VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

MOŽNOSTI NORMALIZACE PROGRAMŮ JAZYKA JAVASCRIPT PŘI VYHLEDÁVÁNÍ ZRANITELNOSTÍ

JAVASCRIPT CODE NORMALIZATION DURING DETECTION OF VULNERABILITIES

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **LUKÁŠ HAVLÍČEK**
AUTHOR

VEDOUCÍ PRÁCE **Ing. LIBOR POLČÁK, Ph.D.**
SUPERVISOR

BRNO 2023

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Zadání bakalářské práce

144947

Ústav: Ústav informačních systémů (UIFS)
Student: **Havlíček Lukáš**
Program: Informační technologie
Specializace: Informační technologie
Název: **Možnosti normalizace programů jazyka JavaScript při vyhledávání zranitelností**
Kategorie: Web
Akademický rok: 2022/23

Zadání:

1. Nastudujte si možnosti minifikace zdrojových kódů v jazyce JavaScript, srovnejte je s obfuskačními nástroji. Zaměřte se také na možnosti různých zápisů stejných literálů (např. různé reprezentace čísel, využití různých druhů uvozovek apod.).
2. Nastudujte diplomovou práci Vojtěcha Randýska, rozšíření vzniklé v této práci a implementované normalizační techniky.
3. Navrhněte vylepšení rozšíření o další techniky normalizace zdrojového kódu.
4. Po konzultaci s vedoucím práce návrh implementujte.
5. Implementaci otestujte, inspirujte se testy provedenými Vojtěchem Randýskem.
6. Svou práci vyhodnoťte a navrhněte možná pokračování.

Literatura:

- Randýsek, Vojtěch. Detekce kódu v jazyce JavaScript se známými bezpečnostními chybami. Brno, 2022. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií.
- Skolka, P.; Staicu, C.-A.; Pradel, M.: Anything to Hide? Studying Minified and Obfuscated Code in the Web. In The World Wide Web Conference, WWW '19, New York, NY, USA: Association for Computing Machinery, 2019, ISBN 9781450366748, str. 1735–1746, doi:10.1145/3308558.3313752.
- Blanc, G.; Miyamoto, D.; Akiyama, M.; a kol.: Characterizing Obfuscated JavaScript Using Abstract Syntax Trees: Experimenting with Malicious Scripts. In 2012 26th International Conference on Advanced Information Networking and Applications Workshops, 2012, str. 344–351, doi:10.1109/WAINA.2012.140.

Při obhajobě semestrální části projektu je požadováno:
První tři body zadání včetně textové zprávy.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Polčák Libor, Ing., Ph.D.**
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1.11.2022
Termín pro odevzdání: 10.5.2023
Datum schválení: 25.10.2022

Abstrakt

Tato práce se zabývá problematikou minifikace, obfuskace JavaScryptu a normalizací abstraktních syntaktických stromů pro rozšíření prohlížeče implementované v rámci diplomové práce pana Randýska. Byly nastudovány nástroje a techniky minifikace i obfuskace JavaScryptu. Tyto informace byly využiti při návrhu a implementaci normalizace abstraktních syntaktických stromů. Stromy jsou využívány v rozšíření prohlížeče Chrome, které detekuje a opravuje JavaScryprový kód. Normalizace jsem otestoval jednotkovými a integračními testy. Otestoval jsem i rozšíření pro detekci chyb, kde jsem detekoval 125 zranitelností po průchodu 1000 webových stránek.

Abstract

This thesis deals with the minification, obfuscation of JavaScript and normalization of abstract syntactic trees for browser extensions implemented in Mr. Randýsek's thesis. The tools and techniques of both JavaScript minification and obfuscation have been studied. This information was used in the design and implementation of abstract syntactic tree normalization. The trees are used in a Chrome browser extension that detects and corrects JavaScript code. I tested the normalizations with unit and integration tests. I also tested the vulnerability detection extension, where I detected 125 vulnerabilities on 1000 websites.

Klíčová slova

JavaScrypt, minifikace, obfuskace, deobfuskace, metriky obfuskace, abstraktní syntaktický strom, rozšíření webového prohlížeče, normalizace

Keywords

JavaScript, minification, obfuscation, deobfuscation, obfuscation metrics, abstract syntax tree, web browser extension, normalization

Citace

HAVLÍČEK, Lukáš. *Možnosti normalizace programů jazyka JavaScript při vyhledávání zranitelností.* Brno, 2023. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Libor Polčák, Ph.D.

Možnosti normalizace programů jazyka JavaScript při vyhledávání zranitelností

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Libora Polčáka, Ph.D. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Lukáš Havlíček
4. května 2023

Poděkování

Chtěl bych poděkovat především mému vedoucímu Ing. Liborovi Polčákovi, Ph.D., který mi poskytl odborné vedení a ochotu při konzultacích.

Obsah

Seznam obrázků

Kapitola 1

Úvod

Pomocí JavaScriptu lze implementovat na webové stránky složité funkce. Pokud stránka provádí více, než že zobrazuje pouze statické informace - zobrazení 2D/3D grafiky, včasné aktualizace obsahu, interaktivní mapy atd. s velkou pravděpodobností byl na této stránce použit právě JavaScript [5].

Vývojáři se snaží, aby s napsaným kódem mohli pracovat i další vývojáři. Jejich cílem tedy je, aby výsledný kód byl přehledný a srozumitelný. K tomu je zapotřebí využívat například komentáře, zalamování řádků a bílé znaky. Webový prohlížeč tyto znaky k provedení kódu nepotřebuje, naopak ho zpomalují. Aby se prohlížeč s těmito znaky nemusel zaobírat, lze se na kód aplikovat minifikace, která nadbytečné znaky odstraní [1]. Minifikováním JavaScript souboru se sníží jeho velikost a tím se také sníží čas stahování, jelikož soubor nebude tak velký. Tímto se může navýšit celková rychlost chodu webové stránky [10].

Pomocí vývojářské konzole lze ve webovém prohlížeči snadno procházet JavaScriptový kód, číst ho a měnit. Tajný kód by se měl uchovávat na backendovém serveru, ale ne vždy je to možné. Obfuskace slouží pro přeměnu snadno čitelného a jednoduchého kódu na kód obtížný pro pochopení a zpětnou analýzu [8].

Vojtěch Randýsek ve své diplomové práci navrhl a implementoval rozšíření pro webový prohlížeč. Toto rozšíření má za úkol detekovat známé bezpečnostní chyby JavaScriptového kódu a následně je opravit. Rozšíření počítá s minifikovaným či obfuskovaným kódem jenom minimálně.

Následující kapitola 2 se zaobírá problematikou minifikace, metodami používanými při minifikaci a také nástroji používané k minifikování. Kapitola 3 popisuje metody obfuskování, nástroje pro provedení obfuskace a samotné téma obfuskace. V kapitole 4 je blíže popsáno rozšíření pro webový prohlížeč pana Randýska. Tato část se zabývá návrhem, implementací rozšíření a použitými normalizačními technikami pro minifikaci a obfuskaci. Praktická část začíná kapitolou 5, ve které analyzuji syntaktické stromy kódů pro nastudované techniky minifikace a obfuskace, nebo techniky které jsem objevil během vývoje a testování. Dále popisuji vytváření normalizací pro zmíněné stromy a implementaci těchto normalizací do webového rozšíření pana Randýska. Kapitola 6 praktickou část končí, jedná se o kapitolu, kde popisuji testování vzniklých normalizací. Kapitola 7 shrnuje celou práci, návrh, implementaci, výsledky testování, porovnání s původní prací pana Randýska a také shrnuje slabiny práce a možná rozšíření.

Kapitola 2

Minifikace

Proces minifikace slouží k minimalizaci kódu webových stránek a skriptů, kde dochází k odstranění zbytečných znaků z celého kódu bez změny jeho funkčnosti. Data nepotřebná pro samotné provedení souboru jsou odstraněna.

K minifikaci dochází převážně po dokončení kódu a to převážně před nasazením aplikace. Místo zaslání plné verze webové stránky, dostane prohlížeč snadný přístup k minifikované verzi webové stránky. Uživatel tak může pracovat s webovou stránkou a přitom mít přístup k veškerému obsahu. Snižuje se doba odezvy a zkracuje se doba načítání.

Minifikace lze použít u jazyků jako jsou HTML, CSS a JavaScript. Běžný vývojář píše kód, taky aby s ním mohli pracovat i další vývojáři. To znamená, že se snaží aby výsledný kód byl přehledný a srozumitelný. Problém nastává, když vývojář napíše kód, který je nepřehledný a dá se v něm těžce orientovat. Aby se těmto problémům předcházelo je potřeba využívat například zalamování řádků, komentáře a bílé znaky. Webový prohlížeč tyto nadbytečné znaky před provedením kódu nepotřebuje. Aby se těmito znaky prohlížeč nemusel zaobírat, aplikuje se minifikace, která tyto nezbytné znaky pro prohlížeč odstraní.

Nepotřebné části kódu, které minifikace běžné odstraňuje, dělají soubor zbytečně velký. V případě, že je minifikace zavedena, dochází k zmenšení velikosti souboru a k rychlejší odezvě webu [1].

```
const variable = "Variable";

function print() {
    console.log(variable);
};

print(); // "Variable"
```

Výpis 2.1: Javascript kód před použití minifikace

Na výpisu 2.1 je jednoduchý `JavaScriptový` kód, kde je deklarovaná proměnná `variable`, deklarovaná funkce `print`, která je následně volaná. Nachází se zde i komentář.

V tomto kódu jsou vidět středníky na konci některých řádků. Ty jsou v `JavaScriptu` používány pro ukončení příkazu, což pomáhá interpretu rozlišovat jednotlivé příkazy.

```
const variable="Variable"function print(){console.log(variable);};print();
```

Výpis 2.2: Javascript kód po použití minifikace

Na výpisu 2.2 je minifikovaná verze stejného kódu, který byl již zmíněný dříve. Obě verze kódu budou mít po spuštění stejný výsledek. Rozdílem je, že druhý kód není na rozdíl od prvního jednoduše čitelný.

Verze na výpisu 2.2 je vhodná pro produkci, verze 2.1 je vhodná pro vývoj. První verze kódu má velikost přibližně 100 bytů, zatímco druhá verze 75 bytů. Toto sice není markantní rozdíl, ale při velkých souborech je již rozdíl znatelný.

Velikost souborů

Proč je zmenšení souborů tolik důležité a chtěné? Čím větší zdrojové soubory budou, tím déle bude trvat jejich stažení. Minifikováním JavaScript souborů se sníží čas stahování, jelikož soubory nebudou tak velké.

Při příchodu na URL adresu v prohlížeči načte prohlížeč .html soubor, který načte propojené CSS a JavaScript soubory. Pokud se vše provede správně, zobrazí se stránka s nastylizovanými prvky a funkčními interakcemi, které jsou prováděny pomocí JavaScript souborů. Při přijímání požadovaných prostředků mohou velké soubory celý proces značně zpomalit. Proces může zrychlit vysoká rychlost internetu uživatele, kterou provozovatel stránek není schopen ovlivnit, ale je schopen ovlivnit množství dat, které potřebuje uživatel stáhnout.

Menší velikost zdrojových souborů zlepšuje počáteční dobu parsování. Po načtení JavaScript souboru prohlížečem, začne parsování souboru. Při parsování se kód prochází řádek po řádku, kontroluje se syntaktická správnost a ignorují se bílé znaky a komentáře.

Pokud vše proběhne v pořádku, kód se přeloží do strojového kódu, kterému prohlížeč rozumí. Pokud ne, zobrazí se chyba.

Čím je větší velikost zdrojového souboru, tím déle bude trvat jeho kontrola. Tudíž menší, minifikované soubory, zrychlují čas počátečního parsování [10].

2.1 Jak minifikovat JavaScript kód

Existuje několik způsobů pro minifikaci JavaScript kódu a každý z nich vyžaduje jiný přístup. Jedním z nich je manuální minifikace. K manuálnímu provedení minifikace je potřeba soubor s JavaScript kódem a textový editor. Následně stačí odstranit veškeré bílé znaky a komentáře, které se v souboru nacházejí. S urychlením tohoto procesu mohou pomoci editory, které podporují regulární výrazy. Varianta odstraňování veškerých bílých znaků a komentářů není v některých případech vhodná. U menších souborů by tento proces mohl trvat pár minut, ale při velkých souborech je tato metoda nepraktická. Veškeré mazání by trvalo příliš dlouho a mohla by se také odmazat důležitá část souboru.

Další možností je instalace minifikačního nástroje, který lze používat pomocí příkazové řádky. Při používání je zapotřebí zadat do argumentů pouze soubor určený pro minifikaci a výstupní soubor. O samotnou minifikaci se postará příslušný nástroj. Některé nástroje není třeba instalovat a lze je používat přímo v prohlížeči [11].

2.2 Nástroje pro minifikaci

2.2.1 JSMin

Jeden z nejpoužívanějších minifikačních nástrojů je JSMin. Jedná se o nástroj a knihovnu pro minifikaci v příkazovém řádku. Stačí ho nainstalovat jako globální skript, který od-

straní veškeré nechtěné bílé znaky a komentáře. JSMin dokáže zmenšit velikost souboru až o 50 % [11].

JSMin je filtr který upravuje nebo vynechává některé znaky. Tím ale nemění chování programu.

Nejprve nahrazuje carriage returns ($\backslash r$) za znaky nový řádek ($\backslash n$). Ostatní řídící znaky (včetně tabulátorů) nahrazuje mezerami. Řetězce mezer jsou nahrazeny jednou mezerou. Všechny za sebou jdoucí řádky jsou vyměněny za jeden řádkový znak. Nahrazuje řádkové komentáře ($\backslash\backslash$) za znak nového řádku ($\backslash n$). Vymění víceřádkové komentáře ($\backslash * *\backslash$) za mezeru.

JSMin nevynechává mezeru pokud se před nebo za ní nachází znak, který není ASCII, nebo písmenem či číslicí které jsou součástí ASCII. Mezera také není vynechána pokud jej předchází nebo následuje jeden z těchto znaků: \, $, _.

Znak nového řádku (\n) je vynechán pokud předchází znaku, který není ASCII, ASCII písmeno či číslice nebo jednomu z následujících znaků: \, $, _, {, [, +, -. A pokud je umístěn za jiným než ASCII znakem nebo za ASCII písmenem či číslicí nebo za jedním z těchto znaků: \, $, _, {, [, +, -, ", '.

JSMin nemodifikuje literály regulárních výrazů a řetězce v uvozovkách. Provádí *uglifikaci*, ale neprovádí *obfuskaci* [3].

Kódovací sada

JSMin vyžaduje kódování vstupního programu UTF-8 nebo ASCII. S jiným kódováním nemusí pracovat správně, to ale nástroj (sám) nekontroluje.

Upozornění

JSMin funguje jednosměrně, jakmile je minifikace hotová, nelze proces vrátit. Je tedy velmi důležité si uchovat původní soubor.

Je třeba nevkládat řídící znaky do uvozeného řetězce a použít zápis \xhh, jelikož řídící znaky jsou zaměněny za mezery nebo konce řádků.

Nutno dávat pozor na používání sekvencí znaků + nebo -. JSMin může zaměnit jejich význam. Například změní a + ++b na a+++b, což je interpretováno jako a++ + b a to není požadovaný tvar. Je možné se tomu vyvarovat za pomocí uvozovek: a + (++b). Nejvhodnější je se sekvencím + a - zcela vyvarovat.

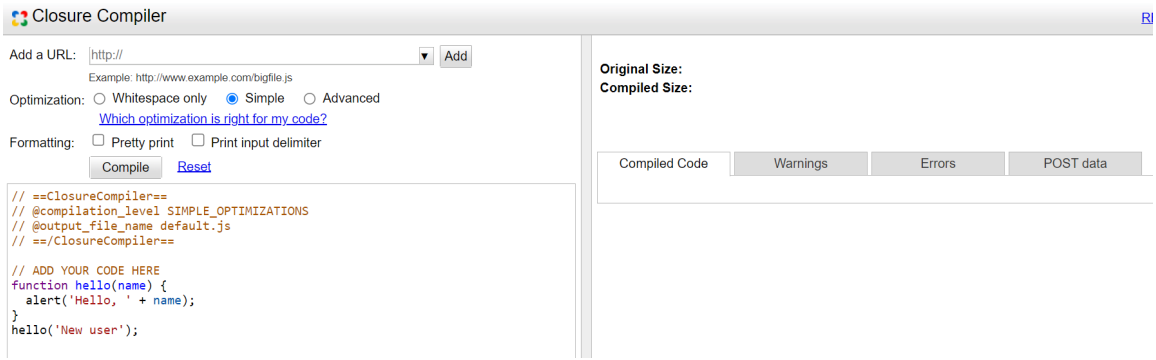
Chyby

JSMin dokáže detekovat čtyři chyby v kódu:

- neukončený komentář
- neukončený řetězec
- neukončen regulární výraz
- neukončená sada v regulárním výrazu [3]

2.2.2 Google Closure Compiler

`Google Closure Compiler` [1] lze spustit z příkazového řádku aplikace nebo prostřednictvím webové služby.



Obrázek 2.1: Google Closure Compiler

Pomocí webové služby lze kód zkompilovat pomocí odkazu na zdrojový soubor nebo vložením kódu do určeného místa na webu. Zminifikovaný kód se zobrazí v pravé části obrazovky [1].

Nástroj nabízí tři úrovně kompilace, od jednoduchého odstranění nechtěných znaků po agresivní změnu kódu.

WHITESPACE_ONLY

Úroveň kompilace `WHITESPACE_ONLY` odstraňuje zalomení řádků, nadbytečné mezery, komentáře, také maže nadbytečnou interpunkci (závorky, středníky) a další bílé znaky. Výstupní JavaScript je funkčně shodný se zdrojovým. Tato možnost kompilace provádí nejmenší kompresi ze všech tří možností.

