FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Deminifikace a deobfuskace JavaScriptu
Student:	Adam Platkevič
Vedoucí:	Ing. Radomír Polách
Studijní program:	Informatika
Studijní obor:	Teoretická informatika
Katedra:	Katedra teoretické informatiky
Platnost zadání:	Do konce letního semestru 2018/19

Pokyny pro vypracování

1) Nastudujte problematiku minifikace a obfuskace jazyka JavaScript.
2) Seznamte se s knihovnami espree a escodegen pro parsování a generování jazyka JavaScript na platformě NodeJS.
3) Navrhněte, analyzujte a implementujte efektivní postup deminifikace a deobfuskace pro Jazyk Javascript za použití zmíněných knihoven.
4) Implementaci testujte na minifikovaných a/nebo obfuskovaných programech dodaných vedoucím práce.

Seznam odborné literatury

Dodá vedoucí práce.

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

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

V Praze dne 9. ledna 2018

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalářská práce

Deminifikace a deobfuskace JavaScriptu

Adam Platkevič

Katedra teoretické informatiky
Vedoucí práce: Ing. Radomír Polách

14. května 2018

Prohlášení

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

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

V Praze dne 14. května 2018 ……

Odkaz na tuto práci

Platkevič, Adam. *Deminifikace a deobfuskace JavaScriptu*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2018.

Abstrakt

Jazyk JavaScript je výhradní prostředek skriptování v rámci webovských stránek. Aby chránili své duševní vlastnictví či skryli nekalý úmysl, někteří programátoři přistupují u svého kódu k záměrnému snížení jeho srozumitelnosti, tj. tzv. obfuskaci. V rámci této práce analyzujeme používané obfuskační metody a následně navrhneme a implementujeme čistě statický nástroj na zvýšení čitelnosti takto obfuskovaných skriptů, tedy tzv. deobfuskátor JavaScriptu. Nakonec ukážeme, že náš nástroj má uspokojivou úspěšnost při deobfuskaci častých obfuskačních metod.

Klíčová slova deobfuskace, JavaScript, návrh a implementace deobfuskátoru, statická analýza, node.js

Abstract

JavaScript is the only language used for website scripting. Some programmers choose to obfuscate their codes to protect their intelectual property or to hide their malicious intents. In this thesis we analyze common obfuscation techniques and implement a purely static deobfuscator of the JavaScript language. Finally we demonstrate it's effectivity on common obfuscation patterns.

Keywords deobfuscation, JavaScript, deobfuscator design and implementation, static analysis, node.js

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Jazyku JavaScript se pro jeho stále se rozšiřující využití v posledních letech dostává značné pozornosti. Už dávno není pouze jazykem pro implementaci jednoduchých interaktivních prvků v rámci webových stránek. Stal se nástrojem pro tvorbu plnohodnotných online i offline aplikací, programových doplňků a je používán dokonce i pro programování vestavných zařízení. Obzvláště stojí za pozornost skutečnost, že každý osobní počítač i chytrý telefon dnes disponuje webovým prohlížečem schopným programy psané v jazyce JavaScript spouštět.

V důsledku toho se u JavaScriptu setkáváme s fenomény vlastními jakékoliv široce rozšířené platformě, zejména s nabídkou komerčních produktů a se škodlivým software. V obou těchto případech často přistupují tvůrci k tzv. *obfuskaci* kódu, tedy typicky automatizovanému snížení srozumitelnosti kódu při zachování funkcionality programu. Srozumitelnost, o které je řeč, může být jak ve smyslu čitelnosti pro lidského čtenáře, tak ve smyslu analyzovatelnosti automatickým nástrojem (např. antivirovým software). Tvůrci komerčních produktů tímto krokem sledují převážně ochranu svého duševního vlastnictví, softwaroví útočníci pak snížení odhalitelnosti svého nekalého úmyslu.

Analýza vzorků škodlivého javascriptovského kódu odhaluje celou řadu užívaných obfuskačních metod. V této práci některé nalezené metody prozkoumáme a navrhneme účinný nástroj pro částečné navrácení čitelnosti předloženému kódu — *deobfuskaci* — využívající výhradně statickou analýzu syntaxe. Je třeba také podotknout, že naší snahou není co nejlépe vystihnout podobu kódu před obfuskací; postačí, pokud bude deobfuskovaný kód snáze interpretovatelný pro lidského čtenáře.

V této práci se budeme věnovat především deobfuskaci škodlivého kódu, abychom se vyhnuli nezbytným etickým a legislativním otázkám spojeným s rozkrýváním kódu obfuskovaného za účelem ochrany duševního vlastnictví.

Cíl práce

Cílem rešeršní části práce je seznámení se se specifiky jazyka JavaScript, metodami jeho obfuskace a také s nástroji k analýze javascriptovských programů, konkrétně s knihovnou Espree a příbuznými. V rámci praktické části práce

potom navrhneme a s použitím zmíněných nástrojů implementujeme účinný statický deobfuskátor javascriptovských programů.

Členění práce

V práci nejprve podrobně rozebereme vybrané obfuskační metody, jejich účel a princip. Poté shrneme nejdůležitější principy z teorie překladačů, které mají využití při implementaci analyzátoru javascriptovského kódu. Dále prodiskutujeme možnosti deobfuskace dříve zmíněných metod. Následně představíme vlastní řešení, které těchto poznatků využívá ke zvýšení čitelnosti kódu. Na závěr prodiskutujeme výsledky nástroje při aplikaci v manuálních testech.

Kapitola 1

Analýza a návrh

1.1 Jazyk JavaScript

Kdykoliv v této práci zmiňujeme jazyk JavaScript, máme na mysli specificky variantu standardizovanou specifikací [1], známou také jako *ECMAScript 5* či *ES5*. Tato sekce se věnuje shrnutí klíčových aspektů jazyka.

JavaScript patří mezi *skriptovací jazyky*, tedy takové, které se užívají k manipulaci existujícího softwarového systému. Svou činnost skript realizuje prostřednictvím rozhraní, které zmíněný systém poskytuje [2].

Rovněž jde o jazyk *interpretovaný*, což znamená, že programy v něm psané na rozdíl od jazyků kompilovaných ke svému spuštění vyžadují *interpret* — program, který zdrojový kód překládá a interpretuje za běhu [3]. JavaScript je distribuován v podobě souborů obsahujících samotný zdrojový kód v prostém textu, oproti např. programům v jazyce Java, jež jsou šířeny prostřednictvím kompilátu zdrojového kódu do interní reprezentace.

Původně jediným zamýšleným využitím jazyka JavaScript bylo skriptování na straně webového prohlížeče. Ten poskytuje skriptu prostředí objektů reprezentujících okna, elementy webovské stránky, události v návaznosti na uživatelskou interakci aj.

Neméně široké využití si však JavaScript našel i na straně serveru, jmenovitě v prostředí node.js. Toto prostředí nabízí například rozhraní pro manipulaci síťových požadavků, souborového systému nebo pro komunikaci s operačním systémem.

Jazyk JavaScript je imperativní, zahrnující všechny základní programovací koncepty známé například z jazyka C (kterému se ostatně stran syntaxe podobá), jakými jsou třeba cykly nebo funkce. Je rovněž objektově zaměřený a poskytuje prostředky k implementaci tradičního objektového paradigmatu obnášejícího třídy, instance a dědičnost. Třídy jako takové vznikají za běhu programu a lze je vytvářet dynamicky díky skutečnosti, že samotná třída je instancí nějaké třídy (podobně, jako třeba v jazyce Smalltalk). Mimo uživatelské definice tříd je součástí specifikace i řada standardních tříd a jejich instancí,

například třída `String` reprezentující řetězec znaků, nebo třída `RegExp` zprostředkující funkcionalitu regulárních výrazů.

Další důležitou vlastností jazyka JavaScript je *dynamické typování*. To znamená, že datový typ není atributem proměnné, nýbrž její hodnoty. Typ hodnoty uložené v určité proměnné tedy není znám v čase překladu, jelikož se za běhu programu může měnit. Stejnou vlastnost vykazuje například interpretovaný skriptovací jazyk Python.

Z konceptů přítomných ve vyšších jazycích pak nalezneme v jazyce JavaScript kupříkladu běhové výjimky nebo některé funkcionální prvky, jako jsou například uzávěry.

1.1.1 Hrozby v jazyce JavaScript

S přihlédnutím k hlavnímu záměru naší práce, kterým je analýza javascriptovského kódu ohrožujícího návštěvníka webovských stránek, je záhodno věnovat podkapitolu hrozbám, které se v prohlížečích interpretujících jazyk JavaScript otevírají.

Johns charakterizuje škodlivý javascriptovský software jako „*útoky zneužívající možnosti webového prohlížeče ke spuštění škodlivého skriptu v rámci lokálního běhového kontextu oběti*“ a dodává, že takové útoky nezneužívají chyb v prohlížeči, nýbrž využívají výhradně jeho legitimní funkcionality [4]. Wang [5] tuto charakteristiku rozšiřuje i o útoky mířené na bezpečnostní chyby prohlížeče nebo některých jeho programových rozšíření, jakými jsou např. Adobe Flash nebo Java.

Jednou z kategorií hrozeb, kterým se Johns věnuje, jsou útoky zaměřené na narušení soukromí oběti. Soukromé informace, ke kterým útočník získává prostřednictvím takových útoků přístup, mohou zahrnovat dříve navštívené stránky, vlastnosti prohlížeče či informace o tom, je-li uživatel přihlášen do určité webové aplikace. Vlastnosti prohlížeče řadíme mezi citlivé údaje zejména proto, že jejich prostřednictvím lze uskutečnit tzv. *browser fingerprinting* a ztotožnit uživatele napříč požadavky, aniž by o sobě poskytl explicitně jakékoliv informace.

Další podstatnou kategorií útoku pojednanou tamtéž je zneužití počítače oběti co by prostředku k vykonání jiného útoku na větší škále, například šíření virů, masové skenování webovských stránek nebo rozesílání nevyžádané pošty. Takové útoky přitom nejsou omezeny na protokol HTTP; mluvíme o tzv. *inter-protocol exploitation*. Ta typicky zneužívá vlastnosti některých dalších protokolů založených na ASCII (např. POP3), která spočívá v tom, že interprety těchto protokolů mlčky ignorují řádky požadavku, jež nelze smysluplně interpretovat. Skript tak může například odeslat vzdálenému serveru HTTP požadavek nesoucí v těle obsah validní v tom kterém protokolu s tím, že v daném koncovém bodě pak dojde k ignoraci hlaviček HTTP a vyhodnocení těla [6]. V jistém ohledu podobné metody lze využít i k provedení XSS [7] (viz dále podkapitolu 1.1.2).

