1.7.1]:

- `[[Writable]]` will cause attempts to set a new value to fail if set to `false`.
- `[[Enumerable]]` controls whether a `for-in` iteration will include this property.
- `[[Configurable]]` controls whether the code can delete the property, change the form of the property, or its attributes. Changing the value or setting `[[Writable]]` to `false` is excluded from this restriction.

A programmer may not modify these attributes directly. However, a programmer may modify them through built-in functions.

Accessor properties provide the functions `[[Get]]` and `[[Set]]`, both of which are optional, but at least one has to be provided for the property to be valid. These work similar to getters and setters in programming languages like Java. The main difference is that the access is handled transparently. Assigning or reading a value from an accessor property does not differ from doing the same thing on a data property. The different forms may be distinguished using the built-in function `Object.getOwnPropertyDescriptor()`.

Accessor properties also contain a set of boolean attributes. It consists of the previously introduced `[[Enumerable]]` and `[[Configurable]]` properties. The former is identical to how it is implemented for data properties, while the latter has no exemptions, unlike those implemented for data properties. Write protection must be implemented by the user within the `[[Set]]` method if desired.

Inheritance is modeled in ECMAScript using so-called prototypes. These prototypes are a property of the object and are also objects. Prototype objects may themselves also possess a prototype property. This linked list of prototypes is referred to as the prototype chain, with `null` being the root element. Suppose a property can not be found on an object. In that case, the prototype chain is traversed upwards in search of the property, similar to how virtual method tables work in programming languages like C++.

3.4.3.8 Symbol

The `Symbol` type poses, as previously mentioned, as an alternative to the `string` type for keys of an object property. “Each possible Symbol value is unique and immutable”[12, § 6.1.5].

The ECMAScript specification defines a set of well-known symbols, which are symbols that are explicitly referred to by algorithms used in the specification. These symbols are denoted using @@ as a prefix.

An example for this would be the well-known $@@iterator$ symbol, which may be set as an object property to override the default iterator of an object. This property is used implicitly by the `for-of` statement.

3.4.4 Scoping rules

ECMAScript differs in the way scoping works from comparable programming languages.

Variables may be declared using one of the three declarations kinds `var`, `let`, and `const`.

Variables declared using `let` and `const` are scoped within their respective block. Any attempt to access them before their declaration will cause a `ReferenceError` exception even if a variable with the same name has been declared in a reachable scope. This behavior can lead to some non-obvious error cases as outlined by the following code snippet.

```
const a = () => {
  console.log("foo");
};
{
  a();
  const a = () => {
    console.log("bar");
  };
}
```

Intuitively one might think that the global declaration of `a()` may be invoked here, but a `ReferenceError` is thrown instead. According to some sources, this phenomenon is called the "temporal dead zone"[39] [25] [1]. The ECMAScript specification does not acknowledge this term.

Specifying an initial value for variables declared as `let` is optional. `undefined` is used as a default value for `let` variables that have been declared but not yet initialized.

Variables declared as **`const`** must specify an initial value. Not initializing it is considered a syntax error. Attempting to assign a value to a variable, which was declared as **`const`** outside of the initialization, will cause a `TypeError` exception.

`var` declarations, on the other hand, are function scoped instead. If a variable is declared using `var` outside of a function, it is considered a global variable instead. Variables declared in such a way are also hoisted. This term means that the declaration is virtually moved to the beginning of the scope but not the initialization. Accessing a variable declared as `var` before it has been declared will not cause an exception to be thrown. Its value will be `undefined` until it has been declared and initialized.

An example for the somewhat non-obvious behavior of `var` declared variables is shown in the following snippet.

```
function foo() {
  var x = 6;
  {
    var x = 7;
    console.log(x);
  }
  console.log(x);
}
```

This code will print the number seven twice as the second declaration of `x` reassigns the existing variable instead of creating a new one.

3.4.5 Strict mode

The ECMAScript standard defines a strict mode, which disables some "error-prone features"[12, § 4.3.2] and provides "enhanced error checking"[12, § 4.3.2]. Some examples for features, that are disabled in the strict mode, will be shown as they are encountered during this thesis along with reasoning as to why they are error-prone. "A complete ECMAScript program may be composed of both strict mode and non-strict mode ECMAScript source text units"[12, § 4.3.2]. In order to conform to the ECMAScript specification an implementation must implement both the strict and non-strict mode of ECMAScript as well as the combination of the two within a composite program[12, § 4.3.2].

3.4.6 Abstract operations

The ECMAScript specification defines several abstract operations, which are used to specify the semantics of the language[12, § 7] [12, § 5.2.1]. These abstract operations will be described as they are encountered throughout this thesis. An abstract operation consists of a name, a list of parameters, and a list of steps applied to reach a result. Some of the parameters may be marked as optional. An example of abstract operations, which will also be explained in detail in later chapters, are the ones concerning type conversion. They may also be treated as polymorphically dispatched methods.

Abstract operations are not part of the ECMAScript language and may therefore not be invoked by ECMAScript code.

3.4.7 Important built-in methods and objects

ECMAScript provides a plethora of built-in objects and methods that will become important later on in this thesis. The most important ones will be introduced in the following along with an explanation as to why they are important. An important issue with these methods, that will be discussed in detail in section 5.2.1.4, is that they may be overridden by the application. A function such as `Math.random` may therefore be overridden with a benign change, such as a better pseudo random number generator, or malicious functions as to obfuscate the control flow of the application. The dynamic nature of ECMAScript makes it exceedingly hard to track these overridden functions, objects, or constants using static analysis.

3.4.7.1 `eval()`

`eval()` allows a programmer to evaluate expressions and statements at runtime. The code is supplied to `eval` as a parameter of the string type (see section 3.4.3.4). If the parameter is of another type, the parameter will be returned by `eval()` without evaluating any code.

A quirk of this function is that `eval()` will use the local scope if invoked directly. If `eval()` is invoked indirectly, for example, through a variable, it will use the global scope instead.

The use of `eval()` is heavily discouraged for both security and performance reasons. When not in strict mode, `eval()` will operate on the local scope and, as such, may define new variables at runtime, making static analysis hard if not impossible. If strict mode is enabled, the variables created within `eval()` are only valid within the code passed to `eval()`.

Utilizing `eval()` hampers modern ECMAScript engines as they implement just-in-time compilation, turning ECMAScript code into native code for a considerable performance gain. The compilation process has to be started anew if `eval()` is encountered as it might break assumptions about the previously generated code.

3.4.7.2 Function

Programmers may use the built-in `Function` class to write dynamic functions compiled from a supplied string. It works similarly to the previously discussed `eval()` function with two significant differences. The first difference is that this method is limited to creating functions. Unlike `eval()`, it may not create code that would run in a local context. Generated functions live in the global scope only.

3.4.7.3 `JSON.stringify()`

`JSON.stringify()` is a built-in function in ECMAScript that a programmer may use to convert an ECMAScript value into the ubiquitous JavaScript Object Notation (JSON) format. String values are escaped according to the JSON specification, matching how quotes are escaped within ECMAScript. This thesis uses this as a shortcut to escaping strings that are newly inserted into the abstract syntax tree.

3.4.8 Automatic semicolon insertion

As with most other C-like programming languages, ECMAScript expects statements to be terminated by a semicolon. Unlike the other programming languages, it is valid to omit semicolons in some situations[12, § 12.9].

The ECMAScript defines three rules where semicolons are inserted automatically.

The first rule is that a semicolon will be inserted if “a token [...] is encountered that is not allowed by any production of the grammar”[12, § 12.9.1] and one or more of the conditions apply:

- One or more LineTerminator characters separate the token from its predecessor
- The token is a closing curly brace
- The token is a closing brace

The second rule applies if the end of the token stream is reached and the goal nonterminal was unable to be parsed from it, a semicolon will be inserted automatically.

The final rule specifies a list of restricted productions that explicitly include within their definition that one or more LineTerminators may not be inserted between the tokens that make up the production. An example of this can be found in the following code snippet.

```
return
1 + 2
```

As the ReturnStatement is included in the list of restricted productions, it will cause this code to be parsed as two separate statements, as may be seen in the following code snippet, where the automatic semicolon insertion has been applied.

```
return;
1 + 2;
```

It becomes apparent that the semantics of the code differs from what might have been the programmer’s intent. The function will return `undefined` instead of the `number` value 3.

3.5 Obfuscation

The term obfuscation describes purposefully making a piece of software harder to understand for humans. This process may take different forms based on the software. There are several reasons why a programmer may want to obfuscate their software. These include the protection of intellectual property from reverse engineering, the evasion of malware detection, or, as discussed in a later chapter, a reduction in file size.

For software compiled to machine code or bytecode, obfuscation may be applied before the compilation is done or afterward. If obfuscation is added to a program before the compilation process, the compiler may undo at least some obfuscation via its optimizations.

Obfuscation of machine code may take many different forms. A commonly used way of obfuscating software is the use of so-called packers. Packers are programs that accept other programs as input and apply various obfuscation techniques to their input to generate a second, obfuscated program that is semantically equivalent to the input program.

Software deployed in its source code form and interpreted or compiled by the user can only reliably be obfuscated on the source code level.

This thesis will refer to a program that obfuscates other programs as obfuscators. Programs that attempt to reverse the effects of obfuscation will be referred to as deobfuscators.

3.5.1 Opaque predicates

This thesis will refer to values that a deobfuscator may not determine statically at various points. An example of this is opaque predicates.

Collberg et al. define opaque predicates in their 1997 paper “Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs” as variables or predicates that have “some property q which is known a priori to the obfuscator, but which is difficult for the deobfuscator to deduce”[10].

As introduced in section 3.4.3.5, an attacker may also use these predicates in ECMAScript to harden a piece of software against deobfuscation. The following example introduces one such predicate that an attacker may use to identify the currently running JavaScript interpreter without explicitly doing so.

Suppose that a function `printRepresentation(value)` accepts a parameter `value` of the `number` type and prints the hexadecimal byte representation of the underlying value. In this example, the two ECMAScript engines to be differentiated were the version of SpiderMonkey shipped with Mozilla Firefox 95 and the version of V8 shipped with Chromium 96.0.4664.93.

In SpiderMonkey the expression `printRepresentation(1**Infinity)` will print `0x7ff8000000000000` while in V8 it will print `0xfff8000000000000` instead. An attacker may use the result of this computation to differentiate the program’s behavior depending on the JavaScript engine currently in use.

However, this implementation-specific detail implies that a deobfuscator that aims to conserve the semantics of the application may not fold all constants that evaluate to NaN without prior knowledge of which engine the program is targeting or wants to target.

The example that was introduced earlier may look like this in practice:

```
const buffer = new ArrayBuffer(8);
const float64 = new Float64Array(buffer);
float64[0] = 1**Infinity;
const dv = new DataView(buffer)

if(dv.getInt8(7) < 0) {
    console.log("Hello V8!");
} else {
    console.log("Hello Firefox!")
}
```

ArrayBuffer provides the backing storage of eight bytes, while Float64Array needs to store a single number type value. 1**Infinity uses the exponentiation operator to calculate $1^{+\infty}$ which is, depending on the sets chosen, indeterminate in mathematics. At the time of the standardization of ECMAScript, the IEEE754 standard did not define the result of the operation. The 2019 revision[17, p. 63] defined it as having the result $+1_{\mathbb{F}}$.

Due to this operation being undefined before the 2019 revision, ECMAScript defines the result as NaN[12, § 6.1.6.1.3]. This particular case was kept even though it is defined in the IEEE754-2019 standard to preserve the arithmetic behavior of the language. Due to this discrepancy with the standard, the implementors of ECMAScript engines have to add a special case in their engines because it can not necessarily be handled by the host CPU natively. This historical detail would explain why the internal representations of NaN differ between different JavaScript engines. A DataView instance is then used to inspect the underlying ArrayBuffer. It is similar to a pointer cast in languages like C and C++ and is used in this case to read the seventh byte of the number value.

Instead of just printing the platform, an adversary may use the sign bit that differentiates the results to calculate a decryption key that may only be calculated on the specific platform the program was intended to run on. This information is known to the obfuscator a priori as it can be set by the user operating the obfuscator. Still, the deobfuscator needs user input and therefore requires information manually gathered by a human or needs to rely on heuristics to identify the targeted platform of the program.

4 Analyzing the targets

To begin deobfuscating ECMAScript, it is necessary first to analyze the types of obfuscation that may be encountered. The obfuscation software in question is the software *javascript-obfuscator*[35], which is available under the BSD 2-Clause “Simplified” License. It was chosen due to its popularity and plethora of settings, influencing the obfuscated code. In addition to that, a few samples were taken from malware written in ECMAScript.

The following chapter will analyze the javascript-obfuscator software and its available settings. Furthermore, it will include solutions for undoing the applied transformations if they are available. Examples of the settings in action are given where appropriate to illustrate the settings’ impact further.

The next chapter will focus on deobfuscation techniques from the malware samples. The contents of that chapter will focus more on solutions as the samples do not include any information on the obfuscation software used.

4.1 javascript-obfuscator

The following chapters describe the available settings in javascript-obfuscator and include solutions if available. Settings that do not directly influence the generated code or merely influence whitespace are not included in this list for reasons of brevity.

4.1.1 Control Flow Flattening

Control flow flattening was first introduced and demonstrated on the language C by Wang et al. in their 2000 paper “Software Tamper Resistance: Obstructing Static Analysis of Programs”. In the paper, the authors describe an obfuscation technique applied on the basic block level.

The transformation introduced by the paper works in two steps. The first step transforms all “high-level control transfers”[40, p. 5] into equivalent “if-then-goto constructs”[40, p. 5]. In the second step, the goto parts of these newly created constructs are modified in such a way that their control flow target is no longer static. “This is done by loading from the content of some data variable location instead of a direct jump address”[40, p. 5].

In the language they’ve chosen to demonstrate this technique, C, this can be achieved by rearranging the control flow constructs as cases of a switch construct. The condition on which the switch construct decides the branch to be taken is then recalculated at the end of every case of the switch construct.

The authors then argue that statically building the control flow graph is then dependent on whether the branch variable can be determined. However, the process of determining the value amounts to a “classical use-and-def” data-flow problem, as they claim.

Furthermore, they expand upon this by introducing alias detection as a data-flow problem, which is the problem of two or more identifiers pointing to the same memory. This problem is undecidable in the presence of “general pointers and recursive data structures”[40, p. 8].

Statical analysis may be slowed down further by adding dummy cases to the switch construct. Given the nature of the predicate, they may not be eliminated unless the predicate can be determined never to take the value that would trigger the execution of any of the dummy cases.

This can be similarly applied to ECMAScript as the obfuscation tool already does. However, a problem that has to be overcome is the lack of a goto statement in the language defined by the ECMAScript specification. Any program that has been obfuscated using this technique will incur a heavy performance loss, especially in ECMAScript engines that utilize just-in-time compilation. The same issues faced by a deobfuscation program also apply to an ECMAScript engine.

The obfuscation software implements control flow flattening on both expressions and statements instead of basic blocks and will use an object literal instead of a switch case. It also explicitly warns the user of the performance impact of this obfuscation technique and accepts a numeric modifier between zero and one which determines the chance of a statement being added to the control flow flattening.

These statements and expressions will be assigned a random key and move as the value of said key within an object literal. Duplicate usages of e.g. a function or a string literal will receive the same key. A call like `console.log("Hello World!")` may be transformed by the control flow flattening to look something like the following code snippet.

```
var obj = {
   'bar': 'log',
   'foo': 'Hello World!',
};

console[obj['bar']](obj['foo']);
```

This example also shows one of the possibilities of obscuring the access of object properties. In addition to simply hiding the member access behind an expression that may not be evaluated statically, it is also possible to define a property proxy for objects. This proxy can then intercept any property access to the object and return different values based upon arbitrary conditions.

The decision has been made to exclude this transformation from the scope of this thesis. This thesis focuses on static analysis with a probabilistic approach at times.

4.1.2 Dead code injection

The dead code injection setting adds randomly generated code fragments to the program and also ships with a percentage modifier that controls at which rate dead code is added. These code fragments may never be executed and are, as such dead code. Alternatively, they do not do any meaningful operations if they are invoked. Enabling this setting also forcefully enables the string array setting, which will be discussed later in this thesis. This setting is enabled so that the newly added dead code may act as a proxy between the consumers of the string array and the string array.

The deobfuscation program already tackles this obfuscation technique by repeatedly applying the inlining, constant folding, dead code elimination, and constant evaluation passes.

4.1.3 Debug protection

ECMAScript prominently features a debugger statement that allows programmers to define a breakpoint within their application from the source code. It is similar to manually causing a software interrupt using `int 3` in x86 assembler. The protection itself aims to disable the functionality to hamper dynamic analysis of the obfuscated application. This is accomplished by repeatedly calling dynamically generated functions containing the `debugger` statement. The javascript-obfuscator manual recommends an interval between 2000 and 4000 milliseconds for the calls.

The impact on regular users is negligible as the `debugger` statement has no effect if a debugger is not available or enabled.

Since this thesis focuses on static analysis, it does not actively remove this technique. The deobfuscation software adds the code for calling generating the functions from a template. This, however, means that the code added from the template is also subject to the other, partly randomized, obfuscation techniques. A simple search and replace operation can, as such, not remove the added code.

However, one possible solution would be to patch the constructor of the built-in `Function` class in such a way that it becomes unusable for the target program. This may take different forms, but the trivial solution would be to set `Function.prototype.constructor` to a function that returns a function that logs the code it was supposed to run in an unpatched environment. An allowlist could then be built manually by the user not to impact other parts of the program that rely on the built-in Function class.

Since built-in objects that are writable may be modified arbitrarily without the possibility of restoring them short of restarting the ECMAScript engine, it becomes impossible for the obfuscator to restore the patched constructor to its original value.

4.1.4 Disable console output

This technique also aims to slow down the dynamic analysis by patching the methods of the built-in console object to do nothing instead of writing to the console.

As this technique also focuses on dynamic analysis and not static analysis, it is also considered outside of the scope of this thesis and especially the accompanying implementation. The code necessary for this is again introduced from a template and, as such, is also subject to the other obfuscation techniques.

A user trying to re-enable the functions of the console object is confronted with the same problem that was used as a solution in the previous chapter. Fortunately, the deobfuscation software could save the original values of the functions before they are replaced and restore them either after the replacement, which requires knowledge of where the replacement is done, or by restoring the original values whenever they are called.

Another approach would be to make the properties of the console object read-only before the control flow reaches the obfuscated code so that they may not be replaced at runtime. Attempting to overwrite a read-only property will silently discard the new value. A simple implementation of this technique may be seen in the following code snippet.

```
Object.defineProperty(console, "log", {
    value: console.log,
    writable: false
});
```

4.1.5 Domain lock

The domain lock functionality allows the user of the deobfuscation software to limit the execution of the users' program to a specific set of domains. The prevention of intellectual property theft is listed as a reason for employing this technique. If the program is executed on domains other than the ones specified by the user, the script will forcefully redirect the user

In addition to focussing solely on the browser environment, it is yet again a technique aimed at hampering dynamic analysis and is considered outside of the scope of this thesis. It is again sourced from a template as with the previous techniques.

The template in question reads the current domain using the property `location` of `window.document` and validates it using a regular expression. The second technique suggested for defeating the disabled console output technique may not be applied here as the location property of the window object is read-only. If it is attempted, a `TypeError` exception is thrown.

Suppose the reverse engineer knows the original domain, which the program was locked to. In that case, they may patch the constructor of the built-in `RegExp` class and intercept all regular expressions that are being constructed. Using this knowledge, the reverse engineer may then develop a patched version of the constructor that only replaces the regular expression used by the domain locking technique with the ones desired by the reverse engineer. Alternatively, the reverse engineer may use the stack trace of the patched constructor to trace the location of the domain locking code within the obfuscated program to remove it.

4.1.6 Identifier renaming

Renaming identifiers is a powerful technique employed by the obfuscator as identifiers may be used as clues by a reverse engineer as to what a variable, function, or class may be used for. The obfuscation program provides several generators for new identifiers:

- New identifiers may be chosen from a user-supplied dictionary. This can not only remove any clues that a reverse engineer could have gained from the original identifiers but may instead

send a reverse engineer down the wrong path by supplying false clues.
- The default setting for new identifiers is to rename all identifiers to an underscore followed by a hexadecimal number, including the `0x` prefix.
- The user may also choose to use short single or double letter names for renamed identifiers.

The original identifier names are lost forever and may not be restored through conventional operations on the abstract syntax tree. Some clues as to what the original names were may be gathered from surrounding strings, but this is far from reliable and may be hampered by the other obfuscation techniques.

This thesis includes *Context2Name*, which attempts to restore identifiers using a machine-learning model trained on a large set of ECMAScript code. The surrounding tokens of an identifier are used to attempt to generate a meaningful identifier.

The user may also separately enable the renaming of global variables and functions, but this setting gives an express warning that it may break the supplied program. This warning is added because statically tracing variables in ECMAScript is an imperfect process as some runtime information may be necessary to accomplish it fully.

A similar warning also adorns the setting that enables the renaming of object properties. The same issues as with global variables apply.

4.1.7 Numbers to expression

The "numbers to expressions" setting makes the obfuscation software replace all numeric literals in the program with an equivalent expression that evaluates to the same value. Special care is taken not to make the calculation overflow with regards to the upper limit a value of the type `number` may represent.