SIMPLE_OPTIMIZATION

Jedná se o výchozí úroveň kompilace. Možnost `SIMPLE_OPTIMALIZATION` v základu provádí to stejné jako předchozí úroveň `WHITESPACE_ONLY`. Ale realizuje i změny uvnitř funkcí a výrazů, dochází k přejmenování lokálních proměnných a parametrů funkcí na kratší názvy. Kratší názvy výrazně pomáhají zmenšit velikost kódu. Jelikož `SIMPLE_OPTIMALIZATION` přejmenovává jenom symboly lokálních funkcí, nedochází k zásahům do interakce mezi kompilovaným souborem a ostatními `JavaScript` soubory.

ADVANCED_OPTIMALIZATION

`ADVANCED_OPTIMALIZATION` provádí stejné transformace jako již zmíněné předchozí metody. Pro dosažení nejvyšší komprese přidává řadu agresivnějších globálních transformací. Aby toto bylo možné, `ADVANCED_OPTIMALIZATION` má předpoklady o kompilovaném kódu. Pokud tyto předpoklady nejsou splněny, `ADVANCED_OPTIMALIZATION` vytvoří nefunkční kód.

Například kompilování pomocí `ADVANCED_OPTIMALIZATION` nemusí fungovat, pokud nejsou podniknuty kroky k zajištění `interoperability`. Pokud nejsou externí funkce a vlastnosti

[1] https://closure-compiler.appspot.com/home

označeny, `ADVANCED_OPTIMALIZATION` je nevhodně přejmenuje a budou v nesouladu s názvy v externím kódu.

Porovnání výstupů `SIMPLE_OPTIMIZATION` a `ADVANCED_OPTIMALIZATION` pro následující kód:

```
function unusedFunction(note) {
  alert(note['text']);
}

function displayNoteTitle(note) {
  alert(note['title']);
}

var flowerNote = {};
flowerNote['title'] = "Flowers";
displayNoteTitle(flowerNote);
```

Výpis 2.3: Closure Compiler vstupní kód

Výstup kompilace s `SIMPLE_OPTIMIZATION`:

```
function unusedFunction(a){alert(a.text)}function displayNoteTitle
(a){alert(a.title)}var flowerNote={title:"Flowers"};displayNoteTit
le(flowerNote);
```

Výpis 2.4: Closure Compiler SIMPLE_OPTIMALIZATION

Výstup kompilace s `ADVANCED_OPTIMIZATION`:

```
alert("Flowers");
```

Výpis 2.5: Closure Compiler ADVANCED_OPTIMALIZATION

Druhý výsledek je mnohem menší. `ADVANCED_OPTIMIZATION` provádí:

- *Dead code removal*: Byla odstraňena funkce *unusedFunction()*, jelikož nikdy nebyla volána.
- *More aggressive renaming*: Byla přejmenována globální proměnná *flowerNote*.
- *Function inlining*: Funkce *displayNoteTitle()* byla nahrazena jednou funkcí *alert()*, která tvoří tělo funkce. Nahrazení volání funkce jejím tělem se nazývá *inlining*. Překladač *Closure Compiler* vyhodnocuje, zda lze *inlining* bezpečně provést. *ADVANCED_OPTIMIZATION* podporuje *inlining* konstant a některých proměnných. [6]

Kapitola 3

Obfuskace

Obfuskace kódu je metoda, která slouží pro přeměnu jednoduchého a snadno čitelného kódu na nový kód, který je záměrně obtížný na pochopení a zpětnou analýzu, jak pro počítač, tak pro lidi.

Obfuskace se používá v různých programovacích jazycích. Zejména v jazycích `C` a `C++`. Nejvíce je mezi vývojáři oblíbená u jazyka `JavaScript`, jelikož zdrojový soubor ukrývá skutečné činnosti programu, v případě `C/C++` je zdrojový soubor převeden do binárního souboru.

`JavaScript` je interpretovaný jazyk, tudíž pro jeho čtení, interpretaci a spuštění je nutný interpret v prohlížeči na straně klienta. To také znamená, že kdokoliv může pomocí debuggeru ve vývojářské konzoli prohlížeče snadno procházet `JavaScript` kód a dle libosti ho číst a měnit [2].

Podle základů zabezpečení aplikací víme, že tajný kód má být uchován v důvěryhodném prostředí, jako je backendový server, ne vždy je to možné. Pokud společnosti uchovávají důležitou logiku na straně klienta, je to obvykle proto, že tento kód reálně nemohou uchovat na straně serveru.

Jedním z důvodů může být právě neexistující backend. Dalším případem může být situace, kdy na straně klienta musí běžet kód související s uživatelským prostředím. Nejčastějším důvodem je ale výkon. Volání serveru zabírá čas a pokud se jedná o službu, kde je výkon klíčový, ukládání `JavaScript` kódu na server není vhodné

Při ochraně citlivých dat v kódu není vhodné spoléhat pouze na obfuskaci. V závislosti na případu použití by se obfuskace měla používat jako doplněk správných bezpečnostních postupů [8].

3.1 Obfuskace JavaScriptu

Jak už bylo zmíněno, `obfuskace` je řada transformací kódu, které mění jednoduchý a snadno čitelný kód na velmi obtížně pochopitelnou verzi kódu. To platí i u `JavaScriptu`.

Na rozdíl od šifrování, kde je potřeba heslo, které je potřeba pro dešifrování, v případě obfuskace v jazyce `JavaScript` neexistuje žádný dešifrovací klíč. Šifrování `JavaScriptu` na straně klienta by bylo zbytečné. Pokud by existoval dešifrovací klíč, který by byl zapotřebí poskytnout prohlížeči, stal by se tento klíč kompromitující a kód by se mohl stát jednoduše přístupným.

Při použití obfuskace může prohlížeč přistupovat k obfuskovanému `JavaScript` kódu, může kód číst a interpretovat stejně snadno jako původní neobfuskovaný kód. Ačkoli ob-

fuskovaný kód vypadá úplně jinak jakožto původní, v prohlížeči vygeneruje úplně stejný výstup.

3.2 Techniky obfuskace

Jelikož cílem obfuskace je skrýt části `JavaScript` kódu, které by mohli být cílem konkurence nebo útočníků, je důležité pochopit, že je potřeba obfuskovat veškerá data v kódu. Pokud budou skryty `proměnné`, `objekty` a `řetězce`, ztíží se tím možnost pochopit, jaké typy dat se v kódu nachází.

Skrývaní dat je jedna z několika věcí, které obfuskace provádí. Silná obfuskace také zakrývá rozložení a tok programu a zahrnuje optimalizační techniky.

Obvykle se zaměřuje na:

- Identifikátory
- Funkce
- Výroky
- Predikáty
- Logické literály
- Regulární výrazy
- Tok řízení programu

Nejčastější techniky obfuskace `JavaScriptu` jsou:

- Kódování
- Změny pořadí
- Rozdělení
- Přejmenování
- Skrytí logiky [8]

3.3 Logické literály na výrazy

Logické výrazy mohou nabývat pouze jednu z hodnot `True` nebo `False`. Cílem obfuskace je přeměnit logické literály na výrazy, které vracejí stejnou logickou hodnotu, ale jsou obtížněji pochopitelné. Na následujících výpisech je uveden příklad [7].

`Příklad:`

Původní kód:

```
true || false;
```

Výpis 3.1: Logické lierály před obfuskací

Kód po obfuskaci:

```
!!1 || !{}
```

Výpis 3.2: Logické lierály po obfuskaci

3.4 Řetězce

3.4.1 Znak na ternární operátor

Řetězec je posloupnost znaků používána k reprezentaci textu. S touto transformací je pro člověka složitější vyhodnotit jednoznakové statické řetězcové hodnoty. Při každém použití je generován náhodný postup, následně kód pokaždé vypadá jinak.
`Příklad:`
Původní kód:

```
var htmlTags = ['a', 'div', 'p'];
```

Výpis 3.3: Znak na ternární operátor před obfuskací

Kód po obfuskaci:

```
var htmlTags = [
  665.01 > 6770 ? 'Q' : (2170, 1050) !== 805 ? (9480, 813.32) >= 5227 ?
  754.03 : 'a' : 7.26e+3,
  'div',
  (8140, 1590) === (7290, 3300) ? 'z' : (602.65, 8544) < 2637 ? (704, 4830)
  == 4240 ? (0xf25, 3.18e+3) : (false, 'e') : 'p'
];
```

Výpis 3.4: Znak na ternární operátor po obfuskaci

Po této transformaci je první a poslední pozice `htmlTags` změněna, ale druhá je stále stejná, jelikož se jedná o řetězec s více než jedním znakem.

Pokud je cílem ztransformovat i prostřední pozici, je potřeba transformaci zkombinovat s rozdělením řetězce Section.2.4.4 maximální silou. Z řetězce `div` se stanou tři jednoznakové řetězce, kde už transformace `znak na ternární operátor` lze aplikovat [7].
`Příklad:`
Původní kód:

```
var htmlTags = ['a', 'div', 'p'];
```

Výpis 3.5: Znak na ternární operátor s rozdělením řetězce před obfuskací

Kód po obfuskaci:

```
var htmlTags = [
  665.01 > 6770 ? 'Q' : (2170, 1050) !== 805 ? (9480, 813.32) >= 5227 ?
  754.03 : 'a' : 7.26e+3,
  ((1300, 103.65) >= (458, 630.33) ? ('k', false) : 307.7 !== 568.74 ?
  562 < (400.31, 4441) ? 'd' : (814.34, 120.78) : 0xffb) + ((2813, 612.26)
  !== 6282 ? 'i' : 377.47 < 5166 ? 266.97 != 9030 ? 120.66 : 6.37e+3 :
  (8.60e+3, 'G')) + (625.02 != (989.8, 6173) ? 'v' : 4940 === 654.38 ?
  (true, 'H') : (9825, 2740) != 35 ? ('B', 8.85e+3) : 'D'),
  (8140, 1590) === (7290, 3300) ? 'z' : (602.65, 8544) < 2637 ? (704, 4830)
  == 4240 ? (0xf25, 3.18e+3) : (false, 'e') : 'p'
];
```

Výpis 3.6: Znak na ternální operátor s rozdělením řetězce po obfuskaci

3.4.2 Skrývání řetězců

Tato transformace se snaží řetězce skrýt, jak je již zmíněno v názvu [7].
`Příklad:`
Původní kód:

```
var protocol = 'http';
var domain = 'jscrambler.com';
var url = protocol + '://' + domain;
```

Výpis 3.7: Skrývání řetězců před obfuskací

Kód po obfuskaci:

```
var o = {
  f: function() { /* decoding algorithm and encoded data */ }
}
var protocol = o.f(13);
var domain = o.f(32);
var url = protocol + o.f(7) + domain;
```

Výpis 3.8: Skrývání řetězců po obfuskaci

3.4.3 Kódování řetězců

Řetězec se transformuje do zakódované podoby [7].
`Příklad:`
Původní kód:

```
'abcde';
```

Výpis 3.9: Kódování řetězců před obfuskací

Kód po obfuskaci:

```
'\u0061\u0062\u0063\u0064\u0065';
```

Výpis 3.10: Kódování řetězců po obfuskaci

3.4.4 Dělení řetězců

Řetězec je rozdělen do více výrazů. Může být ovlivněn počet částí, na kolik je řetězec rozdělen [7].
`Příklad:`
Původní kód:

```
"Hello";
```

Výpis 3.11: Dělení řetězců před obfuskací

Kód po obfuskaci:

```
var a = "He";
a += "ll";
a += "o";
a;
```

Výpis 3.12: Dělení řetězců po obfuskaci

3.5 Výroky

3.5.1 Rozložení operátoru čárky

Tato technika se využívá především, když má být více výrazů zahrnuto do jednoho.

Každý operand operátoru čárky je transformován na samostatný výraz, vznikne více samostatných výrazů, které jsou ve stejném pořadí jako původní tvar [7].
`Příklad:`
Původní kód:

```
(foo = 1, bar = 2, baz = 3);
```

Výpis 3.13: Rozložení operátoru čárky před obfuskací

Kód po obfuskaci:

```
foo = 1;
bar = 2;
baz = 3;
```

Výpis 3.14: Rozložení operátoru po obfuskaci

3.5.2 Nastínění funkce

Příkaz nebo skupina příkazů se převedou do nové deklarace funkce [7].
`Příklad:`
Původní kód:

```
function addPrefix (array, prefix) {
  var i = array.length - 1;
  for (i; i >= 0; i--) {
    array[i] = prefix + array[i];
  }
};
```

Výpis 3.15: Nastínění funkce před obfuskací

Kód po obfuskaci:

```
var o = function (D) {
    return {
        M: function () {
            var V, W = arguments;
            switch (D) {
            case 0:
                V = W[0] - W[1];
                break;
            case 1:
                V = W[1] + W[0];
                break;
            }
            return V;
        },
        E: function (v) {
            D = v;
        }
    };
}();

function addPrefix(array, prefix) {
    var i;
    o.E(0);
    i = o.M(array.length, 1);
    for (i; i >= 0; i--) {
        o.E(1);
        array[i] = o.M(array[i], prefix);
    }
}
```

Výpis 3.16: Nastínění funkce po obfuskaci

3.6 Identifikátory

3.6.1 Převedení operátoru . na operátor []

Složené odkazy v zápisu s tečkou se transformují do tvaru s hranatou závorkou [7].
`Příklad:`
Původní kód:

```
navigator.plugins.length
```

Výpis 3.17: Zápis z tečky do závorky před obfuskací

Kód po obfuskaci:

```
navigator["plugins"]["length"];
```

Výpis 3.18: Zápis z tečky do závorky po obfuskaci

Následně lze kombinovat s obfuskací řetězců.

3.6.2 Nepřímost globální proměnné

Pro každou globální proměnnou, na kterou se v kódu odkazuje se vytvoří alias. Tímto se kód stává méně přehledný například pro útočníka, jelikož najde méně známý kód [7].
`Příklad:`
Původní kód:

```
console.log();
```

Výpis 3.19: Nepřímost globální proměnné před obfuskací

Kód po obfuskaci:

```
var x = console;
var y = ["w", "o", "n", "d", "e", "r", "l", "a", "n", "d", "g"];
var z = x[y[6]+y[1]+y[10]];
```

Výpis 3.20: Nepřímost globální proměnné po obfuskaci

3.6.3 Přejmenování identifikátorů

Názvy identifikátorů jsou nahrazeny náhodně vygenerovanými názvy. Jsou kratší a nemají význam [7].
`Příklad:`
Původní kód:

```
function addNumbers() {
  function sum(number1, number2) {
    return number1 + number2;
  }

  var someNumber = 123;
  var anotherNumber = 45;
  return eval("sum(someNumber, anotherNumber)");
}
```

Výpis 3.21: Přejmenování identifikátorů před obfuskací

Kód po obfuskaci:

```
function addNumbers() {
  function c(d, e) {
      return d + e;
  }
  var a = 123;
  var b = 45;
  return eval("c(a,b);");
}
```

Výpis 3.22: Přejmenování identifikátorů po obfuskaci

3.7 Predikáty

3.7.1 Rozšíření predikátů

Tato transformace rozšiřuje predikáty v kódu o neprůhledné predikáty, což ztěžuje jejich pochopení [7].
`Příklad:`
Původní kód:

```
var i = 0;
while (i < 10) {
  i++;
}
```

Výpis 3.23: Rozšíření predikátů před obfuskací

Kód po obfuskaci:

```
var i = 0;
while (i < 10 && 5II.m()[8][1] == h5II.X()[7][2]) {
  i++;
}
```

Výpis 3.24: Rozšíření predikátů po obfuskaci

3.8 Funkce

3.8.1 Změna pořadí funkcí

Transformace náhodně mění pořadí deklarací funkcí [7].

3.9 Čísla

3.9.1 Číslo na řetězec

Transformace nahradí čísla (desetinná, osminová, šestnáctinná, exponentní zápis) řetězcovou reprezentací, která za běhu vrátí stejnou hodnotu čísla [7].
`Příklad:`
Původní kód:

```
user.phoneNumber = 555666777;
```

Výpis 3.25: Číslo na řetězec před obfuskací

Kód po obfuskaci:

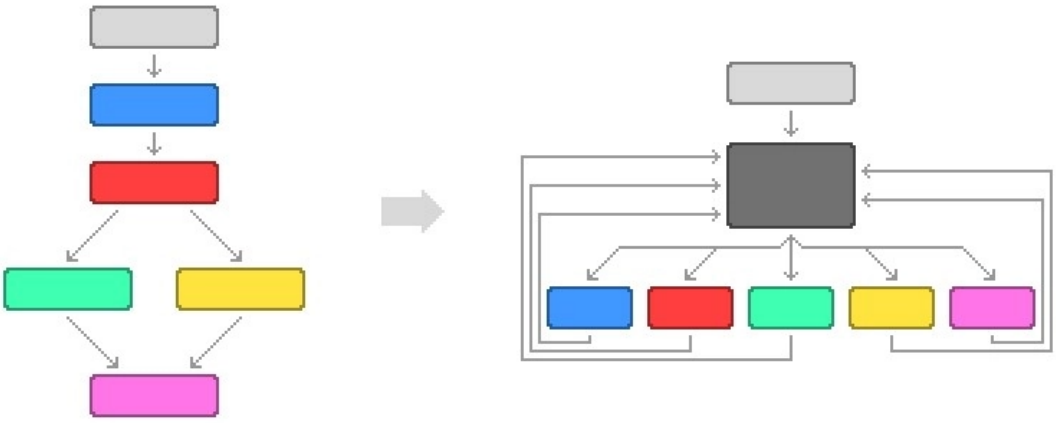
```
user.phoneNumber = '555666777'-+[];
```

Výpis 3.26: Číslo na řetězec po obfuskaci

3.10 Zploštění řídícího toku

Tato technika se snaží znejasnit tok programu tzv. `zploštěním`. Dosahuje toho tak, že rozděluje všechny základní bloky zdrojového kódu (tělo funkce, podmíněné větvení a smyčky), které umístí do jedné nekonečné smyčky s příkazem `switch` pro řízení toku programu. Přirozené podmíněné konstrukce již neexistují, tudíž se tok programu stává značně nepřehledný.

Následující diagram ukazuje abstraktní znázornění toku řízení a co se s ním děje. Vlevo se nachází tok programu před a vpravo po zploštění.

