University of Warsaw
Faculty of Mathematics, Informatics and Mechanics

Tomasz Kępa
Student no. 359746

Detecting Anti-Adblockers using Differential Execution Analysis

Master's thesis
in COMPUTER SCIENCE

Supervisor:
dr Konrad Durnoga
Institute of Informatics

August 2019

Supervisor's statement

Hereby I confirm that the presented thesis was prepared under my supervision and that it fulfils the requirements for the degree of Master of Computer Science.

Date Supervisor's signature

Author's statement

Hereby I declare that the presented thesis was prepared by me and none of its contents was obtained by means that are against the law.

The thesis has never before been a subject of any procedure of obtaining an academic degree.

Moreover, I declare that the present version of the thesis is identical to the attached electronic version.

Date Author's signature

Abstract

Ads are the main source of income of numerous websites. However, some of them are fairly annoying which causes many users to use adblocking browser extensions. Some services, in turn, use specialized scripts to detect such plug-ins and silently report them or block some content as a punishment. The goal of this thesis is to build a pipeline for detecting such scripts based on a differential execution analysis, a method provided by other authors in 2018. Such a mechanism can be used later to analyze the prevalence of anti-adblockers on Polish websites or to build an extension capable of circumventing such scripts.

Keywords

dynamic analysis, differential execution analysis, javascript, anti-adblockers, ads

Thesis domain (Socrates-Erasmus subject area codes)

11.3 Informatics, Computer Science

Subject classification

Software and its engineering. Dynamic analysis

Tytuł pracy w języku polskim

Wykrywanie skryptów blokujących rozszerzenia typu AdBlock w przeglądarkach

Contents

Introduction

In modern-era Internet most webpages operate for profit. Most of them, however, choose to provide free content in exchange for displaying paid ads. Unfortunately, not all websites play fair. Some of them concentrate on displaying as many ads as possible and generate traffic by using click-baits and other shady practices. Even websites with valuable content can have overwhelming amount of ads. This leads to grave dissatisfaction of some portion of users. To make their browsing experience better, they turn to use of ad-blocking extensions.

The amount of users doing that cannot be ignored. In 2019 there was over 615 million devices worldwide with adblocker installed [25]. This, in turn, leads to loss of revenue for many businesses. To combat this, some of them choose to deploy anti-adblockers, which generate warnings or even block content entirely for users with adblockers.

TODO: More on adblockers and anti-adblockers

TODO: Original paper

TODO: My contribution

TODO: Evaluation

Chapter 1

Basic concepts

1.1. Definitions

- Execution event – each occurence of control evaluating some expression, entering or leaving a control statement etc.
- Execution trace – a series of execution events collected during program execution. It is dependent both on program structure and its input (also implicit such as randomly generated numbers).
- Execution index – a concept formally introduced by Xin et al. [37]. For our purposes we can define it is any function that uniquely identifies execution points. In our case it will be a statement source map information (file name and precise location of the statement) with the current function stack.
- Stable marriage problem – given equallly sized sets of men and women and their matrimonial preferences, find a stable matching, i.e. matching in which there exists no pair of man and woman in which they both would have better partner than currently assigned. It can be solved using Gale-Shapley algorithm [35].

1.2. Adblockers

Online advertising market is growing each year. The Interactive Advertising Bureau's report for 2018 [18] states that the Internet advertising revenues surpassed $100 billion annually in the US in 2018. Moreover, according the same report, online advertising is the biggest ad media, approximately 30% bigger than TV.

What it means for the Internet users? That there will be more and more ads. Some sources estimate that an average Internet user is servered 11250 ads per month [9].

Not surprisingly, ads are the main source of income for numerous websites. By displaying ads, authors are able to provide valuable content free of charge.

However, there are multiple reasons why users may not want to see online advertisements [25]:

- There are websites which sole purpose is to earn money by displaying as many ads as possible without providing any interesting or original content. They often generate traffic using click-baits and similar shady practices.

- Ads often lenghten pages loading times. The slower the connection, the more annoying it becomes.
- Ads increase webpage payload size, which generates higher cost on mobile connections.
- Some ads track users, which raises privacy concerns.
- Ads can be used to spread malware (in this case it is called Adware) [3].
- Some of them are annoying.
- They occupy the screen space, leaving less area for the content, which can be especially frustrating on devices with small screens.

One of the solutions is to use special browser extensions, called ad blockers (or adblockers), that prevent ads from being displayed. Their work is based on community-curated list of filters. Those filters are used to first identify some portions of website as ads and later remove them.

The removal depends on the type of an ad. Whole page overlays can simply be not shown without disrupting the website UI. On the other hand, after banners removal, there remains some blank space, which is usually fixed by repositioning adjacent elements. In some cases, particularly when ads are served from domain of some ad provider, the requests that download ads can be blocked, thus saving the bandwidth and reducing data usage.

There is some controversy concerning morality of use of such extensions. One of the most popular ad-blocking extension, uBlock Origin states in its manifesto, that it is the users' right to have control over what content should be accepted in their browser [29]. As if other arguments for using adblockers were not enough of a justification.

Some ad-serving recognize users' frustration and create initiatives like Acceptable Ads [1]. In this program, advertisements meeting certain criteria are whitelisted by the commitee. The whitelist is active by default in adblocking extensions that join the initiative. Some bigger ad providers are also charged for being whitelisted.

1.3. Anti-adblockers

In 2017 PageFair prepared a report on adblockers usage [25]. Worldwide, 11% users use ad blocking solutions. This may seem small, but it is not not uncommon for some markets to around 20% or higher penetration (e.g. USA – 18%, Germany – 29%, Indonesia – 58%).

Such high percentage of users not seeing ads means lost income for many businesses. To recover lost revenue, many websites started deploying anti-adblocking scripts. Their goal is simple – when they detect that content blocking extension is present, they take some action, potentially mitigating the problem. The action can vary from simply just reporting the use of extension to the backend to blocking the content entirely.

The aforementioned PageFair report studied so-called "adblock walls", i.e. mechanism preventing users from seeing the website content when they have an adblocker enabled. The study shows that 90% of ablockers users have come upon an adblock wall. More interestingly, 74% of them leave the website when confronted with such a wall.

Anti-adblockers come in many variants. There are simple, custom scripts written specifically for one service, but there are also some sophisticated scripts provided by third parties, designed to be easily integrated on any site.

One rather simple example is presented by company offering "Adblock Analytics" service [2]. In their example they add a file named *ads.js* with a short JavaScript code that adds

a hidden div block with an unique id. Ad-blocking extensions usually block files named like that. All that remains to detect the extension is to check whether the div element was indeed added to the DOM tree. If not, it is a sign that an adblocker is active.

Another example is the "BlockAdBlock" module [27]. Similarly to the first first example, it first sets up a bait, i.e. a component that should look like an ad to the adblocker, but then the detection phase is a little bit more thorough. It runs the check several times and it inspects several properties of the bait to make sure it has not been tampered with. If something changed – an adblocker is detected. It also allows to register callbacks for both outcomes of the analysis.

Even people that base their websites on content managements systems have an easy access to such scripts. Most notably, around 1 in 3 of the top 10 million most popular website runs on WordPress [36] and in this case there are multiple extensions available, varying in price and capabilities [4].

