UNIVERSITÀ DEGLI STUDI DI BRESCIA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Corso di Laurea
in Ingegneria Informatica

Relazione Finale
Design and implementation of Bulldozer
A decompiler for JavaScript "binaries"

Relatore: Chiar.mo Prof. Francesco Gringoli

Laureando:
Federico Cerutti
Matricola n. 715256

Anno Accademico 2020-2021

Contents

Foreword **1**

1 Introduction **3**
1.1 Motivation . . . 3
1.2 JavaScript . . . 3
1.3 Obfuscation . . . 4
1.4 User tracking and fingerprinting . . . 4

2 Quick Remarks of Compiler Theory **7**
2.1 Graph Theory . . . 7
2.1.1 Graph Traversal . . . 7
2.1.2 Dominance . . . 8
2.2 Abstract Syntax Trees . . . 9
2.3 Basic Blocks . . . 11
2.4 Control Flow Graphs . . . 12

3 Obfuscation and JavaScript **13**
3.1 Obfuscation techniques . . . 13
3.2 Obfuscated JavaScript . . . 14
3.3 Obfuscated tracking . . . 16

4 collina.js **19**
4.1 Control Flow Flattening . . . 20
4.2 Additional Obfuscation Techniques . . . 24

5 bulldozer.js **27**
5.1 Source Code Parsing . . . 27
5.2 CFG Generation . . . 29
5.2.1 AST to Map-of-Maps . . . 29
5.2.2 Basic Blocks Extraction . . . 30
5.2.3 Code Traversal . . . 31
5.2.4 Branching Condition Recovery . . . 32
5.3 Region Identification . . . 33
5.3.1 Cyclic Regions . . . 34
5.3.2 Acyclic Regions . . . 38
5.4 Code Structuring . . . 39

5.4.1 Acyclic Regions 40
5.4.2 Cyclic Regions 42
5.5 Partial Evaluation 42
5.5.1 Variable Descrambling 43
5.5.2 Static Replacement Rules 45
5.5.3 Observations on the Partial Evaluation System 47

6 Findings **49**
6.1 Quirks 50

7 Conclusion **53**

Appendices **55**
A e Function Arguments 57
B Global Storage and Bitmasks 58

Glossary **63**
Acronyms 63
Glossary 63

Bibliography **65**

Foreword

I would like to express my deepest appreciation to Daniele Barattieri, a fellow student who is the best pair programmer I've ever had the pleasure to work with. He somehow decided, yet another time, join me in this project; as if my guiltless mislaid confidence in a swift conclusion never tricked him before. He co-authored part of the code and had to deal with my daily 2AM brain dumps.

I would like to thank Danya and Ettore for proofreading this document.

Particularly helpful to me during these years was prof. Francesco Gringoli, whose acknowledgment encouraged me and my friends to embark on new projects, and who became a point of reference for all the university-related topics.

Thanks to all the members of the Muhack hackerspace for the welcoming and stimulating environment where this and many other personal projects were born. I'm still grateful for how you enthusiastically elected me as a president, just as if I wasn't the *only* candidate.

Finally, I'd also like to extend my gratitude to my family and friends, the ones that were by my side through these years.

Accompanying code for this document can be accessed on the online repository

`https://github.com/ceres-c/bulldozer`

or by scanning this QR code

Chapter 1

Introduction

1.1 Motivation

Websites push JavaScript (JS) to clients for them to execute it in a protected and (often) sandboxed environment, but ultimately code is being run on the user's machine. New vulnerabilities are constantly found in JavaScript engines [NCFb] and web browsers [NCFa], [NCFc], advanced time-dependent attacks have been ported to the web [GMM16] [RJ21] as well as more traditional attacks such as crypto miners [NT18]. JavaScript has proven to be a valuable attack vector for independent and state-sponsored groups.

Defenders need to analyze scripts to ensure no threat is delivered and to respond to potential security incidents. Analysis of a JS script is typically less labor-intensive than binary reverse engineering, being the source in a script file commonly more understandable than the decompiled source of a binary. Nonetheless, this is not always the case, since specific solutions have been developed to make JS unreadable. The object of this thesis will be a tool written to analyze a specific obfuscated JavaScript file, delivered to users by a well-known e-commerce website for tracking purposes.

1.2 JavaScript

JavaScript is a high-level programming language conforming to the ECMAScript (ES) specification by Ecma International. ECMAScript is a standard and JavaScript is only one of the languages implementing the standard, but the two names are often used interchangeably.

Designed by Brendan Eich as a scripting language for the Netscape Navigator browser [Rau14] in 1993, its original purpose was to enrich web pages with minimal client side scripting features such as form validation. Its role remained minor until the development technique known as Asynchronous JavaScript and XML (AJAX) became the standard for dynamic webpages: thanks to AJAX a page is loaded once, but its content changes dynamically. Another breakthrough was Google's V8 engine, with its Just In Time (JIT) compilation approach, which pushed the language possibilities further, drastically improving performance and paving the way to advanced usages and present ubiquity of JavaScript.

JavaScript clearly outgrew its original scope for the first time when the Node.js runtime environment was created. Node.js is a backend runtime environment for JavaScript with an event-driven asynchronous architecture, which granted sufficient performances albeit no multithreading support was available. Once Node.js proved its value, many other frameworks revolving around JavaScript were born for different purposes such as Electron, React and React Native. Nowadays JavaScript is one of the world's most used programming languages.

1.3 Obfuscation

Source code protection has always been an issue for developers and companies. This secrecy requirement is often intensified by the presence of secrets in the source itself, despite security guidelines mandating to never leave sensitive information accessible to users. Rushed code or developers inexperience can lead to poorly engineered software and violations of the aforementioned security precept. One of the approaches to solve this issue is *Security By Obscurity*, a widely criticized belief that a system can be secured by enforcing confidentiality on the system's inner workings alone. To make software obscure, developers obfuscate its code. Obfuscation is therefore the process, be it manual or automatic, of generating code difficult to read.

It's important to note that obfuscation is often used to hide malicious software and for this very reason it is often difficult to quickly discern legitimate intellectual property protection from criminal intents without in-depth analysis.

1.4 User tracking and fingerprinting

Information about users' activities are collected during the navigation by websites operators and third parties to infer preferences and serve targeted content. For example, services such as Netflix and Spotify offer personalized content basing on customer's interests', Google finalizes search results in light on previously visited links, Google's reCaptcha service implements since (version 2) behavioral analysis to separate humans from bots and protect specific parts of websites. At the same time, tracking can be performed for commercial purposes such as advertisement targeting or rating a user trustworthiness.

It is even possible for some actors to track a user through his entire online life. Google's omnipresent ads and Facebook's iconic Like button, embedded in every website, allow the two companies to track which pages a user is visiting, even outside their domain.

Tracking is powered by many different techniques which can be summed up in these main categories:

- **Cookies**: Cookies are small chunks of information saved by every website among other browser data, used to store information about user preferences or session data. Cookies can, of course, store a unique identifier of the user.
- **Fingerprinting**: A digital fingerprint is a set of characteristics relative to a machine or operating system installation, such as screen size/resolution, installed fonts, hardware components, driver behavior and more. By putting all these apparently insignificant information together, it is possible to identify unambiguously a specific user. See the Electronic Frontier Foundation online demo [EFF].
- **Behavioral**: Leveraging the microscopic differences in behavior characterizing every human being, it is possible to identify a visitor by its movements on a webpage. Previous studies showed the possibility to manage authentication via keypress analysis [BBB20] or to unlock a phone with sensors and speaker data alongside usage statistics [Lóp+21].

Cookies and Fingerprinting/Behavioral data greatly differ in pervasiveness. Cookies, being saved on the user's machine, can be deleted or ignored with specific browser features (private/incognito browsing). On the other hand, it is complicated to mask a computer's hardware and its quirks: fingerprint data might be sent to servers alongside required information and processed opaquely by the backend without the user even noticing.

Chapter 2

Quick Remarks of Compiler Theory

This chapter will give a brief account of constructs and data structures used by compilers and decompilers to parse, transform and output code. An in-depth exposition of these concepts is available in many books ([Knu97], [Aho+06]) and university courses ([Fis05]) on the topic.

2.1 Graph Theory

2.1.1 Graph Traversal

The process of visiting all the nodes in a graph is called traversal, the order in which the nodes are visited classifies the traversal. If the graph is directed, it is possible to generate a linear ordering of the nodes, known as **Topological Sorting**, such that for every directed edge $u \rightarrow v$ from vertex u to vertex v, u comes before v.

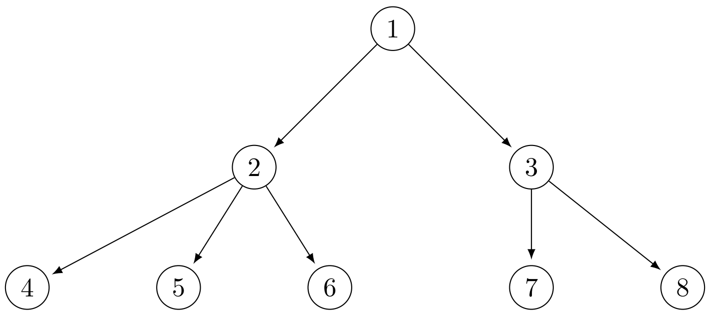


Figure 2.1: Node traversal example graph

- **Breadth-first** - Nodes are visited by level: every node of a level is visited before moving to the next depth level

Topological sort: 1 2 3 4 5 6 7 8

- **Depth-first** - Nodes are visited exploring as far as possible along each branch before backtracking. In this case, the ordering can differ:
  - **Preorder** - the parent node is visited before the children
    Topological sort: 1 2 4 5 6 3 7 8
  - **Postorder** - the parent node is visited after the children
    Topological sort: 4 5 6 2 7 8 3 1
  - **Inorder** - The parent node is visited after the first $n-th$ child. This is mostly used in binary graphs (in which each node has 2 children) with $n = 1$, but can be used in non-binary graphs defining a custom n
    Topological sort with $n = 2$: 4 5 2 6 7 8 3 1
    Topological sort with $n = 1$: 4 2 5 6 1 7 3 8

Depth-first preorder of Figure 2.1 results in an ordering where every node is visited before its successors, but this is not always the case with cyclic graphs. If there was an edge from 6 to 7, the preordering would be 1 2 4 5 6 7 3 8: 7 precedes 3, albeit 7 is a successor of 3; this is not a topological sorting anymore. To solve this issue, **reverse ordering** (topological sorts read in reverse order) is often used, and reverse postorder achieves topological sorting for graphs with cycles: it guarantees a node is seen before all of its successors.

2.1.2 Dominance

Dominance is a characteristic of nodes in a directed graph: a node d **dominates** node i, written $d \operatorname{dom} i$, if every possible execution path from `ENTRY` to i includes d. Dually, p **post-dominates** i, $p \operatorname{pdom} i$, if every possible execution path from i to `EXIT` includes p. $\operatorname{dom}(a)$ and $\operatorname{pdom}(a)$ identify the **set** of **dominators** and **post-dominators** of a.

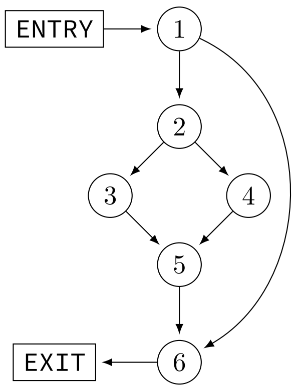


Figure 2.2: Dominance

A node d **strictly (post-)dominates** (sdom) a node i if $d \operatorname{dom} i \ \wedge \ d \neq i$.

The **Immediate (post-)dominator** (idom) of a node i is the "closest" among i's dominators. Formally the immediate dominator is the strict dominator of i which does not dominate any other strict dominator of i.

The **Dominance frontier** of a node i (df(i)) is the set of all nodes that are successors to nodes dominated by i and are not dominated by i.

Given the graph in Figure 2.2:

- $1 \operatorname{dom} 3,\ 2 \operatorname{dom} 3,\ 3 \not\operatorname{dom} 5$
- $6 \operatorname{pdom} 1,\ 6 \operatorname{pdom} 3,\ 3 \not\operatorname{pdom} 2$
- $1 \operatorname{idom} 2,\ 2 \operatorname{idom} 3,\ 2 \operatorname{idom} 5$
- $\operatorname{df}(1) = \emptyset,\ \operatorname{df}(2) = \{6\},\ \operatorname{df}(3) = \{5\}$

2.2 Abstract Syntax Trees

An Abstract Syntax Tree (AST) is a tree structure which represents the *Syntax* of a piece of source code in a hierarchical fashion. The graph is *Abstract* because some details are not explicitly represented in the tree and can be inferred from the tree structure, such as parentheses or statements closures (“`;`” in C or Java). An AST does not bear semantic information, but it’s still a powerful analysis tool, allowing for:

- Code validation
- Code manipulation
- Code highlighting
- Language conversion (among revisions of the same language)

In fact, most modern Integrated Development Environments (IDEs) and source-code editors internally use ASTs for highlighting and spotting syntax errors.

It is clear from Figure 2.3 that Abstract Syntax Trees are information-rich structures, much more verbose than the original source code. Every node of the tree can potentially have a considerable number of children nodes, like in the case of an `if-else` construct with a large body for every case. All the top level statements in a script are children of the main `Script` node (missing in 2.3b) to retain a tree-like structure. Note that AST are not, in general, binary trees since the number of children nodes may vary.

