# ZADÁNÍ DIPLOMOVÉ PRÁCE

| | |
|---|---|
| **Název:** | Minifikace a obfuskace JavaScriptu |
| **Student:** | Bc. Jakub Holub |
| **Vedoucí:** | Ing. Radomír Polách |
| **Studijní program:** | Informatika |
| **Studijní obor:** | Systémové programování |
| **Katedra:** | Katedra teoretické informatiky |
| **Platnost zadání:** | Do konce letního semestru 2019/20 |

## Pokyny pro vypracování

1) Nastudujte problematiku minifikace a obfuskace jazyka JavaScript.
2) Seznamte se s knihovnami espree a escodegen pro parsování a generování jazyka JavaScript na platformě NodeJS.
3) Navrhněte, analyzujte a implementujte efektivní postup minifikace a obfuskace pro jazyk Javascript za použití zmíněných knihoven, maximálně se zaměřte na ztížení deminifikace a deobfuskace.
4) Při návrhu, analýze a implementaci berte v úvahu předchozí práci na stejné téma.
5) Implementaci testujte na programech dodaných vedoucím práce.

## Seznam odborné literatury

Dodá vedoucí práce.

doc. Ing. Jan Janoušek, Ph.D.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 15. února 2019

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Diplomová práce

# Minifikace a obfuskace JavaScriptu

***Bc. Jakub Holub***

Katedra teoretické informatiky
Vedoucí práce: Ing. Radomír Polách

9. května 2019

# Poděkování

Děkuji vedoucímu práce Ing. Radomíru Poláchovi za vedení práce.

# Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo"), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 9. května 2019 . . . . . . . . . . . . . . . . . . . . . .

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2019 Jakub Holub. Všechna práva vyhrazena.
*Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí a nad rámec oprávnění uvedených v Prohlášení na předchozí straně, je nezbytný souhlas autora.*

### Odkaz na tuto práci

Holub, Jakub. *Minifikace a obfuskace JavaScriptu.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2019.

# Abstrakt

Tato práce se zabývá minifikací a obfuskací jazyka JavaScript. Je provedena analýza stávajících prací a volně dostupných knihoven zabývajících se obfuskací tohoto jazyka. Výstupem práce je modulární obfuskátor postavený na platformě NodeJS. V práci jsou navrženy a popsány minifikační a obfuskační transformace, které byly do obfuskátoru implementovány. Transformace byly v práci voleny s cílem ztížit deminifikaci a deobfuskaci. Funkčnost obfuskátoru je ověřena testováním na běžně používaných knihovnách.

**Klíčová slova** Obfuskace, Minifikace, Statická analýza, JavaScript

# Abstract

This master's thesis deals with the minification and obfuscation of the JavaScript language. The analysis of existing theses and publicly available libraries dealing with obfuscation of this language is performed. Output of this master's thesis is a modular obfuscator built on the NodeJS platform. In this master's thesis are designed and described the minification and obfuscation

transformations which were implemented into the obfuscator. The transformations in this master's thesis were chosen with the goal of making the deminification and deobfuscation more difficult. The functionality of the obfuscator is verified by testing on publicly available libraries.

**Keywords** Obfuscation, Minification, Static analysis, JavaScript

# Obsah

**Úvod** **1**

**1 Cíl práce** **3**

**2 Základní pojmy** **5**
2.1 Minifikace . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
2.2 Obfuskace . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
2.3 Deobfuskace . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
2.4 Abstraktní syntaktický strom . . . . . . . . . . . . . . . . . . . 8
2.5 ECMAScript . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
2.6 JavaScript . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10

**3 Existující práce a knihovny** **13**
3.1 Bakalářská práce Samuela Hanáka . . . . . . . . . . . . . . . . . 13
3.2 Volně dostupné knihovny . . . . . . . . . . . . . . . . . . . . . . 14

**4 Analýza a návrh** **17**
4.1 Programovací jazyk . . . . . . . . . . . . . . . . . . . . . . . . . 17
4.2 Návrh struktury obfuskátoru . . . . . . . . . . . . . . . . . . . . 18
4.3 Zaměření transformací . . . . . . . . . . . . . . . . . . . . . . . 19
4.4 Omezení pro vstupní kód . . . . . . . . . . . . . . . . . . . . . . 19
4.5 Použité nástroje knihovny . . . . . . . . . . . . . . . . . . . . . 21

**5 Realizace** **25**
5.1 Odstranění bílých znaků a komentářů . . . . . . . . . . . . . . . 25
5.2 Odstranění redundantních znaků v kódu . . . . . . . . . . . . . . 26
5.3 Optimalizace zdrojového kódu . . . . . . . . . . . . . . . . . . . 26
5.4 Přejmenování identifikátorů . . . . . . . . . . . . . . . . . . . . . 27
5.5 Zastavování debuggeru ve smyčce . . . . . . . . . . . . . . . . . . 29
5.6 Náhodné rozmístění deklarací proměnných . . . . . . . . . . . . . 30

5.7 Převedení vlastností objektů na variantu využívající literál . . . 32
5.8 Obfuskace čísel . . . . . . . . . . . . . . . . . . . . . . . . . . 32
5.9 Náhodné přeuspořádání parametrů funkcí . . . . . . . . . . . . 34
5.10 Přesunutí operátorů do funkcí . . . . . . . . . . . . . . . . . . . 35
5.11 Předefinování konzole . . . . . . . . . . . . . . . . . . . . . . . 37
5.12 Obfuskace řetězců . . . . . . . . . . . . . . . . . . . . . . . . . 38
5.13 Obfuskace vybraných výrazů . . . . . . . . . . . . . . . . . . . 40
5.14 Náhodné přeuspořádání definic funkcí . . . . . . . . . . . . . . 41
5.15 Slučování funkcí . . . . . . . . . . . . . . . . . . . . . . . . . . 41
5.16 Přesunutí části kódu do funkce eval . . . . . . . . . . . . . . . . 43
5.17 Zašifrování části kódu a vynucení integrity kódu . . . . . . . . 44
5.18 Kombinace a posloupnost transformací . . . . . . . . . . . . . . 46

**6 Testování** **49**
6.1 Empirické testování transformací . . . . . . . . . . . . . . . . . 49
6.2 Sada testovacích skriptů přiložená u obfuskátoru . . . . . . . . 49
6.3 Testování obfuskátoru na veřejně dostupných knihovnách . . . 50

**7 Navazání na praci** **53**

**Závěr** **55**

**Literatura** **57**

**A Seznam použitých zkratek** **61**

**B Obsah přiloženého CD** **63**

# Seznam obrázků

21 Ukázka abstraktního syntaktického stromu pro kód 2.1 . . . . . . . . 9

# Seznam tabulek

51 Příklad čísel převedených do výrazů . . . . . . . . . . . . . . . . . . . 33
52 Rozdělení pravděpodobnosti převodu na jednotlivé číselné soustavu 34
53 Mapovací tabulka pro výrazy . . . . . . . . . . . . . . . . . . . . . . . 40

61 Porovnání doby běhu výpočtu původní a obfuskované knihovny . . 51

# Úvod

JavaScript je v současné době nejpoužívanější programovací jazyk vůbec. Jeden z možných způsobů jeho využití je na webových stránkách, kde například dodává stránce interaktivitu nebo slouží pro sledování uživatele. V poslední době se ale možné použití jazyka JavaScript výrazně rozšířilo. Je možné ho použít například na serveru nebo pro vývoj desktopových či mobilních aplikací.

Skripty využité na zobrazené webové stránce musejí být stáhnuty ze serveru a jejich velikost ovlivňuje dobu přenosu. Cílem procesu minifikace je snížit velikost daného souboru. Dále platí, že skripty napsané v jazyce JavaScript jsou ve webovém prohlížeči zobrazitelné uživatelem. To může přinést bezpečnostní rizika, pokud se v těchto skriptech vyskytují informace, které by uživatel neměl znát. Cílem procesu obfuskace je transformovat kód do takové podoby, kdy uživatelem nebude srozumitelný.

Tato práce se zabývá návrhem a implementací efektivního postupu minifikace a obfuskace, který by pro takové skripty mohl být použit.

Kapitola 1

# Cíl práce

Cíl práce se dle zadání skládá z několika bodů. V práci proběhne popsání problematiky minifikace a obfuskace jazyka JavaScript. Dále pak proběhně seznámení se specifickými vlastnostmi jazyka JavaScript, které ovlivní navrhnuté transformace pro minifikaci a obfuskaci. Následně budou analyzovány volně dostupné knihovny pro obfuskaci a také předchozí bakalářská práce na toto téma. Na základě předchozích bodů vznikne obfuskátor, který bude efektivně minifikovat a obfuskovat kód jazyka JavaScript. Při tom bude kladen důraz na ztížení deminifikace a deobfuskace. Návrh struktury obfuskátoru a jeho jednotlivé transformace budou popsány v této práci. Obfuskátor bude v rámci práce také otestován na běžně používaných knihovnách a porovnán s bakalářskou prací na stejné téma.

Kapitola 2

# Základní pojmy

## 2.1 Minifikace

Minifikace je podle [1] transformace zdrojového kódu, jejímž cílem je snížit délku výsledného kódu při zachování funkčnosti vstupního kódu. Taková transformace je užitečná zejména pro zdrojové kódy, které se přenášejí přes internet, a snížením jejich délky tak může být urychlen jejich přenos. Dochází k úpravám kódu, které sice mohou zvyšovat přehlednost a srozumitelnost zdrojového kódu, avšak nejsou nezbytné pro samotný běh programu. S minifikovanou verzí kódu se již zpravidla dále nemanipuluje, a proto se tyto úpravy mohou provést bez újmy na následné použitelnosti kódu, neboť možné aktualizace se provedou v původním zdrojovém kódu, na který se opět aplikuje minifikace.

Při procesu minifikace dochází k odstranění znaků či částí kódu, které neplní v běhu programu žádnou funkci. Jedná se především o tzv. bílé znaky, mezi které patří například mezera, tabulátor nebo odřádkování. Dále mohou být odstraněny komentáře, které ale někdy mohou obsahovat důležitou informaci o verzi zdrojového kódu. V závislosti na vlastnostech programovacího jazyka mohou být odstraněny například redundantní závorky nebo oddělovače příkazů. Pokročilejší techniky mohou identifikovat části kódu, které nejsou dosažitelné nebo neplní žádnou funkci a odstranit je ve výsledném kódu. Příkladem může být deklarace proměnné, která není dále v kódu nikde použita.

Další možností, jak snížit délku výsledného kódu, je přejmenování existujících identifikátorů v kódu na nejkratší možnou alternativu. Přejmenovat je možné například proměnné, funkce nebo třídy. Pro správné fungování programu je nutné přejmenovat i veškeré jejich reference.

## 2.2 Obfuskace

Obfuskace je podle [2] transformace zdrojového kódu, která zastiňuje či utajuje informace obsažené v kódu. Toho je dosaženo tím, že zdrojový kód se změní na takovou podobu, ve které se pro člověka stává nesrozumitelným. Zachovává přitom ale funkčnost původního programu. Cílém obfuskace je tedy zabránit porozumnění kódu, které by mohlo vést k bezpečnostnímu riziku či odhalení důvěrných informací. Dalším cílem může být zabránění modifikaci zdrojového kódu, které by porušilo integritu kódu.

Není ale garantováno, že provedené transformace jsou nereverzibilní. Při dostatku času a úsilí lze reverzním inženýrstvím zpětně pochopit funkčnost libovolného kódu, i kdyby k tomu mělo dojít až na té nejnižší úrovni. Obfuskace tedy není vlastnost, ale je to míra. Měla by změnit kód do takové podoby, aby byl reverzní inženýr odrazen od snahy získat původní kód.

Obfuskační transformace může přistoupit k úpravě kódu z různých pohledů. Může transformovat data obsažená v kódu, tedy například řetězce, konstanty nebo komentáře. Jiné transformace mohou obfuskovat samotný kód, tedy například posloupnost jednotlivých příkazů, názvy identifikátorů nebo šifrování příkazů. Obfuskační transformace mohou ale také upravovat kód tak, aby deobfuskace pro reverzního inženýra zesložitila použití debuggeru.

### 2.2.1 Jednosměrná obfuskační transformace

Jedná se o podmnožinu obfuskačních transformací, které mají navíc vlastnost, že po aplikování transformace není možné kód vrátit do původní podoby. Většinou je ale možné takovou transformaci alespoň částečně anulovat a navrátit kód do pravděpodobné původní podoby.

### 2.2.2 Kvalita obfuskace

Kvalitu obfuskovaného kódu lze zkoumat empiricky, kdy je kvalita vyhodnocena uživatelským testováním. Jako měřítko je zkoumáno, za jak dlouho dobu byl uživatel schopný porozumět obfuskovanému kódu.

U obfuskace lze podle [2] sledovat tři různé parametry, jejichž hodnoty jsou určené charakterem dané transformace. Tyto parametry mohou analytickým přístupem také sloužit k určení kvality obfuskace.

### 2.2.3 Potence

Potence vyjadřuje míru zesložitění čitelnosti zdrojového kódu pro člověka. Potenci lze měřit různými metrikami, jako je například počet řádků kódu, úrovně zanoření nebo počet proměnných v kódu. Při vývoji se programátor snaží potenci minimalizovat, aby byl kód jednoduše srozumitelný pro další programátory, kteří s kódem budou pracovat. Naopak při obfuskaci je snahou potenci zvýšit, čímž se zvyšují nároky na člověka pro pochopení kódu.

### 2.2.4 Odolnost

Odolnost vyjadřuje schopnost obfuskovaného kódu odolávat deobfuskaci a tím předejít odhalení původního kódu. Lze ji rozdělit na dvě části. První částí tvoří energie, kterou musel vynaložit člověk, aby vytvořil deobfuskátor pro vstupní kód. Tedy musí porozumět kódu do takové míry, aby byl schopen vytvořit deobfuskátor na použité obfuskační transformace. Druhou částí je čas výpočtu a paměťové požadavky deobfuskátoru pro provedení zpětné transformace.

### 2.2.5 Složitost

Složitost vyjadřuje míru přidané komplexity do kódu, která zvýší časové nebo paměťové nároky na běh kódu. Přidáním obfuskačních transformací se výsledná složitost kódu může zvyšovat. Je tedy vždy na místě zvážit, zda váha pozitivního obfuskačního efektu transformace je ospravedlnitelná za případnou zvýšenou složitost kódu.