Naturally, there are multiple users that do not want to disable their adblocker to see the website's content. As a reaction, there are several solutions developed to combat anti-adblockers. Usually, they focus on providing new filtering lists, that can be used by the existing adblocking extensions [23, 26]

Unfortunately, those solutions have rather poor effectiveness, as reported by Zhu et al. [38]. This inability to effectively defuse anti-blockers was their most important reason to develop a method of detecting such scripts.

1.4. Differential Execution Analysis

Differential Execution Analysis is a dynamic program analysis method. Its goal is to pinpoint exact differences in two executions of the same program with different inputs or other conditions (e.g. network or memory errors). Good overview of the method is presented by Johnson et al. [19]

The analysis is usually performed by collecting an execution trace for each run and then comparing them using trace alignment. Trace alignment is a process of identifying which fragments of execution traces are common, where they diverge and when they converge again. The exact algorithm is discussed in section 4.3.

The results can be useful in various scenarios, i.e. debugging a program or during security analysis.

1.5. Detecting anti-adblockers

Zhu et al. in their paper "Measuring and Disrupting Anti-Adblockers Using Differential Execution Analysis" [38] introduced a novel method of detecting anti-adblockers using Differential Execution Analysis. The work presented in this thesis is based on that method. The differences and new ideas are explained in the later sections.

The premise is quite simple. They collect execution traces of JavaScript code on given website first without any content blocking and then with an adblocker turned on. Afterwards, they analyze such traces using Differential Execution Analysis. Any differences in execution in both cases are attributed to anti-adblocking activities of the website.

While the idea is pretty straightforward, the are multiple challenges here. First, trace collection is not a trivial task, especially when the interest exceeds just function entry and exit events. The authors instrument V8 engine to achieve the task, but do not provide much

details, apart from briefly stating that their instrumentation is embedded into native code generation process.

Second, due to JavaScript execution model, described in detail in section 1.6, execution traces of different events are interleaved. To battle this issue, traces have to be sliced into subtraces and analyzed pairwise. In a language with a simpler execution model this step would be unnecessary. Authors also do not explain their method of how to pair the subtraces. They just mention that all $m \times n$ pairs have to be analyzed.

Lastly, the biggest challenge is to combat execution noises, e.g. page randomness or variable content. Authors solve the issue by loading the same page three times with an adblocker and three times without and use redundant traces to generate a black list of execution differences.

1.6. JavaScript execution model

JavaScript concurrency model is based on an "event loop" [24]. The engine is essentially single-threaded and concurrency is implemented by utilizing a message queue. This queue processes events one by one, to completion, i.e. a function corresponding to the message starts with a new, empty stack and its processing is done when the stack becomes empty.

The easiest way to add new events to the queue is by calling *setTimeout* or *setInterval*. Furthermore, all callbacks attached to DOM events (e.g. *onClick*) are executed by adding an event to the queue.

It is worth noting that execution of functions can be intertwined, e.g. when generators are used. This is the reason why trace slicing is needed.

Each iframe and browser tab has its own execution environment that include a separate message loop, more on that in section 3.2.

Chapter 2

Trace collection

2.1. Methods overview

There are a few distinct ways to obtain execution trace of JavaScript code.

The simplest (and most limited) methods use only mechanisms present in the language. More elaborate inject special tracing code to the analyzed script. The last kind modify the engine to produce the desired data.

2.2. Dynamic in-JavaScript code injection

JavaScript is a very dynamic language. For this reason, it is relatively easy to write code that will modify each function present in the environment to log each entry and exit, possibly along with all the arguments and return value [28].

Listing 2.1 is an example of a code that instruments all functions in a selected object to log their name and arguments when they are called. Function *instrument* simply goes through all properties of an object and replaces each function with a new function that first logs function name and all provided arguments and then calls the original function. This function can be easily extended to also log return value and instrument all subobjects recursively.

However, this is not enough for the needs of Differential Execution Analysis. The most obvious limitation is that it is not possible to instrument control statements. Another major shortcoming is that function can be instrumented only after they are defined. It is quite an obstacle, because JavaScript allows to define function practically anywhere and it is very common to use anonymous functions as callbacks. It is not possible to instrument such callbacks without modifying the instrumented code, which brings us the static injection methods.

```
function instrument(obj, withFn) {
    for (const name of Object.keys(obj)) {
        const fn = obj[name];
        if (typeof fn === 'function') {
            obj[name] = (function() {
                return function(...args) {
                    withFn(name, ...args);
                    return fn(...args);
                }
            })();
        }
    }
}

```

```
15 const o = {
16     f(w) {
17         /* function body */
18     }
19 };
20
21 instrument(o, console.log);
22
23 o.f("hello");
```

Listing 2.1: Dynamic instrumentation in JavaScript

2.3. Static code injection

Less limited approach is to statically rewrite the instrumented code and inject tracing code wherever it is needed. The upside of such approach is that there are several ready to use frameworks. The downsides will be pointed out when discussing each solution.

2.3.1. Web Tracing Framework

One notable solution is Web Tracing Framework developed by Google [16]. The main use of this framework is to profile web applications to find performance bottlenecks. The functionality is similar to that of *Performance* tab in Chromium developer tools.

Notwithstanding, one of its advanced features is closer to our needs. It allows the user to first instrument JavaScript sources and then collect execution traces and inspect them in a special app.

Having to instrument all source code is cumbersome, especially when we try to analyze code on some arbitrary websites. For this reason Web Tracing Framework also offers an extension and proxy server that cooperate to instrument all JavaScript code online, when it is loaded into the browser.

Unfortunately, this solution has a few deal-breaking downsides:

- It logs only function entry and exit events.
- Logging format is not public
- It is a bit dated, new JavaScript features may not be traced properly
- Function defined using *eval* or *Function* will not be traced

2.3.2. Iroh

Iroh [14] is the most complete solution based on static code injection. Just like Web Tracing Framework, Iroh also needs to patch the code first, but its capabilities go well beyond what the previous solution offers. It allows the user to register arbitrary callbacks to practically any element of JavaScript's Abstract Syntax Tree (AST). It means that this tool is able to instrument *if* statements. This use case is even included in the official examples.

Unfortunately, the framework does not offer proxy that could instrument code loaded into browser on the fly. There is also another, more general concern – performance of such solution may not be acceptable.

2.4. Engine instrumentation

The last option is to modify the JavaScript engine itself to produce execution traces. The most striking benefit is that the engine already has all required info and the solution does not require the analyzed code to be modified. Another advantage is the performance. Implementing tracing code directly in the engine means that there is less indirection. The code does not need to be interpreted by the engine, it is a native code that is called from JavaScript.

Unfortunately, such instrumentation has to be written almost from scratch. Nevertheless, due to the most flexibility and performance advantages, this solution has been chosen for this implementation.

The same choice has been made by Zhu et al. [38] for their implementation, but they did not share their code.

More details on how to instrument the engine in chapter 3.

Chapter 3

Trace collection by V8 instrumentation

3.1. V8 architecture

Most modern browsers do not implement JavaScript interpreter directly. Rather, they utilize a more specialized program called JavaScript engine, which they usually embed.

V8 is an engine used by the most popular browser, Chrome [33], which in June 2019 had over 80% market share [34].

Let us start with introducing some V8-specific glossary [31]:

- Isolate – an instance of V8. There can be more than one Isolate used by one process of embedding application.
- Context – a concept of global variable scope in V8. Each Context has its own global variables and prototype chain. Each iframe has its own separate Context. There can be multiple Context in one Isolate, but due to site isolation (see section 3.2), each iframe is run in another process and has its own Isolate.