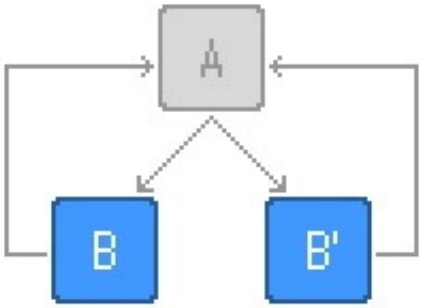


Obrázek 3.1: Diagram zploštění řídícího toku

Vlevo je snadné rozpoznat podmíněné příkazy (červený uzel), který vytváří větve k modrému a žlutému uzlu. Po zploštění jsou veškeré uzly na stejné úrovni vnoření, přepínají se a vracejí se zpět smyčkou k černému uzlu, který vybere další uzel [7].

3.10.1 Klony

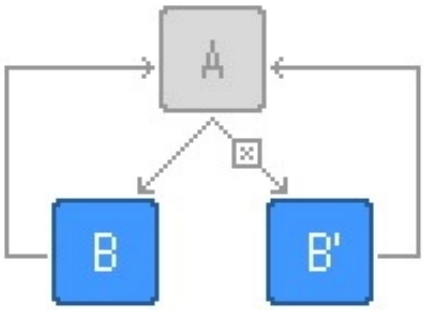
`Klon` je sémanticky ekvivalentní kopie základního bloku. `Zploštění řídícího toku` také používá `klony`. Lze jej zaměnitelně použít s již zmíněnými základními bloky. Tyto ekvivalentní `klony` nutí program mezi nimi náhodně vybírat. Tímto se znesnadňuje pochopení toku programu [7].



Obrázek 3.2: Klon zploštění řídícího toku

3.10.2 Mrtvé klony

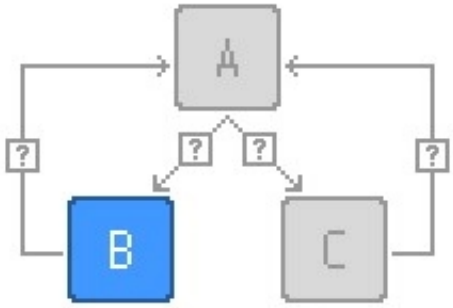
`Mrtvé klony` jsou klony, které nejsou nikdy spuštěny, ale napodobují kód, který bude spuštěn a navyšuje faktor zmatení [7].



Obrázek 3.3: Mrtvý klon zploštění řídícího toku

3.10.3 Neprůhledné kroky

`Neprůhledné kroky` znepřehledňují proměnnou pro přepínání. Tímto je složitější pochopit, který následující případ přepínač vybere. Toto zvyšuje odolnost proti statické analýze, jelikož `neprůhledné kroky` ztěžují určení dalšího kroku bez provedení kódu [7].



Obrázek 3.4: Neprůhledné kroky zploštění řídícího toku

3.10.4 Příklad

Následující úryvek kódu je algoritmus, který je používán k doporučování produktů nakupujícím na webových stránkách elektronického obchodu. Na základě historie nákupů generuje seznam doporučených produktů.

```
function getRecommendations(products, numRecommendations) {
    const weights [ ];
    for (let i = 0; i < products.length; i++) {
        const product products[i];
        const weight = getWeight(product);
        weights.push({_id: product.id, weight});
    }
    weights.sort((recommendation1, recommendation2) =>
        recommendation2.weight - recommendation1.weight
    );
    return weights.slice(0, numRecommendations);
}
```

Výpis 3.27: Zploštění řídícího toku příklad - zdrojový kód

Jedná se o jednoduchý kód, ale jde o proprietární algoritmus vyvinut společností, která nechce, aby se dostalo ke konkurenčním firmám.

Na následujícím výpisu 3.28 je kód po přidání jedné obfuskační techniky.

```
function getRecommendations (02, F5) {
    var C7 = M7wd_; var s2 = [arguments];
    s2[6] = C7.U1()[0][5]; C7.P1;
    for (; s2[6] !== C7.q8()[26][3];) {
        switch (S2[6]) {
            case C7.U1()[3][6]:
                s2[9][C7.F(4)]((function () {
                    C7.P1();
                    var z2 = [arguments];
                    z2[6] = C7.q8()[25][8];
                    for (; z2[6] !== C7.U1()[1][7];) {
                        switch (z2[6]) {
                            case C7.U1()[31][14]:
                            z2[9] = {};
                            z2[9][C7.F(3)] = s2[4][C7.F(6)];
                            z2[9][C7.F(1)] = s2[3];
                            return z2[9];
                            break;
                        }
                    }
                })[C7.F(0)](this, arguments));
                s2[6] = C7.U1()[24][32];
                break;
            case C7.U1()[15][6][26]:
                s2[6] = s2[5] < s2[0][0][C7.w(7)] ?
                C7.q8()[7][4] : C7.q8()[6][26];
                break;
            case C7.q8()[3][16]:
                s2[5] = 0;
                s2[6] = C7.q8()[20][11];
                break;
            case C7.U1()[22][8]:
                s2[5]++;
                s2[6] = C7.q8()[17][5][23];
                break;
            case C7.q8()[18][23]:
                s2[9] = [];
                s2[6] = C7.U1()[28][19];
                break;
            case C7.U1()[8][16]:
                s2[4] = s2[0][0][s2[5]];
                s2[3] = V0oXS$(s2[4]);
                $2[6] C7.q8()[15][18];
                break;
            case C7.U1()[21][8]:}
}
```

Výpis 3.28: Zploštění řídícího toku příklad - jedna obfuskační technika

Úryvek ukazuje jenom prvních pár řádků kódu, ale celý má téměř 700 řádků. Následující výpis 3.29 ukazuje příklad s extrémní obfuskací.

```
[][(![]+[])[+[]]+(![]+[])[!+[]+!+[]]+(![]+[])[+!+[]]+(!![]+[])[+[]]][([][(!
[]+[])[+[]]+(![]+[])[!+[]+!+[]]+(![]+[])[+!+[]]+(!![]+[])[+[]]]+[])[!+[]+!+
[]+!+[]]+(!![]+[][(![]+[])[+[]]+(![]+[])[!+[]+!+[]]+(![]+[])[+!+[]]+(!![]+[
])[+[]]])[+!+[]+[+[]]]+([[]]+[])[+!+[]]+(![]+[])[!+[]+!+[]+!+[]]+(!![]+[]
)[+[]]+(!![]+[])[+!+[]]+([[]]+[])[+[]]+([][(![]+[])[+[]]+(![]+[])[!+[]+!+[]+
]]+(![]+[])[+!+[]]+(!![]+[])[+[]]]+[])[!+[]+!+[]+!+[]]+(!![]+[])[+[]]+(!![]
+[][(![]+[])[+[]]+(![]+[])[!+[]+!+[]]+(![]+[])[+!+[]]+(!![]+[])[+[]]])[+!+[
]+[+[]]]+(!![]+[])[+!+[]]]((!![]+[])[+!+[]]+(!![]+[])[!+[]+!+[]+!+[]]+(!![]
+[])[+[]]+([[]]+[])[+[]]+(!![]+[])[+!+[]]+([[]]+[])[+!+[]]+(+[![]]+[][(
![]+[])[+[]]+(![]+[])[!+[]+!+[]]+(![]+[])[+!+[]]+(!![]+[])[+[]]])[+!+[]+[+!
+[]]]+(!![]+[])[!+[]+!+[]+!+[]]+(+(!+[]+!+[]+!+[]+[+!+[]]))[(!![]+[])[+[]]+
(!![]+[][(![]+[])[+[]]+(![]+[])[!+[]+!+[]]+(![]+[])[+!+[]]+(!![]+[])[+[]]])
[+!+[]+[+[]]]+([]+[])[([][(![]+[])[+[]]+(![]+[])[!+[]+!+[]]+(![]+[])[+!+[]]
+(!![]+[])[+[]]]+[])[!+[]+!+[]+!+[]]+(!![]+[][(![]+[])[+[]]+(![]+[])[!+[]+!
+[]]+(![]+[])[+!+[]]+(!![]+[])[+[]]])[+!+[]+[+[]]]+([[]]+[])[+!+[]]+(![]+
[])[!+[]+!+[]+!+[]]+(!![]+[])[+[]]+(!![]+[])[+!+[]]+([[]]+[])[+[]]+([][(!
[]+[])[+[]]+(![]+[])[!+[]+!+[]]+(![]+[])[+!+[]]+(!![]+[])[+[]]]+[])[!+[]+!+
[]+!+[]]+(!![]+[])[+[]]+(!![]+[][(![]+[])[+[]]+(![]+[])[!+[]+!+[]]+(![]+[])
[+!+[]]+(!![]+[])[+[]]])[+!+[]+[+[]]]+(!![]+[])[+!+[]]][([][[]]+[])[+!+[]]+
(![]+[])[+!+[]]+((+[])[([][(![]+[])[+[]]+(![]+[])[!+[]+!+[]]+(![]+[])[+!+[]
```

Výpis 3.29: Zploštění řídícího toku příklad - extrémní obfuskace

Tento kód neobsahuje alfanumerické obfuskace, což není časté. Pro člověka to může vypadat, jako že tento úryvek nelze zpětně analyzovat, ale automatické nástroje pro reverzní inženýrství jsdou schopny jej snadno převést. Zde se jedná příklad `slabé obfuskace`. Pro rozlišení mezi `silnou` a `slabou obfuskací` je nutné použít `metriky obfuskace` [8].

3.11 Metriky obfuskace

Pro měření síly obfuskace slouží právě `metriky obfuskace`. Existují tři klíčové metriky: `účinnost`, `odolnost` a `náklady` [8].

3.11.1 Účinnost

Metrika účinnost odpovídá na otázku „Do jaké míry je čtenář zmatený?“. Účinnost se v mnoha případech měří pomocí metrik složitosti softwaru (Halsteadovy metriky [4]). Existují i specifické charakteristiky, které se dají použít pro vyhodnocení účinnosti.:

- Skrytí konstant a názvů.
- Ztěžování pochopení pořadí, v jakém je kód prováděn.
- Ztěžování pochopení příslušného kódu.
- Zvyšování celkové velikost programu a zavádění nových tříd a metod.
- Zavádí nové predikáty a přepisuje podmíněné a cyklické konstrukce [8].

3.11.2 Odolnost

Metrika odolnosti odpovídá na otázku „Jak dobře odolávají automatické deobfuskační útoky?“. Například, pokud přidáme do kódu příkaz `if`, který do kódu zavede fiktivní proměnnou, člověku chvíli potrvá, než kód identifikuje jako fiktivní. Deobfuskační nástroj by tento příkaz `if` ihned odstranil.

Odolnost se počítá pomocí dvou hledisek:

- Čas potřebný k vývoji deobfuskačního nástroje, který by byl schopen vrátit výsledek transformace.
- Čas a prostor potřebný k provedení deobfuskačního nástroje, aby bylo možné transformaci účinně vrátit.

Mnoha obfuskačních nástrojů pro `JavaScript` selhává právě na tomto kritériu, kdy vytvoří výstup, který vypadá jako silně obfuskovaný kód, ale jenom pro lidské oko [8].

3.11.3 Náklady

Metrika nákladů představuje dopad transformace na dobu provádění transformované aplikace a také dopad na velikost souboru aplikace. To je důležité, protože je nežádoucí, aby byl výkon aplikace zničen kvůli obfuskaci [8].

Kapitola 4

Diplomová práce Vojtěcha Randýska

Následující kapitola se zabývá diplomovou prací Vojtěcha Randýska na téma `Detekce kódu v jazyce JavaScript se známými bezpečnostními chybami`. Kapitola je zaměřená na vzniklé rozšíření webového prohlížeče v rámci této práce a na použité normalizační techniky.

V teoretické části pan Randýsek došel k závěru, že oblast, ve které tato práce může doopravdy přispět, je oblast zranitelností klientského JavaScriptu z pohledu koncového uživatele webu. Nástroje, které byly v práci pana Randýska zkoumány, kód stránek pouze analyzují, ale neopravují. Jsou technicky zastaralé a spoléhají, že správně identifikují knihovnu i její verzi.

4.1 Návrh

Pan Randýsek navrhl rozšíření pro webový prohlížeč, které má za úkol detekovat a opravit známé bezpečnostní chyby JavaScryptového kódu. Součástí jsou také skripty, knihovny a procesy pro zpracování zranitelností. Jádro tvoří program, který je na základě `commitu` schopný vytvořit abstraktní syntaktický strom původního a v `commitu` upraveného kódu. Tento abstraktní strom je následně převeden na zásobníkový automat. Výsledná implementace s automaty nepracuje, jejich funkce byla nahrazena jednoduchým procházením stromu.

Analyzovaný kód je přetransformován na abstraktní syntaktický strom, který se vkládá na vstup sestrojeného automatu. Pokud je strom automatem přijat, vyskytuje se v analyzovaném kódu známá zranitelnost.

Navržená soustava je rozdělena do tří částí, které značí zpracování dat, detekci chyb s případnou opravou a použití koncovým uživatelem. Vstupem jsou metadata balíčků v systému NPM[1] a existující databáze zranitelností. Výstup tvoří metadata zranitelností, která jsou rozšířena o zdrojové kódy balíčků a obsah `commitů`, které opravují zranitelnost.

V první části soustavy dochází k získávání metadat z databáze zranitelností, která jsou párována s metadaty balíčků v NPM a jsou dohledávány příslušné `commmity`.

V hlavní části se provádí detekce zranitelného kódu. Jakožto prerekvizita je dokončení zpracování `commitů` z předchozí části. Zranitelný kód i jeho oprava je přetransformován na abstraktní syntaktické stromy. Stromy jsou dále převedeny na zásobníkové automaty, které je přijímají. Detekce je zahájena převodem analyzovaného JavaScript kódu na abstraktní syntaktický strom. Tento syntaktický strom je vložen na vstup automatů. Každý z těchto

[1] NPM – Node Package Manager, dále bude používána zkratka.

automatů odpovídá různým zranitelnostem. Pokud je strom přijat některým z automatů, je odhalena zranitelnost.

Poslední část je zaměřena na použitelnost koncovým uživatelem. V této části se získává JavaScriptový kód webové stránky a je poskytován jako vstup pro detekci zranitelného kódu. V případě pozitivního nálezu dojde k nahrazení zranitelného kódu za opravený kód.

Mezi pomyslnými vrstvami se návrh snaží udržet jasná rozhraní. Zajišťuje tak znovupoužitelnost, rozšiřitelnost a odolnost proti chybám v návrhu. Hledaný kód nemusí být zranitelný, nástroj může být použit i pro sofistikované nahrazování. Nástroj navíc není vázaný na NPM, libovolný JavaScript kód může být také jako vstup [12].

4.2 Implementace

Pan Randýsek porovnávání abstraktních syntaktických stromů za pomoci zásobníkových stromů při implementaci odstranil. Pro lepší výkon použil `preorder`[2] průchod stromem společně s `hash`[3] funkcí. Podpůrné skripty systému slouží k tvorbě a zpracování zranitelností [12].

4.2.1 Knihovna pro práci s abstraktními syntaktickými stromy

Knihovna je panem Randýskem považována za jádro nástroje a je označená jako `js-to-ast`. Jedná se o interní NPM balíček implementovaný v jazyce JavaScript. Zmíněné technologie byly zvoleny kvůli použití v rozšíření prohlížeče Chrome. Rozšíření jsou standardně implementována v JavaScriptu a při využití správných pomocných nástrojů lze použít i NPM balíčky. Účelem této knihovny je zapouzdření celé práce s abstraktním syntaktickým stromem [12].

Tvorba abstraktního syntaktického stromu

Převedení kódu na abstraktní syntaktický strom provádí každý interpret JavaScriptu. Pan arndýsek využil překladač `Acorn`[4], který implementuje `ESTree specifikaci`[5]. Specifikace `ESTree` sama sebe označuje jako `lingua franca`[6]. Překladač Acron prošel jedinou úpravou a to přidáním normalizace získaného AST [12].

Formát uložení zranitelností

Velkou část řešení tvoří definice vhodného formátu ukládání dat zranitelností a oprav. Zranitelnosti se nacházejí v souboru `generated_vulnerabilities.json` a jsou uloženy jako jediný objekt. Soubor lze chápat jako víceúrovňový slovník. Nejvyšší úroveň tvoří jednotlivé typy uzlů specifikace ESTree. Jedná se o `Program`, `BlockStatement` a `Literal`. Nižší úroveň uchovává data zranitelnosti příslušného uzlu. Každá zranitelnost je uložena pod klíčem, který odpovídá `SHA1` hashům AST zranitelného kódu. Obsahuje unikátní identifikátor

[2] Preorder je jeden z možných průchodů stromem, kdy je nejprve zpracován kořen a poté po řadě jeho synové.

[3] Matematická funkce pro převod vstupních dat do (relativně) malého čísla. Výstup této funkce se často označuje jako otisk,

[4] https://www.npmjs.com/package/acorn

[5] https://github.com/estree/estree

[6] Lingua franca je jakýkoliv jazyk šířeji využívaný nad rámec rodilých mluvčích.

a identifikátor opravy. Identifikátor opravy je nepovinný, ne každá zranitelnost je v rámci nástroje opravitelná [12].

V souboru `generated_vulnerabilities_meta.json` se nachází slovník s metadaty zranitelností. Identifikátorem zranitelnosti je klíč. Každá zranitelnost má uložený název, popis, vážnost a URL, která odkazuje na další informace o zranitelnosti [12].

Poslední slovník se nachází v souboru `generated_patches.json`, který obsahuje data oprav. Identifikátorem oprav je opět klíč. Opravy obsahují celé AST opravy v normalizované podobě [12].

Detekce zranitelností a tvorba opravy

Knihovna má jako hlavní úkol nalézt známe zranitelnosti. Toto zajišťuje funkce `findMatches(input, vulnerabilities, patches, meta)` ve skriptu `finder.js`. Popis parametrů:

- `input` – vstupní skript v textové podobě
- `vulnerabilities` – data zranitelností
- `patches` – data oprav
- `meta` – metadata zranitelností

Pro generaci výstupního kódu z opraveného AST je použita knihovna `Escodegen`[7]. Algoritmus 1 je pseudokód procesu detekce [12].

