FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Minifikace a obfuskace JavaScriptu
Student:	Samuel Hanák
Vedoucí:	Ing. Radomír Polách
Studijní program:	Informatika
Studijní obor:	Teoretická informatika
Katedra:	Katedra teoretické informatiky
Platnost zadání:	Do konce letního semestru 2018/19

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) 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 22. ledna 2018

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalářská práce

Minifikace a obfuskace JavaScriptu

Samuel Hanák

Katedra teoretické informatiky
Vedoucí práce: Ing. Radomír Polách

13. května 2018

Poděkování

Svůj vřelý dík směřuji své rodině a přítelkyni, jejichž podpora mi umožnila tuto práci dokončit.

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 13. května 2018 .....

Odkaz na tuto práci

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

Abstrakt

Práce se zaměřuje na problematiku minifikace a obfuskace, tj. automatizovaného znepřehlednění zdrojového kódu, za účelem ochrany autorského práva autora zdrojového kódu, při zachování původní funkcionality. Předmětem zkoumání je programovací jazyk JavaScript. Cílem je za použití estools vytvořit v NodeJS obfuskátor, který využívá známé obfuskační a minifikační techniky a aplikuje je k obfuskování JavaScriptu. Důraz je přitom kladen na znemožnění úplného rozklíčování pomocí automatického deobfuskátoru. Použité obfuskační techniky jsou v práci zdokumentovány a vysvětleny.

Klíčová slova Obfuskace, minifikace, abstraktní syntaktický strom, statická analýza, JavaScript

Abstract

The work's main focus is the issue of minification and obfuscation, i.e. automated source code obscuration in order to protect the copyright of the author of the source code while maintaining its original functionality. The subject of the exploration is programming language JavaScript. The goal is to create an obfuscator in NodeJS that uses known obfuscation and minification techniques and applies them to obfuscate JavaScript. Emphasis is placed on making it impossible to completely disassemble generated code using an automatic deobfuscator. Used obfuscation techniques are documented and explained in the work.

Keywords Obfuscation, minification, abstract syntax tree, static analysis, JavaScript

Obsah

Seznam obrázků

Seznam tabulek

Úvod

JavaScript je oblíbený programovací jazyk používaný na moderních webech. Využívá se ke spouštění funkcionality webových aplikací na straně klienta. Jedná se o interpretovaný jazyk, tudíž je potřeba originální zdrojový kód posílat uživateli (respektive prohlížeči uživatele). Uživatel tak má zdrojový kód plně k dispozici, což otevírá cestu plagiátorům. Je tedy přirozenou snahou programátorů v JavaScriptu svůj zdrojový kód znepřehlednit tak, aby pokusy o jeho rozklíčování a případné zneužití byly co možná nejobtížnější. Programy, které toto znepřehlednění automaticky zajišťují, se nazývají obfuskátory. Další potřebou vývojářů je zmenšit velikost JavaScriptových zdrojových kódů, aby přenos souborů mezi serverem a klientem trval co nejkratší dobu. Automatickému zmenšování zdrojových kódů se říká minifikace.

K dnešnímu datu existuje už několik funkčních a používaných obfuskátorů a minifikátorů. Existuje ale také určitý tlak na to, aby bylo nezávislých a dostupných obfuskátorů co nejvíce – obfuskační techniky a přístup k jejich implementaci se díky tomu stanou různorodější a pestřejší, což vede k výraznému ztížení spolehlivé automatické deobfuskace. Kromě osobního zájmu o téma formálních jazyků a překladačů, které je s obfuskací úzce spjato, je tedy mou motivací především rozšíření pomyslného „trhu s obfuskátory“.

Mou bakalářskou práci lze využít jako zdroj některých obfuskačních technik a inspiraci k jejich možné implementaci. Zároveň lze praktickou část práce přímo využít k obfuskaci JavaScriptového kódu, tzn. nasadit ji přímo do praxe.

Kapitola 1

Cíl práce

Cílem kapitoly Vymezení základních pojmů je seznámit se se stěžejními pojmy, které se týkají problematiky minifikace a obfuskace a odhalit motivace pro minifikaci a obfuskaci, přičemž se budeme zaměřovat především na tuto motivaci ve vztahu k jazyku JavaScript. Cílem kapitoly Analýza a rešerše je prozkoumat proces obfuskace a minifikace, seznámit se s knihovnami z rodiny estools (především espree a escodegen) a odhalit dostupná řešení dané problematiky, ať už jde o nasazené projekty nebo výzkum a odborné práce na toto téma. Cílem kapitoly Realizace je navrhnout a implementovat funkční program v NodeJS, který bude za pomoci knihoven estools provádět minifikační a obfuskační transformace na JavaScriptový kód, přičemž hlavní důraz je kladen na ztížení deminifikace a deobfuskace.

Kapitola 2

Vymezení základních pojmů

2.1 Minifikace

Minifikace je transformace, kterou provádíme nad nějakým zdrojovým kódem, abychom redukovali jeho datovou velikost. V případě JavaScriptu se jedná o obzvlášť oblíbenou praktiku, protože JavaScript je interpretovaný (především) webový programovací jazyk a zdrojové kódy jsou tak doručovány uživateli webových stránek v nezkompilované ani jinak nezměněné podobě. Standardní zdrojové kódy ale obsahují mnoho konstruktů, které jsou sice užitečné pro jejich interpretaci z pohledu člověka, nicméně jsou bezvýznamné z hlediska jejich spuštění. Bez minifikace tak musí uživatel naší aplikace stahovat větší soubory, což (kromě jiného) znamená, že musí na zobrazení stránky čekat delší dobu [1]. Mezi nejběžnější minifikační techniky patří odstranění komentářů a bílých znaků (mezer) a přejmenování identifikátorů.

2.2 Obfuskace

Obfuskace je další transformace prováděná nad zdrojovými kódy. Jak uvádí [1]: „There might be a concern that someone could read the program and learn our techniques, and worse, compromise our security by learning the secrets that are embedded in the code.“ Právě na tyto požadavky nabízí odpověď obfuskace. Svými transformacemi se snaží co nejvíce zastřít význam zdrojového kódu, aby byl co možná nejméně rozklíčovatelný třetí stranou (ale zároveň plnil stále stejnou funkci). Ukazuje se, že prakticky všechny minifikační techniky mají zároveň i obfuskační efekt (odstranění komentářů zároveň eliminuje podstatné informace obsažené v kódu, identifikátory, které minifikací přejmenujeme, zase nesly informaci o tom, jakou funkci plní ta která proměnná atd.). Dá se tedy říct, že minifikační transformace jsou podmnožinou těch obfuskačních. Existují důvodné námitky [2] odpůrců obfuskace, kteří tvrdí, že obfuskace je zbytečná praktika, protože důsledný reverzní inženýr vždy odhalí funkcionalitu kódu, bez ohledu na míru obfuskace. Pravdou samozřejmě je, že

obfuskace není ani zdaleka rovna kupř. šifrování, pokud jde o zakrytí dat před útočníkem. Avšak jak uvádí [3], „identifier renaming significantly decreases the efficiency of attacks, at least doubling the time needed to complete a successful attack (even in the worstcase scenario, i.e., against the best attacker). In addition, obfuscation reduces the gap between novice and skilled attackers, making the latter less efficient, and makes systems that are easier to attack in clear more similar to those that are intrinsically harder to break.“ Tyto závěry nám dávají naději, že obfuskace nemusí být zbytečnou transformací a je možné najít jí uplatnění v reálných projektech. Kromě ochrany duševního vlastnictví je dalším obvyklým využitím obfuskace zakrytí činnosti škodlivého kódu. Tvůrci malware používají obfuskaci, aby (především) antivirové programy nebyly schopny rozpoznat jejich záměr a ideálně ani fakt, že kód je škodlivý. [4] uvádí, že většina škodlivých webových stránek používá obfuskovaný kód. Z tohoto pohledu může zkoumání obfuskace přinést ovoce na poli vylepšování deobfuskačních metod a tedy i nástrojů pro odhalování škodlivého softwaru.

Transformace kódu (tedy i obfuskace a minifikace) samozřejmě nesou riziko pozměnění chování kódu – v podstatě zanesení chyb. Mělo by být jednou z hlavních snah každého obfuskátoru, aby se tomuto nešvaru vyhnul, jak jen to bude možné. Jak říká [1], zanesení nových chyb do kódu procesem obfuskace značně zvyšuje čas potřebný pro debugování aplikace, a dokonce jenom samotný fakt, že chyba vůbec mohla vzniknout obfuskací, vývojáře nutně zpomaluje. I my budeme na tuto problematiku dbát a budeme se při obfuskaci snažit předjímat co nejširší škálu možných scénářů, abychom chování kódu zachovali co nejpodobnější originálu.

Přestože minifikační transformace jsou svého druhu podmnožinou těch obfuskačních, platí, že uplatnění některých obfuskačních technik je výrazně na úkor minifikace kódu. My samozřejmě nebudeme v našich transformacích minifikaci opomíjet, nicméně vzhledem k cílům bakalářské práce pro nás nebude nejdůležitějším faktorem při určování kvality našich transformací.

2.3 Deobfuskace

Jak uvádí [5], deobfuskace je proces zjednodušení pogramu, který odstraní obfuskující kód a vytvoří program, který je jednodušší, ale funkčně ekvivalentní. Nelze očekávat, že výsledkem deobfuskačního procesu bude originální zdrojový kód – některé transformace prováděné při procesu obfuskace jsou jednosměrné a tudíž některé původní informace nelze získat zpět. Cílem deobfuskace je tedy v zásadě zjednodušení práce pro reverzního inženýra. Využívá k tomu dva druhy analýzy:

- Statická analýza shromažďuje veškeré informace o zdrojovém kódu pouze zkoumáním samotného kódu a tyto informace následně využívá k deobfuskačním transfromacím.

- Dynamická analýza provádí spouštění kódu (nebo jeho částí) s různými vstupy a na základě pozorování výstupů vyvozuje závěry o chování kódu (a následně je využívá k deobfuskačním transformacím).

Jak uvádí [6], dynamická analýza je problematická z toho důvodu, že nemusí být schopna pokrýt chování zdrojového kódu v celé jeho úplnosti (to platí obzvlášť pro složité a/nebo rozsáhlé zdrojové kódy) – naproti tomu statická analýza ano. To znamená, že provádění transformací jen na základě dynamické analýzy může být v určitých situacích destruktivní vzhledem k chování kódu. [6] ale dále upozorňuje, že současné implementace statické analýzy nejsou schopny důsledně prozkoumat výrazy jako volání funkce eval nebo dynamicky generovaný zdrojový kód. Smysluplným dílčím cílem pro autory obfuskátorů tedy je zajistit, aby obfuskovaný zdrojový kód nebylo možno deobfuskovat statickou analýzou a nutit tak reverzní inženýry používat metody dynamické analýzy, které jsou (teoreticky) méně důvěryhodné.

Kapitola 3

Analýza a rešerše

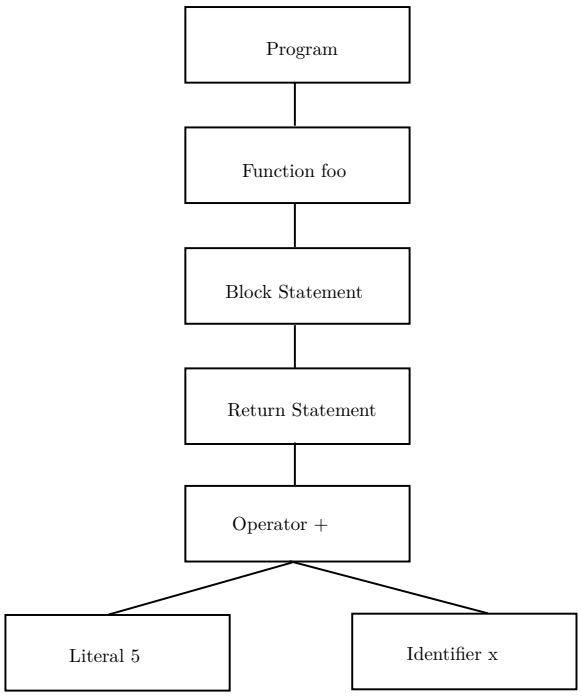
3.1 Obecný proces minifikace a obfuskace

Obfuskace a minifikace – obzvlášť na úrovni vyššího programovacího jazyka, kterým je i JavaScript – se nedá (až na primitivní transformace) příliš úspěšně provádět bez zavedení určité míry abstrakce. Tuto abstrakci typicky poskytují tzv. abstraktní syntaktické stromy (zkráceně AST). AST je konstruováno ze zdrojového kódu pomocí tzv. parsování (čtení řetězce se zdrojovým kódem a získávání v něm obsažených informací). Jak uvádí [7], AST reprezentují parsovaný řetězec ve strukturované formě, přičemž vynechávají veškeré informace, které mohou sice být nezbytné pro parsování, ale jsou nepotřebné pro analýzu a transformace kódu. Grafické znázornění jednoduchého AST pro kód `function foo() { return 5 + x; }` vidíme na obrázku 3.1.

AST je užitečný pro zajištění nezbytné abstrakce nad analyzovaným zdrojovým kódem, ale stejně tak je vhodně využitelný i pro opačný proces, tedy generování kódu (proto je v nějaké formě obvykle využíván v překladačích). Proces minifikace a/nebo obfuskace tedy typicky vypadá takto:

1. Parsování zdrojového kódu a vytvoření abstraktního syntaktického stromu reprezentujícího tento kód.
2. Provádění požadovaných transformací (v našem případě minifikačních a obfuskačních) nad AST.
3. Vygenerování nového zdrojového kódu z námi upraveného AST. Tento kód je použitelný namísto původního a je již obfuskovaný a minifikovaný.