Další klíčovou třídu hrozeb tvoří tzv. *drive-by-download*. Útoky spadající do této kategorie ústí ve stažení libovolného spustitelného souboru přímo do počítače oběti a jeho spuštění, a to bez jakékoliv uživatelské intervence mimo samotnou návštěvu nakažené webovské stránky. Zpravidla bývá pro tyto účely zneužito hrubého bezpečnostního nedostatku komponent ActiveX přítomných ve starších verzích prohlížeče Internet Explorer [8].

Zcela mimo obor našeho zkoumání pak stojí útoky spadající do kategorie tzv. *sociálního inženýrství*, tedy takové, jež stojí na přimění nic netušící oběti k tomu, aby útok v dobré víře sama umožnila. Příkladem může být důvěryhodně vyhlížející přihlašovací formulář pro sběr hesel či falešný hypertextový odkaz, jehož následování vede namísto inzerovaného obsahu ke stažení škodlivého programu.

1.1.2 Vektory útoků

Po nastolení hrubého přehledu o možnostech, které se útočníkům v JavaScriptu otevírají, je na místě prozkoumat i způsoby vpravení škodlivého kódu do webovské stránky.

Většina uvedených útoků se zpravidla vykonává s užitím metod známých souhrnně jako *cross-site scripting* (XSS), jejichž cílem je prostřednictvím legitimních cest vpravit zhoubný javascriptovský kód do těla cizí webovské stránky. Pokud se tento záměr podaří, spuštěný skript pak operuje v běhovém kontextu příslušné stránky a má v důsledku toho přístup ke všem informacím, které prohlížeč skriptu o dané stránce poskytuje, jako jsou např. cookies nebo hodnoty vyplněných formulářových polí. Konkrétní metody XSS jsou velmi rozmanité a jejich podrobnému rozboru se zde vyhneme.

Následuje rozšířená rekapitulace přehledu, který Provos [8] uvádí v kap. 4.

- Získá-li útočník kontrolu nad serverem, na němž je webovská stránka umístěna, dostává se mu možnosti měnit obsah této stránky prezentovaný uživateli. Může toho docílit například modifikací šablon na straně serveru zodpovědných za generování tohoto obsahu. Pozorovány byly například případy vložení elementu `iframe` do částí šablon generujících patičku všech podstránek jisté webovské stránky, takže každá návštěva ústila v pokus o infekci návštěvníka.

- Velké množství stránek umožňuje svým uživatelům přispívat vlastním obsahem, obvykle v podobě komentářů nebo blogu. Pokud je tento uživatelský vstup nesprávně ošetřen vůči výskytu rizikových kusů HTML, může útočník do textu zahrnout škodlivý javascriptovský kód ohrožující návštěvníky stránek zobrazujících jeho příspěvek.

- Umístění reklam na webovskou stránku se zpravidla realizuje vložením obsahu třetí strany do těla vlastní stránky, a to nejčastěji prostřednictvím elementu `iframe` nebo skriptu zodpovědného za načtení reklamy.

Takto vložený obsah je potom zcela ve správě třetí strany, což znamená, že vlastník stránky buď důvěřuje poskytovateli reklamy, že jeho obsah není pro návštěvníky škodlivý, nebo toto bezpečnostní riziko přehlíží. Pokud inzerent navíc zprostředkovává obsah další třetí strany (což je běžná praxe), tento řetězec důvěry se nebezpečně prodlužuje.

- Pro autory legitimních skriptů je na internetu dostupné nepřeberné množství javascriptovských knihoven, které programátorovi usnadňují vývoj aplikace. Ten se však již obvykle nezabývá podrobnou implementací knihovních funkcí; to skýtá útočníkům další potenciální vektor pro šíření škodlivého kódu. Hypotetický útočník může nabízet knihovnu zdánlivě realizující nějakou užitečnou funkcionalitu, která však mimo to obsahuje i škodlivý kód.

- Známé jsou i případy tzv. *typosquattingu*. V současnosti je v provozu mohutný repozitář javascriptovských knihoven npm [9], který uživatelům mimo jiné nabízí možnost snadného stahování knihoven prostřednictvím příkazové řádky. Typosquatting spočívá v tom, že útočník zveřejní škodlivou knihovnu s názvem velmi podobným jiné, takže programátoři, kteří udělají v zápisu jména požadované knihovny chybu omylem zahrnou do svého kódu knihovnu škodlivou. V roce 2017 bylo odhaleno a odstraněno asi 40 škodlivých balíčků z npm, které tohoto principu využívaly [10].

- Někteří uživatelé internetu využívají služeb tzv. *proxy serverů*, což jsou stroje určené k oboustrannému přeposílání HTTP požadavků, sloužící tak jako jacísi prostředníci komunikace klienta se serverem. Nejčastější motivací uživatelů k použití takového řešení je anonymizace — server, na který je požadavek směřován, neuskutečňuje přímou komunikaci s uživatelovým počítačem, ale pouze se zmíněným proxy serverem. Obecně není doporučeno takový server používat, pokud mu uživatel nemůže bezmezně důvěřovat, protože tím proxy serveru umožňuje sledovat veškerou svoji aktivitu na webu. Pro nás je ale jejich využití zajímavé z hlediska, které popisuje Alonso v [11]. Ten užívá proxy serveru jako vektoru útoku, při kterém se přeposílané javascriptovské soubory za běhu modifikují vložením vlastního škodlivého kódu.

1.2 Obfuskace

Jak bylo řečeno výše, prohlížeč ze serveru stahuje javascriptovské soubory v podobě prostého textu se zdrojovým kódem. Jde-li o kód napsaný ručně a nikoliv strojově generovaný (kteroužto variantu vzápětí rozebereme podrobněji), stačí zkušenému programátorovi zpravidla pouhé jeho přečtení k odhalení přítomnosti nekalého úmyslu. Skripty manifestující některý známý typ škodlivého chování navíc často sdílí určité textové vzorce typické pro daný

```
var fs = new ActiveXObject('Scripting.FileSystemObject');
if (typeof fs.GetDrive('C').FreeSpace == 'number') {
  var stream = new ActiveXObject('ADODB.Stream');
  var xhr = new ActiveXObject('MSXML2.XMLHTTP');
  xhr.open('GET', 'http://lollyonn.info/search.php', 0);
  stream.Open();
  var tmpPath = fs.GetSpecialFolder(2)
    + '\\\\' + fs.GetTempName();
  stream.Type = 1;
  xhr.send();
  var shell = new ActiveXObject('WScript.Shell');
  var cmd = 'cmd.exe /c ' + tmpPath;
  stream.Position = 0;
  if (xhr.Status == 200) {
    stream.Write(xhr.ResponseBody);
    stream.SaveToFile(tmpPath);
    stream.Close();
    shell.run(cmd, 0);
  }
}
```

Ukázka kódu 1: útok typu drive-by-download manifestující všechny charakteristické vzorce, zejména instanciaci konkrétních objektů ActiveX

útok (viz ukázku 1) a jsou tak náchylné k odhalení pomocí detekce založené na signatuře souboru (tedy na hledání známých vzorců v souboru), kterou používají antivirové programy k odhalování škodlivého software.

Nejrozšířenějším prostředkem k obcházení detekce škodlivého javascriptovského kódu je jeho obfuskace [12]. *Obfuskátor* je kompilátor jazyka sloužící ke snížení srozumitelnosti kódu (v případě překladu v rámci téhož jazyka), respektive znesnadnění jeho zpětného inženýrství (v případě jazyka kompilovaného). Po zbytek práce se budeme držet definice obfuskační transformace tak, jak ji zavádí Collberg [13]: řekneme, že transformace zdrojového programu na cílový program $\tau : P \to P'$ je *obfuskační transformací*, pokud platí, že

- pokud se P nezastaví, P' se může a nemusí zastavit,
- pokud se P zastaví, P' se zastaví a vypočte stejný výsledek, jako P [13].

Stejným výsledkem je přitom myšlen výsledek zakoušený uživatelem. Podobně lze o P' říci, že má totožné pozorovatelné chování, jako P. Všimněme si, že tato podmínka obecně neklade na P' požadavek, aby nevykazoval vedlejší účinky u P nepřítomné.

Dále budeme při hovoru o obfuskační transformaci předpokládat, že její aplikace vede ke snížení srozumitelnosti či též zvýšení komplexity programu.

Tuto vlastnost Collberg formalizuje poněkud vágně a my se zde spokojíme s její intuitivní interpretací.

Připomeňme, že dalším velmi běžným a tentokrát zcela legitimním případem užití obfuskace kódu je chránění duševního vlastnictví. Výše zmíněné vlastnosti JavaScriptu mající vliv na jednoduchost analýzy cizího programu zároveň umožňují poměrně snadnou krádež jeho klíčových částí. Poskytovatelé komerčního software proto přikračují k obfuskaci jakožto pojistce proti krádeži takových klíčových částí případným tržním soupeřem a potažmo jako ke způsobu zvýšení své konkurenceschopnosti.

1.3 Obfuskační techniky

V této kapitole revidujeme některé konkrétní obfuskační techniky s relevancí k jazyku JavaScript. Poznamenejme, že je obvyklé tyto metody používat v součinnosti, čímž se účinnost obfuskace významně zvyšuje.

Collberg [13] zavádí dělení obfuskačních metod do několika tříd:

- transformace zápisu, nemající vliv na sémantiku kódu,
- transformace výpočtu, tedy modifikace logiky programu, a
- transformace dat, zastřešující techniky tkvící v zakódování konstantních dat v kódu programu a jejich dekódování za jeho běhu

Ještě podotkněme, že v jiných jazycích se otevírá řada možností obfuskace, které v JavaScriptu uplatnit nelze, a proto je zde neuvedeme. Zejména vynecháme ty, které jsou použitelné výhradně u kompilovaných jazyků (jako je převod cílového kódu do podoby bez ekvivalentního vyjádření v kódu zdrojovém), a ty, jejich použití v jazyce JavaScript nemá valného smyslu (jako je modifikace dědičnosti tříd například v jazyce Java).