Algorithm 1 Průběh detekce

```
Get AST from Acorn
Normalize AST
for Every node do
    Get vulnerabilities for node type
    if Vulnerabilities for node type are not empty then
        Stringify node
        Calculate hash
        if Vulnerabilities contain hash then
            Replace node with patch
            Add vulnerability metadata to result list
        end if
    end if
end for
Generate patched code with Escodegen
Return found vulnerabilities and patched code
```

Minifikace

Podrobné vysvětlení a popis `minifikace` je v kapitole 2.

Návrh i implementace nástroje s minifikací zdrojového kódu částečně počítají. Implementace ukázala, že minifikovaný kód nemusí mít s původním kódem totožný abstraktní

[7] https://www.npmjs.com/package/escodegen

syntaktický strom. Velikost vzniklého rozdílu záleží na druhu provedení minifikace. Řešení je však odolné vůči bílým znakům a komentářům, které se na výsledném syntaktickém stromu nijak nepodílí. Stromy jsou ale pozměněny při přejmenování funkcí a proměnných, změně v pořadí jednotlivých prvků a dalších již zmíněných technik minifikace [12].

Jelikož má minifikace značné dopady na postupy detekce, je do knihovny implementovaná podpora normalizačních funkcí. Úkol těchto funkcí je zajistit aby byl ekvivalentní kód převeden na stejný syntaktický strom. Poté, co je analyzovaný kód převeden na abstraktní syntaktický strom, se provede normalizace. V rámci práce Randýsek vybral následující tři typy minifikace, jako ukázku, jak by transformace mohla vypadat:

- odstranění informací o poloze uzlů a typu vstupního kódu (atribut sourceType uzlu typu Program)
- sloučení po sobě jdoucích deklarací proměnných do jedné deklarace
- převod uvozovek řetězců na složené uvozovky [12]

Jak jsem však ukázal v kapitole 2, v praxi se vyskytují další typy minifikace a cílem této práce je nalézt potřebnou normalizaci

Obfuskace

Stejně jako `minifikace` i `obfuskace` je podrobně vysvětlena v předchozí kapitole 3.

Randýskova práce se nesnaží detekci obfuskovaného a ani minifikovaného kódu pokrýt. Vytvořený nástroj ovšem dokáže s transformovaným kódem pracovat bez problému. Pokud je transformovaný kód známý vstup, nástroj je schopen ho rozpoznat. Může se jednat například o minifikovanou verzi knihovny `jQuery` [12].

Známe zranitelnosti a pomocné skripty

Zpracování známých zranitelností je významnou oblastí Randýskové práce. Randýsek považuje zranitelnost za známou, pokud je uveřejněna v některém z veřejných registrů. Jedná se například o databáze NVD, Snyk i Github Advisory. Pro implementaci byla zvolena Github Advisory, jelikož poskytuje jednoduché uživatelské prostředí s funkcemi pro vyhledávání a filtrování pomocí platformy.

Pomocné skripty jsou uchovány ve složce `vulnerability-processing` a jsou zaměřeny na podporu tří typů zpracování zranitelností:

- ručním zpracováním
- automatizovaně se zdrojovými soubory staženými přes FTP
- automatizovaně se zdrojovými soubory verzovanými nástrojem Git [12]

Za zranitelný kód je v této práci považován kód, který byl změněn v `commitu` a byl označen jako oprava nějaké zranitelnosti. Jednotkou kódu, která se dá považovat za atomickou a zranitelnou, může být literál, funkce, výraz či celý skript. Pro jednoduchost byl za atomickou jednotku vybrán celý soubor. Chápat celý soubor atomicky umožňuje snadnou podporu oprav. Ve fázi opravy se celý skript nahradí jiným skriptem [12].

Po manuální identifikaci `commitu` opravující zranitelnost, existují tři možnosti jak postupovat:

- manuální zpracováním
- zpracování na základě URL Git repositáře a hashe commitu
- zpracování na základě URL zranitelné knihovny a opravy [12]

Při manuálním zpracování je hlavní výhodou svoboda označení kódu zranitelnosti a opravy. Postup při manuálním přidání zranitelnosti:

- vložení zranitelného kódu a kódu opravy do skriptu `show_ast_manual.js`
- nastavit příznak `showAstOnly` na true a spustit skript
- vložit získané `AST` ze standardního výstupu do skriptu
- nastavit příznak `showAstOnly` na false a spustit skript
- vložit získané hashe ze standardního výstupu společně s metadaty zranitelnosti do skriptu `js-to-ast/src/run.js` [12]

Při automatizovaném zpracování se jedná o podobný postup, který je ovšem abstrahovaný do skriptů `show_ast.js`, `run.sh`, `process_ftp.sh` a `process_git.sh`. `Shell` skripty jsou použity z důvodu snadné integrity s `Gitem`. Zpracování souborů pomocí FTP je jednodušší oproti Gitu. Soubory zranitelnosti jsou staženy a následně převedeny na AST. Následně se spočítají hashe a ty jsou s metadaty uloženy do generovaných souborů `jstoast/src/generated_*.json`. Situace je složitější v případě Gitu. Nejprve dojde ke stažení repositáře. Následně je uskutečněn průchod souborů, které byly v `commitu` změněny, mimo soubory testů. Prvotní verze je brána jako zranitelná, verze po změně jako opravená. Dokončení procesu je stejné jako u FTP [12].

Rozšíření pro prohlížeč Chrome

Rozšíření prohlížeče Chrome je hlavním výstupem této práce. Proces zpracování v režimu opravy probíhá následovně.

- Při spuštění je do stránky zaveden skript `content_script.js`. Skript zastaví veškeré běžící vykonávání na stránce a zobrazí indikátor průběhu. Stáhne obsah stránky za pomoci `XMLHttpRequest` a spustí její zpracovávání. Odešle veškeré uzly typu `script`, případně `URL` do skriptu `background.js`, která slouží jako service worker.
- Na příchozí události reaguje již zmíněný skript `background.js`. V případě, že je obsahem příchozí události `URL` skriptu, daný skript stáhne obsah prostřednictvím `fetch API`. Následně jsou volány funkce z knihovny `js-to-ast`, které hledají zranitelnosti a vracejí opravený kód. Informace o nalezených zranitelnostech a kódy oprav se vracejí v odpovědích na zprávy do `content_script.js`.
- `Content_script.js` vytváří v paměti `DOM` nové stránky, kde jsou nahrazeny původní zranitelné skripty jejich získanými opravami.
- Následně jsou metadata zranitelností pro danou stránku agregována a poslána do `background.js` k uložení.
- Nakonec je nový `DOM` vložen stránky v prohlížeči, která byla v průběhu zastavená a běžel na ní ukazatel průběhu.

Jsou podporovány 4 režimy běhu, lišící se v chování k analyzované stránce a zranitelným skriptům:

- `Disabled` - `content_script.js` nezastavuje vykonávání stránky ani neprochází její obsah. Rozšíření je ale načteno.
- `Analyze` - vykonávání stránky není pozastaveno, nalezené zranitelnosti jsou ukládány a reportovány. Zranitelné skripty nejsou na stránce nijak omezeny.
- `Block` - vykonávání stránky je zastaveno. Reportování zranitelností probíhá stejně jako v předchozím režimu `Analyze`. Zranitelné skripty jsou odstraněny z `DOM`u a nejsou vůbec vykonány, pokud je rozšíření stihne zablokovat před jejich vykonáním. Web extension API totiž neumožňuje zajistit blokování bez rizika *race condition*.
- `Repair` - vykonávání stránky je zastaveno. Zranitelnosti jsou reportovány stejně jako v režimu `Analyze`. Zranitelné skripty se známými zranitelnostmi jsou opraveny. [12]

Kapitola 5

Návrh a implementace normalizace zdrojového kódu

Následující kapitola se zabývá mnou navrženými normalizacemi a normalizacemi pana Randýska. Veškeré tyto normalizace jsou implementovány v knihovně `js-to-ast`, konkrétně v souboru `finder.js`. Knihovna je více popsána v dřívější sekci práce 4.2.1. Pro jednodušší průchod syntaktickým stromem Randýsek využil modifikovanou knihovnu `Acorn AST walker` [1], kterou jsem následně využil také.

5.1 Normalizace pana Randýska

Jak bylo již zmíněno v předchozí kapitole, pan Randýsek ve své práci provedl pár kroků pro normalizaci minifikovaného kódu. Konkrétně se jedná o následující transformace: převod uvozovek řetězců na složené uvozovky, sloučení po sobě jdoucích deklarací proměnných do jedné deklarace, odstranění informací o poloze uzlů a typu vstupního kódu (atribut sourceType uzlu typu Program). Tyto transformace jsou vysvětleny na následujícím příkladu:

```
var a = "hello";
var b = 'world';
```

Výpis 5.1: Vstup bez minifikace

```
var a="hello",b="world";
```

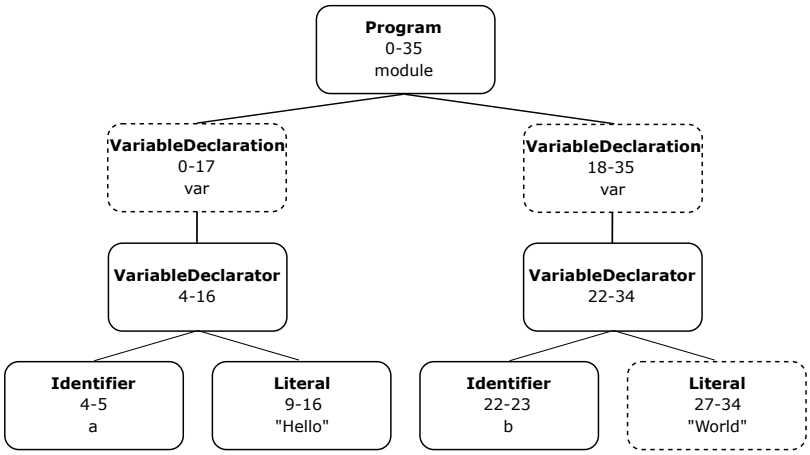
Výpis 5.2: Vstup s minifikací

Pokud jsou vygenerovné abstraktní syntaktické stromy pomocí nástroje AST explorer[2], vznikne grafická podoba na obrázcích 5.1 a 5.2

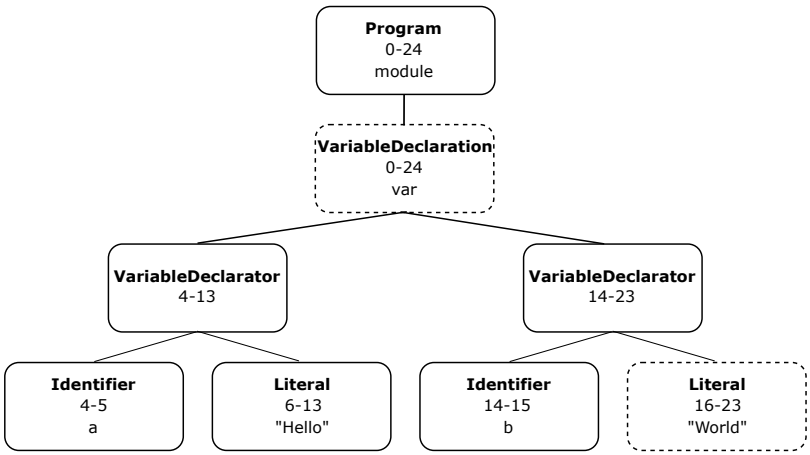
V každém uzlu je uvedený typ, pozice v kódu a v některých případech doplňující data. Uzly označené přerušovanou čarou obsahují změny oproti minifikované verzi.

Uzly *VariableDeclaration* byly sloučeny a hodnota literálu *word* má jiný typ uvozovek. Data o poloze v kódu jsou ve všech uzlech jiná, jelikož minifikace kód zkrátila.

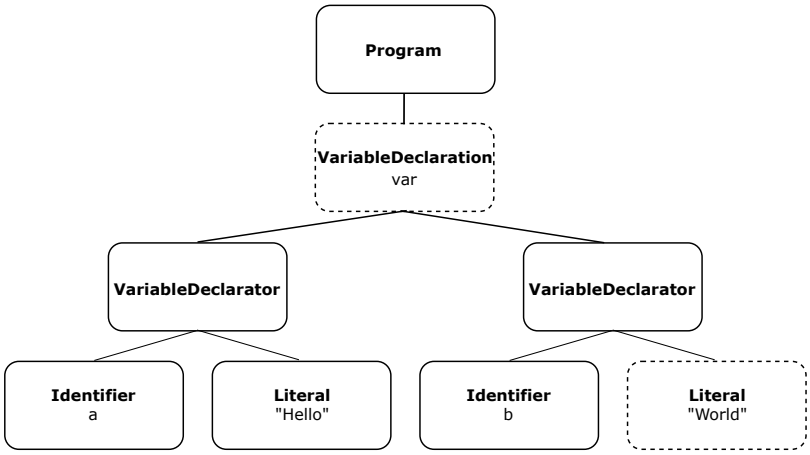
[1] https://www.npmjs.com/package/acorn-walk
[2] https://astexplorer.net/



Obrázek 5.1: Grafická podoba AST výpisu 5.1 [12]



Obrázek 5.2: Grafická podoba AST výpisu 5.2 [12]



Obrázek 5.3: Grafická podoba AST s normalizací [12]

5.2 Vlastní normalizace

Normalizace jsem navrhl pouze pro minifikační a obfuskační metody, které jsem nastudoval, nebo které jsem objevil při testování. Jak je již uvedeno v kapitole 3, obfuskace mění zdrojový kód výrazně více než minifikace. Z tohoto důvodu jsem využil deobfuskačního nástroje `JavaScript Deobfuscator`[3], který disponuje i online verzí. Tento nástroj pomáhá zjednodušit obfuskovaný kód, ale ne vždy jej dokáže vrátit do původní podoby. Vstupní kód je nejdříve deobfuskován a následně je převeden na `AST`. Tento strom je normalizován a použit na vyhledávání pro detekci chyb.

5.2.1 Přejmenování identifikátorů

Minifikační i obfuskační nástroje mění názvy identifikátorů. Na výpisu 5.3 je znázorněn kód před obfuskací a na výpisu 5.4 je ten stejný kód po obfuskaci, kde došlo k přejmenování identifikátorů. Deobfuskační nástroj tuto obfuskaci nijak nezměnil.

```
function addNumbers() {
    function sum(number1, number2) {
      return number1 + number2;
    }
    var someNumber = 123;
    var anotherNumber = 45;
    return sum(someNumber, anotherNumber);
}
```

Výpis 5.3: Přejmenování identifikátorů před obfuskací

```
function addNumbers() {
    function c(d, e) {
      return d + e;
    }
    var a = 123;
    var b = 45;
    return c(a,b);
}
```

Výpis 5.4: Přejmenování identifikátorů po obfuskaci

Normalizace

Původně jsem uvažoval přejmenovat identifikátory pouze pomocí písmen abecedy stejně jako obfuskační nástroj. Toto řešení by se obtížně testovalo, proto jsem přidal některé informace navíc. Názvy proměnných a atributy objektů jsou ve tvaru `číslo_znak`. Názvy funkcí, objektů a metod zase ve tvaru `číslo_znak_fn`. Číslo značí úroveň vnoření v programu, které se inkrementuje každým blokem kódu, znak se mění abecedně podle pořadí v příslušném bloku kódu. Pokud jsou vyčerpány znaky abecedy, název se mění na tvar *aa* až *zz*, *aaa* až *zzz* atd. V JavaScriptu názvy identifikátoru nesmí začínat číslem, zde přejmenovávám identifikátory pouze v `AST`. Na výpisu 5.5 je znázorněna normalizace výpisů 5.3 a 5.4.

[3] https://deobfuscate.io/

```
function 1_a_fn() {
    function 2_a_fn(2_a, 2_b) {
      return 2_a + 2_b;
    }
    var 1_a = 123;
    var 1_b = 45;
    return 2_a_fn(1_a, 1_b);
}
```

Výpis 5.5: Přejmenování identifikátorů po normalizaci

Na výpisu 5.6 je úryvek kódu s objektem a na výpisu 5.7 je jeho normalizovaná verze.

```
const shoppingCart = {
    items: [],
    addItem(item) {
      this.items.push(item);
    }
};

const item1 = { name: 'Shirt', price: 25.99 };
shoppingCart.addItem(item1);
```

Výpis 5.6: Přejmenování identifikátorů objekt

```
const 1_a_fn = {
    2_a_fn: [],
    2_b_fn(2_a) {
      this.2_a_fn.push(2_a);
    }
};

const 1_b_fn = { 1_a: 'Shirt', 1_b: 25.99 };
1_a_fn.2_b_fn(1_b_fn);
```

Výpis 5.7: Přejmenování identifikátorů objekt po normalizaci

Implementace

Normalizaci jsem naimplementoval do funkce `normaliseRenamingIdentifiers(ast)`. Tato normalizace nepoužívá `Acorn AST walker`. Pro průchod `AST` jsem použil breadth-first search algoritmus, jelikož při průchodu je potřeba znát hloubka uzlu a předchozí uzel, toto `Acorn AST walker` neumožňuje. Celkově dochází ke třem průchodům stromu.

Při prvním průchodu dochází k vytvoření nových názvů. Nový název je vytvořen pro uzly *VariableDeclarator*, *FunctionDeclaration*, *ClassDeclaration*, *MethodDefinition*. *Property*, *ArrowFunctionExpression* a *FunctionExpression*, následně jsou uloženy v objektu `newIdentifiers`, jehož struktura je zobrazena na výpisu 5.8. Číslo znázorňuje hloubku vnoření, funkce mají uložený starý název, nový vygenerovaný název, typ a informaci, do které funkce patří. Proměnné jsou uloženy pod funkcí, ve které byly deklarovány, mají uložený starý název, nový název a typ.