## 2.3 Deobfuskace

Deobfuskace je opačný proces k obfuskaci, tedy jedná se o takovou transformací zdrojového kódu, jejíž snahou je navrátit kód do podoby před obfuskací. Aby deobfuskační transformace mohla vzniknout, je nejdříve nezbytné z kódu pochopit, jaké obfuskace jsou na kód použity. K tomu může být využita statická nebo dynamická analýza kódu.

### 2.3.1 Statická analýza

Statická analýzá je podle [3] metoda, která slouží k analýze kódu bez jeho samotného spuštění. Je tak vhodnou metodou v případech, kdy je podezření, že daný kód by mohl být nebezpečný a jehož spuštění by mohlo znamenat bezpečnostní hrozbu. Statická analýza pro snadnější manipulaci s kódem konstruuje dle daného kódu strukturu, která abstrahuje zdrojový kód a usnadní jeho další úpravy.

### 2.3.2 Dynamická analýza

Dynamická analýza podle [3] naopak slouží k analýze kódu při běhu programu. K tomu se nejčastěji využívá debugger, který obsahuje podpurné funkce pro analýzu běžícího programu. Je například možné na libovolném místě zastavit provádění kódu pomocí breakpointu a zároveň sledovat aktuální hodnoty proměnných v daném místě. V některých případech může být také upraven kód při běhu programu a je tak možné na libovolné místo doplnit podpůrné příkazy pro usnadnění deobfuskace. Typicky se při dynamické analýze testuje chování programu pro různé vstupní hodnoty a probíhá pozorování chování takového programu.

Dynamickou analýzu ale nemusí být vždy možné použít, protože kód se takové analýze může bránit. V takovém případě je nutné nejdříve přistoupit ke statické analýze kódu a odstranit tyto zábrany. V kódu může být například počítán čas mezi provedením vybraných příkazů, takže při použití breakpointů, kdy je kód zastaven, může být tolerovaný čas překročen. Může být také kódem kontrolována vlastní integrita, zda se nějaká část kódu nezměnila. Při zjištění takových skutečností může být vykonávání programu přesměrováno do slepé větve.

Mohou být také programem upraveny globální proměnné a objekty, které jsou běžně dostupné v prohlížeči. Reverzní inženýr by takové objekty mohl chtít použít jako podpůrné nástroje při deobfuskaci, ale nemůže se na ně v tomto případě spolehnout, protože jejich funkčnost může být upravena. Příkladem takového objektu je konzole, která může sloužit pro vypsání zadaného výrazu.

## 2.4 Abstraktní syntaktický strom

Abstraktní syntaktický strom (AST) je podle [4] stromová struktura, která reprezentuje zdrojový kód a využívá se při statické analýze. Strom se skládá z různých uzlů, kde každý uzel reprezentuje určitý příkaz či část zdrojového kódu. Vzniká tak určitá abstrakce nad zdrojovým kódem, která umožňuje strojově jednodušeji manipulovat se zdrojovým kódem. Abstraktní syntaktické stromy se tedy využívají zejména při překladu, optimalizaci nebo jiné transformaci zdrojového kódu.

Pro použité termíny v dalších kapitolách jsou zmíněny typy uzlů v následujícím seznamu.

- **Identifikátor**

  Identifikátor reprezentuje název konkrétní proměnné, funkce nebo třídy v kódu.

- **Literál**

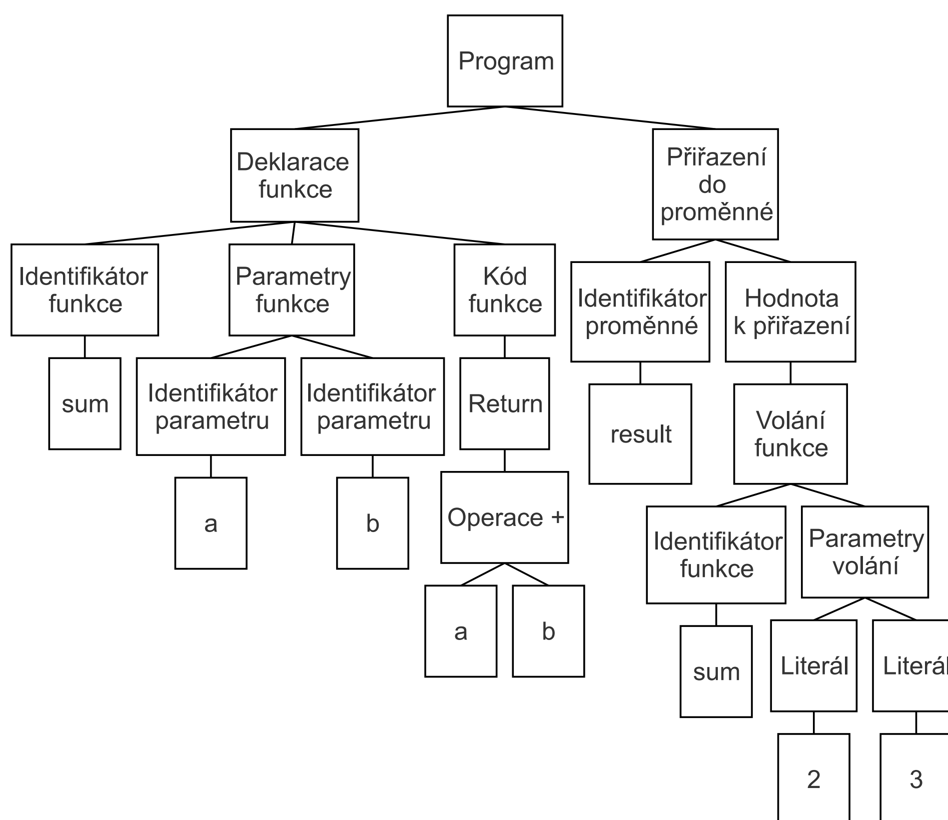
  Literál reprezentuje nějakou konkrétní hodnotu, například číslo, řetězec nebo boolean hodnotu.

- **Blok**

  Blok reprezentuje souvislou posloupnost příkazů, například příkazy v těle cyklu nebo ve větvi podmínky.

```
function sum(a, b) {
  return a + b;
}
result = sum(2, 3);
```

Kód 2.1: Ukázka kódu pro sestavení AST na obrázku 2.4




Obrázek 21: Ukázka abstraktního syntaktického stromu pro kód 2.1

## 2.5 ECMAScript

ECMAScript (ES) je podle [5] specifikace skriptovacího jazyka, která vznikla v roce 1997 za účelem sjednotit rozdílné implementace jazyka JavaScript. K tomu docházelo z toho důvodu, že různé webové prohlížeče používaly vlastní implementaci tohoto jazyka. JavaScript ale není jedinou implementací této specifikace, vznikla jich celá řáda.

V posledních letech se vývoj ES značně urychlil a nové verze vycházejí v ročním cyklu. To ale představuje problém pro webové prohlížeče, u kterých je určitá časová prodleva v implementaci nových verzí. V současné době webové prohlížeče umožňují bezpečně používat ES verze 5.1 z roku 2009, ale většina stávajících webových prohlížečů podporuje podle [6] i ES verze 6 (nebo také ES2015) z roku 2015. Aktuálně poslední číslo verze je 9 (nebo také ES2018)

z roku 2018. Není tedy zatím možné nové verze ES přímo využívat, neboť by pro nezanedbatelnou část uživatelů nemusel pracovat korektně kvůli implementaci ve webovém prohlížeči. Aby vývojáři mohli pracovat s nejnovější verzí ES, vznikly kompilátory a transpilery, které dokážou kód nové verze ES převést na kód starší verze, který funguje identicky.

## 2.6 JavaScript

JavaScript je podle [7] interpretovaný skriptovací jazyk vyhovující specifikaci ES. Jazyk vznikl v roce 1995 a prvotně sloužil primárně pro přidání interaktivity webovým stránkám. Nyní má ale JavaScript mnohem širší možnosti použití a dle statistik [8] se jedná o nejpoužívanější jazyk vůbec.

Využívá se nadále nejčastěji na webových stránkách, kde se jeho možnosti použití výrazně rozšířily. Mimo zmíněnou interaktivitu se používá například pro sledování pohybu uživatele na webových stránkách, asynchronní načítání dat nebo dokonce i pro těžení kryptoměn. Skripty napsané v JavaScriptu jsou ve své originální podobě dostupné uživateli prohlížeče a může s nimi i manipulovat. Při volbě obfuskačních transformací tak bude kladen důraz právě na skripty cílené pro použití na webových stránkách.

JavaScript je ale možné využít i na straně serveru. K tomu je nejčastěji využívána platforma NodeJS. Tato platforma slouží primárně pro implementaci serverové části webové aplikace, ale může být na serveru využita i pro spuštění jednorázových programů. Implementace serverové části v jazyku JavaScriptu mimo jiné přináší tu výhodu, že vhodné soubory je možné použít zároveň na straně serveru i na straně klienta.

Dále je možné použít JavaScript pro tvorbu grafických desktopových aplikací či mobilních aplikací. Výhodou je, že tyto aplikace jsou multiplatformní, ale přesto umějí využívat či simulovat nativní grafické prvky. Na druhou stranu ale mnohdy mají znatelně větší hardwarové nároky než ekvivalentní nativní aplikace.

### 2.6.1 Scope v JavaScriptu

Scope je podle [9] označení pro rámec působnosti a určuje, odkud jsou jaké proměnné dostupné. Ve většině běžných programovacích jazycích bývá scope ohraničen blokem, v JavaScriptu je ale ohraničen funkcí. Z vnitřního scope je možné přistupovat k proměnným, které byly deklarované v nadřazených scope. Z vnějšího scope ale není možné přistupovat k proměnných, které byly deklarované v jeho vnitřních scope. Speciálním případem je globální scope, který sám o sobě nepředstavuje funkci, ale jeho scope má stejné vlastnosti.

### 2.6.2 Striktní mód JavaScriptu

JavaScript má podle [10] od verze vyhovující specifikaci ES 5 podporu pro striktní mód.

```
'use strict';
```

Kód 2.2: Zapnutí striktního módu

Striktní mód může být aktivován v libovolném scope včetně globálního a jeho působnost se tak odráží od místa jeho definování. Musí být ale vždy aktivován prvním příkazem v daném scope. Zapnutí ve zvoleném scope vynutí striktní mód i pro všechny podřízené scope.

Striktní mód napomáhá psát bezpečnější kód a dělá ho průhlednějším. Zakazuje použití určitých konstrukcí, které není bezpečné používat. Přináší hlášení chyb, kterých se programátor v kódu dopustil. Proměnné, které byly definovány ve funkci eval, nejsou v striktním módu přístupné v okolním kontextu.

### 2.6.3 Kompilátory a transpilery

JavaScript má podle [11] k dispozici různé kompilátory a transpilery, které jsou využívány pro zajištění kompatibility s většinou webových prohlížečů. Patři mezi ně například kompilátor Babel, který umožňuje převést kód nejnovější verze specifikace ES na kód kompatiblní se starší verzí specifikace ES. Výhodou tedy je, že programátoři tak mohou při vývoji pracovat s kódem nejnovější verze specifikace ES, protože kód je následně převeden na kompatibilní verzi specifikace ES s prohlížeči.

Dále jsou k dispozici transpilery, které umožňují převést jiný programovací jazyk na jazyk JavaScript. Typicky se jedná o nadmnožinu jazyka JavaScript, která do jazyka přináší nové funkce a vlastnosti. U některých transpilerů je také upravena syntaxe jazyka. Zástupcem transpilerů je například jazyk TypeScript, který rovněž umožňuje používat nejnovější verzi ES, ale který především do jazyka JavaScript přidává typovou podporu a kontrolu. Tím značně zpřehledňuje kód a usnadňuje práci při vývoji a ladění.

Kapitola 3

# Existující práce a knihovny

Tato kapitola se zabývá analýzou bakalářské práce, z které tato práce částečné vychází. Také jsou zde uvedeny a popsány volně dostupné nástroje a knihovny, které obfuskují jazyk JavaScript.

## 3.1 Bakalářská práce Samuela Hanáka

Bakalářská práce Samuela Hanáka [12] obsahuje širokou škálu různých obfuskačních transformací. Většina transformací využívá obecné obfuskační principy nezávislé na programovacím jazyce, ale obsahuje i některé transformace specifické pro jazyk JavaScript. Obfuskační transformace jsou v práci voleny tak, aby plnily funkci znečitělnění zdrojového kódu. V práci se nenachází transformace, která by chránila kód před modifikací a zajišťovala tak integritu kódu. Také v práci nejsou obsaženy transformace, které by byly cílené na znesnadnění analýzy kódu při využití dynamické analýzy a debuggeru.

Práce je postavena na platformě NodeJS a je psána v JavaScriptu bez použití kompilátorů či transpilerů. Transformace obsažené v práci jsou implementovány každá ve svém vlastním souboru. Obfuskátor ale nemá ideální modulární strukturu, protože při přidání nové transformace se musí upravovat i zdrojové kódy, které zašťiťují obfuskátor jako takový. Pokud by vzniknul požadavek na vypnutí nebo upravení konfigurace některé z transformací, rovněž tak by bylo nutné upravit zdrojové kódy.

### 3.1.1 Obfuskační transformace

Část transformací se zabývá přeuspořádáním jednotlivých částí kódu. Z této kategorie se v práci nacházejí transformace pro přeuspořádání parametrů funkcí, přeuspořádání deklarací funkcí, přeuspořádání deklarací proměnných a přeuspořádání příkazů v bloku.

Další transformace přesouvá operátory do funkcí a tím je tak skrývá na původním místě. Přesouvá ale pouze binární operátory a je zde prostor pro

rozšíření funkcionality o unární operátory. Také je možné provést transformaci přiřazujících výrazů do takové podoby, kdy na pravé straně vznikne binární operace a operátor této operace tak může být přesunut do funkce.

Dále je v práci transformace pro přesunutí příkazu do funkce eval. I tato transformace má prostor pro vylepšení. Implementace v práci umožňuje vybrat takový příkaz pro přesunutí, který se nachází přímo uvnitř smyčky. To by mohlo znamenat zásadní zpomalení doby běhu programu.

Práce má zajímavý přístup k obfuskaci řetězců, který ale do této práce nebyl přenesen. Byl zvolen jiný přístup, který je vysvětlen v kapitole, která se zabývá realizací.

Z této bakalářské práce byly do této práce převzaty myšlenky hned několika transformací. U části z nich byly navíc provedeny různé modifikace, které jsou popsané kapitole, která popisuje realizaci transformací.