1.3.1 Obfuskace řetězců

Řetězce mají v JavaScriptu obecně široké využití a pro obfuskaci to platí dvojnásobně, což z obfuskace řetězců dělá typického zástupce transformace dat v JavaScriptu. Za pozornost stojí především vestavěné funkce jazyka umožňující vyhodnotit řetězec jako javascriptovský program, mezi něž patří

- globální metoda `eval`, která jednoduše bere za argument řetězec k bezprostřednímu vyhodnocení,
- konstruktor `Function`, jež vytváří novou funkci s tělem daným řetězcem předaným za argument,
- globální metody `setTimeout` a `setInterval`, přijímající za argument kód ke spuštění po určité prodlevě mj. v podobě řetězce, nebo

- `document.write`, metoda přítomná v prostředí prohlížeče umožňující vystoupit libovolný řetězec do těla dokumentu; je-li ve vystoupeném textu přítomný HTML tag `script`, dojde k jeho vyhodnocení.

Tím se obfuskátorovi otevírá možnost užít velmi silnou metodu obfuskace celého programu popsanou následujícím schématem

1. zašifrovat kód ručně napsaného programu,
2. vytvořit javascriptovskou funkci, která použitou šifru dešifruje,
3. distribuovat kód v podobě

```
var enc = "zašifrovaný kód";
var dec = dešifruj(enc);
eval(dec);
```

Výsledný kód zjevně neobsahuje žádný kus původního zdrojového textu, ale funkčně je zcela totožný. Použitá šifra při tom nemusí být nikterak složitá. Howard [14] mezi nejpoužívanějšími technikami zmiňuje ty uvedené v tabulce 1.1 a další, pro které v JavaScriptu existuje přímočarý způsob dešifrování, ale k úspěšné obfuskaci obvykle bohatě postačí.

Tabulka 1.1: obvyklé metody šifrování řetězců

metoda	zašifrovaný text „Ahoj, světe“
%-kódování	`%41%68%6f%6a%2c%20%73%76%11b%74%65`
kódování unicode entitami	`\u0041\u0068\u006f\u006a\u002c…`
obrácení	`etěvs ,johA`
záměna podřetězců	`AhZZZZvěte`
base64	`S2RvIHRvIGN0ZSBqZSB2dWw=`

Rovněž se často objevuje metoda tvorby řetězců spojováním dílčích a to zejména k docílení jednoho ze dvou efektů: obfuskace řetězce prostřednictvím obfuskace pouhé jeho části a posílení potence obfuskace užitím různých metod na jednotlivé podřetězce.

Zašifrování však nemusí podléhat celý zdrojový text programu. Dalším pro nás významným užitím řetězců je přístup k vlastnostem objektů. K vlastnosti objektu `foo` nazvané „bar“ lze v JavaScriptu totiž přistoupit výrazem

```
foo.bar
```

ale také

```
foo["bar"]
```

a ovšem i obfuskovaně, například

```
foo["rab".split("").reverse().join("")]
```

1.3.2 Přidání mrtvého kódu

Pojmem *mrtvý kód* označujeme část kódu programu, k jejímuž vyhodnocení za běhu nemůže dojít. Za příklad si vezměme tělo podmíněného příkazu, jehož podmínka je z principu nesplnitelná, nebo kód následující po příkazu `return`.

Cílem vkládání takového kódu je zmatení subjektu, který program analyzuje. Zmíněný případ s podmíněným příkazem je pro obfuskaci obzvlášť účinný, je-li zkombinován s některou antiemulační technikou, např. s použitím nějakého časového údaje v podmínce příkazu, jak rozebírá Howard [14].

1.3.3 Přejmenování proměnných

Názvy proměnných jsou programátorem obvykle voleny tak, aby usnadňovaly srozumitelnost a popisovaly jejich obsahy, v důsledku čehož samy nesou obsáhlou informaci o programu jako takovém [13]. Je proto nabíledni, že náhrada jmen proměnných v kódu na náhodné či irelevantní názvy vede k nezanedbatelné obfuskaci. Z hlediska interpretru je transformovaný kód sémanticky ekvivalentní původnímu, přejmenování proměnných je tedy pouze transformací zápisu.

1.3.4 Změna formátování

Bílé znaky v jazyce JavaScript nenesou význam a proto se užívají k přehlednému formátování kódu, např. aby každý příkaz byl na vlastní řádce a těla složených příkazů odsazena. Obvyklou transformací zápisu je tyto formátovací bílé znaky zcela odstranit nebo jich naopak na náhodná místa nesmyslná množství vložit.

1.3.5 Inlining a extrakce funkcí

Funkce představuje část programu, jejíž funkcionalitu lze zpravidla analyzovat nezávisle na zbytku programu, což rozbor nezřídkakdy podstatně zjednodušuje. Inlining je označení pro transformaci výpočtu, která obnáší nahrazení výskytu volání funkce v kódu přímo jejím tělem.

Objevuje se ale i opačná transformace, tedy extrakce nějakého výrazu do vlastní funkce. Tento postup si klade za cíl znemožnit či znesnadnit analýzu výrazu v kontextu jeho použití.

1.3.6 Záměna řídicích příkazů

Řídicí příkazy v JavaScriptu jako jsou `if`, `switch` a `for` lze v jistých případech vzájemně zaměňovat a mást tak analyzující subjekt. Například podmínku na rovnost, obyčejně implementovanou pomocí příkazu `if`, lze ekvivalentně implementovat s pomocí příkazu `switch`, jak demonstruje ukázka 2. Podobně lze nahradit blok příkazů za cyklus, který proběhne právě jednou (ukázka 3).

```
// pomocí if
if(a === 1) b = "jedna";
else b = "více";

// ekvivalentně pomocí switch
switch(a) {
  case 1: b = "jedna"; break;
  default: b = "více"; break;
}
```

Ukázka kódu 2: Záměna příkazu `if` za `switch`

```
while(true) {
  škoď();
  break;
}
```

Ukázka kódu 3: Záměna sekvence příkazů za cyklus

Tato technika nabývá na účinnosti o to více, podmíní-li se ukončující příkaz `break` nějakým neemulovatelným inherentně pravdivým výrazem.

1.3.7 Závislost na prostředí

Početné obfuskační metody spoléhají na prostředí prohlížeče, z nějž čerpají hodnoty k užití ve výpočtu [14]. Obfuskátor Hieroglyphy [15] produkuje program, který skládá spustitelný kód po jednotlivých znacích; například znak „p“ získává pomocí výrazu `([] + location)[3]`, tedy spoléhá na to, že existuje globální objekt `location` představující URL s protokolem HTTP či HTTPS, z nějž čtvrtý znak extrahuje.

Obfuskační potenciál této metody je založen na tom, že je nesnadné vybavit statický nástroj k vyhodnocení všech proměnných spouštěcího prostředí.

1.4 Dosavadní řešení obfuskace jazyka JavaScript

Ve světě JavaScriptu převládá zájem o minifikaci nad obfuskací a odpovídá tomu počet minifikačních nástrojů. Jako příklady uveďme UglifyJS2, Google Closure Compiler nebo YUI compressor. Termín minifikace přitom neodkazuje na konkrétní obfuskační techniku, ale označuje užití některých transformací kódu ke snížení jeho délky a potažmo zrychlení jeho stahování ze serveru. Velmi častými složkami minifikace jsou přejmenování proměnných na kratší jména a náhrada výrazů za jejich kratší ekvivalenty (např. `true` za `!0`). V rámci minifikace se ovšem užívá i transformací jdoucích proti obfuskaci, např. eliminace mrtvého kódu a zjednodušování složitých výrazů. I minifikace

ústí v určitou míru obfuskace, zejména kvůli zmíněnému přejmenování proměnných a odstranění formátování.

Rozšířeným nástrojem určeným výslovně k obfuskaci JavaScriptu je Javascript Obfuscator [16]. Jeho hlavní metodou je přenesení všech řetězců v kódu do globálního pole a následná náhrada jejich původních výskytů za reference do tohoto pole.

Dalšími, poněkud silnějšími, online nástroji jsou volně dostupný JavaScript Obfuscator [17] či komerční Jscrambler [18], které užívají kombinaci více metod.

Zvláštním případem obfuskačně účinného minifikátoru je packer [19]. Ten vychází ze skutečnosti, že kód programu obsahuje obvykle hodně opakujících se slov. Packer tato soustředí do jednoho řetězce, v původním kódu nahradí výskyty slov odkazy do něj a vystoupí krátkou funkci, která z těchto dvou řetězců vytvoří původní kód a nechá jej vyhodnotit.

1.5 Statická analýza

Termínem statická analýza označujeme programatický lexikálně sémantický rozbor vstupního kódu. Pojem zastřešuje řadu konkrétních metod a konceptů vycházejících z teorie překladačů, z nich stěžejní v této podkapitole rozebereme.

1.5.1 Parsing

Uskutečnění jakéhokoliv automatizovaného rozboru programu stojí na převedení kódu zkoumaného programu do vnitřní datové reprezentace, se kterou lze v rámci takového rozboru operovat. To je úkolem takzvaného *parseru*. Šíře toho, co parsováním nazýváme, je velká: parsing může mít na starost převod řetězce držícího dekadický zápis celého čísla na strojovou reprezentaci i sestrojení abstraktního syntaktického stromu počítačového programu z jeho kódu.

Ve stručnosti si představíme obvyklou realizaci posledního zmíněného.

Na vstupu parseru stojí lexikální analyzátor, tzv. *lexer*. Jeho vstupem je kód programu v podobě řetězce, výstupem pak proud lexikálních entit daného jazyka. Nejpřímočařejším způsobem implementace je užití konečného stavového automatu, kterému je vstupní řetězec posloupností přechodů a jenž v určitých stavech emituje patřičnou entitu. Kupříkladu takovýto výsledek může mít lexikální analýza úryvku javascriptovského programu:

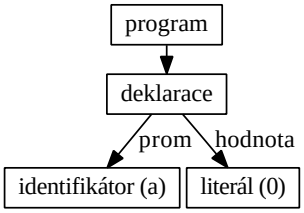
```
var a = 0;
```

klíčové slovo `var` | identifikátor `a` | operátor `=` | literál `0` | konec příkazu

Nad tímto výstupem potom pracuje *syntaktický analyzátor*. Pro něj je opět typická realizace čerpající z konceptu konečného stavového automatu, pro nějž

je tentokrát sledem přechodů právě proud lexikálních entit. Podoba výstupu se různí v závislosti na aplikaci; my budeme pracovat s velmi častou reprezentací pomocí *abstraktního syntaktického stromu* (AST). Jeho uzly představují syntaktické celky daného jazyka zatímco hrany značí jejich náležitost do nadstruktur. Uvedeme si příklad AST sestrojeného pro tentýž úryvek kódu.

```
var a = 0;
```



Reprezentace pomocí AST je pro další analýzu výhodná právě pro svou stromovou strukturu, nad níž lze snadno pracovat pomocí různých rekurzivních algoritmů.