Při druhém průchodu se přejmenovávají názvy metod a funkcí. Pro uzly *Property*, *FunctionDeclaration*, *ClassDeclaration*, *MethodDefinition* a *VariableDeclarator* typu *ObjectExpression* se prochází objekt *newIdentifiers*. Nejdříve se v objektu prochází další objekty, uložené pod stejnou úrovní vnoření, jakou má aktuální uzel. Následně se hledá objekt, který má atribut *oldName* stejný jako název uzlu. Pokud nedojde ke shodě názvů, sníží se úroveň vnoření, kterou se vyhledává v objektu *newIdentifiers*. Tímto způsobem je iterováno maximálně do úrovně vnoření 0. Pokud shoda nastane, aktuální uzel přebere hodnotu atributu *newName*.

Třetí průchod přejmenovává zbytek identifikátorů. Aktuální uzel zná název funkce, ve které se nachází. Nejdříve se prochází objekt *newIdentifiers* podle úrovně vnoření uzlu. Pokud není nalezena funkce, kterou má uzel uloženou, je snížena úroveň vnoření. Při nálezu funkce se prohledává název *identifikátoru*. Pokud se *identifikátor* nenajde, sníží se úroveň vnoření a hledá se funkce, která je uložena v atributu *prevFunc*. Tímto způsobem se hledají volné proměnné. Pokud se najde patřičný název *identifikátoru*, změní se jeho název podle atributu *newName*.

```
1: {
    functionName: {
        oldName: "function1"
        newName: "0_a_fn"
        type: "function"
        prevFunc: "program"
        variableName: {
            oldName: "item1"
            newName: "0_a"
            type: "identifier"
        }
    }

}
```

Výpis 5.8: newIdentifiers

Jelikož obfuskační nástroje jsou nedeterministické a mohou používat jiné techniky, je šance, že obfuskace jenom nepřejmenuje identifikátory.

Při konzultaci s vedoucím nás napadly dva případy, které by mohly nastat a normalizace s nimi nepočítá. Na výpisu 5.9 a 5.10 je znázorněn kód, který je funkčně stejný, ale je změněné pořadí deklarací proměnných.

```
var a = 1;
var b = 2;
```

Výpis 5.9: Pořadí identifikátorů

```
var b = 2;
var a = 1;
```

Výpis 5.10: Změna pořadí identifikátorů

V tomto případě by šlo pořadí měnit podle abecedního pořadí, ale ne vždy by musely být identifikátory pojmenovány abecedně. Další možnost, kterou jsem uvažoval, by bylo řazení podle hodnoty. Problém by ovšem nastal, kdyby proměnné neměly přiřazenou hodnotu stejného typu nebo by byla přiřazena např. funkce.

Na výpisu 5.11 je znázorněna deklarace objektu a následné přiřazení hodnoty do atributu. Výpis 5.12 ukazuje funkčně stejný kód, ale deklarace objektu je v podmínkovém výrazu. Normalizace by v případě výpisu 5.12 přejmenovala pouze objekt vnořený v podmínkovém výrazu, protože přiřazení hodnoty atributu je ve vnějším bloku kódu oproti deklaraci. Normalizace taky nerozlišuje, zda je proměnná typu *var* nebo *let*.

```
var obj = {};
obj = {x: 5};
```

Výpis 5.11: Deklarace objektu

```
if(1){
    var obj = {};
}
obj = {x: 5};
```

Výpis 5.12: Deklarace objektu v podmínkovém výrazu

5.2.2 Zápis atributů a odstranění prázdných výrazů

Výpis 5.13 ukazuje úryvek kódu a na výpisu 5.14 je jeho minifikovaná verze. Nástroj změnil způsob zápisu atributu *text* a také přidal prázdný výraz. Obrázek 5.4 ukazuje AST výpisu 5.13 a obrázek 5.5 AST minifikovaného výpisu 5.14.

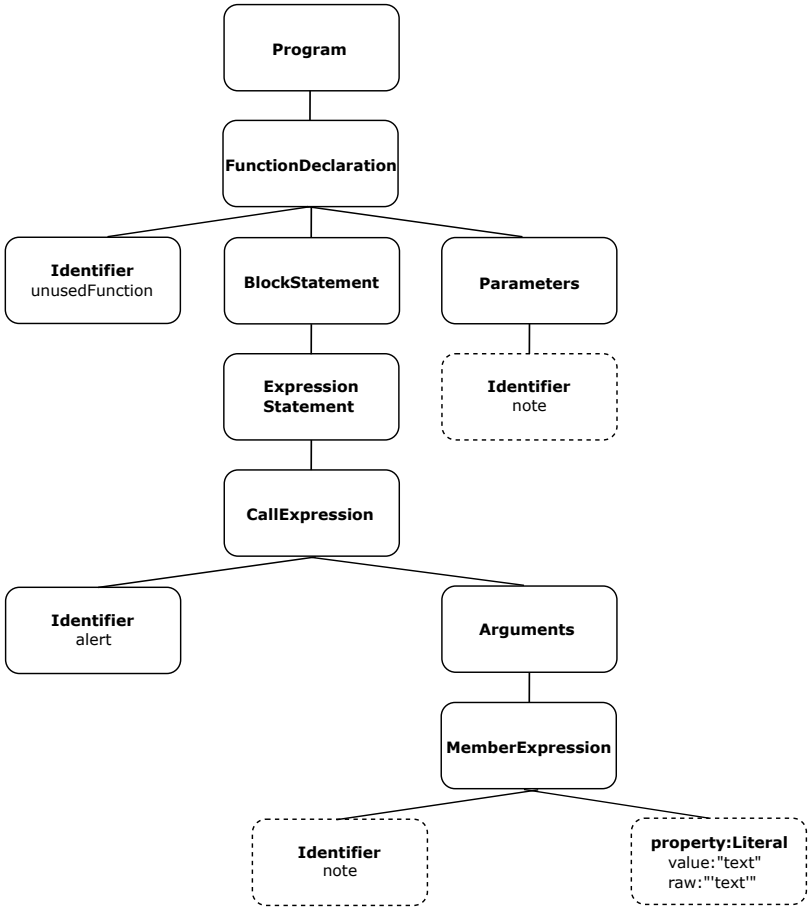
```
function unusedFunction(note) {
    alert(note['text']);
}
```

Výpis 5.13: Vstup bez minifikace

```
function unusedFunction(a){alert(a.text)};
```

Výpis 5.14: Vstup s minifikací

Po minifikaci byl v AST vytvořen uzel *EmptyStatement*, jelikož minifikační nástroj vložil středník za deklaraci funkce. Hodnota identifikátoru, který je parametrem funkce, byla změněna z *note* na *a*. Uzel *literal* byl změněn na uzel *property:identifier*.

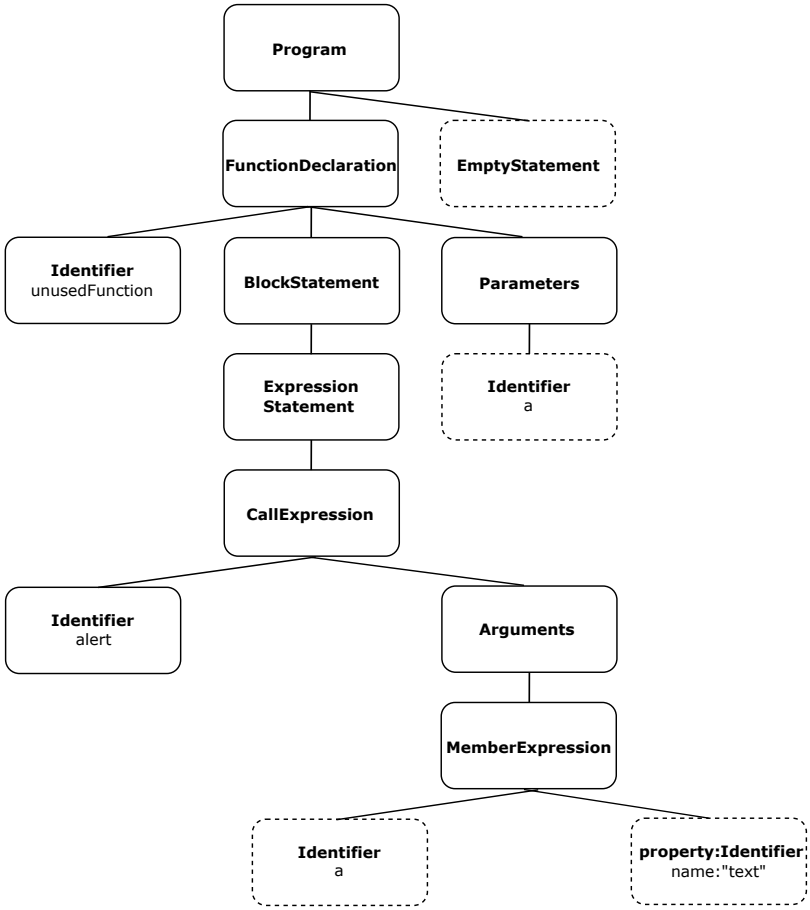


Obrázek 5.4: Grafická podoba AST výpisu 5.13

Normalizace

Uvažoval jsem, který způsob zapsání atributu zvolit, nakonec jsem zvolil variantu s tečkou. Pro normalizaci jsem navrhl tyto transformace, výsledný strom obr. 5.4 je stejný jako strom pro výpis 5.13.

- odstranění uzlů *EmptyStatement*,
- odstranění atributu *computed* v uzlu *MemberExpression*,
- převod property typu *literal* na property typu *indentifier*, atribut *value* převeden na atribut *name*, odstranění atributu *raw*,



Obrázek 5.5: Grafická podoba AST výpisu 5.14

Implementace

Implementaci navržené normalizace jsem rozdělil do dvou hlavních funkcí. Funkce `normaliseEmptyStatement(nodeList)` slouží pro odstranění uzlů typu *EmptyStatement*, argument `nodeList` je list dětských uzlů příslušného rodičovského uzlu.

`normalisePropertyLiteral(node)` je funkce, která předělává uzel *literal* na uzel *identifier*. Atribut *name* převezme hodnotu atributu *value*, atributy *raw* a *value* jsou odstraněny.

5.2.3 Deklarace objektu s atributy

Výpis 5.15 ukazuje deklaraci objektu s atributem, jeho minifikovaná podoba je na ukázce výpisu 5.16. Grafická podoba obou výpisů je znázorněna na obrázcích 5.6 a 5.7

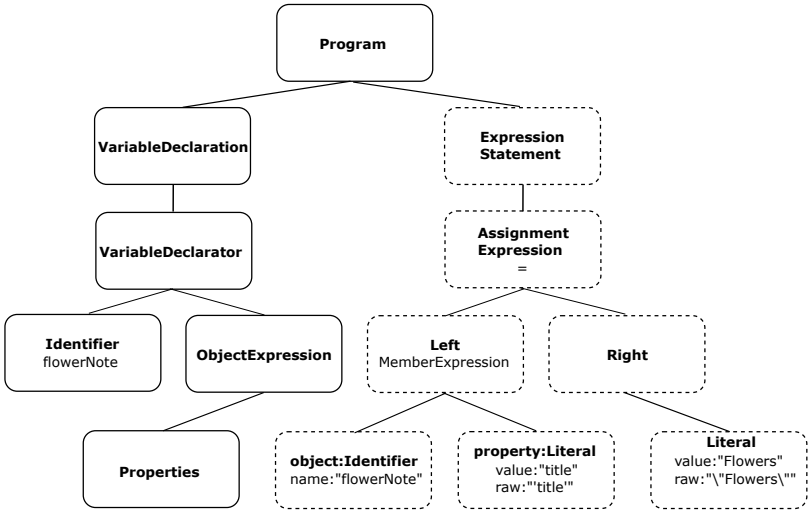
```
var flowerNote = {};
flowerNote['title'] = "Flowers";
```

Výpis 5.15: Vstup bez minifikace

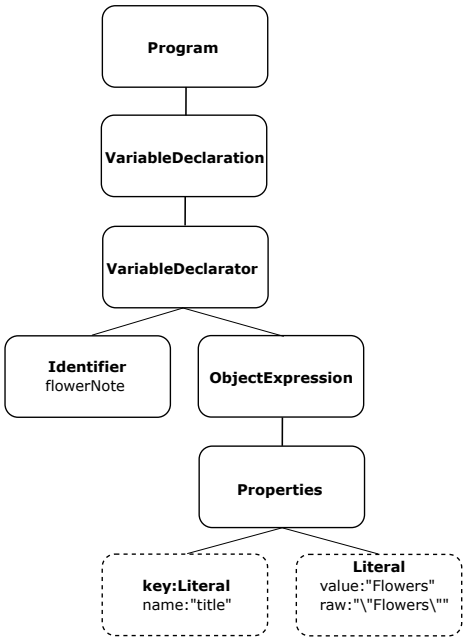
```
var flowerNote={title:"Flowers"};
```

Výpis 5.16: Vstup s minifikací

V tomto případě strom minifikovaného kódu vypadá výrazně jinak. Výraz z výpisu 5.15 je v minifkované verzi brán jako *property* objektu flowerNote, chybí zde celý pravý podstrom uzlu *Program*.



Obrázek 5.6: Grafická podoba AST výpisu 5.15

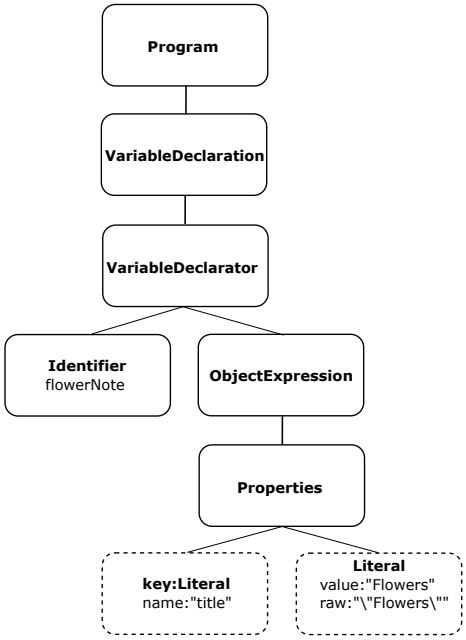


Obrázek 5.7: Grafická podoba AST výpisu 5.16

Normalizace

Normalizovaná podoba `AST` obr. 5.8 vypadá stejně jako `AST` obr. 5.7 pro minifikovaný kód. Přišlo mi jednodušší převést `AST` původního kódu na `AST` minifikovaného kódu. Normalizace pro `AST` obr. 5.6 jsem provedl takto: Jelikož mají uzel *VariableDeclaration* a *Expression-Statement* stejný rodičovský uzel (v tomto případě uzel *Program*), nacházejí se ve stejném

bloku kódu. Podle uzlu *object:Identifier* s atributem *name:"flowerNote"*, který se nachází v podstromu uzlu *ExpressionStatement* a podle uzlu *Identifier* s atributem *flowerNote* v podstromu uzlu *VariableDeclaration* jsou uzlu *Properties* přiřazeny dva potomci. Uzel *property:Literal* je převeden na uzel *key:Literal* s atributem *name:"title"* a společně s uzlem *Literal* s atributy *value:"Flowers", raw:"\"Flowers\""* je přidělen uzlu *Properties*.



Obrázek 5.8: Grafická podoba AST s normalizací

Implementace

Tuto normalizaci jsem implementoval pomocí funkce `normaliseAssignmentExpression(nodeList)`, parametr *nodeList* je seznam dětských uzlů rodičovského uzlu, který obsahuje uzly *VariableDeclaration* a *ExpressionStatement*. Pro každý uzel *VariableDeclaration* se iteruje mezi uzly *ExpressionStatement*. Pokud se jméno proměnné shoduje se jménem v uzlu *object*, který patří k uzlu *ExpressionStatement*, provede se na základě operátoru v uzlu *AssignmentExpression* operace. V tomto případě se jedná o operaci *assign*, tudíž jsou vytvořeny dva nové uzly *key* a *literal*, které obsahují příslušné hodnoty. Nové uzly se přidělí uzlu *properties* patřičné proměnné. Uzel *ExpressionStatement* je následně odstraněn. Výstupem funkce je upravený strom uzlu *VariableDeclaration*. Algoritmus 2 je pseudokód pro normalizaci.