V8 processes JavaScript code in several steps. In this thesis we will focus on steps directly related to implementing trace collection.

In short, JS code is first parsed into AST, which contatins source map information. In the next step V8 traverses the entire AST and emits bytecode for each node. The bytecode is V8-specific and reflects the architecture of V8's abstract machine. More on that in section 3.1.1.

It is worth noting that while user-defined function are translated to bytecode, most built-in functions are implemented in a different way. We will take a closer look at them in section 3.1.2.

Only after the code is translated into bytecode, it is finally executed. At this stage there are two kinds of functions. First – those defined in JavaScript, represented in bytecode, second – builtins defined in other ways and already compiled into native code. This distinction is not important to the user, as those functions do not differ in JavaScript, and can easily call each other. It it will become important once we try adding instrumentation code.

At some point during execution, functions that are called very often with the same argument types, can be compiled into native code by its TurboFan Just In Time compiler [7]. If later the same function is called with some argument of some other type, it simply gets deoptimized into bytecode.

The engine's architecture is focused on achieving superior performance, while conforming to all standards and not jeopardizing security.

3.1.1. JS bytecode

V8's interpreter, Ignition, is a register machine with an accumulator register [15]. While all other registers have to be specified when used as arguments, accumulating register is implicit, it is not specified by bytecodes that use it.

This section is supposed to be only a shallow dive into V8's bytecode. We will have a look at one simple example to be able to understand what is going on in section 3.5.

Let us have a look at a simple JavaScript function and see the bytecode produced by Ignition. [1] Listing 3.1 shows a naive implementation of a function calculating factorial. It includes function call (line 5) because the interpreter is lazy and otherwise the function would not be compiled. List 3.2 presents bytecode generated by V8's interpreter for that function.

```
1 function factorial(n) {
2     return n <= 1 ? 1 : n * factorial(n - 1);
3 }
4
5 console.log(factorial(3));
```

Listing 3.1: Calculating factorial in JavaScript

```
1  [generated bytecode for function: factorial]
2  Parameter count 2
3  Register count 3
4  Frame size 24
5     18 E> 0x167d37c1eb2a @    0 : a5                StackCheck
6     28 S> 0x167d37c1eb2b @    1 : 0c 01             LdaSmi [1]
7     37 E> 0x167d37c1eb2d @    3 : 6b 02 00          TestLessThanOrEqual a0,
      [0]
8           0x167d37c1eb30 @    6 : 99 06             JumpIfFalse [6] (0
      x167d37c1eb36 @ 12)
9           0x167d37c1eb32 @    8 : 0c 01             LdaSmi [1]
10          0x167d37c1eb34 @   10 : 8b 15             Jump [21] (0x167d37c1eb49
      @ 31)
11    48 E> 0x167d37c1eb36 @   12 : 13 00 02          LdaGlobal [0], [2]
12          0x167d37c1eb39 @   15 : 26 fa             Star r1
13          0x167d37c1eb3b @   17 : 25 02             Ldar a0
14    64 E> 0x167d37c1eb3d @   19 : 41 01 04          SubSmi [1], [4]
15          0x167d37c1eb40 @   22 : 26 f9             Star r2
16    52 E> 0x167d37c1eb42 @   24 : 5d fa f9 05       CallUndefinedReceiver1 r1,
       r2, [5]
17    50 E> 0x167d37c1eb46 @   28 : 36 02 01          Mul a0, [1]
18    69 S> 0x167d37c1eb49 @   31 : a9                Return
19 Constant pool (size = 1)
20 0x167d37c1ead9: [FixedArray] in OldSpace
21  - map: 0x0a0b39f807b1 <Map>
22  - length: 1
23            0: 0x167d37c1e741 <String[#9]: factorial>
24 Handler Table (size = 0)
```

Listing 3.2: Ignition bytecode for function *factorial*

The listing starts with the parameter count, register count and frame size. The first one may be baffling at first since *factorial* takes only one number as an argument. We have to remember that all JavaScript functions also take implicit arguments *this*.

[1] V8 prints out bytecode when flag *--print-bytecode* is provided

After the header, the actual bytecode is listed. The first number and letter, e.g. "18 E" in line 5 identifies expression (E) or statement from the source file. The number is an offset in characters. It is followed by the code's address in memory, offset (in bytes) from function start and the code itself in a hexadecimal form. All of them are rather useless for us. The most useful parts are the codes in human-readable form at the end of each line.

Once we recall that most codes use accumulating register, reading the code becomes relatively self-explanatory. To make things easier, the use of accumulating register is reflected in the code's name, e.g. *Ld**a**Constant [0]* loads constant numbered 0 to the accumulating register.

We should now be ready to interpret each line of *factorial*'s bytecode. Upon function entry (line 5) the validity of the stack is checked. Later, integer 1 is loaded into accumulating register. Next, argument 0 (symbol $a0$), which happens to be n, is tested to be less that or equal to the value stored in the accumulating register (1). If not, the jump (to line 11 in the listing) is performed. If the conditional was true, integer 1 is loaded into accumulating register, then the control jumps to the last instruction which returns from the function. The return value is always stored in the accumulating register, so 1 is returned. If the conditional was true, and we are in line 11, constant 0 is loaded, which happens to be the name of our function. Later, that name is stored in register $r1$, argument 0 is loaded into the accumulating register, 1 is subtracted and the result is stored in register $r2$. Next, function of name stored in $r1$ (*factorial*) is called with value stored in register $r2$ ($n - 1$). The result of the call, stored in the accumulating register, is then multiplied by the first argument (n). The result of multiplication is already in the accumulating register and the function can now return.

3.1.2. JS built-in functions

According to V8's documentation post [30], JavaScript built-in functions can be implemented in three different ways. They can be written in JavaScript directly, implemented in C++ (runtime functions) or defined using an abstraction called Code Stub Assembly (CSA). The post, however, is slighty dated. Since then, new abstraction, Torque [32], has been added.

It is not important to know how to write CSA or Torque code. It is only important to remember that most built-in functions are implemented in a different way than a user-defined ones.

3.2. V8 usage in Chromium

Today's usage of V8 in Chromium is determined in large part by security concerns [6]. For us, the most important decision was made after discovery of Spectre [20] and Meltdown [21] vulnerabilities. To increase protection against attacks based on those two vulnerabilites, Chrome's team decided to make Site Isolation enabled by default. [12]

Each website can have multiple iframes – the default one and some embedded ones. All of them are run in a separate process and, as a consequence, have their own instances (Isolates) of V8.

3.3. Chrome's extensions architecture

In the current model, Chrome's extensions may consist of the following components [11]:

- Manifest – a file describing an extension, listing all its files and capabilities.

- UI Elements – code adding extension's user interface.
- Options Page – a page allowing the user to customize the extension.
- Background script – a file with callbacks for browser events. Run only when an event with a registered callback occurs. It runs in its own Context, in a separate process
- Content script – extension's code that is run in the page's Context. This code can read and modify DOM of the website and communicate with partent extension via messages or storage. It can also access a limited subset of Chrome's APIs directly, mostly those needed to communicate with parent extension [10].

3.4. V8's *--trace* flag

Usually the easiest way to implement some new functionality is to find a code that provides a similar functionality and extend it. In case of tracing, such base is provided by the V8's *--trace* flag.