As this is a relatively trivial transformation with no information irreversibly lost it may be entirely removed. The deobfuscation program described by this thesis applies the constant folding described in section 5.2 to remove it entirely.

4.1.8 Self defense

If enabled, the self-defense setting adds a piece of templated code to the program. This code will check if there has been an attempt to reformat the source code to make it more readable, and if that is the case, the program will cease to function.

This technique works by abusing the fact that a function may read the source code of arbitrary ECMAScript functions within its scope. This is accomplished by calling the member method `toString()` of the function in question. This string representation is identical to the one parsed by the engine and, as such, also contains the newly inserted whitespaces. The actual self-defense is then happening using a regular expression, which aims to trigger the so-called catastrophic backtracking issue. This issue is a denial-of-service attack on regular expression engines, and the presence of many space characters will trigger it. Enabling this setting also forcefully enables the setting to remove extraneous whitespace.

As with the previous technique that employed regular expressions, this may also be bypassed by patching the constructor of the built-in `RegExp` class to monitor or intercept the malicious regular expressions created by the obfuscated code. An alternative version of the template employs the built-in `Function` class and the built-in `eval()` method. However, the code passed to those functions will still trigger the patched constructor of the `RegExp` class. While this technique will trigger after the deobfuscation program has run, it does not affect the actual deobfuscation process and is therefore not included in the deobfuscation program.

4.1.9 Simplification

The "simplification" setting makes the obfuscation software perform several transformations that shorten the code and make it harder to understand for a human. The first transformation included in this is the merging of variable declarations. An example of this would be the following code snippet:

```
let a = 2;
let b = 4;
```

The obfuscation software will merge these two declarations, turning them into `let a = 2, b = 4;`. A merge can, of course, only be done if the kinds of declarations are compatible with each other. Merging a declaration of the **`const`** kind with one of the `let` kind could lead to a runtime error if the resulting kind is **`const`** and the variable formerly declared as `let` is attempted to be written to after its initialization. One solution for this problem would be to downgrade the variable defined as **`const`** to a `let` binding. However, this downgrade could be used by the input program to detect whether it has been obfuscated by attempting to write to the formerly **`const`** variable and catching the resulting `TypeError`. If the exception is not thrown, this is a sign that the application has been obfuscated. Since this transformation does not only make the program more challenging to read but also smaller in terms of file size, it is also often employed by compression software.

The deobfuscation software may undo declaration merging by splitting the declaration back into separate declaration statements.

Another transformation employed by the simplification setting is the dissolution of the optional block statements around **`if`** statements if they only contain a single statement. An example of this behavior would be the following snippet:

```
if(a > b) {
  console.log("a is larger than b");
} else {
  console.log("b is larger than a");
}
```

The obfuscation software will turn this **`if`** statement into the equivalent but harder to read and maintain version that looks like the following snippet.

```
if(a > b) console.log("a is larger than b");
else console.log("b is larger than a");
```

The deobfuscation software can undo this by re-adding the block statements for the if and else blocks. The braces should be added back in all cases, even if the input source code omitted the braces, as the variant with explicit block statements is the recommended way, as indicated by its inclusion in the popular ECMAScript linter ESLint[11].

A third obfuscation introduced by this setting is the merging of multiple expression statements into a single expression statement using the comma operator. An example of this behavior is shown using the following code snippet as an input to the obfuscation software.

```
function foo() {
  console.log("bar");
  return Math.random();
}
```

The resulting code, after this transformation, merges the two expression statements into one.

```
function foo() {
  return console.log("bar"), Math.random();
}
```

The return value is kept intact as the last value of the comma operator is the value of expression as a whole. Unlike the simplification introduced before this one, the comma operator provides legitimate uses that do not worsen the readability. One such use case would be using multiple variables within a `for` statement. Therefore, the deobfuscation software will only separate expressions within a comma operator expression if not included within a `for` statement.

Boolean literal values like `true` are converted to `!![]`. This technique is not yet a part of the simplification setting but is included in this chapter nonetheless as it is closest in purpose. The deobfuscation software includes special peephole rules to undo these transformations. Boolean literals are easier to understand than following the complicated coercion rules of ECMAScript, so these may be applied in every case. This transformation is done using constant folding, which is explained in further detail in section 5.2.1.6.

While not explicitly included in this setting, the transformation of member accesses using an identifier for the property to a member access using a string is also included in this category. This transformation will for example change `console.log()` into the semantically equivalent `console['log']`.

4.1.10 Split strings

The “split strings” setting is equivalent to the “numbers to expressions” setting for the `string` type values. The main difference is that only a single binary operator accepts `string` values as an input while producing `string` values. The operator is the binary plus operator, which will coerce its operands into’ string’ values if at least one operand is a string. The plus operator will then concatenate both values.

If this setting is enabled, the obfuscation software will split string literals into smaller parts. An example of this would be the string literal `"foobar"`. The obfuscation software may turn this into `"foo"+ "bar"`.

As long as both operands of the plus operator remain in their literal form, it is trivial for the deobfuscation software to merge them back together. Merging every occurrence of this may prove counterproductive as the string might've been split in the input source code to meet a limitation on line length. Merging all strings and running a code quality tool like ESLint on the deobfuscated code may mediate this issue.

4.1.11 String Array

The "string array" setting provides another technique for obfuscating the usage of strings. It will copy some string literals from their place within the input source code to a global array if enabled. The literal is then replaced with a member access expression on the global array with the index of the string literal within the global array. These indices may take the form of hexadecimal literals or hexadecimal numbers within a `string` value. The latter will be parsed as a `number` value before being used as an index. The obfuscation software may use these two methods simultaneously within the same program. The user of the obfuscation software may set a percentage that decides the chance at which a string is copied to the global string array.

Even though this obfuscation might look trivial to undo, a few issues prevent it. The difference between inlining literal values within (global) variables and inlining literal values from a global array is that some methods may mutate the array in place before the index is read. Therefore, a deobfuscation program would have to trace all paths starting from the declaration and initialization of the array to each member access expression. While this may be feasible for small applications, it will lead to a phenomenon known as path explosion if applied to more extensive programs. Considering the nature of obfuscation programs to inflate code artificially and, therefore, the possible paths the control flow could take along the way, it would be unfeasible to implement this, especially within the scope of this thesis. This problem is further complicated because functions that cause an in-place mutation of the array may themselves be obfuscated in a way that is not visible to the deobfuscation program.

In addition to the previously described complexity, the obfuscation software offers additional settings to harden the array accesses further.

Among these settings is the option to replace the numeric or `string` values used to index into the global string array with calls to functions. The parameters used in these calls are chosen so that the function will determine the original index without having it directly in the function call's parameters. These function calls have a varying but configurable number of parameters, including unused parameters. These unused parameters serve as clutter to make the functions harder to read and make it harder for a deobfuscation program to optimize the source code. Dead parameter elimination can be tricky in some contexts, as the parameters may be unused in the base method of a class but will be overwritten by an inheriting class.

The global string array may also be randomly shuffled or rotated at the beginning of the program. This technique further hampers an automatic deobfuscation as the shuffling and rotating is essentially a black box due to the in-place nature of these mutations. As the code for the shuffle and rotation routines will also be obfuscated, a reverse engineer could do a preliminary run of the deobfuscation program. Afterward, they may manually analyze the code responsible for the mutations and apply the transformations to the array statically or dynamically. They may then replace the original string array with the manually calculated one.

Another technique employed by the obfuscation program is the encoding of entries of the string array. The available encodings for this are Base 64 and RC4, along with the original plain text version. The strings within the array may be encoded in a heterogeneous way. Some may be encoded using RC4, while other values within the array are encoded using Base 64. While Base 64 is an encoding, RC4 is a stream cipher, which requires recovering the encryption key before decoding the value is possible. An issue with Base 64 is that the obfuscation program does not opt to use the built-in function to convert between plain text and Base 64 but instead implements the routine using a template. This custom implementation allows the obfuscation program to choose an arbitrary alphabet for the Base 64 encoding. Both methods, therefore, require the recovery of a critical value. Both values are critical for decoding entries from the string array, which implies that they may not be themselves included in the string array.

4.2 Malware samples

Malware samples provide a good way of acquiring different obfuscation techniques as their motivation to hide the code is two-fold. Malware authors tend to obfuscate their code to evade traditional signature-based malware detection, where patterns of text or bytes found in other malware are added to a signature database. New files will trigger an alarm if a file matches any signatures. Another reason malware tends to be obfuscated is to hamper reverse-engineering efforts. These efforts may be made by good actors seeking to defuse the threat or other malware authors seeking to copy its techniques.

The samples discussed below were, according to their description, found in the wild between 2015 and 2019. Their relatively old age makes them negligibly dangerous to a modern system running an up-to-date operating system.

The thesis will mention why a sample was picked to be analyzed and the description of the sample. The repository contains over 40.000 entries, which means that this thesis can not analyze every sample individually. Any names given to the samples are purely descriptive and do not reflect any official naming.

4.2.1 Russian malware sample

The sample discussed in this section is called “Russian malware sample” because its filename is in Cyrillic characters and translates to “Order information” according to automated translation software. It contains a few obfuscation techniques that javascript-obfuscator is not utilizing, which qualify it to be analyzed within a dedicated chapter. This thesis chose it because it was previously used as a sample in SAFE-DEOBS, which was discussed in section 2.4.

These techniques will be illustrated using a snippet from the obfuscated code.

```
function kKP(Oey)
{
        var Rm = "charA";
        var iMG = "t";
        var ne = Rm + iMG;
        return ne;
}
```

The above snippet is merely one of several functions within the obfuscated code, which calculate a `string` value that is subsequently returned to the caller. The calculation of the string value itself is trivially done using the previously discussed constant folding. Applying the mentioned techniques results in a function that looks something like the following snippet.

```
function kKP()
{
        return "charAt";
}
```

These functions essentially wrap string literals and worsen the readability. The deobfuscation program can determine that the value of the return expression is side effect free in a static manner. It may, therefore, simply copy the value to all call sites that see this definition of the function. This solution is discussed in further detail in section 5.5.2.

Another technique used by this malware sample may be seen in the following snippet.

```
var fHC=String.fromCharCode(6688/88+0);
nDO = fHC + String.fromCharCode(2600/52-0);
vM = nDO + String.fromCharCode(736/16+0);
```

The declarations of the variables `nDO` and `vM` are not included as there are no separate declarations. This implicit declaration is possible because assigning a value to a non-existent identifier will create a new variable within the global scope in ECMAScript. However, the primary technique in this snippet is using built-in functions to obfuscate string constants. The expressions used as parameters evaluate to $76, 50$, and 46, respectively. They are then translated to the ASCII characters with equivalent indices. The obfuscation program reverses this technique by replacing some invocations of built-in functions with their result using the method described in section 5.5.1. This constant evaluation is limited to functions that have no visible side effects.

Another phenomenon may be observed in this sample, although not strictly an obfuscation technique. It may be seen in the following snippet of code. To showcase the phenomenon better the code has been deobfuscated and manually shortened.

```
var w = new ActiveXObject("Scripting.Dictionary");
w.Add("a", "b");
if (w.Exists("a")) {
```

A new `ActiveXObject` is being created. `ActiveXObject` is a built-in class in Microsofts proprietary ECMAScript implementation JScript. A programmer may use it to interact with the operating system directly. It is used as an opaque predicate here. A new dictionary instance is created. Its standardized counterpart is the ECMAScript built-in `Map`. A value for the key `"a"` is then assigned. The subsequent `if` checks whether a value for the key `"a"` exists, which will continually evaluate to `true`. The deobfuscation program could handle this case using heuristics, but a more generalized solution is not feasible.

However, the deobfuscation has already solved another problem within this snippet. Namely, that `ActiveXObject` is not referred to directly but instead returned as a value from a function. None of the currently existing solutions account for the case of a non-standardized built-in object being returned. This optimization should be safe in the same cases as regular identifiers are. However, as they are never declared, they will not be inlined by the existing mechanisms. This case is handled in section 5.7.

4.2.2 Static switch sample

This thesis chose the following sample because it was previously used as a sample for the case study in SAFE-DEOBS, which was discussed in section 2.4.

The samples defining features are an overabundance of dead code and inlineable functions. The reader may find an example of these functions in the following code snippet.

```javascript
function elypa() {
    var egnoqqy = null;
    return egnoqqy;
}
```

The previously discussed inlining and constant evaluation mechanisms already cover these obfuscation artifacts.

A more exciting technique is found in its use of switch statements. They take the form of a switch statement, whose condition is an inlineable variable. All cases defined on the switch statements use a literal value or one of the reserved identifiers as test expressions. This configuration allows the deobfuscation program to determine which case will be taken statically. Special care has to be taken to account for the fall-through nature of the switch cases in ECMAScript. This deobfuscation technique was already showcased in SAFE-DEOBS[14, p. 4] and will be discussed in detail in chapter 5.9.

4.2.3 Array literals sample

This thesis chose the following sample because it showcases an edge case for solving a more complex issue using a naive approach.

The sample contains many globally declared variables that are initialized using an array filled with literal values. An example of those arrays is `var foo = ["tr", 84];`. The array's name has been shortened as it was previously 99 characters long and nonsensical.

Most of these arrays are being used only once. This behavior is unlike the one discovered in javascript-obfuscator in section 4.1, which composes all strings within a single global array. Being side effect free and only used once allows the inlining mechanism to come into action and inline these arrays to become array literals within the code. This transformation allows the deobfuscation program to evaluate the array access operator statically. An example of this using the previous example would be `["tr", 84][0]`, which the deobfuscation program may transform to `"tr"`.

ECMAScript defines two different ways of initializing an array. The first one is the `ArrayExpression`, which encloses zero-or-more elements in square brackets. The second one involves invoking the built-in `Array` object to construct the array. Apart from the slightly different parameters that a user may pass to the built-in object, the main difference between these paths is that an attacker may not overwrite the former by patching a prototype object. The `ArrayExpression` does not invoke a constructor[12, § 13.2.4.1]. Therefore, the transformation is safe to be applied if the expressions within the array are side-effect free.

5 Implementation

In this chapter, the necessary steps to deobfuscate ECMAScript will be discussed. This chapter will also include some technical details to explain some decisions taken during this process.

5.1 Design choices

This chapter will focus on the software design choices taken while designing the deobfuscation program.

5.1.1 Parsing infrastructure

At the foundation of the program is a robust open-source ECMAScript parser. Since it is in the best interest of obfuscation programs to utilize quirks and edge cases, it is imperative to select a parser that strictly adheres to the specification with optional support for well-known deviations from the specification. As designing and implementing a dedicated parser for this thesis would have exceeded the scope of this thesis, the decision was taken to embed an existing parser instead. The parser to be used was decided before the programming language of the deobfuscation program was chosen.

A prototype implementation was done in the Rust programming language. The static and strict typing discipline of that programming language made some aspects of this thesis exceedingly complex to implement. The Rust programming language struggles with recursive data structures due to its strong notion of data ownership. While this would not have made implementing all aspects of this thesis impossible, it would've increased the required time for implementation.

ECMAScript, the only readily available scripting language in web browsers, inspired other languages to be compiled to ECMAScript instead of being interpreted directly or compiled to machine code. These languages are widely popular, which influenced the availability of tools that operate on ECMAScript source code.

A community standard within this space is the ESTree specification, which a later chapter will explain more in-depth. It standardizes the structural layout of the abstract syntax tree and allows for interoperability between tools. Support for this format is a critical requirement for a parser.

The prevalence of self-hosted parsers supporting the ESTree spec leads to only ECMAScript / TypeScript parsers being considered.

The first part of the selection was done using a simple smoke test. The task was parsing an obfuscated malware sample.

The first parser analyzed is called *acorn*. It is freely available under the MIT license. Attempting to parse the malware sample yielded a `SyntaxError` because of a duplicated identifier violating ECMAScripts scope rules within the malware sample. This error was indicative of the parser performing scope analysis in addition to the parsing it was supposed to be doing. The obfuscated malware sample was designed to exploit this fact as the definition was part of a branch that is never executed or parsed by a real-world system in the first place. No obvious switch in the configuration of *acorn* was provided to disable this behavior.

The second parser analyzed is called *esprima*. It is also freely available but under the BSD license instead. Attempting the smoke test failed at first because of a strict mode violation but enabling the tolerant mode made the parse succeed.

The final parser analyzed for this thesis is called *meriyah*. It is freely available under the permissive, open-source ISC license. The smoke test parsed without having to set any additional flags.

Having established that two of the analyzed parsers can parse the malware sample, a secondary criterion had to be utilized to select a parser. Both meriyah[31] and esprima[32] publish an automated benchmark suite, which allows running multiple parsers on the same target and timing the execution. These benchmarks are implemented as websites, allowing running these benchmarks quickly on both major ECMAScript engines. The benchmark site of meriyah compares meriyah with Cherow, which was excluded from the selection due to its apparent abandonment, acorn, and esprima. Meriyah consistently outperforms the competition in the benchmark, requiring only about half the time of both acorn and esprima.

The benchmark site of esprima compares esprima with acorn, among others. esprima outperforms the competition in these benchmarks. Both benchmark sites feature “React 0.13.3” as a test program for the parsers. Even though meriyah is not featured directly on the benchmark site of esprima, it may still be compared using the timings for the “React 0.13.3” run. The results for this specific test on the meriyah benchmark match the results of the benchmark site of esprima.

The superior benchmark results and the effortless passing of the smoke test made meriyah a sensible choice. However, if meriyah ever ceases development, the adherence to the ESTree standard can be used to replace it as a parser easily.

5.1.2 ESTree

This project uses a format for representing the ECMAScript abstract syntax tree known as ESTree, which was in the past the format used by the SpiderMonkey ECMAScript engine that ships with the browser *Firefox*. It has since been deprecated from the ECMAScript engine. However, the community has adopted it as a standard, which is still updated to reflect changes in the ECMAScript standard[7].

All available nodes within the specification have a `type` field with a string value, which indicates the type of the node. This field allows building a generic traversal method for the abstract syntax tree. The built-in `Object.entries` method is used to iterate over the properties of the abstract syntax tree nodes. The recursive traversal is initiated by passing a root node.

The properties are analyzed on whether they are of the type `object`. If not, they are skipped in the iteration. A secondary type distinction is then made using the built-in method `Array.isArray`. If the property value is an array, it is assumed to be an array of abstract syntax tree nodes. These nodes are then visited in the order they appear in the property value. An example of this would be the body of a statement list. If the value is not an array, it is assumed to be a direct reference to another abstract syntax tree node. It is then visited directly.

The deobfuscation program defines an abstract class called `DefaultASTWalker`. It defines two abstract methods that have to be implemented by classes that extend it. The first one is `onNodeEnter`, which is invoked when a new node is visited but before its child nodes have been visited. The second one is `onNodeLeave`, which is invoked for each node after its child nodes have been visited. Both methods receive a reference to the node object as a parameter. In addition to that, a reference of the parent node, the property key, and an optional index are supplied. Both the reference to the parent node and the index may optionally be null as not all nodes have a parent node, and not all nodes were found within an array.

A subclass of the `DefaultASTWalker` class, called `FilterASTWalker`, provides a convenient interface to be notified only about a specific set of node types while still traversing the whole abstract syntax tree.

These parameters uniquely identify a node within the abstract syntax tree. They are also required to perform transformations that require knowledge of entire blocks. Suppose one were to replace all identifiers using this generic visitation algorithm. The complex scoping rules of ECMAScript require analyzing scopes entirely to account for `var`, `let`, and **`const`** declarations. Replacing or deleting a node requires modification of the parent node. Therefore, saving references to the nodes will not work since the abstract syntax tree defined by the ESTree specification may only be traversed top-down and not bottom-up.

This problem is solved by extending the ESTree structures generated by meriyah. Knuth and Wegner initially described this procedure in their 1968 research “Semantics of context-free languages”. In their paper, Knuth and Wegner describe a method of transferring information bi-directionally within an abstract syntax tree. The information is transferred by allowing terminals and non-terminals to be annotated with attributes.

The deobfuscation program adds three attributes to each node to allow top-down and bottom-up traversal of the abstract syntax tree. These attributes are called `parent`, `next`, and `prev`. `parent` optionally contains a structure that consists of the three values introduced earlier. This structure allows the replacement of nodes outside of the traversal process.

`prev` is only populated if the current node is part of a list of nodes in its parent node. If that is the case, `prev` will contain the sibling node visited before the current node. It will also be empty if the current node is the first element of the node list.

Analog to that, `next` will contain a reference to the sibling node visited after the current node.

These fields were added for convenience as the deobfuscation program may also calculate their value from the `parent` node.

These attributes are explicitly skipped in the property iteration process to prevent infinite loops and double traversal. These attributes are re-calculated after each modifying traversal to prevent stale connections from accumulating in the `parent` fields.

5.1.3 Design pattern

The deobfuscation program is implemented using the concept of passes. A pass describes one-or-more entire traversals of the abstract syntax tree. The traversal is done to apply modifications to or gather information from the abstract syntax tree. This design concept is taken from optimizing compilers as they work using optimization passes.

Each pass is implemented as a class, which inherits from either `DefaultASTWalker` or a subclass thereof.

`DefaultASTWalker` includes two methods for replacing and removing nodes within the abstract syntax tree. The replacement of nodes is relatively trivial, consisting only of a type distinction to differentiate whether the node to be replaced is part of an array in the parent node or not. This distinction is then used to determine the correct way of assigning the new node in the place of the old node. The `parent` property of the replacement node is adjusted accordingly before the transformation is applied. The `prev` and `next` fields of both the replacement node and the sibling nodes are modified accordingly. This modification is done before the replacement to ensure that the abstract syntax tree is always in a valid state.