## 3.2 Volně dostupné knihovny

### 3.2.1 Confusion [13]

Tento obfuskátor se zaměřuje na obfuskaci řetězců. Přesouvá veškeré řetězce do jednoho globálního pole a tím skrývá jejich spojitost s původním místem řetězce. Na původním místě však vzniká přímá reference indexem na jejich nové místo a cílená deobfuskace by tak v tomto případě byla triviální.

### 3.2.2 JavaScript Obfuscator Tool [14]

Tento obfuskátor obsahuje nejkomplexnější možnosti obfuskace ze zkoumaných volně dostupných obfuskátorů. Obsahuje i minifikační transformace, které zahrnují odstranění bílých znaků a komentářů. Má možnost přidání ochranné transformace, která znemožní správné fungování programu při modifikaci kódu.

Nabízí širší možnosti obfuskace řetězců. Jednotlivé řetězce umí podobně jako obfuskátor Confusion přesunout do globálního pole. Další část transformace kóduje řetězce pomocí Base64 nebo RC4. Také obsahuje transformaci pro převedení řětězců na jejich unicode reprezentaci.

### 3.2.3 JavaScript Obfuscator [15]

Tento obfuskátor na první pohled provádí poměrně zajímavou transformaci kódu. Z celého kódu jsou vyňaty veškeré posloupnosti znaků obsahující pouze znaky základní abecedy. Ty jsou následně seřazeny podle četnosti výskytu a na jejich původní místo je dosazen číselný index odkazující na původní vyňatou část kódu. Do kódu je přidána funkce, která takto upravený kód konvertuje zpět na původní kód a ten je následně vyhodnocen funkcí eval. Výsledný obfuskovaný kód tedy působí značně nesrozumitelně. Problémem

této knihovny však je, že jednoduchým vypsáním argumentu funkce eval do konzole lze získat zpět původní kód, a proto v praxi není moc použitelná.

### 3.2.4 JSFuck [16]

Tento obfuskátor konvertuje zdrojový kód na kód složený pouze ze 6 různých znaků. Je postavený na vlastnosti JavaScriptu, že lze spustit libovolný kód, který je možné zapsat jako řětězec. Výsledný kód je člověkem bez skutečně velkého úsilí takřka nečitelný. Nevýhodou obfuskátoru ale je, že výsledný kód je řádově delší než zdrojový kód. Například jednoduchý kód `console.log("test")`, který obsahuje 19 znaků je převeden na kód, který obsahuje 15896 znaků. Zároveň se doba vykonání takto obfuskovaného kódu neúměrně zvyšuje. Další nevýhodou je, že již existují deobfuskátory, které dokážou poměrně spolehlivě získat zpět původní zdrojový kód.

# Kapitola 4

# Analýza a návrh

Tato kapitola se zabývá analýzou a návrhem, na základě kterých následně dojde k realizaci obfuskátoru.

## 4.1 Programovací jazyk

Obfuskátor bude dle zadání postaven platformě NodeJS. Musí být tedy napsán v takovém programovacím jazyce, aby mohl být na platformě NodeJS spuštěn. Pro obfuskátor byl zvolen jazyk TypeScript, který je transpilován do jazyku JavaScript, který je spustitelný na platformě NodeJS.

Jazyk TypeScript je podle [17] nadmnožinou jazyka JavaScript a přináší tak nové funkce. Zásadní důvodem pro zvolení tohoto jazyka je typová podpora a kontrola. V práci bude u každé transformace pracováno s AST, jehož uzly mají různé typy dle jejich významu. Typová podpora tedy usnadní orientaci v kódu a statická typová kontrola předejde chybám, které by mohly nastat nedůsledným programováním. Další výhodou jazyka TypeScript je, že umožňuje psát kód nejnovějších verzí specifikace ES. Je to jazyk objektově orientovaný a umožňuje tak psát dobře strukturovaný kód.

Vedlejší pozitivní vlastností toho, že se jedná o jazyk transpilovaný je skutečnost, že se musí provést jeho kompilace. Při kompilaci může kompilátor odhalit množství chyb, které programátor při programování mohl přehlédnout. To je výhoda oproti JavaScriptu, u kterého není prováděna kompilace, protože se jedná o jazyk interpretovaný.

```
class Greeter {
  greeting: string;
  constructor(message: string) {
    this.greeting = message;
  }
  greet() {
    return "Hello, " + this.greeting;
  }
```

```
}

let greeter = new Greeter("world");
console.log(greeter.greet());
```

Kód 4.1: Ukázka kódu v jazyce TypeScript

## 4.2 Návrh struktury obfuskátoru

Při návrhu struktury obfuskátoru je bráno v potaz několik různých faktorů. Struktura kódu by měla být modulární. To znamená, že bude možné manipulovat s jednotlivými transformacemi obfuskátoru bez nutnosti zasahovat do kódu, který zastřešuje obfuskátor jako takový. Bude možné měnit pořadí jednotlivých transformací, přidávat nové transformace, měnit konfiguraci transformací a odebírat transformace. Tyto všechny modifikace bude možné provést bez zásahu do kódu.

Aby tyto modifikace bylo možné provést bez zásahu do kódu, je k obfuskátor nedílně spjat konfigurační soubor ve formátu YAML pro serializaci strukturovaných dat. V tomto konfiguračních souborů se nachází obecná konfigurace pro samotný obfuskátor, například verze ES vstupního kódu nebo zda mají být do konzole vypisovány informace o průběhu obfuskace kódu. Dále se zde nachází seřazená posloupnost definicí transformací, které mají být na kód aplikovány. Jejich pořadí v tomto souboru určuje i následné pořadí při provádění transformací obfuskátorem. U jednotlivých definic transformací je možné nastavit jejich název, aby bylo srozumitelné, o jakou transformaci se jedná. Dále je zde uvedena cesta k souboru, ve kterém se nachází daná transformace, a kterou obfuskátor použije pro zahrnutí transformace. Jednotlivé definice je také možné deaktivovat, ale zároveň je zachovat v tomto konfigurčním souboru. Každá definice zde může mít také svojí vlastní konfiguraci pro odpovídající transformaci.

Taková struktura konfiguračního souboru umožňuje i vícenásobné spuštění stejné transformace, a to i s možností nastavit odlišnou konfiguraci. To může být užitečné, pokud transformace přinese obfuskační efekt před aplikováním nějaké jiné transformace, ale zároveň i po jejím aplikování.

```
ecmaVersion: 5
verbose: true
optimizeInput: false
optimizeOutput: false
identifiers:
  rename: true
  renameGlobals: true
stream:
  - name: Debugger Breakpoint Loop
    file: ./transformations/debuggerBreakpointLoop.js
```

```
    enabled: true
  - name: Operator Outlining
    file: ./transformations/operatorOutlining.js
    enabled: true
    settings:
      unaryOperatorChance: 0.75        //75% chance
      assignmentOperatorChance: 0.75   //75% chance
      binaryOperatorChance: 0.75       //75% chance
  - name: Console Redefinition
    file: ./transformations/consoleRedefinition.js
    enabled: false
```

Kód 4.2: Příklad obsahu konfiguračního souboru ve formátu YAML

Výhodou také je, že mohou být vytvořené různé konfigurační soubory v závislosti na jejich použití. Může tak být připraveno více různých konfiguračních souborů a pouhým nahrazením aktivního konfiguračního souboru dojde k jeho použití.

Jednotlivé transformace jsou implementovány jako třídy, které jsou potomkem společné abstraktní třídy. Tato abstraktní třída zajišťuje, že každá transformace bude mít k dispozici kompletní AST programu a svojí konfiguraci z konfiguračního souboru. Transformace tak má veškeré nutné vstupy pro svoje provedení.

Obfuskátor prvním parametrem na vstupu přijímá cestu k zdrojovému souboru. Může být zadán i druhý parametr pro cílovou lokaci obfuskovaného kódu. Pokud nebude cílová lokace zadána, použije se cesta zdrojového souboru, ve které se na konci názvu souboru přidá přípona obf.

## 4.3 Zaměření transformací

Pokud u některé z transformací dojde k situaci, kdy se minifikační a obfuskační efekt transformace vzájemně vylučuje, bude upřednostněn obfuskační efekt.

Obfuskační transformace budou v práci voleny tak, aby pokryly více obfuskačních funkcí. Největší část budou tvořit transformace, které budou cílit na sníženi srozumitelnosti kódu a znesnadnění navrácení kódu do původní podoby. Bude také zahrnuta transformace, která bude šifrovat kód a zároveň zajišťovat integritu kódu. Kód tak bude chráněn před modifikací. Poslední část tvoří transformace, které znesnadní reverznímu inženýrovi deobfuskaci v debuggeru při dynamické analýze.

## 4.4 Omezení pro vstupní kód

Aby obfuskace na vstupní kód mohla být provedena, musí být splněno několik podmínek.

### 4.4.1 Funkce eval

Funkce eval podle [18] v JavaScriptu umožňuje vykonat kód, který je funkci předán parametrem jako řetězec. Volání takové funkce v kódu pro obfuskační transformace představuje problém, protože by při statické analýze nebyl brán v potaz kód, který se vyskytuje jako řetězec ve funkci eval. V takovém kódu by se mohly nacházet reference na vnější proměnné nebo funkce, které by kvůli tomu nemohly být přejmenovány. Volání funkce eval je tedy obfuskátorem zakázáno.

### 4.4.2 Příkaz with

Příkaz with podle [19] svůj vnitřní blok rozšiřuje o proměnné, které vzniknou z vlastností vstupního objektu příkazu with. Pokud v daném scope již existuje identifikátor se stejným názvem jako nějaká vlastnost daného objektu, získává hodnota vlastnosti objektu přednost.

```
var a = 2;
var o = {
  test: 'Example'
  a: 5,
}

with ( o ) {
  console.log(test); // 'Example'
  console.log(a + a); // 10
}
```

Kód 4.3: Ukázka použití příkazu with

I v tomto případě by tedy nastal problém s přejmenováním identifikátorů. Nebylo by možné rozhodnout, kdy je možné identifikátor přejmenovat, protože při statické analýze nejsou známy všechny vlastnosti objektu a tedy ve vnitřním bloku příkazu with nebude jisté, kdy se jedná o původní proměnnou a kdy o vlastnost objektu.

Nicméně se jedná o zastaralý příkaz, který dnes již není moc používán, protože byl zdrojem častých chyb právě kvuli vlastnosti, kdy je hodnota proměnné uvnitř bloku přepsána hodnotou vlastnosti objektu. Jeho zakázání tedy není žádný velký ústupek.

### 4.4.3 Vlastnost funkce s jejim názvem

Podle [20] každá funkce v JavaScriptu obsahuje vlastnost name, jejíž hodnotou je název funkce ve formě řetězce. Použití takové vlastnosti v kódu by mohlo znamenat problém pro obfuskační transformace, protože by žádná funkce nemohla být přejmenována. Tato vlastnost ale běžně není používána, takže její zakázání není velkou překážku.

## 4.5 Použité nástroje knihovny

### 4.5.1 Specifikace ESTree

ESTree [21], vycházející ze specifikace SpiderMonkey AST, je specifikace abstraktního syntaktického stromu pro jazyky vyhovujícím specifikaci ES. Pro jednotlivé příkazy a konstrukce jazyka definuje uzly, jejich vlastnosti, typy těchto vlastností a jejich možné hodnoty. Tato specifikace tak slouží k sjednocení různých nástrojů pro práci s ES kódem a ulehčení jejich komunikace mezi sebou. Typicky například parser vytvoří AST ze vstupního kódu a následně nezávislý generátor kódu vytvoří z daného AST znovu kód.

Specifikace jednotlivých uzlů také přináší výhodu u programovacích jazyků s typovou podporou, v případě této práce u jazyka TypeScript. V kódu je tak poté možné pracovat s uzly stejně jako s ostatními datovými typy.

### 4.5.2 Parser Espree

Espree [22] je parser jazyků vyhovujících specifikaci ES. Původně vycházel z parseru Esprima, ale nyní je postaven na základech parseru Acorn, který díky modulární architektuře umožňuje rozšíření základních funkcí. Parser Espree podporuje jazyky nejnovější verze specifikace ES.

### 4.5.3 Generátor kódu Escodegen

Escodegen [23] generuje výsledný kód na základě vstupního AST dle specifikace ESTree. Pomocí parametrů umožňuje různé nastavení formátu výstupního kódu. Například pomocí kterého bílého znaku bude provedeno odsazení kódu nebo zda mají být v kódu redundantní středníky a závorky. Je ale také možné nastavit, aby byl výsledny kód kompaktní, tedy aby do kódu nebyly zaneseny žádné bílé znaky.

### 4.5.4 Nástroj Estraverse

Nástroj Estraverse slouží k procházení a modifikaci AST [24]. Je stěžejním nástrojem pro implementované transformace v této práci. Na vstupu přijímá AST dle specifikace ESTree a provádí jeho průchd. Umožňuje navštívit každý uzel AST a provést jeho analýzu nebo nahrazení. Také parametrem poskytuje informace o nadřazeném uzlu, pokud existuje. Analýzu a náhradu umožňuje provést nejen při procházení směrem k listům AST, ale také při návratu směrem ke kořenu.

Nástroj je složený ze dvou funkcí. První funkce traverse umožňuje průchod AST. Na místě konkrétního uzlu lze vynechat procházení jeho větve AST nebo kompletně zastavit průchod AST. Druhá funkce replace umožňuje navíc modifikaci nebo kompletní nahrazení uzlu. U této funkce je také možné odříznutí celé větve AST v místě daného uzlu.

```
estraverse.traverse(ast, {
  enter: function (node, parent) {
    if (node.type == 'FunctionExpression' || node.type
        == 'FunctionDeclaration')
      return estraverse.VisitorOption.Skip;
  },
  leave: function (node, parent) {
    if (node.type == 'VariableDeclarator')
      console.log(node.id.name);
  }
});
```

Kód 4.4: Ukázka použití nástroje Estraverse [24]

### 4.5.5 Nástroj Esmangle

Nástroj Esmangle umožňuje optimalizovat AST [25]. Průchodem AST hledá příkazy nebo částí kódu, které jsou redundantní a nemají efekt na chod programu nebo nemohou nikdy nastat. Takové příkazy nebo části kódu potom upraví nebo odstraní.

### 4.5.6 Nástroj Escope

Nástroj Escope slouží k analyzování scope v JavaScriptu [26]. Na vstupu je AST dle specifikace ESTree a výsledkem je přehled všech nalezených scope. U jednotlivých scope zjistí například dostupné a deklarované proměnné a jejich reference. Pro každý scope také poskytuje informace o vnitřních a nadřazených scope.