First, we will inspect what this flag can do. The example from the section 3.1.1 (Listing 3.1) will be reused. Listing shows console output of V8 with *--trace* flag enabled. The output is well-formated and self-explanatory. Unfortunately, it has some shortcomings. First, there is no source map info. Second, due to the nature of JavaScript, function traces can get intertwined (more on this in section 1.6. And since stack information is limited to just the stack depth, it may be impossible to untangle events in some cases.

```
1    1: ~(this=0x1bd8e6501521 <JSGlobal Object>) {
2    2:  ~factorial(this=0x1bd8e6501521 <JSGlobal Object>, 3) {
3    3:   ~factorial(this=0x1bd8e6501521 <JSGlobal Object>, 2) {
4    4:    ~factorial(this=0x1bd8e6501521 <JSGlobal Object>, 1) {
5    4:    } -> 1
6    3:   } -> 2
7    2:  } -> 6
8 6
9    1: } -> 0x1bc7923804d1 <undefined>
```

Listing 3.3: V8's output for *factorial* with *--trace* flag

Nevertheless, this flag is a good starting point. Let's have a deeper dive into how it works.

We have already seen the bytecode for *factorial* in listing 3.2. Listing 3.4 shows the bytecode for the same function when *--trace* flag is enabled. There are two differences compared to the bytecode produced without tracing flag. First – there is a call to runtime function *TraceEnter* before *StackCheck* upon function entry. Second, just before returning, the result is saved to register $r0$ and runtime function *TraceExit* is called with return value as its argument. *TraceExit* stores its argument back in accumulating register so there is no change in semantics of the inspected code.

```
1 [generated bytecode for function: factorial]
2 Parameter count 2
3 Register count 3
4 Frame size 24
5          0x315c6429eb32 @    0 : 61 a7 01 fb 00    CallRuntime [TraceEnter],
     r0-r0
6    18 E> 0x315c6429eb37 @    5 : a5                StackCheck
7    28 S> 0x315c6429eb38 @    6 : 0c 01             LdaSmi [1]
8    37 E> 0x315c6429eb3a @    8 : 6b 02 00          TestLessThanOrEqual a0,
      [0]
```

```
         0x315c6429eb3d @   11 : 99 06              JumpIfFalse [6] (0
    x315c6429eb43 @ 17)
         0x315c6429eb3f @   13 : 0c 01              LdaSmi [1]
         0x315c6429eb41 @   15 : 8b 15              Jump [21] (0x315c6429eb56
    @ 36)
   48 E> 0x315c6429eb43 @   17 : 13 00 02          LdaGlobal [0], [2]
         0x315c6429eb46 @   20 : 26 fa              Star r1
         0x315c6429eb48 @   22 : 25 02              Ldar a0
   64 E> 0x315c6429eb4a @   24 : 41 01 04          SubSmi [1], [4]
         0x315c6429eb4d @   27 : 26 f9              Star r2
   52 E> 0x315c6429eb4f @   29 : 5d fa f9 05       CallUndefinedReceiver1 r1,
     r2, [5]
   50 E> 0x315c6429eb53 @   33 : 36 02 01          Mul a0, [1]
         0x315c6429eb56 @   36 : 26 fb              Star r0
         0x315c6429eb58 @   38 : 61 a8 01 fb 01    CallRuntime [TraceExit],
    r0-r0
   69 S> 0x315c6429eb5d @   43 : a9                Return
Constant pool (size = 1)
0x315c6429eae1: [FixedArray] in OldSpace
 - map: 0x3806c0d807b1 <Map>
 - length: 1
           0: 0x315c6429e741 <String[#9]: factorial>
Handler Table (size = 0)
```

Listing 3.4: Ignition bytecode for function *factorial* with *--trace* enabled

3.5. Bytecode injection

Finally, we have come to the gist of the current chapter – tracing implementation. Once we have seen how *--trace* flag works, we can improve it to our needs.

The entire implementation requires a few steps:

1. Adding two new flags – one for turning on our tracing (*--trace-dea*), and one for specyfing the file with tracing info

2. Preparing a function that prints out the entire stack with source map information

3. Preparing a set of new runtime functions, one for each type of event we want to trace

4. Injecting calls to runtime functions in the appropriate places

We will not delve too deeply on how to implement each part. Points 1 and 3 are pretty straightforward. Both of them just require adding declaration to special header files.

Point 2 seems the hardest, but luckily there is a similar function in the Chromium codebase. It is worth noting that it is possible to recover source map information during runtime and have the entries contain precise line and column info. However, it turned out to be too slow for use in Chromium, as there was a noticeable slowdown. Almost no webpage could finish loading in a reasonable time. The reason for such poor performance is that those locations are not stored directly in AST. Rather, only offset in characters is stored. To recover line and column info it is necessary to first go through a few levels of abstraction to get the source code and then calculate the coordinates by traversing it. As a consequence, only character offset is displayed. It is always possible to recover more friendly line, column location later, so it is not that big of a deal.

The challenge of part 4 is to have the right offsets of statements/expressions. It is the easiest with functions, as location of beginning of their definition is stored in an object representing the function during runtime. In case of usual statements it is harder as their locations are available only during parsing and bytecode generation stages. To have those offsets accessible during runtime, they are stored as code constants and passed as arguments to runtime functions that print out the code events.

Listing 3.5 shows bytecode generated by Ingition with our tracing flag enabled. Similarly to the original tracing, there are calls to runtime functions upon entry (line 5) and just before returning (lines 23-24). The new part is that also the information which branch was taken is logged. In lines 7-8 the offset information is stored in register $r0$ which is later used by runtime functions that perform the actual logging (lines 12, 15)

```
1  [generated bytecode for function: factorial]
2  Parameter count 2
3  Register count 4
4  Frame size 32
5           0x3dd338d9eb3a @    0 : 61 a9 01 fb 00    CallRuntime [TraceDeaEnter
       ], r0-r0
6     18 E> 0x3dd338d9eb3f @    5 : a5                StackCheck
7     28 S> 0x3dd338d9eb40 @    6 : 12 00             LdaConstant [0]
8           0x3dd338d9eb42 @    8 : 26 fb             Star r0
9           0x3dd338d9eb44 @   10 : 0c 01             LdaSmi [1]
10    37 E> 0x3dd338d9eb46 @   12 : 6b 02 00          TestLessThanOrEqual a0,
       [0]
11          0x3dd338d9eb49 @   15 : 99 0b             JumpIfFalse [11] (0
       x3dd338d9eb54 @ 26)
12          0x3dd338d9eb4b @   17 : 61 af 01 fb 01    CallRuntime [
       TraceDeaIfStmtThen], r0-r0
13          0x3dd338d9eb50 @   22 : 0c 01             LdaSmi [1]
14          0x3dd338d9eb52 @   24 : 8b 1a             Jump [26] (0x3dd338d9eb6c
       @ 50)
15          0x3dd338d9eb54 @   26 : 61 b0 01 fb 01    CallRuntime [
       TraceDeaIfStmtElse], r0-r0
16    48 E> 0x3dd338d9eb59 @   31 : 13 01 02          LdaGlobal [1], [2]
17          0x3dd338d9eb5c @   34 : 26 f9             Star r2
18          0x3dd338d9eb5e @   36 : 25 02             Ldar a0
19    64 E> 0x3dd338d9eb60 @   38 : 41 01 04          SubSmi [1], [4]
20          0x3dd338d9eb63 @   41 : 26 f8             Star r3
21    52 E> 0x3dd338d9eb65 @   43 : 5d f9 f8 05       CallUndefinedReceiver1 r2,
        r3, [5]
22    50 E> 0x3dd338d9eb69 @   47 : 36 02 01          Mul a0, [1]
23          0x3dd338d9eb6c @   50 : 26 fb             Star r0
24          0x3dd338d9eb6e @   52 : 61 aa 01 fb 01    CallRuntime [TraceDeaExit
       ], r0-r0
25    69 S> 0x3dd338d9eb73 @   57 : a9                Return
26 Constant pool (size = 2)
27 0x3dd338d9eae1: [FixedArray] in OldSpace
28  - map: 0x0f45366007b1 <Map>
29  - length: 2
30            0: 35
31            1: 0x3dd338d9e741 <String[#9]: factorial>
32 Handler Table (size = 0)
```