However, removing a node from the abstract syntax tree requires more context. Suppose one were to remove the declaration of a variable from the abstract syntax tree. A variable is declared in the ESTree specification using a `VariableDeclaration` node containing one or more `VariableDeclarator` nodes. This structure is used to implement the declaration of multiple variables with the same declaration kind. An example for this would be `let a = 3, b = 4;`. This example would be represented on the abstract syntax tree as a `VariableDeclaration` node containing two `VariableDeclarator` nodes. If a pass removed both of the `VariableDeclarator` nodes, the `VariableDeclaration` node would be pointless and would be invalid syntax. Therefore, the removal method has to take the superordinate nodes into account when removing a node. If the node to be removed is part of an array in its parent node, it may be handled by simply removing the element from the array. The sibling nodes succeeding the removed element in the parent node have their `parent` fields modified to account for the changed index. The `prev` and `next` fields are also adjusted accordingly.

All passes are designed to be idempotent after one or more pass runs. They, therefore, implement a field that tracks the number of transformations done during the current run. Every time a node is modified, replaced or removed, the pass increases this transformation counter. The implementation could automate this tracking by adding it to the pre-defined methods for replacing and removing nodes. This automation would make implementing a pass less obvious, though, as modifying a node property that is not a child node would still require manually incrementing the counter. It would also prevent the implementation of passes from grouping transformations that form a single logical transformation.

This verbosity would skew the number of modifications applied in the case of using them as a metric. The pass is then executed anew until the number of transformations applied is zero.

5.2 Constant Folding

Steven Muchnik introduces Constant Folding (or Constant-Expression Evaluation) in his 1997 book as "the evaluation at compile time of expressions whose operands are known to be constant"[26, p. 329]. A few differences have to be taken into account to apply the processes from the book. The main difference is that the book focuses on compiling and not interpreting code, as is the case for ECMAScript. ECMAScript also does not throw an exception upon overflow or division by zero but will instead return `NaN`.

The following chapter will examine all operations upon expressions defined in the ECMAScript specification on their potential for constant folding. In some cases, this constant folding may transform the source code without changing any semantics, while in other situations, the semantics may change in rare edge cases. If the possibility of changing the program semantics exists for a given transformation, the benefits of applying the transformation regardless usually far outweigh the downsides of risking a faulty transformation. The user will be able to choose to disable the transformations in question using a flag in the configuration. This probabilistic approach allows the program to fold some expressions where only one operand out of two is known.

Additional technical foundations will be introduced as deemed necessary in this chapter. Embedding the technical knowledge within this chapter serves the added cohesion and comprehensibility of the chapter as some concepts only arise in this chapter and are rather complex.

5.2.1 Unary operators

The first class of operators that will be analyzed in this thesis are those that accept only a single operand. The ECMAScript specification also defines a set of unary operators but excludes some operators with a single operand.

5.2.1.1 Unary minus operator

In order to fold numeric constants that involve negative numbers, it is easiest to make use of the fact that the standard numeric types in ECMAScript are always signed. The deobfuscation software can use this to replace the unary minus operation on a numeric literal with the exact literal times minus one. This optimization simplifies further interactions with the value, as it may be used directly without traversing the tree further when, e.g., a multiplication involving the value is done. When the abstract syntax tree is subsequently turned back into human-readable source code, this will be automatically corrected into a unary minus operation if necessary. The particular case of a double negative (e.g. `--x`) does not exist as such in ECMAScript as the preceding minuses will instead be interpreted as the prefix decrement update expression.

Even though this operation appears to be relatively trivial, special attention must be paid to the data structure representing literal values in the abstract syntax tree.

The data structure used to represent literal values consists of its type as the string constant `"Literal "` and a field called `value`. This example ignores the default fields like the beginning and end of the node in the source code. The latter is defined as having a union type consisting of `string`, **`boolean`**, **`null`**, `number`, `RegExp`, and `bigint` as of the 2020 version of the ESTree specification. In the library used by this project, this is extended by a field called `raw`, which contains, as the name implies, the unfiltered string representation of whatever value this literal represents. This raw value is helpful in situations where a literal value can be represented in multiple ways, such as the different radixes supported in ECMAScript.

The unary minus coerces the type of its operand to `number`, so a method is needed to emulate the type coercion as the unary minus operator does it. The steps necessary to fold the unary minus operator occurrences can be taken directly from the ECMAScript standard[12, § 13.5.5.1]. As a first step, the value is passed to the abstract operation $ToNumeric$, which returns the passed value if it is of the type $BigInt$ or passes it on to the abstract operation $ToNumber$.

$ToNumber$ then defines cases for each of the possible input types. The trivial case of the input type being `number` returns the input value without any conversion. If the input value is of the **`null`** type, it is converted into $+0_{\mathbb{F}}$. If the input value is of the `undefined` type, the resulting value will be `NaN`. Instances of the boolean type are converted according to their value. **`true`** is converted to $+1_{\mathbb{F}}$ and **`false`** to $+0_{\mathbb{F}}$ respectively.

String values pose a particular case in the conversion, albeit a relatively trivial one, as outlined in section 7.1.4.1 of the 2021 edition of the ECMAScript standard. In essence, the conversion is limited to `string` values that follow the format of literal numeric values, so a string like `"123"` will be converted to the numeric value `123`. This parsing process also includes `Infinity`, `-Infinity`, and non-decimal integer representations and the scientific notation of numeric values in either decimal or non-decimal representation. BigInt values, however, are explicitly excluded.

Object type values are passed to the abstract operation $ToPrimitive$, which tries to convert "its input argument to a non-Object type"[12, § 7.1.1]. A secondary argument called $hint$ is used if the input value can be converted into multiple primitive types. If no hint is present,`"number"` is assumed, which can be overridden by objects. The exact mechanism can assign a numeric value for an object instance to be converted to. The coercion caused by the unary minus operator passes `"number"` for the hint parameter. If no override is present, the abstract operation $OrdinaryToPrimitive$ is invoked with the same parameters as $ToPrimitive$. It asserts that the hint is either `"string"` or `"number"`. Depending on the hint, it orders the invocation of two member functions. If the hint parameter is `"string"`, the `toString` function will be invoked to provide a primitive value before `valueOf` is invoked. In the case of the hint being `"number"`, the order of invocation is reversed. The result of the abstract operation is then passed to $ToNumber$ again.

Values of either the symbol type or the BigInt type throw a TypeError exception. The latter case can be explained by the preliminary filtering of BigInt values in the abstract operation $ToNumeric$. In the former case, the explanation is a bit less obvious. One argument is that the symbol type is a recent addition to the relatively small set of primitive types supported in ECMAScript. Except for BigInt, which was added in the 2020 edition of ECMAScript, the primitive types of ECMAScript have not had any additions since the conception of the language. While BigInt has an obvious identity mapping into the numeric space, Symbol does not. So the reasoning could have been that instead of adding another less than obvious implicit type conversion, as is the case with `null` and `undefined`, the better solution would be to instead force an explicit type conversion by the user. ECMAScript does not define symbol literals, and therefore it is not included in the union type of the value field of the literal either way.

The identifier for `NaN` can be folded in every case as it can not be reused for identifiers. A negative variant of `NaN` does not exist as it is not a number. The absence of a negative `NaN` implies that every instance of −`NaN` can be replaced with `NaN` regardless of its context[12, § 6.1.6.1.1].

5.2.1.2 Unary plus operator

The unary plus operator explicitly coerces an expression into a number. It uses most of the mechanisms of the unary minus operator just without negating the result. Instead of the abstract operation $ToNumeric, ToNumber$ is invoked directly. Therefore, it is implied that the unary plus operator is not defined on BigInt nor Symbol and will lead to a `TypeError` exception.

5.2.1.3 `delete` operator

The unary delete operator is used to delete a property from an object and, as such, does not have meaningful optimization opportunities.

5.2.1.4 `void` operator

The void operator evaluates the targeted expression and returns `undefined` instead of the result of the expression. It may be folded if the expression provably has no side effects. If that is the case, the expression may be removed entirely. Removing the node is done by checking the type of the parent node. If the parent node is an expression, the identifier for `undefined` is inserted to prevent invalid code from being generated.

Determining whether the expression is pure requires a recursive search depending on the type of the expression. All available expressions will be analyzed in the following to illustrate this. This side-effect algorithm will be reused throughout this thesis.

The arrow function expression is side effect free as it does not change the program's state by merely existing. Only assigning or calling it might change the state. Assignment expressions (=, +=, %=, and so forth) change the program's state as they (re-)assign a value to a variable. They may be removed if the assignment is proven to be a dead store and if the assigned value is side effect free. A dead store is an assignment operation on a variable that is never read after the assignment. The newly assigned value is therefore never used and may be discarded. This optimization needs more context than the abstract syntax tree can provide on its own.

Binary expressions are only pure if both the left and the right operand are side effect free. Conditional expressions, also known as the ternary operator, are side effect free if the condition, the consequent, and the alternative are side effect free.

Chain expressions allow chaining member access while checking whether a reference in the chain is valid. An example of this would be `value.field?.subfield`. In this example, the property `subfield` would only be accessed if the property `field` of `value` is valid. The ESTree specification only allows `SimpleCallExpression` and `MemberExpression` as child expressions for chain expressions. Both are hard, if not impossible, to prove as side effects free from a static perspective as they could be accessor properties. Nevertheless, a chain expression can be considered pure if its sub-expression is also pure.

Logical expressions are essentially the same as binary expressions but with the operators "logical or" (||) and "logical and" (&&). The new expression invokes a constructor and, as such, does not qualify as side effect free. This also includes the **`super`** expression. The sequence expression also called the comma operator, allows chaining expressions. An example of this would be `(x(), 3)`, where `x()` would be evaluated first and then 3. The result of this expression is always the right sub-expression. Therefore, a sequence expression is side effect free if all sub-expressions in the chain are side effect free.

An await expression is only side effect free if whatever is awaited is also side effect free. The purity of a promise can not be proven statically for every case and needs more context than an abstract syntax tree could provide to eliminate even the most trivial of cases.

Call expressions could, in theory, be side effect free if the called function can be resolved statically, and all statements within the function body are also side effect free. Doing static analysis would require more information about control and data flow within the program, which is unavailable at this optimization stage. Import expressions load modules into the program and may run arbitrary code across module boundaries. It is unfeasible to extend the analysis outside the scope of the main module at this time.

While the deobfuscation program may not reason about arbitrary functions, it may reason about a subset of the built-in functions. These pure built-in functions are defined by name in a list within the deobfuscation program. However, the dynamic nature of ECMAScript allows an adversary to overwrite these functions in a manner that the deobfuscation program can not trace. Therefore, the user may specify a value from the override tolerance enumeration to specify whether the program should assume standard library functions to be overridden. If the value is set to ignore the possibility of these functions being overwritten, the program will consider call expressions involving the pre-defined list of built-in functions pure. A second requirement for this is that all arguments supplied are pure as well.

ECMAScript features two ways of specifying classes that work similarly to the declaration of functions. Class expressions allow the declaration of anonymous classes and, similarly to function expressions, do not execute any code merely by being defined, making them side effects free. On the other hand, class declarations implicitly add identifiers to the global scope and, as such, are not side effect free.

Member expressions also referred to as the dot operator, allow property access to an object. For example, calling the function `console.log` involves a member expression on the global `console` object. Accessing a property may result in code execution. A simple example of this would look like this:

```
let v = {};
v.__defineGetter__('random', () => Math.random());
console.log(v.random);
```

The `__defineGetter__` allows associating an arbitrary function with an object's property and, as such, suffers from the same limitations as the call expression. The same reasoning leads to identifier expressions not being side effect free. An example of this would be:

```
window.__defineGetter__('x', function() {return Math.random()})
```

To understand this, one must understand the global object in an ECMAScript engine works first. Both V8, which is the ECMAScript engine of Chromium-based browser, and SpiderMonkey, which handles ECMAScript within Firefox, respectively use the `window` global object. V8 is also used for node.js, which runs standalone ECMAScript files instead of running them within the context of a website. Node.js, however, uses the current ECMAScript file, also known as a module, as the global object instead. Following the ECMAScript standard, the global object "is created before control enters any execution context"[12, § 19]. The properties of the respective global object are added to the global scope and can be accessed by their name without specifying member access on the global object.

The global object, precisely its properties, hosts most of the standard library. All built-in objects, such as `Math`, `JSON`, or `Object`, are properties of the global object. Even built-in constants such as `undefined`, **`null`**, and `Infinity` are properties of the global object. They are, however, marked as non-writable, non-enumerable, and non-configurable and, as such, can not be modified[12, § 19.1].

However, a user may decide similarly to the handling of call expressions above to ignore the possibility of this occurring by specifying a higher tolerance for overridden values.

Returning to the above example, it becomes apparent that while accessing regular variables may not trigger side effects, properties of the global object may trigger a function and, as such, may have arbitrary side effects. If `x` is used subsequently and has not been defined otherwise, it will run the assigned function instead. One way of solving this issue would be to execute the constant folding that involves identifiers after data flow analysis, and the known variables at a given point in the abstract syntax tree are known. A local definition of `x` takes precedence over implicit member access to the global object. Instances where a variable is known to be defined locally instead of globally could allow all optimizations to be applied. This implicit access also works for **`this`**, which is implicitly added in some contexts. While in the global scope, it is equal to the global object. Functions and classes implicitly bind their own **`this`**. Arrow function expressions are exempt from this, and **`this`** may only be bound explicitly.

Object expressions are side effect free if all assigned properties are side effect free. Checking this is complicated as object expressions allow method definitions, properties, rest elements, and spread elements. Method definitions themselves are side effects free. Properties are side effect free if both the key and the value are side effect free. Both spread and rest elements depend on their sub-expression being side effect free.

Template literals embed expressions within a string literals. An example for this would be `let message = `Hello ${123} ${x()}``. The value of the message would be `"Hello 123 "` concatenated with the result of `x()`. Such a template literal is context-free if all embedded expressions are side effect free.

Unary expressions are side effects free only for some operators. The `delete` operator is not side effect free when the program runs in strict mode. If the program is not running in strict mode, some cases may be side effect free. The remaining unary operators are side effect free if and only if the argument is side effect free.

Update expressions are side effect free if their argument is also side effect free. Yield expressions are not side effect free as they influence the control flow.

5.2.1.5 Bitwise negation operator

Returning to the unary operators, the unary, bitwise negation operator is relatively trivial as it is analogous to the unary plus and minus operators in coercing its argument into a number and then inverts all bits of the numeric value. The coercion into a `number` is done using the abstract operation $ToNumeric$ and supports BigInt values.

All bitwise operations on `number` values use the abstract operation $ToInt32$ before applying the bitwise operations. This operation converts the `number` type values into a twos-complement integer representation to apply bitwise operations as they appear in C-like languages. BigInt values are preserved as they are already integers. It maps `NaN`, $+0_{\mathbb{F}}$, $-0_{\mathbb{F}}$, $\infty_{\mathbb{F}}$, and $-\infty_{\mathbb{F}}$ to $+0_{\mathbb{F}}$ making the operation idempotent[12, § 7.1.6]. After the bitwise operation, the resulting value is converted back into the `number` type respectively, if necessary.

5.2.1.6 Logical negation operator

The unary, logical negation operator coerces its value into a boolean, using the abstract operation $ToBoolean$ instead of a numeric value as the previous operators did. This conversion obeys the following rules:

- `undefined` is converted into **`false`**.
- **`null`** is converted into **`false`**.
- **`boolean`** is returned without conversion.
- `number`: $+0_{\mathbb{F}}$, $-0_{\mathbb{F}}$, and NaN are converted into **`false`**. All other values are converted into **`true`**.
- `string`: an empty string will be converted to **`false`**, and all other strings are converted to **`true`**.
- All values of the type `symbol` are converted to **`true`**.
- `bigint`: Only $0_{\mathbb{Z}}$ is converted to **`false`**, all other values are converted to **`true`**.

- All values of the type `object` are converted to **`true`**.

The negation operator then inverts the value, so **`true`** becomes **`false`** and vice versa. This optimization can be applied relatively easily following the above rules.

Exceptional cases for the immutable, global identifiers `NaN` and `Infinity` have been added to simplify the source code further.

5.2.1.7 `typeof` operator

The `typeof` operator returns a string that indicates the type of the argument passed to it. This definition leaves an easy opportunity for constant folding by folding the types of literal values. In a later step, this could also include side-effect-free identifiers of known types. This additional optimization would require type inference, however. The conversion from a value of a type into a string representing the type works as follows:

- The values of the type **undefined** returns the string `"undefined"`
- **null** returns the string `"object"` due to legacy compatibility reasons. “In the first implementation of JavaScript, JavaScript values were represented as a type tag and a value. The type tag for objects was 0. null was represented as the NULL pointer (0x00 in most platforms). Consequently, null had 0 as the type tag, hence the `typeof` return value `"object"`”[36].
- Objects that implement `[[Call]]` (e.g., functions) return `"function"`, while other objects return `"object"`.
- All other types return the name of the type in all lowercase characters, so for example, a value of the `BigInt` type will return `"bigint"`.

Another opportunity for folding can be applied if the user tolerates that the program ignores the possibility of standard library functions and constants being overridden. If that is the case, the program will analyze both $MemberExpressions$ as well as $CallExpression$ on whether they refer to elements of the standard library. In this case, the program will apply the appropriate types for both the invocation of functions and the access of constants. A few examples of this behavior:

- `typeof Math.PI` will be folded to `"number"`.
- `typeof Math.random()` will be folded to `"number"`.
- `typeof Math.random` will be folded to `"function"`.
- `typeof Math.random(x)` will not be folded since its arguments are not necessarily side effect free.

5.2.1.8 Update expressions

This thesis will consider update expressions as unary expressions, even though newer versions of the ECMAScript standard have a dedicated, separate chapter for update expressions. The operators included in this category are the postfix and prefix variants of the increment (`++`) and decrement (`--`) operators. While these update expressions can, according to the context-free grammar, be applied to

$UnaryExpression$ for the prefix operators or $LeftHandSideExpression$ for the postfix operators, they are limited by the syntax-directed operation $AssignmentTargetType$. It limits the expressions that can be supplied to the update expressions to identifiers, member access, and the result of call expressions. As such, the update expressions have no meaningful constant folding to be applied.

5.2.2 Binary operations

Binary operations have, as the name implies, two operands. As with the unary expressions, they will be analyzed individually, listing opportunities for constant folding wherever possible. Contrary to unary expressions, though, this section will also try to fold operations on identifiers and other expressions, which may or may not have side effects. Most of these operations are relatively trivial on literal values alone, so this extension is added. This behavior can be toggled in the software if faulty optimizations are applied to the source code.

All binary operations that accept two numeric parameters will throw a `TypeError` when two differing numeric types are passed to the operator unless it is explicitly stated that different types are accepted. This inherent problem leads to all binary operations potentially throwing an exception, and as such, most of the constant folding opportunities discussed in the following could not be applied if this were to be followed strictly.

5.2.2.1 Exponentiation operator

The exponentiation operator `**` allows exponentiating a base, which is supplied by the left-hand side operand of the binary expression, by an exponent, which is supplied by the operator's right-hand side. The trivial case for constant folding is when both arguments are literal values and can be exponentiated as they are in the abstract syntax tree. Special care has to be taken if the constant folding is not done using ECMAScript as there is a derivation from the IEEE 754 floating-point standard. IEEE 754 dictates, that the result of exponentiating $1_{\mathbb{F}}$ or $-1_{\mathbb{F}}$ with either $+\infty_{\mathbb{F}}$ or $-\infty_{\mathbb{F}}$ results in $1_{\mathbb{F}}$. ECMAScript differs in its implementation from this in so far as that the result of this operation will be `NaN` instead of $1_{\mathbb{F}}$. This deviation from the IEEE 754 standard exists because this behavior was only specified in the 2019 revision of the IEEE 754 standard. ECMAScript predates this specification and has chosen to keep the pre-existing behavior for legacy reasons[12, § 6.1.6.1.3].

The evaluation of the exponentiation operator begins with the abstract operation

$EvaluateStringOrNumericBinaryExpression$, which receives the base, the exponent, and the operator (`**`) as parameters. It then extracts the values of both sides of the binary operation and passes these on to the abstract operation $ApplyStringOrNumericBinaryOperator$, which will coerce both parameters to a numeric type. If the numeric types of the sides do not match, e.g., the left side has the type `BigInt`, and the right side has the type `number`, a type error will be thrown. After the type check, the evaluation goes on to either `BigInt::exponentiate` or `Number::exponentiate`, depending on the type of both parameters.

`BigInt::exponentiate` throws a `RangeError` if an exponent $< 0_{\mathbb{Z}}$ is passed as BigInt can not represent the result correctly. A shortcut if both the base and the exponent are $0_{\mathbb{Z}}$ is defined with the result $1_{\mathbb{Z}}$. The absence of reserved values causes the default case to exponentiate the base by the exponent and return the result.