The concepts behind AST are universal and applicable to every programming language, but different languages require specific AST implementations due to the strong connection to the syntax and language capabilities. Multiple AST specifications have been written for JavaScript (JS) such as SpiderMonkey’s internal ESTree [Moz], Babel’s own format [Bab] and, finally, Shift-AST from Shape Security [Shaa]. A great variety of tools spawned from every of these projects, but Shift-AST was picked for the project which will be discussed in this thesis by virtue of the powerful `shift-reducer`, a monoidal tree traversal library which will be introduced.

```javascript
if (i > 5) {
  console.log('Greater');
} else {
  console.log('Less');
}
```

(a) Example code

```
statement[0]
├ type:IfStatement
├ test:
│  ├ type:BinaryExpression
│  ├ left:
│  │  ├ type:IdentifierExpression
│  │  └ name:i
│  ├ operator:>
│  └ right:
│     ├ type:LiteralNumericExpression
│     └ value:5
├ consequent:
│  └ statement[0]
│     ├ type:ExpressionStatement
│     └ expression:
│        ├ type:CallExpression
│        ├ callee:
│        │  ├ type:StaticMemberExpression
│        │  ├ object:console
│        │  └ property:log
│        └ arguments:
│           └ arg[0]
│              ├ type:LiteralStringExpression
│              └ value:Greater
└ alternate:
   └ statement[0]
      ├ type:ExpressionStatement
      └ expression:
         ├ type:CallExpression
         ├ callee:
         │  ├ type:StaticMemberExpression
         │  ├ object:console
         │  └ property:log
         └ arguments:
            └ arg[0]
               ├ type:LiteralStringExpression
               └ value:Less
```

(b) AST representation of 2.3a

Figure 2.3: The AST generated from a piece of JavaScript code, correspondent sections are highlighted

2.3 Basic Blocks

A Basic Block is a maximal sequence of statements with a single entry and a single exit: no code inside a basic block is destination of a jump and only the last instruction can cause the program to execute code in a different basic block. This is the most granular unity a program can be divided in. Given the code in a basic block is not branching, it can be treated as an atomic opaque group of statements.

```c
void function() {
    int i = 0;
    int c = 0;
    int x = 10;

    if (x > 5) {
        x = x % 2;
    }

    while (x-- > 0) {
        i++;
        c = 5;
    }

    x = 10;
    c = 10;
}
```

Listing 2.1: Example C code with basic blocks highlighted

2.4 Control Flow Graphs

A Control Flow Graph (CFG) is a directed graph which represents all the paths that might be traversed during the execution of the program. Every node of the graph is a basic block and the edges represent possible transfers of control flow from one basic block to another. Two additional node types are present: `ENTRY` blocks to mark the entry point of the section and `EXIT` blocks where execution ends; multiple entries/exits are possible.

A CFG can be generated at different levels: source, Intermediate Representation (IR), and machine code; for example, the graph in Figure 2.4 is a source-level CFG of the code in Listing 2.1: edges connect every basic block to its possible successors.

Compilers leverage heavily CFGs to analyze and optimize programs; likewise, decompilers use them to recover control flow structures from machine code. In fact, this structure bears semantic value: given a node in the graph, it is possible to deduce information regarding the program past and future state.

Being a *Graph*, the presence of cycles must be accounted for when traversing a CFG.

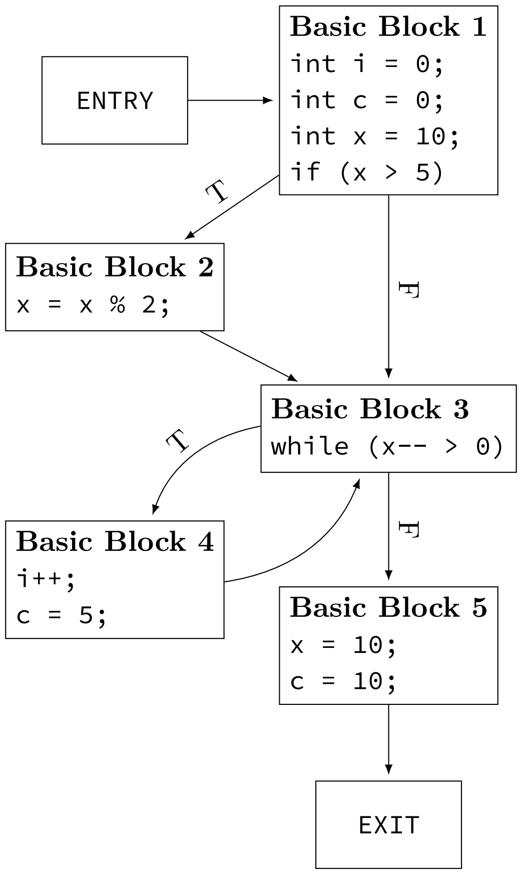


Figure 2.4: CFG of Listing 2.1

Chapter 3

Obfuscation and JavaScript

3.1 Obfuscation techniques

Held since 1984, the International Obfuscated C Code Contest (IOCCC) had some incredibly complex entries: chess engines, flight simulators, ray tracers, emulators and complete operating systems, all in fewer than 4096 bytes. These are excellent examples of *recreational obfuscation*: the art of writing unreadable code. Each of the IOCCC entries took the developer(s) months or even years to write, which is of course incompatible with the needs of enterprise software development.

For this purpose, countless automatic code obfuscators have been authored and released online or marketed. These programs accept source code in input and produce either obfuscated source or directly compiled binaries. Automated tools apply to the source code predefined tactics, such as:

- **Control logic obfuscation**
  The logic ruling branching and decision-making in the software is obfuscated
  - *Control flow flattening*: The Control Flow Graph (CFG) is shallow, does not develop in depth, because every basic block is redirected to a main dispatcher; the dispatcher steers execution from one basic block to the subsequent.
  - *Bogus control flow*: Useless control flow paths that cannot possibly be followed are deliberately added to the code, undermining a purely static analysis as the amount of code to analyze increases greatly. The complexity can be increased even further, via opaque predicates, preventing both software and a human analyst from inferring the execution flow.
  - *Probabilistic execution*: An evolution of the aforementioned Bogus control flow: blocks with different syntax, but equal behavior, are executed in an apparently arbitrary way.
- **Layout obfuscation**
  The layout of the code is obfuscated retaining the original syntax
  - *Identifier replacement*: Original variables are replaced with meaningless names

such as 1-letter identifiers or purposefully confusing names (characters such as `O`, `o`, `0`). Identifiers are also reused in different scopes.
    - *Dead code injection*: Useless instructions are added, such as `NOP` in the case of binaries.
    - *Symbols stripping*: Non-necessary symbols, such as debugging information or symbols table in `ELF` files, are removed.
- **Data obfuscation**
  Data referenced in the program is obfuscated
    - *Data encoding*: Data is encoded with some compression or encryption algorithm and decoded at runtime.
    - *Data splitting*: Contiguous data is divided and relocated in order to appear scrambled.
    - *Literal to function call conversion*: Literals are replaced with calls to a function which returns the literal, given some specific arguments.
- **Anti debugging**
  The software, noticing it is being analyzed with a debugging tool such as gdb, x64dbg, and WinDbg, refuses to execute or take specific code paths hiding relevant information.
- **Anti virtual machine**
  Likewise, noticing being run in a virtual machine environment, the software changes its behavior.

Some of the above solutions are relevant to binaries and come naturally in assembly; for example in ELF Specification [Com95] all the headers are marked as optional, while operating systems may require some of those headers, the remaining ones can be stripped. Moreover, in assembly it is customary, albeit not recommended, to `JMP` straight into parts of memory (leading to *spaghetti code*), which means programmers unwillingly achieve a sort of control flow flattening. Intensive usage of jumps, especially if jumping from or to the middle of a procedure, confuses readers greatly.

The lack of **`goto`** statements in high-level languages such as JavaScript (JS) is an intentional choice by language designers, made to force developers into writing more understandable code. At the same time the absence of jump-like instructions hinders the application of control-flow flattening obfuscation but, as it will be discussed in chapter 4, it is still possible to emulate this behavior.

3.2 Obfuscated JavaScript

Being JavaScript extremely widespread and due to it being normally distributed as plain text `.js` files, it is a recurring need for developers to find a way to protect their source code. To meet the demand, there are multiple open source and commercial obfuscation tools such as:

- javascript-obfuscator `https://obfuscator.io/` (open source)
- jsfuck `http://www.jsfuck.com/` (open source)

- Terser `https://terser.org/` (open source)
- JavaScript Obfuscator `https://javascriptobfuscator.com` (closed source)
- JSDefender `https://www.preemptive.com/products/jsdefender` (closed source)
- Jscrambler `https://jscrambler.com/` (closed source)

which all implement a (sub)set of the techniques cited in section 3.1.

It is worth nothing that, with JS sources, identifier replacement serves as a performance improvement technique as well. In fact, replacing descriptive variables with shorter identifiers shrinks the size of the file pushed to the user, reducing download times and therefore improving user experience. For example, Listing 3.2 is 60% smaller than the original Listing 3.1.

```
// Promise to roll a dice
let rollASix = new Promise(function (resolve, reject) {
  let result = Math.floor(Math.random() * 6) + 1;
  if (result === 6) {
    resolve(result);
  } else {
    reject(result);
  }
});

function logAndRollAgain(roll) {
  console.log("Rolled a " + roll + ", try again.");
  return rollASix;
}
function logSuccess(roll) {
  console.log("Yay, you rolled a " + roll + ".");
}
function logFailure(roll) {
  console.log("Bad luck, you rolled a " + roll);
}

// Use promise to roll a dice 3 times, waiting for a 6
rollASix.then(null, logAndRollAgain) // Roll first time
  .then(null, logAndRollAgain) // Roll second time
  .then(logSuccess, logFailure); // Roll third and last time
```

Listing 3.1: Example JS code to roll a dice 3 times

```
let o=new Promise((function(o,l){let
↪ n=Math.floor(6*Math.random())+1;6===n?o(n):l(n)}));function
↪ l(l){return console.log("Rolled a "+l+", try
↪ again."),o}o.then(null,l).then(null,l).then((function(o){
↪ console.log("Yay, you rolled a
↪ "+o+".")}),(function(o){console.log("Bad luck, you rolled a
↪ "+o)}));
```

Listing 3.2: Listing 3.1 after being processed by Terser

Code in Listing 3.2 might seem incomprehensible at a first glance, but a simple beautification (see Listing 3.3) will make it instantly recognizable as the same code of Listing 3.1. Identifier replacement alone, therefore, does not increase significantly the effort required to reverse engineer a piece of software

```
let o = new Promise((function (o, l) {
  let n = Math.floor(6 * Math.random()) + 1;
  6 === n ? o(n) : l(n)
}));

function l(l) {
  return console.log("Rolled a " + l + ", try again."), o
}
o.then(null, l).then(null, l).then((function (o) {
  console.log("Yay, you rolled a " + o + ".")
}), (function (o) {
  console.log("Bad luck, you rolled a " + o)
}));
```

Listing 3.3: Listing 3.2 beautified

3.3 Obfuscated tracking

As anticipated in section 1.4, user tracking can be accomplished in 3 main ways. In case cookies are used, the server can assign the client a random identifier to save as a cookie, thus all the actions marked with the same identifier will be tied to the same visitor and can be aggregated to draw a picture of the user. This identifier can be random and anonymous, since it is saved on the user's machine and the user is "voluntarily" sending it to the server in order to be identified. As soon as a user deletes the cookies, he would become a new, unknown, visitor and be given a fresh identifier.

On the contrary, when fingerprinting is used, code has to be run on the client to fetch all the required information, which are then sent to the server. In this case, users cannot delete local data to get a new identity as there is nothing explicitly associated to the fingerprint. The only way to get a new identity is to mask the specific information used by the fingerprinting code but, given how many different characteristics can be used to identify a user, the only way to successfully hide from a specific website is to know how that site is tracking users. Developers then defend their tracking system with obfuscation to retain the possibility to closely observe users. One example is Google's reCaptcha v2, which implements a Virtual Machine (VM) running custom bytecode, and have been reverse-engineered [ReC14] to find it was tracking:

- Cookies
- Installed plugins
- Browser's user-agent
- Screen resolution
- Execution time, timezone
- Number of click/keyboard/touch actions in the `<iframe>` of the captcha
- Browser-specific functions and CSS rules
- Rendering of canvas elements.

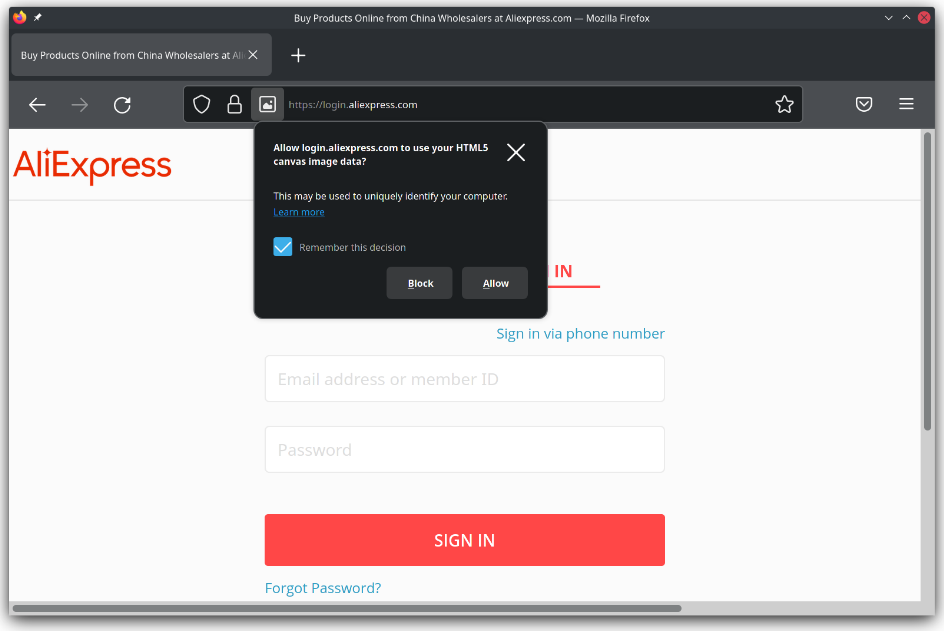
Other websites known to have complex and obfuscated tracking and anti-bot systems are:

- `https://www.cloudflare.com/`
- `https://www.akamai.com/`
- `https://datadome.co/`
- `https://supremenewyork.com`

Chapter 4

collina.js

The subject of this thesis is the process of reverse engineering an obfuscated user tracking script and the purposely written tooling will be discussed. The script in question, named `collina.js` (hereafter simply referred to as *collina*), is deployed by e-commerce websites operated by the Alibaba Group company: Alibaba, AliExpress, Taobao, Tmall and possibly others. Its existence was noticed thanks to Firefox's anti-fingerprinting features: while logging in to AliExpress, the user is notified of an ongoing canvas tracking attempt.