### 4.5.7 Nástroj Estemplate

Nástroj Estemplate [27] generuje AST ze vstupního řetězce, který obsahuje kód. To samo o sobě tedy plní stejnou funkci jako parser, ale tento nástroj navíc umožňuje v řetězci definovat různé části jako proměnné, které budou následně nahrazeny uzly AST, které jsou předány dalším parametrem tohoto nástroje. V okamžiku, kdy je potřeba generovat rozsáhlejší část AST a do ní doplnit několik existujících uzlů, je použití této knihovny vhodnou volbou. Odpadá tak nutnost definovat v kódu celou strukturu AST, která může být velice dlouhá, špatně srozumitelná a nepřehledná. Kód se tedy při použití tohoto nástroje zapíše jako řetězec a pouze budou doplněny případné existující uzly na vybraná místa.

```
var ast = estemplate('var <%= varName %> = <%= value %>
    + 1;',{
  varName: {type: 'Identifier', name: 'myVar'},
  value: {type: 'Literal', value: 123}
```

```
});
```

Kód 4.5: Ukázka použití nástroje Estemplate [27]

Kapitola 5

# Realizace

V této kapitole jsou popsaný transformace, které byly v rámci práce implementovány do obfuskátoru.

## 5.1 Odstranění bílých znaků a komentářů

Z kódu jsou odstraněny bílé znaky, mezi které patří například mezera, tabulátor nebo odřádkování. Tyto znaky jsou v kódu použity především pro vizuální separaci jednotlivých příkazů nebo částí kódu. Mohou ale také vizualizovat úroveň zanoření jednotlivých bloků. Pomáhají tedy snadnější orientaci v kódu. Odstraněním těchto znaků tak dojde ke snížení přehlednosti kódu a tím ke zvýšení potence kódu. Vedlejším efektem této minifikační transformace tedy je, že se jedná i o transformaci obfuskační. Potence kódu se po odstranění bílých znaků zvyšuje, ale deobfuskace této transformace je přímočará operace. Navíc pro deminifikaci je k dispozici mnoho volně dostupných nástrojů a většinou jde i o zabudovanou funkci editorů nebo vývojových prostředí.

Komentáře v kódu mohou plnit několik odlišných funkcí.

- Mohou pomáhat dalším lidem k snadnějšímu pochopení určitých příkazů nebo částí kódu. Tedy mohou prozradit i bezpečnostní postupy v kódu či jiné citlivé informace.

- Mohou složit jako dokumentace jednotlivých proměnných, funkcí nebo tříd. Například u funkce mohou být obsaženy informace o datových typech vstupních a výstupních proměnných. Zároveň u typů bývají i samotné názvy těchto proměnných. Tato dokumentace napomáhá ke snadnější práci ve vývojovém prostředí, které dokáže tyto informace použít a napovídat programátorovi při psaní kódu.

- Na začátku souboru může být popsán význam samotného souboru a příklady jeho použití. Může zde být i informace o verzi kódu.

- Může být zakomentován i zdrojový kód, který například z nějakého důvodu nebyl použit.

Komentáře v běhu programu neplní žádnou funkci a jejich odstraněním tak dojde ke zmenšení výsledného kódu. Zároveň jejich odstraněním dojde i ke snížení srozumitelnosti kódu a tím se zvýší potence. Odstranění komentářu je tedy zároveň také jednosměrnou obfuskační transformací. Všechny výše zmíněné funkce komentářů mohou prozradit důležité informace o kódu, které by mohly vést k snadnější deobfuskaci.

### 5.1.1 Implementace transformace

Odstranění bílých znaků je transformace, kterou v sobě má zabudován generátor kódu escodegen. Při generování kódu lze nastavit parametr compact, který ovlivňuje, zda se provede transformace.

K odstranění komentářů dojde pomocí knihovny espree, která na základě konfigurace knihovny do abstraktního syntaktického stromu nezanese žádné komentáře.

## 5.2 Odstranění redundantních znaků v kódu

V kódu se mohou vyskytovat symboly, které mohou být v kódu z důvodu dohodnutého standardu psaní kódu, ale které jsou redundantní. Jejich odstraněním tak dojde ke zmenšení programu bez vlivu na funkčnost. Jedná se tedy o transformaci minifikační.

Z kódu se v určitých případech mohou odstranit závorky, pokud mezi nimi není žádný obsah. Například při vytváření objektu je možné závorky vynechat.

```
// oba zapisy jsou mozne
var a = new Object();
var b = new Object;
```

Kód 5.1: Ukázka vytvoření nového objektu bez argumentů

Dále JavaScript nevyžaduje zakončení příkazu středníkem, pokud je na daném řádku právě jeden příkaz. Odstraněním bílých znaků ale došlo k tomu, že celý kód bude pouze na jednom řádku, a proto není možné odstranit všechny středníky. Je ale stále možné odstranit středník za posledním příkazem v každém bloku.

Tyto transformace kódu provede knihovna escodegen na základě nastavení parametrů semicolons a parentheses.

## 5.3 Optimalizace zdrojového kódu

Jedná se o sadu transformací, které v kódu lokalizují a upraví nebo odstraní části, které jsou redundantní nebo neoptimálně zapsané. Tyto transformace

tedy mají minifikační povahu a jejich aplikováním spíše dochází ke snížení obfuskace kódu, neboť jsou z kódu například odstraněny nedostupné bloky, které by mohly ztížit a prodloužit deobfuskaci. Při úvaze, jestli tuto transformaci na kód aplikovat, je třeba zvážit, zda je větší prioritou minifikace nebo obfuskace.

Samotná optimalizace se skládá z více různých transformací. Pro představu bude uvedeno několik příkladů.

- Mohou být například odstraněny větve podmínek, do kterých se program nemůže nikdy dostat.
- Mohou být odstraněny příkazy, které se nemohou nikdy provést. Typicky například příkazy, které se nacházejí za příkazem return, break nebo continue.
- Boolean hodnoty true a false mohou být nahrazeny kratším ekvivalentem !0 a !1.
- Jednotlivé deklarace proměnných, pokud to bude v daném kontextu možné, mohou být sjednoceny do jednoho příkazu.

Implementaci těchto transformací zajištuje knihovna esmangle. Ve výchozím nastavení jsou tyto transformace vypnuty, neboť je v této práci upřednostěna obfuskace před minifikací.

## 5.4 Přejmenování identifikátorů

Identifikátory ve svém názvu zpravidla ukrývají informaci o svém významu a funkci v kódu. Přejmenováním identifikátorů dojde k odstranění této informace. Jedná se o transformaci jednosměrnou, protože po provedení se v kódu nikde nenacházejí původní názvy. Deobfuskace této transformace je tedy strojově nemožná.

### 5.4.1 Vlastnosti transformace

Tato transformace zvyšuje potenci kódu, neboť přejmenováním identifikátorů člověk ztratí informaci o původním významu jednotlivých proměnných a funkcí. Typicky například číselný iterátor ve smyčce je označen jako $i$. Už z názvu takto pojmenované proměnné bude reverznímu inženýrovi jasné, jakou plní v kódu funkci. Transformaci lze pojmout jako minifikační, kdy je jednotlivým identifikátorům přidělen nejkratší možný název. Zároveň je ale i obfuskační vzhledem ke ztrátě informace o významu. Pokud se k transformaci přistoupí z pohledu minifikace a použije se pro identifikátory nejkratší možný název, stává se pro reverzního inženýra snáze zapamatovatelným. Naopak z pohledu obfuskace, kdy je použit delší název, je potence vyšší, neboť název nebude

pro reverzního inženýra tak snadno zapamatovatelný. Formát delšího názvu navíc může být volen s takovým cílem, aby zmátnul reverzního inženýra tím, že použije název podobný klíčovým slovům nebo literálům.

Transformace má vysokou odolnost, protože člověk musí vynaložit velké úsilí, aby kompletně porozumněl významu jednotlivých identifikátorů. Existuje knihovna JSNice [28], která dokáže jednotlivým identifikátorům přiřadit pravděpodobný název podle jeho významu v kódu. K tomu využívá mnoho nasbíraných zdrojových kódu, z kterých statisticky vyvozuje pravděpodobný původní název identifikátoru. Nicméně ani tato knihovna nemůže z principu fungovat spolehlivě a určitá míra odolnosti v kódu zůstane.

Naopak nikterak se nezvyšuje složitost výsledného kódu a je proto možné z tohoto hlediska tuto transformaci použít na celý kód bez žádných omezení. Výjimkou jsou globální proměnné a funkce, u kterých je v určitých situacích oprávněné požadovat, aby jejich původní název zůstal zachován. Důvodem pro zachování může být skutečnost, že tyto proměnné a funkce mohou být využívaný i mimo samotný kód, který byl zadán jako vstup pro obfuskaci.

### 5.4.2 Implementace transformace

Nové identifikátory se zvolí tak, aby působily náhodně a zároveň aby nedošlo ke kolizi s jiným identifikátorem. Rovněž tak nemůže být použít identifikátor, který by kolidoval s nějakým klíčovým slovem samotného jazyka.

Speciálním případem jsou globální proměnné a funkce, u kterých není jasné, zda lze bezpečně provést jejich přejmenování. Pokud se jedná o kód určený pro prohlížeč, tak lze ke globálním proměnných přistoupit i pomocí objektu window. Přístup ke globální proměnné tak může být učiněn indexem do objektu window pomocí výrazu, jehož hodnotu nemusí být možné určit statickou analýzou.

```
a = 5;
// ruzne zpusoby pristupu ke globalni promenne v
   prohlizeci
console.log(window["a"]);
console.log(window.a);
console.log(a);
```

Kód 5.2: Ukázky přístupu ke globální proměnné

Zároveň ale také není jasné, jaké nové identifikátory je možné zvolit, aby nebyl vybrán již existující globální identifikátor, který se deklaroval mimo vstupní kód. Vyžadované chování bude záviset na charakteru vstupního kódu. V konfiguračním souboru obfuskátoru je tedy možné nastavit, zda se mají identifikátory globálních proměnných a funkcí přejmenovávat.

Při výběru nového názvu identifikátoru je v implementaci transformace upřednostněna obufuskační povaha před minifikační a nové názvy tedy nebudou nejkratší možné. Validní název identifikátoru v JavaScriptu ve verzi ES

5 může začínat dolarem, podtržítkem nebo libovolným písmenem. Při transformaci kódu, který je určen pro webové prohlížeče, je třeba brát v úvahu speciální proměnné a funkce, které jsou v kódu dostupné. Seznam těchto proměnných lze získat příkazem `Object.keys(window)`. Z tohoto seznamu lze pozorovat, že množství takto definovaných proměnných se pohybuje v řádu stovek a že žádná z těchto proměnných nezačíná podtržítkem. Podtržítko se tedy jeví jako vhodná předpona pro nové názvy identifikátorů, aby nedocházelo ke kolizím s dalšími identifikátorami. Samotný základ nového názvu identifikátoru bude tvořit hexadecimální číslo složené z osmi cifer. Takto zvolený identifikátor působí na první pohled zaměnitelně s hexadecimálně zapsaným číslem. Nový formát identifikátorů bude ve tvaru

$$_0xHHHHHHHH$$

kde $H$ představuje cifru hexadecimálního čísla. Takový formát názvu identifikátoru vyhovuje všem výše zmíněným podmínkám.

Při implementaci je použita knihovna escope. Postupně jsou procházeny všechny scope v kódu. U každého scope se zaznamená, které proměnné jsou zde deklarované. Zároveň v tomto scope budou dostupné i proměnné ze všech nadřazených scope. Při procházení proměnných je třeba dát pozor na speciální objekt s identifikátorem arguments, který je dostupný v každém těle funkce [29]. Tento objekt obsahuje všechny argumenty, které byly funkci voláním dodány. Jeho přejmenováním by došlo ke ztrátě těchto informací, a proto musí být název zachován. Pro všechny nalezené identifikátory, které byly v daném scope deklarované, se vygeneruje nový identifikátor. Následně je nový název identifikátoru aplikován i na veškeré reference daného identifikátoru. Pokud je v konfiguraci obfuskátoru zapnuto přejmenování globálních proměnných, dojde navíc k přejmenování těch identifikátorů, které nikde v kódu nejsou deklarovány.

## 5.5 Zastavování debuggeru ve smyčce

Při provádění dynamické analýzy v debuggeru, kdy je kód spuštěn, může být na libovolné místo vložen breakpoint, který na daném místě zastaví vykonávání kódu. Označit takové místo lze ale i přímo v kódu pomocí příkazu debugger. Tato transformace přidává do kódu smyčku, která v pravidelém intervalu aktivuje breakpoint příkazem debugger a v debuggeru tak vynutí zastavení vykonávání kódu. Přidáním této transformace bude dynamická analýza znepříjemněna, protože kód se bude opakovaně zastavovat a měnit aktuální kontext v debuggeru. Pokud je při deobfuskací prováděna dynamická analýza, bude nutné tento přidaný kus kódu lokalizovat a odstranit, aby mohla být provedena dynamická analýza. Debugger je obvykle možné nastavit tak, aby ignoroval breakpointy, ale v takovém případě nepůjdou využívat ani vlastní

breakpointy reverzního inženýra, které by mohly být napomocné při pochopení kódu. Pokud kód není spuštěn v debuggeru, nemá transformace žádný vliv na funkčnost.

### 5.5.1 Vlastnosti transformace

Potence se transformací nijak výrazně nezmění, pouze se výsledná délka kódu zvětší. Na druhou stranu přidaný kód je nezávislý na původním kódu, takže mezi sebou nemají žádnou spojitost a rozklíčování přidaného kódu nemusí trvat dlouho.

Odolnost kódu se také zvyšuje, protože musí být vynaložen čas, aby byl přidaný kód nalezen a odstraněn. To může být poměrně triviální úkol, ale v kombinaci s dalšími obfuskacemi se bude zesložiťovat.

Složitost kódu se zvyšuje, neboť po celou dobu běhu programu běží v pravidelném intervalu i smyčka, která pridává breakpoint. Zvyšují se tedy časové nároky na vykonání kódu v závislosti na zvoleném intervalu.