`Number::exponentiate` has a few shortcuts and special cases at the beginning[12, § 6.1.6.1.3]:

1. $\forall x \in \mathbb{F} : x^{NaN_{\mathbb{F}}} = NaN_{\mathbb{F}}$
2. $\forall x \in \mathbb{F}, p \in \{+0_{\mathbb{F}}, -0_{\mathbb{F}}\} : x^p = 1_{\mathbb{F}}$
3. $\forall x \in \mathbb{F} : NaN_{\mathbb{F}}^x = NaN_{\mathbb{F}}$
4. $\forall x \in \mathbb{F} : +\infty_{\mathbb{F}}{}^x = +\infty_{\mathbb{F}}$, for $x > +0_{\mathbb{F}}$, otherwise the result is $+0_{\mathbb{F}}$
5. If the base is $-\infty_{\mathbb{F}}$
    1. If the exponent is larger than $+0_{\mathbb{F}}$
        1. $-\infty_{\mathbb{F}}$, if x is an odd integer, return $-\infty_{\mathbb{F}}$.
        2. Otherwise return $+\infty_{\mathbb{F}}$
    2. If the exponent is smaller or equal to $+0_{\mathbb{F}}$
        1. $-\infty_{\mathbb{F}}$, if x is an off integer, return $-0_{\mathbb{F}}$.
        2. Otherwise return $+0_{\mathbb{F}}$
6. If the base is $+0_{\mathbb{F}}$
    1. If the exponent is larger than $+0_{\mathbb{F}}$.
    2. Otherwise return $+\infty_{\mathbb{F}}$
7. If the base is $-0_{\mathbb{F}}$
    1. If the exponent is larger than $+0_{\mathbb{F}}$
        1. $-\infty_{\mathbb{F}}$, if x is an odd integer, return $-0_{\mathbb{F}}$.
        2. Otherwise return $+0_{\mathbb{F}}$
    2. If the exponent is smaller or equal to $+0_{\mathbb{F}}$
        1. $-\infty_{\mathbb{F}}$, if x is an off integer, return $-\infty_{\mathbb{F}}$.
        2. Otherwise return $+\infty_{\mathbb{F}}$
8. If the exponent is $+\infty_{\mathbb{F}}$
    1. If $|x| > 1 \Rightarrow x^{+\infty_{\mathbb{F}}} = +\infty_{\mathbb{F}}$
    2. If $|x| = 1 \Rightarrow x^{+\infty_{\mathbb{F}}} = NaN_{\mathbb{F}}$
    3. If $|x| < 1 \Rightarrow x^{+\infty_{\mathbb{F}}} = +0_{\mathbb{F}}$
9. If the exponent is $+\infty_{\mathbb{F}}$
    1. If $|x| > 1 \Rightarrow x^{+\infty_{\mathbb{F}}} = +0_{\mathbb{F}}$
    2. If $|x| = 1 \Rightarrow x^{+\infty_{\mathbb{F}}} = NaN_{\mathbb{F}}$
    3. If $|x| < 1 \Rightarrow x^{+\infty_{\mathbb{F}}} = +\infty_{\mathbb{F}}$
10. If base $< -0_{\mathbb{F}}$ and the exponent is not an integral number, return NaN

This list is sorted by priority, so the exponentiation `Infinity ** 0` will result in $+1_{\mathbb{F}}$ because rule number two takes precedence over rule number five. Another non-obvious example is the case of `NaN ** 0`. Intuitively the answer would probably be `NaN` since a value that is not a number can not be turned into a number. However, since rule number two precedes rule number three, the result will be $+1_{\mathbb{F}}$.

Not all of these rules are useful for the process of constant folding. The first rule allows the program to fold all occurrences of `x ** NaN` to `NaN`, given that `x` is either an expression without side effects or, in the case of the user setting the program to assume that identifier access is side effect free, an identifier.

The second rule can also be used for constant folding by rewriting all occurrences of `x ** 0` to a numeric literal with the value $+1_{\mathbb{F}}$. While the third rule is similar to the first rule, it can only be used for constant folding if the user is willing to accept a risk of a misfolding. Given the expression `NaN ** x`, one could apply the third rule and fold this expression to `NaN`. This transformation would be valid as long as `x` does not evaluate to either $+0_{\mathbb{F}}$ or $-0_{\mathbb{F}}$ as that would instead invoke the second rule. Setting the assumption of `x` not evaluating to zero here would be sensible in the context of code written by humans but considering that the target program is obfuscated to deliberately prevent it from being readable, it would be unreasonable to assume it by default. The user may, however, enable this behavior in the program's settings.

While interesting for the evaluation, the remaining rules are of little importance for the current step of the deobfuscation. In theory, a SAT solver like z3 could limit the possible values the exponents or bases of the operations could take. This addition would require a significant amount of additional context that the abstract syntax tree in its current form can not provide.

5.2.2.2 Multiplication operator

The multiplication operator `*` works in the same way as the exponentiation operator as it also invokes the abstract operation $EvaluateStringOrNumericBinaryExpression$ followed by $ApplyStringOrNumericBinaryOperator$. Therefore, it also coerces both operands of the operation into a numeric type and throws a `TypeError` if those types differ from each other.

Implementing the multiplication operator for values of the type `BigInt` is again relatively trivial. It simply states that a value "that represents the product of x and y"[12, § 6.1.6.2.4] is returned, with x and y being the names of the operands. In order to allow interoperability with the ECMAScript standard, the operands of the multiplicative operator will also be called x and y in this thesis. The only form of constant folding that may be applied here concerns the trivial case of both operands being `BigInt` literal values.

The real opportunities for constant folding arise, as with the exponentiation operator, from implementing the `number` type. Multiplication is handled in `Number::multiply` and includes the following exceptional cases[12, § 6.1.6.1.4]:

1. If either of the operands is `NaN` the result will also be `NaN`

2. If x is $+\infty_{\mathbb{F}}$ or $-\infty_{\mathbb{F}}$
    1. If y is $+0_{\mathbb{F}}$ or $-0_{\mathbb{F}}$, return `NaN`.
    2. If $y > +0_{\mathbb{F}}$, return x.
    3. Otherwise return -x.
3. Since “finite-precision multiplication is commutative”[12, § 6.1.6.1.4], rule number two also applies with x and y swapped.

Rule one allows constant folding of all occurrences of either `x * NaN` or `NaN * x`, with `x` being either a side effect free expression or an identifier if the user has chosen to consider identifier access side effect free. Another opportunity for constant folding is not derived directly from the exceptional cases but instead from the mathematical properties of multiplication. The neutral element in multiplication is the number one, and this is of use for constant folding in so far as that all expressions of the form `x * 1` or `1 * x` may be folded to simply `x` with the above limitations placed on `x`. This transformation works because even if `x` evaluates to `NaN`, the result will not change since `1 * NaN` is equal to `NaN`. Multiplication by zero, however, may not be folded unless `x` is proven not to be `NaN`, because of rule one and not to be either $+\infty_{\mathbb{F}}$ or $-\infty_{\mathbb{F}}$, due to rules two and three.

5.2.2.3 Division operator

The division operator / also follows the already established path of abstract operations and, as such, also coerces both operands into numeric types. The operands are the dividend and the divisor, with the former being on the left-hand side of the operator and the latter on the right-hand side. Unlike the previous implementations for `BigInt`, it contains a particular case as with the implementations for `Number`. If the divisor is zero, a range error will be thrown. This particular case can fold `BigInt` division expressions by replacing them with a throw statement that directly throws a `RangeError` instead of waiting for the exception to be thrown at runtime. The result of the `BigInt` division is rounded to the next integer value.

The exceptional cases for `number` values are as follows[12, § 6.1.6.1.5]:

1. If either operand is `NaN`, the result will also be `NaN`.
2. If the dividend is $+\infty_{\mathbb{F}}$ or $-\infty_{\mathbb{F}}$:
    1. If the divisor is $+\infty_{\mathbb{F}}$ or $-\infty_{\mathbb{F}}$, the result will be `NaN`
    2. If the divisor is larger or equal to $+0_{\mathbb{F}}$, the result will be equal to the dividend
    3. Otherwise, the result will be the dividend with the sign inverted.
3. If the divisor is $+\infty_{\mathbb{F}}$
    1. If the dividend is larger or equal to $+0_{\mathbb{F}}$, the result will be $+0_{\mathbb{F}}$
    2. Otherwise, the result will be $-0_{\mathbb{F}}$
4. If the divisor is $-\infty_{\mathbb{F}}$
    1. If the dividend is larger or equal to $+0_{\mathbb{F}}$, the result will be $-0_{\mathbb{F}}$

    2. Otherwise, the result will be $+0_{\mathbb{F}}$

5. If the dividend is $+0_{\mathbb{F}}$ or $+0_{\mathbb{F}}$
    1. If the divisor is $+0_{\mathbb{F}}$ or $+0_{\mathbb{F}}$, the result will be `NaN`
    2. If the divisor is larger than $+0_{\mathbb{F}}$, the result would be equal to the dividend
    3. Otherwise the dividend with inverted sign will be returned.
6. If the divisor is $+0_{\mathbb{F}}$:
    1. If the dividend is larger than $+0_{\mathbb{F}}$, the result will be $+\infty_{\mathbb{F}}$
    2. Otherwise the result will be $-\infty_{\mathbb{F}}$
7. If the divisor is $-0_{\mathbb{F}}$:
    1. If the dividend is larger than $+0_{\mathbb{F}}$, the result will be $-\infty_{\mathbb{F}}$
    2. Otherwise the result will be $+\infty_{\mathbb{F}}$

The opportunities for constant folding directly derived from these exceptional cases are scarce, as most have many conditions. The most obvious is the first rule, and folding occurrences of `x / NaN` or `NaN / x` with `NaN`, with `x` being an alias for side effect free expressions or if the user enabled side effect free identifier access. Another opportunity derives from the mathematical properties of division over real (and floating-point) numbers. Division by one will always yield the dividend as a result. This opportunity can be proven by applying the rules:

- `Infinity / 1` or `-Infinity / 1` will evaluate to `Infinity` or `-Infinity` respectively because of rule number 2.2.
- `NaN / 1` will evaluate to `NaN` as per rule number one.
- `0 / 1` or `-0 / 1` will evaluate to `0` or `-0` respectively due to rule number five.

The default case comes from the mathematical properties of the division operator. Therefore, the program may replace all occurrences of `x / 1` with `x`.

5.2.2.4 Modulo operator

The final operator from the category of the multiplicative operators is the remainder operator, also known as the modulo operator. ECMAScript uses the percent sign for this operator, similar to other programming languages. As with the operators previously discussed, it invokes $EvaluateStringOrNumericBinaryExpression$ followed by $ApplyStringOrNumericBinaryOperator$ again. The operands are called `n` on the left-hand side and `d` on the right-hand side by the ECMAScript standard[ECMA International [12], 6.1.6.1.6][12, § 6.1.6.2.6].

`BigInt::remainder` includes two exceptional cases. The first one is concerned with the case of `d` being zero. Trying to determine the remainder using a zero divisor causes a `RangeError` to be thrown. However, if `n` is zero, the result will be zero too. The opportunities for constant folding here are similar to the ones found in the division operator on `BigInt`. Occurrences of `x % 0` could be replaced with ThrowStatement instead to preempt the exception being thrown implicitly at runtime.

However, the latter case does not provide a straightforward opportunity of applying constant folding as occurrences of `0 % x` could lead to a `RangeError` being thrown if `x` evaluates to zero. The program has added a switch to allow this folding regardless of its theoretical impact.

the exceptional cases become more numerous when moving on to `Number::remainder`, as with the previous operators[12, § 6.1.6.1.6]:

1. If either operand is `NaN`, the result will also be `NaN`
2. If `n` is either $+\infty_{\mathbb{F}}$ or $-\infty_{\mathbb{F}}$, the result will be `NaN`
3. If `d` is either $+\infty_{\mathbb{F}}$ or $-\infty_{\mathbb{F}}$, the result will be `n`
4. If `d` is either $+0_{\mathbb{F}}$ or $-0_{\mathbb{F}}$, the result will be `NaN`
5. If `n` is either $+0_{\mathbb{F}}$ or $-0_{\mathbb{F}}$, the result will be `n`

The non-conditional nature of these exceptional cases provides a lot of constant folding opportunities:

- The first one can be derived from the first rule and allows the rewriting of all occurrences of `NaN % x` or `x % NaN` to `NaN`, with x being either a side effect free expression or an identifier if the user has chosen to consider them side effect free.
- Rule number two allows folding all occurrences of `Infinity % x` or `-Infinity % x` to `NaN`.
- The following rule allows rewriting `x % Infinity` and `x % -Infinity` to simply `x`.
- The penultimate rule allows rewriting all occurrences of `x % 0` and `x % -0` to `NaN`.
- The final rule allows rewriting all occurrences of `0 % x` and `-0 % x` to `n` if `x` is side effect free.

5.2.3 Additive operators

After concluding its section on multiplicative operators, the ECMAScript standard moves on to the category of additive operators consisting of the addition and subtraction operator. Both of them reuse the symbols for their operators as the addition operator uses the plus sign previously used in the unary plus operation, which tries to coerce its operand into the `number` type explicitly. In turn, the subtraction operator reuses the minus sign previously used by the unary minus operator.

5.2.3.1 Addition operator

The addition operator forms a particular case in contrast to the previously discussed operators as it does not strictly operate on numeric types. It is also defined for the `string` type and will coerce both operands to the string type if either operand is of the type `string`. This coercion happens because it serves both as the operator for adding two instances of numeric types and the operator for string concatenation.

This two-fold usage leads to some expressions with non-obvious results that have gained notoriety among ECMAScript enthusiasts and its critics[23]. An example of this would be the expression `[] + []`. Intuitively the result might be an empty array, but the solution here is not quite as obvious. The

`Array` object, as defined in the ECMAScript specification, does not provide a `valueOf` method by default, so when passed to the previously introduced abstract operation $OrdinaryToPrimitive$, it will invoke the `toString` method, which the `Array` object defines[12, § 23.1.3.31] in the absence of a property with the `@@toPrimitive` symbol as a key. The `toString` method, if not overridden, will then invoke the `join` method with the separator `""`. Invoking this method causes all array elements to be converted into a `string` representation and subsequently concatenated. Applying these steps now means that the expression `[] + []` will evaluate to `""+ ""` which in turn will evaluate to the empty string (`""`).

A particular case of this can be found in the expression `{} + {}`. If this expression is part of a statement or an expression, it evaluates to `"[object Object][object Object]"`, which is the result according to the ECMAScript standard and the rules as explained in the previous example. However, if the expression is evaluated using `eval()`, its result becomes `NaN` in both V8 and SpiderMonkey. This quirk happens due to an ambiguity in the ECMAScript syntax. The code can be parsed both as an expression and an empty code block, followed by a unary expression. The unary expression is the unary plus invoked on an empty object, which will evaluate to `NaN`[33]. This explanation also applies to `{}+[] === 0`, with the only difference being that an empty array defines a different conversion to number than an object does by default. A compelling and similarly misleading expression is `{foo: "bar"}+{}`. Instead of the first pseudo object literal defining an object with the property `foo` and the value `"bar"`, it defines another code block with a labeled statement. These labeled statements are combined with **`continue`** or **`break`** in the context of nested loops. So the actual meaning of the expression `{foo: "bar"}` is a code block containing a labeled statement with the label `foo`, which in turn labels an expression statement containing the string literal `"bar"`.

By coercing the other parameter into a string when the other side has the type `string`, many constant folding possibilities are prevented. For example, the expression `1 + x + 1` could be simplified to `x + 2` if only mathematical laws were applied. In ECMAScript, however, `x` could be of the type `string` or could evaluate to a value of the type string when converted into a primitive value. An example of this is the case of `x` evaluating to `"foo"`. The result of the above example would then be `"1foo1"`. Even the expression `x + 1 + 1` suffers from this limitation as it is evaluated left-to-right, causing the result to become `"foo11"`.

This paragraph will now focus on applying the addition operator to numeric values by first considering the semantics of adding two values of the type `BigInt`. No exceptional cases exist for this operation, and the definition consists solely of returning a `BigInt` value that represents the sum of both operands[12, § 6.1.6.2.7].

`Number::add`, which names its operands `x` and `y`, contains several exceptional cases[12, § 6.1.6.1.7]:

1. If either operand is `NaN`, the result will also be `NaN`
2. If one operand is $-\infty_{\mathbb{F}}$ and the other is $+\infty_{\mathbb{F}}$, the result will be `NaN`
3. If either operand is $+\infty_{\mathbb{F}}$, the result will be $+\infty_{\mathbb{F}}$
4. If either operand is $-\infty_{\mathbb{F}}$, the result will be $-\infty_{\mathbb{F}}$
5. If both operands are equal to $-0_{\mathbb{F}}$, the result will also be $-0_{\mathbb{F}}$

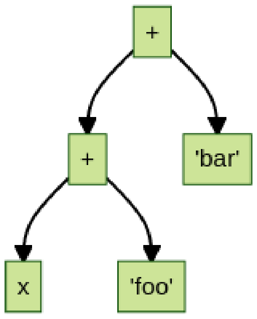

Figure 5.1: Simplified AST of a string concatenation

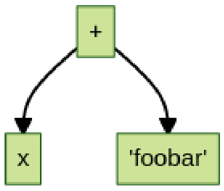

Figure 5.2: Simplified AST of a string concatenation after constant folding has been applied

All rules except for the first one either implicitly or explicitly require knowing both tangible values to activate, and as such, they are of no use when it comes to constant folding. The first rule is only applicable if it can be asserted that the other operand is not a string.

The overloading of the plus operator also introduces another problem. Consider the expression `x + "foo"+ "bar"`. The abstract syntax tree for this expression may look something like shown in figure 5.1. The nodes labeled with a plus represent a binary expression with the plus operator with its left and right-hand side being displayed as child nodes. The issue with this expression is that the program can not fold it by only visiting each binary expression node. If the root node is visited, the program will analyze both child nodes and conclude that the root operation can be folded if the left-hand side can also be folded. However, once the left-hand side of the root node is being analyzed, the program will conclude that, since this expression includes an unknown value and none of the special rules outlined above can be applied, this subexpression can not be folded either.

This particular case is where the program can use the associative property of string concatenation in ECMAScript. Since the right side of the child's expression is already known to be a string, the left side will be coerced into a string. The program can use this to concatenate both strings `"foo"` and `"bar"` despite not being in the same binary operation. The result of this transformation can be seen in figure 5.2

A simple lookahead accomplishes this behavior in handling the binary plus operator. The program checks whether the left-hand side is also a binary expression with a plus operator and a literal string on its right-hand side. This behavior does not work with addition since the coercion of the operands into the string type can not be ruled out.

5.2.3.2 Subtraction operator

The subtraction operator uses the same symbol for its operator as the unary minus operator. It also uses the same abstract operations as the previous operators. Its implementation for values of the type `BigInt` states that a value representing the difference between its operands is returned[12, § 6.1.6.2.8].

However, its implementation for values of the type `Number` has a peculiarity. It is the first operator analyzed in this thesis that invokes another operator. To be precise, the operator to be invoked is the binary plus operator. Internally, ECMAScript transforms the expression `x - y`, with both `x` and `y` evaluating the `number` type values into `x + (-y)`. It is explicitly stated that this transformation will always yield the same result[12, § 6.1.6.1.8]. Because the binary plus operator is being invoked here with arguments that have been coerced to be of the type `number`, it is safe to apply the constant folding opportunities for the binary plus operator that were previously unavailable due to the string coercion. The foremost folding opportunity resulting from this transformation is the ability to fold expressions that have a `NaN` on either side to `NaN`.

An additional constant folding opportunity arises from the mathematical properties of subtraction. This opportunity is the subtraction of zero from another value. This rule means that all expressions of the format `x - 0` may be folded to simply `x`. Any potential side effects are preserved in this transformation. Another opportunity for constant folding places the zero on the other side of the operation and allows simplifying expressions that look like `0 - x` to `-x`. This particular case may trigger the first rule of the plus operator. However, if `x` evaluates to NaN and since there is no negative `NaN`, as explained in section 5.2.1.1, the expression would not hold. However, this can be disregarded since `NaN` is never equal to `NaN`.

The mathematical definition of the subtraction operator yields one more possible constant folding opportunity. Expressions of the form `x - -y` can be simplified to `x + y` with some caveats. Since the subtraction operator implicitly coerces its operands to a numeric type, it has to be assumed that both operands are already of a numeric type for this constant folding opportunity to work. If either operand is of the `string` type and this folding is applied regardless, the result may be invalid. A simple example would be if `y` evaluates to `"foo"`. In the case of `x - -y`, it would lead to `-y` evaluating to `-NaN`, which in turn evaluates to `NaN`, which will omit the initial value of `y` from the rest of the evaluation. If this expression were now to be transformed to `x + y`, it would coerce `x` into the string type, and the resulting value would be a string with the suffix `"foo"`. The user may choose to enable this folding in the settings regardless.

5.2.3.3 Bitwise operators

The following chapter will cover the bitwise shift operators, which allow shifting the binary values to the left and right, and the bitwise operators, which allow direct manipulation of the underlying bits of a numeric value. Using any of the operators analyzed in the following chapters with a value of the type `number` on one side and a value of the type `BigInt` on the other will raise a `TypeError` exception, as with the binary operators that were previously discussed.