Buy Products Online from China Wholesalers at Aliexpress.com — Mozilla Firefox

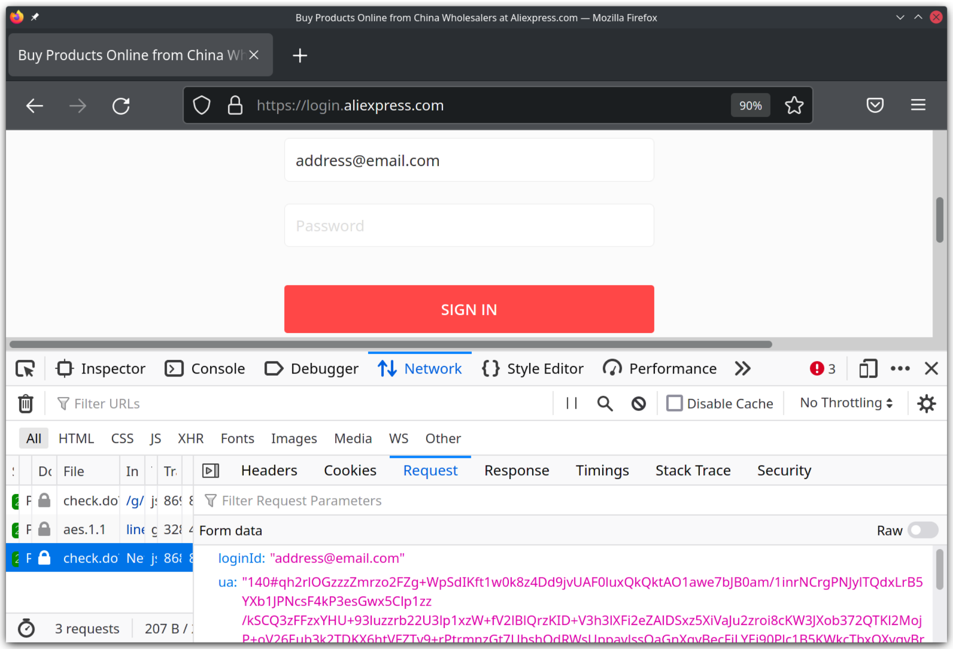
https://login.aliexpress.com

Allow login.aliexpress.com to use your HTML5 canvas image data?

This may be used to uniquely identify your computer.

Figure 4.1: Firefox canvas tracking notice

Appearing in a supposedly simple webpage, the popup in Figure 4.1 piques a hacker's curiosity, so initial analysis of the webpage was undertaken to pinpoint which action was triggering the fingerprinting and how pervasive it was. It was determined that all the fingerprinting actions were performed by a self-contained script, collina, operating independently of rest of the front end. When fingerprinting is deemed necessary, another script in the webpage requests collina to the server, executes it, then waits for collina to be completely loaded; finally the script can retrieve the machine's fingerprint via a function exposed in the HTML DOM by collina. This fingerprint will be sent in HTTP requests, alongside strictly necessary information, as a parameter named `ua`. `ua` is an opaque blob of pseudo-Base64-encoded data: it correctly decodes as binary data once stripped of a prepended version number (`140#`). The binary data is pseudo-random, hinting compression or encryption, thus black box reverse engineering was ruled out.



Form data

loginId: "address@email.com"
ua: "140#qh2rlOGzzzZmrzo2FZg+WpSdIKft1w0k8z4Dd9jvUAF0luxQkQktAO1awe7bJB0am/1inrNCrgPNJylTQdxLrB5YXb1JPNcsF4kP3esGwx5Clp1zz/kSCQ3zFFzxYHU+93luzzrb22U3lp1xzW+fV2lBlQrzKlD+V3h3lXFi2eZAlDSxz5XiVaJu2zroi8cKW3JXob372QTKl2MojP+oV26Fub3k2TDKX6htVFZTv9+rPtrmnzGt7UbshOdRWsUnpavlssOaGnXqvBecFiLYFi90Plc1B5KWkcTbxOXvqvBr

Figure 4.2: The `ua` parameter being sent along email addess check

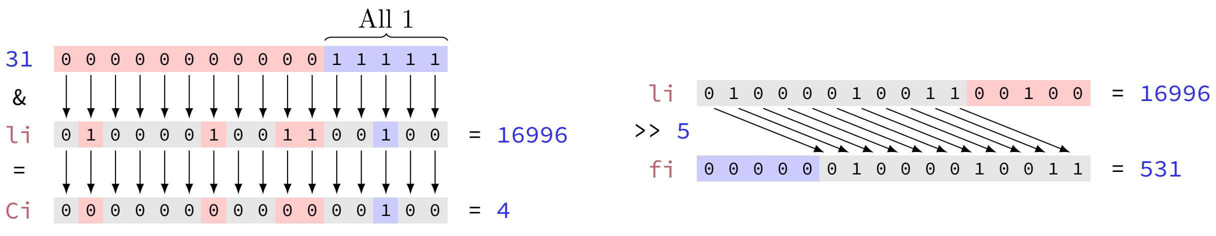
4.1 Control Flow Flattening

Given the difficulties in making sense of the `ua` parameter, white box reverse engineering becomes the only feasible approach. The original collina source, as downloaded from Alibaba servers, presents itself as a long minified one-liner (actually it is split in 2 long lines); the result of beautification can be seen in Listing 4.1. A quick analysis of this code shows it is composed of a big anonymous function, cast to a Function Expression (via the prepended `!`), containing a main for loop and multiple nested functions.

```
!function () {
  function e(r, s, d, p, v) { /*...*/ }
  function o(e, o) { /*...*/ }
  function n(e, n, t) { /*...*/ }
  function t() { /*...*/ }
6    function r(e) { /*...*/ }
7    function a() { /*...*/ }
8    function i() { /*...*/ }
9    function h(e) { /*...*/ }
10   function c(e) { /*...*/ }
11   function s(e) { /*...*/ }
12   function d(e) { /*...*/ }
13
14   for (var p = 2; void 0 !== p;) {
15     var v = 7 & p,
16       u = p >> 3,
17       g = 7 & u;
18     switch (v) {
19       case 0:
20         ! function () { y = {}, p = 3 }();
21         break;
22       case 1:
23         var l = 0, C, f, m, b = 0,
24           A = "",
25           k = "\x00\x01 ... \xfe\xff".split(""),
26           S = new RegExp("0+$"),
27           x = {};
28         b = 11, e(27), p = void 0;
29         break;
30       case 2:
31         var w = [];
32         w.unshift([]);
33         var j = "__acjs_awsc_140",
34           O = [],
35           y = window.UA_Opt;
36         p = y ? 3 : 0;
37         break;
38       case 3:
39         window.UA_Opt = y;
40         var E = window.UA_Opt;
41         p = E.loadTime ? 4 : 1;
42         break;
43       case 4:
44         var R = new Date;
45         E.loadTime = +R, p = 1
46     }
47   }
48 }();
```

Listing 4.1: Collina source beautified (function bodies omitted)

The body of the **for** at line 14 is interesting: it begins with the declaration and subsequent assignment of the variables v, u and g through a series of bitwise operations (see Figure 4.3). The value of these new variables is hence directly coupled to the value of the loop variable p: writing a single value to p corresponds with writing the 3 variables v, u and g, all at the same time. v is then used in the switch statement to decide which snippet shall be executed. Given how the loop variable controls the execution flow, it can be assimilated to the Program Counter (PC) of a CPU and every assignment to the loop variable can be considered the analogous of a C **goto** or an ASM JMP: it specifies which snippet of code shall be executed next. With this workaround it becomes possible to implement *control flow flattening* in JavaScript (JS) as well, even if the language lacks a **goto**-like statement.

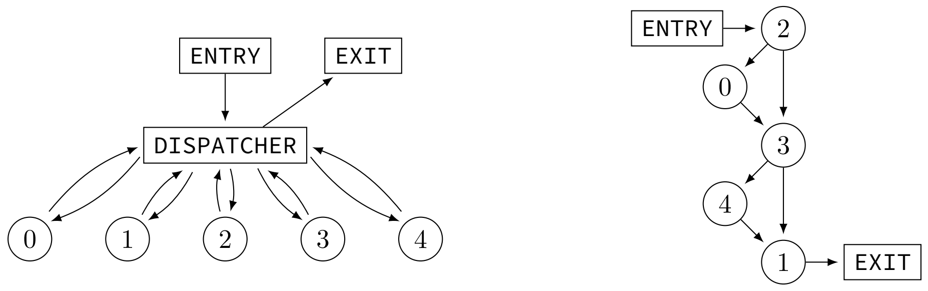

All 1
31 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1
&
li 0 1 0 0 0 0 1 0 0 1 1 0 0 1 0 0 = 16996
=
Ci 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 = 4
li 0 1 0 0 0 0 1 0 0 1 1 0 0 1 0 0 = 16996
>> 5
fi 0 0 0 0 0 0 1 0 0 0 0 1 0 0 1 1 = 531

(a) Logical AND (b) Right shift

Figure 4.3: Applying the AND operator (a) with 31 as an operand, the 5 least significant digits are isolated and then shifted out (b)

Also, all the **case**s end with an assignment to p (be it conditional or not) and a **break** statement to restart the loop. There is always exactly one assignment to p, and it is always the last statement of the **case** block. On the next iteration, the variables v, u and g will be updated with the values derived from the new p and the right fragment will be executed. The cycle continues until the stop condition **void** 0 is reached.

Given the aforementioned characteristics, the **case** bodies can be considered basic blocks as they were defined in section 2.3. This abstraction allows for further analysis of the functions thanks to a more precise Control Flow Graph (CFG): the dispatcher (the **switch**) shown in Figure 4.4a can be ignored since it's transparent to the actual control flow and, likewise, all the links from the dispatcher to the nodes and vice versa are unnecessary. Given the limited number of nodes, it is trivial to read the code in Listing 4.1, determine the connections between nodes and generate a "semantic Control Flow Graph (sCFG)" starting from state 2, since 2 is the initialization value of the **for** loop and thus the first block executed.



(a) Actual CFG (b) sCFG

Figure 4.4: Different CFGs for Listing 4.1

Analogous structures are repeated in most of the subfunctions: the bigger the function, the more nested switching. For example, in Listing 4.2, there are 5 auxiliary variables and 2 nested switch constructs (lines 8, 11). Additionally, the same snippet also highlights another obfuscation technique: nested ternaries. All the ternaries in the innermost function starting at line 13 can be translated to a series of cascading **if-else**, as per Listing 4.3.

```
1  for (var li = 16996; void 0 !== li;) {
2    var Ci = 31 & li,
3      fi = li >> 5,
4      mi = 31 & fi,
5      bi = fi >> 5,
6      Ai = 31 & bi;
7    switch (Ci) {
8      case 0:
9        ! function () {
10         switch (mi) {
11           case 0:
12             ! function () {
13               12 == Ai ? (N = Se[vo], Q = N[Z](), li = Q ? 11460 :
                 ↪  1475) : 12 > Ai ? 5 == Ai ? li = 8804 : 5 > Ai ? 2 ==
                 ↪  Ai ? (W = $ % 128, ie = [], se = ie, li = 16390) :
                 ↪  (Dn.push(0), li = 11522) : (L = mo, li = 24641) : (Re
                 ↪  = 1 | y[0], li = 16034) /*...*/
14             }();
15             break;
16           case 1: /*...*/
17         }
18       }();
19       break;
20     case 1: /*...*/
21   }
22 }
```

Listing 4.2: Excerpt from function e

```
if (Ai == 12) {
  N = Se[vo], Q = N[Z](), li = Q ? 11460 : 1475;
} else if (12 > Ai) {
  if (Ai == 5) {
    li = 8804;
  }
  else if (5 > Ai) {
    if (Ai == 2) {
      W = $ % 128, ie = [], se = ie, li = 16390;
    } else {
      Dn.push(0);
      li = 11522;
    }
  } else {
    L = mo, li = 24641;
  }
} else {
  Re = 1 | y[0], li = 16034;
}
```

Listing 4.3: Unrolled ternaries from nested function at line 13 of Listing 4.2

4.2 Additional Obfuscation Techniques

In addition to control flow flattening, other obfuscation techniques are present in collina:

- **Function-wide variables reuse**
  All the required variables, as many as 350, are declared at the beginning of a function and there will be little to no declarations in the body: every variable is used multiple times to store different data, leading to confusing snippets, such as:

```
me = pe;
pe = ge;
Jn[3] = me ^ re;
Jn[11] = pe ^ re;
Vn.push(Jn);
me = uo[56];
pe = me.length;
me = pe > 8192;
```

- **Strings obfuscation**
  Strings are split and, at times, reversed to prevent a simple search in the file. Additionally, every char in a string can be incremented/decremented/xor-ed with a predefined value and decoded at runtime

```
qo = "c";
En = "t";
qo += "on";
En += "x";
En += "et";
qo += "s";
```

```
fe = "";
Ln = 0;
be =
  ↪  "\u02D5\u02D3\u02E2";
/*...*/
while (Ln < be.length)
  ↪  {
```

```
En += "no";
En += "c";
qo += "o";
En = En.split("").re-
↪  verse().join("");
qo += "le";
/* qo = "console" */
/* En = "context" */
```

```
  zo =
  ↪  be.charCodeAt(Ln)
  ↪  - 622;
  fe += String.from-
  ↪  CharCode(zo);
  Ln++;
  continue;
}
/* fe = "get" */
```

- **Virtual stack**
  An alternative string decoding system relies on a global stack to emulate assembly-level function calls; it will be discussed in subsection 5.5.2.