1.5.2 Analýza referencí

V JavaScriptu a většině ostatních jazyků je na data, s nimiž program pracuje, odkazováno prostřednictvím pojmenovaných proměnných. Jazyk specifikuje pravidla pro určení proměnné, na kterou odkazuje konkrétní výskyt pojmenování v kódu programu. Problém ilustrujme na příkladu 4. Povšimněme si především, že jméno „a“ odkazuje v různých kontextech použití na různé proměnné.

```
1  var a = 0;
2  var b = 0;
3
4  function fun() {
5      var a = 0;
6      a++;            // inkrement proměnné deklarované na ř. 5
7      b++;            // inkrement proměnné deklarované na ř. 2
8  }
9
10 a++;                // inkrement proměnné deklarované na ř. 1
```

Ukázka kódu 4: příklad referencí v jazyce JavaScript

Reprezentace konkrétních proměnných a identifikace referencí na ně je zodpovědností *analýzy referencí*.

V některých jazycích ovšem nastávají situace, kdy nelze odkazovanou proměnnou určit staticky, a JavaScript toho není výjimkou. Příkladem nám budiž javascriptovský příkaz `with`, kterým lze do referenčního kontextu dočasně

zahrnout vlastnosti nějakého objektu. Jelikož lze tyto vlastnosti měnit dynamicky (a to i co do jejich přítomnosti v příslušném objektu), lze jedině za běhu určit, na který obsah jisté jméno proměnné v kontextu příkazu `with` odkazuje.

1.5.3 Analýza řídicího toku

K zachycení větvení řídicího toku v programu a podmíněných skoků obecně se používá struktura zvaná *graf řídicího toku* (control-flow graph, CFG). CFG je orientovaný graf nad atomickými příkazy v programu; hrana (s_1, s_2) je v grafu přítomná právě tehdy, může-li vykonání příkazu s_2 při běhu programu následovat těsně po vykonání s_1. Posloupnost vykonaných příkazů za běhu programu je potom vždy nějakým orientovaným sledem v tomto grafu.

Upřesněme, že formalizmus, který zde uvádíme, je aplikovatelný na jazyk symbolických instrukcí. V podkapitole 2.1.2 následně ukážeme, jak jej adaptujeme pro použití v jazyce JavaScript. Jazyk symbolických instrukcí reprezentuje program jako posloupnost příkazů, které se vykonávají v pořadí této posloupnosti, nebo, v případě příkazů uskutečňujících skok, v pořadí určeném cílem tohoto skoku.

Tradiční implementace reprezentují CFG s pomocí tzv. *základních bloků*. Základní blok je fragment programu určený následujícími pravidly:

- příkaz je prvním příkazem bloku, jestliže
  - je prvním příkazem nějakého podprogramu,
  - je cílem nějakého skoku, nebo
  - následuje za nějakým příkazem skoku;
- pro každý první příkaz bloku tento blok obsahuje všechny následující příkazy až po první příkaz jiného bloku (vyjma) [20].

Hrany v této reprezentaci potom značí možný přechod řízení z jednoho bloku do druhého.

1.5.4 Analýza toku dat

V momentě, kdy každému bodu v programu dovedeme přiřadit množinu sledů příkazů, jež mu mohou při běhu předcházet a následovat, jsme schopni leccos soudit o přenosech dat napříč částmi programu. Možnosti toho, co je možné v rámci *analýzy toku dat* (data-flow analysis, DFA) zjišťovat, jsou rozmanité, nicméně všechny sem spadající rozbory se obvykle implementují podle jednotného schématu. Zaveďme si pro něj nejprve následující definice:

$Succ_G(n) = \{m : (n, m) \in E(G)\}$ je množina následníků uzlu n v grafu G.

$Pred_G(n) = \{m : (m, n) \in E(G)\}$ je množina předchůdců uzlu n v grafu G.

Algoritmus 1 Obecný algoritmus dopředné analýzy toku dat

Vstup graf řídicího toku CFG
Výstup mapování bloků na množiny: *in* a *out*

```
1:  for blok ∈ N(CFG) do
2:      in(blok) ← ∅
3:      out(blok) ← ∅
4:  end for
5:  worklist ← N(CFG)
6:  while ||worklist|| > 0 do
7:      blok ← Pop(worklist)
8:      in(blok) ← ⋃_{pred∈Pred_CFG(blok)} out(pred)
9:      out(blok) ← SpočtiVýstupníMnožinu(blok)
10:     if došlo ke změně v out(blok) then
11:         worklist ← worklist ∪ Succ_CFG(blok)
12:     end if
13: end while
```

Samotné schéma je popsáno algoritmem 1. Jeho výstupem je mapování základních bloků grafu řídicího toku na vstupní a výstupní množiny (nehledě na to, co tyto množiny v konkrétní analýze představují). Dosahuje toho algoritmem na principu *fixed-point iteration*, v rámci nějž dochází k opakovanému výpočtu daných množin až do chvíle, kdy se výsledky poslední iterace přestanou různit od dosavadních.

Uvedený algoritmus realizuje specificky dopřednou analýzu toku dat; algoritmus zpětné analýzy je totožný až na vzájemnou záměnu $Pred$ a $Succ$ na řádcích 8, resp. 11.

Pro nás nejvýznamnější analýzou toku dat je *analýza dosažitelných definic* proměnných (reaching definition analysis), která umožňuje pro každý výskyt identifikátoru v kódu určit všechna místa, která definují obsah příslušné proměnné v místě zmíněného výskytu. Množiny $in(blok)$, resp. $out(blok)$, z algoritmu 1 zde představují množiny definic platných na začátku, resp. na konci, bloku. Pro definici funkce SpočtiVýstupníMnožinu figurující v tomto algoritmu nejprve zaveďme dvě pomocné množiny.

$gen(blok)$ představuje množinu definic vytvořených blokem. Pro každou proměnnou definovanou v bloku obsahuje její poslední definici.

$kill(blok)$ drží všechny definice, které jsou v rámci bloku zneplatněny, což jsou definice z počátku bloku takových proměnných, jejichž definice je přítomná v $gen(blok)$.

Samotnou funkci SpočtiVýstupníMnožinu potom definujeme podle [20] následovně:

$$\text{SpočtiVýstupníMnožinu}(blok) = gen(blok) \cup (in(block) \setminus kill(blok)).$$

1.6 Deobfuskace jazyka JavaScript a její dosavadní řešení

Tématem této práce je hledání způsobů, jak zvrátit některé obfuskační transformace a přidat obfuskovanému kódu na srozumitelnosti, což je obecně řádově náročnější úloha, než dosud probírané znesrozumitelnění. Lze říci, že situace obfuskace vůči deobfuskaci není nepodobná příslovečné dualitě vytlačování zubní pasty z tuby a snahy dostat pastu do tuby zpět. Collberg tuto vlastnost charakterizuje přirovnáním k asymetrickému kryptosystému, který stojí na dramatickém rozdílu nároků na zašifrování textu a jeho rozšifrování [13].

Blanc nicméně konstatuje, že samotné spuštění programu představuje jistou formu deobfuskace [21]. Na této ideji je postavena deobfuskace prostřednictvím *dynamické analýzy*, protějška analýzy statické.

Dynamická analýza se realizuje spuštěním zkoumaného programu v nějakém izolovaném prostředí schopném podávat zprávy o činnosti programu a zároveň neumožňujícím mu uskutečnit případný útok. Je to velmi univerzální metoda vzhledem k tomu, že podává přesnou informaci o činnosti programu, je nicméně také velmi náročná na přípravu takového bezpečně izolovaného prostředí i co do výpočetní náročnosti samotné analýzy.

Při analýze programů v jazyce JavaScript se pro jeho dynamickou povahu často přikračuje právě k analýze dynamické [22].

Množství literatury se soustředí na detekci škodlivého javascriptovského kódu, ve které hraje deobfuskace větší či menší roli v závislosti na zvolené metodě detekce. Provos [8] a Rieck [23] při identifikaci škodlivého kódu například obfuskaci nezohledňují vůbec; namísto toho spoléhají na dynamickou analýzu, při níž spouští kód v izolovaném prostředí a mapují jeho činnost.

Mimo nástrojů detekujících škodlivý JavaScript existují i prostředky k určení obfuskovanosti kódu. AL-Taharwa [12] implementuje statický nástroj na detekci obfuskovaného kódu, který extrahuje vlastnosti abstraktního syntaktického stromu a následně klasifikuje program pomocí metod strojového učení. Velmi podobné metody detekce používá i Kaplan [24], který navíc adresuje nesprávnost záměny obfuskovaného kódu s kódem škodlivým. Xu [25] pro detekci obfuskace mapuje výskyt volání funkcí typických pro obfuskovaný kód. Choi [26] se věnuje detekci obfuskovaných řetězců, které jsou obvyklým prostředkem obfuskace, a to prostřednictvím statické analýzy hodnot proměnných a následnou aplikací statistických metod na nalezené řetězce.

Značně nižší množství výzkumů se věnuje samotné deobfuskaci. Blanc v [27] a [21] popisuje využití rovnostní logiky a automatických dokazovacích

systémů k přepisu kódu do normalizované, sémanticky ekvivalentní podoby. Lu [22] nechává program spustit v upraveném prohlížeči, načež s pomocí dat získaných přímo z interpretru analyzuje relevantní části programu a dekompiluje je do srozumitelného kódu.

Zvláštní kategorii deobfuskátorů tvoří nástroje na zvrácení obfuskace jmen proměnných rozebrané v [28] a [29], které využívají metod strojového učení nad korpusy neobfuskovaného kódu.

Na několik jednodušších obfuskačních vzorců cílí webový nástroj Javascript Unobfuscator [30], který například umí rozkódovat řetězec v hexadecimální notaci nebo vyhodnotit jednoduché výrazy.

1.7 Aspekty deobfuskace

V této podkapitole ukážeme, jaké výzvy přestavují některé obfuskační techniky z hlediska deobfuskace a jaká řešení se při jejich zvracení nabízejí.

1.7.1 Formátování

Ač je odstranění formátování (popsané v podkapitole 1.3.4) velkou překážkou čitelnosti z hlediska lidského čtenáře, je opětovné zformátování kódu snadné. Postačující je při výstupu kódu z AST předřadit každému příkazu patřičný počet odsazovacích znaků podle hloubky zanoření a následovat jej řádkovým zlomem.