Algorithm 2 normaliseAssignmentExpression(nodeList) - Assign

```
1: for Every variable do
2:     for Every expressionStatement do
3:         if variable.name = expressionStatement.name then
4:             properties ← new key
5:             properties ← new literal
6:             Delete expressionStatement
7:         end if
8:     end for
9: end for
```

5.2.4 Převod ternárního operátoru na podmínkový výraz

Při testování jsem objevil, že minifikační nástroje převádějí v některých případech *podmínkový výraz* na *ternární operátor*. *Podmínkové výrazy* jsou minifikovány dvěma způsoby podle toho, jestli obsahují *else statement* nebo ne. Na výpisu 5.18 je ukázka minifikovaného *podmínkového výrazu* bez *else statementu* a na výpisu 5.20 s *else statementem*. Pokud *if* nebo *else* obsahuje více statementů, které jsou odlišné, minifakční nástroj nepřevede *if* na *ternární operátor* ale pouze odstraní bílé znaky. Na obrázku 5.9 je znázorněn `AST` pro výpis 5.17 a na obrázku 5.10 pro minifikovanou verzi 5.18.

```
if (index !== -1) {
    this.items.splice(index, 1);
}
```

Výpis 5.17: If statement bez else před minifikací

```
-1!==index&&this.items.splice(index,1);
```

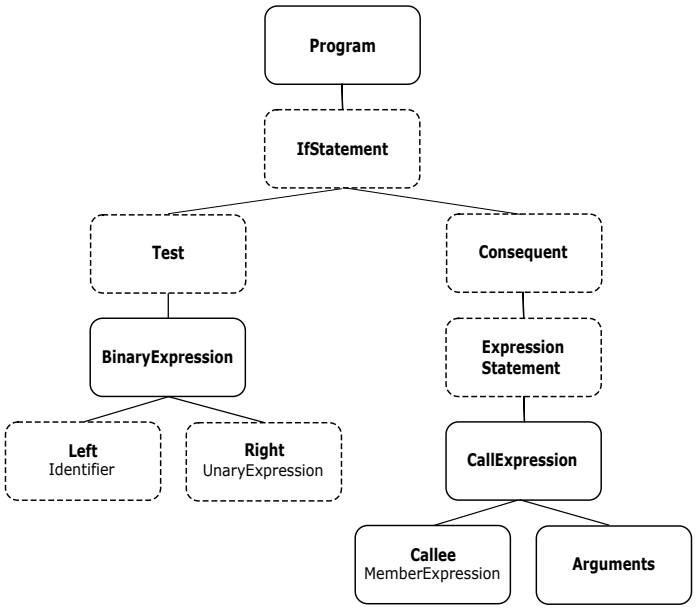
Výpis 5.18: If statement bez else po minifikaci

```
if (index !== -1) {
    this.items.splice(index, 1);
} else {
    console.log();
}
```

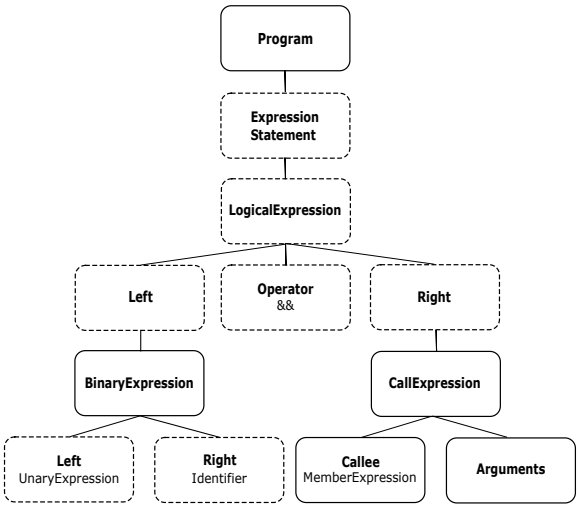
Výpis 5.19: If statement s else před minifikací

```
-1!==index?this.items.splice(index,1):console.log();
```

Výpis 5.20: If statement s else po minifikaci



Obrázek 5.9: Grafická podoba AST výpisu 5.17

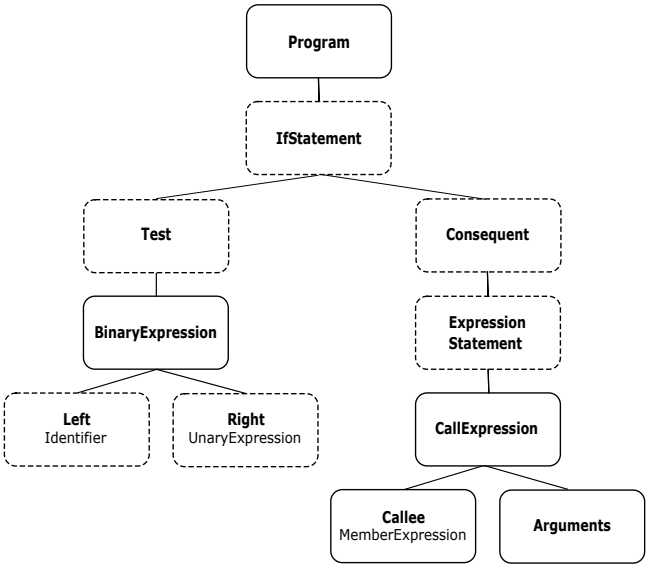


Obrázek 5.10: Grafická podoba AST výpisu 5.18

Minifikační nástroj prohazuje uzly *Left* a *Right* podle jejich typů. Pokud je uzel *Left* typu *Identifier*, nástroj uzly prohodí. Ovšem pokud jsou oba uzly stejné, nástroj je neprohodí.

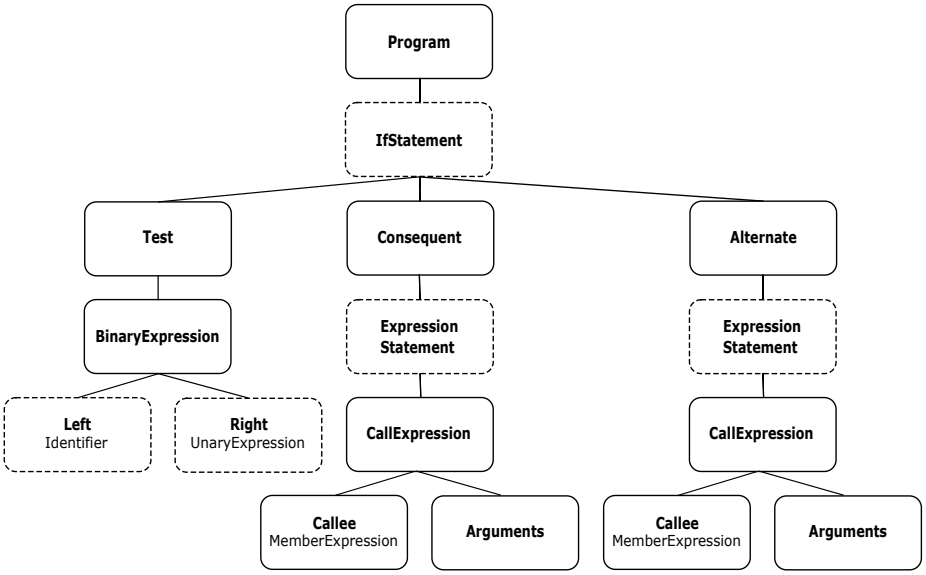
Uvažoval jsem, zda převádět podmínkový výraz na ternární operátor nebo naopak. V prvním případě bych převáděl podmínkový výraz i v situaci, kdy to není potřeba. Z tohoto důvodu jsem normalizoval ternární operátor na podmínkový výraz: Uzel *Expression Statement* a *Operator* jsou odstraněny. Uzly *Left* a *Right*, které mají za rodičovský uzel *LogicalExpression*, jsou transformovány na *Test* a *Consequent*. *LogicalExpression* je změněn na

uzel *IfStatement*. Uzly *Left* a *Right*, které patří pod uzel *BinaryExpression*, jsou prohozeny. Výsledný strom je zobrazen na obrázku 5.11.

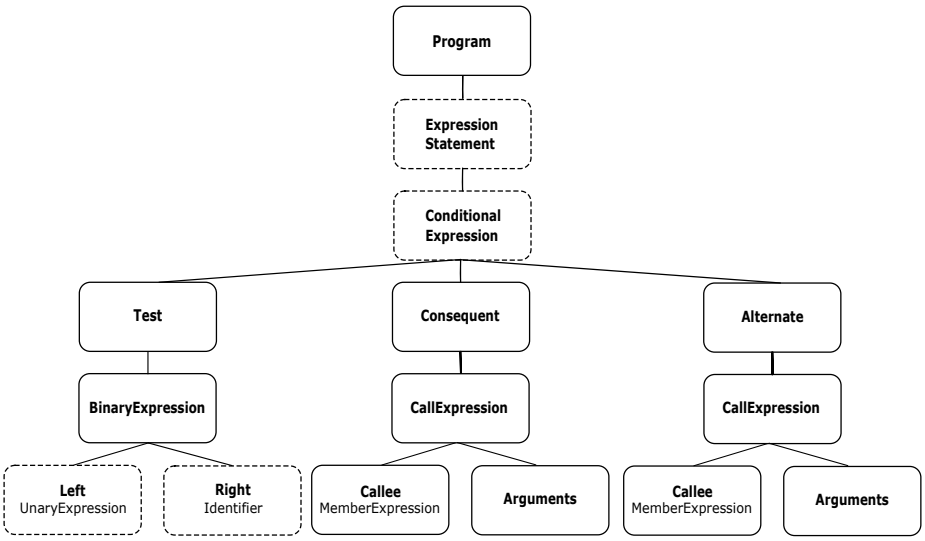


Obrázek 5.11: Grafická podoba AST s normalizací

Abstraktní syntaktické stromy pro *if* s *else* obr. 5.19 a *ternárního operátoru* obr. 5.20 jsou velice podobné, proto došlo jen k pár změnám. Jejich `AST` jsou znázorněny na obrázcích 5.12 a 5.13.



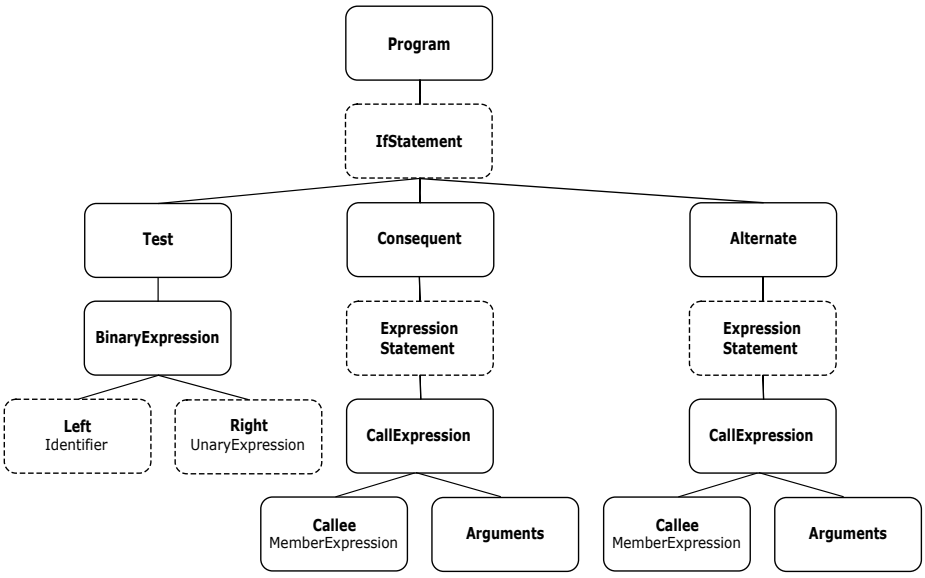
Obrázek 5.12: Grafická podoba AST výpisu 5.19



Obrázek 5.13: Grafická podoba AST výpisu 5.20

Normalizaci jsem opět aplikoval pouze na strom ternárního operátoru 5.13. Uzel *Expression Statement* je odstraněn. *Conditional Expression* je změněn na *IfStatement*. Uzly *Left* a *Right* jsou prohozeny. Nad uzly *CallExpression* jsou přidány uzly *ExpressionStatement*. Výsledný strom je znázorněn na obrázku 5.14.

Normalizace fungují i případě, kdy je v podmínce více výrazů, podmínka vyhodnocuje proměnnou nebo volání funkce s pravdivostní hodnotou.



Obrázek 5.14: Grafická podoba AST s normalizací

Tyto normalizace jsem implementoval do funkcí `normlaiseTernalOperator (expression, node)` a `normaliseANDexpression(expression, node)`. První zmíněná funkce implementuje normalizaci pro minifikovaný *if* s *else* statementem, druhá pro *if* bez *else* statementu. Pro identifikaci správného použití těchto funkcí jsem přidal další funkce `isTernalOperator(expression)` a `isANDExpression(expression)`. Tyto funkce na základě typu dětských uzlů *ExpressionStatement* vrací hodnotu *true* nebo *false*. Pokud ani

jedna z funkcí nevrátí hodnotu *true*, zavolá se funkce pro další normalizaci, která je popsaná dále v této práci 5.2.6.

Algorithm 3 normaliseTernalOperator

```
1: Get expressionStatement
2: if isTernalOperator(expressionStatement.expression) then
3:     normlaiseTernalOperator(expressionStatement.expression, expressionStatement);
4: else if isANDExpression(expressionStatement.expression) then
5:     normaliseANDexpression(expressionStatement.expression, expressionStatement);
6: else
7:     normaliseLogicalExponent(expressionStatement.expression);
8: end if
```

5.2.5 While cyklus na For cyklus

Minifikační nástroje předělávají `while` cyklus na `for` cyklus. Provedl jsem se udělat to stejné. AST pro `while` i `For` jsou velice podobné, tudíž se jedná o opravdu jednoduchou normalizaci. Typ uzlu *WhileStatement* je změněn na *ForStatement*, jsou přidány atributy *init* a *update* oba s hodnotou *null*. Normalizaci jsem aplikoval do funkce `normalseWhileStatement(node)`.

5.2.6 Logické literály na výrazy

Na výpisu 5.21 a 5.22 je znázorněn kód, který je popsaný v předchozí sekci 3.3 této práce. Deobfuskovací nástroj převedl výpis 5.22 na jednu logickou hodnotu `true`. Z tohoto důvodu nemá AST 5.23 deobfuskovaného kódu uzel *LogicalExponent* ale uzel *Literal*. `AST` výpisů 5.21 a 5.22 jsou znázorněny na obrázku 5.15.

```
true || false;
```

Výpis 5.21: Logické lierály před obfuskací

```
!!1 || !{}
```

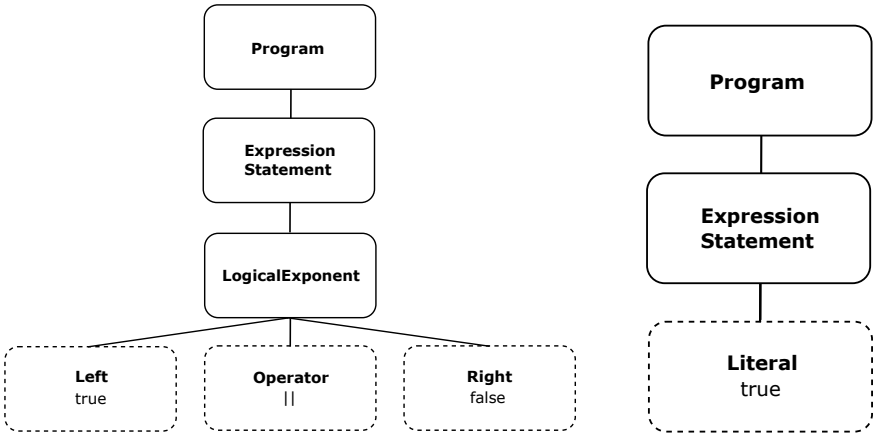
Výpis 5.22: Logické lierály po obfuskaci

```
true;
```

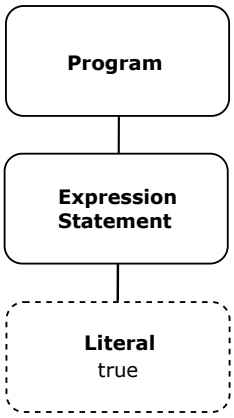
Výpis 5.23: Deobfuskovaná verze výpisu 5.22

Normalizace

Výsledný normalizovaný strom obr. 5.16 vypadá totožně jako strom obr. 5.23 pro deobfuskovaný kód. Pro AST 5.21 původního kódu jsem normalizaci provedl následovně: Na základě hodnot uzlů *Left*, *Operator* a *Right* je vyhodnocena pravdivostní hodnota výrazu `true`. Uzel *LogicalExponent* je nahrazen uzlem *Literal* s příslušnou hodnotou *true*. Normalizovaný strom je znázorněn na obrázku 5.16.



Obrázek 5.15: Grafická podoba AST výpisu 5.21 a 5.23



Obrázek 5.16: Grafická podoba AST s normalizací

Implementace

Normalizaci jsem aplikoval ve funkci `normaliseLogicalExponent(expression)`. Parametr *expression* je výraz, který je uložený v uzlu *ExpressionStatement*, hodnota může být *LogicalExpression* nebo *ConsitionalExpression*. V této normalizaci se jedná o *LogicalExpression*. Podle uzlu *operátor* se vyhodnotí výsledek operace a vytvoří se nový uzel *Literal* s hodnotou výsledku. Uzel *LogicalExponent* je odstraněn a nahrazen novým uzlem *Literal*.

5.2.7 Znak na ternární operátor

Výpis 5.24 je ukázka znaku a výpis 5.25 je jeho obfuskovaná verze. Deobfuskační nástroj převedl obfuskovaný kód 5.25 na jednodušší podobu ternárního operátoru 5.26, ze kterého se dá snadno určit výsledná hodnota.

```
'a';
```

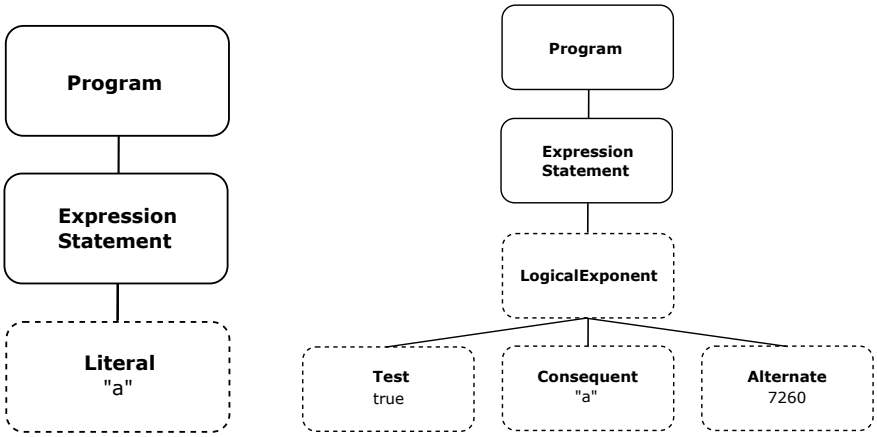
Výpis 5.24: Znak na ternární operátor před obfuskací

```
665.01 > 6770 ? 'Q' : (2170, 1050) !== 805 ? (9480, 813.32) >= 5227 ?
754.03 : 'a' : 7.26e+3
```

Výpis 5.25: Znak na ternární operátor po obfuskací

```
true ? "a" : 7260;
```

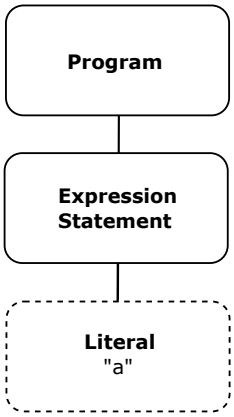
Výpis 5.26: Deobfuskovaná verze výpisu 5.25



Obrázek 5.17: Grafická podoba AST výpisu 5.24 a 5.26

Normalizace

Normalizovaný abstraktní syntaktický strom je stejný jako strom 5.17 před obfuskací kódu. Pro strom po deobfuskaci je provedena transformace: Podstrom s kořenem *ConditionalExpresion* má uzly *Test*, *Consequent* a *Alternate*. Na základě hodnoty uzlu *test* se vybere hodnota z uzlu *Consequent* nebo *Alternate*. Tato hodnota se vloží do nového uzlu *Literal*, který nahradí uzel *ConditionalExpression* a jeho potomky. Normalizovaný strom je na obrázku 5.18.