Note: for clarity's sake, the above snippets were taken from an already partially deobfuscated collina.

Chapter 5

bulldozer.js

The obfuscation techniques implemented in collina have been described in chapter 4; this chapter will introduce a tool written in JavaScript (JS) to defeat most of them.

The name bulldozer was picked as a joke on *collina*: in Italian, collina means hill and, although it is unclear why an Italian word was picked by developers at a Chinese company, it seemed evident that many techniques were *piled up* to make the source incomprehensible. So, the name *bulldozer* was deemed fit for a tool that would *demolish* this mass of obfuscation.

The semantic Control Flow Graph (sCFG) in Figure 4.4b had been manually generated, given how simple the loop in Listing 4.1 was. Other, more complex functions, such as `e`, contain as many as 4286 nodes and 5937 edges. While understanding the behavior of the code in Listing 4.1 might have been trivial, the sheer amount of nodes in `e` made it imperative to automate both the generation of the sCFG and every subsequent manipulation of the graph.

5.1 Source Code Parsing

The first step in analysis is lifting the source code to an easily traversable structure such as the Abstract Syntax Tree (AST) presented in section 2.2. As it was previously pointed out, using a AST allows for programmatic analysis of source code without resorting to fragile static rules and regular expressions. The tree structure, which might have looked like a burdensome overhead for simple code such as one in Figure 2.3, has proven essential for all the upcoming processes. The AST is generated by the `shift-parser` library by Shape Security [Shac], which accepts in input a valid JS source file and returns its AST.

As said in chapter 4, collina's source contains a main function with multiple subfunctions, some of which even have other functions inside them. The previously generated AST is then analyzed to find suitable functions; this matching process is based on a simple pattern: if a function body, be it at any depth level, has a **switch** statement inside a **for** loop, then it should be analyzed further.

Every eligible function found in the above step will have a structure similar to Listing 4.1 and Listing 4.2. The two aforementioned functions have a similar structure, but it is impossible to establish a way to analyze them *a priori*: they have a different amount of nested **switch** constructs and the variable names are different. For this reason, every function must be parsed to retrieve:

- `next_state_identifier`: the variable where next state is written, the "Program Counter (PC)".
- `init_state`: the first state that will be executed, PC initialization value.
- `stop_condition`: the stop condition, as an AST-compatible object.
- `bitwise_identifiers`: the names of variables on which switching will be performed.
- `next_state_eval`: a string with placeholders, used to `eval` the next `bitwise_identifiers` values given a specific `next_state_identifier`.

```
{
  next_state_identifier: "li",
  init_state: 16996,
  stop_condition: {
    type: "UnaryExpression",
    operator: "void",
    operand: {
      type: "LiteralNumericExpression",
      value: 0
    }
  },
  bitwise_identifiers: ["Ci", "fi", "mi", "bi", "Ai"],
  next_state_eval: "var li = 0xBAAAAAAD;var Ci = 31 & li, fi =
  ↪  li >> 5, mi = 31 & fi, bi = fi >> 5, Ai = 31 & bi;\n[Ci,
  ↪  fi, mi, bi, Ai];"
}
```

Listing 5.1: Analysis of function `e` from Listing 4.2

The placeholder value `0xBAAAAAAD` in `next_state_eval` will be replaced by the appropriate value of `li`, then the string will be passed to JS' `eval` function. The return value will be an array of 5 numbers with the values of the variables `Ci`, `fi`, `mi`, `bi`, `Ai` after the required calculation have been performed. Given `16996` as an input, the result would be `[4, 531, 19, 16, 16]`. Note this is the only part of the tool where a `eval()` is used, and it is performed on a relatively safe string, since `next_state_eval` is generated by bulldozer itself.

5.2 CFG Generation

This step is split in two phases. At first, the `switch-case` structures are converted to a map-of-maps hierarchical structure, which is faster to traverse, then every assignment to `next_state_identifier` is analyzed, adding an edge and, if necessary, the successor node in the graph.

5.2.1 AST to Map-of-Maps

Thanks to the `shift-reducer` library [Shad], it is possible to implement simple monoidal reducers of complex tree structures. Like the name suggests, a reducer applies a function to every element of a complex structure (be it an array or, in this case, a tree), returning a single output value. The monoidal class `MapMonad` was defined to hold a `Map`; it is then passed to the `SwitchReducer` class, which extends the library-provided `MonoidalReducer`, implementing a specific way of reducing nested switch-case structures. The result of the reduction process is a `MapMonad` object, which can be converted to a `Map` via the recursive `toMap()` method.

In the final Map, the **`case`** test value is associated to the body of the **`case`** itself.

```
switch (n) {
  case 0:
    ! function () {
      1 == a ? (i(), o = 8)
      ↪  : 1 > a ? (h =
      ↪  r(e), o = 2) : o
      ↪  = void 0
    }();
    break;
  case 1:
    var h = 1 !== m;
    o = h ? 0 : 2;
    break;
  case 2:
    var c = h;
    o = c ? 4 : 8
}
```

Listing 5.2: Example switch-case

```
{
  0: {
    type: "ExpressionState-
     ↪  ment",
    expression: {
      type: "UnaryExpres-
       ↪  sion",
      operator: "!",
      operand: /*...*/
    }
  },
  1: {[
    {type: "VariableDeclara-
     ↪  tionStatement",
    declaration: /*...*/},
    /*...*/
  ]},
  2: /*...*/
}
```

Listing 5.3: Map relative to Listing 5.2

If any of the **`case`**s in Listing 5.2 had a nested switch statement, then the corresponding entry in Listing 5.3 would have a submap for the given **`case`**.

The preliminary reduction process described above, which takes only fractions of a second, makes the Control Flow Graph (CFG) generation order of magnitude faster than querying the whole AST structure for every required state.

5.2.2 Basic Blocks Extraction

This process is static counterpart to what would happen during execution: the code goes further and further in depth until it finds the appropriate snippet and, instead of executing it, the matching AST chunk is returned.

If there are as many **`switch`** structures as there are `bitwise_identifiers`, it is straightforward to extract the appropriate basic block: the maps are progressively selected until a non-map element is reached and returned.

However, due to the different kinds of obfuscation in place, more advanced processing is required: not all the functions have solely **`switch`** structures. As anticipated in chapter 4, in some cases "switching" is performed via a chain of **`if`** clauses or ternaries. To circumvent this problem a specifically tailored static evaluation system, based on pattern matching, was put in place. The tree is traversed in depth and, when an `IfStatement` or a `ConditionalExpression` (a ternary) is found, the test expression is statically evaluated against the previously calculated values of `bitwise_identifiers`. If the expression evaluates to **`true`**, then the `consequent` branch is recursively fed to the evaluator, otherwise the evaluation will continue with the `alternate` branch. At the same time, useless code such as Listing 5.4 is filtered away.

```
Ai > 0 && (To++, li = 16128);
```

Listing 5.4: Useless code: if `Ai = 1`, then the right side only can be returned

The implementation of the static evaluation feature is greatly simplified by the choice of writing bulldozer in JS, the same language of the target code. While, technically, there were other possibilities, it allows to gloss over subtle implementation details, such as operator quirks with different data types, which would not be straightforward to emulate in a different language.

It is worth noting that the extraction process does not require any dynamic `eval()` or symbolic execution tool: everything is performed statically. This is possible because of the lack of opaque predicates and by the overall simplicity of test conditions.

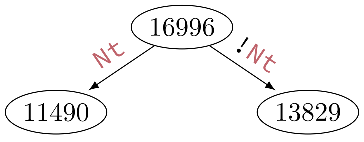
5.2.3 Code Traversal

Starting from the initial state, available as `init_state`, basic blocks are extracted from the AST and the CFG is generated with the following process:

1. A basic block is extracted from the AST as described in subsection 5.2.2
2. Basic block's code is analyzed by another reducer to recover all the jumps, which are, in fact, assignments to `next_state_identifier`: there should be at most 1 assignment, possibly with two destinations in order to perform conditional branching
3. If there is more than one destination, both branching conditions are recovered, where possible, by generating the inverse condition (`gi > 5` $\implies$ `gi <= 5`), otherwise negating the available one (`Qo.state[10]` $\implies$ `!Qo.state[10]`)
4. Jumps are removed from the basic block's statements list
5. The basic block is added as a node to the CFG
6. As many edges as jump destinations are added to the CFG; edges themselves hold the jump conditions as attributes
7. Edges are followed recursively, repeating the process from step 1 with the newly identified jump targets.

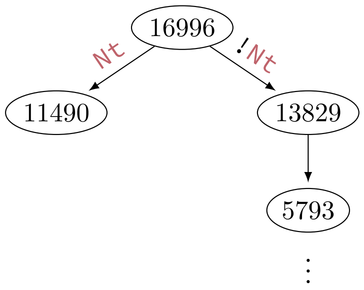
The above process stops when a jump to `stop_condition` is identified, or if the jump target is already in the graph.

```
ba = window;
Nt = r;
ur = void 0;
Jo = void 0;
ei = 0;
li = Nt ? 11490 : 13829;
```



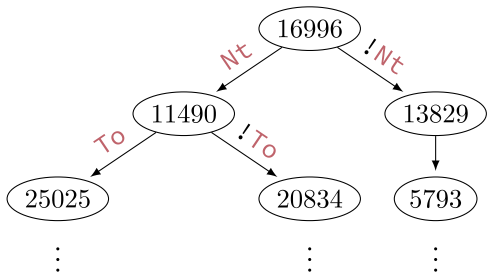
Listing 5.5: `li = 16996`

```
Jo = void 0;
li = 5793;
```



Listing 5.6: `li = 13829`

```
Qn = "ob";
Qn += "jec";
Qn += "t";
To = typeof Nt !== Qn;
li = To ? 25025 : 20834;
```

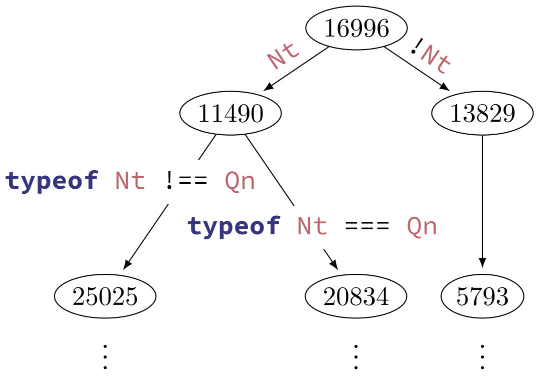


Listing 5.7: `li = 11490`

5.2.4 Branching Condition Recovery

In some cases, such as Listing 5.7, branching is performed basing on the value of an identifier, in this case `To`. Graph's readability would be greatly increased if the edges reported the actual condition, thus becoming

```
Qn = "ob";
Qn += "jec";
Qn += "t";
To = typeof Nt !== Qn;
li = To ? 25025 : 20834;
```



Listing 5.8: Listing 5.7 with improved readability

When the identifier is assigned in the same basic block, such as `To` in Listing 5.7, a function takes care of refining the branching condition to the indirectly referenced expression. Nevertheless, the original assignment is not removed from the basic block, as it could be used multiple times before the branching takes place and discarding it would result in broken code.

5.3 Region Identification

The CFG alone could aid human analysis of the functions, graphically highlighting loop structures and branching, but, given the amount of nodes, this is still an impractical task. On the other hand, a direct transposition of the CFG to JS code would yield unreadable code and, most importantly, would be simply impossible due to the lack of **`goto`**-like structures. At the moment it is not known if a branch is an **`if-then`**, an **`if-else`** or a **`while`** loop, or even where the branching is going to close. For example, the graph in Figure 5.1 branches twice:

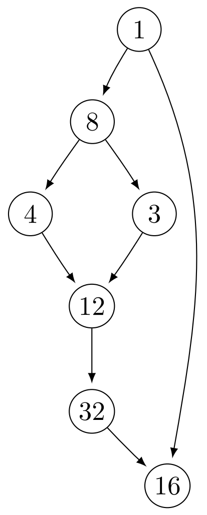


Figure 5.1: Branching

- The branch starting on 1 either follows the path from 8 to 32 and then finally reaches 16, or goes straight to 16. This is analogous to an **`if-then`** in a high-level language: if a condition is met, then the branch starting on 8 will be executed, otherwise the code will continue directly to 16, which is reached in any case.
- The branch starting on 8 goes to 3 or 4, then each of these nodes will end in 12. There is no direct connection between 8 and 12, but execution will nonetheless end on 12. The set of nodes represents a **`if-else`** branch: if a condition is met execution continues (supposedly) to 4, otherwise 3 is executed; then finally both branches merge back at 12.

Similarly, for loops, it is mandatory to identify all the nodes composing the loop. Loops also have specific characteristics such as *backedges*: edges going from a node of the loop back to the head (colored red in Figure 5.2). There can be multiple backedges, and they must all be identified, since they represent **`continue`** statements in the original code. Similarly, edges leaving loop are **`break`**s.

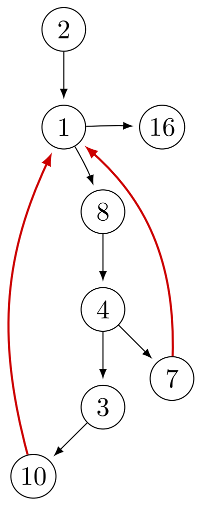


Figure 5.2: Loops

All the previously raised points require an in-depth analysis of the CFG, implementing the techniques commonly found in decompilers. Incidentally, this explains why the word “decompiler” was used in the title of the document to describe this tool.