Struktura AST může být navržena různým způsobem, takže odlišné parsery mohou vytvořit odlišně strukturované AST (byť typicky jsou výstupy různých parserů pro stejný zdrojový kód ve stejném programovacím jazyku velmi podobné). Transformace, které nad daným AST provádíme, tedy musí být vytvořené na konkrétní formu AST – jinými slovy, než začneme implementovat transformace, je potřeba zvolit parser.



Obrázek 3.1: Grafické znázornění jednoduchého AST.

3.2 Technologie pro práci s AST JavaScriptu

Jak říká [8], všeobecně rozšířená, zavedená a dobře zdokumentovaná struktura JavaScriptových kódů je SpiderMonkey AST. Dva nejpopulárnější parsery, které produkují AST v tomto formátu jsou Acorn a Esprima. My budeme pracovat s Esprimou, resp. s její upravenou verzí, kterou je Espree.

Jak uvádí [9], Espree je parser, který se oddělil od Esprimy verze 1.2.2. Espree vznikla proto, že autoři projektu ESlint [1] nebyli spokojení s podporou Esprimy pro novější verze JavaScriptu a po několika pokusech vyřešit tento problém jinak se rozhodli vytvořit vlastní parser – který ovšem staví na základech Esprimy. Uzly ve SpiderMonkey AST jsou objekty obsahující celou řadu podstatných i (pro nás) méně podstatých informací – např. typ uzlu, jeho umístění ve zdrojovém kódu atd. Valná většina properties těchto objektů je specifická pro určité typy uzlu a u ostatních se nevyskytuje. Např. uzel typu BinaryExpression obsahuje kromě standardních property ještě levý operand (property left), pravý operand (property right) a operátor (property

[1] ESLint je nástroj, který vyhledává podobnosti s určitými vzory v JavaScriptovém kódu, především aby pomohl vývojářům odhalovat chyby a vytvářet konzistentní kód [9].

operator). Podobným způsobem jsou konstruovány i další uzly v AST.

Pro práci se SpiderMonkey AST byla vytvořena celá řada nástrojů. Významným projektem jsou estools, které zahrnují několik užitečných knihoven. Ty knihovny z rodiny estools, které budeme používat v našem obfuskátoru, si nyní stručně rozebereme.

3.2.1 Knihovna estraverse

Estraverse bude v této práci nejčetněji používáná knihovna. Poskytuje nám dvě podstatné funkce:

- Traverse, která nám umožní procházet AST odshora dolů, navštívit postupně každý uzel stromu a provádět nějaké operace při jeho návštěvě i při jeho opuštění. O každém uzlu se kromě standardních informací, kterými ho naplnil parser, dozvíme ještě informaci o tom, kdo je ve stromové hierarchii jeho rodičem. Procházení stromu lze v libovolném uzlu zastavit pomocí volání `this.break()`, případně zakázat prohledávání podstromu určitého uzlu pomocí `this.skip()`. Typické použití knihovny (podobně jak ho uvádí i oficiální dokumentace knihovny [10]), které uvádíme především proto, že se v našem obfuskátoru jedná o velice častou operaci, lze vidět na výpisu kódu 3.1.
- Replace, která se chová téměř identicky jako traverse, jen navíc umožňuje vyměnit právě navštívený uzel za jiný.

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

Výpis kódu 3.1: Použití knihovny estraverse

3.2.2 Knihovna escope

Knihovna escope, jak už naznačuje její název, nám zjednodušuje práci se scopem v JavaScriptu. Escope má relativně přímočaré použití. Jak radí dokumentace knihovny [11], funkce `escope.analyze(ast)`, která přijímá jako argument AST (vygenerovaný pomocí parseru espree), zpracuje strukturu zdrojového kódu a vrací objekt, pomocí něhož můžeme analyzovat scope libovolného uzlu

v AST. Využití této knihovny v praxi uvidíme v kapitole 4. Obzvlášť potřebná bude pochopitelně při implementaci přejmenování identifikátorů v sekci 4.2.

3.2.3 Knihovny esmangle, escodegen a esvalid

Knihovna esmangle nám jednoduchým voláním funkcí mangle a optimize umožní minifikovat kód [12]. Escodegen je zodpovědná za generování JavaScriptového kódu z AST (tuto knihovnu tedy využijeme až po provedení potřebných transformací) [13]. Esvalid nám pomůže pomocí série předpřipravených testů zjistit, zda je uživatelem vkládaný kód validní a případně identifikovat syntaktické chyby v kódu [14].

3.3 Měření účinnosti obfuskačních technik

Měření účinnosti minifikačních technik je relativně snadné – minifikace je tím účinnější, čím více se jí daří zmenšit datovou velikost minifikovaného souboru bez poškození chování kódu. Pro škálování účinnosti obfuskačních technik použijeme metriku navrženou v [15]. Každá transformace má tři měřítka, přičemž snaha o optimalizaci jednoho může způsobit poškození druhého.

Prvním měřítkem je potence. Potence udává, jak obtížné pro člověka je pochopit význam zdrojového kódu, tzn. jak moc je kód „nečitelný“. Cílem obfuskátoru je přirozeně maximalizovat toto měřítko. [15] uvádí, že vysokou míru potence mají např. transformace, které zvětšují počet parametrů funkcí, zvyšují počet zanoření bloků podmínek a cyklů, zvětšují velikost kódu nebo zvyšují počet používaných funkcí (zde už vidíme první jednoznačnou protichůdnou tendenci ve vztahu k minifikaci).

Druhým měřítkem je odolnost (v originálu resilience). Odolnost určuje schopnost obfuskační transformace překonat automatickou deobfuskaci. Je zřejmé, že některé techniky, přestože výrazně zhorší čitelnost kódu (např. odstranění bílých znaků), jsou snadno zdolatelné jednoduchým deobfuskátorem – tyto techniky mají nízkou odolnost. Obfuskátor by se měl samozřejmě snažit maximalizovat i tento parametr transformací. Nejodolnější transformace jsou jednosměrné – tzn. ty, jejichž deobfuskace je zcela nemožná (např. odstranění komentářů). My budeme obzvlášť vnímaví směrem k měření odolnosti, protože je úzce spjato se zadáním naší práce – co největší znemožnění deobfuskace našich transformací.

Třetím a posledním měřítkem jsou náklady. Tento parametr nám říká, jak moc byla naše transformace vykoupena časovou náročností programu nebo rozsahem zdrojového kódu. Snahou obfuskátoru je náklady minimalizovat. Některé transformace (např. přejmenovávání proměnných nebo odstranění komentářů), přestože mají nějakou kladnou míru potence a odolnosti, s sebou nenesou žádné negativní dopady, pokud jde o náklady. Takové transformace jsou v repertoáru obfuskačních technik však spíše výjimkou a obvykle se setkáme s protichůdnou tendencí potence a odolnosti oproti nákladům. Proto

je typicky potřeba najít zdravou hranici kompromisu, aby byla obfuskace dostatečně silná a zároveň nezpůsobovala neúnosné zpomalení aplikace nebo prodloužení kódu.

3.4 Souhrn používaných obfuskačních technik

Obfuskačních technik je nepřeberné množství. Jak si ukážeme v kapitole 4, některé z transformací mají zvláštní vztah k JavaScriptu, protože jsou JavaScriptovými pravidly v některém směru posíleny nebo naopak oslabeny. Dostatečný orientační (byť ne zcela úplný) vhled do problematiky nám poskytne [15], která obfuskační techniky zkoumá a dělí do několika skupin:

- Obfuskace layoutu – zahrnuje transformace přejmenování identifikátorů, odstranění komentářů a změnu formátování (tyto transformace mají při správném použití zároveň i minifikační účinky).
- Obfuskace výpočtů řídicí logiky – zahrnuje vkládání mrtvého nebo irelevantního kódu, rozšiřování podmínek pro cykly o nadbytečné výrazy, přidávání redundantních výrazů a paralelizace výpočtu.
- Agregace prvků řídicí logiky – obsahuje inlining a outlining funkcí a metod, klonování funkcí, slučování funkcí a rozbalení cyklů (známe-li dopředu jejich hranice).
- Přeuspořádání prvků řídicí logiky – zahrnuje přeuspořádání příkazů a přeuspořádání výrazů.
- Kódování dat – zahrnuje změnu kódování dat, povýšení skalárů na objekty, změnu životnosti (scopu) proměnných, rozdělení informace nesené jednou proměnnou do několika proměnných.
- Agregace datových struktur – obsahuje vkládání falešných tříd, rozdělování nebo slučování tříd, rozdělování nebo slučování polí.
- Změna uspořádání dat – zahrnuje přeuspořádání metod, funkcí, třídních proměnných nebo prvků v poli.
- Preventivní transformace zamířené proti deobfuskaci – obsahuje přidání falešných datových závislostí, vytváření nových predikátů a jejich využití k vedlejším účinkům v kódu.

3.5 Používané obfuskátory

V současnosti jsou nejoblíbenějšími nástroji pro transformace zdrojových kódů v JavaScriptu minifikátory. Patří mezi ně UglifyJS, YUI Compressor a Google Closure Compiler [16]. V kapitole 2 jsme uvedli, že minifikační transformace

plní zároveň i obfuskační účel. Přestože to je pravda, ani jeden z výše uvedených nástrojů se nezaměřuje na obfuskaci v pravém slova smyslu, místo toho je obfuskace vedlejším účinkem jejich transformací. Mezi nástroje plnící primárně obfuskační účel patří Jscrambler [17], JavaScript Obfuscator [18] od Tiaga Serafima a Javascript Obfuscator [19] od CuteSoft Components Inc.

3.6 Dostupnost proměnných v JavaScriptu

Dostupnost proměnných v JavaScriptu je pro nás podstatná problematika, protože je potřeba vědět, které transformace (při nichž pracujeme s proměnnými nebo funkcemi) mohou mít destruktivní efekt pokud jde o chování kódu. V našem obfuskátoru budeme pracovat s JavaScriptem verze ES5. Informace o přístupu k proměnným v této verzi nalezneme v následujících dvou subsekcích.

3.6.1 Scope v JavaScriptu

Problematiku scopu v JavaScriptu dobře popisuje [20]. Scope funguje v JavaScriptu podobně jako v jiných jazycích, ale s jistými odlišnostmi. Zatímco ve většině jazyků je základním stavebním kamenem scopu blok, v JavaScriptu je to funkce. Je-li proměnná deklarovaná v nějaké funkci, tvoří tato funkce scope proměnné, a proměnná je všude v této funkci dostupná. Pokud je proměnná deklarovaná v nějakém scopu, je dostupná i ve všech vnořených scopech (výjimkou je, pokud je ve vnořeném scopu deklarovaná jiná proměnná stejného jména – v tom případě se vždy použije ta proměnná, jejíž deklarace je v hierarchii scopů nejblíže). V každém scopu jsou všechny deklarace proměnných interně přemístěné na začátek (tzv. hoisting proměnných), takže jsou potom dostupné všude v daném scopu.

Existuje globální scope, k němuž jsou přiřazeny proměnné deklarované mimo jakoukoliv funkci. Zároveň přiřazení hodnoty do nedeklarované proměnné (v jakémkoliv scopu) automaticky umístí tuto proměnnou do globálního scopu. V globálním scopu výraz this odkazuje na tzv. globální objekt, který obsahuje veškeré globální proměnné jako své property. Prohlížeče navíc obsahují globální proměnnou window, která je referencí na globální objekt. V NodeJS roli této reference na globální objekt plní proměnná global.

Pokud je funkce volaná z jiného scopu, než v němž byla vytvořena, má stále přístup k proměnným deklarovaným v původním rodičovském scopu. Tuto situaci dobře prezentuje výpis kódu 3.2.

3.6.2 Context v JavaScriptu

Context v JavaScriptu je relativně složité a obsáhlé téma, které v podstatě hledá odpověď na otázku, na jaký objekt poskytuje referenci výraz this. Uve-

```
function foo() {
   var x = 10;
   return function() { return x; }
}

var fun = foo();
console.log(fun()); //vypíše 10
```

Výpis kódu 3.2: Demonstrace vazby funkce na scope, v němž byla vytvořena

deme pro nás podstatné, ale zdaleka ne veškeré informace týkající se tohoto tématu, načerpané z [21].

Mimo tělo jakékoliv funkce se this odkazuje na globální objekt. V jednoduchém volání funkce se chování this liší na základě toho, zda pracujeme ve strict mode. Pokud není strict mode zapnutý, bude se v jednoduchém volání this odkazovat na globální objekt (pokud nebyl voláním funkce nastaven na jinou hodnotu). Pokud je strict mode zapnutý, bude se this odkazovat na stejnou hodnotu, na kterou byl nastaven při vstupu do volané funkce (není-li tato hodnota nastavena, bude this rovno hodnotě undefined). Pomocí metod call a apply lze změnit hodnotu this při volání libovolné funkce.

Ve voláních metod objektu X se this odkazuje na objekt X. V konstruktorech se this odkazuje na právě konstruovaný objekt.

Kapitola 4

Realizace

4.1 Odstranění komentářů a bílých znaků

Odstraňování komentářů je velice účinná minifikační technika (odstraňuje z kódu často velké množství textu, který je z pohledu funkcionality programu zbytečný) i obfuskační technika (komentáře poskytují cenné a mnohdy nenahraditelné informace, které pomáhají rychlému pochopení kódu – to je ostatně jejich účel). Zároveň je odstranění komentářů jednosměrná transformace, tedy její deobfuskace je prakticky nemožná (tj. má vysokou odolnost). Přestože je tato úprava v kódu do takové míry významná, její provedení je triviální a lze realizovat konečným automatem. V našem případě jednoduše budeme komentáře při vytváření AST ignorovat, takže se po generování kódu na závěr už nikde neobjeví.