In order to apply any of these operations, values of the `number` type will need to be converted to 32-bit integers first. This conversion is done using the abstract operation $ToInt32$, which, as the name implies, converts a value that has been passed to it into the ECMAScript internal integer data type. “It converts argument to one of 2^{32} integral Number values in the range $\mathbb{F}(-2^{31})$ through $\mathbb{F}(2^{31}-1)$, inclusive”[12, § 7.1.6]. Values unique to IEEE 754, like $-\infty_{\mathbb{F}}$, $+\infty_{\mathbb{F}}$, $-0_{\mathbb{F}}$, or NaN that do not have a direct mapping to an integer are replaced by $+0_{\mathbb{F}}$ before the conversion. The input value is then rounded down to the largest integer that is not larger than the initial value [12, § 5.2.5] [12, § 7.1.6] while keeping the sign. This calculation is done on the mathematical value, so restrictions of data types do not apply. The remainder of the input value with the divisor 2^{32} is then calculated to accommodate for the fact that the `number` types allow for much larger values than a 32-bit integer can fit. This result is then mapped back into a concrete data type. Because of the sign bit, values above 2^{31} have 2^{32} subtracted from them before the conversion to accommodate for the overflow. This for example causes the expression `(2 ** 32)- 1 | 0` to evaluate to `-1` instead of `(2 ** 32)-1` as it would have in other programming languages.

Values of the type `BigInt` do not have to be converted to be used in a bitwise operation.

5.2.3.4 Bitwise AND

The bitwise AND operation takes two operands, `x` and `y`, and compares them bit-by-bit as the name implies. For each bit set in both operands, the bit with the same index will be set in the result. An example of this would be the operation `0b1000 & 0b1111`, where the operation would leave `0b1000`.

Neither `number` nor `BigInt` define any special rules that would allow for constant folding. There are two theoretical opportunities for constant folding regardless. The first is the neutral element, which would be one side having all bits set to one. Having the neutral element on one side will lead to the result being the value on the other side. However, there is a limitation to this, as the input value will still be converted to a 32-bit integer when dealing with the `number` type. The `number` value with all of its bits set is `-1`. An example of this behavior would be `Infinity & -1`, as `Infinity` will be turned into zero by the abstract operation $ToInt32$.

To still apply constant folding here, the other operand would have to be turned into a 32-bit integer explicitly. Since $ToInt32$ is an abstract operation that can not be called directly, another method must be found to invoke it implicitly. An obvious solution would be to use a bitwise operator. However, this leads right to where the transformation started and can, as such, be disregarded. Furthermore, it would also ruin the idempotency of these transformations.

This restriction does not apply to the operation on `BigInt` as there are no special values or overflows to account for. However, an issue that arises is concerned with the variable size of `BigInt` values as there is no maximum value. This lack of a maximum value makes applying this constant folding opportunity impossible without knowing the other value. The size of the result will be the size of the highest bit set in both operands.

The second opportunity for constant folding that can be used is one of the values having no bits set.

This condition is fulfilled for values of the type `number` by all values that are turned into zero by the abstract operation $ToInt32$:

- $+\infty_{\mathbb{F}}$
- $-\infty_{\mathbb{F}}$
- $+0_{\mathbb{F}}$
- $-0_{\mathbb{F}}$
- NaN

If either side evaluates to either of these values and the other side is either side effect free or considered to be side effect free by the user, the operation can be replaced with a `number` literal with the value of $+0_{\mathbb{F}}$. This rule also applies to values of the type `BigInt` with the limitation of there only being a single value for zero ($0_{\mathbb{Z}}$).

5.2.3.5 Bitwise OR

Bitwise OR also takes two operands, `x` and `y`, and will calculate a value that has each bit set, that is set in either `x` or `y`. Returning to the previous example values, the expression `0b1000 | 0b1111` can be constructed. The result of the above expression would be `0b1111` as the right-hand side already has every bit set.

There are no inherent exceptional cases defined for the bitwise OR operator, similar to the bitwise AND operator. However, the first opportunity for constant folding can be derived from the above example. When either side has all available bits set, applying the bitwise OR operator will always lead to the resulting value having all bits set.

This rule allows folding all occurrences of `x | -1` or `-1 | x` to `-1` if `x` is side effect free. This transformation also works for `BigInt` values, as negative values are implemented "as having bits set infinitely to the left"[12, § 6.1.6.2].

A second opportunity for constant folding arises from the number zero or the values turned into zero when operands of the type `number` are mapped to 32-bit integers. This opportunity suffers from the same limitations the AND operator features when folding expressions where one side has every bit set. To be correctly folded, the remaining side would still have to be converted into a 32-bit integer. The shortest way of applying this would be another bitwise operation with a constant value.

This limitation, again, does not apply to values of the type `BigInt` with the same reasoning as there is no implicit type conversion.

5.2.3.6 Bitwise XOR

Bitwise XOR takes two operands, `x` and `y`, and calculates a value with all bits set in one operand but not the other one. If the bits of a numeric value were considered a set, the XOR operator would work as the symmetric difference between the two sets. Returning to the established example of `0b1000` and `0b1111`, this would yield the expression `0b1000 ^ 0b1111`, which evaluates to `0b111`.

As with the other bitwise operators analyzed so far, the standard does not specify any special rules that could be used to apply constant folding. The opportunities for this arise, again, from the mathematical properties of the operator.

The first value is zero, as all bits set on one side would result in the result.

However, this particular case suffers from the implicit conversion to a 32-bit integer for values of the `number` type again. Values of the `BigInt` type are, again, unaffected.

The XOR operator has a special rule when both sides are equal. This situation causes the result to become zero, as every bit set on one side is also set on the other side. This rule is not limited to values of the `BigInt` type as the resulting value is a constant that is unaffected by the conversion to a 32-bit integer. This behavior has the somewhat non-obvious consequence that `NaN ^ NaN` will evaluate to zero. A limitation of this approach is that the type of the resulting zero will have to be guessed in most cases. The deobfuscation program defaults to `number` here.

5.2.3.7 The Left Shift Operator

The left shift operator also takes two operands called `x` and `y`. Unlike the previous bitwise operators, however, the left shift, and for that matter, all shift operators, are not commutative. That means that `x << y` is not necessarily equal to `y << x`. The operator serves as both arithmetic, and logical left shift as those operations are equivalent. This equivalence also explains the absence of an unsigned left shift operator like <<<.

When considering the implementation of the operator for the `number` type, `x`, the left operand is converted to a 32-bit integer using the previously introduced abstract operation $ToInt32$. However, the right-hand side operand, `y`, is converted into an unsigned 32-bit integer using the abstract operation $ToUint32$. Similarly to its signed counterpart, it also maps $+0_{\mathbb{F}}$, $-0_{\mathbb{F}}$, $-\infty_{\mathbb{F}}$, $+\infty_{\mathbb{F}}$, and NaN to $0_{\mathbb{Z}}$. All other values are rounded down to the nearest integer that is not larger than the input value.

After conversion, the remainder of `y` with the divisor 32 is calculated. This value is then used as the actual value for shifting. The standard does not define any specific shortcut rules for this operator. There are, however, a few that arise from the mathematical properties. One of them is the shifting of zero as in `0 << x`. If `x` is side effect free and evaluates to a value that is not of the type `BigInt`.

The implementation for values of the type `BigInt` requires, as with all previous operators, that both sides are of the type `BigInt`. Otherwise, a `TypeError` exception will be thrown. A significant difference between the implementations for both numeric types is that the one for `BigInt` allows for the `y` parameter to be negative. The calculation result then depends on the sign of the left operand. If $y < 0_{\mathbb{Z}}$, the result will be equal to $\mathbb{R}(x)/2^{-y}$. In all other cases, the result will be equal to $\mathbb{R}(x) * 2^{y}$. Applying this constant folding, which assumes that the binary operation is not used to throw a `TypeError` exception implicitly, allows turning bit shifts into either division or multiplication. The assumption here is that a human can intuitively understand multiplication and division, so this transformation should be applied.

Another opportunity for constant folding for values of the `number` type arises from considering the case of a shift by zero or a shift by a number that divides by 32 with a remainder of zero. However, doing this will still convert the left-hand side operand into a 32-bit integer, which leads to the same issues as already explained with the bitwise operators. However, one simplification that can be done here is to apply the modulo operation to `y` so that any values larger than 32 are converted to the equivalent smaller value.

This zero shift optimization also applies to the `BigInt` implementation, except there is no wrap-around for the `y` parameter. Because there is no conversion to a 32-bit integer, no transformation is applied to the `x` parameter. The lack of this restriction means that expressions like `x << 0n` can be folded to `x`. This risks `x` not evaluating to a value of the type `BigInt`, however, which would throw a `TypeError`.

One more opportunity for values of the `BigInt` type that can be used for constant folding is the equivalency of multiplication and left shifting. Essentially, a left shift like `x << y` may be rewritten as $x \cdot b^y$, with b being the base of the underlying numeric system. For ECMAScript, this is binary, so the resulting equivalent expression is $x \cdot 2^y$. The expression `4n << 5n` could be rewritten as `4n * (2n ** 5n)`, which can subsequently be simplified to `4n * 32n`, which can then be further simplified to 128. This transformation can not be applied if the exponent is negative, as negative exponents are not allowed for values of the type `BigInt`.

Another opportunity exists for `BigInt` when the left-hand side is `0n`. The result will always be `0n` regardless of which `BigInt` value is on the right-hand side.

5.2.3.8 The Right Shift Operators

The right shift operators (`>>` and `>>>`) act as the counterpart to the left shift operator discussed in section 5.2.3.7. Both operands, `x` and `y`, are converted into a 32-bit signed and unsigned integer respectively.

The constant folding opportunities for both `number` and `BigInt` with `y = 0` apply the same as with the left shift operator.

The `BigInt` implementation for the signed right shift operator invokes the left shift operator while inverting the sign of `y`. This means, that `x >> y` may also be written as `x << -y`, when `x` and `y` are of the type `BigInt`. The program may also assume that both parameters are of the type `BigInt` if one parameter is known to be of the type `BigInt`. This transformation is applied if a literal value for either `x` or `y` is present to re-use pre-existing code for the left shift operator in a separate pass.

There exist two distinct right shift operators for values of the type `number`. The first is the signed right shift, which will shift all bits right while filling empty slots with the sign bit. The sign bit itself is preserved in the most significant bit. `-(0x80000000)>>31` will consequently evaluate to `-1`, the number that has all 32 bits set.

However, the unsigned variant will shift in zero bits instead of the sign bit. `-(0x80000000)>>31` will therefore evaluate to `1`, the number with only the least significant bit set.

`BigInt` does not implement the unsigned right shift as the sign bit is considered to be infinitely far to the left in `BigInt` values. It is not involved in shift operations and will be ignored. `x >>> y` will throw a `TypeError` if either operand is of the type `BigInt`.

5.2.4 Binary Logical Operators

The following operators define the logical conjunction and disjunction operators, which can be used on expressions.

5.2.4.1 Logical AND operator

The logical AND operator (&&) allows linking two expressions and will return **`true`** if both expressions evaluate to **`true`**. It employs short-circuiting, which means that if the left expression evaluates to **`false`**, the operator will short-circuit, and the right-hand-side expression will not be evaluated. Not evaluating the right-hand side means the whole expression can be replaced with a literal boolean, **`false`**. If the right side evaluates to **`false`** statically, the expression can be replaced by the left expression. In some contexts, this might require explicit boolean coercion by prefixing the expression with `!!` though. This explicit coercion works because of the nature of the unary logical negation operator coercing its target into the type **`boolean`**. The second unary negation operator undoes the first one's negation of the target expression. Those extraneous operators may be removed if the outer statement or expression already implicitly coerces the value to be boolean. The trivial folding would be when the left-hand side and the right-hand side expressions are side effects free and evaluated to **`true`** at compile time. In this case, the whole expression may be replaced with a literal boolean, **`true`**.

5.2.4.2 Logical OR operator

The logical OR operator (||) takes two expressions and coerces them to the boolean type. If either operand evaluates to **`true`** the whole expression will evaluate to **`true`**. It employs short-circuiting when the left expression evaluates to **`true`**. Short-circuiting here leads to the right-hand side expression not being evaluated at all. The right-hand side not being evaluated can be used for constant folding in that if the left-hand side expression statically evaluates to **`true`** the whole expression can be replaced with a literal boolean **`true`**.

5.2.4.3 Conditional operator

The conditional operator, also known as the ternary operator in other programming languages, is a way of expressing a condition within an expression. It is formed with a test expression `x` coerced to **`boolean`**, an expression `y` that is evaluated when `x` evaluates to **`true`**, and an expression `z` that forms the other part and is evaluated if `x` evaluates to **`false`**. Unlike the **`if`** statement, the else expression is not optional. If `x` can be evaluated statically and without side effects, the conditional expression may be replaced with the resulting expression. No special rules are defined for this operator.

5.2.5 Relational operators

ECMAScript provides a range of relational operators that set two expressions in relation to each other. The result of these expressions is always of the type **`boolean`**, even though the abstract operations and algorithms invoked may return `undefined`.

5.2.5.1 Less than operator

The less-than operator, denoted by <, provides relational tests on whether the left-hand side expression evaluates to less than the right-hand side expression. It is the first occurrence of the "Abstract Relational Comparison" algorithm[12, § 7.2.13], which implements a less-than operation for all types. All other relational operators can be derived from inverting and combining the result with an equality check.

The algorithm used to compare values has to be visited first to understand this operator and the ones building off it. The following paragraphs will explain the steps the algorithm takes to come to a result.

The "Abstract Relational Comparison" accepts two values called `x` and `y` and a flag called `LeftFirst`, which defaults to **`true`** and specifies the order in which the arguments are evaluated. Depending on the order defined using the `LeftFirst` parameter, the operands are passed to the previously introduced abstract operation $ToPrimitive$. The possible return values of this algorithm are **`true`**, **`false`**, and `undefined`.

If both `x` and `y` are of the `string` type after being passed to $ToPrimitive$, the abstract operation $IsStringPrefix$[12, § 7.2.9] will be invoked. It also takes two operands, `p` and `q`, and returns a boolean value indicating whether `q` "can be the string-concatenation of `p` and some other String `r`"[12, § 7.2.9]. If that is the case, **`true`** is returned and **`false`** otherwise. The first invocation of this operation is done with the primitive versions of `x` and `y` in inverted order. `y` being a prefix of `x` implies that either `x` and `y` are equal or that `y` is a prefix of `x` and shorter in length. Both cases lead to the less-than operator evaluating to **`false`**. Therefore the algorithm returns **`false`** as well.

$IsStringPrefix$ is invoked a second time afterward with the parameters in the same order as they were passed to the operator. If this evaluates to **`true`**, the whole algorithm returns **`true`** because the previous check already established that `x` is neither equal to nor longer than `y`. Therefore, `x` has to be an actual prefix of `y` and is considered less than `y` by the semantics of ECMAScript.

If both checks fall through, the first index where both strings differ is determined. The code point at this index is then compared to determine the result of the operation.

If one side is of the type `BigInt` and the other side of the type `string`, the algorithm will try to transform the `string` value to `BigInt`. In the case of this transformation succeeding, the `BigInt` operand and the transformed `string` value will be passed to `BigInt::lessThan`, where the comparison is equivalent to the less than operator on $\mathbb{Z}$.

If the algorithm has not returned a result so far, both operands will be transformed to one of the numeric types using the previously introduced abstract operation $ToNumeric$. If the resulting types of both operands are equal, either `Number::lessThan` or `BigInt::lessThan` will be invoked.

`BigInt::lessThan`, as per the previous definition, has no exceptional cases, and as such only trivial constant folding where both values are known can be applied. `Number::lessThan`, however, given the nature of IEEE 754 floating-point numbers, has a plethora of exceptional cases to consider when evaluating[12, § 6.1.6.1.12]:

1. If either side evaluates to `NaN`, the result will be `undefined`
2. If both operands are the same Number value, the result will be **`false`**
3. If one side is $+0_{\mathbb{F}}$ and the other side is $-0_{\mathbb{F}}$, the result will be **`false`**
4. If `x` is $+\infty_{\mathbb{F}}$, return **`false`**.
5. If `y` is $+\infty_{\mathbb{F}}$, return **`true`**.
6. If `y` is $-\infty_{\mathbb{F}}$, return **`false`**.
7. If `x` is $-\infty_{\mathbb{F}}$, return **`true`**.

If all of these exceptional cases fall through, the values of `x` and `y` are compared using the less than operation while projected into $\mathbb{R}$.

The first constant folding opportunity derives from rule number one. If either side is `NaN`, the operation may be replaced by a boolean literal of the value **`false`**. `undefined` is converted into **`false`** by the semantics of the less-than operator after returning from the algorithm. This case is also limited to expressions where the other side is side effect free.

The next opportunity that does not require knowledge of both parameters arises from rules 4 through 7. The logic behind those rules is that no values are beyond infinity, so the less-than operation may short circuit and end early. This behavior is consistent with the IEEE 754 standard. From this, expressions like `Infinity < x` and `x < -Infinity` may be replaced with a boolean literal with the value **`false`** if `x` is side effect free. In theory, it would also follow that expressions like `x < Infinity` and `-Infinity < x` could be replaced with **`true`** given the same conditions on `x`. However, `x` may evaluate to `NaN`, which would make rule number one apply instead, which would make the operation return **`false`** instead.

If the numeric types of `x` and `y` differ after their transformation, the same rules are repeated to filter out `NaN` and $\pm\infty$. The values of `x` and `y` are then projected into $\mathbb{R}$ and compared using the less-than operator. This handling of type differences breaks with the previous behavior of binary operators throwing a `TypeError` if the numeric types of the operands differ. The problem of a loss in precision can be ignored since the result will be of the type **`boolean`** regardless.

5.2.5.2 Greater than operator

The greater than operator also invokes the “Abstract Relational Comparison” algorithm but switches the order of its operands. In addition to that, it also sets the `LeftFirst` flag, which defaults to **`true`**, to **`false`** so that `x` will be evaluated first. This reversal is done to conserve the left-to-right property of expression evaluation in ECMAScript[12, § 7.2.13]. A result of `undefined` from the relational algorithm will be transformed to **`false`**. In all other cases, the result of the algorithm is returned. The constant folding opportunities and the compatibility of the two numeric types from the less-than operator also apply to the greater-than operator.

5.2.5.3 Less than or equal operator

The “Abstract Relational Comparison” algorithm also synthesizes the less-than-or-equal operator. The opposite of the less-than-or-equal operator is the greater-than operator, which allows negating the result of the greater-than operator. The greater than operator is not invoked directly but rather copied, with the only difference being that the result of the algorithm is mapped to the return value of the operator in a different way. `undefined` and **`true`** make the operator return **`false`** while a result of **`false`** from the algorithm will make the operator return **`true`**.

Not doing a true equality check here leads to some logically inconsistent behavior where `[] < []` evaluates to **`false`**, `[] == []` evaluates to **`false`**, and `[] <= []` evaluates to **`true`**. `[] < []` evaluating to **`false`** is due to both operands being passed to $ToNumeric$, which leads to both values becoming $+0_{\mathbb{F}}$. The same transformation happens in the case of `[] <= []`, which makes the operations trivially understandable because while $+0_{\mathbb{F}} \leq +0_{\mathbb{F}}$ is true, $+0_{\mathbb{F}} < +0_{\mathbb{F}}$ is not. The behavior of the equality operation not returning **`true`** in this case will be analyzed in section 5.2.6. The constant folding opportunities and the compatibility of the two numeric types from the less-than operator apply with the less than or equal operator.

5.2.5.4 Greater than or equal operator

The greater than operator is implemented in the same way as the less-than operator par for the switching `x` and `y` when they are passed to the “Abstract Relational Comparison” algorithm. `LeftFirst` is not changed to **`false`** but instead left at its default value of **`true`**. The constant folding opportunities and the compatibility of the two numeric types from the less-than operator apply with the greater-than-or-equal operator.

5.2.5.5 `instanceof` operator

The **`instanceof`** operator takes an object instance (`V`) and a constructor (`target`) as arguments and follows a “generic algorithm for determining if `V` is an instance of [the] target either by consulting `target`’s `@@hasInstance` method or, if absent, determining whether the value of `target` ’s”prototype” property is present in `V`’s prototype chain”[12, § 13.10.2]. A `TypeError` is thrown, if `target` is not an object.

The constant folding opportunities for this operator are limited as it is usually used in conjunction with two identifiers. Trivial cases, where the left-hand side is a literal value, may still be folded to **`false`**.

5.2.5.6 `in` operator

The `in` operator takes an object instance on the right-hand side and an expression on the left-hand side, which will be transformed into a value of the type `string` if it is not already of the type `string` or `Symbol`. The result of the operator is a boolean value indicating whether the object instance has a property with the key passed in the left-hand side operand. If that is not the case, the prototype chain of the object instance will be traversed in search of the property. If the right-hand side is not an object, a `TypeError` will be thrown. There are no constant folding opportunities known to the author that can be meaningfully applied here.

5.2.6 Equality operators

ECMAScript defines two kinds of equality operators. Strict comparison short-circuits and will immediately return **`false`** if the types of the operands are not equal, while loose comparison allows for differing types.

5.2.6.1 Strict equality operator

The string equality operator (===) has a left and a right-hand side operand and passes these to the "Strict Equality Comparison" algorithm defined in section 7.2.15 of the ECMAScript standard. If the types of both operands are not equal, the algorithm returns **`false`** immediately. If the left-hand side operand is either of the type `number` or `BigInt`, `Number::equal` or `BigInt::equal` are invoked, respectively. The right-hand side operand does not have to be checked separately since the first check already ensured it is of the same type.

`BigInt::equal` returns **`true`** if the two operands have the same value when mapped into $\mathbb{R}$ and **`false`** otherwise.