The analysis relies on the recognition of regions: connected sets of nodes having homogeneous characteristics. This part of the work was heavily inspired by the binary analysis framework angr [Sho+16], which is written in python and features, among other things, a binary decompiler for different architectures. The graph structure is managed through JSNetworkX [Kli12], a partial JS port of the NetworkX Python library for studying graphs and networks. It was necessary to port some unimplemented functions of NetworkX to JS, such as depth-first traversal and dominators computation.

Regions are objects with specific properties:

- **Graph** The graph of the nodes inside the region
- **Successors** A list of successors to the region
- **Graph with successors** The graph of the nodes inside the region, plus their successors with edges connecting them to the nodes inside the graph
- **Head** The head of the region, where its graph is entered
- **Cyclic** A boolean value used to store whether the region is cyclic or not.

5.3.1 Cyclic Regions

A loop can be defined as a strongly connected component of the CFG whose main components are

- **Head**: The target of entry edge
- **Entry edge**: An edge having tail not in the loop and head in the loop (pointing to the head)
- **Exit edge**: An edge having tail in the loop and head outside the loop
- **Backedge**: An edge having both tail and head in the loop, pointing to the loop's head.

A **natural loop** is a loop with only a single head; two natural loops, if they have a different head, can either be disjoint or nested within each other.

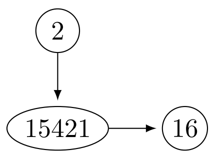


Figure 5.3: Loop in Figure 5.2, replaced with a region

The presence of loops in a graph makes nontrivial to traverse it, since every routine analyzing the topology must take care of not looping back; furthermore, an analysis started on a node inside a loop will have no knowledge of the loop's head and might not notice at all it is looping back. Due to this reason, the first manipulation of the graph consists of removing loops. All the nodes and the edges constituting the loop are replaced with a *cyclic region* which, externally, appears as a single atomic node.

Loops are identified performing a depth-first postorder traversal (see subsection 2.1.1), taking care of not looping back: when the target of an edge is an already visited node, a backedge is found. Every loop head is identified by (at least) one backedge, thus the list of backedges gives a rough idea about the loops in the CFG.

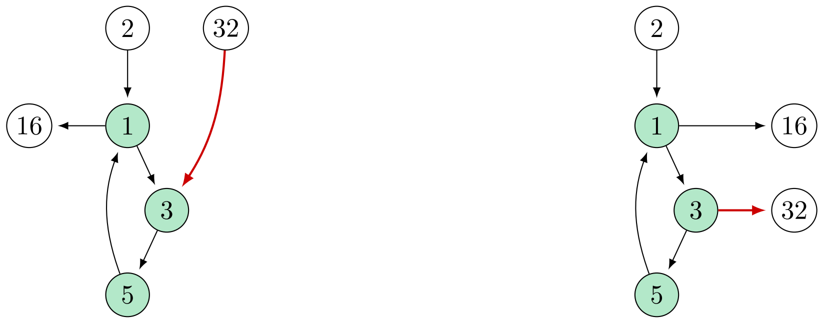
The first step to identify a loop's nodes consists in searching all the backedges returning to a given head; the starting nodes of these backedges are considered *frontier nodes* (in Figure 5.2 they would be 7 and 10). Then, a breadth-first traversal of the graph is started on the head: if a path exists from the current node to any of the frontier nodes, then the node is part of the loop. This process continues until the frontier itself is reached. If the set of identified nodes contains any header other than the current target, then the region is temporarily discarded as there is a nested loop which must be replaced with a region first. Once a region has been identified, entry and exit nodes are selected. The loop is then refined to account for some edge cases. For example if a supposed exit node is, in fact, dominated by the loop's head, it can be included in the loop itself and one of its successors will be a new exit node.

Finally, all the loop's nodes are removed from the graph and replaced with a single region node, containing a graph of the loop along with additional information such as the loop's head.

Abnormal Entries or Exits

A perfectly legal loop might have multiple entries, all pointing to the head, which means multiple code paths are ending on that loop; multiple exits are acceptable as well, as long as they point to the same exit node, since they can be represented as **`break`** statements. This kind of structures can be directly translated to JS constructs.

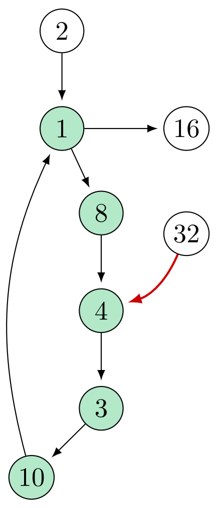
On the contrary, if a region has multiple entry/exit edges pointing to different nodes, then those entries/exits are considered *abnormal*. Two examples can be seen in Figure 5.4, where 32 is an abnormal predecessor in Figure 5.4a and an abnormal successor in Figure 5.4b. While most binary decompilers would solve this situation injecting **`goto`** statements, being JS the target language of this tool, further refinement is required. This problem is solved via a label system, adding new nodes to both the region and the global graph. Albeit this approach has been developed independently, a mostly identical implementation was described in [Yak+15], which, however, was brought to my attention only at the end of this project.



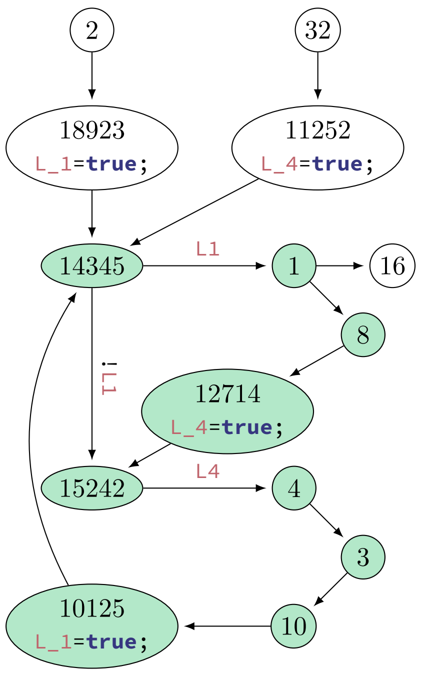
(b) Multiple entries (a) Multiple exits

Figure 5.4: Cyclic regions (green), abnormal entries and exits (red)

Entry chains When a loop can be entered on different nodes, it means that one part of the loop is optional. For example in Figure 5.5a, when the loop is entered from node 32, nodes 1 and 8 are skipped on first iteration but will be executed afterwards. The solution to this issue was inherent in the optionality of the nodes: all the entries can be redirected to a chain of **`if-then`** which wraps the optional nodes.



(a) Multiple entries



(b) Graph (a) after refinement

Figure 5.5: Cyclic region (green) with multiple entries

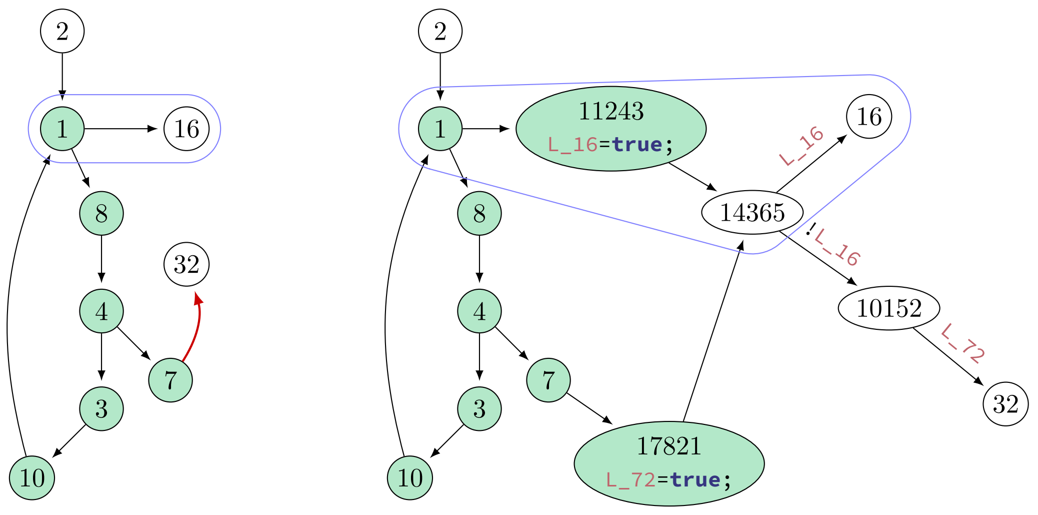
With regard to Figure 5.5, an empty stack of entry nodes is created and then, for each target of an entry:

1. A new dispatcher node (14345) is added inside the loop and pushed in the entry nodes stack. This node is empty.
2. The dispatcher (14345) is connected to the original entry target (1) with an edge which can be followed only if a label (`L_1`) is **`true`**.
3. For each predecessor `p` (2 and 10) of the target `t` (1):
   (a) A new node is added to the graph: if `p` is outside the loop, the new node (18923) will not be included either, otherwise it (10125) will be part of the loop. This node holds a single statement to assign a label (`L_1`) to **`true`**.
   (b) If `p` is outside the loop, then the new node (18923) is connected to the first node in the entry stack (14345), otherwise it is connected to the dispatcher created in step 1 (still 14345, in this case).
   (c) `p` (2) is connected to the label setter node (18923), the original edge (2→1) conditionality is preserved.
   (d) The original edge (2→1) is deleted.
4. If the stack contains at least another dispatcher, the current dispatcher is connected to the previous one (thus 15242 will be connected to 14345).

When the loop in Figure 5.5 is entered from node 32, the label `L_1` will be unset and then the first half will be skipped, going straight to node 15242. There, given the label `L_4` is set, nodes 4-3-10-101125 will be executed. On 101125 the label `L_1` will be assigned to **`true`**, enabling execution of nodes 1-8 on the next iteration.

Due to how each dispatcher node is connected to an entry and to the next dispatcher, the resulting graph contains standard **`if-then`** structures; this allows for 1:1 translation of multiple entries loops into pure JS.

Exit chains The links from nodes in the loop to the exits are made *indirect* and reachable only through a chain of nodes; the outcome is a region with only one successor and the original logic is preserved. The two paths circled blue in Figure 5.4a and Figure 5.4b are equivalent.



(a) Multiple exits (b) Graph (a) after refinement

Figure 5.6: Cyclic region (green) with multiple successors

With regard to Figure 5.6, an empty queue of exit nodes is created and then, for each exit edge:

1. The exit edge (1→16) is deleted from the graph.
2. A new node (11243) is added inside the loop; the node holds a single statement to assign a label (`L_16`) to **`true`**.
3. The start of the original exit edge (1) is connected to the label node (11243) with a new edge. The original edge's (1→16) conditionality is retrieved and preserved here.
4. A new empty exit dispatcher node (14365) is added outside the loop.
5. The dispatcher node (14365) is connected to the destination of the original exit (16) via an edge which can be followed only if the previously set label (`L16`) is **`true`**.
6. If the queue contains at least another node, the dispatcher (14365) is connected to the last node in the exit nodes queue

7. The dispatcher node (14365) is appended at the end of the exit nodes queue.
8. The label node (11243) is connected to the first dispatcher in the exit nodes queue.

Due to the fact all the label nodes have been connected to the first node in the queue, the region now has only one successor, but all the original successor are still reachable. This CFG structure can be easily seen as a standard **`while`** loop followed by a series of nested **`if-else`** constructs.

This approach, different from the one used in angr, was chosen as it was the safest way to guarantee data integrity. In fact, chaining the reaching conditions and using them as entry/exit edges conditions, if not carefully performed, could produce incorrect output. For example, the edge 4→7 might depend on the value of `a` and, in basic block 7, `a` might be overwritten to **`false`**. If the check for the value of `a` were performed on a successor of the whole region, `a` would then be considered **`false`** and the basic block 32 would never be reached. Using labels solves this problem nicely, since they are managed by bulldozer and totally invisible to the code being reverse-engineered.

5.3.2 Acyclic Regions

Once the detection and extraction of cyclic region is complete, the CFG can be assumed to be *acyclic*. The remaining global graph is added to an acyclic region with the start node being the head of the new region.

There are now fewer nodes in the graph, compared to the original CFG, since some parts of it have been removed and replaced with complex nodes. These new complex nodes are the cyclic regions, which contain the subgraphs removed from the global graph. The region analysis process continues by searching for acyclic regions, both in the global graph and in all the graphs contained in the new cyclic regions; cyclic regions can be considered atomic nodes in the global graph, but their content has to be structured as well.

Depending on the region type, a different graph is used for the upcoming analysis:

- Cyclic region: graph with successors. This is necessary since, in this case, the graph does not represent some important aspects of the region. As an example, assuming that in Figure 5.6a both nodes 1 and 7 exited on node 16 (as if there were no abnormal exits), considering only the nodes in the loop (green ones), both the exit edges would not be considered. This means that node 1 would have only one successor and 7 no successors at all.
- Acyclic region: graph

Dominators, postdominators and dominance frontier are calculated for all the nodes in the graph, starting from the region's head. If the graph has multiple end nodes, they are all connected to a single dummy end node to allow postdominator calculation. After these preliminary information have been gathered, for every node in the graph the following process is performed:

1. The node is considered the start of a potential region and its immediate postdominator is picked as a candidate end node.
2. It's checked whether the potential region is single-entry and single-exit.
3. The nodes are extracted from the graph and replaced with a new acyclic region.
4. The loop is restarted from step 1 as long as the graph can be modified further.

With the above process, visualized in Figure 5.7, nested regions are created progressively.