Odstranění bílých znaků nám kód opět trochu zmenší co do datové velikosti. Kód bez odsazení je samozřejmě velmi obtížně čitelný, nicméně proces doplnění mezer ke znovuobnovení odsazení není nijak obtížný (odstranění bílých znaků má vysokou potenci, ale nízkou odolnost). Samozřejmě nelze deobfuskací navrátit například původní odřádkování, které naznačovalo myšlenkové oddělení dvou částí kódu – takové drobnosti ovšem nenesou podstatně velké množství informace. O tuto tranformaci se za nás postará knihovna escodegen.

4.2 Přejmenovávání identifikátorů

Přejmenovávání identifikátorů je běžná a velmi efektivní minifikační technika. Zkrácením názvů proměnných na nezbytné minimum (mnohdy si vystačíme s identifikátory o délce jednoho znaku) jsme schopni dosáhnout velmi výrazného zmenšení zdrojového souboru. Ještě účinnější je tato technika na poli obfuskace. Identifikátory v běžném zdrojovém kódu nesou mnohdy důležitou informaci o tom, jaký účel konkrétní proměnná či funkce plní. Zahlazením těchto stop kód značně znepřehledníme.

Vzhledem k tomu, že se jedná o jednosměrnou transformaci (identifikátory se jednoduše přejmenují a jejich původní názvy pak už nejsou nikde v kódu dohledatelné), je automatická deobfuskace přejmenování identifikátorů prakticky nemožná. Deobfuskátory mohou změněné identifikátory znovu přejmenovat po svém a dodat jim tak alespoň nějakou informační hodnotu – např. páté pole v programu lze pojmenovat jako „array5“, třetí parametr funkce jako „param3“ apod. Je to však jen velmi slabá náhrada za původní soběstačné názvy. V JavaScriptu nemůžeme na rozdíl od mnohých jiných programovacích jazyků přejmenovat veškeré identifikátory. Pojďme se podívat, proč tomu tak je a v jakých případech tuto tranfsormaci smíme provést.

4.2.1 Funkce eval

Prvním přímočarým problémem je samozřejmě funkce eval. Eval je funkce umožňující vykonat JavaScriptový kód, který je předán v podobě řetězce jako argument této funkci. [22] uvádí, že eval do zdrojového kódu může injektovat deklaraci proměnné, zatímco se na tuto proměnnou objevují reference na jiném místě, než v argumentu evalu. Stejně problematická je i opačná situace – deklarace mimo eval a reference uvnitř. Ukázku takového kódu vidíme ve výpisu kódu 4.1.

```
function foo() {
   var bar = 0;
   eval("bar = 1");
   return bar;
}
```

Výpis kódu 4.1: Problematický kód používající funkci eval

Jelikož bez dynamické analýzy (a v mnohých případech reálně ani s ní) nejsme schopni argument funkce eval rozklíčovat (nebo alespoň ne vždy), je zřejmé, že přejmenovávání identifikátorů nemůžeme bezpečně provést prakticky nikde. Toto chování funkce eval nám bude působit podobně fatální problémy i v mnoha dalších transformacích, které chceme se zdrojovým kódem provádět. Zároveň platí, že používání funkce eval se důrazně nedoporučuje, jednak proto, že musí sám volat JavaScriptový interpreter na vyvolání předaného kódu (což vede ke zpomalení programu), ale především proto, že jeho používání může snadno vést k problémům s bezpečností. Pokud by se totiž útočníkovi podařilo podstrčit svůj kód do argumentu evalu (například pomocí napadení neošetřeného vstupu), celý jeho škodlivý kód se bez problémů vykoná, což je samozřejmě velké bezpečnostní riziko. Neklademe proto na uživatele našeho obfuskátoru přehnané nároky, když požadujeme, aby tuto funkci ve zdrojovém kódu nepoužíval, a na oplátku nabízíme daleko účinnější minifikaci a obfuskaci.

4.2.2 Příkaz with

Na podobnou překážku, jakou pro nás představovala funkce eval, narazíme ještě u příkazu with. Jak uvádí [23], with umožňuje programátorovi odkazovat se ve vyhrazeném bloku kódu na properties vybraného objektu jako na standardní proměnné, bez nutnosti reference prostřednictvím objektu. Ve výpisu kódu 4.2 vidíme chování příkazu with.

```
var obj = { prop1: "str" };

function f() {
   var v = 10;
   var prop1 = 20;
   with(obj) {
      console.log(prop1); //vypíše str
      console.log(v);     //vypíše 10
   }
}

f();
```

Výpis kódu 4.2: Použití příkazu with

Výčet properties jednotlivých objektů není v JavaScriptu nikdy stanoven, lze ho určit jen pomocí dynamické analýzy kódu. Důsledkem toho nejsme schopni vždy určit, zda jsou proměnné použité uvnitř bloku příkazu with proměnnými zděděnými z rodičovského scopu, nebo zda to jsou property příslušného objektu. Vzhledem k tomu, že i tento JavaScriptový konstrukt se nedoporučuje používat a je obecně vnímán jako zastaralý, budeme i zde pro správný chod obfuskátoru požadovat, aby nebyl v kódu použit, přestože jeho podporova by nám způsobila menší potíže než eval.

4.2.3 Function.name

Do třetice zakážeme ještě používání Function.name. [24] uvádí, že name je property objektu funkce v JavaScriptu a je to řetězec nesoucí hodnotu identifikátoru funkce. Jeho přítomnost v kódu by nám znemožnila přejmenovávání identifikátorů funkcí, což by byla stejně jako v případě evalu velká překážka i v následujících transformacích kódu.

4.2.4 Implementace

S těmito omezeními se můžeme vrhnout do práce. Velkým pomocníkem nám v tomto úkolu bude kromě standardní estraverse také knihovna escope, která nás podpoří při řešení problémů spojenými s lokalizací proměnných.

Začneme vytvořením třídy, která pro nás bude generovat akceptovatelné identifikátory. Je potřeba ošetřit, aby nekolidovaly s žádným slovem rezervovaným JavaScriptem pro speciální identifikátory. [25] uvádí seznam všech

rezervovaných slov. Je nutné vyřešit i případné kolize s názvy globálních proměnných. Spuštěním příkazu `Object.keys( window );` v prohlížeči získáme seznam těchto proměnných. Vidíme, že ani mezi rezervovanými slovy ani mezi globálními proměnnými neexistují identifikátory začínající podtržítkem. Proto před každý námi generovaný identifikátor předsadíme podtržítko a budeme mít jistotu, že je z tohoto pohledu bezpečný. Identifikátory budeme generovat pseudonáhodně (aby nebylo možné zjistit, které byly vygenerované dřív a které později, což je informace, kterou nechceme reverzním inženýrům odevzdat zadarmo) a každý potom zařadíme do mapy použitých identifikátorů, abychom nevygenerovali dvakrát ten samý. Jednu a tu samou instanci této generující třídy budeme používat i v pozdějších transformacích kódu, čímž zajistíme unikátnost každého jednoho vytvořeného identifikátoru po celý proces obfuskace.

S touto podpůrnou třídou máme vše připraveno k samotným přejmenováváním. Pomocí estraverse budeme procházet AST programu a kdykoliv narazíme na nový scope (tzn. při vstupu do uzlu celého programu a poté při každém vstupu do těla nějaké funkce nebo do klauzule catch), vyhledáme všechny proměnné, které jsou deklarací navázané na tento navštívěný scope. Identifikátor každé takové proměnné nahradíme vygenerovaným jménem a tuto změnu propagujeme i u všech referencí na tuto proměnnou.

Zvláštní pozornost při tom musíme věnovat globálním proměnným. K proměnným, které jsou uložené jako členské proměnné objektu window, totiž můžeme přistupovat stejným způsobem jako ke globálním proměnným. Toto chování je znázorněno ve výpisu kódu 4.3.

```
window["x"] = 20;
x = 10;

console.log(x); //vypíše 10
console.log(window["x"]); //vypíše rovněž 10
```

Výpis kódu 4.3: Globální proměnné a objekt window

Globální proměnné proto necháme nepřejmenované. Jak se ukáže po dalších transformacích, které s kódem provedeme, velká část těchto globálních identifikátorů nakonec nezůstane úplně odkrytá, takže není třeba zoufat.

Nepřejmenované zůstanou i property všech objektů. Transformaci názvu property si nemůžeme dovolit proto, že k properties lze v JavaScriptu přistupovat nejen prostřednictvím operátoru tečky (`objekt.property`), ale také pomocí hranatých závorek (`objekt["property"]`), jak uvádí [26]. To znamená, že v mnoha případech nelze statickou analýzou zjistit, k jaké property daný zápis vlastně přistupuje, neboť argumentem operátoru hranatých závorek může být jakýkoliv výraz (tedy například proměnná, v níž je uložen řetězec nesoucí název kýžené property). S tímto problémem si poradíme později.

4.3 Inlining funkcí

Inlining funkcí je proces, při kterém nahrazujeme volání funkce přímo tělem této funkce. [15] uvádí, že se v zásadě jedná o jednosměrnou transformaci. Dodávají však také, že k tomu je potřeba, aby bylo tělo funkce doplněno všude, kde dochází k volání této funkce, a aby původní deklarace funkce byla z kódu vymazána. Jedině tak zahladíme veškeré stopy po dřívější abstrakci, kterou existence funkce představovala. My však podmínku vymazání deklarace funkce v naší implementaci porušíme, především z pragmatických důvodů.

4.3.1 Ošetření návratových hodnot

Při volání funkce dojde k úplnému ukončení vykonávání těla funkce ve chvíli, kdy se vykoná příkaz return (případně když funkce dojde na svůj konec). Toto chování (tedy úplné přeskočení celé části kódu až na konec těla funkce) je potřeba nějakým způsobem simulovat při inliningu funkcí. V jazycích jako C by nám k tomuto účelu velmi dobře posloužil příkaz goto, kterým bychom mohli vykonávání programu nasměrovat na návěští umístěné hned za konec těla původní funkce. V JavaScriptu budeme muset být kreativnější.

JavaScript nám umožňuje vytvořit návěští (label) pro pojmenování cyklu [27]. Kdybychom tedy vnořili celé tělo funkce do (jinak v podstatě nepotřebného) cyklu s vlastním labelem X, mohli bychom chování příkazu return simulovat pomocí příkazu break (odkazujícím se na label X). Break ukončí vykonávání našeho obalujícího cyklu a způsobí, že program skočí rovnou na jeho konec – tedy i na konec pomyslného těla původní funkce. Funkci a její inlining za použití obalujícího cyklu a příkazu break vidíme ve výpisu kódu 4.4.

Vykonání funkce nemusí vždy končit příkazem return. Kdybychom použili inlining v této podobě na takovou funkci, došlo by k situaci, že program nenarazí na žádný příkaz break a tělo obalujícího cyklu ze začne vykonávat znovu. Proto na konci každého obalujícího cyklu nastavíme návratovou hodnotu na undefined a za toto přiřazení vložíme příkaz break. Tím zajistíme, že námi vložený cyklus bude vždy opuštěn už po první iteraci, a to s nedefinovanou návratovou hodnotou, jak bychom očekávali.

4.3.2 Ošetření předání argumentů

Nejsnadnějším řešením problému předání argumentů při inliningu je vytvořit pro každý parametr P na pozici i původní funkce foo samostatnou proměnnou a tu naplnit výrazem, který se při volání funkce foo vyskytuje na i-té pozici v seznamu argumentů. Pro volání funkce `foo(1,2,3)` tedy musíme vytvořit tři nové proměnné a ty inicializovat postupně na hodnoty 1, 2 a 3. Tyto nové proměnné potom v inlinovaném těle původní funkce budeme používat namísto původních parametrů. Jediný háček tkví ve výrazech, které obsahují tzv. UpdateExpression v postfixovém tvaru (například `i++`). Takovými výrazy

```
//funkce, jejíž tělo chceme inlinovat
function foo() {
   if (podminka) {
      return 10;
   }
   return 20;
}

//výsledek inliningu
var navratovaHodnota = undefined;
obalujiciCyklus:
   /*cyklus je zdánlivě nekonečný, to ale nevadí,
   opustíme ho vždy už v první iteraci*/
   for (;true;) {
      if (podminka) {
         navratovaHodnota = 10;
         break obalujiciCyklus;
      }
      navratovaHodnota = 20;
      break obalujiciCyklus;
   }
```

Výpis kódu 4.4: Simulace příkazu return příkazem break

nemůžeme inicializovat proměnné simulující argumenty inlinované funkce, protože inkrementace nebo dekrementace proměnné se v těchto případech má provést až po dokončení běhu funkce. Řešením je odstranění operátorů inkrementace a dekrementace z těchto výrazů a jejich vložení za pomyslné tělo funkce. Pro lepší představu si prohlédneme výpis kódu 4.5, kde jsou tyto transformace názorně vidět.

```
foo(x++, 5); //původní volání funkce

//verze po provedení inliningu:
var arg1 = x; //první argument obsahuje nezměněnou hodnotu x
var arg2 = 5;
obalujiciCyklus: for(;true;) {
   //upravené tělo funkce foo...
}
x++; //až nyní provedeme inkrementaci
```

Výpis kódu 4.5: Simulace příkazu return příkazem break

4.3.3 Implementace

Teorii inliningu funkcí máme za sebou. Je evidentní, že každé nahrazení volání funkce jejím tělem bude v kódu lidským okem snadno rozpoznatelné, protože bude ohraničené podezřelou inicializací proměnných a na pohled nesmyslným cyklem s nápadným labelem. Zatímco inicializace proměnných, které zastupují argumenty funkce, se nám v pozdějších transformacích kódu podaří do jisté míry rozptýlit, cyklus s labelem zůstane na svém místě a je tak jasným majákem pro reverzní inženýry, že v této části kódu probíhal inlining. Z tohoto pohledu není proto nezbytně nutné, abychom odstraňovali původní deklaraci funkce, protože stopy po abstrakci, kterou v kódu zprostředkovávalo volání funkce, se nám nepodařilo zahladit, a tedy transformace stejně není jednosměrná.