### 5.5.2 Implementace transformace

Implementace této transformace je přímočará. V kódu vznikne nová globální funkce, která jednorázově vykoná příkaz debugger. Tento příkaz bude spuštěn pomocí funkce eval, aby mohl být příkaz debugger zadán jako řetězec, což umožní zadat ho v takové formě, aby nebyl v kódu jednoduše nalezitelný. Funkce je následně prováděna ve smyčce pomocí funkce setInterval. Deklarace funkce a její spouštění je nakonec obaleno do funkce eval a celý kód vzniklý transformací je tak převeden na řetězec, díky čemuž ho bude následně možné obfuskovat dalšími transformacemi. Interval je nastaven na 500ms, aby výrazně nezvyšoval náročnost provádění programu, ale aby byl i tak dostatečně často spouštěn při dynamické analýze v debuggeru.

```
function addBreakpoint(){
  eval('debugger');
}
setInterval(addDebugger, 500);
```

Kód 5.3: Přidaný kód bez kombinace s dalšími obfuskacemi

## 5.6 Náhodné rozmístění deklarací proměnných

Pořadí deklarací proměnných může prozradit jejich důležitost v následujícím kódu. V jednom příkazu pro deklaraci je také možné provést deklaraci více proměnných najednou a tím může být ukázána spojitost mezi sousedícími proměnnými.

Jazyk JavaScript umožňuje pracovat s proměnnou, která v daném místě ještě nebyla deklarovaná. Podmínkou je, aby tato proměnná byla deklarovaná

na libovolném místě v příslušném scope. Tedy klidně i na místech, kde nikdy nemůže nastat jejich provedení, jako například za příkazem return. Pokud je navíc pracováno s kódem nestriktní verze ES, pak tyto deklarace nejsou vyžadovány vůbec a mohou se vypustit. Této vlastnosti je možné využít pro tuto transformaci.

První část transformace nalezne v kódu veškeré deklarace proměnných a rozdělí je na posloupnost deklarací o jedné proměnné. Následně jsou veškeré deklarace z kódu vyjmuty a uloženy do paměti. Pokud byla proměnná při deklaraci zároveň inicializovaná nějakou hodnotou, je výraz nahrazen přiřazující operací do dané proměnné, aby byla zachována původní funkčnost programu. V posledním kroku jsou deklarace pseudonáhodně rozmístěny v rámcí svého scope. Deklarace může být zanořena do bloku libovolné úrovně zanoření v rámci stejného scope. Při implementaci je tak deklarace s určitou pravděpodobností zanořena hlouběji do kódu. Tuto pravděpodobnost je možné nastavit v konfiguraci transformace. Zanoření je prováděno rekurzivně a deklarace se tak může objevit na libovolném místě v rámci scope.

```
'use strict';
function example(a){
  var b = 10, c = 20;

  if ( a < b ) {
    return b;
  }
  else if ( a < c ) {
    return c;
  }

  return a;
}
```

Kód 5.4: Příklad kódu před provedením transformace

```
'use strict';
function example(a){
  b = 10;
  c = 20;

  if ( a < b ) {
    return b;
  }
  else if ( a < c ) {
    return c;
    var b;
  }

  return a;
  var c;
```

```
}
```

Kód 5.5: Příklad stejného kódu po provedení transformace

### 5.6.1 Vlastnosti transformace

Umístění deklarace na náhodné místo v rámci scope může vizuálně roztrhnout jiné, dva kontextem související příkazy, kdy mezi ně bude vložena náhodná deklarace. Dochází tak ke zvýšení potence, kód se stává nepřehlednějším. Odolnost transformace je vysoká a kompletní deobfuskace na původní podobu nemusí být ve většině případů vůbec možná. Složitost kódu není transformací nijak ovlivněna. jedná se o jednosměrnou transformaci.

## 5.7 Převedení vlastností objektů na variantu využívající literál

Tato transformace sama o sobě nemá minifikační nebo zásadní obfuskační efekt. Její aplikování ale zvýší účinnost následujících obfuskačních transformací. Vlastnost objektu lze v JavaScriptu zapsat více způsoby. Jeden ze způsobů využívá pro název vlastnosti řetězec a toho bude následně možné využít při obfuskaci řetězců.

```
var object = {
  property: 5
};
// oba zapisy jsou mozne
console.log(object.property); // 5
console.log(object["property"]) // 5
```

Kód 5.6: Ukázka přístupu k vlastnosti objektu

Sjednocením těchto zápisů napříc celým kódem dojde také k tomu, že se vytratí původní zápis, který by mohl prozradit určité informace o daném objektu. Například u pole se k hodnotě přistupuje typicky číselným indexem přes hranaté závorky, ale u objektu se k vlastnosti přistupuje většinou přes tečkovou notaci.

## 5.8 Obfuskace čísel

Transformace pro obfuskaci čísel převádí jednotlivá čísla na ekvivalentní zápis v jiné číselné soustavě. Čísla jsou s různou pravděpodobností převáděna do binárního, oktalového a hexadecimálního zápisu.

Jiným způsobem tato transformace obfuskuje čísla 0 až 9, která jsou v kódu více frekventovaná. Jedná se typicky o řídící konstanty, indexy pro přístup do pole nebo členy aritmetických operací. Tato čísla jsou převáděna do

speciální výrazů, které jsou těžko srozumitelné a znepřehledňují kód. K tomu je v základu použito prázdné pole, které se přidáním unárního operátoru + přetypuje na celé číslo s hodnotou 0. Přidáním operátoru negace před tento výraz vznikne boolean hodnota true, před kterou když se opět přidá unární operátor +, vznikne tak celé číslo s hodnotou 1. Z těchto výrazů už je možné řětězením binárních operací sčítaní sestavit všechna celá čísla.

Tabulka 51: Příklad čísel převedených do výrazů

| **Původní výraz** | **Nový výraz** |
|---|---|
| 0 | +[] |
| 1 | +!+[] |
| 2 | +!+[]+ +!+[] |
| ... | ... |

### 5.8.1 Vlastnosti transformace

Potence kódu se aplikováním této transformace zvyšuje. Jednotlivá čísla jsou převedena do různých číselných soustav, které nejsou pro člověka přirozené. Zároveň jsou hexadecimální čísla vizuálně zaměnitelné s přejmenovanými identifikátory. Čísla, která jsou převedena na speciální výrazy, jsou těžko srozumitelná. V kombinaci s další obfuskační transformací, která převede binární a unární operace na volání funkce, se srozumitelnost těchto výrazů ještě více sníží.

Odolnost kódu se zvyšuje, protože musí dojít k rozklíčování jednotlivých výrazů. To v kombinaci s nahrazením binárních a unárních operací voláním funkce není úplně triviální. Pokud je navíc původní číslo součástí většího výrazu, stane se celý výraz nepřehlednější.

Složitost kódu se zvyšuje o konstantní počet operací, který nebude mít zásadní vliv na časové a paměťové nároky. Interpret navíc může takové výrazy předvyhodnotit a jejich vyhodnocení tak proběhne pouze jednorázově.

### 5.8.2 Implementace transformace

Při převodu čísel do zápisu jiných číselných soustav je dán největší prostor hexadecimálnímu zápisu čísla. K tomu je přistoupeno především z toho důvodu, že identifikátory jsou přejmenovány na formát obdobný zápisu hexadecimálního čísla a lidským okem může snadno dojít k záměně. Zároveň je hexadecimální zápis čísla vždy kratší nebo stejně dlouhý jako zápis v oktalové nebo binární soustavě. Je tím tak do jisté míry plněna i minifikační funkce. Naopak nejmenší prostor je dán binárnímu zápisu čísel, neboť je pro vyšší čísla výrazně delší než ostatní zápisy. Jeho funkcí je tak spíše dodat vícetvárnost zápisu celých čísel.

Tabulka 52: Rozdělení pravděpodobnosti převodu na jednotlivé číselné soustavu

| Číselná soustava | Šance |
|---|---|
| hexadecimální | 70% |
| oktalová | 20% |
| binární | 10% |

## 5.9 Náhodné přeuspořádání parametrů funkcí

Pořádí parametrů může o funkci prozradit důležité informace. Nejdříve jsou zpravidla voleny parametry, které jsou v kódu nejpodstatnější. K tomu dochází proto, aby čitelnost deklarace funkce byla srozumitelná. Pořádí parametrů může být také voleno tak, že sousední parametry jsou spolu spjaty v těle samotné funkce. Například prvním parametrem může být pole a sousedním parametrem jeho velikost. Pořádí parametrů tedy má nějaký řád a při deobfuskaci mohou poskytnout reverznímu inženýrovi informace o významu funkce. Je tedy smysluplné v rámci obfuskace provést náhodné přeuspořádání parametrů.

### 5.9.1 Vlastnosti transformace

Transformace má nízkou potenci, srozumitelnost kódu se nijak zásadně nemění. Odolnost transformace je vysoká, protože je transformací jednosměrnou. Transformace nijak nezvyšuje výslednou složitost kódu.

### 5.9.2 Implementace transformace

Při průchodu AST jsou postupně lokalizovány všechny definice funkcí, které jsou zároveň pojmenovány nějakým identifikátorem. U anonymních funkcí není statickou analýzou možné lokalizovat veškerá volání dané funkce, a proto jsou z této transformace vynechány. Pokud se v tělé funkce pracuje se speciálním objektem arguments, pak není možné provést přeuspořádání parametrů funkce a zároveň zaručit správné fungování programu. Takové funkce tedy budou také vynechány. Dále budou vynechány funkce, u kterých budou libovolná dvě různá volání obsahovat odlišný počet argumentů.

```
// u teto funkce neni mozne preusporadat parametry
function example(a, b) {
  var i = getIndex();
  var c = arguments[i];
}
```

Kód 5.7: Ukázka použití objektu arguments

Pro jednotlivé funkce jsou v AST vyhledána všechna jejich volání, u kterých dojde k přeuspořádání argumentů tak, aby odpovídalo novému uspořádání parametrů funkce. Pokud alespoň jedno volání obsahuje jako argument jiné volání funkce nebo aktualizaci proměnné, není možné provést přeuspořádání parametrů dané funkce, protože by nemuselo být zaručeno správné fungování programu.

```
function example(a, b) {
  ...
}

var c = 5;
example(c++, c); // a = 5, b = 6

// po provedeni preusporadani parametru
function example(b, a) {
  ...
}

var c = 5;
example(c, c++); // a = 5, b = 5
```

Kód 5.8: Ukázka odlišného chování při předání argumentů před a po přeuspořádání parametrů funkce

Bylo by možné dané výrazy vyjmout z volání funkce a deklarovat je před samotným voláním. Jako argumenty volání by potom byly dosazeny jen reference na nově vzniklé proměnné. Potom už by bylo možné provést přeuspořádání parametrů. Samotné pořadí deklarací by ale muselo zůstat zachováno a transformace tak postrádá smysl, protože původní pořadí argumentů bude prozrazeno.

## 5.10 Přesunutí operátorů do funkcí

Tato transformace přesunuje binární a unární operátory do vlastních funkcí. Ve výrazu, kde se nachází takový operátor, je operace nahrazena voláním odpovídající funkce. Je tím tak v daném výrazu utajen původní operátor.

```
// puvodni vyraz
var a = 2 + 3;

// po tranformaci
function sum(a, b){
  return a + b;
}
var a = sum(2, 3);
```

Kód 5.9: Příklad nahrazení výrazu voláním funkce

Tato transformace také upravuje přiřazující výrazy s operací tak, že z nich vznikne jednoduché přiřazení, kde na pravé straně výrazu je binární operace. Taková úprava umožní upravení vytvořených binárních operací na volání funkce stejně jako u původních binárních operací.

```
// puvodni vyraz
a += 5;
// vyraz po uprave
a = a + 5;
```

Kód 5.10: Příklad nahrazení přiřazujícího výrazu

### 5.10.1 Přehled upravených operátorů

Z unárních operatorů jsou vynechány operátory delete a typeof. Tyto operátory není možné nahradit voláním funkce s operandy jako argumenty, protože nemůže být zajištěna správná funkčnost programu.

Dále jsou vynechány logické operátory && a ||. Operandy těchto operátorů také není možné použít jako argumenty pro volání funkce vzhledem k vlastnostem těchto operací a opět by nemohla být zajištěna správná funkčnost programu. Příkladem může být situace, kdy je jako pravý operand operace && použita nedeklarovaná proměnná. V kódu může být spoléháno na to, že se pravý operand nevyhodnotí, pokud nebude pravdivý levý operand. Transformováním operátoru na volání funkce by ale pravý operand byl předán jako argument volání a vznikla by výjimka ReferenceError, neboť je pracováno s nedeklarovanou proměnnou.

- **Unární operátory**

  $-, +, !, \ , void$

- **Binární operátory**

  $==, ! =, ===, ! ==, <, <=, >, >=, <<, >>, >>>, +, -, *, /, \%, **, |, \hat{}, \&, in, instanceof$

- **Přiřazující operátory**

  $+=, -=, *=, /=, \%=, **=, <<=, >>=, >>>=, |=, \hat{}=, \&=$

### 5.10.2 Vlastnosti transformace

Po aplikování transformace se zvyšuje potence, protože dochází k utajení operátorů v původním výrazu. Kód se tak stává méně srozumitelným a je nutné dohledávat významy jednotlivých funkcí.

Odolnost kódu se zvyšuje, neboť reverzní inženýr musí rozklíčovat význam jednotlivých volání funkcí a připravit deobfuskátor tak, aby je mohl nahradit původními operacemi.

Složitost se může lišit dle implementace interpreteru, ale ve většině případů by měla zůstat nezměněná. Zvýšený počet volání funkcí by neměl mít vliv na rychlost programu, protože interpter takové jednoduché funkce může snadno optimalizovat pomocí inliningu [30].

### 5.10.3 Implementace transformace

Pro jednotlivé skupiny operátoru je v konfiguraci transformace možné nastavit, s jakou pravděpodobností dojde k nahrazení operace voláním funkce. V případě, kdy se při průchodu AST objeví nový operátor, je pro něj vytvořena nová funkce a uložena do paměti k pozdějšímu použití. Pro další výrazy s tímto operátorem je následně použita již vytvořená funkce. Jednotlivým funkcím je vygenerován unikátní identifikátor a jsou přidány do globálního scope, aby byly dostupné v celém kódu.

Při transformaci přiřazujících výrazů je třeba zkontrolovat, zda v levém operandu není použit výraz, který provádí aktualizaci proměnné nebo volání funkce. Příkladem může být přiřazení do prvku pole, kde je jako index použito volání funkce nebo proměnná s inkrementálním operátorem. V tomto případě by nebylo možné provést transformaci na binární operaci, protože by se daný výraz vyhodnotil dvakrát a program by nemusel mít zachovanou původní funkčnost.