1.7.2 Deobfuskace dat

Jak bylo řečeno v podkapitole 1.3.1 o obfuskaci řetězců, tkví obfuskace dat obecně v náhradě konstantních dat v kódu programu za nějaký netriviální výpočet, jehož výsledkem jsou původní data. Cestou k deobfuskaci je v tomto případě emulace daného výpočtu. Z hlediska obfuskátora zde přichází ke slovu antiemulační techniky, jež byly zmíněny v podkapitole 1.3.2.

1.7.3 Deobfuskace řídicího toku

Mnohé obfuskační transformace je možné zvrátit s pomocí analýzy řídicího toku (viz podkapitolu 1.5.3).

Záměnu řídicích příkazů (viz podkapitolu 1.3.6) je v jistých instancích možné eliminovat nalezením nesouhlasu mezi použitým příkazem a strukturou hran, kterou vnáší do CFG. Například příkazy iterace obvykle vytváří v CFG orientovanou kružnici; ne-li, může jít o případ, kdy lze příkaz ekvivalentně nahradit jeho tělem.

Podobně mrtvý kód lze charakterizovat tím, že ve smyslu CFG není dosažitelný ze vstupního uzlu.

1.7.4 Deobfuskace jmen proměnných

Záměna jmen proměnných v kódu představuje ztrátovou a tedy víceméně nevratnou transformaci. V některých případech lze však srozumitelné jméno pro proměnnou vyvodit z kontextu jejího použití. Například identifikuje-li deobfuskátor jistou proměnnou jako řídicí proměnnou iterace, může jí přejmenovat na „i“, což je konvenční název pro takovou proměnnou.

Další možností je přiřazovat proměnným názvy podle výrazů do nich přiřazovaných. Je například vhodné proměnnou inicializovanou na výsledek volání funkce `getMinimum` pojmenovat „minimum“.

U deobfuskace jmen je dobré umět obfuskované názvy detekovat a nahrazovat je jen v případě kladného nálezu. Například v javascriptovském kódu

```
var záškodník = new ActiveXObject(cosi);
```

není vhodné proměnnou `záškodník` přejmenovat na „activeXObject“.

Kapitola 2

Realizace

Nyní se dostáváme k popisu námi implementovaného deobfuskátoru javascriptovského kódu. Za jazyk implementace jsme si zvolili samotný JavaScript, protože v něm existuje řada hotových řešení pro analýzu javascriptovských programů. Nástroj je spustitelný v prostředí node.js.

Složky implementovaného deobfuskátoru lze rozdělit do dvou fází: v rámci první analyzujeme předložený kód, druhá potom slouží k provedení konkrétních transformací na základě výstupů analytické fáze.

2.1 Analýza vstupu

Tato podkapitola se věnuje popisu složek našeho deobfuskátoru zodpovědných za rozbor vstupního obfuskovaného kódu.

2.1.1 Sestrojení AST a analýza referencí

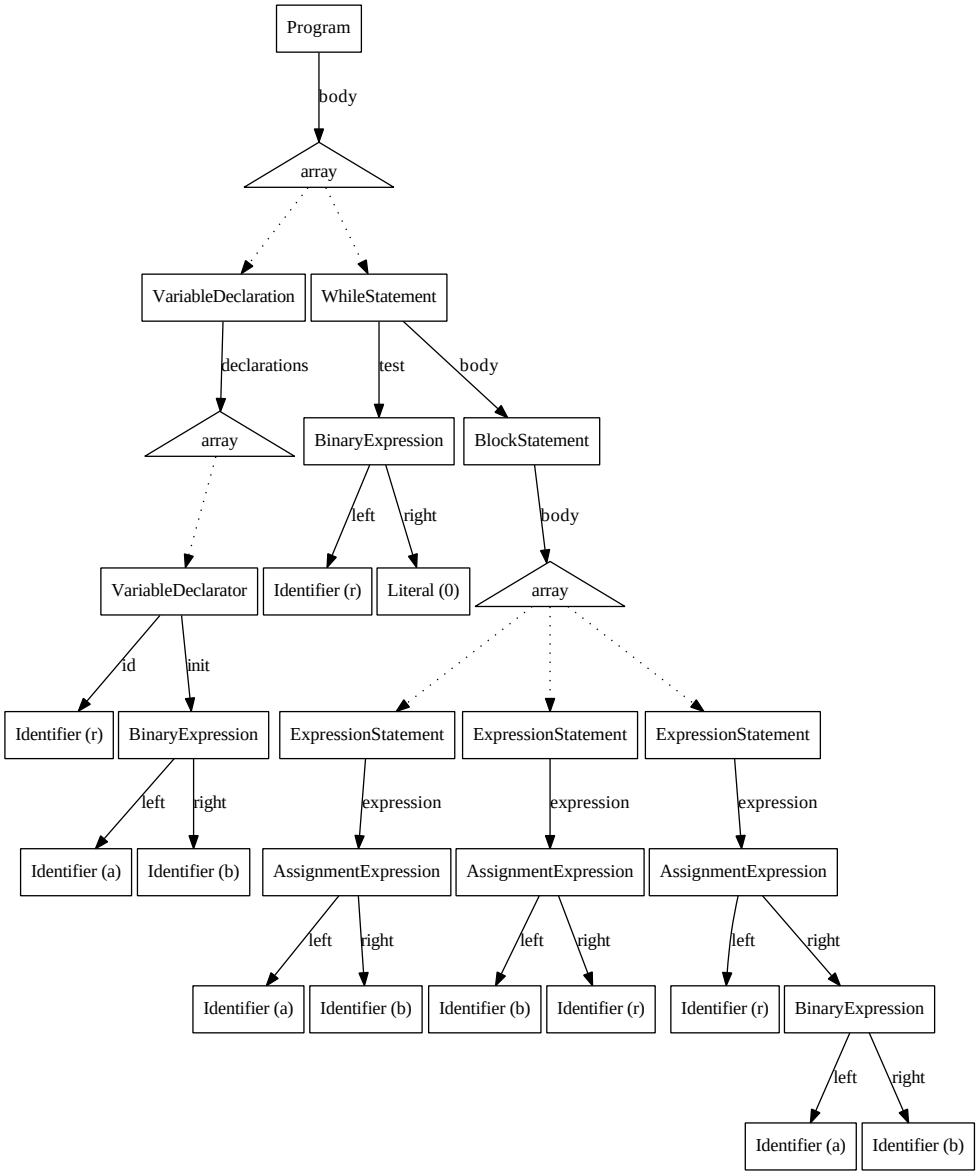
Pro reprezentaci programu jsme si zvolili podobu AST (viz podkapitolu 1.5.1), konkrétně formát ESTree [31], který je *de-facto* standardem pro zpracování JavaScriptu. Sestrojení AST ze vstupního kódu neimplementujeme sami, nýbrž zodpovědnost za něj přenecháváme knihovně Espree [32]. Reprezentace krátkého programu v tomto formátu je zachycena na obrázku 2.1.

Další knihovnu, Escope [33], používáme k analýze referencí (viz podkapitolu 1.5.2). Ta pracuje nad ESTree stromem, jehož uzly reprezentující deklarace proměnných a jejich použití uvádí do vzájemných souvislostí.

Knihovna Escope zavádí distinkci referenčních kontextů na statické a dynamické podle toho, je-li v rámci kontextu možné na základě jména určit odkazovanou proměnnou staticky, nebo až za běhu. Příkladem dynamického kontextu je globální kontext nebo kontext funkce, v níž je volána funkce `eval`. Reference z dynamického kontextu tedy Escope zcela správně neztotožňuje s žádnou proměnnou, což nám ovšem znemožňuje ji použít například k nalezení proměnné `a` odkazované v ukázce 5 na ř. 4. Tento problém obcházíme

Obrázek 2.1: Ukázka AST pro krátký program

```
var r = a % b;
while(r !== 0) {
  a = b;
  b = r;
  r = a % b;
}
```



```
1 function konej() {
2   eval("/* zde se nic neděje */");
3   a = "kamufláž";
4   alert(a);
5 }
```

Ukázka kódu 5: dynamický kontext funkce volající `eval`

```
1 try {
2   throw "kamufláž"
3 } catch(e) {
4   alert(e);
5 }
```

Ukázka kódu 6: vyvolání výjimky

zavedením *pseudoproměnných*, které reprezentují proměnnou jistého jména odkazovanou z dynamického kontextu.

Další pseudoproměnnou používáme k reprezentaci hodnoty vyvolané výjimky. Díky tomu jsme s to určit proměnnou `e` na ř. 4 v ukázce 6.

Aby se s abstrakcí nad proměnnými i pseudoproměnnými v kódu pracovalo konzistentně, implementujeme pro jejich správu třídu `Scoper`, která je tenkým obalem kolem správce referencí knihovny Escope.

2.1.2 Sestrojení CFG

Graf řídicího toku nám zejména poskytuje základ pro provedení dalších analýz z kategorie DFA. Otázkou při jeho implementaci je, jak podrobnou reprezentaci vykonávání jednotlivých javascriptovských příkazů zvolit.

Jednou z možností je zanášet do grafu pouze příkazy, jako je tomu ve WALA [34]. Graf řídicího toku zkonstruovaný nad ESTree uzly reprezentující příkazy ilustruje ukázka 2.2.

Tato úroveň granularity nám však nedostačuje. Uvažme například následující příkaz

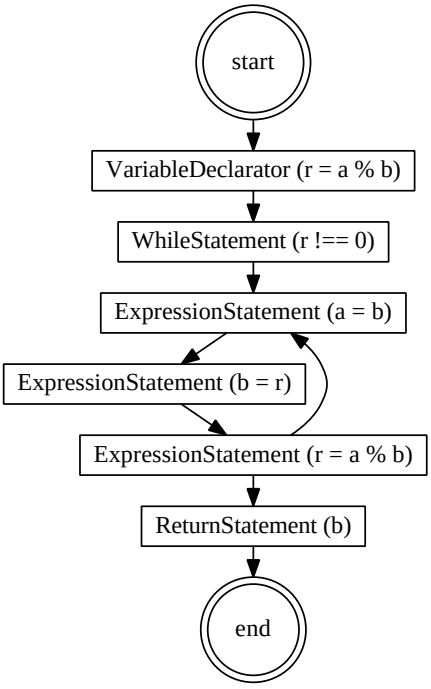
```
a = b && fn(b),
```

kde vzhledem k tomu, že operátor `&&` podléhá tzv. *částečnému vyhodnocování*, k volání `f(n)` nedojde, drží-li `b` nepravdivou hodnotu. Toto větvení řídicího toku však v CFG sestaveném uvedeným způsobem není postiženo.