Inlining funkcí sice pracuje dobře ve prospěch obfuskace, nicméně rozkopírovávání dříve zapouzdřeného kódu nám ubližuje z hlediska minifikace. Budeme se proto snažit s inliningem držet při zemi. Jednak shora omezíme počet uzlů v AST těla funkce na 100 – rozsáhlejší funkce inlinovat nebudeme. Jednak umožníme uživateli obfuskátoru (pomocí objektu config deklarovaného v souboru index.js), aby definoval maximální počet provedení inliningu jedné funkce a maximální počet provedení inliningu celkově. Také zakážeme inlining přímo rekurzivních funkcí, u kterého hrozí, že neustálým vnořováním téhož kódu nabyde do monstrózních rozměrů.

Z předchozích odstavců je zjevné, že není velký tlak na to, abychom inlinovali co nejvíce funkcí, ani na to, abychom inlinovali funkce na všech místech, odkud jsou volané. Dovolíme si proto v naší implementaci určité ujednodušení a znemožníme inlining funkcí, které obsahují klíčové slovo this, abychom se vyhnuli nepříjemnému ošetřování problémů s contextem. Dále znemožníme inlinig ve všech případech kromě těch nejběžnějších. K inliningu propustíme:

1. Příkaz, který obsahuje pouze volání funkce – např. `foo(a,b,c);`
2. Příkaz přiřazení, jehož pravá strana je tvořena pouze voláním funkce – např. `x += foo(a,b,c);`
3. Deklarace proměnné, jejíž inicializační hodnota je definována voláním funkce – např. `var x, y = foo(a,b,c), z = 10;`

V těch případech, kdy v kódu využíváme návratovou hodnotu funkce (tedy případ 2 a 3 z předchozího seznamu), vložíme vhodný příkaz přiřazení nebo deklarace těsně před každý příkaz break, který nahrazuje return původní funkce. Tuto transformaci si můžeme prohlédnout ve výpisu kódu 4.6.

K pojmenování identifikátorů (ať už je řeč o proměnných určených pro simulaci argumentů, nebo o návěštích pro obalující cykly) využijeme stejnou instanci třídy pro generování identifikátorů, kterou jsme využili v sekci 4.2. Tím zajistíme unikátnost vygenerovaných identifikátorů napříč celým kódem.

```
//deklarace funkce, kterou budeme inlinovat
function foo() {
   return 10;
}
var x = foo(), z = 20; //původní deklarace s voláním funkce

//po provedení inliningu:
obalujiciCyklus:
   for (;true;) {
      var x = 10, z = 20; //deklaraci vložíme sem
      break obalujiciCyklus; //break namísto returnu
   }
```

Výpis kódu 4.6: Zužitkování návratové hodnoty při inliningu funkce

Když jsme si už ujasnili, jak bude vypadat inlining jedné funkce, je potřeba ještě vymyslet, jak realizovat inlining funkcí v celém dokumentu na všech příslušných místech:

1. Pomocí estraverse projdeme celý AST a hledáme přitom nějakou deklaraci funkce. Jakmile narazíme na deklaraci D, zkontrolujeme, zda splňuje naše požadavky pro umožnění inliningu (tělo není příliš dlouhé, neobsahuje klíčové slovo this apod.). Pokud požadavky splňuje, její identifikátor označíme jako ID, rodičovský uzel jako R a pokračujeme dalším krokem. Pokud požadavky nesplňuje, hledáme jinou deklaraci, až dokud požadavky nejsou splněny.

2. Projdeme pomocí estraverse všechny potomky uzlu R a hledáme všechny příkazy P obsahující volání funkce s identifikátorem ID. (Předpokládáme, že každá funkce má v tuto chvíli unikátní identifikátor – díky přejmenování identifikátorů ze sekce 4.2 – takže nemůže tímto způsobem dojít k inliningu nesprávné funkce. Zároveň víme, že uzel umístěný v AST hierarchicky výše než R nemá přístup k deklaraci funkce D a nemůže ji tedy volat – kromě výjimečných případů, které bychom stejně nebyli schopni statickou analýzou detekovat. Proto nemá smysl takové uzly prohledávat.)

3. Pro každý příkaz z množiny P provedeme příslušný inlining, za předpokladu, že jsme dosud nepřekročili počet povolených provedení inliningu pro jednu funkci.

4. Po dokončení inlinování příkazů zkontrolujeme, zda jsme nepřekročili celkový počet povolených provedení inliningu. Pokud ne, pokračujeme prvním bodem a hledáme další deklarace.

4.4 Obfuskace řetězců a celých čísel

Jak uvádí [28]: „The top two most popular obfuscation technique is data obfuscation (e.g., string splitting) and encoding obfuscation (e.g., ASCII/Unicode/Hex encodeing)“. Ani my se těmto obfuskačním technikám nevyhneme.

Obfuskace řetězců je pro nás zvlášť důležitá, protože nám umožní obfuskovat některé identifikátory, které dosud zůstaly odkryté – konkrétně identifikátory properties objektů. Jak už jsme zmínili dříve, k properties objektů můžeme přistupovat dvojím způsobem – pomocí operátoru tečky a pomocí operátoru hranatých závorek. Operátor hranatých závorek přijímá jako argument řetězec, který nese název property, k níž chceme přistoupit. To je pro nás dobrá zpráva.

Nejprve tedy pomocí estraverse projdeme celý AST a každou MemberExpression upravíme tak, aby byla konstruována pomocí hranatých závorek a nikdy pomocí operátoru tečky. Jelikož tato transformace lze realizovat celkem úsporně, pro představu uvádíme (mírně upravený) úryvek kódu 4.7, který je za ni odpovědný:

```
estraverse.replace(ast, {
   enter: function (node, parent) {
      if (node.type === 'Identifier'
         && parent.type === 'MemberExpression'
         && parent.property === node
         && parent.computed === false) {
         parent.computed = true;
         return { type: 'Literal', value: node.name };
      }
   }
});
```

Výpis kódu 4.7: Převod zápisu property ze syntaxe tečky na hranaté závorky

Touto transformací jsme docílili toho, že veškeré reference na property nějakého objektu budou v kódu ve formě řetězců. Pokud se nám tyto řetězce podaří dostatečně zaobfuskovat, nebudou názvy identifikátorů properties nikde dostupné ani snadno rozklíčovatelné.

Nyní se vrhneme na obfuskace řetězců. Jako základní způsob pro obfuskaci řetězců zvolíme transformaci do kódování Base64. [29] uvádí: „The Base 64 encoding is designed to represent arbitrary sequences of octets in a form that requires case sensitivity but need not be humanly readable.“ Technické specifikace tohoto kódování nás nemusí tolik zajímat, podstatné je, že se jedná o text, který nelze člověkem na první pohled přečíst. Řetězec, který budeme chtít tímto způsobem obfuskovat, tedy nejdříve převedeme do formátu Base64 a následně ho obalíme funkcí, která ho převede zpět do standardního formátu. JavaScriptová funkce, která je dostupná v prohlížečích a která převádí řetězec z Base64 do jednoduchého textu, se nazývá atob [30]. Například výraz `"hello"`

tedy budeme převádět do tvaru `atob("aGVsbG8=")`. Je potřeba také poznamenat, že v NodeJS funkce atob (a její sesterská funkce btoa, která provádí transformaci z jednoduchého textu do Base64) neexistuje. Místo ní se zde pracuje s metodami objektu Buffer. Do objektu config v souboru index.js proto umístíme property browserJS, která zajistí, že se v místě transformace vždy dosadí správná funkce. Řetězec z našeho příkladu tedy pro NodeJS převedeme do poněkud delší verze `Buffer.from("aGVsbG8=", 'base64').toString()`.

Využití Base64 je relativně jednoduchý způsob obfuskace řetězců, dá se tedy předpokládat, že stadardní deobfuskátor by měl být schopen si s touto transformací poradit. Obfuskaci tedy rozšíříme. Využijeme k tomu výjimky. JavaScript zná šest druhů výjimek [31], mezi nimi také ReferenceError, TypeError a URIError. Tuto trojici použijeme.

Myšlenka je taková, že pomocí námi vytvořeného výrazu vygenerujeme JavaScriptovou výjimku (přičemž dopředu víme, jaký ponese název), tuto výjimku odchytneme a určité písmeno z názvu této výjimky umístíme namísto původního písmene v obfuskovaném řetězci. Nahrazované a nahrazující písmeno je samozřejmě shodné, nicméně tato informace prakticky nelze odhalit statickou analýzou, což nám hraje do karet. V dokumentaci firmy Mozilla [32] [33] [34] se dočteme o snadných a úsporných způsobech, jakými můžeme vyvolat tu kterou výjimku. Pro představu se můžeme podívat na výpis kódu 4.8, kde lze vidět obfuskaci písmene R pomocí výjimek.

```
function generateURIError() {
   try {
      decodeURIComponent('%'); //vytvoří výjimku URIError
   } catch (e) {
      return e.name; //vrací řetězěc 'URIError'
   }
}

//vypíše 2. písmeno názvu výjimky, tedy R
console.log(generateURIError().charAt(1));
```

Výpis kódu 4.8: Obfuskace písmene R pomocí výjimky URIError

Problém je, že máme k dispozici pouze ta písmena, která jsou obsažena v názvech JavaScriptových výjimek, o jiných než abecedních znacích ani nemluvě. Řešení je snadné. Jiné znaky, než které jsou přímo v názvech výjimek, budeme vytvářet sečtením/odečtením ASCII hodnoty nějakého písmene z názvu výjimky a pevně určeného celého čísla. Písmeno J tedy vytvoříme například tímto výrazem:

```
String.fromCharCode(generateURIError().charAt(1).charCodeAt(0) - 8)
```

Řetězením takto vygenerovaných písmen jsme schopni poskládat jakýkoliv řetězec, který se v kódu objeví. Taková obfuskace však má příliš velké náklady – volání funkce a generování a odchytávání výjimky kvůli každému jednomu znaku může obfuskovaný program výrazně zpomalit. Zvolíme proto strategii

spojení obfuskace pomocí Base64 a generování výjimek.

Pomocí estraverse projdeme celý AST obfuskovaného programu a hledáme jakýkoliv řetězec. Jakmile na nějaký narazíme, zkontrolujeme, zda se jedná o řetězec použitý ve funkci eval (obfuskace takových řetězců je pro nás zvlášť důležitá, jak uvidíme v sekci 4.6) a nebo řetězec o počtu znaků od tří do pěti. Pro takové řetězce totiž platí:

- Lze je rozdělit alespoň do tří neprázdných částí.
- Jsou dostatečně krátké na to, aby ztráta informace o jediném písmenu z řetězce představovala větší než zanedbatelný problém pro reverzního inženýra.

Pokud se tedy skutečně jedná o řetězec splňující tato kritéria, rozdělíme ho do tří částí – první znak, druhý znak a zbytek znaků. První a třetí část zaobfuskujeme pomocí Base64, druhou část pomocí výjimek, a všechny tři části následně spojíme operátorem +. Výsledkem je plně obfuskovaný řetězec, který lze velmi obtížně úplně rozklíčovat a u kterého zároveň platí, že jeho neúplné rozklíčování může mít fatální následky pro deobfuskaci. U ostatních řetězců (tedy kratších než tříznakových nebo delších než pětiznakových, které nejsou argumentem funkce eval) budeme obfuskovat pouze pomocí Base64, čímž zajistíme, že k vyvolávání výjimek nebude docházet příliš často, nebo v situacích, kdy nebudou dostatečně efektivní.

Nyní už jsme jen krůček k obfuskaci celých čísel. Opět využijeme obfuskaci pomocí výjimek a ze stejných důvodů jako u obfuskace řetězců se budeme snažit nepoužívat ji příliš často. Proto tímto způsobem zaobfuskujeme jen přibližně každé páté číslo, na které při traverzování abstraktním syntaktickým stromem narazíme. Obfuskace celého čísla probíhá následujícím způsobem:

1. Vyvoláme a odchytneme pseudonáhodně vybranou výjimku V z repertoáru námi podporovaných výjimek.
2. Pseudonáhodně vybereme nějaký znak z výjimky V a získáme jeho ASCII hodnotu A.
3. K hodnotě A přičteme takové celé číslo, aby výsledek byl roven číslu, které chceme zaobfuskovat.
4. Celý takto zkonstruovaný výraz vložíme namísto obfuskovaného čísla.

A máme hotovo. Budeme důslední a zmíníme ještě, že generování a odchytávání každého druhu výjimek se děje v námi vytvořených samostatných funkcích, podobně, jako jsme to viděli ve výpisu kódu 4.8.

4.5 Outlining příkazů do funkcí

Outlining příkazů do funkcí je transformace se středně vysokou potencí a silnou odolností. Tato transformace v podstatě vytváří v kódu dojem abstrakce na místech, kde ve skutečnosti žádná není. Zároveň, jak uvádí [15], se jedná o transformaci vhodně se doplňující s inliningem funkcí, protože dále zahlazuje stopy po dřívějším inliningu a zvyšuje tak jeho odolnost. Je pravda, že chytřejší deobfuskátory mohou relativně snadno pojmout podezření, že funkce není originální, ale je outlinovaná – jasným indikátorem může být například fakt, že je v kódu volána právě jednou. Je potom otázka, zda se deobfuskátor rozhodne podstoupit riziko, že odstraní z kódu užitečnou abstrakci, a kód inlinuje zpět. Předcházet tomu však částečně můžeme pomocí slučování funkcí, a proto se budeme snažit této transformaci vyjít co nejvíce vstříc, jak uvidíme dále v této kapitole.