`Number::equal`, on the other hand, defines special rules to deal with the special values found in $\mathbb{F}$[12, § 6.1.6.1.13]:

1. If either side is `NaN`, the result will be **`false`**.
2. If both operands are the same Number value, return **`true`**.
3. If one side is $+0_{\mathbb{F}}$ and the other side is $-0_{\mathbb{F}}$, return **`true`**.

If none of these rules apply, **`false`** is returned.

In the case of the operands being of a non-numeric type, the operands are instead passed on to the abstract operation $SameValueNonNumeric$. Depending on the input type of the operands, a few different paths are applied:

- `undefined`: the result will be **`true`**. All occurrences of `undefined === undefined` may therefore be replaced with **`true`**.
- **`null`**: the result will be **`true`**. Therefore, all occurrences of **`null`** `===` **`null`** may be replaced with **`true`**.
- `string`: the sequence of code units for both strings has to be equal for the operation to return **`true`**.
- **`boolean`**: the operation acts as an `XNOR` by only returning **`true`** if both operands evaluate to **`true`** or both operands evaluate to **`false`**.
- `Symbol`: only returns **`true`** if both operands are the same Symbol value.

Values of the type `Object` are compared by their object value, which means the memory location of the object. The expression `{} === {}` will evaluate to **`false`** because the two operands are two distinct object instances. Given the example

```
let a = {};
let b = a;
```

, `a === b` will evaluate to **`true`** because both identifiers reference the same underlying object. This also explains the curious behavior of `[] == []` evaluating to **`false`** in section 5.2.5.3.

The most straightforward opportunity for constant folding here is if either side of the operator is `NaN` since the other side will either also be of the type `number`, which will result in the comparison returning **`false`**, or the other side could be of a different type, which would also make the comparison return **`false`**.

5.2.6.2 Strict inequality operator

The strict equality operator (`!==`) simply acts as syntactic sugar for combining the unary minus operator with the strict equality operator. The same abstract operation as the strict equality operator is invoked, but the result is negated.

5.2.6.3 Loose equality operator

The loose equality operator (`==`) invokes the "Abstract Equality Comparison" algorithm to check its operands, `x` and `y`, for equality. This algorithm returns **`true`** or **`false`**[12, § 7.2.14].

The first step in the algorithm is to check whether `x` and `y` are of the same type. If that is the case, the "Strict Equality Comparison" algorithm is used to determine equality. Therefore, the remaining cases assume that `x` and `y` have distinct types. The ECMAScript spec defines several rules to determine equality[12, § 7.2.14]:

1. If either operand is **`null`** and the other is `undefined`, return **`true`**.
2. If either operand is of the type `string` and the other is of the type `number`, the `string` operand will be converted to `number` using the abstract operation $ToNumber$. Strict comparison is then implicitly invoked because both operands are of the same type now.
3. If either operand is of the type `string` and the other is of the type `BigInt`, the `string` operand will be converted to `BigInt` using the abstract operation $StringToBigInt$. Strict comparison is then implicitly invoked because both operands are of the same type now.
4. If either operand is of the type **`boolean`**, the **`boolean`** value is converted to `number` using the abstract operation `ToNumber` and then compared again to the other side invoking the algorithm from the start.
5. If either operand is of the type `Object` and the other is of any of the types `string`, `BigInt`, `number`, or `Symbol`, the operand of the type `Object` will be passed to $ToPrimitive$. This conversion guarantees that the `Object` value will no longer be of the type `Object` after trans-

formation.

6. If the operands are of different numeric types (e.g., `x` is a `BigInt` and `y` is a `number`), the numbers are projected into $\mathbb{R}$ and then compared. If either side is `NaN` or $\pm\infty$, the result will be **`false`**.

If none of the above rules apply, **`false`** will be returned instead. These rules do not allow for any constant folding to be applied.

Comparing an expression to a boolean literal can be simplified if the boolean literal that is being compared is **`false`**. An example would be `x == false`, which can be transformed to `!x`. The other way, comparison to a boolean **`true`** literal does not work due to the implicit type conversion to `number`. For example, the expression `x == true` would evaluate to **`false`** if `x` was `123`, since **`true`** would be converted into $+1_{\mathbb{F}}$ making the comparison `1 == 123`.

5.2.6.4 Loose inequality operator

Like the strict inequality operator, the loose inequality operator (`!=`) negates the associated equality operator. This does invert the constant folding opportunity `x == false` to `x != true`.

5.2.6.5 Assignment operators

ECMAScript defines a plethora of assignment operators, which combine the assignment of a value to an identifier with an operation. Since assigning a value is never free of side effects, it is of no use to search for constant folding opportunities. The assignment operators are therefore only included for the sake of completeness.

5.2.7 Comma Operator

The comma operator is a construct that allows the chaining of expressions using a comma character. The result of the evaluation of the comma operator will be the result of the last expression in the chain. Since this construct combines multiple expressions, there is little opportunity for constant folding that is not handled further down the tree. One way of removing potential obfuscation here would be to eliminate side-effect-free expressions from the chain as long as they are not the last expression in the chain. This proof can be accomplished using the side effect algorithm defined in section 5.2.1.4.

5.3 Inlining

Inlining within this thesis describes the process of statically replacing a variable with its value. It only makes sense to inline variables in scenarios where the variable does not store the result of an expensive computation. In the context of programming languages that produce a native binary, this term also refers to the process of inlining functions to improve the performance of a program.

The effectiveness of this transformation will be demonstrated in the following code snippets.

```javascript
const _0x5351 = function(url, whensCollection) {
    /** @type {number} */
    url = url - (9141 + 898 * 7 + -14939 * 1);
    let _0x1e802f = _0x3806[url];
    return _0x1e802f;
};
```

It becomes apparent immediately that the numeric constant has been artificially expanded to make the code harder to understand. This expansion was caused by the obfuscation software's "Numbers To Expressions" setting. Using the strategies shown in the chapter on constant folding simplifies the numeric constants to an easier-to-understand value. Since all operands are literal values, the result of the expression can be determined trivially. These transformations applied to the code sample would change it to the following:

```javascript
const _0x5351 = function(url, whensCollection) {
    /** @type {number} */
    url = url - 488;
    let _0x1e802f = _0x3806[url];
    return _0x1e802f;
};
```

The current chapter will try to further simplify this code by inlining variables. In this example, the variable `url` could be inlined to simplify the code as follows:

```javascript
const _0x5351 = function(url, whensCollection) {
    let _0x1e802f = _0x3806[url - 488];
    return _0x1e802f;
};
```

Applying the inlining again the code could again be simplified by eliminating the temporary variable `_0x1e802f` entirely:

```javascript
const _0x5351 = function(url, whensCollection) {
    return _0x3806[url - 488];
};
```

Inlining in this thesis is implemented using static scope analysis. This analysis essentially attempts to rebuild what the known variables at a given point within the program are.

The implementation for this builds on the existing abstract syntax tree traversal infrastructure. It filters the abstract syntax tree for all statements and expressions that allow introducing new variables or those that create a new scope. The tree nature of the abstract syntax tree along a stack structure facilitates this analysis as visiting a node may push new scopes onto the stack. The newly created scope is visible to all child nodes of the node that created the scope. Once the visitor leaves a node that created a scope, the scope is popped from the stack structure.

All statements and expressions that create new scopes will be listed in the following, beginning with the **BlockStatement**. It already covers a large part of all available statements as it serves as a child node to many of them. No intrinsic variables are added to the scope it creates.

The root statement, which may either be a script or a module, functions virtually the same except for the addition of the intrinsic global object. It takes different identifiers based upon the execution environment. In browsers, for example, it is named `window`.

The following three scope-creating statements and expressions will be grouped as they function similarly. The expressions and statements in question are the **function expression**, the **arrow function expression**, and the **function declaration**. All of them allow the declaration of a new function with an optional list of parameters. A new scope containing the parameters is created when a function declaration is visited. These parameter variables are marked as parameters within their scope, as inlining parameters is impossible because their values are never known at the point of declaration. However, parameters may be inlined at the point of invocation if all invocations specify the same set of statically evaluatable parameters. This behavior is explained in section 5.5.2.

Function declarations add their identifier to the scope that encloses them. They essentially become variables of the type `object`.

In addition to the function parameters, there is also an intrinsic variable with the identifier `arguments`. This variable does not exist for arrow functions. It provides array-like access to the function's parameters and is used in functions that accept a variable number of parameters, such as the `console.log` function. Array access is, as previously explained, very hard to trace, and this thesis will therefore exclude the `argument` object from its scope. The introduction of rest parameters has largely superseded its usage. The deobfuscation software will abort with an error asking the user to manually replace the `argument` access if its usage is detected.

The following statement in the list is the **TryStatement**, which consists of a block statement for the `try` part. It must also have a `catch` clause, a `finally` block, or both. The `finally` block is also a block statement and therefore needs no special handling if present. However, the `catch` clause provides an optional parameter, an identifier that the programmer may specify to bind the exception that caused the `catch` clause into its scope. This identifier is added to a separate scope and marked as a parameter as it may not be inlined either.

The penultimate group of statements that may open a new scope consists of variations of the `for` statement. ECMAScript provides two different derivations in strict mode for the `for` statement. The first one consists of three optional expressions, with the first one being evaluated before the first iteration is done. This expression is usually used to assign a starting value for the iteration to an existing identifier. The second expression is the loop condition. Its absence will be interpreted as a loop condition that continuously evaluates to `true`. `for(;;){}` is therefore an infinite loop. The final expression will be evaluated whenever an iteration has been finished. This variant does not add any intrinsic variables to the scope of its body.

The second variant has a non-optional list of variable declarations as its first part, followed by the same two optional expressions from the previous variant. The variables declared within the `for` statement are added as parameters as they may not be inlined trivially.

The final group is the iterating `for` statements, called $ForInOfStatement$ within the ECMAScript specification. Both introduce a single variable into their child's scope that may not be inlined. The `for of` construct iterates over objects that define `Symbol.iterator`, such as the built-in classes `Map`, `Set`, and `String`. `for in` constructs iterate over the keys of the supplied object instead.

The list of the statements that may create a new scope within ECMAScript is now concluded. A few ways that an ECMAScript programmer may declare variables have already been mentioned, but that list is not exhaustive. It is imperative to mention the declaration statement to finalize that list.

A declaration consists of a declaration type and a list of bindings. The declaration type may be `var`, `let`, or `const`. The `let` type indicates that the variable is mutable and may be reassigned. A variable declared as let may defer its initialization to a later point in the scope. Until it is initialized, its value will be `undefined`. `const` indicates that the variable is constant and therefore immutable. Attempting to defer the initialization of a variable that is declared as `const` will cause a syntax error to be thrown. Variables declared as `var` are also mutable. The main difference to the `let` type is that the variable is hoisted. This term means that the location of the variables' declaration in the code does not matter. It is equivalent to being declared at the beginning of the scope. This functionality works similarly to how functions work in ECMAScript, where a function may be invoked before it is declared. Hoisting is implemented by walking the abstract syntax tree twice and adding all variables declared as var into their scopes on the first traversal. The deobfuscation program ensures that they are declared before any potential usage of the variables on the second traversal. Every scope is assigned an id. This id is then used to match the scopes of the first traversal with those from the second traversal. The initialization of these variables is not hoisted, and the values of the variables will be `undefined` until the initialization is reached.

The bindings may consist of an identifier or a binding pattern, which a programmer may use to destructure objects. Binding to an array and using the rest element is also possible.

Once a variable is used, the lookup works by traversing the stack of scopes starting from the topmost stack. Any variable usage is recorded within a list associated with the variable on the scope object. It is valid in ECMAScript to assign to an undeclared variable. This assignment will be interpreted as an implicit declaration of the variable as a property of the global object. It is considered an error if this process can not find a variable along the scope chain, which may cause the deobfuscation software to generate invalid code.

If an already initialized variable is reassigned within the code, it is marked as not inlineable as doing so would require intricate knowledge about the control flow within the program.

Scopes are tagged with the node that created them. This tagging ensures that scopes may only be popped from the stack when the node it was created by is left. This safety measure allows the deobfuscation software to prevent programming mistakes that imbalance the scope stack.

The actual inlining is performed once a scope is left. Leaving the scope will remove variables defined within that scope from the set of known variables. Therefore, all usages of these variables must have been visited already.

The current inlining implementation only inlines variables that have been used a total of once. This limit is set to reduce the size of the source code instead of increasing it. The implementation considers only variables that were initialized using literal values. A user may optionally add identifiers to this check, but this is done at the user's risk as this may cause visible side-effects to be moved within the control flow.

If a variable has been successfully inlined, the program will try to remove its declaration as it is now unused.

Calculating the number of nodes on the abstract syntax tree inlining would save is not a fitting metric, as only literal values (or identifiers) are currently being inlined. As such, the calculation would always allow the inlining to happen since all usages are identifiers with a magnitude of one being replaced by literal values (or identifiers) with an extent of one node. The positive result for the inline check would stem from removing the declaration of the variable to be inlined.

5.4 Dead Code Elimination

Dead code elimination within this thesis is based upon the MIT-licensed implementation found in the closure compiler, which was introduced in section 2.3. That implementation is based partly upon Söderberg et al.'s paper "Declarative Intraprocedural Flow Analysis of Java Source Code"[30]. It is implemented by constructing a control flow graph within the abstract syntax tree. In the paper, Söderberg et al. describe a control flow analysis framework for the Java programming language implemented in less than 300 lines of code. They scope their work, as the title implies, on intraprocedural analysis. Therefore, they only construct a control flow graph within functions. This implementation is done by extending the formal grammar using attributes. Söderberg et al. utilize an attribute grammar for the Java programming language to extend the definition of all statement nodes through four additional attributes. These attributes consist of two sets of links and two higher-order attributes. For this explanation, higher-order attributes are attributes that may themselves have attributes. They are used to signify the entries and exits of a method in the control flow graph as that simplifies the flow analysis, according to Söderberg et al. ECMAScript does not offer multiple entries for methods, but a method may contain numerous exits, e.g., by utilizing the return statement. Therefore, this thesis only utilizes a virtual exit node to consolidate the exits of a method.

The two sets included as attributes are the successor and predecessor sets. The successor set contains all statements that may follow this statement, while the predecessor set includes all statements that precede a particular statement. Both of these sets may be calculated from the other one, so the closure compiler elected to only implement the successor set. This thesis elected to include both sets explicitly to simplify the processing of the resulting graph.

Implementing the control flow graph in such a way allows authors of syntax-directed transformation tools to reason about the control flow of an application without having to transform the abstract syntax tree into another structure. These transformations would significantly increase the complexity of the tool as a way of transforming the control flow graph back into an abstract syntax tree would have to be implemented as well.

Dead code elimination becomes trivial once the control flow graph has been constructed. The deobfuscation program may purge all statements with an empty set of predecessors as they are never invoked. The deobfuscation program may disregard the side effects of the purged code since the code is never invoked in the first place. The only side effect that the obfuscated application could observe is the previously encountered integrity checking using the enclosing function's `.toString()` method. No protection against this kind of integrity checking is provided by this thesis as it is considered an edge case. A reverse engineer may manually overwrite the `.toString()` method to intercept attempts to perform this integrity check.

The following chapter will describe the construction of an intraprocedural control flow graph in ECMAScript by explaining how the successor sets are calculated per statement. While not explained explicitly, the predecessor set is calculated simultaneously by inverting the direction of each generated link between nodes.

5.4.1 Constructing a control flow graph

To explain the construction of a control flow graph, all statements defined in the ECMAScript standard will have to be examined for their control flow altering properties.

The **BlockStatement** groups zero or more statements together and operates them in sequence. The control flow moves on to the subsequent statement. "No matter how control leaves the Block the LexicalEnvironment is always restored to its former state"[12, § 14.2.2].

DeclarationStatements consist of either the `let` or `const` keyword and one or more bindings in a binding list. Each binding in the binding list may have an initializer expression, which, when evaluated, forms the initial value for the variable. Variables marked `const` produce a `SyntaxError` if no initializer is provided. A DeclarationStatement jumps to the following statement after completion. This statement can be taken from the `next` attribute. Still, a link to a potential exception handler will have to be established as the initializer expressions may throw through various means. The link to the exception handler may be skipped if no initializer expressions are provided or if the expressions can not throw.

A **VariableStatement** works the same way as the previous statement from a control flow perspective but uses the `var` keyword instead of `let` or `const`.

The **EmptyStatement** does nothing and jumps to the following statement. The deobfuscation program may eliminate it from the control flow graph and the abstract syntax tree.

ExpressionStatements can be used to evaluate an expression while discarding the result. It simply jumps to the following statement after completion. However, the expression may throw, which would lead to the execution jumping to an exception handler or a program abort. The construction of the control flow graph may omit this link if the expression can not throw.

The **IfStatement** provides a test expression, a then block, and optionally an else block. The then block is executed if the expression evaluates to `true` and optionally. The else block is executed if the condition expression evaluates to `false`. Both blocks receive a link on the control flow graph to the statement following the if statement. The condition expression receives two links, one on `true` and one on `false` to the respective blocks. Another link is added if an exception handler is present in combination with a conditional expression that may throw.

DoWhileStatements consist of an expression and a block statement. The block statement is executed once and then repeated until the condition evaluates to `false`. One link is added from the preceding statement to the block statement. One link is added to the expression, and a conditional link from the expression back to the block statement. A supplemental link is added in the presence of an exception handler to accommodate the possibility of the condition throwing an exception. A final conditional link is added from the expression to the following statement since the condition will either evaluate to `true` forever or evaluate to `false` at some point. Infinite loops that may occur if a condition never evaluates to `false` are not handled during the control flow graph construction. Every expression, even a literal `true` value, is considered to become `false` eventually.

The **WhileStatement** works similarly to the previous statement. The main difference is that the statement's body is not run unconditionally before the condition is evaluated. The control flow graph construction must add a link from the expression to a possible exception handler if the expression could throw an exception. A conditional link, followed if the condition evaluates to `true`, is added from the expression to the block statement. Another conditional link is added from the expression to the following statement of the WhileStatement. A final link is added from the block statement to the condition, which builds the looping property of the WhileStatement.

The control flow within a **ForStatement** depends on the three expressions supplied to it. The header of a `for` loop consists of zero-or-more variable assignments or declarations. However, a programmer may not mix declarations and assignments within the same `for` loop. The second expression is the test expression. It is also optional, and its absence will be interpreted as continuously evaluating to `true`. The third expression is the loop expression. It is most commonly used to increment or decrement the variables from the first expression. It is also optional. The control flow goes from the declarations or assignments to the loop expression. From there on, two links are added. If the condition evaluates to `true`, the control flow is transferred to the body of the `for` loop. If the condition evaluates to `false`, the control flow instead transfers to the statement following the `for` loop. These scenarios are represented accordingly by conditional links on the respective nodes. After reaching the end of the `for` loops body, the control flow is transferred to the loop expression and subsequently to the loop condition. All three conditions must be connected to the exception handler if one is present, and the deobfuscation program can not rule out that the expression will not throw.

Another variant of `for` loops is provided by the so-called **ForInOfStatement**. The differences between `for in` and `for of` loops are discussed in another chapter. The distinction does not matter here. The control flow initially transfers to the header of the ForInOfStatement, which may throw. The control flow may then be transferred to the statement's body if any objects are left to iterate or to the statement following the ForInOfStatement. The end of the ForInOfStatements body is connected to the expression to model this statement's looping nature correctly.

The **ContinueStatement** allows skipping to the next iteration of a loop. "It is a Syntax Error if this ContinueStatement is not nested, directly or indirectly (but not crossing function boundaries), within an IterationStatement"[12, § 14.8.1]. As such, the enclosing iterating statement should always be able to be located statically. A single link is drawn from the ContinueStatement to the condition of the enclosing iteration statement. If the control flow graph construction can find no enclosing iteration statement, an error is thrown, and the processing is halted as the program expects valid ECMAScript code as input. A ContinueStatement has an optional parameter containing a label of an enclosing loop. This label allows skipping an iteration in an outer iteration statement by explicitly specifying the label of the outer iteration statement.

The **BreakStatement** allows aborting an enclosing iteration statement. It requires the same enclosing iteration statement, which does not cross function boundaries[12, § 14.9.2]. Similar to the ContinueStatement, it also provides an optional parameter containing a label of an iteration statement. The execution moves to the following statement of the iteration that has been aborted.

A **ReturnStatement** is the equivalent for functions of what a break statement is for iteration statements. It always returns a value to the caller of the function. If the programmer does not specify a value, `undefined` is returned as a default value. The control returns to the caller, but as this thesis does not construct inter-procedural control flow graphs, the branch of the graph ends when a return statement is encountered.

The **WithStatement** has an expression and a statement as parameters. The expression is evaluated, and the properties are added to the front of the unqualified name lookup mechanism. An example of this would be

```
with (console) {
  log("Hello World!");
}
```

This statement makes it exceedingly hard to analyze variable data flow statically or to perform liveness analysis and will, as such, lead to an error in the deobfuscation program. This limitation is justified as the statement is not recommended to be used either way. The ECMAScript standard underlines this, as using this statement in strict mode code will lead to a `SyntaxError` being thrown[12, § 14.11.1]. If the deobfuscation program included the `with` statement, the control flow would go from the preceding statement to the expression and then to the enclosed statement.