Jinou variantou je konstruovat CFG nad jednotlivými abstraktními operacemi, pomocí nichž jsou ve specifikaci [1] definována vyhodnocení jednotlivých syntaktických prvků. Tato reprezentace se pro naše potřeby ukázala být příliš výřečná, jak je vidět v ukázce 2.3, kde navíc není postižena situace, kdy dané vyhodnocení vyvolá výjimku.

Obrázek 2.2: CFG sestrojený nad příkazy

```
var r = a % b;
while(r !== 0) {
  a = b;
  b = r;
  r = a % b;
}
return b;
```



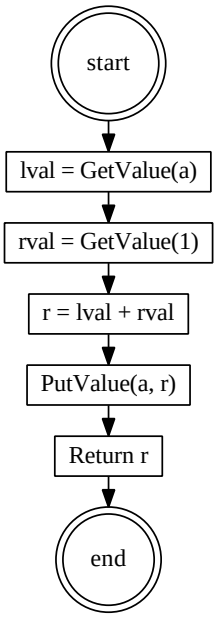
Pojetí CFG, ke kterému se v rámci této práce nakonec přikláníme, je jistým kompromisem mezi dvěma dosud uvedenými možnostmi. Konstruujeme jej ze struktury, kterou si nazvěme *uzlový CFG*.

Za prvky uzlového CFG CFG_u volíme ESTree uzly, nikoliv však pouze ty reprezentující příkazy, nýbrž všechny. Navíc přidáváme speciální druhy uzlů reprezentující vstupní bod podprogramu, normální výstupní bod programu a výstup z podprogramu při nezachycené výjimce; pro podprogram f je značíme po řade $enter_u(f)$, $return_u(f)$ a $throw_u(f)$. Hrana $(n_1, n_2) \in E(CFG_u)$ je potom zavedena, pokud

- n_1 je $enter_u(f)$ a n_1 je prvním příkazem v těle f,
- n_2 je $return_u(f)$ a vyhodnocení n_1 v rámci těla f nic nenásleduje,
- n_2 je $throw_u(f)$ a n_1 může vyvolat výjimku,
- n_2 je potomek n_1 ve smyslu AST a vyhodnocení n_2 je prvním při vyhodnocování n_1,
- n_1 a n_2 mají v AST společného rodiče *stmt*, n_1 nemusí vyvolat výjimku a vyhodnocení n_2 následuje přímo po vyhodnocení n_1 při vyhodnocování *stmt*, nebo

Obrázek 2.3: CFG sestrojený nad abstraktními operacemi v jazyce JavaScript

```
a += 1
```



- n_1 nemusí vyvolat výjimku, je v rámci vyhodnocování svého rodiče *stmt* vyhodnocován jako poslední a vyhodnocení n_2 následuje vyhodnocení *stmt*.

Ve finální implementaci dodáváme do uvedeného schématu několik drobných výjimek vedoucích k usnadnění práce s CFG. Ty jsou dokumentované v kódu a zde jejich výčet vynecháme.

Finální CFG, který si nazvěme *blokový*, má za uzly posloupnosti ESTree uzlů — *bloky* —, které jsou adaptací konceptu základního bloku pro naše použití. Před uvedením algoritmu konstrukce blokového CFG z uzlového nejprve formalizujme podmínku charakterizující uzel, jímž končí blok:

Pro uzel $n \in N(CFG_u)$ řekneme $KončíBlok(n)$, pokud

- $\|Succ_{CFG_u}(n)\| > 1$,
- $\|Pred_{CFG_u}(succ_n)\| > 1$,
- $succ_n$ je $return_u(f)$ pro nějaký podprogram f, nebo
- $succ_n$ je $throw_u(f)$ pro nějaký podprogram f,

kde $succ_n$ označuje takový uzel, že $Succ_G(n) = \{succ_n\}$.

Blokový CFG_b potom konstruujeme tak, aby splňoval

- $\forall$ blok $b = (n_i)_{i=1}^{delka}, \forall i : 1 \leq i < delka : (n_i, n_{i+1}) \in E(CFG_u)$ a zároveň
- $(b_1, b_2) \in E(CFG_b)$ právě tehdy, platí-li
  - n_1 je posledním uzlem b_1,
  - n_2 je prvním uzlem b_2,
  - $KončíBlok(n_1)$ a
  - $(n_1, n_2) \in E(CFG_u)$

Stejně jako v případě uzlového CFG zavádíme v blokovém speciální uzly $enter_b(f)$, $return_b(f)$ a $throw_b(f)$. Postup konstrukce blokového CFG je potom zanesen v algoritmu 2.

2.1.3 Analýza dosažitelných definic

Vzhledem k našemu soustředění na deobfuskaci dat patří analýza definic proměnných mezi stěžejní komponenty našeho analytického frameworku. Vycházíme při ní z algoritmu popsaného v podkapitole 1.5.4, který ale mohutně modifikujeme pro naše potřeby.

Především umožňujeme zneplatnění definice, aniž bychom ji nahradili jinou. Nalezneme-li během analýzy volání funkce, která je již zanalyzovaná, zneplatníme definice těch proměnných, do kterých daná funkce zapisuje. Pokud ovšem nejsme schopni nalézt definici volané funkce, jsme nuceni zneplatnit definice všech proměnných, poněvadž v daném případě nelze určit, které z nich mohou být voláním přepsány. Aby tento mechanizmus nevedl k přílišnému snížení efektivity analýzy, používáme seznam standardních funkcí JavaScriptu, jejichž volání žádné definice nezneplatňují.

Podobně slabý předpoklad užíváme i v případě přiřazení do dynamické pseudoproměnné (viz podkapitolu 2.1.1), ve kterém zneplatníme definice všech ostatních dynamických pseudoproměnných.

Vliv na definice proměnných zohledňujeme u syntaktických uzlů následujících typů:

- deklarátor proměnné,
- deklarátor funkce,
- operátor přiřazení,
- operátory inkrementace a dekrementace,
- operátor `delete` (má vliv na definici objektu, jehož vlastnost se maže),
- klauzule `catch` (definující proměnnou držící obsah výjimky),

Algoritmus 2 Konstrukce blokového CFG z uzlového

Vstup uzlový CFG_u
Výstup blokový CFG_b

1: $stack \leftarrow EMPTYSTACK$
2: $first \leftarrow EMPTYMAP$
3: $E(CFG_b) \leftarrow \emptyset, N(CFG_b) \leftarrow \emptyset$
4: **for all** podprogram f **do**
5: $N(CFG_b) \leftarrow N(CFG_b) \cup \{enter_b(f), return_b(f), throw_b(f)\}$
6: $first(entry_u(f)) \leftarrow enter_b(f)$
7: $first(return_u(f)) \leftarrow return_b(f)$
8: $first(throw_u(f)) \leftarrow throw_b(f)$
9: PUSH($stack$, $enter_b(f)$)
10: **while** $\|stack\| > 1$ **do**
11: $n \leftarrow pop(stack)$
12: $block \leftarrow first(n)$
13: **loop**
14: **if** $KončíBlok(n)$ **then**
15: **for all** $m \in Succ_{CFG_u}(n)$ **do**
16: **if** m nemá záznam v $first$ **then**
17: $first(m) \leftarrow (m)$
18: PUSH($stack$, m)
19: **end if**
20: $E(CFG_b) \leftarrow E(CFG_b) \cup \{(block, first(m))\}$
21: **end for**
22: **break**
23: **else**
24: PUSH($block$, $succ_n$)
25: $n \leftarrow succ_n$
26: **end if**
27: **end loop**
28: **end while**
29: **end for**

- příkaz `for-in` (zapisující do své řídicí proměnné),
- volání funkce a instanciace objektu (viz výše) a
- přístup k vlastnosti objektu (který může vyvolat getter a týká se ho tím pádem totéž, co volání funkce).

2.1.4 Emulace výpočtů

Jak bylo řečeno v podkapitole 1.7.2, obfuskovaná data rozkrýváme pomocí emulace výpočtu, který je dekóduje. Vzhledem ke stromové struktuře užité

Algoritmus 3 Vyhodnocení operátoru +

```
1: function VyhodnoťPlus(uzel)
2:     l ← hodnota(levýPotomek(uzel))
3:     if l = NEZNÁMO then
4:         return NEZNÁMO
5:     end if
6:     r ← hodnota(pravýPotomek(uzel))
7:     if r = NEZNÁMO then
8:         return NEZNÁMO
9:     end if
10:    return l + r
11: end function
```

reprezentace programu je algoritmus vyhodnocující výrazy poměrně triviální; jako příklad uvádíme algoritmus 3 emulující výpočet binárního operátoru +.

Vyhodnocení identifikátoru potom obnáší nalezení proměnné, na kterou tento odkazuje (viz podkapitolu 2.1.1), nalezení jejích definic platných v bodě použití identifikátoru (viz podkapitolu 2.1.3) a, v případě nalezení právě jedné definice, její vyhodnocení. Jak je vidno, snoubí se zde všechny dosud zmiňované aspekty analýzy programu.

Implementujeme i vyhodnocení volání funkce. Prvním krokem je opět nalezení definice volané funkce. Následně vložíme na vrchol zásobníku vyhodnocené argumenty volání a necháme vyhodnotit výraz, který je argumentem příkazu `return` v těle volané funkce (pokud je zde jediný). Když se při vyhodnocování střetneme s identifikátorem označujícím parametr funkce, vyčteme jeho hodnotu právě z vrcholu zmíněného zásobníku. Abychom zabránili potenciálně nekonečné rekurzi při emulaci, klademe omezení na velikost zásobníku.

V případech, kdy není možné výpočet emulovat, vracíme z vyhodnocení symbolickou hodnotu `NEZNÁMO`. Mezi ně patří například vyhodnocení identifikátoru, jehož platnou definici nemáme k dispozici.

2.2 Transformace

Bylo řečeno, jaké informace jsme s to z předloženého programu extrahovat. Nyní si předvedeme, jak získané výsledky uplatňujeme při samotné deobfuskační transformaci kódu.

Po vzoru knihovny Esmangle [35] dosahujeme výsledné transformace kombinací mnoha modularizovaných dílčích transformací. Jejich součinnost organizujeme na principu *fixed-point iteration*, který je zachycen v algoritmu 4.

Předpokladem tohoto algoritmu je, že jednotlivé transformace konvergují k jakési normální podobě kódu. Ukončovací podmínku v něm pak chápeme tak, že bylo této normální podoby dosaženo a není proto co dále transformovat.