Deklarace funkcí, do nichž budeme outlinovat, budeme umisťovat vždy na začátek stejného scopu (tedy buď na začátek mateřské deklarace funkce, nebo na začátek programu u globálního scopu), ze kterého pochází outlinované příkazy. Tím si trochu ujednodušíme práci a snížíme náklady transformace, protože veškeré proměnné, které jsou dostupné pro outlinované příkazy, budou dostupné i pro nově vzniklou funkci. Nebudeme tedy muset předávat žádné argumenty. Poznamenejme ještě, že pokud mateřský scope obsahuje direktivu `'use strict';`, je potřeba outlinované funkce umisťovat až za tuto direktivu.

Pojďme si nyní popsat fungování algoritmu. Příkazy k outlinování budeme vyhledávat uvnitř bloků, na které narazíme pomocí estraverse. Pseudonáhodně určíme první a poslední příkaz, který budeme outlinovat, a oba (a s nimi i všechny, které leží mezi nimi) umístíme do nové deklarace funkce. Abychom nenarušili chování programu, musíme dbát na to, aby v outlinovaných příkazech nebyl ani jeden z těchto výrazů:

- Identifikátor arguments (nově vzniklá funkce bude mít pochopitelně vlastní proměnnou arguments a tudíž by příkaz obsahující identifikátor arguments měl po outlinování jinou hodnotu).
- Příkaz return, který by po outlinování reprezentoval návrat z outlinované funkce, přestože byl zamýšlen jako návrat z původní funkce.
- Příkazy break a continue, které by při nešikovném outliningu mohly způsobit narušení chování cyklů.
- Deklarace funkcí, které nelze outlinovat, protože bychom tím narušili jejich dostupnost a mohli bychom tak způsobit, že některá volání těchto funkcí by se stala neplatná.
- Výraz this, jehož outlining by mohl zapříčinit nesprávné chování kvůli změně contextu.

Vynechání těchto výrazů z outliningu zařídíme tak, že projdeme všechny zvolené příkazy od prvního až po poslední a jakmile na pozici i narazíme na takový, který obsahuje jeden z výše vyjmenovaných výrazů, označíme příkaz na pozici $i-1$ za nový „poslední příkaz“. Outlining potom provedeme stejně, jen tímto způsobem zmenšíme počet outlinovaných příkazů. Pokud bychom měli outlinovat méně, než jeden příkaz, pochopitelně proces zastavíme.

Outlinované příkazy samozřejmě odstraníme z původního bloku a namísto nich vložíme volání funkce, do nichž byly outlinovány. Takto zkonstruovaná transformace je funkční až na jedno slabé místo – outlinované deklarace proměnných. Pokud bychom totiž provedli outlining deklarace proměnné, tato proměnná nebude už dostupná v původním scopu. Mohli bychom sice zařadit deklarace proměnných na seznam výrazů, které nelze outlinovat, jednalo by se však o relativně velké omezení. Řešení je relativně přímočaré.

Kdykoliv bychom outlinovali deklaraci proměnné x spojenou s inicializací, použijeme namísto toho jednoduchý příkaz přiřazení. Například namísto `var x = 10;` budeme outlinovat pouze `x = 10;`. Samotnou deklaraci (nyní už bez inicializace, tedy například `var x;`) potom umístíme na začátek těla scopu, z něhož jsme outlinovali (i zde platí, že pokud tělo scopu obsahuje direktivu `'use strict'`, umístíme deklarace až za ni). Tím zajistíme, že proměnná bude dostupná jak v původním scopu, tak ve scopu námi vytvořené funkce (protože tato dědí rodičovský scope, tedy i naši proměnnou x).

Zmíníme ještě, že vzhledem k faktu, že slučování funkcí ze sekce 4.10 nám pomůže zvýšit potenci i odolnost této transformace, vyjdeme této transformaci vstříc a na každý navštívený blok se pokusíme o outlining dvakrát – zajistíme tím, že v mnoha blocích vzniknou dvě funkce, což je už dostatečný počet pro sloučení.

4.6 Outlining příkazů do funkce eval

Podobnou transformací, jakou byl outlining příkazů do funkcí, je outlining do funkce eval. Zásadní rozdíl však spočívá ve vlastnostech obou operací. Outlining do funkce eval v podstatě tkví v tom, že určitou část kódu přetransformujeme do řetězce a ten vložíme jako argument funkce eval, kterou zavoláme. Kód se vykoná stejně, jako by se vykonal bez outliningu a na první pohled je tato transformace i docela dobře rozklíčovatelná – deobfuskátoru stačí přemístit kód z argumentu funkce eval na místo samotného volání eval a je hotovo. Tato transformace má tedy nízkou potenci i odolnost, zato náklady budou relativně vysoké – jak jsme uváděli už v sekci 4.2, používání funkce eval povede ke zpomalení programu. Potence této transformace však výrazně vzroste po aplikaci obfuskace řetězců – bez úspěšné deobfuskace řetězců zůstanou celé části kódu reverznímu inženýrovi úplně zakryté.

Na začátek stanovme omezení pro tento typ transformace. Zakážeme outlining pro výrazy obsahující deklaraci funkce nebo proměnné, protože jak

uvádí [35]: „Eval of strict mode code does not introduce new variables into the surrounding scope“. Dále nebudeme připouštět outlining výrazů obsahující break, continue nebo return, protože by tím došlo k nečekanému chování kódu – eval by přišel o nezbytný kontext (například z jaké funkce byl volán return nebo který cyklus chceme přerušit pomocí příkazu break).

Algoritmus samotného outliningu je relativně přímočarý. Pomocí estraverse hledáme příkazy zakořeněné v nějakém bloku, které splňují výše popsané podmínky. Takové příkazy potom pomocí escodegen transformujeme do řetězce S a nahradíme je voláním funkce eval s parametrem S.

Zmíníme ještě, že vzhledem k nákladům této transformace umožníme uživateli obfuskátoru prostřednictvím property chanceForEval objektu config změnit pravděpodobnost na provedení transformace nad každým navštíveným příkazem. Zároveň omezíme počet potomků outlinovaného příkazu na číslo 25 – rozsáhlejší AST nebudeme outlinovat, abychom nevytvářeli přehnaně dlouhé řetězce pro interpret, což by vedlo k značnému zpomalení aplikace.

4.7 Přeuspořádání příkazů v bloku

[15] říká, že přeuspořádání příkazů je jednosměrná transformace s nízkou potencí. Jejím smyslem je v zásadě zakrýt takové vazby mezi příkazy, které jsou v kódu promítnuty jejich prostorovou blízkostí. V implementaci této funkce budeme používat pouze jeden druh záměny příkazů, a to výměnu dvou bezprostředně sousedících příkazů, které sdílejí stejného rodiče – blok kódu. Pojďme se podívat, kdy takovou záměnu budeme moci provést.

Nejprve označíme příkazy, jejichž pozici nebudeme měnit nikdy. Jedná se o příkazy, u kterých nelze v žádném případě výměnu provést, nebo ji lze provést jen v případech, které jsou příliš obtížně odhalitelné. Konkrétně jde o příkazy, které v rámci AST obsahují ve svém podstromu jedno nebo více z následujícího:

- Příkaz return
- Příkaz break
- Příkaz continue
- Příkaz try (a potažmo příkazy catch a finally, které jsou však v AST vytvořeném pomocí parseru knihovny espree uložené jako property uzlu TryStatement, takže je nemusíme ošetřovat zvlášť)
- Direktiva `'use strict'`;
- Volání funkce – přítomnost volání funkce v tomto seznamu nás bude bolet nejvíce, protože se jedná o velice častý výraz. Problém však je,

že zjišťování závislostí mezi dvěma příkazy, z nichž jeden nebo oba obsahují volání funkce, by pro nás znamenalo velice složitou rutinu (obsahující analýzu scopu, zanořování se do těl funkcí použitých uvnitř prověřovaných volání – s prevencí rekurzivního zanoření – a další záludnosti), na jejímž konci zdaleka ne vždy bude uspokojivý výsledek. Obzvlášť nepříjemné jsou případy, kdy se v některé z volaných funkcí používá jiná funkce (například z knihovny, která pro nás není dostupná), k jejímuž tělu nemáme přístup. Raději proto využijeme svou energii na účinnější obfuskační techniky.

Nyní už tedy známe druhy příkazů, které nelze nikdy s jiným příkazem vyměnit. Co s těmi ostatními? Jednoduchou odpovědí by mohlo být, že pokud se někde v podstromu příkazu A vyskytuje proměnná x a někde v podstromu příkazu B (který následuje po příkazu A) se vyskytuje stejná proměnná x, nelze příkazům A a B vyměnit pořadí. Tato úvaha by však byla zbytečně zjednodušující – záleží totiž na tom, zda a který příkaz do proměnné x zapisuje. My se budeme v implementaci řídit jiným modelem (sice také mírně zjednodušeným, ale nikoliv zásadním způsobem):

1. Pokud existuje proměnná, do které příkaz A zapisuje a kterou příkaz B čte, příkazy nelze prohodit.

2. Pokud existuje proměnná, do které zapisuje příkaz A i příkaz B, příkazy nelze prohodit.

3. Pokud existuje proměnná, do které příkaz B zapisuje a kterou příkaz A čte, příkazy nelze prohodit.

4. Pokud pro všechny proměnné použité v příkazu A i v příkazu B platí, že jsou pouze čteny, příkazy prohodit smíme.

V tomto modelu jsme zanedbali okrajové situace, například když A i B zapisují do stejné proměnné, ale pokaždé stejnou hodnotu (v takovém případě bychom příkazy prohodit mohli). Zohledňování těchto případů zanedbáme, jednak proto, že by jejich odhalování mnohdy nebylo statickou analýzou možné, jednak proto, že jejich četnost bude minimální. Popsaným modelem nemůžeme nikdy narušit datové závislosti mezi příkazy, takže obfuskací nezpůsobíme nežádoucí chování programu.

Otázka je, jak poznat, kdy do proměnné kód zapisuje a kdy z ní čte. Zde si také dovolíme jisté ujednodušení: Řekneme, že z proměnné čteme v situacích, kdy se v rámci podstromu prohledávaného příkazu objevila na pravé straně každého operátoru přiřazení a zároveň není vnořena do výrazu typu UpdateExpression (tj. např. inkrementace nebo dekrementace). Řekneme, že do proměnné zapisujeme, pokud se v podstromu prohledávaného příkazu objevila v jakékoliv situaci odlišné od výše zmíněné. Tento model sice opět opomíjí

některé neobvyklé situace, v nichž bychom mohli volit mírnější verzi závislosti čtení, ale zvolíme místo ní závislost zápisu. Chování kódu však tímto zjednodušením opět neutrpí žádné změny (neboť v nedořešených situacích volíme vždy přísnější variantu závislosti), proto se s ním spokojíme.

Nyní se podívejme na fungování algoritmu pro míchání příkazů v bloku jako na celek. Pomocí estraverse procházíme AST obfuskovaného programu a hledáme bloky kódu. Jakmile na nějaký narazíme, označíme ho *body* a spustíme na něj algoritmus z výpisu kódu 4.9, jehož některé části jsou pro přehlednost nahrazeny pseudokódem.

```
for (var i = 1; i < body.length; i++) {
   //Každý příkaz body[i] "probubláváme" nahoru
   for (var j = i; j > 0; j--) {
      //začleníme prvek náhody
      if (randomInt({min:0, max:2}) === 0) {
         //Zde ověříme, zda příkazy nekolidují v závislostech
         if (!statementsInterfere(body[j], body[j-1])) {
            //prohodíme příkazy
            switch(body[j],body[j-1]);
         }
      }
   }
}
```

Výpis kódu 4.9: Algoritmus prohazování příkazů v bloku

Můžeme si všimnout jisté podobnosti mezi naším algoritmem a bubble sortem, která nám dá lepší představu o tom, jak algoritmus pracuje. Jak je vidět, nikdy neprohazujeme jiné než sousedící příkazy, takže námi stanovené podmínky jsou dodržené. Výsledkem je uspořádání příkazů, které se sice zdaleka není zcela náhodné, ale které zahladilo mnohé souvislosti mezi příkazy bez poškození chování původního kódu.

4.7.1 Roztržení deklarací proměnných

V JavaScriptu platí, že pokud je nějaká proměnná deklarovaná v určitém scopu S pomocí konstrukce var, je tato proměnná dostupná kdekoliv ve scopu S (což ovšem neznamená, že bude inicializovaná). Nezáleží tedy na tom, zda je proměnná deklarována před nebo po tom, co s ní pracujeme – důležité je, že někde deklarovaná je. Toto chování je prezentováno ve výpisu kódu 4.10.

Této vlastnosti JavaScriptu využijeme k obfuskaci. Projdeme AST našeho programu a budeme hledat deklarace proměnných. Takové deklarace, které obsahují více proměnných (tj. více položek v poli declarators) než jednu, rozdělíme na samostatné deklarace tak, aby v každé byla právě jedna proměnná. Následně ještě u těch deklarací, které obsahují inicializaci, oddělíme deklaraci a inicializaci do dvou samostatných příkazů. Deklaraci `var x, y = 8, z;`

```
function foo() {
   x = 5;
   var x; //deklarujeme x až po inicializaci
   return x;
}

console.log(foo()); //vypíše 5
console.log(x); //vyvolá výjimku (x není v tomto scopu definováno)
```

Výpis kódu 4.10: Chování deklarace proměnných

tedy například transformujeme do podoby `var x; var y; y = 8; var z;`. Jednotlivé deklarace potom přemístíme na různá místa (ovšem při zachování stejného scopu). Toto přemístění může být zcela náhodné, díky chování JavaScriptu není potřeba dbát na jakékoliv datové závislosti ostatních příkazů.