## 5.11 Předefinování konzole

Do konzole lze v JavaScriptu vypisovat informace na výstup. V případě použití v prohlížeči je však ale tato funkčnost zbytečná. Na výstup by přes konzoli mohly být vypsány informace, které by mohly napomoct s porozuměním kódu. Účelem této transformace je tedy upravit kód tak, aby na výstup nebyly vypsány žádné informace. Zároveň dojde k předefinování samotného objektu konzole, aby reverzní inženýr nemohl vypsat do konzole vlastní výrazy. Samotná volání funkcí pro výpis do konzole zůstanou v kódu zachována. Bude tak navozen dojem, že konzole je v programu dostupná. Dojde ale k odstranění všech argumentů těchto volání tak, že se odstraní všechny argumenty, které neobsahují výraz, který by přiřazoval nějakou hodnotu do proměnné nebo volal nějakou funkci. Tyto argumenty musejí být zachovány z důvodu správného fungování kódu. Ostatní argumenty jsou odstraněny, protože by mohly odkrýt důležité informace vedoucí k snadnější deobfuskaci.

### 5.11.1 Vlastnosti transformace

Potence kódu se transformací nijak výrazně nezmění, pouze se předefinováním konzole výsledná délka kódu zvětší.

Odolnost kódu se zvyšuje, protože odebráním konzole může kód přijít o důležité informace a zároveň reverzní inženýr nemůže použít konzoli pro

vypsání vlastních výrazů. Ve výsledku tedy deobfuskace může trvat déle. Reverzní inženýr bude muset zjistit, že je konzole přepsána a následně lokalizovat a odstranit předefinování v kódu.

Složitost kódu se po aplikování transformace nijak nezmění, dojde pouze k přidání zanedbatelného počtu konstantních operací.

### 5.11.2 Implementace transformace

Nejdříve se přepíše objekt console prázdným objektem. Následně dojde k definování všech standardních funkcí původního objektu console na prázdnou funkci. Tím se předejde tomu, aby nedošlo k chybě při volání neznamené funkce.

```
var console = {};
var dummy = function(){
  /* funkce pro nahrazeni metod objektu console */
}
for (var method in standardConsoleMethods) {
  console[method] = dummy;
}
```

Kód 5.11: Předefinování konzole

## 5.12 Obfuskace řetězců

Obfuskace řetězců je jedna z nejpodstatnějších transformací při obfuskaci zdrojového kódu. Na rozdíl od většiny ostatních aplikovaných transformací, kde je především snaha snížit srozumitelnost kódu pro reverzního inženýra, je u této transformace navíc cílem ukrýt řetězce, ve kterých mohou být obsaženy důležité informace.

Kromě původních řětězců v kódu jsou obfuskovány také řětězce vzniklé z předchozích transformací. Zejména názvy vlastností objektů, respektive názvy volaných funkcí v objektu.

Tato obfuskace je složena ze čtyř různých transformací, kdy každá z nich přistupuje k obfuskaci řetězce jiným způsobem. Jejich kombinací pak vzniká výsledná silná obfuskace původního řetězce.

První transformace rozděluje řetězec v náhodném místě na dva kratší řětězce, které jsou binární operací + zřetězeny a po vyhodnocení tvoří původní řetězec. Z jednoho řetězce tak vzniknou v kódu dva nové samostatné řetězce, na které je možné aplikovat následující transformace. Na obě části může být aplikována jiná transformace a tím je přidána různorodost obfuskace původního řětězce.

Druhá transformace přesouvá řětězce ze svého původního místa do jednoho společného globálního pole pro všechny řetězce. V tomto poli jsou uloženy pouze unikátní řětězce a tato transformace tak může mít v určitých případech

i minifikační efekt, pokud se v kódu objevuje více stejných řětězců větší délky. Po uložení všech řetězců do pole je provedeno pseudonáhodné přeuspořádání prvků v poli. Přeuspořádání je provedeno z toho důvodu, aby z posloupnosti prvků v daném poli nebylo možné zjistit původní posloupnost řětezců podle výskytu v kódu. Následně je vytvořena globální funkce, která na základě vstupních parametrů vrátí řětězec z globálního pole. Původní řětězec je nahrazen voláním této funkce s argumenty, které vrátí původní řětezec.

Další transformace zakóduje řětězec pomocí Base64 kódování. Řetězec se tak stane pro člověka nečitelným a pro porozumění je nutné provést jeho dekódování. Původní řetězec bude nahrazen voláním funkce atob s argumentem zakódovaného řetězce, která provede zpětné dekódování. Tím bude zachována původní funkčnost programu. Délka výsledného řětězce se použitím kódování zvýší zhruba o třetinu.

Poslední transformace konvertuje řetězce do jejich unicode reprezentace. Výsledný řětězec je výrazně delší než původní řětězec, ale nejsou kladeny žádné časové nároky na zpětné dekódování, neboť interpret umí pracovat s unicode reprezentací řetězce.

### 5.12.1 Vlastnosti transformace

Transformací se výrazně zvyšuje potence, protože jednotlivé řetězce budou nesrozumitelné. Při kombinace všech transformací bude původní řetězec rozdělen na dva, následně budou oba řetězce přesunuty z původního místa pryč a ještě zakódovány pomocí Base64 a převedeny na unicode reprezentaci. Odolnost kombinace všech transformací je poměrně vysoká a dohledání původního řetězce není úplně triviální. Nejedná se ale už z principu o jednosměrnou transformaci. Zvyšuje se i výsledná složitost, protože je potřeba získat zpět přesunuté řetězce a případně provést zpět dékódování na původní řetězec z Base64 kódování.

### 5.12.2 Implementace transformace

U jednotlivých transformací této obfuskace je v konfiguraci možné nastavit šance na jejich aplikování. Výchozí šance jsou zvoleny tak, aby u jednotlivých řetězců byly použity různé kombinace transformací a aby tak k jejich deobfuskaci nestačil pouze jeden postup.

Implementace rozdělení řetězce, zakódávání pomocí Base64 a převedení na unicode reprezentaci jsou poměrně přímočaré operace a není nutné zde rozebírat jejich implementaci. Zajímavější je nahrazení řetězce voláním funkce, které vrátí daný řetězec. Aby nebyl argumentem funkce přímo číselný index do globálního pole s řetězci, je index rozšířen o multiplikativní a aditivní konstantu. Původní index je vynásoben číslem 2 umocněném s pseudonáhodně vygenerovanou mocninou od jedné do devíti. K výsledné hodnotě je přičtena pseudonáhodně vygenerována konstanta společná při přístupu

ke všem řetězcům. Tento výpočet je proveden při obfuskaci a nenachází se ve výsledném kódu. Výsledek je následně použit jako první argument vytvořeného volání funkce. Druhým argumentem je vygenerovaná mocnina. V těle funkce je následně proveden opačný výpočet, kdy je od prvního parametru odečtena aditivní konstanta, která se nachází v těle funkce, a následně je vydělen číslem 2 umocněným druhým argumentem. Informace o multiplikativní konstantě se tedy nachází v samotném volání funkce a aditivní konstanta je ukryta v těle funkce.

```
var literals = [ ... ];
function getLiteral(index, power){
  return literals[index - shift >> power]
  // aditivni konstanta shift je vygenerovana
      jednorazove pri obfuskaci vstupniho kodu
}
```

Kód 5.12: Vygenerovaná funkce pro přístup k řetězcům v poli

## 5.13 Obfuskace vybraných výrazů

Myšlenka této transformace byla částečně převzata z obfuskátoru JSFuck popsaného v sekci 3.2.4. Tato transformace nahrazuje vybrané výrazy jinými výrazy, které jsou méně srozumitelné. Nové výrazy jsou voleny tak, aby byly ekvivalentní nejen hodnotově, ale i typově. Tím bude zajištěna nezměněná funkčnost programu.

Potence se touto transformací zvyšuje, protože nové výrazy budou na rozdíl od původních pro člověka nesrozumitelné. Na druhou stranu jsou to výrazy, které nejsou nijak výrazně dlouhé a lze si je zapamatovat. Pokud se ale tyto výrazy objeví jako část většího výrazu, bude orientace v daném výrazu složitější.

Odolnost kódu se také zvyšuje, protože musí být rozklíčováno, co nově zvolené výrazy představují. Jejich deobfuskace je však potom už triviální operace, kdy se jednoduchým nahrazením v kódu dosadí původní výrazy.

Složitost kódu se zvyšuje o konstantní počet operací, který nebude mít zásadní vliv na časové a paměťové nároky. Interpret navíc může takové výrazy předvyhodnotit a jejich vyhodnocení tak proběhne pouze jednorázově.

Tabulka 53: Mapovací tabulka pro výrazy

| **Původní výraz** | **Nový výraz** |
|---|---|
| undefined | [][[]] |
| false | ![] |
| true | !![] |

Prvním výrazem je výraz undefined. U nového výrazu je využito vlastnosti JavaScriptu, kdy při přístupu k indexu v poli, který se v poli nenachází, je vrácen výraz undefined. Při sestavení nového výrazu je tedy v základu použito prázdné pole. Následně je třeba přistoupit k nějakému indexu, které se v poli nenachází. Aby výraz nebyl úplně čítelný jako přístup k hodnotě v poli, použije se jako index další prázdné pole. Takový index původní pole vyhodnotí jako prázdný řětězec, který se v původním poli nenachází, a je vrácen výraz undefined.

Další výrazy jsou boolean hodnoty true a false. U těchto výrazů je v základu také využito prázdné pole. Takové pole je v JavaScriptu vyhodnoceno jako true, ale není typově ekvivalentní. Samotné prázdné pole tedy ještě nelze využít. Přidáním negace před prázdné pole bude vynuceno přetypování na boolean a vznikne výraz false. Dalším přidáním negace pak vznikne výraz true.

## 5.14 Náhodné přeuspořádání definic funkcí

Podobně jako u pořadí parametrů funkcí, i pořadí samotných definic funkcí může usnadnit orientaci v kódu a prozradit souvislosti v kódu. Zároveň tato transformace zesílí následující transformaci, kdy bude docházet ke slučování funkcí.

### 5.14.1 Vlastnosti transformace

Transformace má nízkou potenci, srozumitelnost kódu se nijak zásadně nemění. Odolnost transformace je vysoká, protože se jedná o transformaci jednosměrnou. Složitost kódu zůstává nezměněná.

### 5.14.2 Implementace transformace

Definice je možné bezpečně přeuspořádat vždy pouze na stejné úrovni zanoření, aby nedošlo k situaci, kdy by se nějaká funkce stala nedostupnou. Při průchodu AST jsou lokalizovány všechny bloky včetně samotného globálního scope. V každém takovém bloku jsou vyhledány všechny definice funkcí. Tyto definice jsou z bloku vyjmuty a uloženy do paměti, kde jsou následně pseudonáhodně přeuspořádány. V posledním kroku jsou funkce vráceny do bloku ve vygenerovaném pořadí.

## 5.15 Slučování funkcí

Slučování funkci je transformace, která vybere dvě funkce ze stejného scope a provede jejich sloučení. Takovou transformaci není možné provést se všemi funkcemi a jsou zde určitá omezení.

Funkce musí mít identifikátor a zároveň v dané funkci nesmí být použit objekt arguments. Funkce také nemůže být použita jinak než při přímém volání. Pokud by například funkce byla přiřazena do jiné proměnné, nebylo by možné statickou analýzou provést odpovídající upravení kódu tak, aby byly podchyceny všechny volání dané funkce.

```
function alpha(a, b) {
  return a + b - 3;
}

function beta(c, d) {
  console.log('example');
  return c * d;
}

alpha(5, 10);
beta(7, 7);
```

Kód 5.13: Ukázka dvou funkcí a jejich volání před provedením transformace

Při sloučení dvou funkcí dochází k vytvoření nové funkce, která má počet parametrů odpovídající počtu parametrů té z funkcí, která má více parametrů. Sloučení funkcí s různým počtem parametrů je možné, protože jazyk JavaScript umožňuje volat funkci s méně nebo i více argumenty než definice očekává. K těmto parametrům je přidán navíc další parametr, který bude sloužit jako řídící parametr pro rozhodnutí, jaká původní funkce se měla vykonat. Tento řídící parametr bude v nově vzniklé funkci vždy na první pozici. Ostatní parametry slouží jako parametry původních funkcí.

Tělo této funkce je pak rozděleno podmínkou, která testuje řídící parametr a na jeho základě rozhodne, která větev, resp. původní funkce, bude provedena. Těla původních funkcí se tak tedy přesunou do větví této podmínky. V souvislosti s tím proběhne přejmenování identifikátorů z původního těla funkce tak, aby odpovídalo novým parametrům nové funkce.

Následně je pro každou větev přiřazen literál, který bude předán jako hodnota řídícího parametru. Všechna volání původních dvou funkcí jsou nahrazena voláním sloučené funkce. Do těchto volání je přidán jako první argument zvolený literál a tím bude zajištěno, že bude provedena odpovídající větev podmínky. Hodnota tohoto literálu je pro každou větev vybrána vytknutím existujícího literálu z dané větve a jeho nahrazením řídícím parametrem. Tím proběhne zanesení řídícího parametru do větví podmínky a nebude sloužit pouze pro rozhodnutí, která větev se má provést. To zvyšuje odolnost transformace, protože kód na první pohled nepůsobí pouze jako dvě sloučené funkce. Pokud není k dispozici žádný literál pro vytknutí, použije se výchozí hodnota 0 pro první větev a hodnota 1 pro druhou větev.

Původní dvě funkce jsou z kódu odstraněny a je přidána nově vytvořená funkce. Pokud je v konfiguraci zakázáno přejmenování globální identifikátorů, neproběhne sloučení funkcí v globálním scope.

```
function gamma(a, b, c) {
  if ( a === 3 ) {
    return b + c - a;
  }
  else {
    console.log(a);
    return b * c;
  }
}

gamma(3, 5, 10);
gamma('example', 7, 7);
```

Kód 5.14: Ukázka vzniklé funkce a její volání po provedení transformace

### 5.15.1 Vlastnosti transformace

Transformace zvyšuje potenci, protože dvě původní funkce, které mohly být srozumitelné, jsou sloučeny do jedné. Navíc pokud v se v tělech původních funkcí nacházel literál, je vytknut do volání funkce. Kód se tak stává méně srozumitelným. Transformace je deobfuskovatelná, ale reverzní inženýr nejdříve musí rozklíčovat, že proběhlo sloučení funkcí a že byl vytknut literál do volání funkce. Není nijak znatelně zvýšena složitost výsledného kódu, jedná se pouze o přidání podmínky do těla funkce.

## 5.16 Přesunutí části kódu do funkce eval

Tato transformace vyjme příkazy z vybraného bloku kódu a nahradí je funkcí eval, které budou argumentem předány vyjmuté příkazy ve formě řetězce.