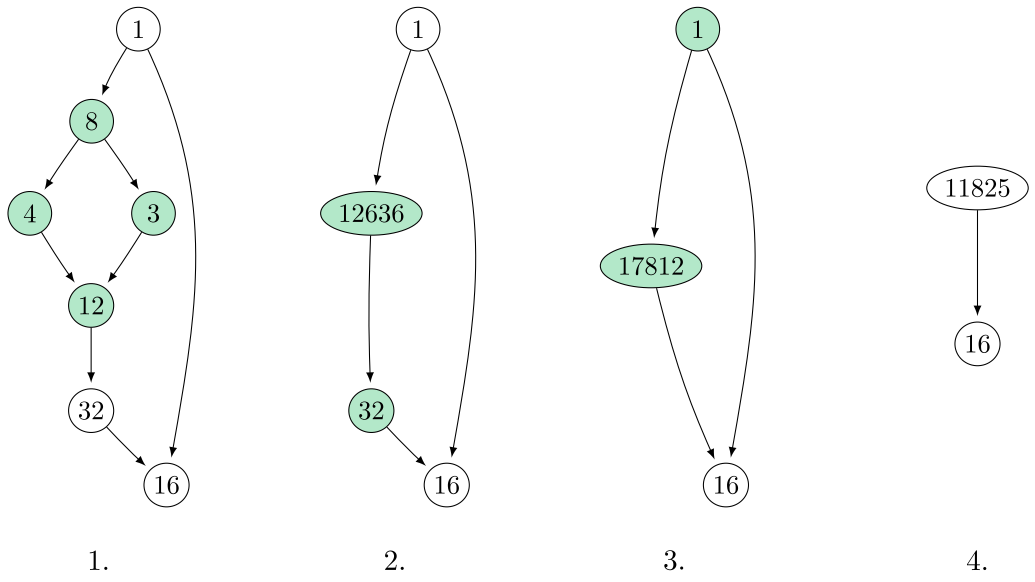
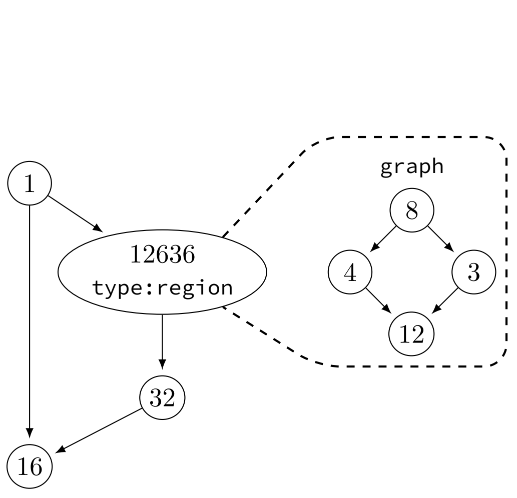


Figure 5.7: Successive steps in acyclic region analysis, green nodes will be encapsulated

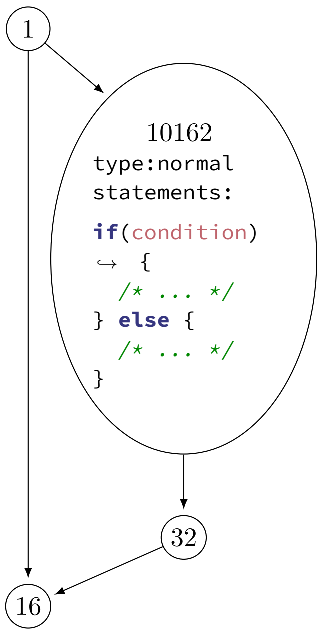
5.4 Code Structuring

Once regions have been recognized and isolated, the CFG of the original, pre-obfuscation, function has been (mostly) recovered and is finally possible to regenerate JS code from it. This process is easily feasible only thanks to the region properties enforced in section 5.3.

Due to the hierarchical structure of the regions created earlier, it is necessary to recursively analyze all the regions, diving in every child region, structuring it, and replacing it with a node bearing the corresponding source code. The process starts with the topmost region, the one containing what's left of the original CFG once all the regions have been generated. This macro-region's graph is traversed and, for every one of its nodes, if the node is a region, it is pushed in a stack. This process continues, adding new regions to the stack, until a region with no subregions is found: then the actual code structuring begins. Once the region has been converted to its AST representation, it is popped from the stack and replaced in the graph with a simple node. Statements of this new node are the output of code structuring and can be considered the translation into source code of the region's CFG. At this point, the parent of the structured region will not have any subregion, thus it can be structured as well and the code "bubbles up" backwards to the topmost region.



(a) Graph from Figure 5.7.2



```
if(condition)
↪ {
  /* ... */
} else {
  /* ... */
}
```

(b) Region 12636 replaced with code

Figure 5.8: After structuring

The rest of this section will introduce the actual way in which the CFG is transformed to code. Once again, this part of the project is powered by a member of the shift-AST family: `shift-codegen` [Shab]; this library accepts a valid shift-ast tree and outputs JS code.

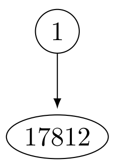
5.4.1 Acyclic Regions

Acyclic region structuring is straightforward. The region's graph is traversed depth-first, starting at the head of the region, and the statements carried by every node are appended to a flat list of AST nodes, all at the same depth level. The traversal is part of the code generation itself since, depending on a node's outdegree ($\deg^+$):

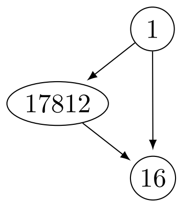
- $\deg^+(v) = 0$: No successors to this node, the code generation stops here.
- $\deg^+(v) = 1$: The only outgoing edge is followed, next node's statements will be appended right after the current node ones.
- $\deg^+(v) = 2$: A branch is starting here, the successors must be wrapped in an **`if-then`** or **`if-else`** construct.

Branch Analysis

Thanks to the properties enforced by the previous acyclic region analysis, strong assumptions can be made to ease the detection of branch type.



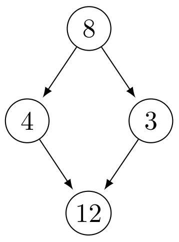
(a) Graph



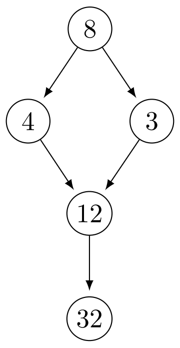
(b) Graph with successors

Figure 5.9: Region 11825 from Figure 5.7

`if-then` can be separated from **`if-else`** considering the `graph_with_successors`. As can be seen in Figure 5.9b, node 1 has an outdegree of 2, this means it is a branching node. Given how one target of the branch is in graph (17812) and the other is only present in `graph_with_successors` (16), it can be assumed to be an **`if-then`**. To reach this conclusion it suffices to think about any programming language or even pseudocode: when the branching condition is met, the body of the **`if`** is executed and, once that is over, the execution continues to what follows the conditional. If the branching condition is not met, then the code right after the **`if`** will be immediately executed. This behavior is exactly what is being represented by the graphs in Figure 5.9.



(a) Graph



(b) Graph with successors

Figure 5.10: Region 12636 from Figure 5.7

Contrarily, as can be seen in Figure 5.10a, both the successors of node 8 in region 12636 are part of the graph, which leads to an **`if-else`** translation in JS. Again, this is exactly what happens when an **`if-else`** condition is reached during code execution: either the **`if`** body is executed or the **`else`** body, then both will execute what follows the conditional.

Finally, the branching condition is recovered from the CFG edges as well, and it is used to populate the **`if`** test condition.

5.4.2 Cyclic Regions

Unlike acyclic regions, cyclic regions require more care due to exits and backedges, which correspond to **`break`** and **`continue`** statements.

For every backedge, a new node is added to the graph, containing a single **`continue`** statement. Each source node of the backedge is connected to one of these new nodes, and the original backedge is removed from the graph. This process effectively changes the target of backedges to make them point to new, non-looping nodes, which will bear the same meaning once JS code is generated. An artifact of this process is the presence of a **`continue`** statement at the bottom of every loop, since the function rewriting backedges is not aware of which node of the CFG will be placed last in the loop. This issue could easily be fixed performing a breadth-first search and ignoring the last backedge, or with a pass over the final AST to remove all the **`continue`** laid at the end of loops. Nonetheless, these refinements were deemed useless as these spurious statements can be manually removed or simply ignored by the reader.

An analogous process is performed on exits, which get redirected to new nodes carrying **`break`** statements; exit edges are therefore deleted from the `graph_with_successors`. The new **`break`** nodes are not only added to `graph_with_successors`, but to the graph as well, since they are an actual part of the loop albeit the exit nodes were not.

The loop condition is recovered by looking at the edges leaving the region's head: if there are 2 edges leaving the head and one of them points to a successor in `graph_with_suc-cessors`, then the edge pointing to a node inside the region holds the condition to keep looping. If there is only one outedge, then the loop might be a **`do-while`** or some sort of nonstandard loop which can be obtained only through **`goto`** statements. In this case a default **`while(true)`** loop is emitted and the previously generated **`break`**s will still allow to leave it. Future improvements of this analysis might involve recognizing **`do-while`** and **`for`** loops.

When these steps have been performed, a cyclic region can be simply traversed to reconstruct its code, as if it was a standard acyclic region, and the resulting AST portion can be wrapped by a AST's `WhileStatement` block.

5.5 Partial Evaluation

The result of the above CFG reconstruction is human-readable code, where the loops are represented as such, branching is performed via **`if-else`** structures, and blocks of code boundaries are explicit. Although the improvement in readability over the original version of collina is impressive, reverse engineering the 34 259 lines-long output would still pose a non-indifferent burden on the analyst. The obfuscation techniques presented in section 4.2, still in place and untouched by the previous processes, slow down the interpretation of the codebase. For this reason, the code is subjected to one final pass of deobfuscation, as it is commonly intended. To perform this task a minimal partial evaluation analysis routine was implemented which, albeit not generic enough to work with any ECMAScript (ES) piece of code, was sufficient to defeat many of the remaining obfuscation constructs.

5.5.1 Variable Descrambling

As shown in section 4.2, strings and literals are split and then consolidated at runtime. Usually, the assignments and modifications of the variable take place in different regions of the code, in order to baffle the reader even further, leading to hours of painstaking scrolling through the code in order to mentally reconstruct the complete string.

Types of access to a variable can be categorized in 3 main classes:

- Assignment: The variable is initialized or overwritten, its previous content is irrelevant and discarded.
- Read: The current content of the variable is read to perform operations on it or basing on its value.
- Write: The current content of the variable is modified but the final result depends on what was written in the variable before (e.g. the `+=` operator in C).

The value of a variable can be manipulated by the obfuscator lots of times, but must reach its final value by the time it is read. Because of this, if a value is assigned twice before being read, the first value can be safely discarded, and if a value is written multiple times, only the final value before a read is of interest. This means that all the writes can be collapsed in a single assignment to the final value, right before the first read.

Access Reducer

To analyze which kind of accesses are present in a complex statement, the preferred way was implementing another monoidal reducer for `shift-reducer` [Shad], and feeding it with the program's statements one by one. Doing so, complex structures are easily analyzed, all without resorting to dangerous regular expressions or string matching. For each statement, the reducer returns an object with 3 attributes: `reads` (Set object), `writes` (Map), and `assignments` (Map). An example of the analysis of the simple statement `res = x + a++;` can be seen in Listing 5.9.

```
{
  "reads": ["x", "a"],
  "writes": {
    "a": {
      "type": "UpdateExpression",
      "isPrefix": false,
      "operator": "++",
      "operand": {"type": "AssignmentTargetIdentifier", "name":
       ↪  "a"}
    }
  },
  "assignments": {
    "res": {
      "type": "BinaryExpression",
      "left": {"type":"IdentifierExpression","name":"x"},
      "operator": "+",
      "right": {"type": "UpdateExpression", "isPrefix": false,
       ↪  "operator": "++", "operand": {"type":
       ↪  "AssignmentTargetIdentifier", "name": "a"}}
    }
  }
}
```

Listing 5.9: Parsing of variable accesses in `res = x + a++;`

The `reads` property in Listing 5.9 is a simple set of the variables being read in the statement. The `writes` map associates an identifier to the operation it is being modified with, for example `a` is written via the `++` operator; same goes for `assignments`. The whole AST section is reported in both writes and assignments to enable the virtual management of variables.

Virtual Management of Variables The actual deobfuscation is performed via a variables map in which identifiers are associated to their current values, similarly to what happens in the `context` object allocated on the heap by the V8 JS engine [Goo19]. Along with the current value, an additional boolean value tells whether the variable has already been emitted in the output code in its actual form or if it needs to be printed.

Once the accesses are retrieved, these steps are executed:

- `reads`
  Is the variable being read in the variables map?
  - Yes:
    1. Has its current value been printed already?
       - Yes: Do nothing with the variable
       - No:
         (a) Append new statement to return list with this variable being assigned to the value taken from the map
         (b) Mark variable as printed in the map
    2. Append current statement to the return list
  - No: Append current statement to the return list
- `assignments`
  Is the value being assigned to this variable a literal (the only safe values to work on)?
  - Yes: Add/overwrite variable in variables map, mark as not printed
  - No: Append current statement to the return list

- `writes`
  Is the variable being written in the variables map (which means it was assigned to a literal in the first place)?
  - Yes: Is the value being written to it a literal as well?
    - Yes: Update value in map, mark as not printed
    - No:
      1. Append new statement to return list with this variable being assigned to the old value from the map
      2. Delete variable from the variables map
      3. Append current statement to the return list
  - No: Append current statement to the return list

A positive side effect of this process is that a variable will be printed just before its first usage, which further helps in going through tens of thousands of lines of code.

Another advantage of the reducer is the possibility to analyze complex and irreducible blocks, such as function declarations. In JS functions can be declared inside other functions and directly called (function expressions), which means that complex block statements can appear in the middle of another block of code. Furthermore, nested functions can access and modify surrounding variables, being in the same scope. When this happens, all the variables read inside the function's body must be printed just before the function is declared, so that they'll be accessible with their updated values from the body. So, when a nested function is met, it will be analyzed as a single statement, applying the above process; then a new partial evaluation process is started to analyze the body of the function, effectively mimicking a new scope, since the variables declared inside the nested function can be ignored by the code outside it.