Listing 3.5: Ignition bytecode for function *factorial* with *--trace-dea* enabled

Listing 3.6 shows the output produced by the new flag. Each execution event entry consists of event type, location (optional, depends on event type) and full call stack. Events returning value also log that value, but it is not used later in the pipeline. Stack is represented as a

list of locations. Each location consists of function name, file of origin and offset in the file (in characters). The -1 that appers after the offset is a remnant of the implementation that recovers full position info (line, column). The next part of the pipeline expects two numbers here. This way it is easy to turn on full position recovery and have the pipeline still working.

Execution events that are logged in the current implementation:

- Function enter and exit
- Generator enter, suspend, yield (exit is indistinguishable from a normal function exit)
- If statement then/else paths
- Ternary expression truthy/falsy value (same as if statement paths)

It is easy to extend the implementation to also log execution events associated with loops, but there is a tradeoff between coverage and size of log files created, so it was not done.

```
FUNCTION ENTER
   0: ~ at factorial.js @@ 0,-1 ()
=
--
FUNCTION ENTER
   0: ~factorial at factorial.js @@ 18,-1 ()
   1: ~ at factorial.js @@ 0,-1 ()
=
--
IF STMT - ELSE
   2: ~factorial at factorial.js @@ 35,-1
   0: ~factorial at factorial.js @@ 18,-1 ()
   1: ~ at factorial.js @@ 0,-1 ()
=
--
FUNCTION ENTER
   0: ~factorial at factorial.js @@ 18,-1 ()
   1: ~factorial at factorial.js @@ 18,-1 ()
   2: ~ at factorial.js @@ 0,-1 ()
=
--
IF STMT - ELSE
   3: ~factorial at factorial.js @@ 35,-1
   0: ~factorial at factorial.js @@ 18,-1 ()
   1: ~factorial at factorial.js @@ 18,-1 ()
   2: ~ at factorial.js @@ 0,-1 ()
=
--
FUNCTION ENTER
   0: ~factorial at factorial.js @@ 18,-1 ()
   1: ~factorial at factorial.js @@ 18,-1 ()
   2: ~factorial at factorial.js @@ 18,-1 ()
   3: ~ at factorial.js @@ 0,-1 ()
=
--
IF STMT - THEN
   4: ~factorial at factorial.js @@ 35,-1
   0: ~factorial at factorial.js @@ 18,-1 ()
   1: ~factorial at factorial.js @@ 18,-1 ()
   2: ~factorial at factorial.js @@ 18,-1 ()
   3: ~ at factorial.js @@ 0,-1 ()
=
```

```
--
FUNCTION EXIT
   0: ~factorial at factorial.js @@ 18,-1 ()
   1: ~factorial at factorial.js @@ 18,-1 ()
   2: ~factorial at factorial.js @@ 18,-1 ()
   3: ~ at factorial.js @@ 0,-1 ()
-> 1
=
--
FUNCTION EXIT
   0: ~factorial at factorial.js @@ 18,-1 ()
   1: ~factorial at factorial.js @@ 18,-1 ()
   2: ~ at factorial.js @@ 0,-1 ()
-> 2
=
--
FUNCTION EXIT
   0: ~factorial at factorial.js @@ 18,-1 ()
   1: ~ at factorial.js @@ 0,-1 ()
-> 6
=
--
6
FUNCTION EXIT
   0: ~ at factorial.js @@ 0,-1 ()
-> 0x0da3956804d1 <undefined>
=
--
```

Listing 3.6: V8's output for *factorial* with *--trace-dea* flag

Careful reader might have noticed that in listing 3.1 in the last line there is a call to *factorial* and the result is passed to *console.log*, but only the former call is logged. The reason for this is that the implemented solution works only for functions defined in JavaScript (it is also true for the default *--trace* flag) Functions defined in other ways (see section 3.1.2) are not logged. It is certainly possible to add logging to each one of them by modyfing their code or by modifying CSA/Torque compiler, but it has not been done here. The amount of work required to instrument also those function seems disproportionate to the potential improvements. Another justification is that they are a black box anyway, there is no JavaScript code corresponding to them and we cannot see branch divergences happening inside them.

The last obstacle worth noting is the Chrome's process separation. As explained in section 3.2 there are multiple instances of V8 running at the same time, in different processes. When some flag is passed to Chrome, all instances see the same value. Therefore, if some file is passed, all processes will write to the same file, possibly resulting in interlacing and unusable output. An easy workaround is to create a new file for each Isolate and later just select only the interesting one (the one that corresponds to the analyzed website). There will be always at most one such file (theoretically a website could not contain any JavaScript code) and it is easy to select it by greping by source location. All other files can be deleted.

V8 Isolate is generally pretty oblivious to what code it runs. Website code is the same to V8 as extension code. After all, the environment (Web API, DOM, etc.) is provided by the browser. Nevertheless, it is preferable not to log adblocker events. Such extensions have really long lists of filters and their intialization takes 1-2 seconds when the browser is not producing execution traces. When it does, the initialization takes more than half a minute on a slow computer and the trace itself occupies 4.5GB. Again, we can resort to a trick: inspect the printed event and stop tracing in a given Isolate when some extension-specific file has been

encountered. As a result, extension code is not traced and it has no noticeable performace deterioration.

3.6. Controlling Chrome programatically

Adblocking extensions need a second or two to initialize their code. Even in a stock browser, opening a website immediately after the browser starts can result in ads not being blocked. As a consequence, simply passing the website address through console is not enough when we want to collect traces automatically. The extension will simply not be fully initialized and website will not be blocked. Some more sophisticated solution is needed.

Fortunately, there is a framework for building end-to-end tests – Selenium [5]. It has binding in all major languages, in our case we will use Python.

Selenium is capable of doing any interaction with the website that user can do. It communicates directly with a specialized WebDriver, different for each browser. In case of Chrome it is called ChromeDriver. WebDriver issues the command to the browser through the debugging interface, which is a special port with well-defined communication protocol. And that is all we know about it, as we are not using any of its sophisticated features, just connecting to the browser, opening a website and closing it after some time.

Chapter 4

Trace analysis

4.1. Parsing

Although the format presented in chapter 3 is really simple, parsing it can prove to be challenging. The sole reason is the size of the files. It is really common for websites to produce files of size in the range of a few Gigabytes. Some can even output as much as 48 GB in about 2 minutes!

The analyzing program was written in Haskell [17]. Haskell is a purely functional, lazy, statically typed language. It has been chosen because it is relatively easy to write parsers in a language like this. Further, static typing with pattern matching proves convenient when manipulating well-structured data like log entries. Last but not least, automatically derived instances [1] help avoid writing tedious code and focus on implementing non-trivial parts.