Není ale možné zvolit libovolný blok, protože některé příkazy mohou mít při vykonání ve funkci eval odlišné chování. Blok, který obsahuje příkazy continue, break a return není možné použit, protože dané příkazy nebudou znát vnější kontext a program nebude fungovat správně. Obdobně není možné zvolit blok, ve kterém jsou deklarované proměnné a funkce, protože mimo funkci eval nebudou dostupné.

Tato transformace by měla být aplikována jako jedna z posledních. Její dřívější použití by mohlo zapříčinit znemožnění použití jiných obfuskačních transformací, protože kód už bude převeden na řetězec a nebude s ním možné nadále manipulovat. Do funkce eval tedy bude argumentem předán již obfuskovaný kód jako řetězec. Tato transformace sama o sobě ale nemá velký význam, protože kód je v řetězci v čitelné podobě a jednoduchým vyjmutím z funkce eval a nahrazením příkazů na daném místě by byla provedena kompletní deobfuskace této transformace. Užitečnost této transformace se projeví až v kombinaci s obfuskací řetězců. Na daný řetězec s kódem se aplikuje ob-

fuskace řětězců a tím bude zajištěno skrytí původního kódu. Tato transformace tedy umožňuje využít obfuskaci řetězců na obfuskaci samotného kódu.

### 5.16.1 Vlastnosti transformace

Potence této transformace je vysoká, protože dochází k zakrytí samotného kódu do takového tvaru, že není čítelný.

Odolnost transformace je z principu závislá na odolnosti obfuskace řětězců. Samotná transformace bez kombinace s obfuskací řetězců by měla mizivou odolnost.

Složitost kódu výrazně vzroste kvůli časovým nárokům, protože kód vykonaný ve funkci eval je výrazně pomalejší. To je zapříčeněno tím, že pro každé vykonání volání funkce eval je nutné znovu spustit interpret pro daný kód. Zároveň interpret vynechá některé optimalizace, které by bez použití funkce eval provedl, protože nemůže nahlédnout dovnitř funkce eval, aby možné optimalizace provedl [18].

Je tedy třeba s touto transformací pracovat s vědomím toho, že přemírou jejího použití by mohlo dojít k výraznému zpomalení běhu programu. Ve výchozím nastavení je nastavena nížší šance na aplikování transformace, aby výskyt nebyl až příliš častý. Dále jsou vynechány transformace takových bloků, které se nacházejí právě uvnitř smyčky.

## 5.17 Zašifrování části kódu a vynucení integrity kódu

Tato transformace naplňuje dvě různé obfuskační funkce. První funkcí je obdobná jako u většiny předchozích transformací, tedy snaží se zabránit porozumění kódu reverzním inženýrem. K tomu přistupuje transformace tak, že šifruje vybraný blok kódu. Druhou funkcí je, že vynucuje zachování integrity kódu. To znamená, že při modifikaci kódu reverzním inženýrem může dojít k znefunkčnění programu.

V prvním kroku je vybrán vhodný blok k zašifrování. Omezení pro výběr bloku jsou nastavena stejně jako při výběru bloku u předchozí transformace, kdy se vybíral vhodný blok pro přesunutí do funkce eval. Vynechají se tedy bloky, které obsahují deklarace proměnných nebo funkcí, příkaz continue, příkaz break a příkaz return. Dále je pro každý blok pseudonáhodně vybraná jedna funkce, která je v daném bloku dostupná a je pojmenována identifikátorem. Tato funkce bude sloužit jako klíč pro zašifrování a dešifrování odpovídajícího bloku. Vybraná funkce se od tohoto okamžiku nemůže žádnou další transformací změnit a musí zůstat v dané podobě. Může být ale použita jako klíč i pro další blok. Tato transformace tedy musí být provedena jako úplně poslední, protože další transformace by mohly změnit tento klíč.

Výpočet klíče na základě zadané funkce je postaven na vlastnosti JavaScriptu, že každá funkce může zavoláním funkce toString vrátit svojí přesnou podobu jako řetězec. Tento řetězec je následně rozdělen na pole, které obsahuje jednotlivé znaky řetězce. Pole je poté redukováno na jedno číslo, které se stane výsledným klíčem. Redukce pole začíná hodnotou 0 a postupným procházením všech prvků pole je aktualizováno výsledné číslo vzorcem znázorněnén v kódu 5.15.

```
return result ^ num + index;
// kde result je prubezna vysledna hodnota klice
//     num je ciselna reprezentace aktualniho znaku
//     index je index aktualniho znaku v poli
```

Kód 5.15: Znázornění funkčnosti generování klíče

V kódu níže je znázorněna kompletní funkčnost generování klíče. V implementaci obfuskátoru je použita složitější verze, která umožňuje obfuskaci názvů volaných funkcí objektů. Zde je v kódu 5.16 pro názornost použita srozumitelná verze.

```
var key = func.toString().split('').reduce(function(
    result, num, index){
  return result ^ num.charCodeAt(0) + index;
}, 0)
```

Kód 5.16: Znázornění funkčnosti generování klíče

Podobně jako funkce se i samotný blok převede na pole složené z jednotlivých znaků kódu. Číselná podoba každého znaku tohoto pole je následně zašifrována operací XOR s vypočteným klíčem. Do výsledného kódu se poté vloží výraz, který analogicky pomocí klíče dešifruje blok na původní podobu a provede jeho spuštění. Tento výraz je opět v zjednodušené podobě zobrazen v ukázce kóde níže.

```
try {
  eval(
    // promenna chars je pole se zasifrovanymi znaky
       puvodniho bloku
    chars.map(function(value) {
      return String.fromCharCode(value ^ key)
    }).join('')
  )
}
catch (e){}
```

Kód 5.17: Dešifrování bloku v kódu

Šifrování pomocí operace XOR je symetrické, a proto lze použít stejný proces pro šifrování i dešifrování bloku. Vzhledem k tomu, že kód bloku je transformován na textovou podobu, je nutné tento kód spustit ve funkci eval.

Funkce eval je vložena do try-catch bloku. To souvisí s druhou vlastností této transformace, kdy je vynucena integrace kódu. Tím, že z textové podoby zvolené funkce je vytvořen klíč, by libovolná úprava, byť jen přidáním mezery, znamenala znefunkčnění zašifrovaného bloku. Funkce eval by se tak snažila vykonat kód, který nedává smysl, protože byl dešifrován nesprávným klíčem. V konzoli by se ale v takovém případě objevila chyba SyntaxError a reverznímu útočníkovi by tak prozradila, kde se nachází problém. Odchycením těchto vyjímek v try-catch bloku je zajištěno, že se v konzoli tato chyba neobjeví. Pokud tedy dojde k úpravě funkce a tím změně klíče, reverzní inženýr se nedozví, že část kódu v zašifrovaném bloku se neprovedla.

### 5.17.1 Vlastnosti transformace

Transformace výrazně zvyšuje potenci, protože dochází k zašifrování zvolených bloků kódu a stávají se tak pro člověka nesrozumitelné.

Odolnost transformace je také vysoká, ale z principu se nejedná o transformaci jednosměrnou. Kód pro správnou funkčnost vyžaduje nezměněnou podobu kódu, tedy i formátování celého souboru. Pokud jsou tedy například odstraněny bílé znaky, nemůže reverzní inženýr ani naformátovát kód do přehlednější podoby, kdy je doplněno odřádkování a odsazení kódu. Slabinou operace XOR je, že dvě aplikace této operace se stejným operandem na pravé straně se vzájemně vyruší a je vrácena původní hodnota. Pokud by se tedy do kódu funkce doplnil například pouze sudý počet mezer a odřádkování, kód by nadále fungoval správně. Výpočet klíče tak nemůže být pouhým zřetězením operace XOR na všechny znaky řetězcové reprezentace funkce. Proto je při výpočtu klíče ve vzorci použit ještě index znaku, tedy jeho pozice v řetězcové reprezentaci, a tím je tato slabina potlačena.

Nevýhodou transformace je, že se zvyšuje složitost výsledného kódu. Ta se zvyšuje už jen použitím funkce eval, která má popsané nevýhody u předchozí transformace při přesunu částí kódu do funkce eval.

Klíč není možné vypočítat jednorázově pro všechny funkce a následně jen přistupovat k vypočtené hodnotě. Před každým zašifrovaným blokem musí být proveden výpočet klíče a tím zajistit integritu kódu. Při dynamické analýze kódu totiž může reverzní inženýr s kódem manipulovat a pokud by byly klíče předvypočítané, kód by i po modifikaci nadále fungoval správně.

Další nevýhodou transformace může být, že po aplikování by s kódem už nemělo být nadále manipulováno, protože už jen jednoduchou změnou odsazení kódu by mohl přestat fungovat.

## 5.18 Kombinace a posloupnost transformací

Pozitivní vliv na výslednou obfuskaci nemají jen samotné transformace, ale také jejich kombinování. Zásadní vliv má také posloupnost transformací. Určité

transformace mohou pomoci zesílit jiné následující transformace, které by samostatně jako takové nemusely být příliš silné. Právě kombinace a posloupnost transformací tak má velký vliv na výslednou potenci a odolnost kódu.

### 5.18.1 Použitá posloupnost transformací

- **Odstranění komentářů**

  Odstranění komentářů provede parser, který je nezanese do vygenerovaného AST.

- **Optimalizace zdrojového kódu**

  Pokud se dle konfigurace má provést optimalizace vstupního kódu, je provedena před samotnými obfuskačními transformacemi.

- **Přejmenování identifikátorů**

  Přejmenování identifikátorů proběhne jako první vlastní transformace. Transformace tak vygenerováním nových identifikátorů zároveň zanese informaci obfuskátoru o použitých identifikátorech, aby následné transformace nevygenerovaly již existující identifikátor.

- **Zastavování debuggeru ve smyčce**

  Tato transformace nemá žádný pozitivní efekt na následující transformace, a proto je použita jako jedna z prvních.

- **Předefinování konzole**

- **Náhodné rozmístění deklarací proměnných**

- **Převedení vlastností objektů na variantu využívající literál**

  Tato transformace vytváří v kódu nové řetězce, a proto by měla být před obfuskací řetězců.

- **Obfuskace čísel**

  Obfuskace čísel do kódu přidává nové binární a unární operace, a proto se v posloupnosti nachází před přesunem operátorů do funkcí.

- **Náhodné přeuspořádání parametrů funkcí**

- **Přesunutí operátorů do funkcí**

- **Obfuskace řetězců**

- **Obfuskace vybraných výrazů**

- **Náhodné přeuspořádání definic funkcí**

- **Slučování funkcí**

  Slučování funkcí slučuje dvě sousední funkce, a proto je aplikována až po přeuspořádání definic funkcí. Tím je zajištěno, že nebudou sloučeny dvě funkce, které jsou si blízké i v původním kódu.

  Transformace také proběhne až po přeuspořádání parametrů funkcí, protože po sloučení funkcí už nemusí být možné parametry sloučené funkce přeuspořádat.

- **Přesunutí části kódu do funkce eval**

  Tato transformace je provedena jako jedna z posledních, protože by mohla zavinit nemožnost použití následných transformací. Na výsledný řetězec obsahující kód, který bude spuštěn funkcí eval, je speciálně znovu spuštěna obfuskace řetězců.

- **Zašifrování části kódu a vynucení integrity kódu**

- Transformace pro zašifrování části kódu musí být podobně jako předchozí transformace ze stejného aplikována jako jedna z posledních. Zároveň tato transformace zajišťuje integritu kódu a do vybraných funkcí už nemůže být vůbec zasáhnuto.

  Tato transformace ale nebude narušena následující transformací, která odstraňuje bílé a redundantní znaky. Funkce, jejichž podoba je použita pro výpočet klíče pro zašifrování bloku, jsou předgenerovány a klíč je tak počítán ze výsledné podoby funkce.

- **Odstranění bílých znaků a redundantních znaků v kódu**

  Escodegen při generování výsledného kódu dle nastavení formátuje kód bez bílých znaků. Také vynechá redundantní znaky v kódu.

Kapitola 6

# Testování

Tato kapitola je věnovaná testování realizovaného obfuskátoru. Cílem testování je ověřit, zda implementované transformace pracují správně a porovnat vlastnosti obfuskovaného kódu s původním kódem. Výsledky testování se mohou odlišovat v závislosti na zvolené konfiguraci obfuskátoru. Pro testování byly použity výchozí konfigurační hodnoty v obfuskátoru.

## 6.1 Empirické testování transformací

Jedním z hlavních cílů obfuskačních transformací je upravit kód do takové podoby, ve které nebude člověkem srozumitelný. Takový cíl ale nelze jednoznačně měřit a testovat, protože každý člověk má odlišné zkušenosti s kódem.

Také je možné empiricky testovat, zda výstup obfuskovaného kódu odpovídá výstupu původního kódu.

Některé transformace dle jejich povahy lze otestovat pouze experimentálně. Například transformace, která v debuggeru vynucí zastavení provádění kódu příkazem debugger, nebo transformace, která v kódu znemožní použití konzole. Správné chování těchto transformací bylo ověřeno v debuggeru Chrome DevTools.

## 6.2 Sada testovacích skriptů přiložená u obfuskátoru

Při vývoji byla vytvořena sada testovacích skriptů, která testovala implementované transformace. Tyto testovací skripty jsou přiloženy k obfuskátoru. Vzhledem k tomu, že tuto sadu připravoval autor obfuskátoru, z principu může nastat situace, kdy pokud autor opomněl některé okrajové případy v implementaci, nebudou takové případy pokryty ani v testovacích skriptech.

## 6.3 Testování obfuskátoru na veřejně dostupných knihovnách

Testování obfuskátoru je možné provést také tak, že bude obfuskátor použit na veřejné dostupné knihovny. Při testování bude podstatné porovnávat dobu běhu, pokud to povaha knihovny umožní, a také velikost výsledného souboru. Vzhledem k tomu, že výsledný kód z obfuskátoru ovlivňuje mnoho náhodných faktorů, může se pro různá spuštění obfuskátoru na stejný soubor odlišovat doba běhu i výsledná velikost souboru. Určení sledovaných hodnot obfuskovaného kódu tedy bude provedeno tak, že obfuskace na zdrojový kód bude provedena opakovaně a jako výsledná hodnota se použije jejich průměr. Pokud knihovna obsahuje i vlastní sadu testů, je možné otestovat, zda obfuskátor zachoval původní funkčnost knihovny.

### 6.3.1 Otestování výsledné velikosti souboru

Pro otestování výsledné velikosti byly použity veřejně dostupné knihovny, které jsou často používány ve skriptech pro prohlížeče.