5.5.2 Static Replacement Rules

The partial evaluation algorithm described above is fairly simple and does not take loops in account, they are simply considered complex structures and variables accessed in the body are printed just before the loop itself. This means that all the string-decoding loops remained in the output code, still leaving incomprehensible strings in the program. Given all these loops shared a static common structure, except for some specific details, a static matching pattern was put in place; when this pattern is matched, parameters are retrieved and the string is deobfuscated in place. In case of Listing 5.10, the parameters would be: `255`, the `&` operator, and the accumulator variable name `et`; given these and the input string, `et` can be replaced with its final value and the loop deleted.

```
while (Ue < G.length) {
  ze = G.charCodeAt(Ue);
  Go = 255 & ze;
  et.push(Go);
  Ue++;
  continue;
}
```

Listing 5.10: String decoding loop

Also, as mentioned in section 4.2, a stack-based string obfuscation technique was found in collina. A global array mimics the behavior of a standard C compiler stack, with its own calling convention. This function accepts 2 mandatory and multiple other optional parameters: the first argument (topmost in the stack) corresponds to which magic string should be used as a key, then comes the number of optional parameters and finally the arguments themselves. Inside the function these arguments are popped one by one, the algebraic operations are performed, and finally the return value is pushed on the now empty stack, from where the caller will pop it. The code can be seen in Listing 5.11.

```
global_stack.push(470134139, 14982613639, 12215088532, 3, 0);
string_decoder();
result = global_stack.pop(); /* = "fromCompatibility" */

function string_decoder() {
  magic_array = [];
  magic1 = "zeWURhDQZoAbrw_F4km9tlOI5ysBHYE0JC67KS8avqz1gdG-pNX3uTnL2VMiPcfj$";
  magic_array.push(magic1);
  magic2 = "OnlSegCJXqkRd9UsrtoD57fhyviG0Qc2IWTaP_KNmM-LZA18FEzVu3BYjzp4bHw6$";
  magic_array.push(magic2);
  string_index = global_stack.pop(); /* 1st param */
  chunk_number = global_stack.pop(); /* 2nd param */
  magic3 = "jGi8LrT1_SpIE7930DOtezvhgzamMZbuwQUBRdYnJlKN4sc6XPo-HWCVk52FfqAy$";
  magic_array.push(magic3);
  magic = magic_array[string_index];
  chunk_index = 0;
  result = "";
  while (chunk_number > chunk_index) {
    key = global_stack.pop(); /* n-th param: data */
    char_index = 0;
    semi_string_from_chunk = "";
    while (key > 0) {
      char_index = key % (magic.length + 1);
      semi_string_from_chunk += magic.charAt(char_index - 1);
      key = Math.floor(key / (magic.length + 1));
    }
    result += semi_string_from_chunk;
    chunk_index++;
  }
  global_stack.push(En);
}
```

Listing 5.11: Stack-based string decoding procedure, extracted and annotated

This obfuscation is not yet automatically defeated by bulldozer due to time constraints.

5.5.3 Observations on the Partial Evaluation System

This partial evaluation system is limited in its scope, since it was meant to analyze only literals such as strings and numbers, and to support only basic operations such as string concatenation, algebraic additions and the like; nonetheless, it reduces the final line count by 5000 lines. More advanced analysis tools are available, either specifically targeted at a single obfuscator (esdeobfuscate [m1e14]), or generic (JStillery [Di 15], Prepack [Fac14]), but those did not absolve the task of reducing line count, which was a pressing issue on itself, given the sheer size of the codebase. In fact, the cited tools, in a worthy attempt to avoid breaking code, may replace consecutive accumulation of strings with multiple separate assignments of increasing length (see Listing 5.12); or may unroll loops, making comprehension even harder (see Listing 5.13). Implementing from scratch a custom partial evaluator allowed for tailored analyses and to cut some corners.

```
qo = "c";
En = "t";
qo += "on";
En += "x";
En += "et";
qo += "s";
En += "no";
En += "c";
qo += "o";
En = En.split("").re-
↪  verse().join("");
qo += "le";
```

```
qo = 'c';
En = 't';
qo = 'con';
En = 'tx';
En = 'txet';
qo = 'cons';
En = 'txetno';
En = 'txetnoc';
qo = 'conso';
En =
↪  ['c','o','n','t','e','x',
↪  't'].join('');
qo = 'console';
```

Listing 5.12: Growing strings processed by JStillery

```
fe = "";
Ln = 0;
be = "\u02D5\u02D3\u02E2";
/*...*/
while (Ln < be.length) {
```

```
fe = "";
Ln = 0;
be = "\u02D5\u02D3\u02E2";
zo = 103;
fe = "g";
```

```
  zo = be.charCodeAt(Ln) -
  ↪ 622;
  fe += String.fromChar-
  ↪ Code(zo);
  Ln++;
  continue;
}
```

```
Ln = 1;
zo = 101;
fe = "ge";
Ln = 2;
zo = 116;
fe = "get";
Ln = 3;
```

Listing 5.13: A loop processed by Prepack

Finally, there are no **`eval`** statements in this process: the operations are matched against a lookup table and the appropriate action is taken. This conservative approach was chosen to avoid the security concerns related to execution of arbitrary code on the reverse engineer's machine.

Chapter 6

Findings

After an accurate reading of bulldozer's output, the hypothesis advanced in chapter 4 were confirmed: extensive user tracking and fingerprinting are performed by collina. The resulting code, albeit still convoluted, shows which information is recovered, how it is stored and the process to go from the raw data to a token.

The structure of the program revolves around the `e` function, which occupies 99.5% of the lines of code in the output file. This function performs a variety of different tasks and, probably, is the result of multiple inlining operations, given the amount of duplicated code. The behavior of `e` changes according to the arguments it is fed with: of its 5 parameters, the first one is always present, others are optional. An incomplete list of the possible values passed as the first argument, along with the consequent behavior, can be found in section A. This function recursively calls itself to perform many tasks, such as the stack-based string deobfuscation described in subsection 5.5.2, or registers itself as a callback to events spawned by HTML elements. It does even generate an HTML `<script>` object which is injected in the webpage, to expose its own external interface, containing calls to `e` with predefined values. This runtime generation, associated with the string scrambling, means that a simple text search would not find the methods available in a webpage where collina is loaded, making it harder to understand where they are implemented.

As soon as collina is loaded in the webpage, an initialization process is undertaken to retrieve many browser features and quirks, which allows for precise version pinpointing. This is due to the fact that JavaScript (JS) Application Programming Interfaces (APIs) often vary with new ones being added to browsers and other being deprecated. Different versions of the same browser will support different features, just like different browsers would. This allows to verify whether the information reported in the User-Agent are coherent with the browser actually in use, preventing the user from masking its User-Agent. Other checks are performed on advanced features, such as touchscreen support: most desktop browsers can emulate mobile devices, and they will report touchscreen support via the presence of `ontouchstart` in `window.document.documentElement`. If this is the case, collina will check whether the current platform is any of `arm|iphone|ipad|ipod`, which will not be the case with desktop browsers.

Collina registers itself as an eventlistener for a series of different events, such as `touchstart`, `touchend`, `mousemove`, `deviceorientation`, `keyup`. All these events allow tracking all the interactions with the webpage, following the user's movements, to detect whether or not he's a bot. Oftentimes bots will not actually perform mouse movements but, instead, directly click on elements, which would result in an incoherent movement history. Furthermore, specific detection is performed to avoid testing frameworks such as Selenium [Sel] or Puppeteer [Pup], which are commonly used for web browser automation.

All the information retrieved are placed in a "global storage" array which is passed around during execution, a detailed list of the global storage's fields can be seen in section B. When a token is requested to collina via the exposed interface, some of the fields in the global storage array are encrypted and Base64 encoded. The encryption process has not yet been reverse-engineered.

Also, some anti tampering features are added, such as a duplicate timestamp: the execution timestamp is saved in a variable, then the result of the integer division by `4294967296` (2^{32}) is saved in another variable alongside the remainder of the division. This prevents reuse of tokens, since it is impossible to use again the same data updating the timestamp alone, as the two values would diverge and trigger an alert server-side. Such features are useful against bots, because the authors might want to use data from a genuine navigation session multiple times refreshing the timestamp. Without a complete reverse engineering of the data contained in the array, it would be impossible to perform coherently such an update.

6.1 Quirks

Additionally, during the reverse engineering process, some *creative* techniques made to exploit JS features were noticed; these precautions are especially targeted against human analysis, since they would be of little help against automatic tools, but can greatly confuse a reader.

The code in Listing 6.1 leverages JS object paradigm: everything in JS is an object, even functions, even built-in functions. The built-in `charAt` returns the character at a given position in a string. Using the `defineProperty` method on the `charAt` function, it is possible to add a property the `charAt` object, which will be invisible (the function will continue to work as usual) and always accessible.

The code in Listing 6.2 leverages JS binding peculiarities. The **`this`** keyword references the execution context of a function, be it the global object (`window` in a browser), or the context of the caller. Since ES5, the method `bind` allows for setting the context prior a function call, *binding* it to a specific execution context which can be then referenced inside the function via the **`this`** keyword. Nothing forbids the "context" from actually being a number or some other non-object type, thus it is possible to pass an invisible parameter to functions, since it is not in the function signature.

```
"string".charAt(2); /* "r" */
"string".charAt["new_prop"]; /* undefined */
Object.defineProperty("".charAt, "new_prop", {value:
 ↪  "init_value", writable: true})
"string".charAt(2); /* "r" */
"string".charAt["new_prop"]; /* "init_value" */
```

Listing 6.1: Adding properties to function objects

```
function a(param) {
  hidden_param = +this; /* Coerce to number */
  console.log(hidden_param, param);
}
a("test") /* "NaN test" */
a.bind(1337,"test")(); /* 1337 test */
```

Listing 6.2: Passing a hidden parameter via `bind`

The code in Listing 6.3 leverages JS truthy evaluation: everything not explicitly marked in the ES specification to evaluate to **`false`**, will be considered **`true`** in a boolean context. This means that `0` will be of course **`false`**, but `-1` will be **`true`**, contrarily to what happens in most other languages such as C and Java. Functions such as `findIndex`, which finds the first occurrence of a substring in a string, return `-1` on failure and a number greater than `0` on success. The bitwise negation of a number greater or equal to `0` is a number strictly lesser than `0`, thus it will be considered **`true`**.

```
if (~1) { // ~1 = -2
  /* Is executed, although -2 is negative */
}
if (~-1) { // ~-1 = 0
  /* Isn't executed */
}
```

Listing 6.3: Bitwise negation and JS's truthy evaluation

Chapter 7

Conclusion

The GDPR raised users' awareness worldwide about online privacy, publicizing what used to be a niche topic. Privacy has always been dear to the hacker community, and researchers often reverse engineer products to ensure the promises of confidentiality made by companies are honored. This project was born in the same spirit and, indeed, undisclosed and pervasive user tracking has been found in collina.

All the processes in chapter 5 made possible to recover the original control flow, transforming a chaotic state machine in a readable source file in which the execution flows from top to bottom, as code is normally written and read. The obfuscation could have been more intricate, potentially preventing an effortless reconstruction of the Control Flow Graph (CFG) by means of opaque predicates. To overcome such obstacles, advanced deobfuscators ([Gab14], [SBP18]) leveraging symbolic execution have been devised in the past. If required, ExpoSE [LMK17], a JS symbolic execution engine, could have been adopted to work around sophisticated countermeasures. This makes it clear once again that obfuscation, no matter how initially daunting, can not stop a determined hacker trying to understand the behavior of a piece of code.

As mentioned in this thesis, some corners were cut, but bulldozer's code can be considered mostly solid and retargetable. The CFG parsing routines are, of course, specific to this target and the variable descrambling could be fragile in some edge cases, but the core of the project, the graph restructuring routines, might be fed with graphs from different sources and deliver reasonable results. In the future this project could be expanded to accept source from commercial obfuscators, such as those mentioned in section 3.2: this would require analyzing the structure used by other tools to emulate `goto`s and write specific parsers. The task would be facilitated by the availability of the obfuscation tool as the result of deobfuscation could be compared with input code to spot possible errors.