All that being said, Haskell is a bit like C++ – inexperienced user can easily make mistakes that render the code slow and memory-greedy.

The logging format was described in section 3.5. Each entry consist of event type, and several locations. Each location comprises function name, source file and position in the file.

At first, internal format for the event was exactly the same. One object consisted of two strings and two numbers (it could be one number, but this way line and column info logging can be turned on any time).

The first mistake, specific to Haskell, was to use *String* type to represent function and file names. The use of default *String* to represent textual data makes the code inefficient because it is a linked lists of *Char*s, i.e. to store one character 9 bytes of memory are used. It is so wrong and inefficient to use this type that it is not worth trying to profile such implementation.

That error was fixed by changing the representation to *ByteString* type from *bytesting* package [13]. *ByteString*s are represented internally as real byte arrays, not linked lists. This change was accompanied by rewrite of the parsing code to *attoparsec* [8], which can operate of *ByteString*s as an input.

The code was now much less memory-consuming and faster, but it still seemed to consume too much memory. Profiling proved that suspicions were warranted. The profiling diagram is shown on figure 4.1. All profiling diagrams in this section were collected by running the analyzing program on the same set of 6 input files. First three occupy 48MB each, the last three 4MB each. All files combined occupy around 150MB, while peak memory usage of the program is above 900MB.

[1] Without going into details, classes in Haskell are a bit like interfaces in object-oriented languages, e.g. class *Ord* defines objects that can be compared

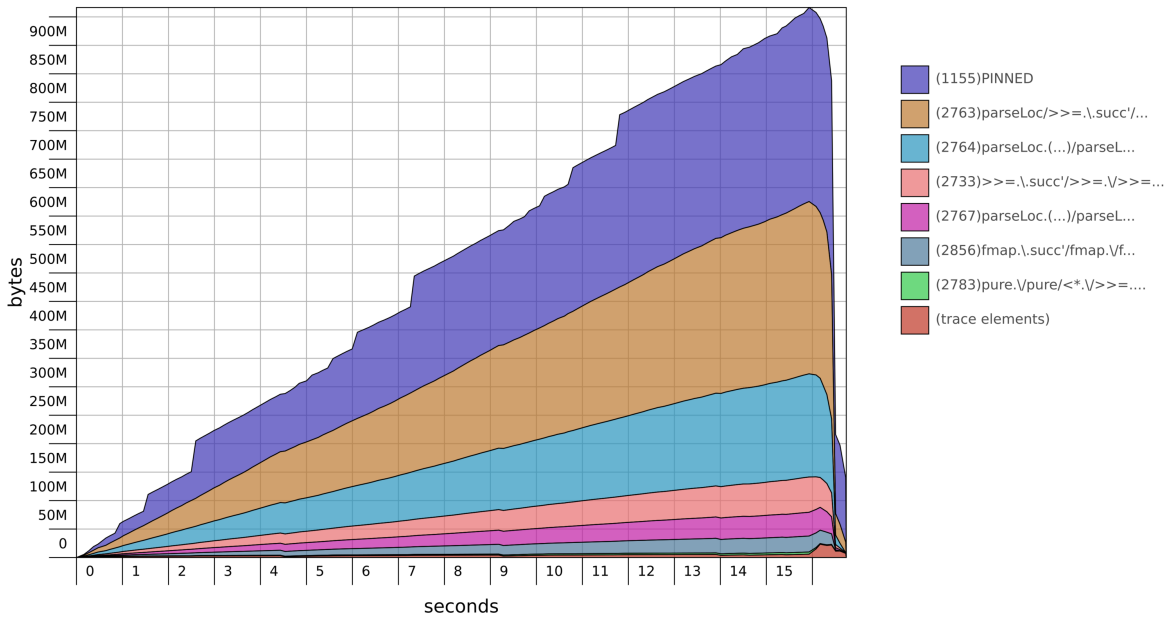

Figure 4.1: Memory usage of the first implementation using ByteStrings as internal strings representation

Looking at the diagram, pinned memory seemed to be never freed during the execution. Also, traces seemed to reside in memory for too long and occupy too much space.

The first problem was a reflection of how Haskell keeps *ByteString*s. They are stored in pinned memory, which is kept on a heap but not managed by garbage collector [22] and cannot be moved around. As a result, it leads to fragmentation of the heap and cannot be effectively managed. The fix turned out to be simple. There is a more suitable storage format – *ShortByteString* (also part of the *bytestring* package). It is managed by GC and stored just like usual heap object, which can be moved around and easily garbage collected. [2]

The second problem was harder to track down. This time the cause was the laziness. Laziness can improve the perfomance when some objects are never used and the language never needs to fully compute them. However, when we know that all objects will eventually be used, it is more efficient to calculate them as soon as possible. The enforcement of eager evaluation seems to be an obscure feature, but when we know what is going on, it is fine. The improvement was visible (figure 4.2) but it was not the end, as peak usage of almost 240MB is still greater than 150MB.

It seems suspicious that the entire file has to be kept in memory to be parsed (pinned memory on figure 4.2 represents input files read into memory). After all, if it was written in C++, probably one line at a time would be read and processed. So why this parser consumes so much memory?

This is what *attoparsec* states about incremental input:

> Note: incremental input does not imply that attoparsec will release portions of its internal state for garbage collection as it proceeds. Its internal representation is equivalent to a single ByteString: if you feed incremental input to a parser,

[2] So why even use *ByteString*? Because it is stored in pinned memory, it can be passed to foreign functions. Also, it is more efficient when the strings are really long and have long livespan

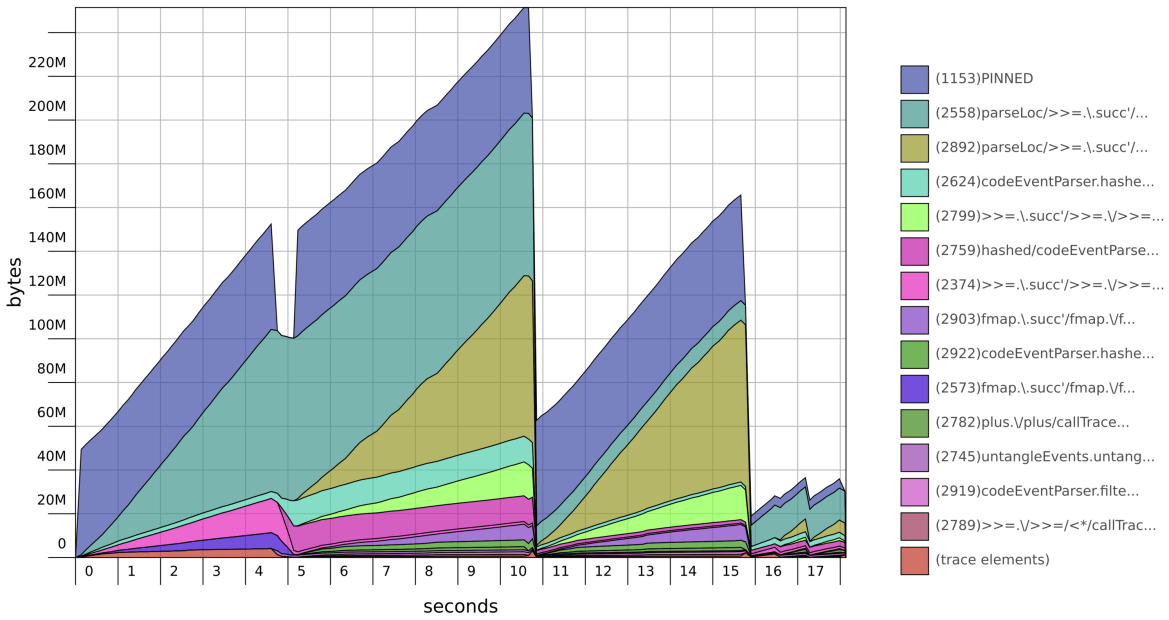

Figure 4.2: Memory usage after switching to ShortByteString and forcing eager evaluation

> it will require memory proportional to the amount of input you supply. (This is necessary to support arbitrary backtracking.)