Obrázek 5.18: Grafická podoba AST s normalizací

Implementace

Normalizaci jsem implementoval ve funkci `normaliseLogicalExponent(expression)` stejně jako normalizaci `Logické literály na výrazy` 5.2.6. Parametr *expression* v uzlu

ExpressionStatement nabývá hodnoty *ConditionalExpression*. Výsledek je vyhodnocen podle uzlu *test*, který je následně vložen do nového uzlu *Literal*. Ten nahradí uzel *LogicalExponent*.

Algoritmus 4 zobrazuje pseudokód normalizace logických výrazů, který je použit pro tuto normalizaci 5.2.7 a normalizaci 5.2.6.

Algorithm 4 normaliseLogicalExponent(expression)

```
1:  if expression = LogicalExpression then
2:      if operator = OR then
3:          result ← left.value OR right.value
4:      else if operator = AND then
5:          result ← left.value AND right.value
6:      end if
7:  else if expression = ConditionalExpression then
8:      if test = true then
9:          result ← consequent.value
10:     else if test = false then
11:         result ← alternate.value
12:     end if
13: end if
```

5.2.8 Rozložení operátoru čárky

Výpis 5.27 zobrazuje ukázkový kód, obfuskovaná verze se nachází na výpisu 5.28. Deobfuskační nástroj neprovedl žádné změny. Normalizace se tedy vztahuje pro obfuskovaný kód. `AST` výpisů se nacházejí na obrázcích 5.19 a 5.20.

```
(foo = 1, bar = 2, baz = 3);
```

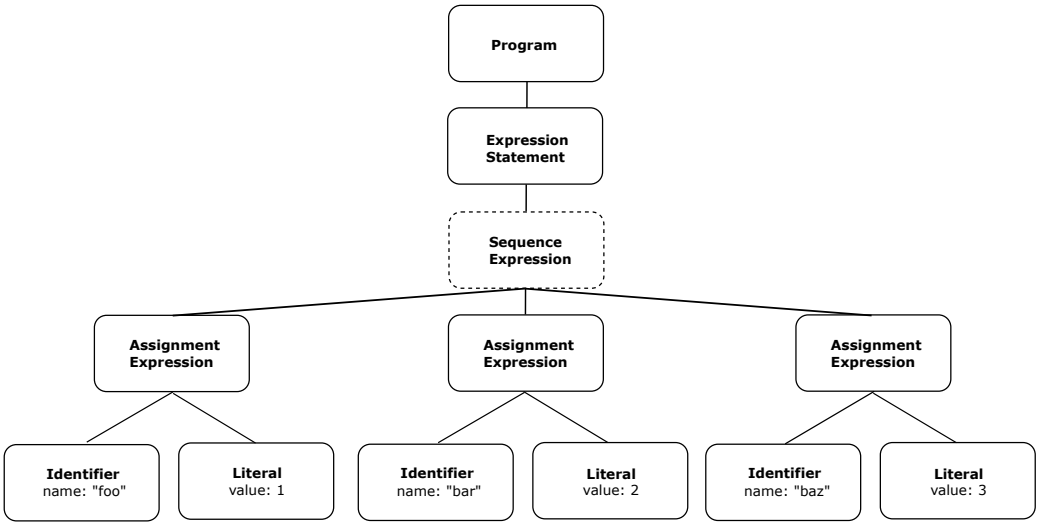
Výpis 5.27: Rozložení operátoru čárky před obfuskací

```
foo = 1;
bar = 2;
baz = 3;
```

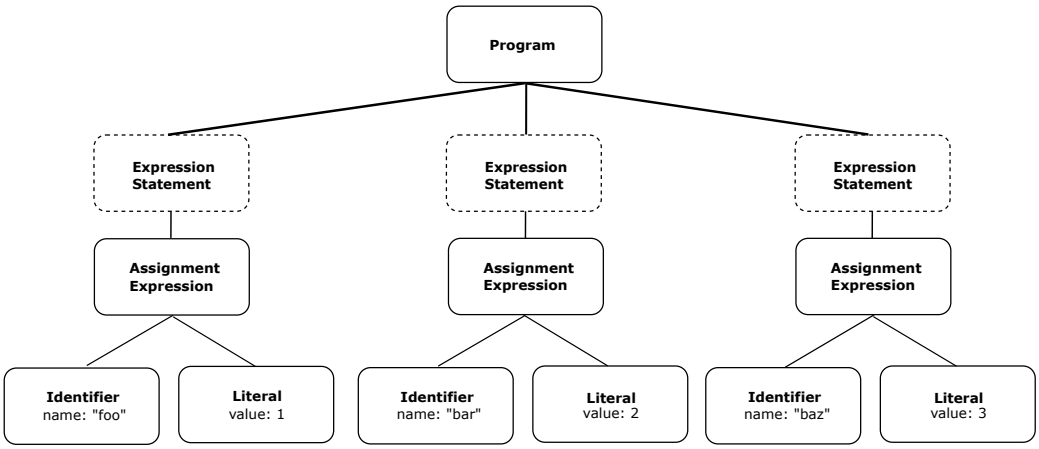
Výpis 5.28: Rozložení operátoru čárky po obfuskaci

Normalizace

Stromy se liší jenom v tom, jestli uzly `AssignmentExpression` patří pod `SequenceExpression`. Vložil jsem tedy uzly `AssignmentExpression` do uzlu `SequenceExpression` : Uzly *ExpressionStatement* jsou sloučeny do jednoho uzlu, pod kterým je vytvořen uzel reprezentující sekvenci *SequenceExpression*. Normalizace je znázorněna na obrázku 5.21



Obrázek 5.19: Grafická podoba AST výpisu 5.27



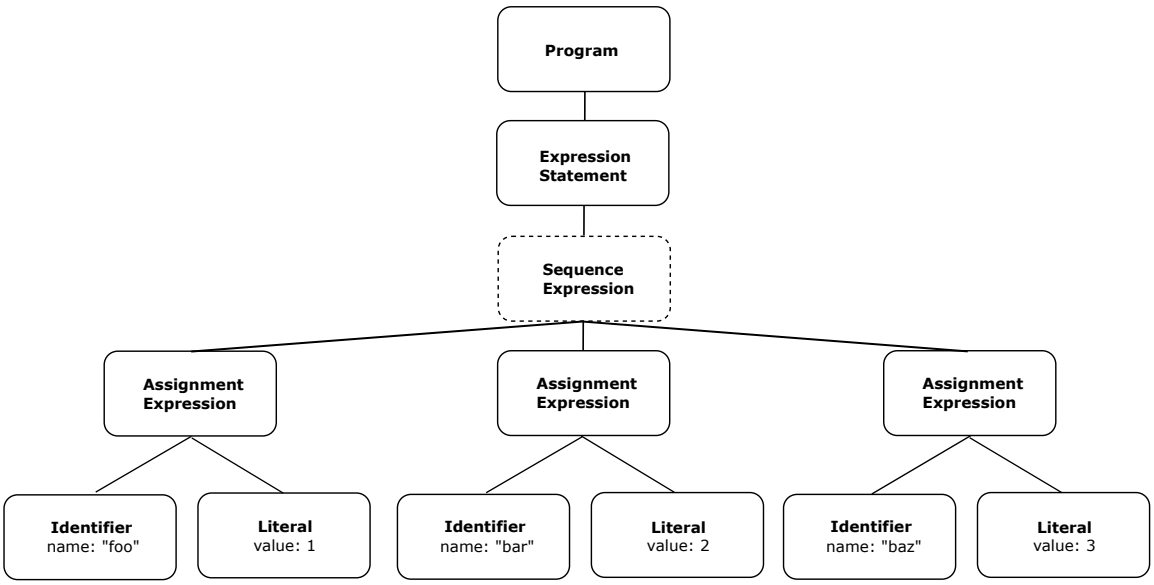
Obrázek 5.20: Grafická podoba AST výpisu 5.28

Implementace

Pro implementaci normalizace jsem vytvořil funkci `normaliseSequenceExpression(nodeList)`, parametr *nodeList* je seznam dětských uzlů rodičovského uzlu. Jsou získány uzly *AssignmnetExpression* a je vytvořen nový uzel *SequenceExpression*, který má atribut *expressions* obsahující list již zmíněných uzlů *AssignmnetExpression*. Již nepotřebné uzly *ExpressionStatement* jsou odstraněny. Algoritmus 6 je pseudokód pro normalizaci expression statementů s konkatenací

Algorithm 5 normaliseSequenceExpression(nodeList)

1: Get assignmnetExpressions
2: New sequenceExpression
3: $sequenceExpression.expressions \leftarrow assignmnetExpressions$
4: Delete expressionStatements



Obrázek 5.21: Grafická podoba AST s normalizací

5.2.9 Dělení řetězců

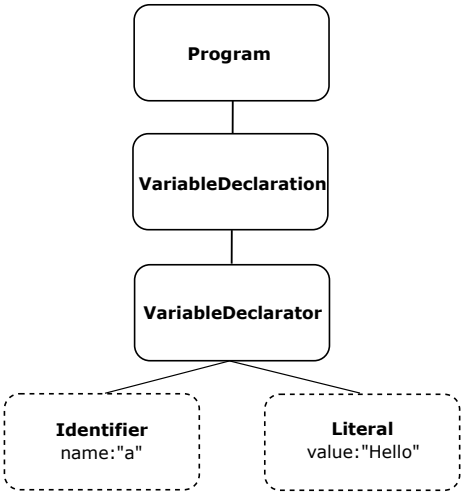
Ukázkový kód pro řetězec je na výpisu 5.29, jeho obfusovaná verze se nachází na výpisu 5.30. V tomto případě deobfuskační nástroj neprovedl s obfuskovaným kódem 5.30 žádnou změnu, tudíž je zapotřebí znormalizovat obfuskovaný kód.

```
var a = "Hello";
```

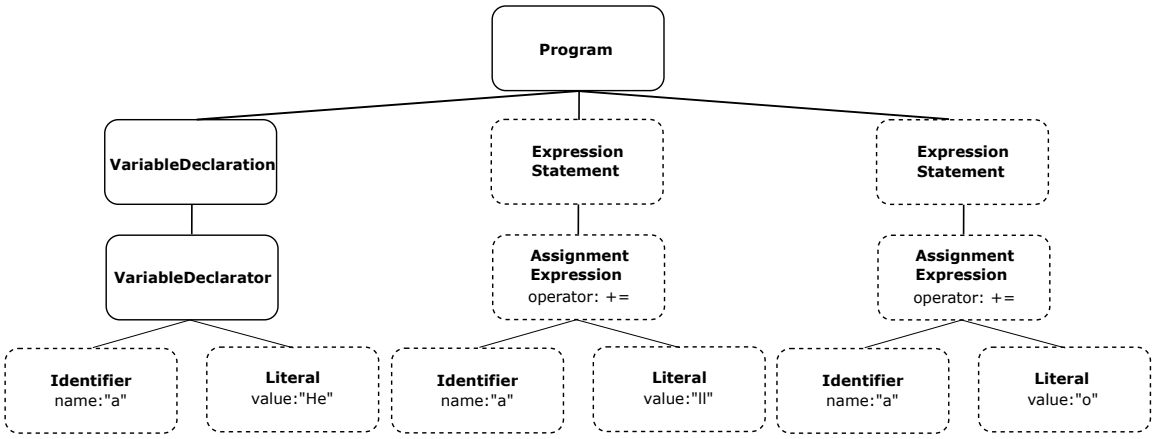
Výpis 5.29: Dělení řetězců před obfuskací

```
var a = "He";
a += "ll";
a += "o";
a;
```

Výpis 5.30: Dělení řetězců po obfuskací



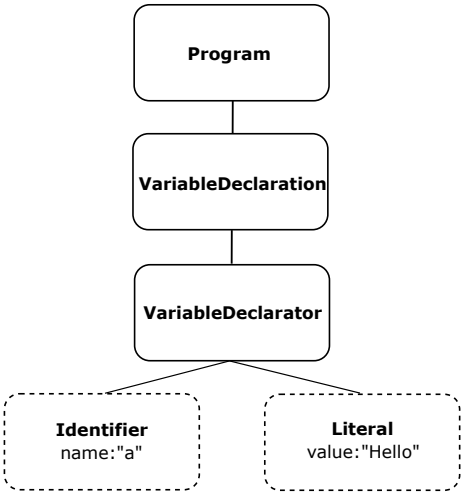
Obrázek 5.22: Grafická podoba AST výpisu 5.29



Obrázek 5.23: Grafická podoba AST výpisu 5.30

Normalizace

Na strom 5.23 jsem vytvořil normalizace: Hodnota *name* ve všech uzlech *Identifier* je stejná a podstromy s kořeny *VariableDeclaration* a *ExpressionStatement* jsou na totožné úrovni. V podstromech s kořenem *ExpressionStatement* je na hodnotách uzlů *Literal* provedena operace podle uzlů *AssignmentExpression*. Výsledek je vložen do uzlu *Literal* v podstromu s kořenem *VariableDeclaration*. Podstromy s kořeny *ExpressionStatement* jsou následně odstraněny. Znormalizovaný strom 5.24 je stejný jako strom 5.22 pro kód před obfuskací.



Obrázek 5.24: Grafická podoba AST s normalizací

Implementace

Normalizaci jsem implementoval do funkce `normaliseAssignmentExpression(nodeList)` stejně jako v normalizaci 5.8. Začátek je stejný, pro každou proměnnou se hledá uzel *ExpressionStatement* se stejným jménem. V tomto případě je hodnota operace *concatenation*, proto je nad atributem *value* příslušné proměnné provedena konkatenace s uzlem *Literal* uzlu *ExpressionStatement*. Výsledek je uložen do zmíněného atributu *value* a uzel *ExpressionStatement* je odstraněn. Algoritmus 6 je pseudokód pro normalizaci expression statementů s konkatenací.

Algorithm 6 normaliseAssignmentExpression(nodeList) - Concatenation

```
1: Get variables
2: Get expressionStatements
3: for Every variable do
4:     for Every expressionStatement do
5:         if variable.name = expressionStatement.name then
6:             variable.value ← variable.value += expressionStatement.value
7:             Delete expressionStatement
8:         end if
9:     end for
10: end for
```

5.2.10 Nepřímost globální proměnné

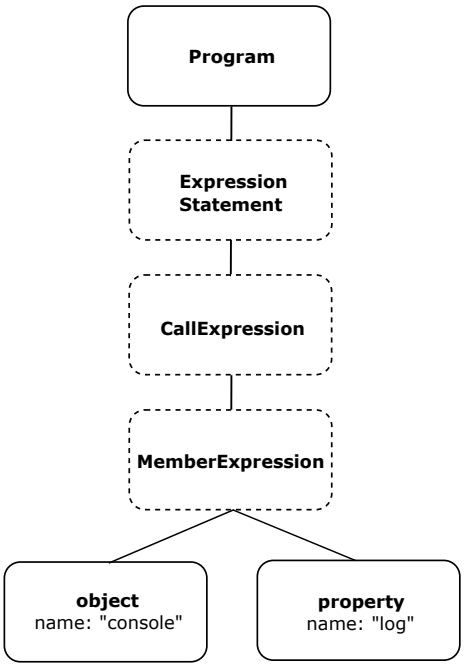
Referenční kód je znázorněn na výpisu 5.31, obfuskovaná verze se nachází na výpisu 5.32. Deobfuskační nástroj opět v tomto případě neprovedl žádné změny. Normalizace se vztahuje tedy pro obfuskovaný kód. Grafické znázornění `AST` pro oba výpisy je zobrazeno na obrázcích 5.25 a 5.26.

```
console.log();
```

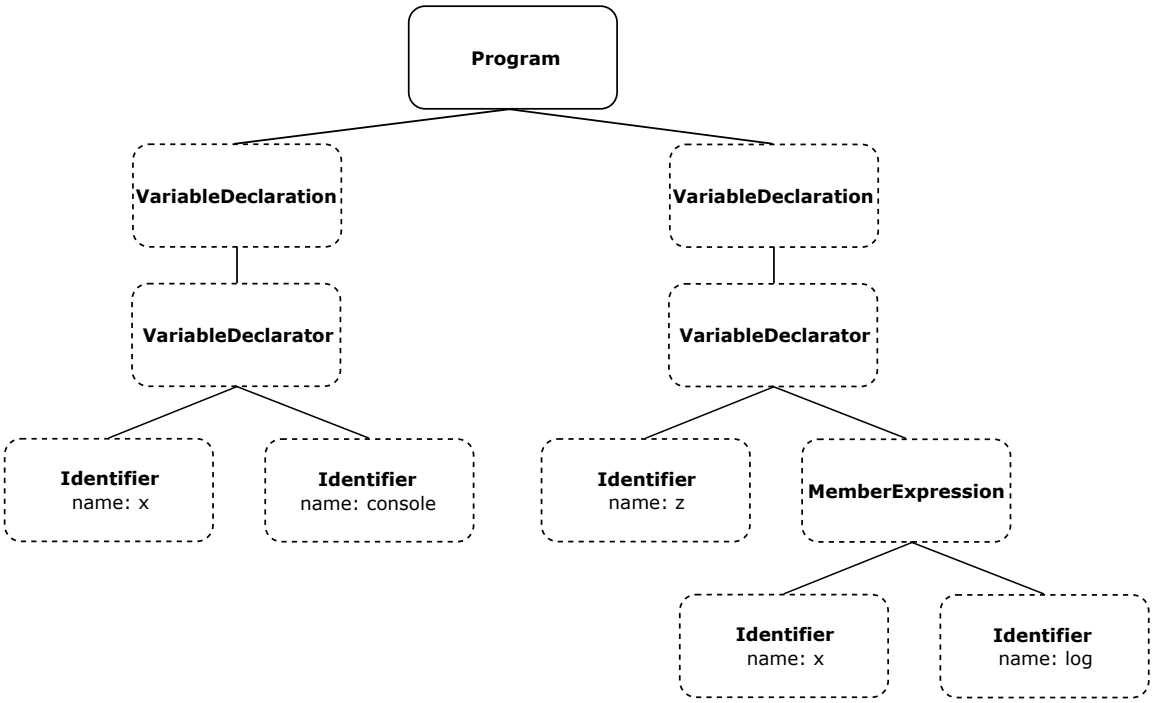
Výpis 5.31: Nepřímost globální proměnné před obfuskací

```
var x = console;
var y = ["w", "o", "n", "d", "e", "r", "l", "a", "n", "d", "g"];
var z = x[y[6]+y[1]+y[10]];
```

Výpis 5.32: Nepřímost globální proměnné po obfuskaci



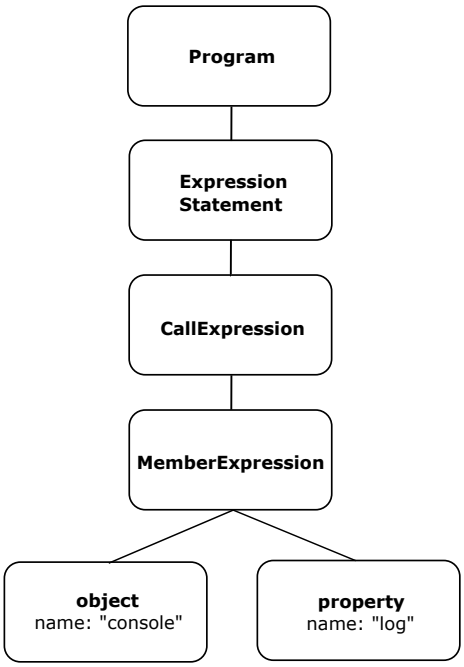
Obrázek 5.25: Grafická podoba AST výpisu 5.31



Obrázek 5.26: Grafická podoba AST výpisu 5.32

Normalizace

Normalizaci druhého stromu 5.26 jsem vytvořil následovně: V pravém podstromu s kořenem *VarialeDeclaration* je proměnná *z* obsahující uzel *MemberExpression*, který má uzel *Idetifier*. Tento uzel odkazuje na proměnnou *x* v levém podstromu s kořenem *VaribaleDeclaration*. Oba zmíněné podstromy jsou smazány a jsou nahrazeny uzly *ExpressionStatement*, *CallExpression* a *MemberExpression*. *MemberExpression* má dva poduzly: *object* a *property*. Uzel *object* obsahuje hodnotu proměnné *x* a uzel *property* proměnné *z*. Normalizace je znázorněna na obrázku 5.27.