Konkrétně naše implementace bude fungovat tak, že pro každou samostatnou deklaraci D náhodně zvolíme příkaz P, který je umístěný ve stejném bloku (deklarace umístěné jinde, než bezprostředně uvnitř bloku, budeme ignorovat – takových bude minimum). Pokusíme se deklaraci D umístit do podstromu příkazu P pomocí algoritmu ve výpisu kódu 4.11. Pokud se nám to nepovede, jednoduše vložíme deklaraci D bezprostředně za příkaz P v rámci výčtu příkazů společného bloku.

```
estraverse.traverse(statementSubtree, {
   enter: function (node) {
      if (/Function/.test(node.type) || node.type === 'CatchClause') {
         /* Nechceme se zanořit do jiného scopu */
         this.skip();
      }
      if (node.type === 'BlockStatement' && randomInt(0, 2) > 0) {
         /* Pokud jsme narazili na nový blok, s pravděpodobností
         2/3 umístíme naši deklaraci do něj na náhodné místo.
         Jinak bude traverse pokračovat dál. */
         var newPosition = randomInt(0, node.body.length - 1);
         node.body.splice(newPosition, 0, declaration);
         propagated = true;
         this.break();
      }
   }
});
```

Výpis kódu 4.11: Propagace deklarace proměnné do podstromu příkazu

Tento algoritmus by měl zajistit dostatečné rozprostření deklarací, aby jejich původní poloha nebyla při deobfuskaci dohledatelná. Ve většině případů se bude jednat o jednosměrnou transformaci.

4.8 Přeuspořádání parametrů funkcí

Pořadí parametrů ve funkci mívá typicky nějaký řád a význam. Jak uvádí [15], „the closer two items or events are in space or time, the higher the likelihood that they are related in one way or another.“ Přeuspořádání parametrů funkcí do náhodného pořadí je technika s relativně velmi nízkou potencí, protože informace obsažená v pořadí parametrů není velká, zato se v ideálním případě jedná o prakticky jednosměrnou (a tedy vysoce odolnou) obfuskační techniku (s jistými výjimkami, které si popíšeme níže) s téměř nulovými dopady na náklady a minimálními negativními dopady na minifikaci.

Nejprve pojďme stanovit situace, kdy přeuspořádání parametrů smíme provést. Problém bude funkce, která používá proměnnou arguments ve svém těle. Arguments je proměnná každé funkce, která umožňuje přístup k argumentům prostřednictvím indexu (například k druhému předanému argumentu funkce bychom přistoupili pomocí výrazu `arguments[1]`) [36]. Na funkce, které používají tuto proměnnu ve svém těle, nebudeme tuto transformaci aplikovat, protože bychom chování proměnné arguments porušili.

Další funkce, které z této transformace vynecháme, jsou ty, jejichž identifikátor se v programu objevuje v jiné situaci, než je jednoduché volání (a deklarace funkce samozřejmě). Představme si například situaci ve výpisu kódu 4.12. Pokud se v programu předává reference na funkci, přestáváme mít pod kontrolou, kdy je vlastně funkce volaná, a tudíž si nemůžeme dovolit měnit pořadí parametrů. Z toho důvodu tyto funkce vynecháváme.

```
function foo(p1, p2) {
   console.log(p1 - p2);
}
var x = foo;
x(3,2);
```

Výpis kódu 4.12: Použití identifikátoru funkce v jiné situaci než jednoduché volání

Třetí a poslední případ, u kterého přeuspařádání parametrů vynecháme, je situace, kdy se volání dané funkce vyskytuje ve výrazu, kterým inicializujeme nějakou proměnnou. Z důvodů, které si vysvělíme později, budeme muset některá volání transformovaných funkcí změnit na tzv. SequenceExpression (několik výrazů oddělených čárkou), který se ovšem ze syntaktických důvodů nemůže objevit v deklaraci proměnné.

Než se pustíme do popisu algoritmu na zamíchání parametrů, je potřeba si říct, jaké mohou mít argumenty mezi sebou závislosti, které bychom mohli touto transformací porušit. Situace je velmi podobná, jako u přeuspořádání příkazů v bloku v sekci 4.7, jen namísto příkazů nyní pracujeme s argumenty funkce. V rychlosti shrneme, oč jde. U jednotlivých argumentů zjistíme, s jakými pracuje proměnnými – tzn. do kterých proměnných se zapisuje a ze kterých se čte (a případně také, u kterých argumentů je možné, že se do nich zapisuje nebo se z nich čte, jen to nejsme schopni detekovat). Na základě těchto informací zjistíme, které dva argumenty nemůžeme prohodit, aniž bychom riskovali poškození chování programu. Pro každý argument A, u kterého zjistíme, že by mohl tímto způsobem kolidovat, vytvoříme deklaraci proměnné V a tu inicializujeme (ještě před voláním funkce) hodnotou argumentu A. Všechny takto „oddelegované“ inicializace argumentů budeme vkládat do SequenceExpression (na jejíž konec umístíme původní volání funkce), a to ve stejném pořadí, v jakém se původně argumenty předávaly do funkce. Takto vzniklý výraz potom umístíme namísto původního volání funkce. Pro lepší představu uvádíme výpis kódu 4.13, kde lze postup vytváření SequenceExpression dobře odpozorovat.

```
foo(++x, x, 5); //původní volání funkce

var p1, p2; //deklarace 'oddelegovaných' argumentů
//SequenceExpression nahrazující původní volání:
p1 = ++x, p2 = x, foo(p1, p2, 5);
```

Výpis kódu 4.13: Nahrazení volání funkce nově vytvořenou SequenceExpression

Deklarace proměnných reprezentující „oddelegované“ argumenty umístíme do stejného scopu, ve kterém se objevilo volání funkce. Nyní si můžeme popsat obecný algoritmus přeuspořádání parametrů funkcí. Pomocí estraverse budeme procházet AST a hledat jakoukoliv deklaraci funkce. Pro každou z těchto deklarací provedeme následující:

1. Funkci, na jejíž deklaraci jsme narazili, označíme F. Pro pseudonáhodné roztřídění parametrů použijeme mírně upravenou implementaci Fisher-Yatesova algoritmu uvedenou v [37]. Tímto algoritmem zamícháme parametry v deklaraci funkce F.

2. Projdeme pomocí estraverse AST obfuskovaného kódu a hledáme všechna volání funkce F (předpokládáme, že díky přejmenovávání identifikátorů ze sekce 4.2 mají všechny funkce unikátní identifikátor, proto není potřeba starat se o scope a hledáme pouze shodu identifikátorů deklarace a volání). V každém takovém volání nejprve provedeme kroky ošetřující případnou kolizi argumentů uvedenou výše. Poté doplníme

hodnotami undefined seznam argumentů tak, aby byl výsledný počet argumentů stejný, jako je počet parametrů funkce F (neuvedení žádného argumentu na příslušném místě totiž v podstatě znamená, že předáváme jako argument hodnotu undefined). Poté seznam argumentů zamícháme do stejného pořadí, jako jsme zamíchali parametry deklarace funkce F.

4.9 Přeuspořádání deklarací funkcí

Přeuspořádání deklarací funkcí má pro nás obdobný význam jako přeuspořádání parametrů. Jak uvádí [15]: „The potency of these transformations is low and the resilience is one-way.“ Tato technika však pro nás má ještě jeden podstatný efekt, protože jak uvidíme v sekci 4.12, bude aplikace této transformace zároveň zvyšovat potenci slučování funkcí ze sekce 4.10.

Implementace této obfuskační techniky je velmi přímočará a jedna z nejjednodušších v celé práci. Pomocí estraverse procházíme AST a hledáme bloky kódu (ohraničené složenými závorkami). Pro každý takový blok B vytvoříme pole funkcí, které bezprostředně v bloku B obsahují svou deklaraci. Zároveň všechny tyto deklarace funkcí z bloku odstraníme. Deklarace funkcí potom zamícháme pomocí Fisher–Yatesova algoritmu (stejná implementace, kterou jsme použili v sekci 4.8) a v tomto novém pořadí je umístíme na začátek bloku B (jako obvykle je potřeba ohlídat, abychom deklarace umístili až za případnou direktivu `'use strict';`).

4.10 Slučování funkcí

Slučování funkcí znamená spojení těl několika funkcí dohromady a upravení volání původních funkcí tak, aby odpovídaly nové struktuře (samozřejmě při zachování původního chování volání funkcí). Tato transformace tak, jak ji budeme implementovat my, má nízkou potenci, nicméně výrazně zvyšuje odolnost některých jiných obfuskačních technik (více v sekci 4.12). Náklady této transformace jsou nízké.

Označme nejprve, které funkce lze sloučit dohromady. Vyloučíme z transformace všechny funkce, které pracují s proměnnou arguments (při slučování budeme měnit počet a do jisté míry i pořadí parametrů funkcí, což by chování proměnné arguments narušilo). Dále ze stejných důvodů, které jsme probrali už v kapitole 4.8, nebudeme povolovat sloučení funkcí, které se používají v jiných situacích než je jednoduché volání.

Pomocí estraverse procházíme AST a hledáme bloky kódu. Příkazy v každém takovém bloku kódu procházíme for cyklem a hledáme deklarace funkcí. Každou funkci s indexem i (kde i je liché) sloučíme s funkcí s indexem $i + 1$. Pokud funkce s lichým indexem i je poslední deklarací v daném bloku, slučovat ji nebudeme.

```
//původní funkce A
function A(p1, p2) {
   var x = 10;
   return x + p1 + p2;
}

//původní funkce B
function B(p1) {
   if ("foo")
      return true;
   return false;
}
```

Výpis kódu 4.14: Příklad dvou funkcí A a B

Mějme dvě funkce, A (která má n parametrů) a B (která má m parametrů). Pro představu uvádíme jako příklad dvě takové funkce ve výpisu kódu 4.14, nad kterými budeme provádět dále popisované transformace.

Sloučení těchto funkcí probíhá tak, že vytvoříme novou deklaraci funkce M (identifikátor této funkce je vytvořen generátorem ze sekce 4.2, stejně tak identifikátory jejích parametrů) a tu umístíme do stejného bloku, ve kterém byly umístěny deklarace funkcí A a B. Počet parametrů funkce M je $max(n, m) + 1$, přičemž první parametr se bude používat k rozhodnutí, které ze dvou původních těl se má provést (tzv. rozhodující proměnná) a ostatní parametry budou sloužit jako parametry pro původní funkce. Díky tomu, že JavaScript je slabě typovaný jazyk, nemusíme dbát na typy jednotlivých parametrů. Druhý až $(n+1)$-ní parametr nové funkce M bude odpovídat popořadě prvnímu až n-tému parametru funkce A. Druhý až $(m + 1)$-ní parametr budou analogicky odpovídat parametrům funkce B. V tělech původních funkcí je potřeba přejmenovat identifikátory referencí na původní parametry na odpovídající nové identifikátory parametrů funkce M.

V těle funkce A nalezneme pomocí estraverse nějaký literál a označíme ho l_1 (pokud funkce žádný literál neobsahuje, dosadíme hodnotu 0). Všechny výskyty tohoto literálu v těle funkce A zaměníme za referenci na první parametr nové funkce M. Stejný proces provedeme pro funkci B a literál l_2 (přičemž hledáme takový literál, pro který platí, že $l_2 \neq l_1$). Do (dosud prázdného) těla funkce M vložíme příkaz if-else. Podmínka tohoto příkazu bude, že první parametr funkce M (tedy rozhodující proměnná) je roven literálu l_1. Do then-bloku této podmínky vložíme tělo funkce A, do else-bloku vložíme tělo funkce B. Na výpisu kódu 4.15 se můžeme podívat, jak tato transformace proběhla.

Nyní procházíme AST a hledáme volání funkce A. Identifikátor každého takového volání změníme na identifikátor funkce M. Argumenty zůstanou stejné,

```
//sloučená funkce M. l1 = 10, l2 = "foo"
function M(rozhodujiciPromenna, p2, p3) {
   if (rozhodujiciPromenna == 10) {
      //upravené původní tělo funkce A
      var x = rozhodujiciPromenna;
      return rozhodujiciPromenna + p2 + p3;
   } else {
      //upravené původní tělo funkce B
      if (rozhodujiciPromenna)
         return true;
      return false;
   }
}
```

Výpis kódu 4.15: Slučování těl funkcí A a B do funkce M

```
//původní volání:
x = A(5, 20);
B(18);
//nová volání, kterými nahradíme ta původní:
x = M(10, 5, 20); //před první argument umístíme l1 (tzn. 10)
M("foo", 18); //před první argument umístíme l2 (tzn. "foo")
```

Výpis kódu 4.16: Slučování volání funkcí A a B

jen před první argument předsadíme literál l_1. Stejným způsobem změníme volání funkce B na volání funkce M (a před první argument vložíme l_2). Na upravená volání funkcí A a B z výpisu 4.15 se podíváme ve výpisu kódu 4.16.

Nyní už jen odstraníme deklarace původních funkcí A a B a máme hotovo. Pro úplnost ještě doplňme, že literály l_1 a l_2 nelze vybírat zcela libovolně. Problematické jsou situace, kdy zmíněné literály plní funkci názvu property nějakého objektu. Například kód `var x = {'prop': 20};` nevytvoří stejnou proměnnou x jako kód `var l = 'prop'; var x = {l : 20};`. Z tohoto důvodu musíme při výběru literálů l_1 a l_2 dbát na to, aby nebyly v těle původních funkcí použity v obdobných situacích.