Algoritmus 4 Fixed-point iteration

```
1: repeat
2:     for transformace t do
3:         proveď t
4:     end for
5: until během poslední iterace nedošlo ke změně kódu
```

Důvody ke zvolení iterativního vzoru byly jednoduchost rozšiřování o další dílčí transformace, možnost je aktivovat po jedné a v neposlední řadě jejich zapouzdření odnímající nutnost řešit jejich vzájemné závislosti.

Dílčí transformace (nebo též průchody) zpravidla účinkují na podstromech AST nehledě na jejich umístění v rámci celkového stromu. K rekurzivnímu procházení AST za účelem nalezení relevantních podstromů je použita knihovna Estraverse [36].

2.2.1 Vepsání hodnot

Dokážeme-li nějaký výraz v kódu staticky vyhodnotit, jak bylo popsáno v podkapitole 2.1.4, nahradíme jej literálem odpovídající hodnoty. Učiníme tak však pouze tehdy, je-li zajištěno, že zůstane zachována funkcionalita programu a také, že vepsání hodnoty povede ke snížení složitosti kódu. Kritéria, na jejichž základě se rozhodujeme, jsou komplexní a jejich vyčerpávající výčet neuvádíme, zejména však jde o následující:

- nahrazovaný výraz nesmí mít vedlejší účinek, který by se při nahrazení ztratil,
- je-li výraz pouhým identifikátorem a hodnotou je řetězec přesahující jistou délku, nahrazujeme jej pouze v případě, že jde o jedinou referenci na danou proměnnou, abychom zabránili nežádoucímu bobtnání výsledného kódu.

2.2.2 Vepsání jednoduchých těl funkcí

Častým obfuskačním vzorcem je nahrazení výskytu jednoduchého výrazu voláním funkce, která hodnotu tohoto výrazu vrací (viz podkapitolu 1.3.5).

Tato transformace nahrazuje volání funkce, jejíž tělo je tvořeno jediným příkazem `return`, za argument tohoto příkazu. I zde musí však dojít ke splnění jistých kriterií v zájmu zachování funkcionality, především

- nepřítomnost vedlejšího účinku volání,
- nepřítomnost rekurze a
- shodný referenční kontext volané funkce s kontextem volajícím (jinak by mohlo dojít k záměně referencí shodných jmen).

2.2.3 Expanze vyhodnocovaných řetězců

Tato dílčí transformace adresuje další typický obfuskační vzorec popsaný v podkapitole 1.3.1. V jejím rámci nahrazujeme výskyt volání funkce `eval` kódem, který má toto volání za argument, pokud jej dokážeme staticky vyhodnotit.

Podobně nahrazujeme instanciaci nové funkce konstruktorem `Function` ekvivalentním vyjádřením pomocí anonymní funkce.

2.2.4 Zjednodušení řídicího toku

Série transformací má za úkol zjednodušit řídicí tok programu.

Jednou z nich je nahrazení příkazu `if` za tělo některé jeho větve v případě, že lze staticky vyhodnotit jeho podmínku. Případ kdy je podmínka vyhodnocena jako pravdivá ilustruje následující příklad.

```
if(fun()) a = 1;
else a = 2;
```

```
fun();
a = 1;
```

Jiná transformace má na starost odstranění nedosažitelného kódu, který detekuje pomocí prostého prohledávání grafu řídicího toku (viz podkapitolu 1.7.3).

Poslední transformací zjednodušující řídicí tok je zjednodušení ternárního podmínkového výrazu, které je principiálně podobné zmíněnému zjednodušení příkazu `if`.

2.2.5 Začišťující a normalizační transformace

Řada dílčích transformací byla implementována za účelem začištění výsledného kódu. Sledujeme jimi nejen čitelnost kódu, ale také nápravu po činnosti jiných transformací.

Jedna z nich převádí jisté výskyty sekvenčního výrazu na sledy jednotlivých příkazů tvořených podvýrazy:

```
for(arr = getArray(),
len = arr.length,
i = 0; i < len; i++) {
  // ...
}
```

```
arr = getArray();
len = arr.length;
for(i = 0; i < len; i++) {
  // ...
}
```

Jiná zaměňuje užití samostatně stojícího podmiňovacího operátoru za příkaz `if`:

```
a ? b = 1 : b = 2;          ⟶      if(a) b = 1;
                                    else b = 2;
```

Dále mezi transformace z této kategorie patří náhrada dynamického přístupu k vlastnosti objektu za statický (`foo["bar"]` → `foo.bar`), odstranění zbytečných labelů příkazu `break`, odstranění deklarací nepoužitých proměnných, odstranění samostatně stojících výrazů bez vedlejšího účinku nebo eliminace zbytečných vnořených bloků.

2.2.6 Převod známých obfuskačních vzorců

Abychom adresovali některé jednoduché obfuskační vzorce nepostihnutelné obecnými transformacemi, implementujeme průchod, v rámci nějž testujeme shodu podstromů AST s nějakými známými vzorci a v případě kladného nálezu nahradíme nějakou hodnotou. V současné chvíli je použita jediná transformace tohoto typu, která převádí výraz `[] + location` na `"http"` (viz podkapitolu 1.3.7).

2.2.7 Přejmenování proměnných

Volitelnou transformací, která je jako jediná prováděna až po iteraci, je přejmenování proměnných řízené několika jednoduchými pravidly.

- Proměnnou inkrementovanou v rámci příkazu `for` se pokusíme pojmenovat „i“.
- Proměnou, jíž je přiřazen výsledek volání funkce `getCosi` se pokusíme přejmenovat na „cosi“.
- Proměnnou, které je přiřazen výsledek instanciace `new Cosi()` se pokusíme přejmenovat na „cosi“.
- Nalezneme-li přiřazení jehož pravou stranu jsme schopni staticky vyhodnotit (viz podkapitolu 2.1.4), pokusíme se proměnnou, do které je přiřazováno, přejmenovat na název třídy, jíž je daná hodnota instancí, s prvním písmenem malým.

Pokud pokus o přejmenování neuspěje z důvodu kolize jmen, doplníme jméno číslem, které zvyšujeme dokud se kolizi nevyhneme.

Podotkněme, že neimplementujeme žádnou detekci obfuskovanosti jmen a přejmenováváme jednoduše všechny proměnné.

2.3 Zhodnocení výsledků

Účinnost deobfuskace byla testována nad kódy z několika zdrojů.

Součástí testů bylo spuštění našeho deobfuskátoru nad výstupy obfuskátorů Javascript Obfuscator [16], JavaScript Obfuscator [17], Jscrambler [18] a packer [19] s kódem jednoduchého javascriptovského programu na vstupu. Zatímco výstupu prvého jmenovaného se nám daří zcela navrátit původní podobu před obfuskací (až na názvy proměnných), deobfuskace ostatních byla úspěšná méně. JavaScript Obfuscator a packer nicméně vnášejí do kódu dobře detekovatelné vzorce a bylo by možné zcela odhalit původní podobu jimi obfuskovaného kódu pomocí cílené transformace, na kterou jsme se v této práci nesoustředili (opět ovšem vyjma jmen proměnných).

Další test proběhl na sadě vzorků složené z reálných vzorků škodlivého software poskytnutých vedoucím práce doplněných o výstupy z některých zmíněných obfuskátorů a vlastní příspěvky na celkový počet 15. Následně bylo 29 JavaScriptu znalých lidí osloveno, aby ohodnotili srozumitelnost vzorků a jejich námi deobfuskovaných protějšků na stupnici od jedné do deseti. Tento test prokázal jistou úspěšnost. V průměru byly deobfuskované vzorky hodnoceny 1,45krát lépe, než jejich původní verze. Nejlepší vzorek byl deobfuskát výstupu nástroje Javascript Obfuscator, jehož srozumitelnost byla navýšena 2,2krát. Nejhorším potom byl vzorek reálného malware, který byl jediný, jemuž byla podle dotazníku srozumitelnost dokonce snížena. Přičíst to lze na vrub tomu, že jisté srozumitelné pasáže kódu byly zcela eliminovány v rámci statického vyhodnocení, což zvýšilo poměr nesrozumitelných částí vůči srozumitelným. Podrobné výsledky jsou k dispozici na přiloženém CD (příloha A).

Vzhledem k subjektivní povaze srozumitelnosti kódu jsme nepřistoupili k automatizovanému testování; žádná z nabízejících se metrik složitosti programu nevypovídá pro naše účely dostatečně o složitosti při čtení člověkem.

Jak ale bylo řečeno, v některých případech se daří téměř přesně zrekonstruovat podobu kódu před obfuskací, což z našeho nástroje činí i kandidáta na součást automatického detektoru škodlivého kódu založeného na hledání známých vzorců.

2.3.1 Známé problémy

V současné implementaci deobfuskátoru je přítomno několik nedostatků, které v krajních případech vedou ke změně funkcionality vystoupeného kódu.

```
1  var obj = {};
2
3  Object.defineProperty(obj, "a", {
4      get: function() { return this.b * 2; },
5      set: function(hodnota) { return this.b = hodnota; }
6  });
7
8  with(obj) {
9      a = 2;
10     console.log(a); // vystoupí se 4, nikoliv 2
11 }
```

Ukázka kódu 7: dynamický kontext příkazu `with`

Jedním z nich je změna referenčního kontextu kódu expandovaného z řetězce (viz podkapitolu 2.2.3). Dle specifikace při volání funkce `eval` nepřímým odkazem má k vyhodnocení dojít v globálním kontextu, což náš nástroj nereflektuje. Uvažme následující příklad.

```
var a = 1, evil = eval;
function vraťA() {
  var a = 5, výs = evil("a");
  return výs;
}
console.log(vraťA());
```

→

```
var evil = eval;
console.log(5);
```

Původní kód vypíše do konzole hodnotu `1`, avšak transformovaný vypíše `5`. Stejný nedostatek je přítomný i u zpracování těla funkce v argumentu konstruktoru `Function`.

Jiným vědomým zanedbáním je analýza dynamických referenčních kontextů jiných, než těch vytvořených funkcí. Například v těle příkazu `with` dynamické pseudoproměnné nezavádíme. Tento příkaz se však v reálném kódu prakticky nevyskytuje a práce s ním by stále byla nespolehlivá, jak ilustrujeme v ukázce 7 (věnuj pozornost řádku 10).

Závěr

Cílem práce bylo analyzovat způsoby obfuskace jazyka JavaScript a možnosti rozboru tohoto jazyka a následně navrhnout a implementovat nástroj pro jeho deobfuskaci.