- **jQuery**

  První testovánou knihovnou je jQuery [31]. Velikost této knihovny je v době psaní práce 281 KB. Po aplikování obfuskátoru na knihovnu byla výsledná velikost v průměru 475 KB. Výsledná velikost obfuskované knihovny je tak zhruba dvojnásobná. Předchozí bakalářská práce [12] vygenerovala soubor o průměrné velikosti 590 KB.

  Knihovna jQuery také obsahuje svojí sadu testů, která testuje funkčnost knihovny. Původní knihovna tak byla nahrazena obfuskovanou verzí a sada testů byla spuštěna. V této sadě testů se nacházelo několik testů, které neprošly už při neobfuskované verze knihovny jQuery. To bylo způsobeno převážně z důvodu prostředí, ve kterém byly testy spuštěny. Při spuštění sady testů na obfuskovanou verzi knihovny nevyhověly stejné testy, ostatní prošly v pořádku.

- **Chart.js**

  Druhou testovanou knihovnou je Chart.js [32].s Velikost této knihovny je v době psaní práce 594 KB. Po aplikování obfuskátoru na knihovnu byla výsledná velikost v průměru 980 KB. Tato knihovna umožňuje na webové stránce v prohlížeči vykreslovat grafy na základě vstupních dat. Bylo tak možné provést vizuální ověření funkčnosti obfuskované verze knihovny. Předchozí bakalářská práce [12] vygenerovala soubor o průměrné velikosti $1,7$ MB.

Z výsledků lze pozorovat, že výsledná velikost souboru je u sledované bakalářské práce znatelně vyšší. To je primárně způsobeno tím, že tato práce ob-

sahuje transformací, která provádí inlining funkcí. Tato transformace zásadně zvyšuje délku kódu.

### 6.3.2 Otestování doby běhu

Pro otestování doby běhu byla zvolena knihovna [33], která provádí výpočet MD5 otisku. Nemělo by například smysl testovat dobu běhu knihovny jQuery, protože sama o sobě neprovádí výpočetně náročně operace. Při testování bylo postupně ve smyčce vypočítáno 100, 1000 a 10000 MD5 otisků. Naměřené hodnoty jsou zobrazeny v tabulce 61.

Tabulka 61: Porovnání doby výpočtu původní a obfuskované knihovny

| **Počet výpočtů** | **100** | **1000** | **10000** |
|---:|:---:|:---:|:---:|
| Původní knihovna v ms | 8 | 50 | 90 |
| Obfuskovaná knihovna touto prací v ms | 50 | 460 | 3800 |
| Obfuskovaná knihovna s BP [12] v ms | 2300 | $> 60000$ | neměřeno |

Hodnoty zjištěné měřením nelze brát nijak závazně, protože doba běhu je ovlivěna i strojem, na kterém byl výpočet proveden, a také samotným interpretem jazyku JavaScript. Lze ale pozorovat, že doba výpočtu po obfuskaci se zvyšuje rychleji než původní knihovna. Pro 10000 výpočtů je již rozdíl mezi původní a obfuskovanou knihovnou znatelný.

Nicméně pro opodstatnění takového zvýšení času je nutné poznamenat, že na knihovnu byl použit obfuskátor s výchozí konfigurací, který primárně cílí na snížení čitelnosti kódu. Rychlost takové konfigurace tedy nebyla primární prioritou. Z toho vyplývá, že výchozí konfigurace obfuskátoru nemusí být vhodná pro každý vstup.

V původní knihovně může interpret provést potřebné optimalizace pro rychlejší výpočet. Po provedení obfuskace se v kódu mohou nacházet funkce eval, které interpret nemusí umět optimalizovat. To je také důvodem, proč existující bakalářská práce dosahuje tak vysoké doby výpočtu. Při generování obfuskovaného kódu vygenerovala uvnitř určité smyčky funkci eval, která vykonává nějaký příkaz, který se na daném místě původně nacházel. V rámci této diplomové práce bylo ošetřeno, aby se funkce eval zapříčiněním nějaké transformace nevyskytla přímo uvnitř smyčky.

V prohlížeči se typicky provádějí jednorázové operace, které svojí povahou nejsou výpočetně náročně, a je tak smysluplné zvýšit potenci i za cenu vyšší výpočetní složitosti.

### 6.3.3 Testování deobfuskace

Zásadní částí testování je také ověření, do jaké míry implementované transformace odolávají veřejně dostupným deobfuskátorům. Cílená deobfuskace by

z principu byla mnohem efektivnější, ale testování na veřejně dostupných deobfuskátorech také přinese určitou informaci o odolnosti obfuskovaného kódu.

Pro testování byly zvoleny deobfuskátory Online JavaScript Beautifier dostupný z [34], de4js dostupný z [35], JStillery dostupný z [36] a JSNice dostupný z [28].

Všechny testované deobfuskátory kód minimálně částečně znefunkčnily, protože provádějí jeho formátování, tedy přidávají odřádkování a odsazení příkazů. To je neslučitelné s provedenou transformací, kdy je podoba funkce použita jako klíč pro zašifrování částí kódu. Pro následující popis deobfuskátorů tak tato transformace nebude brána v potaz, aby mohly být zkoumány jiné vlastnosti deobfuskátorů.

Ze zkoumaných deobfuskátorů nedokázal žádný navrátit řetězce na jejich původní místo před obfuskací. Tyto řetězce zůstaly také zakódovány pomocí Base64, ale převedení řetězců na unicode reprezentaci příliš odolné nebylo.

Některé deobfuskátory dokázaly rozklíčovat výraz nebo čísla zaobfuskované pomocí transformací popsaných v 5.8 a 5.13. Většina z těchto výrazů ale zůstala zachována, protože obsahují operátory a ty jsou transformací 5.10 přesunuty do vlastních funkcí. Výrazy se tak stanou pro deobfuskátory špatně čitelnými.

Transformace pro přejmenování identifikátorů, přeuspořádání parametrů funkcí, přeuspořádání definic funkcí a přeuspořádání deklarací proměnných odolaly ze své podstaty všem deobfuskátorům. Žádný z testovaných deobfuskátorů také nedokázal identifikovat předefinování konzole a vloženou smyčku pro debugger.

Z těchto výsledků tedy vyplývá, že automatizovaná deobfuskace není příliš efektivní. To je způsobeno také tím, že obfuskátor napsaný v rámci této práce není deobfuskátorům znám. Pokud by se obfuskátor stal frekventovaně používaným, deobfuskátory by se mohly adaptovat na obsažené transformace a jejich účinnost by se tak zvýšila. Ukazuje se tedy, že cílená deobfuskace dosahuje mnohem lepších výsledků než deobfuskace automatizovaná.

Kapitola 7

# Navazání na praci

Obfuskátor je koncipován modulárně a nabízí se tak jeho rozšíření v navazujících pracích.

Další zajímavá transformace popsána v [37] by mohla využívat WebGL, které přidává podporu 3D grafiky ve webových prohlížečích. Díky tomu by mohl být vlastním algoritmem vygenerován obrázek, z kterého by se následně na předdefinovaných místech posbíraly informace o jeho pixelech. Tyto informace by se následně použily jako klíč pro zašifrování nějaké části kódu. Reverzní inženýr by potom musel mít k dispozici debugger s podporou WebGL, aby se dostal k původnímu kódu.

# Závěr

V práci proběhlo seznámení se s problematikou minifikace a obfuskace jazyka JavaScript. Byly analyzovány volně dostupné knihovny pro obfuskaci a také předchozí bakalářská práce na toto téma. Proběhla analýza a návrh implementace, na základě které byl obfuskátor vytvořen.

Byl navrhnut efektivní postup minifikace a obfuskace, který byl do obfuskátoru implementován. U jednotlivých tranformací byla v práci popsána jejich implementace a jejich vlastnosti. Obfuskátor byl otestován na běžně používaných knihovnách a porovnán s předchozí bakalářskou prací.

Bylo provedeno testování volně dostupných deobfuskátorů na obfuskovaný kód, kdy bylo zjištěno, že navrhnutý obfuskátor úspěšně odolává automatizovaným deobfuskátorům. Obfuskátor je implementován modulárně a je tak v budoucnosti jednoduše rozšiřitelný o nové transformace.

# Literatura

[1] Komunita WebPlatform: Code minification [online]. [cit. 2019-04-02]. Dostupné z: https://webplatform.github.io/docs/concepts/programming/javascript/minification/

[2] Collberg, C.; Thomborson, C.; Low, D.: A Taxonomy of Obfuscating Transformations. Technická zpráva, Department of Computer Science, The University of Auckland, 1997. Dostupné z: https://researchspace.auckland.ac.nz/bitstream/handle/2292/3491/TR148.pdf

[3] Stuttard, D.: Dynamic analysis of JavaScript [online]. [cit. 2019-04-02]. Dostupné z: https://portswigger.net/blog/dynamic-analysis-of-javascript

[4] Osbourn, T.: Abstract Syntax Trees [online]. [cit. 2019-04-02]. Dostupné z: https://tosbourn.com/abstract-syntax-trees/

[5] Ilegbodu, B.: History of ECMAScript [online]. [cit. 2019-04-05]. Dostupné z: http://www.benmvp.com/learning-es6-history-of-ecmascript/

[6] Zaytsev, J.: ECMAScript 6 compatibility table [online]. [cit. 2019-04-05]. Dostupné z: https://kangax.github.io/compat-table/es6/

[7] Mozilla: JavaScript [online]. [cit. 2019-04-05]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/JavaScript

[8] StackOverflow: Developer Survey Results 2019 [online]. [cit. 2019-04-05]. Dostupné z: https://insights.stackoverflow.com/survey/2019

[9] Jahoda, B.: Scope v JavaScriptu [online]. [cit. 2019-04-05]. Dostupné z: http://jecas.cz/scope

[10] Mozilla: Strict mode [online]. [cit. 2019-04-05]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode

[11] Sengstacke, P.: JavaScript Transpilers: What They Are & Why We Need Them [online]. [cit. 2019-04-05]. Dostupné z: https://scotch.io/tutorials/javascript-transpilers-what-they-are-why-we-need-them

[12] Hanák, S.: *Minifikace a obfuskace JavaScriptu*. Bakalářská práce, Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2018.

[13] uxebu: Confusion [online]. [cit. 2019-04-10]. Dostupné z: https://github.com/uxebu/confusion

[14] Kachalov, T.: JavaScript Obfuscator Tool [online]. [cit. 2019-04-10]. Dostupné z: https://obfuscator.io

[15] Dan's Tools: JavaScript obfuscator [online]. [cit. 2019-04-10]. Dostupné z: https://www.cleancss.com/javascript-obfuscate/index.php

[16] Komunita Esolangs: JSFuck [online]. [cit. 2019-04-10]. Dostupné z: https://esolangs.org/wiki/JSFuck

[17] Microsoft: TypeScript Documentation [online]. [cit. 2019-04-16]. Dostupné z: https://www.typescriptlang.org/docs/home.html

[18] Mozilla: eval() [online]. [cit. 2019-04-16]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/eval

[19] Mozilla: with [online]. [cit. 2019-04-16]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/with

[20] Mozilla: Function.prototype.name [online]. [cit. 2019-04-16]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/name

[21] Dave Herman a další přispěvatelé: The ESTree Spec [online]. [cit. 2019-04-18]. Dostupné z: https://github.com/estree/estree

[22] JS Foundation: Introducing Espree, an Esprima alternative [online]. [cit. 2019-04-18]. Dostupné z: https://eslint.org/blog/2014/12/espree-esprima

[23] Yusuke Suzuki a další přispěvatelé: Escodegen [online]. [cit. 2019-04-18]. Dostupné z: https://github.com/estools/escodegen

[24] Yusuke Suzuki a další přispěvatelé: Estraverse [online]. [cit. 2019-04-18]. Dostupné z: https://github.com/estools/estraverse

[25] Yusuke Suzuki a další přispěvatelé: Esmangle [online]. [cit. 2019-04-18]. Dostupné z: https://github.com/estools/esmangle

[26] Yusuke Suzuki a další přispěvatelé: Escope [online]. [cit. 2019-04-18]. Dostupné z: https://github.com/estools/escope

[27] Ingvar Stepanyan a další přispěvatelé: Estemplate [online]. [cit. 2019-04-18]. Dostupné z: https://github.com/estools/escope

[28] Vechev, M.: JSNice - Statistical renaming, type interference and deobfuscation [online]. [cit. 2019-05-02]. Dostupné z: http://jsnice.org

[29] Mozilla: arguments [online]. [cit. 2019-04-22]. Dostupné z: https://developer.mozilla.org/cs/docs/Web/JavaScript/Reference/Functions/arguments

[30] Zeunert, M.: What's the performance cost of a function call? [online]. [cit. 2019-04-24]. Dostupné z: https://www.codereadability.com/performance-cost-javascript-function-call-and-foreach/

[31] The jQuery Foundation: jQuery JavaScript Library [online]. [cit. 2019-04-30]. Dostupné z: https://github.com/jquery/jquery

[32] Kolektiv autorů: Chart.js - Simple yet flexible JavaScript charting for designers & developers [online]. [cit. 2019-04-30]. Dostupné z: https://www.chartjs.org

[33] Tschan, S.: JavaScript MD5 [online]. [cit. 2019-04-30]. Dostupné z: https://github.com/blueimp/JavaScript-MD5

[34] Lielmanis, E.; Newman, L.: Online JavaScript Beautifier [online]. [cit. 2019-05-02]. Dostupné z: https://beautifier.io

[35] Thien, T. T.: JavaScript Deobfuscator and Unpacker [online]. [cit. 2019-05-02]. Dostupné z: https://lelinhtinh.github.io/de4js/

[36] Paola, S. D.: Advanced JavaScript Deobfuscation via Partial Evaluation [online]. [cit. 2019-05-02]. Dostupné z: https://github.com/mindedsecurity/JStillery

[37] Fernández, J. M.: JavaScript AntiDebugging Tricks [online]. [cit. 2019-05-03]. Dostupné z: https://x-c3ll.github.io/posts/javascript-antidebugging/

Příloha A

# Seznam použitých zkratek

**ES** ECMAScript

**JS** JavaScript

**AST** Abstraktní syntaktický strom

**XOR** Exkluzivní disjunkce

Příloha **B**

# Obsah přiloženého CD

```
readme.txt ........................ stručný popis obsahu CD
src
  impl ............................ zdrojové kódy implementace
  thesis .......................... zdrojová forma práce ve formátu LaTeX
text .............................. text práce
  thesis.pdf ...................... text práce ve formátu PDF
```