The **SwitchStatement** provides a C-like **`switch`** construct that requires an expression as an argument and accepts zero or more case clauses with an optional default clause, which is executed if none of the cases apply. A link is created to the switch expression, which is then linked to the expression of the first case. The control flow then branches on the switch clause to the enclosed statement if the condition evaluates to **`true`** and to the following switch clause if the condition was **`false`**. If the case clause has no enclosed statement, the control flow will instead move on to the following enclosed statement it finds regardless of the case clause expressions along the way. An example for this behavior would be the following snippet.

```
let value = 5;
switch (value) {
  case 5:
  case 6:
  case 7:
  case 8:
    console.log('value was between 5 and 8');
    break;
}
```

Special consideration must be taken if the case clause encountered is the `default` case, as the link will instead be non-conditional and alone. The `default` case does not implicitly terminate. The switch statement will continue executing after the traversal encountered with the same fallthrough behavior as standard case clauses. Fallthrough links will also have to be added as the control will move from one switch case to the block of the following case unless a break statement is inserted. Alternatively, the fallthrough behavior may also be interrupted by returning instead. If a switch statement defined no case clauses, the control instead moves to the statement succeeding the switch statement.

LabelStatements may act as a prefix for any statement except for function declaration statements which will cause a `SyntaxError` to be thrown instead[12, § 14.13.1]. In conjunction with the `break` and `continue` statements discussed previously, the label statement may be used as a replacement for the absent `goto` functionality. A single link is created from the label statement to the inner statement to calculate its impact on the control flow graph.

The **ThrowStatement** causes an exception, which is generated from an expression, to be thrown. A value generated by this expression does not have to be of a specific type. ECMAScript does not provide any kind of `Error` interface that has to be inherited. Calculating the control flow for this expression is challenging as the exception handler may cross function boundaries. If an exception handler within the function can be found, the control flow graph construction will create a link to it. Otherwise, the `throw` statement will be considered an exit to the current function.

A **TryStatement** consists of a block statement executed unconditionally and at least a `catch` clause or a `finally` clause. A `try` statement may also provide both. While the first block will be executed unconditionally, the block enclosed by the optional catch clause will only be executed if an exception was thrown during the execution of the first block.

The **DebuggerStatement** acts as a manual breakpoint that a programmer may insert into a piece of ECMAScript source code. Its impact on the control flow is unpredictable as a user with an attached debugger may alter the control flow unpredictable. It will simply create a link to the following statement in the control flow. There is no method for an ECMAScript program to attach a debugger to itself known to the author at the time of writing. Self-debugging would enable an application to abuse a breakpoint as, for example, a multi-threaded native application could. A program could then use it to alter the control flow arbitrarily. The single-threaded nature of ECMAScript halts the execution of everything if a debugger statement is encountered. This statement has no effect if a debugger is unavailable or not enabled[12, § 14.16.1].

5.5 Constant Evaluation

Another simple optimization that may be done statically is the so-called constant evaluation. It is related to constant folding from section 5.2 in so far as that expressions with a constant result are evaluated statically and the expression is then replaced with a constant value instead. While constant folding focused on expressions that contain unary or binary operators, constant evaluation focuses on the evaluation of functions instead.

Two types of functions will be looked at in this chapter. The first one dealing with call expressions, which reference a function within the standard library of ECMAScript and how they may be evaluated at constant time. The second category deals with functions defined by the obfuscated code and under which conditions they may be evaluated at a constant time.

5.5.1 Standard library functions

The main issue regarding built-in functions is, as alluded to in previous chapters, that they may be overwritten by the obfuscated application. This can be done without ever referencing the original function explicitly. To illustrate this we first need a way of expressing a string in a non-obvious way so that it may not be evaluated by the deobfuscation program. A simple way of doing this would be to hide the value behind a member expression. The access of the built-in function would then be accomplished by indirectly accessing it over the global object of the engine. In a web browser this would look like this:

```javascript
const a = {
  b: "console",
  c: "log"
};
window[a.b][a.c] // This evaluates to console.log
```

In order to hijack this built-in function a new function may simply be assigned to e.g. `window[a.b][a.c]`. At the time of writing, there is no method known to the author to restore hijacked functions to their original value without first storing a reference to the original function.

In order to evaluate calls to functions from the standard library, the user has to accept the risk that the functions may have been patched in the obfuscated code. The deobfuscation program does not attempt to trace or warn the user if patches are detected.

A further limitation on the standard library functions is that they have to be free from side effects. `console.log` has the side effect of writing to the console, for example. This effect can obviously not be replaced by a constant value.

The deobfuscation program contains a list of functions from the standard library that are known to be side effect free. Since the deobfuscation program is written in TypeScript, which is transpiled to ECMAScript during the build process, it may simply pass references to the functions within the standard library to the list.

The deobfuscation program then walks the abstract syntax tree of the input program and analyzes every call expression to determine both the callee and the arguments. All interesting call expressions for this step are using a member expression consisting of two identifiers as their callee. The arguments to the call expressions are evaluated on whether they are exclusively literal values. Therefore, this optimization should be executed after the constant folding and inlining passes to maximize the potential for constant evaluation.

An example for an easy to evaluate function would be `String.fromCharCode`, which takes one-or-more UTF16 code points as values of the type `number` and returns a value of the type `string` which contains the textual representation of the code points. `String.fromCharCode(100)`, for example, evaluates to `"d"` as 100 is the ASCII index character for the lowercase D character. This behavior has been observed in real world malware samples[16].

Another example, which has also been observed to occur in malware samples, is the use of `parseInt()` to obfuscate numeric values. The function requires one parameter, which is either already of the type `string` or will be converted to `string` using the abstract operation $ToString$. It also accepts an optional second parameter to specify the radix of the first parameter.

The issue with this trivial looking function is the optional nature of the radix parameter. If no value is passed, it will default to `10` in most cases. An exception for this behavior is, if the first parameter begins with `0x` or `0X` as the base will then default to 16. Previous versions of the ECMAScript specification also defined the special case of the string beginning with `0` as that would indicate the beginning of an octal number and as such the radix would be `8` instead of `10` for decimal. This would the somewhat paradoxical situation of `parseInt('08')` evaluating to `0` as `08` is not a valid octal number.

5.5.2 User defined functions

The constant evaluation of user-defined functions leverages the pre-existing scope analysis infrastructure introduced in section 5.3. Constant evaluation for user-defined functions also requires two traversals of the abstract syntax tree since a program may reference a function before it is declared. An example of this is contained within the following code snippet.

```
function a() {
    console.log("foo");
}

{
    a();
    function a() {
        console.log("bar");
    }
}
```

The function `a` is defined twice within the global scope and once in a local scope. It is then called in the local scope but before the local declaration of `a` was done. The result is that the local declaration has precedence over the global declaration and will be called in its stead.

This behavior changes if the local version of `a` is instead bound to a variable, as shown in the following code snippet.

```
function a() {
    console.log("foo");
}

{
    a();
    const a = () => {
        console.log("bar");
    }
}
```

In this version, a `ReferenceError` is thrown. This error happens because a program may not access variables declared as `let` or **`const`** before being declared in their scopes.

Eligible functions are functions that only consist of a single return statement. The argument for this return statement has to be free of side effects as a distinction may be made by creating a new instance of the `Error` built-in and reading the stack trace to determine the calling function. This decision was taken to avoid having to map identifiers from the old function into the new function and ensure that the resulting code's size is reduced.

As with the constant evaluation of built-in functions, all arguments passed at call sites must be literal values. The deobfuscation program may replace the occurrences of the parameter identifiers within the function to be evaluated as scope analysis has already been performed.

5.6 Member expression cleanup

Obfuscation software, like the one discussed in section 5.3, tends to obfuscate member expressions by abusing the fact that ECMAScript allows for member expressions that utilize identifiers to be replaced by equivalent member expressions that use string values. The simplest form of this replaces a member expression like `console.log` with its equivalent string version `console['log']`. As long as the parameter is kept as a string literal, the deobfuscation program can trivially return it to its original form.

The deobfuscation program traverses the abstract syntax tree and analyzes every member expression for two properties. At first, the program checks whether the object accessed is referenced using an identifier. If that is the case, the program then checks whether the property is supplied using a literal.

Special care has to be taken concerning the literal's type as primarily numeric types are prone to be used in conjunction with arrays.

Another potential issue is that object keys allow more characters than identifiers do. For example, assigning a value to an object using the key `"1foo"`only works if the string version of the member access is used. This restriction exists because identifiers may not start with numbers.

Keywords and other reserved names may be used as the production used by $MemberExpression$ is $IdentifierName$ and not $Identifier$. $IdentifierName$, unlike $Identifier$[12, § 13.1], does not exclude the $ReservedWord$ production[12, § 12.6].

The literal value in the abstract syntax tree is then replaced by a newly created identifier, which carries the value of the string literal as its name. In addition to that, the computed attribute of the member expression is also set to **`false`**. This transformation is done to change the representation of the expression from `console['log']` to `console.log`.

5.7 Well-known globals

Well-known globals are, within this thesis, the set of non-standard intrinsic objects within an ECMAScript engine. Malware samples, as seen in section 4.2.1, may use these objects for persistence or opaque predicates. Therefore, the deobfuscation program keeps a user-extensible list for these well-known global objects. Adding an entry to this list will cause it to be considered a variable declared in the global scope without a concrete declaration being added. This declaration makes the built-in objects eligible for inlining and a valid return statement for constant evaluation.

The currently implemented well-known globals are `ActiveXObject`, which may be used to interact with the Microsoft Windows operating system, and `WScript`, which similarly allows interaction but furthermore contains meta-information about the currently running script. Malware has been observed using these global objects to delete the currently running file after the persistence of another piece of malware has been ensured.

A user may add more values to this list to accommodate particular use-cases or new versions of the ECMAScript standard.

5.8 Identifier recovery

A programmer may decide to rename identifiers for a multitude of reasons.
One of them is that it serves as an easy, lossless compression. The lossless, of course, only applies to the semantics of the program. Information is still unrecoverably lost when this transformation is applied. Another reason is to harden the program against reverse engineering, as a reverse engineer may deduce a lot of the meaning of a piece of code from its identifiers.

This problem is an established problem field with multiple competing solutions as outlined in chapter 2. Therefore, this thesis elected to embed an existing solution rather than inventing a novel one. The chosen solution is Bavishi et al.'s Context2Name[4], which provides similar accuracy to the already introduced solutions JSNice and JSNaughty but improves on these solutions in terms of efficiency.

This chapter will briefly outline how Context2Name works without diving too deep into the details of its machine learning, as that would exceed the scope of this thesis.

Context2Name was trained on the same dataset already used with JSNice. It provides 150.000 ECMAScript files, which are, similarly to both JSNice and JSNaughty, processed with UglifyJS to simulate obfuscation within Context2Name. UglifyJS does not cover the range of transformations supported by the deobfuscation program described in this thesis. However, the impact of this should be minimal, as the identifier restoration routine is only invoked once all other deobfuscation passes have been executed.

Context2Name uses the usages of an identifier to recommend a better name. The deobfuscation program already collects usages of identifiers as a part of its scope analysis routine. This detail differentiates this implementation from the original one published with Context2Name as, e.g., identifiers used as properties in member accesses are filtered before reaching the renaming stage. These usages are then iterated, and the lexical tokens surrounding the identifiers are sent to Context2Name via an HTTP request. The necessary mapping from abstract syntax tree nodes to tokens is done by slightly modifying the parser to extend the built-in support for interacting with the lexical tokens. The deobfuscation program will skip the renaming without a running Context2Name server. The number of surrounding tokens also referred to as width, defaults to five. Therefore, the deobfuscation program will send the five preceding tokens and the five succeeding tokens of every identifier occurrence to the server. The identifier itself is not included in the tokens, but the transmission may contain other identifiers.

The Context2Name server returns an object in the JSON format upon success. This object contains multiple suggestions with an accompanying value between $0_{\mathbb{F}}$ and $1_{\mathbb{F}}$ indicating the accuracy of the proposed identifier. This accuracy may be utilized by the user of the deobfuscation program to filter suggestions by specifying a threshold.

An example of identifier recovery applied to the previously discussed malware sample from section 4.2.1 is shown in the following code snippet. This snippet shows the best-case scenario and does not represent the average success rate of this method.

```
var xhr = new ActiveXObject("MSXML2.XMLHTTP");
xhr.open("GET", src, 0);
try {
  xhr.send();
} catch {
  return false;
}
if (xhr.Status != 200) {
  return false;
}
```

The variable `xhr`, an acronym often used as a variable name for instances of the built-in `XMLHttpRequest` object, was renamed from `zs`. The deobfuscation program performed this renaming because Context2Name suggested it with an accuracy of 62% based upon five analyzed usages. Instead of `XMLHttpRequest`, the object is a proprietary way of performing HTTP requests within ECMAScript.

5.9 Static switch case evaluation

Another deobfuscation technique implemented in the deobfuscation program is the static evaluation of **`switch`** statements. The implementation is adapted from the implementation in SAFE-DEOBS[14].

It works on a subset of all possible **switch** statements, namely those with a literal value as the discriminant and literal values in all cases test conditions par for the default case.

Once an appropriate **switch** statement has been identified, the cases will be analyzed for matches. If no case matches the discriminant, the entire **switch** statement is removed from the abstract syntax tree. This removal is valid because no case will ever be executed.

However, if a case matches the discriminant, all preceding cases are eliminated from the switch statement as they are essentially dead code. Determining the end of live code is not as trivial due to the fallthrough semantics of the **switch** statement. If the end of a case is reached without an intermittent **break** statement, the execution will go into the succeeding case even though the test expression of the case might not evaluate to **true**. The deobfuscation program merges the statements of the case that matched with the statements of the succeeding cases. This merge allows the deobfuscation program to search the resulting array for the index of the first **break** statement. It then removes all statements from the array that succeed the **break** statement.

SAFE-DEOBS now replaces the **switch** statement with the statement array. This behavior does not always yield the correct code, however. An example of this can be seen in the following snippet.

```
switch (123) {
  case 1:
    break;
  case 123:
    if (Math.random() > 0.5) {
      break;
    }
  case 124:
    console.log("Hehe");
    break;
}
```

Following the algorithm from SAFE-DEOBS would yield the following code:

```
  {
    if (Math.random() > 0.5)
    {
      break;
    }
    console.log("Hehe");
  }
```

It becomes apparent that the naive approach misses a **break** statement that is only executed conditionally. Attempting to execute this code will yield a `SyntaxError`. In a different context, it might even silently change the semantics of the program. An example of this would be if the **switch** statement were enclosed within a loop.

The implementation in the deobfuscation program introduced in this thesis prevents this by adding a check on whether the statement list contains a **break** statement on any level. This way of addressing the issue prevents some edge cases where **switch** statements could still be statically evaluated. These cases could be addressed in a future version by utilizing the control flow graph.

6 Evaluation

The evaluation of this thesis will be two-fold. The first part will consist of a case study that focuses on the same piece of obfuscated code that was used as the case study for SAFE-DEOBS, which was introduced in chapter 2.4.

The second part will present a study that attempts to capture programmers' opinions on the usefulness of the applied deobfuscations. The study will include a few of the deobfuscation passes parameters previously mentioned. A significant reason for the study is that formal methods may not measure some aspects of code. For example, the renamed identifiers can only be judged on their accuracy by a human or another machine-learning-based solution.

6.1 Case study

The code sample used for the case study is the same code used in the evaluation of SAFE-DEOBS. Using the same code allows a direct comparison of code metrics between the two deobfuscation solutions. Some of the obfuscation techniques present have already been discussed in previous chapters. The defining characteristics of this sample are the frequent use of statically evaluatable `switch` statements and extensive amounts of dead code.

SAFE-DEOBS replaces identifiers with animal names, annotating the original name as a comment on the declaration. The names are assigned deterministically, so sequential deobfuscation runs yield the same identifiers. This approach might remove meaningful identifiers in some code samples, but the current sample is unaffected.

SAFE-DEOBS reduces the source lines of code from 474 to twelve, which constitutes a reduction of around 97.5%. The deobfuscation program introduced in this thesis reduces the 474 lines of code to fifteen lines of code which constitutes a total reduction in lines of code of 96.8%. The lines of code were measured using the UNIX tool `wc`.

All functions were inlined in both programs.

Reading the output from either tool lets a reverse engineer deduce that the sample is designed for the JScript engine as it references the non-standard built-in `WScript`.

The deobfuscation program introduced in this thesis uses Context2Name to recover identifiers, which allows the program to rename the variable `uvacdykadq` to `isNode`. The value assigned to this variable is the result of the expression `typeof window == "undefined"`. `window` is the global object used by browsers. If it is of the type `undefined`, the execution environment is likely not a browser. While a more appropriate name would have been `isJScript`, the intent of the variable remains clear.

Both tools still leave superfluous code within the program. The code that is left differs between programs, however. The deobfuscation program introduced within this thesis leaves a lot of dead stores. An example of this can be seen in the following snippet.

```
var expected = "996719";
expected = expected + "ofvy";
```

Since the variable is used more than once, the inlining pass will not inline `"996719"`. Utilizing the control flow graph more extensively could allow removing these dead stores. SAFE-DEOBS eliminates them.

SAFE-DEOBS explicitly hoists `var` declarations by moving all declarations to the top of the function scope. The initialization of these variables is not moved to preserve the semantics of the code. This explicit hoisting leads to an increase in the lines of code as every declaration that was previously done in a single line now requires two lines of code.

Another moment where SAFE-DEOBS gains lines of code are the elimination of superfluous block statements. The deobfuscation program introduced in this thesis flattens block statements contained within other block statements if they contain no block-scoped declarations.

Combining the two solutions does not improve the results in this case because the deobfuscation program introduced in this thesis can not handle the split declaration and initialization that SAFE-DEOBS introduces. SAFE-DEOBS, on the other hand, does not support non-standard built-ins such as `WScript` and aborts the deobfuscation process.

Both solutions allow a reverse engineer to extract the main properties of the sample, however. It checks whether it runs within a browser and then attempts to launch the Windows command processor, launching the Windows PowerShell to execute a payload.

The escomplex tool used in evaluating SAFE-DEOBS has not been maintained and could therefore not be executed and thus used as part of this thesis.

6.2 Survey

The study accompanying this thesis is meant to measure the increase in readability and legibility created by the deobfuscation program. The separation between legibility and readability is according to the definition in Oliveira et al.'s literature analysis in "Evaluating Code Readability and Legibility: An Examination of Human-centric Studies"[27].

According to their definition, legibility defines how easily understood the code is based on its layout. An example of code layout would be code style guidelines such as limiting the length of a single line of code. This criterion may be complied with automatically by the use of a linter.

They define readability, on the other hand, as "the structural and semantic characteristics of the source code of a program that affect the ability of developers to understand it while reading the code"[27,

p. 2].

The deobfuscation techniques implemented by the deobfuscation program address legibility and readability, and their effects intertwine. Legibility is also restored by the routine that generates source code from the abstract syntax tree. This routine eliminated virtually all formatting-based obfuscation techniques.

The study consists of subjects being given code samples of both obfuscated, unobfuscated, and deobfuscated source code. The task is then comprised of extracting properties from the given source code sample within a time limit of 15 minutes. The accuracy of the extracted properties is measured using a predefined schema. Extracting all properties defined by the schema will not yield the full amount of points for the obfuscated and deobfuscated samples. As both samples are well-known algorithms, it is necessary to name the algorithm to receive full marks. No negative scoring is applied as that would require accounting for every possible mistake and evaluating its impact on the understanding of the user.

The survey was conducted with the participants remaining anonymous to avoid the implications of storing personally-identifying information.

Measuring readability is challenging, though, as it is a very abstract concept. In order to design an accurate study, the 2021 article "Considerations and Pitfalls in Controlled Experiments on Code Comprehension" was taken into account. In this article, Feitelson performed a comprehensive analysis to provide considerations and problems that may occur during a study on code comprehension. It concludes with a helpful checklist of things to keep in mind. Some of the items on the checklist do not apply to this thesis in particular. This deviation follows from the objective of this thesis which includes measuring the readability of intentionally obfuscated code.

The checklist is divided into four groups by their relation to the elements of such a study. As such, there is a group of items related to the code itself, one related to the tasks, one related to the measurement of results, and the final group deals with items related to the study subjects.

The following paragraphs describe where this thesis has to deviate from the recommendations of Feitelson.

Feitelson warns about including misleading code within the first group of items. Examples of this described within the article, such as using non-decimal representations for numbers, are actively employed by many obfuscation programs. Another example of this introduced by Feitelson includes misleading variable names. This checklist item is challenging for this thesis as it includes obfuscation programs that intentionally obfuscate variable names. The names generated by these obfuscation programs can include random characters and commonly used variable names from different projects. As described in the case study, the counter-measures employed by this thesis are not flawless either, as Context2Name is not necessarily trained to expect malware in particular.

The same restrictions apply to the suggestion to use variable names such as `a`, `b`, or `c`. A core part of this thesis is the restoration of identifiers, and as such, it would be counter-productive to obfuscate them again.

Another point is the inclusion of dead code. Especially the obfuscated code shown to users will include dead code. Feitelson warns that the inclusion of dead code may give the subjects of the study hints to aid in comprehending the source code. However, this warning can be ignored as the dead code introduced by obfuscation programs is intended to mislead instead of aid.

Unlike the previous group, not all items apply to all studies in the group of items related to the tasks given to the subjects. Feitelson suggests that an interpretation task should be chosen to gauge the subjects' semantics comprehension. This requirement has been fulfilled by letting the user extract properties of a given code sample.