V rámci rešerše byly rozebrány obvyklé obfuskační vzorce, výzvy, které představují pro deobfuskaci a také některé koncepty z teorie překladačů, které mají využití při implementaci deobfuskačního nástroje.

Následně byla představena naše implementace čistě statického deobfuskátoru jazyka JavaScript. V práci jsme se věnovali klíčovým algoritmům využitým při jeho implementaci s odkazy na metody překladačové teorie, z níž tyto algoritmy čerpají. Implementovaný nástroj nabízí řadu specializovaných transformací kódu, které vedou ke zvýšení jeho srozumitelnosti, a to zejména konstantních dat v něm. Zvolený návrh zároveň umožňuje budoucí rozšiřování o další dílčí transformace.

Na závěr jsme provedli praktický test výsledků implementovaného nástroje, který prokázal, že jeho užití vede k nezanedbatelnému zvýšení srozumitelnosti kódu pro lidského čtenáře a že je smysluplné se směrem námi zvoleného návrhu ubírat i nadále.

Do budoucna se nabízí rozšiřování nástroje o transformace cílené na výstupy konkrétních obfuskátorů. Dále by bylo vhodné rozšířit analytický framework, a to zejména o schopnost pracovat s vlastnostmi objektů stejně, jako s běžnými proměnnými, jejíž absence je v současnosti překážkou deobfuskace některých známých vzorců. Za pozornost stojí i vyhodnocování hodnot počítaných v cyklech, které v dosavadní podobě nástroje rovněž není dobře postiženo. Samozřejmě by bylo na také místě adresovat známé problémy zmíněné v práci.

Zajímavým námětem na další výzkum by mohl být optimalizační aspekt některých transformací: protože ostatně využíváme konceptů z překladačové optimalizace, jisté transformace (zejména statická vyhodnocení výrazů) by mohly vést ke zrychlení běhu programu.

Literatura

[1] ECMAScript Language Specification. Standard, ECMA International, Geneva, CH, 2009.

[2] Scripting language. In *Wikipedia: the free encyclopedia* [online]. St. Petersburg (Florida): Wikipedia Foundation, 1.5.2018 [cit 6.5.2018]. Dostupné z: `https://en.wikipedia.org/wiki/Scripting_language`.

[3] Interpreted language. In *Wikipedia: the free encyclopedia* [online]. St. Petersburg (Florida): Wikipedia Foundation, 5.5.2018 [cit 6.5.2018]. Dostupné z: `https://en.wikipedia.org/wiki/Interpreted_language`.

[4] JOHNS, M.: On JavaScript malware and related threats. *Journal in Computer Virology*, ročník 4, č. 3, 2008: s. 161–178, doi:10.1007/s11416-007-0076-7.

[5] WANG, J.; Xue, Y.; Liu, Y.; aj.: Jsdc: A hybrid approach for javascript malware detection and classification. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, ACM, 2015, s. 109–120, doi:10.1145/2714576.2714620.

[6] TOPF, J.: The HTML Form Protocol Attack. *BugTraq posting [online]*, 2001 [cit. 6.5.2018]. Dostupné z: `http://www.remote.org/jochen/sec/hfpa/hfpa.pdf`

[7] OBSCURE: Extended HTML Form Attack. Making use of Non-HTTP protocols to launch Cross Site Scripting attacks. *Eye on Security*, Copyright © 2001,2002 eyeonsecurity Inc. Dostupné z: `http://eyeonsecurity.org/papers/ExtendedHTMLFormAttack.htm`

[8] PROVOS, N.; McNamee, D.; Mavrommatis, P.; aj.: The Ghost in the Browser: Analysis of Web-based Malware. *HotBots*, ročník 7, 2007: s. 4–4.

[9] Oakland, Kalifornie: npm Inc. *npm* [online]. 2014 [cit. 6.5.2018]. Dostupné z: `https://www.npmjs.com`.

[10] BRADLEY, B. npm removes malicious JavaScript packages that were caught stealing data. In *SC Magazine* [online]. Copyright © 2018 Haymarket Media, Inc., 2017 [cit. 5.6.2018]. Dostupné z: https://www.scmagazine.com/npm-removes-malicious-javascript-packages-that-were-caught-stealing-data/article/680029/.

[11] DEFCON 20: Owning Bad Guys {And Mafia} With Javascript Botnets. In: Youtube [online]. 22.5.2013 [cit. 6.5.2018]. Dostupné z: https://youtu.be/0QT4YJn7oVI. Kanál uživatele Maligno Alonso.

[12] AL-TAHARWA, I. A.; aj.: JSOD: JavaScript obfuscation detector. *Security and Communication Networks*, ročník 8, č. 6, 2015: s. 1092–1107, doi:10.1002/sec.1064.

[13] COLLBERG, C.; Thomborson, C.; Low, D.: A taxonomy of obfuscating transformations. Technická zpráva, Department of Computer Science, The University of Auckland, New Zealand, 1997.

[14] HOWARD, F.: Malware with your Mocha? Obfuscation and antiemulation tricks in malicious JavaScript. *Sophos Technical Papers*, 2010: str. 14.

[15] PALLADINO, P: Brainfuck beware: JavaScript is after you!. In *Patricio Palladino blog* [online]. 10.4.2012 [cit. 6.5.2018]. Dostupné z: http://patriciopalladino.com/blog/2012/08/09/non-alphanumeric-javascript.html.

[16] Javascript Obfuscator [online]. In *GitHub*. 2016 [cit. 6.5.2018]. Dostupné z: https://github.com/javascript-obfuscator/javascript-obfuscator.

[17] Javascript Obfuscator [online]. 2017 [cit. 6.5.2018]. Dostupné z: https://javascript-obfuscator.org/.

[18] Jscrambler [online]. Copyright © Jscrambler 2018 [cit. 6.5.2018]. Dostupné z: https://jscrambler.com/.

[19] /packer/ [online]. Copyright © 2004-2018 Dean Edwards [cit. 6.5.2018]. Dostupné z: http://dean.edwards.name/packer/.

[20] MAZA, M. M. *Compiler Theory: Code Optimization*. Ontario, Kanada: University of Western Ontario, 2004 [cit. 6.5.2018]. Dostupné z http://www.csd.uwo.ca/~moreno/CS447/Lectures/CodeOptimization.html/CodeOptimization.html.

[21] BLANC, G.; Ando, R.; Kadobayashi, Y.: Term-rewriting deobfuscation for static client-side scripting malware detection. In *New Technologies,*

Mobility and Security (NTMS), 2011 4th IFIP International Conference on, IEEE, 2011, s. 1–6, doi:10.1109/NTMS.2011.5720649.

[22] LU, G.; Debray, S.: Automatic simplification of obfuscated JavaScript code: A semantics-based approach. In *Software Security and Reliability (SERE), 2012 IEEE Sixth International Conference on*, IEEE, 2012, s. 31–40, doi:10.1109/SERE.2012.13.

[23] RIECK, K.; Krueger, T.; Dewald, A.: Cujo: efficient detection and prevention of drive-by-download attacks. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACM, 2010, s. 31–39, doi:10.1145/1920261.1920267.

[24] KAPLAN, S.; Livshits, B.; Zorn, B.; aj.: "NOFUS: Automatically Detecting"+ String. fromCharCode (32)+"ObFuSCateD". toLowerCase ()+"JavaScript Code. *Technical report, Technical Report MSR-TR 2011–57, Microsoft Research*, 2011.

[25] XU, W.; Zhang, F.; Zhu, S.: JStill: mostly static detection of obfuscated malicious JavaScript code. In *Proceedings of the third ACM conference on Data and application security and privacy*, ACM, 2013, s. 117–128, doi:10.1145/2435349.2435364.

[26] CHOI, Y.; Kim, T.; Choi, S.; aj.: Automatic detection for javascript obfuscation attacks in web pages through string pattern analysis. In *International Conference on Future Generation Information Technology*, Springer, 2009, ISBN 978-3-642-10509-8, s. 160–172.

[27] BLANC, G.; Kadobayashi, Y.: A step towards static script malware abstraction: Rewriting obfuscated script with maude. *IEICE TRANSACTIONS on Information and Systems*, ročník 94, č. 11, 2011: s. 2159–2166, doi:10.1587/transinf.E94.D.2159.

[28] RAYCHEV, V.; Vechev, M.; Krause, A.: Predicting program properties from big code. In *ACM SIGPLAN Notices*, ročník 50, ACM, 2015, s. 111–124, doi:10.1145/2676726.2677009.

[29] VASILESCU, B.; Casalnuovo, C.; Devanbu, P.: Recovering clear, natural identifiers from obfuscated JS names. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ACM, 2017, s. 683–693, doi:10.1145/3106237.3106289.

[30] Javascript Unobfuscator [online]. © 2018 dCode — The ultimate 'toolkit' to solve every games / riddles / geocaches [cit. 6.5.2018]. Dostupné z: `https://www.dcode.fr/javascript-unobfuscator`.

[31] The ESTree Spec [online]. In *GitHub*. 2015 [cit. 6.5.2018]. Dostupné z: `https://github.com/estree/estree`.

[32] Espree [online]. In *GitHub*. 2018 [cit. 6.5.2018]. Dostupné z: `https://github.com/eslint/espree`.

[33] Escope [online]. In *GitHub*. 2018 [cit. 6.5.2018]. Dostupné z: `https://github.com/estools/escope`.

[34] WALA [online]. In *GitHub*. 2016 [cit. 6.5.2018]. Dostupné z: `https://github.com/wala/JS_WALA`.

[35] SUZUKI, Y.: Escodegen and Esmangle: Using Mozilla JavaScript AST as an IR. In *Proceedings of the 2013 ACM on Aspect-Oriented Software Development*, Aspect-Oriented Software Development, 2013.

[36] Estraverse [online]. In *GitHub*. 2016 [cit. 6.5.2018]. Dostupné z: `https://github.com/estools/estraverse`.

Příloha A

Obsah přiloženého CD

- `deobfuscate` ..... adresář s implementací
  - `README.md` ..... stručný manuál k použití
- `thesis`
  - `src` ..... zdrojová forma práce ve formátu LaTeX
  - `BP_Adam_Platkevic_2018.pdf` ..... text práce ve formátu PDF
- `vysledky_dotazniku` ..... data z dotazníku na srozumitelnost kódu
  - `samples` ..... předkládané vzorky
  - `results.csv` ..... záznam odpovědí
  - `stats.nb` ..... Mathematica notebook se statistikami
- `README.txt` ..... stručný popis obsahu CD