Appendices

A `e` Function Arguments

Value	Behavior
0	Clear the action of one parameter in this list, which is passed as 2nd argument
1	`getUA`
2	Register element events callbacks in the window
4	`decryptJSON`
5	`getBattery` callback on promise rejection - seems to be not actually handled
6	Return a callback handler
7	Check console support/tampering
8	`setUM`
9	`getBattery` callback on success
10	Working with chrome-data (other argument is passed as **`this`**)
11	Calls `s(7)` where `s` is the second argument passed to this function
12	Set `global_storage[59]=1`, `global_storage[63]=0`, `global_storage[8]=""`
13	`isReadyForSC`
14	Stack-based string decoder
16	Save audio fingerprint data
17	Canvas fingerprint
18	Increments `"".charAt.value` counter by one, if it was `1` before incrementing, clears `global_storage[8]` (firefox data)
19	Instantiate a function in `global_storage[27]` to check if two objects equal
21	Increments `global_storage[13]` by one
23	Unknown, something mobile-specific, possibly Apple only. Returns the fingerprint data in a bitmask number
25	`resetSA`
27	Initialize collina
28	Unknown, has to be called with all the `e` function's arguments populated

Table A.1: (Incomplete) list of values passed as `e` first argument

Value	Behavior
0	history
1	bookmarks
2	storage
3	cookies

Table A.2: (Incomplete) list of values passed as **`this`** when working with chrome-data

B Global Storage and Bitmasks

Position	Content
0	Google Chrome data
3	Device battery charging status
7	Touch info
8	Firefox promise timestamp
12	Initialization timestamp
13	Counter
16	Automation detection
23	Firefox data
24	Set to 0 after 44 has been populated
25	AudioContext object
27	Function to check if two objects are not equal
28	`console.log` configuration
31	set to 0 after 44 has been populated
33	WebGL renderer position in array 55
35	Chrome data
38	Device battery percentage
40	Set to 1 if `window.workbench` is not found
41	WebGL info
44	Tracking data, as per `Dn` below
45	Size of collina.js XOR init timestamp
48	Anti tampering timestamp
50	Battery info fetch status
55	WebGL info
58	set to 0 after 44 has been populated
65	Automation detector
67	Audio fingerprint
69	WebGL vendor position in array 55
72	`setUM` target
73	Firefox promise timestamp

Table B.1: (Incomplete) data in global storage array

Position	Data	Notes
0	`Ce`	
1	`mn`	
2	`255 & global_storage[66]`	
3	`255 & global_storage[11]`	
4	length of `window.navigator.buildID`	
5	`window.navigator.buildID`	
6	`window.ScriptEngineBuildVersion`	`[0,0]` if not running in IE
7	`window.ScriptEngineMajorVersion`	`0` if not running in IE
8	`window.ScriptEngineMinorVersion`	`0` if not running in IE
9	`8`	Length of upcoming data
10	`mn2` split in a 4 byte array	`[0,0,0,0]` if running in IE
11	`mn3` split in a 4 byte array	`[0,0,0,0]` if running in IE

Table B.2: `Dn`

Bit	Value
0	`!!window["webkitRTCPeerConnection"] || !!(window.Element)`
1	`window["mozPaintCount"]`
2	`window["mozInnerScreenX"] !== void 0`
3	`!!window["debug"]`
4	`!!window["WebKitPlaybackTargetAvailabilityEvent"]`

Table B.3: `Ce` bitmask (mozilla specific data)

Bit	Value
0	`window["__wxjs_environment"] === "miniprogram"`
1	`P[__wxjs_environment] === "browser"`
2	`!!window["WindVane"] && !!window["WindVane"]["isAvailables"]`
3	`!!window["AlipayJSBridge"]`
4	`!!window["WeixinJSBridge"]`
5	`!!~window.location.href.indexOf("pc_native=1")`
6	`!!~window.location.href.indexOf("tmd_nc=1")`
7	`!!~window.location.href.indexOf("&native=1")`

Table B.4: `mn` bitmask (Alibaba services/libraries specific)

Bit	Value
0	`window.Symbol.hasOwnProperty("species")`
1	`window.hasOwnProperty("Reflect")`
2	`window.Symbol.hasOwnProperty("toPrimitive")`
3	`window.WeakMap.prototype.hasOwnProperty("clear")`
4	`window.DOMTokenList.prototype.hasOwnProperty("replace")`
5	`window.Symbol.hasOwnProperty("hasInstance")`
6	`window.hasOwnProperty("isSecureContext")`
7	`window.self.hasOwnProperty("origin")`
8	`window.PerformanceTiming.prototype.hasOwnProperty("secureConnectionStart")`
9	`window.hasOwnProperty("showModalDialog")`
10	`window.HTMLDocument.prototype.hasOwnProperty("getSelection")`
11	`window.HTMLMediaElement.prototype.hasOwnProperty("mozAutoplayEnabled")`

Table B.5: `mn2` bitmask

0	`!![]["copyWithin"]`
1	`!![]["includes"]`
2	`window.hasOwnProperty("Touch")`
3	`window.hasOwnProperty("Proxy")`
4	`window.Symbol.hasOwnProperty("match")`
5	`Xo = function() {}; !!Xo.name`
6	`window.Object.hasOwnProperty("values")`
7	`window.OfflineAudioContext.prototype.hasOwnProperty("close")`
8	Unused
9	`!!"".padStart`
10	`window.PointerEvent.prototype.hasOwnProperty("getCoalescedEvents")`
11	Unused
12	`window.hasOwnProperty("BudgetService")`
13	`!!window.document.createElement("canvas") && !!window.document.createElement("canvas").getAttributeNames`
14	`!!window.performance && !!window.performance.timeOrigin !== void 0`
15	`window.Intl.hasOwnProperty("PluralRules")`
16	`window.hasOwnProperty("getMatchedCSSRules")`
17	`window.hasOwnProperty("PerformanceServerTiming")`
18	`!![]["values"]` (Check if array has method `values`)
19	`window.hasOwnProperty("BigInt")`
20	`window.document.wasDiscarded !== void 0`
21	`window.hasOwnProperty("OffscreenCanvas")`
22	`window.hasOwnProperty("globalThis")`
23	`window.Intl.hasOwnProperty("ListFormat")`
24	`window.Object.hasOwnProperty("fromEntries")`
25	`window.Intl.hasOwnProperty("Locale")`
26	`window.MediaStreamTrack.prototype.hasOwnProperty("getCapabilities")`
27	`window.Promise.hasOwnProperty("allSettled")`
28	`window.hasOwnProperty("FormDataEvent")`

Table B.6: `mn3` bitmask

Glossary

Acronyms

AJAX	Asynchronous JavaScript and XML.
API	Application Programming Interface.
AST	Abstract Syntax Tree.
CFG	Control Flow Graph.
ES	ECMAScript.
GDPR	General Data Protection Regulation. *see:* GDPR.
IDE	Integrated Development Environment.
IE	Internet Explorer.
IOCCC	International Obfuscated C Code Contest.
IR	Intermediate Representation. *see:* Intermediate Representation.
JIT	Just In Time.
JS	JavaScript.
PC	Program Counter. *see:* Program Counter.
sCFG	semantic Control Flow Graph.
VM	Virtual Machine.

Glossary

ASM	Abbreviation of assembly language.
Base64	Base64 is a encoding used to represent binary data as an ASCII string.
GDPR	The General Data Protection Regulation (2916/679) is a regulation issued by the European Union on data protection and privacy..

HTML DOM The HTML DOM is a JavaScript (JS) Application Programming Interface (API) which allows JS to add/remove/manipulate HTML elements and react to HTML events (click, mouse over an element, keypress, element change...).

inlining Optimization technique that replaces a function call with the body of the function itself.

Intermediate Representation An Intermediate Representation is the data structure used internally by a compiler, decompiler or virtual machine to represent source code. It sits between machine code and high level programming languages and provides a way of accurately representing the original source without losses of information while being independant of any language or architecture peculiarity.

monoid A monoid, in algebra, is a set equipped with a binary assciative operation $\delta : M \times M \rightarrow M$ and a neutral (or *identity*) element 1, such that $\delta(x, 1) = x \cdot 1 = x$. In computer programming, it can be represented by a type along with two functions, acting as the associative operation and the neutral element. Values of the type can be combined into a new value using the function implementing the operation..

monoidal Relative to monoids.

opaque predicate "*A predicate is opaque if a deobfuscator can deduce its outcome only with great difficulty, while this outcome is well known to the obfuscator*" [CTL97].

outdegree In graph theory, the number of edges leaving a node in a directed graph. It is denoted $\deg^+$.

partial evaluation Partial evaluation is a technique which, applied to a program, modifies it by precomputing the operations whose operands are available at the time of the analysis. The remaining parts are left for runtime evaluation.

Program Counter The Program Counter, in most CPU architectures, is a register used to store the address in memory of the current or next instruction.

SpiderMonkey SpiderMonkey is Mozilla's JavaScript and WebAssembly Engine, used in Firefox, Servo and various other projects. It is written in C++, Rust and JavaScript.

strongly connected component In graph theory, a directed graph is called *strongly connected* if every vertex is reachable from every other vertex. The strongly connected components of a graph are the partitions of the given graph bearing the strong connection property..

symbolic execution Symbolic execution is a software analysis technique aimed at exploring all the possible execution paths in a program, executing it in a *symbolic execution engine* with symbols as input instead of concrete values. Once a program is analyzed, it is possible to determine which input is required to execute a specific piece of code.

User-Agent HTTP request header containing a string which reports information about the browser, such as product name and version.

Bibliography

[Com95] TIS Committee. *Executable and Linking Format (ELF) Specification*. Standard. TIS Committee, May 1995. URL: https://refspecs.linuxfoundation.org/elf/elf.pdf.

[CTL97] Christian Collberg, Clark Thomborson, and Douglas Low. "A Taxonomy of Obfuscating Transformations". In: (1997). URL: https://researchspace.auckland.ac.nz/bitstream/handle/2292/3491/TR148.pdf.

[Knu97] Donald E. Knuth. *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*. USA: Addison Wesley Longman Publishing Co., Inc., 1997. Chap. 2.3. ISBN: 0201896834.

[Fis05] Charles N. Fischer. *Construction of Compilers*. 2005. URL: http://pages.cs.wisc.edu/~fischer/cs701.f05/lectures/LectureAll.4up.pdf.

[Aho+06] Alfred V. Aho et al. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. USA: Addison-Wesley Longman Publishing Co., Inc., 2006. Chap. 8.4. ISBN: 0321486811.

[Kli12] Felix Kling. *JSNetworkX*. 2012. URL: https://github.com/fkling/JSNetworkX.

[Fac14] Facebook Inc. *Prepack*. 2014. URL: https://prepack.io/.

[Gab14] Francis Gabriel. *Deobfuscation: recovering an OLLVM-protected program*. 2014. URL: https://blog.quarkslab.com/deobfuscation-recovering-an-ollvm-protected-program.html.

[m1e14] m1el. *esdeobfuscate*. 2014. URL: https://github.com/m1el/esdeobfuscate.

[Rau14] Axel Rauschmayer. *Speaking Javascript*. Ed. by O'Reilly. 1st ed. O'Reilly, 2014. Chap. 4. ISBN: 9781449365035. URL: http://speakingjs.com/es5/ch04.html.

[ReC14] ReCaptchaReverser. *InsideReCaptcha*. https://github.com/ReCaptchaReverser/InsideReCaptcha. 2014. URL: https://archive.is/Jgn2L.

[Di 15] Stefano Di Paola. *JStillery*. 2015. URL: https://github.com/mindedsecurity/JStillery.

[Yak+15] Khaled Yakdan et al. "No More Gotos: Decompilation Using Pattern-Independent Control-Flow Structuring and Semantics-Preserving Transformations". In: Feb. 2015. DOI: 10.14722/ndss.2015.23185.

[GMM16] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. "Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript". In: *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer International Publishing, 2016, pp. 300–321. DOI: 10.1007/978-3-319-40667-1_15.

[Sho+16] Yan Shoshitaishvili et al. "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis". In: *IEEE Symposium on Security and Privacy*. 2016.

[LMK17] Blake Loring, Duncan Mitchell, and Johannes Kinder. "ExpoSE: Practical Symbolic Execution of Standalone JavaScript". In: *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*. SPIN 2017. New York, NY, USA: Association for Computing Machinery, 2017, pp. 196–199. ISBN: 9781450350778. DOI: 10.1145/3092282.3092295. URL: https://doi.org/10.1145/3092282.3092295.

[NT18] Robert Neumann and Abel Toro. *In-browser mining: Coinhive and WebAssembly*. 2018. URL: https://www.forcepoint.com/blog/x-labs/browser-mining-coinhive-and-webassembly.

[SBP18] Jonathan Salwan, Sébastien Bardin, and Marie-Laure Potet. "Symbolic Deobfuscation: From Virtualized Code Back to the Original". In: June 2018, pp. 372–392. ISBN: 978-3-319-93410-5. DOI: 10.1007/978-3-319-93411-2_17.

[Goo19] Google Inc. *V8 documentation*. Apr. 2019. URL: https://v8.dev/blog/preparser#variable-allocation.

[BBB20] Stepan Bilan, Mykola Bilan, and Andrii Bilan. "Interactive Biometric Identification System Based on the Keystroke Dynamic". In: *Biometric Identification Technologies Based on Modern Data Mining Methods*. Springer International Publishing, Dec. 2020, pp. 39–58. DOI: 10.1007/978-3-030-48378-4_3.

[Lóp+21] Juan Manuel Espín López et al. "S3: An AI-Enabled User Continuous Authentication for Smartphones Based on Sensors, Statistics and Speaker Information". In: *Sensors* 21.11 (May 2021), p. 3765. DOI: 10.3390/s21113765.

[RJ21] Stephen Röttger and Artur Janc. *leaky.page*. 2021. URL: https://leaky.page/.

[Bab] Babel. *Babel AST Specification*. URL: https://github.com/babel/babel/blob/main/packages/babel-parser/ast/spec.md.

[EFF] EFF. *Cover Your Tracks*. URL: https://coveryourtracks.eff.org/.

[Moz] Mozilla. *ESTree AST Specification*. URL: https://github.com/estree/estree.

[NCFa] NCF. *Firefox CVE*. URL: https://www.cvedetails.com/vulnerability-list/vendor_id-452/product_id-3264/Mozilla-Firefox.html.

[NCFb] NCF. *V8 CVE*. URL: https://www.cvedetails.com/vulnerability-list/vendor_id-1224/product_id-17734/Google-V8.html.

[NCFc] NCF. *Webkit CVE*. URL: https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=WebKit.

[Pup] Puppeteer devs. *puppeteer*. URL: https://github.com/puppeteer/puppeteer.

[Sel] Selenium devs. *selenium*. URL: https://github.com/SeleniumHQ/selenium.

[Shaa] Shape Security. *Shift AST Specification*. URL: https://github.com/shapesecurity/shift-spec.

[Shab] Shape Security. *shift-codegen*. URL: https://github.com/shapesecurity/shift-codegen-js.

[Shac] Shape Security. *shift-parser*. URL: https://github.com/shapesecurity/shift-parser-js.

[Shad] Shape Security. *shift-reducer*. URL: https://github.com/shapesecurity/shift-reducer-js.