So, our parser keeps the whole file contents in memory to be able to backtrack. But it is not necessary with such a simple format. Fortunately, it is possible to improve this behaviour. Currently, the parser tries to parse the entire file into a list of events. To be sure that it can backtrack, it keeps the whole input in memory. Instead, we can ask the parser to parse only one event. It will do it and return the part of the input that was not parsed yet. We can repeat that in a loop and parser will never keep more memory that just a few kilobytes. Figure 4.3 shows the improvement.

Last, but not least, the internal representation of the event can be greatly improved. Locations consist of file names and function names. But both sets are limited! Usualy there are only a few sources containing just a few hundreds of unique functions. We can create a map of all source and function names and just keep appropriate ids in location objects. Just 4 numbers instead of 2 strings and 2 numbers!

This representation also has another major benefit – comparisons of such objects are much fasters.

The final memory consumption is presented in figure 4.4

4.2. Trace untangling

Due to the nature of JavaScript execution model (see section 1.6), execution events corresponding to different JavaScript events or functions can be intertwined.

For this reason, all events forming a trace have to be untangled into subtraces, each corresponding to one script or callback.

Let us recall that all events are logged by our instumented Chromium with their call stacks. Now, we have three cases:

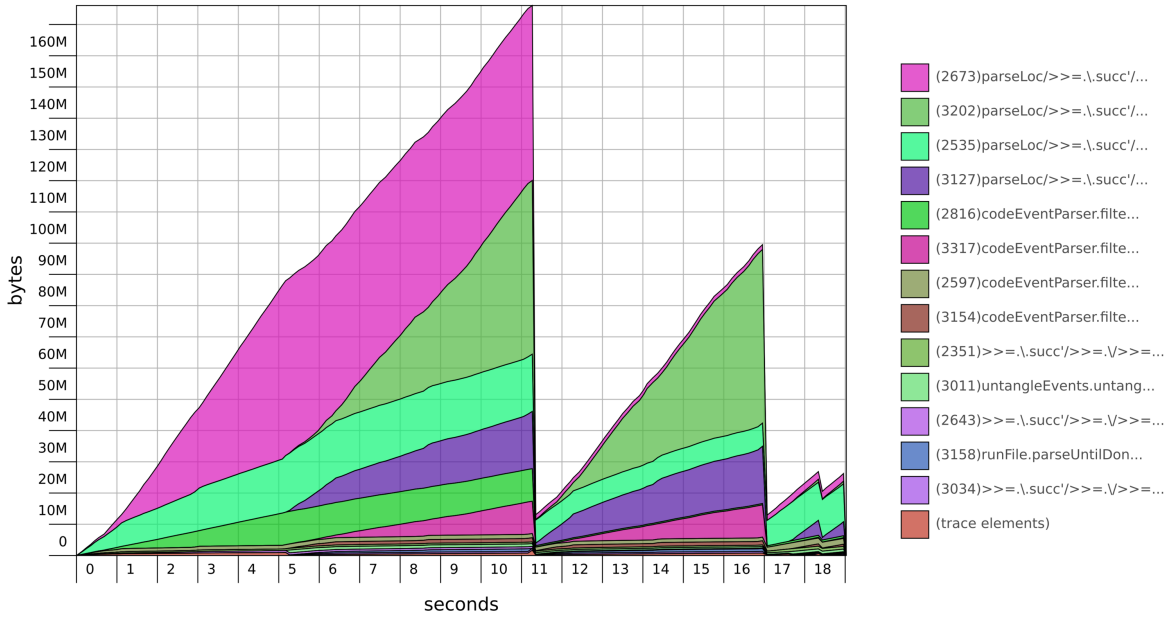

Figure 4.3: Memory usage after preventing parser from keeping the entire input file in the memory

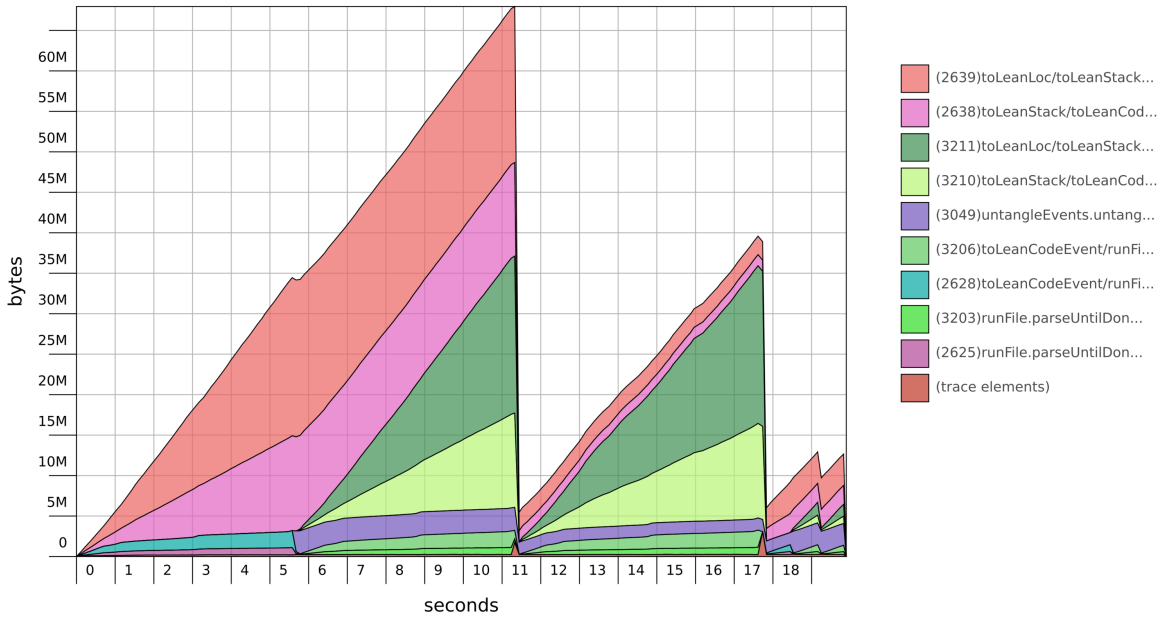

Figure 4.4: Final memory usage

- The event is a function enter with an empty call stack – it starts a new subtrace
- The event is a function exit removing the last item from the stack – it ends a subtrace
- Any other event – it is a continuation of a subtrace, whose last event has the same call stack

Naturally, some care has to be taken to ensure that the right stack is compared. Some events change the call stack, e.g. function enter and exit, some do not, e.g. "if statement – then". Here, the stack *after* the event occured is saved and it is appopriately modified (one location can be added, removed or nothing is changed) when comparing with past events stacks.