Docílili jsme toho, že se funkce sloučily v jednu a přidali jsme jeden nový parametr (rozhodující proměnnou), jenž určuje, které z původních dvou těl se má vykonat. Rozhodující proměnná zároveň nese i význam takzvaného „neprůhledného predikátu“ [2], protože je zároveň začleněn i uvnitř původních těl funkcí a tudíž zdánlivě plní roli plnohodnotného parametru. Hodnota l_1 (v našem příkladu hodnota 10) je sice přímo viditelná v obalující podmínce uv-

[2] Výraz neprůhledný predikát je přeložen z „opaque predicate“ v [15].

nitř těla funkce M a je tudíž pro deobfuskátor snadno odhalitelná, hodnota l_2 však z těla funkce M zmizí úplně – je přítomná pouze ve voláních funkce – a je tudíž dobře zakrytá. To je také důvod, proč neslučujeme nikdy více než dvě funkce dohromady – vždy pouze u poslední ze slučovaných funkcí (jejíž tělo bude jako jediné obsažené v else-bloku) se podaří tuto hodnotu zakrýt v těle funkce, takže dává z hlediska obfuskace dobrý smysl slučovat vždy maximálně dvě funkce.

Ještě je potřeba ošetřit funkce deklarované v globálním scopu. Takové funkce nemůžeme jednoduše nahradit novou sloučenou funkcí s novým identifikátorem – globální funkce totiž mohly být volané prostřednictvím globálního objektu window nebo výrazu this použitém v globálním scopu. Abychom tento problém vyřešili, vytvoříme u každé slučované funkce z globálního scopu takzvanou „zástupnou deklaraci“. V podstatě se jedná o deklaraci umístěnou v globálním scopu, jejíž identifikátor i seznam parametrů je stejný, jako měla původní funkce. V těle této zástupné deklarace je pouze volání výsledné sloučené funkce s odpovídajícími parametry umístěné jako argument příkazu return. Například kdyby funkce z výpisu kódu 4.14 byly umístěné v globálním scopu, vytvořili bychom pro ně zástupné deklarace uvedené ve výpisu kódu 4.17.

```
function A(p1, p2) {
   return M(10, p1, p2);
}

function B(p1) {
   return M("foo", p1);
}
```

Výpis kódu 4.17: Zástupné deklarace funkcí

Tímto zajistíme, že volání původních funkcí prostřednictvím globálního objektu si udrží očekávané chování i návratovou hodnotu. Za zmínku stojí fakt, že i když vytvoříme zástupné deklarace v globálním scopu, jistý prostor pro deviaci od původního chování kódu existuje. Vytvořili jsme totiž v globálním scopu funkci s novým identifikátorem. To znamená, že pokud byla v kódu někdy prostřednictvím globálního objektu vytvořena funkce se stejným identifikátorem (například jako ve výpisu kódu 4.18), mohli jsme tuto deklaraci funkce nešikovně přepsat.

Přestože možnost takové kolize pro nás představuje určité riziko, pravděpodobnost, že dojde k jeho naplnění, je velice nízká. Pokud bychom přesto chtěli mít v tomto ohledu jistotu, museli bychom slučování globálních funkcí vypnout, což ale v naší implementaci neuděláme.

```
//deklarace funkce M umístěná v původním kódu
window["M"] = function() {
   console.log("x");
}
/* Pokud nyní v globálním scopu sloučíme dvě funkce a vytvoříme novou
funkci M, přepíšeme tím původní funkci M a poškodíme chování kódu. */
```

Výpis kódu 4.18: Kolize identifikátorů funkce v globálním scopu způsobená slučováním funkcí

4.11 Outlining operátorů

Outlining operátorů znamená oddelegování binárních operátorů do samostatných funkcí. Například výraz `x + 10` transformujeme na `plus(x, 10)` a vytvoříme novou deklaraci funkce plus, která bude implementovat funkcionalitu sčítání dvou operandů (přičemž identifikátor funkce plus může být – a měl by být – náhodně zvolený). Pokud tuto transformaci provedeme u všech binárních operátorů, značně tím znepřehledníme základní početní operace, kterých je v běžném kódu celá řada. Odolnost této transformace je samozřejmě relativně nízká, nicméně v kombinaci s dalšími obfuskačními technikami (zejména slučování funkcí v sekci 4.10) začne její odolnost sílit.

Implementace je přímočará a jednoduchá. Využijeme knihovnu estraverse a projdeme celý AST. Když narazíme na výraz E s nějakým binárním operátorem, zkontrolujeme, zda jsme pro tento operátor už nevytvořili odpovídající delegovanou funkci F. Pokud ne, vytvoříme delegovanou funkci F v globálním scopu. Ve výrazu E potom nahradíme objevený binární operátor za volání funkce F s odpovídajícími argumenty.

Jistě nás napadne, že outlining bychom mohli použít i na logické operátory. Zde však narazíme na zásadní problém. Uvažujme výraz `(false && x)`, kde x je v kódu dosud nedefinovaná proměnná. Takový výraz se vyhodnotí jako false a provádění kódu pokračuje bez problému dál. Kdybychom však výraz transformovali do tvaru `and(false, x)` (kde and je funkce implementující funkcionalitu operátoru `&&`), JavaScript nás vytrestá výjimkou ReferenceError, protože do funkce and předáváme jako argument nedefinovanou proměnnou. Analogicky bychom mohli zkonstruovat stejným způsobem problematický výraz i pro operátor `||`. Z tohoto důvodu od outliningu logických operátorů upustíme.

Při této transformaci narazíme na stejný problém jako při slučování funkcí v sekci 4.10 – totiž že umisťujeme nové deklarace funkcí do globálního scopu. Ani zde nepřistoupíme k řešení tohoto problému. Umisťování outlinovaných funkcí do dílčích scopů by způsobilo nabobtnání kódu (protože bychom namísto jedné deklarace pro každý operátor potřebovali deklarací několik) a zapříčinilo tak větší náklady a poškození minifikace. Navíc outlinování operátorů použitých v globálním scopu by nebylo možné.

Na závěr ještě zmíníme, že i při outlinování operátorů je potřeba dbát na to, abychom nové deklarace funkcí v globálním scopu neumístili před direktivu `'use strict';`.

4.12 Syntéza a pořadí vykonávání obfuskačních transformací

Pro kvalitu obfuskace nejsou podstatné jen použité techniky, ale i pořadí jejich provedení a četnost jejich provedení. Mělo by určitý smysl používat některé transformace vícekrát (například obfuskace řetězce provedené na už jednou obfuskovaný řetězec způsobí dvojí zakódování pomocí Base64), ale typicky se setkáme s tím, že pokud je deobfuskátor schopen transformaci zvrátit, bude ji pravděpodobně schopen zvrátit i v případě, že ji aplikujeme vícenásobně (jen bude muset provést stejný počet iterací deobfuskace, jaký byl počet našich iterací obfuskace). Opakováním obfuskační transformace tedy obvykle (ne vždy) mírně zvyšujeme potenci a zanedbatelně zvyšujeme odolnost, zato náklady většinou zvyšujeme tolikrát, kolikrát transformaci zopakujeme, tzn. tato praktika se typicky velmi negativně podepíše na minifikaci. Opakování obfuskačních transformací tedy budeme provádět jen tehdy, budeme-li k tomu mít dobrý důvod. Pořadí transformací zvolíme následovně:

1. Přejmenování proměnných – touto transformací zajistíme kromě její primární funkce i skutečnost, že všechny proměnné a funkce budou mít unikátní identifikátory nejen v rámci scopu, ale v rámci celého AST. To nám ujednoduší práci při dalších transformacích.

2. Inlining funkcí

3. Outlining příkazů do funkcí – tuto tranfromaci je potřeba provést až po inliningu funkcí. Kdybychom zvolili pořadí opačné, mohlo by snadno dojít k tomu, že část outlinovaných operátorů si nešikovně deobfuskujeme sami.

4. Outlining operátorů

5. Přeuspořádání parametrů funkcí – tuto transformaci je potřeba spustit až po outliningu operátorů, abychom randomizovali pořadí parametrů i v těchto outlinovaných funkcích.

6. Přeuspořádání deklarací funkcí – díky tomu, že tuto transformaci provádíme až po outliningu operátorů a funkcí, docílíme toho, že budou zamíchány i podpůrné funkce v globálním scopu, které jsme do AST vložili.

7. Slučování funkcí – jsou-li všechny podpůrné funkce vytvořeny a zamíchány, můžeme je nyní začít spojovat společně se všemi funkcemi, které

jsou v AST už od uživatele. Díky tomu nejen obfuskujeme původní kód, ale zároveň zvyšujeme odolnost transformací jako outlining.

8. Přeuspořádání parametrů funkcí podruhé – touto transformací zajistíme, že rozhodující proměnná bude v nově sloučených funkcích umístěna na náhodném místě (ostatní parametry jsou už v tuto chvíli randomizované).

9. Přeuspořádání příkazů v blocích, přičemž nejprve provedeme roztržení deklarací proměnných. Touto metodou zároveň mírně zvyšujeme odolnost inliningu funkcí, protože po provedení této operace se může inlinovaný kód lišit od funkce, jíž jsme inlinovali, čímž zahlazujeme stopy po jejich spojení.

10. Outlining příkazů do funkce eval – tuto transforamci chceme provést téměř až na samém konci, protože pro správný chod mnoha našich transformací předpokládáme nepřítomnost funkce eval v kódu.

11. Obfuskace literálů (které pochopitelně předchází převod zápisu MemberExpressions z tečkové syntaxe do syntaxe hranatých závorek). Zároveň se jedná o jedinou transformaci provedenou až po outliningu příkazů do funkce eval – zajistí nám, že argument volání evalu nebude triviálně rozklíčovatelný a výrazně tak zvýší potenci předchozí transformace.

Jak si lze všimnout, přeuspořádání parametrů funkcí se v našem seznamu vyskytuje na dvou různých místech. Mohli bychom být podezřívaví a namítnout, že k randomizaci veškerých parametrů všech funkcí by mělo stačit až druhé přeuspořádání parametrů. To je sice pravda, problémem však je transformace slučování funkcí. Kdybychom nepřeuspořádali parametry předtím, než funkce sloučíme, stane se to, že pro každou dvojici slučovaných funkcí A a B bude n-tý parametr funkce A reprezentován v nové funkci M stejným parametrem, jako n-tý parametr funkce B. To znamená, že kdyby reverzní inženýr například odhalil, že třetí parametr funkce M odpovídá druhému parametru funkce A, okamžitě by věděl i to, že třetí parametr funkce M odpovídá zároveň i druhému parametru funkce B (pokud ovšem funkce B přijímá alespoň dva parametry). Tuto informaci randomizací parametrů funkcí (které provedeme před sloučením funkcí) vymažeme.

Kapitola 5

Testování

Stejně jako každý software je i náš program potřeba otestovat. Testování proběhne v několika fázích. Ruční proprietární testování s vizuální kontrolou výstupního (obfuskovaného) zdrojového kódu a automatizované testování obfuskovaných JavaScriptových knihoven. Obojí si popíšeme v této kapitole.

5.1 Proprietární testování

V průběhu vývoje budeme testovat každou jednu transformaci na kratším jednoduchém kódu, který je k testování dané transformace vhodný (tzn. například obsahuje takové konstrukce, u kterých lze předpokládat, že by mohly při obfuskaci způsobit neočekávané chování). Vizuálně potom budeme kontrolovat kód vytvořený po aplikaci této transformace a budeme zkoumat:

- Zda transformace proběhla očekávaným způsobem, tj. zda je kód obfuskovaný (případně minifikovaný) tak, jak bychom očekávali.
- Zda při tranformaci nedošlo k takové úpravě kódu, která by způsobila změnu chování kódu (nebo která by byla jiným způsobem nežádoucí nebo nečekaná).

Obfuskované kódy lze samozřejmě také spustit a sledovat, zda dávají stejné výsledky, jako před obfuskací. Je nutné ale říci, že tento způsob testování sám o sobě (tj. bez vizuální kontroly výsledného kódu) není dostačující, protože opomíjí to hlavní – zda kód je a nebo není rozklíčovatelný. Zároveň toto testování neodhalí takové transformace, které sice na daném kódu nezpůsobí nesprávné chování, ale přesto jsou nějakým způsobem závadné.

Je vhodné tento druh testování provádět v průběhu vývoje a často – podchytíme tím spoustu potenciálních chyb už v zárodku a umožní nám to jistý vhled do toho, jak naše transformace fungují. Několikačetné testování je vhodné také proto, že v několika transformacích pracujeme s prvkem náhody

a je tedy možné, že případnou chybu odhalíme až na několikátý pokus. Stejným způsobem lze testovat i kombinace transformací, u nichž máme podezření, že by mohly zapříčinit problém (např. kombinace inlining a outlining funkcí).

5.2 Automatizované testování

Přestože proprietární testování je hodnotné a v mnoha směrech efektivní, jako takové není dostatečné. Vzhledem k tomu, že testy píše sám autor obfuskátoru, je pravděpodobné, že ve snaze vytvořit potenciálně problematické vstupy opomene právě ty konstrukty, které opomněl i při vývoji transformací. Navíc rozsah testovaných kódů nemůže být velký, má-li pokaždé dojít k osobní vizuální kontrole výstupu. Z toho důvodu přistoupíme ještě k sofistikovanějšímu testování.

Na internetu jsou dostupné rozsáhlé JavaScriptové knihovny, které lze stáhnout i s celou řadou předpřiravených testů určených k testování funkčnosti té které knihovny. Pokud původní kód knihovny obfuskujeme a nad touto obfuskovanou verzí spustíme připravené testy, můžeme výsledky těchto testů srovnat s výsledky testů spuštěnými nad původní verzí knihovny. Relativně spolehlivě tak budeme schopni určit problémy v našem programu.