Obrázek 5.27: Grafická podoba AST s normalizací

Implementace

Funkce `normaliseGlobalVariableIndirectness(nodeList)` se stará o implementaci této normalizace. Nejprve jsou získány všechny uzly *VariableDeclaration*, které jsou rozděleny na proměnné obsahující uzel *MemberExpression* a na proměnné, které jej neobsahují. Pro každou proměnnou s *MemberExpression* se iteruje mezi proměnnými bez *MemberExpression*. Pokud má uzel *Identifier* proměnné s *MemberExpression* stejné jméno jako proměnná bez *MemberExpression*, vytvoří se nový uzel *ExpressionStatement*. Tento uzel obsahuje uzel *CallExpression*, který obsahuje *MemberExpression* a ten obsahuje uzly *object* a *property* s příslušnými atributy. Uzly typu *VariableDeclaration* jsou odstraněny. Algoritmus 7 je pseudokód pro normalizaci expression statementů s konkatenací.

Algorithm 7 normaliseGlobalVariableIndirectness(nodeList)

```
1: Get variableDeclarations
2: Get variables
3: Get variablesWithMember
4: for Every variableWithMember do
5:     for Every variable do
6:         if variableWithMember.init.name = variable.name then
7:             New expressionStatement
8:             Delete VariableDeclarations
9:         end if
10:    end for
11: end for
12: sequenceExpression.expressions ← assignmnetExpressions
13: Delete expressionStatements
```

Kapitola 6

Testování

Samotné vytvoření softwarového produktu nestačí, je třeba ho také řádně otestovat, aby bylo zajištěno jeho kvalitní a bezproblémové fungování. V této práci jsem se zaměřil především na jednotkové testy, integrační testy a test celého rozšíření pro prohlížeč Chrome.

6.1 Jednotkové testy

Pro prvotní testování funkcionality jsem využil jednotkových testů. K implementaci těchto testů jsem použil testovací soubor `normalise.test.js`, který byl v knihovně `js-to-ast` v souboru `test` implementovaný panem Randýskem.

V tomto souboru byly již Randýskem vytvořeny testy *String quotes normalisation*, *Variable declaration merging* a *Variable declaration test*. Pomocí těchto testů jsem zjistil, že mnou implementované normalizace nekolidují s Randýskovými normalizacemi.

K jednotlivým testům jsem vymyslel úryvek kódu pro danou normalizaci a tento úryvek jsem manuálně zminifikoval nebo zobfuskoval. Oba kódy jsem vložil do funkce `tryParse(code)`, která vstupní kód zdeobfuskuje, převede na *syntaktický strom* a následně jej znormalizuje. Funkce vrací výsledný normalizovaný strom. Stromy obou kódů jsou převedeny do řetězcové podoby a jsou porovnány. Na výpisu 6.1 je ukázka minifikačního jednotkového testu pro podmínkový výraz.

```
test("If - test 1", () => {
    const input1 = `
    if (index !== -1) {
      this.items.splice(index, 1);
    }`;
    const input2 = `-1!==index&&this.items.splice(index,1);`;

    let parsed1 = sut.tryParse(input1);
    let parsed2 = sut.tryParse(input2);

    let stringified1 = stringifyAndClean(parsed1);
    let stringified2 = stringifyAndClean(parsed2);

    expect(stringified1).toBe(stringified2);
})
```

Výpis 6.1: Jednotkový test

6.2 Integrační testy

Pomocí integračních testů jsem zkoumal, jestli jednotlivé normalizace zvládnou fungovat pohromadě s ostatními. Vygeneroval jsem několik náhodných testovacích `JavaScript` kódů pomocí online generátoru `codepal`[1]. Kontrola probíhala stejně jako u *jednotkových testů*. Kód jsem zminifikoval nebo zobfuskoval a porovnal jsem znormalizované syntaktické stromy.

V některých případech po normalizaci dochází k jinému výslednému abstraktnímu syntaktickému stromu, než je ukázáno v návrhu normalizace sekce 5.1. Například při normalizaci sekce 5.2.10 (Nepřímost globální proměnné) má ve svém výsledném stromu navíc uzel *SequenceExpression*, jelikož došlo také k normalizaci sekce 5.2.8 (Rozložení operátoru čárky). Jedná se o chtěné a předpokládané chování.

Testy pro obfuskaci byly téměř ve všech případech neúspěšné. Zjistil jsem, že obfuskační nástroje jsou nedeterministické, tedy vždy generují jiný obfuskovaný kód. Deobfuskační nástroj jej málokdy převede do takové podoby, aby si s ním navržené normalizace byly schopny poradit. Největší problém je nadbytečný kód, který nemá na funkcionalitu vliv.

Při testování jsem objevil minifikační techniku, jejíž normalizace nebyly implementovány, jelikož nebyly popsány v dokumentaci o nástroji a neměl jsem již dostatek času ji nastudovat, navrhnout normalizaci a implementovat ji. Na výpisu 6.2 je ukázána funkce, která vrací vytvořený a stylizovaný *div* element. Výpis 6.3 ukazuje minifikovanou verzi této funkce.

```
function createNewDiv() {
    const myDiv = document.createElement('div');
    myDiv.style.backgroundColor = generateRandomColor();
    myDiv.style.width = randomNumber(50, 150) + 'px';
    myDiv.style.height = randomNumber(50, 150) + 'px';
    return myDiv;
}
```

Výpis 6.2: Retrun statement

```
function createNewDiv(){const e=document.createElement("div");
return e.style.backgroundColor=generateRandomColor(),e.style.width=
randomNumber(50,150)+"px",e.style.height=randomNumber(50,150)+"px",e}
```

Výpis 6.3: Retrun statement po minifikaci

Na minifikované verzi 6.3 je vidět, že minifikační nástroj vložil stylizování `Div` elementu do *return* statementu. To změnilo `AST` oproti neminifikované verzi kódu.

6.3 Test rozšíření pro prohlížeč Chrome

Pro otestování efektivity rozšíření jsem se pokusil vyžít *web crawl* verzi pana Randýska. Tato verze se mi bohužel nepodařila zprovoznit, proto jsem si napsal jednoduchý script, který prochází seznam nejnavštěvovanějších stránek [9]. Script otevírá jednotlivé stránky podle indexu na nové kartě prohlížeče, čeká 5 vteřin, zavře kartu a otevře další stránku. Testoval jsem stránky, které mají v seznamu index 500 000 až 501 000. Nejdříve jsem testoval rozšíření s mnou navrženými normalizacemi a poté jsem pro porovnání prošel stránky s původním rozšíření bez aplikovaných normalizací, oboje v módu *analyze*. Tabulka 6.1 ukazuje porovnání výsledků.

[1] https://codepal.ai/code-generator/javascript

Rozšíření	Počet zpracovaných skriptů	Počet zranitelností
Původní	12722	333
Normalizované	10634	125

Tabulka 6.1: Výsledky testu.

Bylo prokázáno, že rozšíření, které mělo aplikované normalizace, odhalilo mnohem méně zranitelností. Jedním z důvodům je, že rozšíření s normalizací potřebuje ke zpracování skriptů stránky více času, proto mnohdy rozšíření nestihlo zpracovat zdrojový kód, než se stránka zavřela. To je ukázáno na počtu zpracovaných skriptů. Testování ukázalo, že použitý deobfuskační nástroj `JavaScript Deobfuscator` často generuje kód, který není validní a knihovna `Acorn AST walker` jej nedokáže zpracovat, vrací chybu *Assigning to an rvalue* označující špatné přiřazení hodnoty. Na výpisu 6.4 je ukázka kódu, který tuto chybu vyvolá.

```
// Attempting to assign a value to an rvalue
10 = 20;
```

Výpis 6.4: Assigning to an rvalue

Dalším důvodem, proč rozšíření našlo méně zranitelností je, že různé minifikační a obfuskační nástroje mohou používat jiné techniky, na které normalizace nejsou navržené. Dle mého názoru v některých případech mohou normalizace spíše uškodit.

6.4 Zhodnocení a možná pokračování práce

Navrhl a implementoval jsem normalizace pro abstraktní syntaktické stromy minifikovaného a obfuskovaného kódu. Značným problémem je, že nástroje pro minifikaci a obfuskaci mohou používat rozdílné metody změny vstupního kódu. Pokusil jsem se proto zpracovat pouze často používané techniky nebo ty, které jsem objevil při testování nástrojů. Pro ulehčení normalizace technik obfuskace, jsem využil deobfuskační nástroj, který vrací obfuskovaný kód částečně do původního stavu. Za zmínku stojí normalizace `Přejmenování identifikátorů` sekce 5.2.1, kde jsem navrhl systematické přejmenování identifikátorů, nebo normalizace `nepřímost globální proměnné` sekce 5.2.10, která mění skoro celý syntaktický strom pro daný úsek kódu. Normalizace jsem testoval jednotkovými a integračními testy. Pro závěrečné otestování funkcionality jsem normalizace vyzkoušel společně s rozšířením pro prohlížeč Chrome pana Randýska v režimu *Analyze*. Rozšíření odhaluje méně chyb než původní původní verze pana Randýska. Je to způsobeno především deobfuskačním nástrojem, který často generuje chybný kód a také málokdy převede kód do podoby, pro kterou jsou normalizace navrženy. Na základě těchto výsledků, považuji práci za splněnou.

Jedním z rozšíření práce může být normalizování technik a případů, které jsou v práci zmíněny, ale nejsou zpracovány. Také nastudovat další, méně známé obfuskační a minifikační techniky a navrhnout pro ně normalizace.

Další možné rozšíření je vytvořit deterministický deobfuskační nástroj, který generuje především validní kód a dokáže odstranit nadbytečný kód, který nemá na funkcionalitu žádný vliv. Jedná se, dle mého názoru o velmi složité rozšíření, jelikož obfuskace mění zdrojový kód tak, aby člověk ani nástroj nebyl schopen vrátit změny do původní podoby. Jde o rozšíření, které je spíše pro skupinu, než-li pro jedince.

Kapitola 7

Závěr

Tato práce se zabývá normalizací abstraktních syntaktických stromů. Jedná se o rozšíření diplomové práce pana Randýska na téma *Detekce kódu v jazyce JavaScript se známými bezpečnostními chybami*. Navrhl jsem normalizace pro časté minifikační a obfuskační techniky, nebo jsem je objevil při implementaci a testování. Vyzdvihl bych normalizaci *přejmenování identifikátorů*, ve které jsem navrhl systematické přejmenování identifikátorů, nebo normalizaci *nepřímost globální proměnné* pro její složitost.

Pro zjednodušení obfuskovaného kódu jsem využil deobfuskovačního nástroje *JavaScript Deobfuscator*, který částečně vrací kód do původního stavu. Normalizace jsou implementovány v knihovně *js-to-ast*, v souboru *finder.js*. Vstupní kód je nejprve deobfuskován a následně je pomocí nástroje *Acorn AST walker* převeden na abstraktní syntaktický strom. Tento strom je normalizován a použit pro detekci chyb v kódu.

Normalizace jsem testoval pomocí jednotkových a integračních testů. Manuálně jsem minifikoval nebo obfuskoval vstupní kód, který jsem znormalizoval a porovnal s normalizovanou verzí původního kódu. V poslední řadě jsem testoval samotné rozšíření pro prohlížeč. Při tomto testu jsem zjistil, že použitý deobfuskační nástroj často generuje nevalidní JavaScriptový kód, což způsobuje chybu při tvorbě stromu. Rozšíření jsem automatizovaně vyzkoušel na 1000 stránkách a výsledky jsem porovnal s původním rozšířením bez mých normalizací. Původní rozšíření vyhledalo 333 zranitelností, rozšíření s normalizacemi pouze 125. Toto je převážně způsobeno chybou deobfuskačního nástroje.

Zpětně bych použil jiný deobfuskační nástroj. Chybovost použitého nástroje jsem zjistil příliš pozdě. Z časových důvodů nebylo možné nastudovat způsob deobfuskace jiného nástroje a navrhnout nové normalizace.

Budoucí pokračování práce by mohlo obsahovat normalizaci dalších minifikačních a normalizačních technik. Složitější pokračování by mohlo být vytvoření deobfuskačního deterministického nástroje, který by dokázal především odstranit nadbytečný kód, který nemá vliv na funkcionalitu a který by generoval validní kód.

Literatura

[1] Barker, T. *Pro JavaScript Performance: Monitoring and Visualization*. Apress, 2012 [cit. 2022-11-05]. Expert's voice in Web development. ISBN 9781430247500. Dostupné z: https://books.google.cm/books?id=pWgatPcRfKYC.

[2] Brunton, F. a Nissenbaum, H. *Obfuscation: A User's Guide for Privacy and Protest*. MIT Press, 2016 [cit. 2022-12-17]. ISBN 9780262529860. Dostupné z: https://books.google.cz/books?id=eGIrEAAAQBAJ.

[3] Crockford, D. *What JSMin Does* [online]. [cit. 2022-11-18]. Dostupné z: https://www.crockford.com/jsmin.html.

[4] Electric, S. *Metric: Halstead Complexity* [online]. [cit. 2023-01-05]. Dostupné z: https://olh.schneider-electric.com/Machine%20Expert/V1.2/en/CodeAnly/CodeAnly/Metrics/Metrics-9.htm.

[5] Foundation, M. *What is JavaScript?* [online]. [cit. 2023-01-12]. Dostupné z: https://developer.mozilla.org/en-US/docs/Learn/JavaScript/First_steps/What_is_JavaScript.

[6] Google. *Google Closure Compiler*. [cit. 2022-11-19]. Dostupné z: https://developers.google.com/closure/compiler.

[7] Jscrambler. *Code Integrity Transformations* ["online"]. [cit. 2022-12-10]. Dostupné z: https://docs.jscrambler.com/code-integrity/documentation/transformations.

[8] Jscrambler. *JavaScript Obfuscation: The Definitive Guide*. [cit. 2022-12-17]. Dostupné z: https://blog.jscrambler.com/javascript-obfuscation-the-definitive-guide.

[9] Le Pochat, V., Van Goethem, T., Tajalizadehkhoob, S., Korczyński, M. a Joosen, W. Tranco: A Research-Oriented Top Sites Ranking Hardened Against Manipulation. In: *Proceedings of the 26th Annual Network and Distributed System Security Symposium*. únor 2019. NDSS 2019. DOI: 10.14722/ndss.2019.23386.

[10] Megida, D. *JavaScript Minify – Minifying JS with a Minifier or jsmin* [online]. [cit. 2022-11-15]. Dostupné z: https://www.freecodecamp.org/news/javascript-minify-minifying-js-with-a-minifier-or-jsmin/.

[11] Odell, D. *Pro JavaScript Development: Coding, Capabilities, and Tooling*. Apress, 2014 [cit. 2022-11-16]. Expert's voice in Web development. ISBN 9781430262695. Dostupné z: https://books.google.cz/books?id=pyZIBAAAQBAJ.

[12] Randýsek, V. *Detekce kódu v jazyce JavaScript se známými bezpečnostními chybami*. Brno, CZ, 2022. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Dostupné z: https://www.fit.vut.cz/study/thesis/23311/.

Příloha A

Obsah paměťového média

V této příloze je popsaná struktura přiložené SD karty. Karta obsahuje rozšíření pro prohlížeč Chrome pana Randýska. V souboru `js-to-asc/src/finder.js` jsou implementované vlastní normalizace i normalizace pana Randýska, které jsou popsané v kapitole 5. Složka `test-web-script` obsahuje testovací stránku, která na nové kartě prochází webové stránky ze seznamu nejnavštěvovanějších stránek. Soubor `README.txt` obsahuje návodem k lokální instalaci rozšíření a spuštění webu k průchodu stránek. Ve složce `DOC` je dokument `BP.pdf` (tento dokument) a zdrojové soubory této zprávy.