The three samples chosen for this study were a piece of code that issues an HTTP request to fetch a list of tasks and count the total and completed tasks for a given user. The first sample is unobfuscated and is intended to gauge the subjects' general ability to understand a given piece of code. Unlike the remaining two samples, extracting all properties for a subject to receive the full amount of points suffices. In order to receive points for this exercise, the subject has to identify the following points:

- Identify that an HTTP request is being issued.
- Identify what type of data is being returned
- Identify what is being calculated from the context
- Identify that it counts both the total as well as the completed

The sample is also considerably more complex than the remaining ones.

The second sample prompts for a number and outputs the Fibonacci sequence until the index equals the given number. This sample is obfuscated using the previously introduced javascript-obfuscator. The obfuscation settings used in javascript-obfuscator for this and the deobfuscated sample included the following:

- The renaming of local and global variables
- Compact & Simplify
- Numbers to expressions
- Transform object keys

While these settings only present a small sample of the available ones, they already significantly impair the readability of the code while not increasing its size as other obfuscation techniques would.

The following is the code for the Fibonacci series after the obfuscation has been applied.

```javascript
const _0x34dcdc=parseInt(prompt('Enter\x20the\x20number\x20of\x20terms
    :\x20'));let _0x4b671c=-0xf85+-0x287*0x3+0x171a,_0x4e8322=0x1f54+-0
    x16fc+-0x857,_0x1995d0;for(let _0x2d63ca=0x87e+-0x2*0xa9+-0x72b;
    _0x2d63ca<=_0x34dcdc;_0x2d63ca++){console['log'](_0x4b671c),
    _0x1995d0=_0x4b671c+_0x4e8322,_0x4b671c=_0x4e8322,_0x4e8322=
    _0x1995d0;}
```

To receive the full points, correctly identifying the sequence as the Fibonacci series would be sufficient. Partial points are awarded for:

- Identifying that the user is providing a number
- The loop

- That a calculation is being performed involving the input value
- That results are printed on the console

The final sample is the deobfuscated sample that calculates the digit sum of a number. The deobfuscated code can be seen in the following code sample.

```javascript
function _0x3fc2aa(data) {
  if (typeof data !== 'string') {
    data = data.toString();
  }
  if (data.length < 2) {
    return parseInt(data);
  }
  return _0x3fc2aa(data.split('').reduce((_0x5f4c55, fraction) =>
     _0x5f4c55 += parseInt(fraction), 0));
}
```

Full points are awarded if the subject successfully recognizes that the digit sum is being calculated. Partial points are awarded for:

- Recognizing that the parameter is supposed to be a `string` value
- The trivial case of the input being smaller than two characters
- That the input is being split so that it becomes a list of characters
- Recognizing the recursion

6.2.1 Results

The survey received a total of 51 anonymous replies. The average score in the sample meant to gauge the subjects' proficiency in reading code was 3.29 points, with a median of four out of four.

The average score for the obfuscated sample was 2.61, with a median score of three out of five. 8 out of 51 participants (15.69%) identified the code sample as an implementation of the Fibonacci series. The exact score distribution can be seen in figure 6.1.

The average score for the deobfuscated sample was 3.08, with a median score of four out of five. 25 out of the 51 participants (49.02%) were able to identify the sample as an implementation of calculating the digit sum. The exact score distribution can be seen in figure 6.2.

These values suggest an increase in the average readability of the code by 17.65%. The number of participants who could fully identify the code sample increased by 212.5%. This result might be skewed, as the timer of 15 minutes was for the entire survey and not per task due to technical limitations. This limitation could skew the result and will have to be examined in a future study.

The result of the first sample meant to gauge the subjects' ability to read code was disregarded as correlating it with the scores of the deobfuscated sample led to no clear indication in either direction. A graph of this can be seen in figure 6.3. This result might stem from some subjects' unwillingness to list all aspects found in the first sample in the required level of detail.

The ability to decipher the obfuscated code, however, appears to correlate strongly with the thorough-

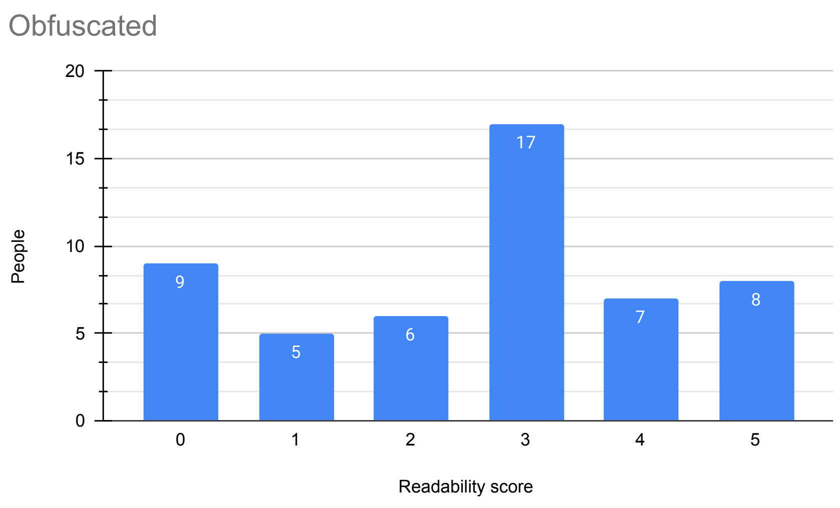

Figure 6.1: Distribution of scores in the obfuscated sample

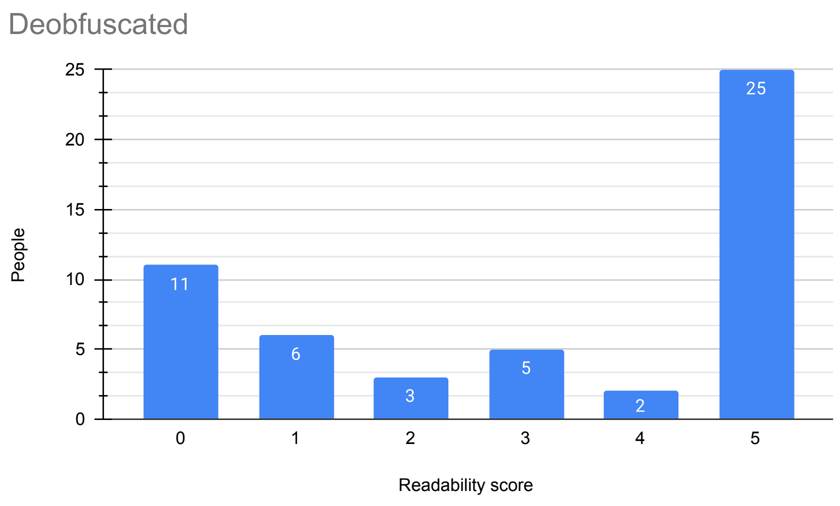

Figure 6.2: Distribution of scores in the deobfuscated sample

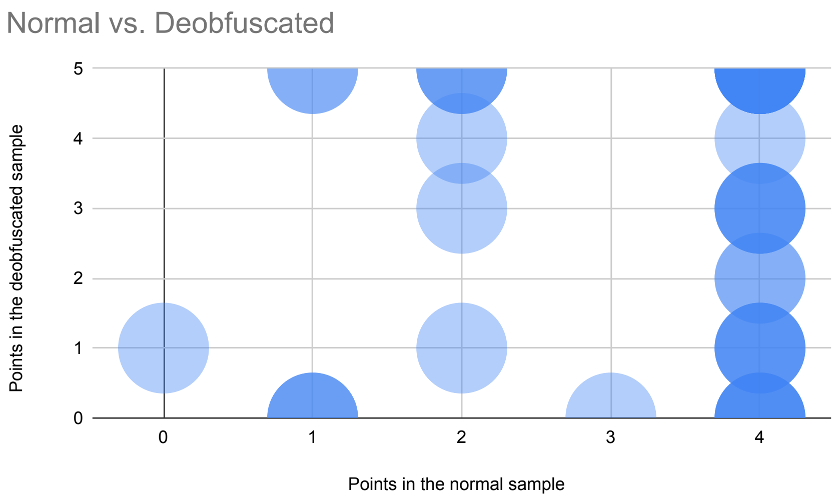

Figure 6.3: Scatter plot of results in normal and deobfuscated sample

ness of the control sample analysis. A graph of this can be seen in figure 6.4.

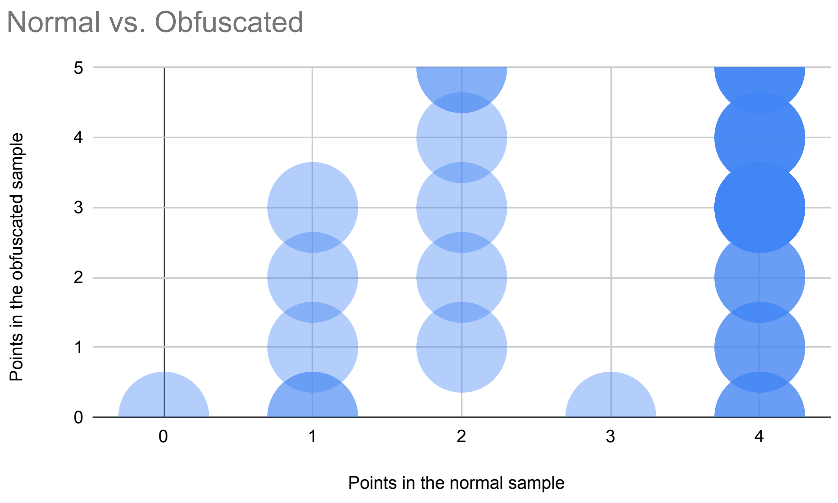

Figure 6.4: Scatter plot of results in normal and obfuscated sample

Two out of the 51 participants were misled by the renamed second parameter of the reduce function `fraction`. This misdirection presumably led them to misidentify the function. Statistically, this misidentification is negligible, especially considering the number of participants who correctly identified the deobfuscated sample despite the renamed variable. This problem may also be addressed by modifying the threshold value to filter the suggestions from Context2Name more aggressively.

An interesting observation in the data is that some subjects misinterpreted `data.length < 2` as `data.length <=2` in the deobfuscated code sample. This result raises the question of whether a more explicit comparison like `data.length <= 1` should have been used instead.

One of the subjects followed up on the survey, criticizing that they were actively looking for purposefully placed misdirections in the deobfuscated code. The survey could have been more clear about its intent here.

Another criticism arose from the technical limitation on the amount of time given for the exercise. This global limit instead of a limit per exercise could skew the results either way. Some subjects may feel pressure in the obfuscated sample to move on rather quickly with the timer continuously ticking down, while others may only reach the deobfuscated sample with little time to spare. This problem should be addressed in future explorations of this work.

The recursive property of the last code example was often missed by subjects, indicating that keeping the hexadecimal identifiers when Context2Name did not provide a better name was not optimal.

The higher-order function reduce was also sometimes incorrectly identified by subjects, presumably due to a lack of knowledge in functional programming.

7 Future work

This chapter will focus on the opportunities for future work derived from this thesis.

More deobfuscation techniques besides dead code elimination may be derived from a control flow graph. A future expansion of the deobfuscation program described in this thesis may utilize it to reason more in-depth about variable inlining. The closure compiler, which this thesis discussed at various points, already implements a more aggressive, control flow-based inlining algorithm that may be adapted to fit this thesis.

This improved inlining algorithm may also attempt to tackle the issues surrounding the inlining of values within an array.

Another use for the control flow graph would be to attempt to remove control-flow flattening, which was deemed to be out of scope for this thesis.

Another opportunity for future work would be to provide a graphical user interface similar to a visual source code merge tool. This addition would allow the users of the deobfuscation tool to understand the process more in-depth. It would also allow the users to interact with the process and fine-tune the obfuscation passes. This user interface could then, for example, be used by a user to vet the variable name suggestions from Context2Name on an individual basis.

The use of comments could also implement this fine-tuning. It is a well-established way of excluding certain parts of the code from static source code analysis tools.

Another angle for future work would be to explore whether a deobfuscation program may utilize machine learning for further deobfuscation. JSNice already established that type annotations may be generated using machine learning with acceptable accuracy. A future version of the deobfuscation program could use these inferred type annotations to bridge the gap currently filled by probabilistic methods in constant folding. This approach becomes more interesting with the proposed adoption of type annotations into the ECMAScript standard[34].

Replacing or combining Context2Name with JSNaughty would also be an exciting approach to further developing the deobfuscation program described in this thesis.

Another path for improving on this thesis would be to add a few heuristics to the constant folder to prevent the folding of, e.g., well-known bitflags as that could potentially be harmful to the readability. This addition would require research into detecting those bitflags first, however.

8 Conclusion

This thesis provides a comprehensive overview of the state of the art of ECMAScript obfuscation and deobfuscation. This obfuscation overview is complemented by an analysis of publicly available obfuscation software.

In addition, the extensive analysis of constant folding within ECMAScript provides new insights into a probabilistic approach to deobfuscation.

Meanwhile, the overview in deobfuscation is accompanied by appropriate countermeasures and a fully working implementation.

The deobfuscation software introduced as part of this thesis was evaluated with the result of improving the average understanding meaningfully while increasing the complete understanding of the tested code samples significantly.

The direct comparison with the state-of-the-art tool SAFE-DEOBS yielded a clear direction where the deobfuscation program introduced in this thesis can be taken. While it has already performed comparably with SAFE-DEOBS, it can still be improved by utilizing the control flow graph more extensively. In addition to its performance, it also fixed a flaw found in the logic of SAFE-DEOBS.

The question introduced at the beginning of the thesis was whether the deobfuscation program would be able to improve the readability of ECMAScript code can confidently be answered with yes. The case study, in particular, provided evidence that the renaming process, which is powered by machine learning, can help in understanding foreign or obfuscated code. However, this success must be combined with the complaints from the two subjects within the study. The potential for improvement depends massively on the code sample at hand and the threshold chosen for Context2Name. The survey results indicate success, but they should be validated in a future iteration with the already levied criticisms. In addition to that, it could be interesting to perform an interdisciplinary evaluation with experts in cognition.

While this question may have been answered, a plethora of new and exciting questions have come up as part of this thesis. Some of which are discussed in section 7.

While the deobfuscation program introduced in this thesis can deobfuscate many different obfuscation techniques, its limitations have been clearly defined. Especially boundaries that make static analysis unfeasible, like `eval()`, have been highlighted and elaborated.

The generalizability of this approach is limited in its current form as most of the rules apply only to ECMAScript in particular. However, the concepts introduced and applied in this thesis are tried and tested for the most part and can be transferred to other programming languages. Especially the functionality of Context2Name could be easily transferred given enough sample data in the target language.

Bibliography

[1] Mar. 2022. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let#temporal_dead_zone_tdz.

[2] Stefano Crespi Reghizzi (auth.) *Formal Languages and Compilation*. 1st. Texts in Computer Science. Springer London, 2009. ISBN: 9781848820494; 1848820496; 9781848820500; 184882050X.

[3] Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. *Compilers: principles, techniques, & tools*. 2nd ed. Pearson/Addison Wesley, 2007. ISBN: 9780321486813.

[4] Rohan Bavishi, Michael Pradel, and Koushik Sen. *Context2Name: A Deep Learning-Based Approach to Infer Natural Variable Names from Usage Contexts*. 2018. DOI: 10.48550/ARXIV.1809.05193. URL: https://arxiv.org/abs/1809.05193.

[5] Mihai Bazon. *Uglify-JS*. URL: https://www.npmjs.com/package/uglify-js.

[6] Eli Benderskys. *How clang handles the type / variable name ambiguity of C/C++*. July 2012. URL: https://eli.thegreenplace.net/2012/07/05/how-clang-handles-the-type-variable-name-ambiguity-of-cc.

[7] P.W.D. Charles. *The ESTree Spec*. https://github.com/estree/estree. 2022.

[8] N. Chomsky. "Three models for the description of language." In: *IRE Transactions on Information Theory* 2.3 (1956), pp. 113–124. DOI: 10.1109/TIT.1956.1056813.

[9] Noam Chomsky. "On Certain Formal Properties of Grammars." In: *Information and Control* 2 (June 1959), pp. 137–167. DOI: 10.1016/S0019-9958(59)90362-6.

[10] Christian Collberg, Clark Thomborson, and Douglas Low. "Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs." In: *Conference Record of the Annual ACM Symposium on Principles of Programming Languages* 184–196 (Nov. 1997). DOI: 10.1145/268946.268962.

[11] *Curly*. URL: https://eslint.org/docs/rules/curly.

[12] ECMA International. "Standard ECMA-262 - ECMAScript Language Specification." In: 12th. June 2021. URL: https://www.ecma-international.org/wp-content/uploads/ECMA-262_12th_edition_june_2021.pdf.

[13] Lars Marius Garshol. *BNF and EBNF: What are they and how do they work?* Aug. 2008. URL: https://www.garshol.priv.no/download/text/bnf.html.

[14] Adrian Herrera. "Optimizing Away JavaScript Obfuscation." In: *CoRR* abs/2009.09170 (2020). arXiv: 2009.09170. URL: https://arxiv.org/abs/2009.09170.

[15] *History*. June 2021. URL: https://www.ecma-international.org/about-ecma/history/.

[16] Hynek Petrak. *Javascript Malware Collection*. https://github.com/HynekPetrak/javascript-malware-collection. 2019.

[17] "IEEE Standard for Floating-Point Arithmetic." In: *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (2019), pp. 1–84. DOI: 10.1109/IEEESTD.2019.8766229.

[18] Google Inc. *Closure Compiler Compilation Levels*. 2019. URL: https://developers.google.com/closure/compiler/docs/compilation_levels (visited on 04/19/2022).

[19] Google Inc. *Closure Tools*. URL: https://developers.google.com/closure (visited on 04/19/2022).

[20] ISO. *ISO/IEC 14882:2020 Information technology — Programming languages — C++*. Geneva, Switzerland: International Organization for Standardization, Dec. 2020, p. 1853. URL: https://www.iso.org/standard/79358.html.

[21] Rajeev Motwani John E. Hopcroft and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation*. 3rd ed. Pearson/AddisonWesley, 2006. ISBN: 9780321455369.

[22] DONALD KNUTH and LUIS PARDO. "The Early Development of Programming Languages." In: Dec. 1980. DOI: 10.1016/B978-0-12-491650-0.50019-8.

[23] Brian Leroux. *WTFJS*. YouTube. 2012. URL: https://www.youtube.com/watch?v=et8xNAc2ic8.

[24] Peter Linz. *An Introduction to Formal Languages and Automata*. 3rd ed. Jones and Bartlett, 2001. ISBN: 9780763714222.

[25] Fabrício S. Matté. *Temporal dead zone (TDZ) demystified*. Jan. 2015. URL: http://jsrocks.org/2015/01/temporal-dead-zone-tdz-demystified.

[26] Steven Muchnick. *Advanced Compiler Design Implementation*. Jan. 1997. ISBN: 9781558603202.

[27] Delano Oliveira et al. "Evaluating Code Readability and Legibility: An Examination of Human-centric Studies." In: *CoRR* abs/2110.00785 (2021). arXiv: 2110.00785. URL: https://arxiv.org/abs/2110.00785.

[28] Axel Rauschmayer. "Speaking JavaScript." In: London: O'Reilly, 2014.

[29] Veselin Raychev, Martin Vechev, and Andreas Krause. "Predicting Program Properties from "Big Code"." In: *ACM SIGPLAN Notices* 50 (Jan. 2015), pp. 111–124. DOI: 10.1145/2775051.2677009.

[30] Emma Söderberg et al. "Declarative Intraprocedural Flow Analysis of Java Source Code." In: *Electronic Notes in Theoretical Computer Science* 238 (Oct. 2009), pp. 155–171. DOI: 10.1016/j.entcs.2009.09.046.

[31] *Speed Comparison adapted from Esprima's Speed Comparison*. URL: https://meriyah.github.io/meriyah/performance/.

[32] *Speed Comparison keeps everything in perspective*. URL: https://esprima.org/test/compare.html.

[33] Abhinav Suri. *The why behind the wat- an explanation of JavaScript's type system*. Jan. 2018. URL: https://abhinavsuri.com/blog/2018/watjs/.

[34] Gil Tayar et al. *ECMAScript proposal: Type Annotations*. https://github.com/tc39/proposal-type-annotations. 2022.

[35] Timofey Kachalov. *JavaScript obfuscator*. https://github.com/javascript-obfuscator/javascript-obfuscator. 2022.

[36] *Typeof - JavaScript: MDN*. Feb. 2022. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/typeof#typeof_null.

[37] *U.S. Trademark Serial No. 75026640*. Dec. 1995. URL: https://tsdr.uspto.gov/#caseNumber=75026640&caseType=SERIAL_NO&searchType=statusSearch.

[38] Bogdan Vasilescu, Casey Casalnuovo, and Premkumar Devanbu. "Recovering Clear, Natural Identifiers from Obfuscated JS Names." In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2017. Paderborn, Germany: Association for Computing Machinery, 2017, pp. 683–693. ISBN: 9781450351058. DOI: 10.1145/3106237.3106289. URL: https://doi.org/10.1145/3106237.3106289.

[39] William Vincent. Aug. 2017. URL: https://wsvincent.com/javascript-temporal-dead-zone/.

[40] Chenxi Wang et al. "Software Tamper Resistance: Obstructing Static Analysis of Programs." In: (June 2000).

[41] K Zuse. "Über den Plankalkül." In: *it - Information Technology* 1 (Dec. 1959). DOI: 10.1524/itit.1959.1.14.68.