Výhodou tohoto druhu testování je fakt, že nemusíme psát vlastní testy – lze využít tisíce již připravených testů, které správnost obfuskace ověří za nás. Efektivní vizuální kontrola kódu obfuskovaných rozsáhlých knihoven však není prakticky možná – v zásadě se totiž už jedná o reverzní inženýrství, které se obfuskací snažíme znemožnit, a (nezbytné) opakování tohoto procesu by vedlo k neúměrné časové náročnosti testování. Při ověřování budeme tedy spoléhat pouze na výstupy testů – nekontrolujeme tedy samotnou účinnost obfuskace (potenci a odolnost), ale pouze její negativní dopady na chování kódu.

Abychom mluvili konkrétněji, k testování nezávadnosti obfuskace využijeme knihovny jQuery a jQuery UI. Výsledky negativních testů zde nebudeme uvádět, protože jejich zkoumání vedlo k opravě chyb v obfuskátoru, které jsme zohlednili už i v textu předchozích kapitol. Uvádíme tedy pouze výsledky závěrečných testů, které probíhaly už na finální verzi našeho obfuskátoru.

5.2.1 Testování nad jQuery

Všechno nezbytné k sestavení a testování knihovny jQuery nalezneme například na GitHubu. Jak uvádí přímo dokumentace jQuery [38], knihovnu sestavíme příkazem `npm run build`. Testy potom spustíme pomocí lokálního serveru s podporou PHP (nejjednodušší je spustit příkaz `php -S localhost:8000` ve složce jquery a otevřít v prohlížeči stránku localhost:8000).

Na otevřené webové stránce se automaticky spustí testy (celkem 8035 testů) nad sestavenou knihovnou jQuery. Z důvodů, které nebudeme v rámci práce rozebírat, mohou některé z testů selhat už na neobfuskované verzi knihovny. Identifikátory takových testů je potřeba si poznamenat a jejich selhání

nebrat v potaz, až je budeme spouštět na obfuskované verzi. Následně naším programem obfuskujeme soubor jquery/dist/jquery.js a spustíme testy znovu (v prohlížeči je potřeba zaškrtnout checkbox „Load unminified“, aby se načetla námi obfuskovaná verze a ne minifikovaná verze vytvořená při sestavování knihovny).

Testování neobfuskované knihovny (o velikosti 266 KiB) trvalo v průměru 2967611 ms, přičemž vždy selhalo 96 nebo 101 testů. Deset postupných testů provedených na obfuskované verzi trvalo v průměru 3461883 ms, přičemž vždy selhalo 96, 97 nebo 101 testů. Obfuskovaná verze knihovny měla v průměru zhruba 570 KiB. Všechna testování pochopitelně probíhala na stejném stroji. Poznamenejme, že počet selhaných testů se liší o jednotky z důvodů nezávislých na funkčnosti testované verze knihovny (například nastavený timeout na konkrétní test).

5.2.2 Testování nad jQuery UI

Knihovna jQuery UI je dostupná rovněž z GitHubu [39]. Knihovna obsahuje několik samostatných souborů, tzv. „widgetů“. Zdrojové kódy těchto widgetů jsou dostupné v adresáři jquery-ui/ui/widgets. Pro každý widget existuje v adresáři jquery-ui/tests/unit vlastní sada testů (tyto jsou spustitelné v HTML souborech, které stačí otevřít v prohlížeči).

Testování proběhlo vždy jedenkrát na neobfuskované verzi a desetkrát na obfuskované, přičemž testované widgety byly tyto: checkboxradio, tooltip, spinner, accordion, slider, resizable, selectable, datepicker, droppable, sortable, draggable, autocomplete, selectmenu, controlgroup, button, dialog a menu. V obfuskovaných verzích vždy selhaly pouze ty testy, které selhaly i v neobfuskované verzi (vždy se jednalo o jednotky selhaných testů).

Widget	accordion	autocomplete	button	controlgroup
Původně KiB	16	17	12	9
Obfuskované KiB	39	19	28	17

Tabulka 5.1: Srovnání velikostí původních a obfuskovaných souborů (1)

Widget	datepicker	dialog	draggable	droppable	menu
Původně KiB	79	23	35	13	19
Obfuskované KiB	172	54	91	32	42

Tabulka 5.2: Srovnání velikostí původních a obfuskovaných souborů (2)

V tabulkách 5.1 a 5.2 můžeme pro představu vidět velikosti některých widgetů před a po obfuskaci. Lze odpozorovat, že po obfuskaci se typicky zvětší velikost souborů na více než dvojnásobek. Vidíme, že obfuskace má (podle očekávání) neblahý vliv na minifikaci, a to přesto, že jsme v našich transformacích zahrnuli i minifikační techniky. Zároveň se provádění programů očividně zpomalilo, což bylo taky očekávané, protože náklady většiny použitých transformací jsou nenulové.

Závěr

V práci jsme se zabývali analýzou problematiky obfuskace a minifikace. Vyhledali jsme známé techniky obfuskace a odhalili jejich vztah k JavaScriptu. Prozkoumali jsme knihovny estools a zjistili, jakým způsobem nám mohou pomoci při implementaci vlastního obfuskátoru. Zjistili jsme, jakým způsobem lze klasifikovat a určovat účinnost obfuskačních technik.

Za použití knihoven estools jsme navrhli a implementovali program v NodeJS, který v promyšleném pořadí uplatňuje jedenáct minifikačních a obfuskačních technik na JavaScriptový kód. Zaměřili jsme se přitom především na ztížení automatické deobfuskace (tj. zajištění co nejvyšší odolnosti prováděných transformací). Tuto implementaci minifikace a obfuskace jsme patřičně otestovali.

Zjistili jsme, že obfuskace je široké téma a tedy i náš obfuskátor lze dále rozšiřovat. Ukázalo se, že další obfuskační techniky mohou nejen prohlubovat obfuskaci kódu, ale i zesilovat odolnost předchozích a následujících tranformací, proto má rozšiřování obfuskátoru o další transformace obzvláštní význam. Tato možná rozšíření přenecháme budoucí práci.

Literatura

[1] Komunita YUI: *Minification v Obfuscation [online]*. [cit. 2018-04-11]. Dostupné z: https://yuiblog.com/blog/2006/03/06/minification-v-obfuscation/

[2] jefflunt: *The case for code obfuscation? [online]*. [cit. 2018-04-11]. Dostupné z: https://softwareengineering.stackexchange.com/questions/129296/the-case-for-code-obfuscation/129300

[3] Ceccato, M.; Penta, M. D.; Nagra, J.; aj.: *The Effectiveness of Source Code Obfuscation: an Experimental Assessment [online]*. University of Sannio, [cit. 2018-04-11]. Dostupné z: http://www.rcost.unisannio.it/mdipenta/icpc09-tr.pdf

[4] Choi, Y.; Kim, T.; Choi, S.: Automatic Detection for JavaScript Obfuscation Attacks in Web Pages through String Pattern Analysis [online]. *International Journal of Security and Its Application*, Květen 2010, [cit. 2018-04-12]. Dostupné z: https://softwareengineering.stackexchange.com/questions/129296/the-case-for-code-obfuscation/129300

[5] Udupa, S. K.; Debray, S. K.; Madou, M.: Deobfuscation, Reverse Engineering Obfuscated Code [online]. *12th Working Conference on Reverse Engineering*, Leden 2006, doi:10.1109/WCRE.2005.13, [cit. 2018-04-14]. Dostupné z: http://www2.cs.arizona.edu/~debray/Publications/unflatten.pdf

[6] Lu, G.; Debray, S.: Automatic Simplification of Obfuscated JavaScript Code: A Semantics-Based Approach [online]. *2012 IEEE Sixth International Conference on Software Security and Reliability (SERE)*, Srpen 2012, doi:10.1109/SERE.2012.13, [cit. 2018-04-14]. Dostupné z: https://pdfs.semanticscholar.org/1e4b/8045eec4790d2a92e11d8465d88b0b0c1be0.pdf

[7] Bendersky, E.: *Eli Bendersky's website [online]*. [cit. 2018-04-13]. Dostupné z: `https://eli.thegreenplace.net/2009/02/16/abstract-vs-concrete-syntax-trees/`

[8] Bazon, M.: *Lisperator.net [online]*. [cit. 2018-04-14]. Dostupné z: `http://lisperator.net/uglifyjs/spidermonkey`

[9] JS Foundation: *ESLint [online]*. [cit. 2018-04-14]. Dostupné z: `https://eslint.org/blog/2014/12/espree-esprima`

[10] Yusuke Suzuki a další přispěvatelé: *estools / estraverse [online]*. [cit. 2018-04-14]. Dostupné z: `https://github.com/estools/estraverse`

[11] Yusuke Suzuki a další přispěvatelé: *estools / escope [online]*. [cit. 2018-04-14]. Dostupné z: `https://github.com/estools/escope`

[12] Yusuke Suzuki a další přispěvatelé: *estools / esmangle [online]*. [cit. 2018-04-14]. Dostupné z: `https://github.com/estools/esmangle`

[13] Yusuke Suzuki a další přispěvatelé: *estools / escodegen [online]*. [cit. 2018-04-14]. Dostupné z: `https://github.com/estools/escodegen`

[14] Yusuke Suzuki a další přispěvatelé: *estools / esvalid [online]*. [cit. 2018-04-14]. Dostupné z: `https://github.com/estools/esvalid`

[15] Collberg, C.; Thomborson, C.; Low, D.: A Taxonomy of Obfuscating Transformations. *Department of Computer Science, The University of Auckland, New Zealand, Technical report*, Leden 1997.

[16] keparo: *How can I obfuscate (protect) JavaScript? [online]*. [cit. 2018-04-14]. Dostupné z: `https://stackoverflow.com/questions/194397/how-can-i-obfuscate-protect-javascript`

[17] Jscrambler: *Jscrambler [online]*. [cit. 2018-04-14]. Dostupné z: `https://jscrambler.com`

[18] Serafim, T.: *JavaScript Obfuscator [online]*. [cit. 2018-04-14]. Dostupné z: `https://javascript-obfuscator.org/`

[19] CuteSoft Components Inc: *Javascript Obfuscator [online]*. [cit. 2018-04-14]. Dostupné z: `http://javascriptobfuscator.com`

[20] Rauschmayer, A.: *Speaking JavaScript*. O'Reilly Media, Inc., Březen 2014, ISBN 1449365035.

[21] Mozilla: *this - JavaScript — MDN [online]*. [cit. 2018-05-01]. Dostupné z: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this`

[22] Mozilla: *eval() - JavaScript — MDN [online]*. [cit. 2018-04-05]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/eval

[23] Rauschmayer, A.: JavaScript's with statement and why it's deprecated. *2ality - JavaScript and more [online]*, Červen 2011, [cit. 2018-04-05]. Dostupné z: http://2ality.com/2011/06/with-statement.html

[24] Mozilla: *Function.name - JavaScript — MDN [online]*. [cit. 2018-04-06]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/name

[25] W3Schools: *JavaScript Reserved Words [online]*. [cit. 2018-04-06]. Dostupné z: https://www.w3schools.com/js/js_reserved.asp

[26] Flanagan, D.: *JavaScript: The Definitive Guide, 6th Edition*. O'Reilly Media, Inc., Květen 2011, ISBN 9781449393854.

[27] Mozilla: *label - JavaScript — MDN [online]*. [cit. 2018-04-07]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/label

[28] Xu, W.; Zhang, F.; Zhu, S.: *The power of obfuscation techniques in malicious JavaScript code: A measurement study*. IEEE, Říjen 2012, ISBN 978-1-4673-4880-5, 6 s.

[29] The Internet Society: *The Base16, Base32, and Base64 Data Encodings [online]*. doi:10.17487/RFC3548, [cit. 2018-04-08]. Dostupné z: https://tools.ietf.org/html/rfc3548

[30] Mozilla: *Base64 encoding and decoding - Web APIs — MDN [online]*. [cit. 2018-04-09]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/API/WindowBase64/Base64_encoding_and_decoding

[31] Andrew: *JavaScript list of exceptions [online]*. [cit. 2018-04-09]. Dostupné z: https://stackoverflow.com/questions/3656854/javascript-list-of-exceptions

[32] Mozilla: *URIError - JavaScript — MDN [online]*. [cit. 2018-04-09]. Dostupné z: https://developer.mozilla.org/cs/docs/Web/JavaScript/Reference/Global_Objects/URIError

[33] Mozilla: *TypeError - JavaScript — MDN [online]*. [cit. 2018-04-09]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/TypeError

[34] Mozilla: *ReferenceError - JavaScript — MDN [online]*. [cit. 2018-04-09]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/ReferenceError

[35] Mozilla: *Strict mode [online].* [cit. 2018-04-25]. Dostupné z: https://developer.mozilla.org/cs/docs/Web/JavaScript/Reference/Strict_mode

[36] Microsoft: *Developer Network [online].* [cit. 2018-04-09]. Dostupné z: https://msdn.microsoft.com/cs-cz/library/87dw3w1k(v=vs.94).aspx

[37] Neckář, J.: *Algoritmy.net [online].* [cit. 2018-04-09]. Dostupné z: https://www.algoritmy.net/article/43610/Fisher-Yatesuv-algoritmus

[38] The jQuery Foundation: *jQuery JavaScript Library [online].* [cit. 2018-05-09]. Dostupné z: https://github.com/jquery/jquery

[39] The jQuery Foundation: *The official jQuery user interface library. [online].* [cit. 2018-05-09]. Dostupné z: https://github.com/jquery/jquery-ui

Příloha A

Seznam použitých zkratek

AST Abstraktní syntaktický strom

Příloha B

Obsah přiloženého CD

- `src`
  - `impl` ..... zdrojové kódy implementace
    - `index.js` ..... ústřední zdrojový soubor připravený ke spuštění
    - `transformations` ..... implementace jednotlivých transformací
    - `node_modules` ..... využívané externí knihovny
    - `tests` ..... jednoduché testovací soubory
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `BP_Hanák_Samuel_2018.pdf` ..... text práce ve formátu PDF