Listing 4.1 presents the pseudocode of the algorithm.

```
def untangleEvents(events):
    closedTraces := ∅
    openTraces := ∅

    for event in events:
        if event.type == FunctionEntry and |event.stack| == 1:
            openTraces.insert([event])
        else:
            matchingSubtrace := findMatchingSubtrace(event, openTraces)
            openTraces.remove(matchingSubtrace)
            matchingSubtrace.append(event)

            if event.type == FunctionExit and |event.stack| == 0:
                closedTraces.insert(matchingSubtrace)
            else:
                openTraces.insert(matchingSubtrace)

    return closedTraces

def findMatchingSubtrace(ev, traces):
    st := case
        ev.type in [FunctionEnter, GeneratorEnter] → tail ev.stack
        ev.type in [FunctionExit, GeneratorSuspend] → (ev.loc : ev.stack)
        otherwise → ev.stack

    return trace whose last event's stack equals st
```

Listing 4.1: Pseudocode of the trace untangling algorithm

The above code is rather uncomplicated, but it is slow when implemented naively. The most wasteful part is finding the trace with the right stack in *findMatchingSubtrace*. To make it faster, two things were done:

- Location objects are just 4 numbers instead of 2 strings and 2 numbers. This makes stack comparisons one or two orders of magnitude faster
- Open traces are kept in a map indexed by stack of the last event. The lookup becomes logarithmic instead of linear. This optimization is especially important when there are lots of intertwined subtraces. It also must be noted that this optimization alone would not help much if the comparisons between stacks were slow

4.3. Trace alignment

Trace alignment is a technique of identyfing execution differences of two runs of the same program. To be able to align two traces, events in both have to be assigned execution indices. In our the execution index is the event type, its location and stack at the time of its occurence. In this thesis execution event is often used interchangeably execution event as everything that is logged when an event occurs is the index.

The algorithm used in this implementation is based on an article by Johnson et al. [19] The result of trace alignment is an execution trace diff. It is a list of events of three kinds:

- Common – an event that occured in both traces
- Left – an event that occured only in the left trace
- Right – an event that occured only in the right trace

Each diff event includes the assiciated execution event. Listing 4.2 present the pseudocode of the algorithm.

```
1  def alignTraces(leftTrace, rightTrace):
2      results := []
3      score := 0
4
5      if leftTrace.size() == 0 || rightTrace.size() == 0
6          || leftTrace[0] != rightTrace[0]:
7          return (-1, [])
8
9      while leftTrace.size() > 0 || rightTrace.size() > 0:
10         if leftTrace.size() == 0:
11             results.append(map(Right, rightTrace))
12             rightTrace := []
13         elif rightTrace.size() == 0:
14             results.append(map(Left, leftTrace))
15             leftTrace := []
16         else:
17             (leftEvent:leftTail) := leftTrace
18             (rightEvent:rightTrail) := rightTrace
19
20             if leftEvent == rightEvent:
21                 score := score + 1
22                 results.append(Common(leftEvent))
23                 leftTrace := leftTail
24                 rightTrace := rightTail
25             elif leftEvent.stack.size() <= rightEvent.stack.size():
26                 result.append(Right(rightEvent))
27                 rightTrace := rightTail
28             else:
29                 result.append(Left(leftEvent))
30                 leftTrace := leftTail
31
32     return (score, results)
```

Listing 4.2: Pseudocode of the trace alignment algorithm

The algorithm output includes a score, which is a number of common events. This concept was not present in the original article and its purpose will be explained in section 4.4.

The algorithm is rather simple. We assume that matching traces have to start with the same event. If they do not (lines 5-6), we just terminate and return score of -1, which signifies that traces do not match.

Apart from the case when one of the traces ends, there are only two cases:

- Both unprocessed traces start with the same event (that includes stacks equality), line 20 – in that case we add an event of type Common and remove one event from both lists

- The top events do not match (lines 25 and 28) – in that case the event with longer stack is removed first. The reason is if we repeat that long enough, eventually the that stack will shrink and the traces will reconverge.

  TODO: Explain better why larger stack has to be processed first

4.4. Trace matching using SMP

4.5. Noise filtering

Chapter 5

Evaluation

5.1. Evaluated websites

5.2. Detected anti-adblockers

Listings

List of Figures

Bibliography

[1] Acceptable Ads initiative. Acceptable Ads. https://acceptableads.com, 2019. [Online; accessed 6-August-2019].

[2] Adblock Analytics. Detect Adblock. https://www.detectadblock.com, 2019. [Online; accessed 29-July-2019].

[3] AdBlock authors. What is Adware? How to Identify and Prevent Adware. https://blog.getadblock.com/what-is-adware-ce135d866898, 2019. [Online; accessed 6-August-2019].

[4] Adelina Tuca. 5 best anti adblock wordpress plugins. https://themeisle.com/blog/best-anti-adblock-wordpress-plugins/, 2018. [Online; accessed 6-August-2019].

[5] Baiju Muthukadan. Selenium with Python. https://selenium-python.readthedocs.io, 2019. [Online; accessed 6-August-2019].

[6] Ben L. Titzer, Jaroslav Sevcik. A year with Spectre: a V8 perspective. https://v8.dev/blog/spectre, 2019. [Online; accessed 2-August-2019].

[7] Ben Titzer et al. TurboFan JIT Design. https://docs.google.com/presentation/d/1sOEF4MlF7LeO7uq-uThJSulJlTh--wgLeaVibsbb3tc/edit#slide=id.p, 2019. [Online; accessed 29-July-2019].

[8] Bryan O'Sullivan. attoparsec: Fast combinator parsing for bytestrings and text. http://hackage.haskell.org/package/attoparsec, 2019. [Online; accessed 5-August-2019].

[9] Christopher Elliott. Yes, There Are Too Many Ads Online. Yes, You Can Stop Them. Here's How. https://www.huffpost.com/entry/yes-there-are-too-many-ads-online-yes-you-can-stop_b_589b888de4b02bbb1816c297, 2017. [Online; accessed 6-August-2019].

[10] Chrome team. Content Scripts. https://developer.chrome.com/extensions/content_scripts, 2019. [Online; accessed 2-August-2019].

[11] Chrome team. Extensions Overview. https://developer.chrome.com/extensions/overview, 2019. [Online; accessed 2-August-2019].

[12] Chrome team. Site Isolation. https://www.chromium.org/Home/chromium-security/site-isolation, 2019. [Online; accessed 2-August-2019].

[13] Don Stewart, Duncan Coutts. bytestring: Fast, compact, strict and lazy byte strings with a list interface. http://hackage.haskell.org/package/bytestring, 2019. [Online; accessed 5-August-2019].

[14] Felix Meier. Iroh - Dynamic code analysis for JavaScript. https://maierfelix.github.io/Iroh/, 2018. [Online; accessed 29-July-2019].

[15] Franziska Hinkelmann. Understanding V8's Bytecode. https://medium.com/dailyjs/understanding-v8s-bytecode-317d46c94775, 2017. [Online; accessed 1-August-2019].

[16] Google LLC. Web Tracing Framework. https://google.github.io/tracing-framework/index.html, 2019. [Online; accessed 29-July-2019].

[17] Haskell contributors. Haskell. https://www.haskell.org, 2019. [Online; accessed 5-August-2019].

[18] IAB. IAB Internet Advertising Revenue Report 2018 Full Year Results. https://www.iab.com/insights/iab-internet-advertising-revenue-report-2018-full-year-results/, 2019. [Online; accessed 6-August-2019].

[19] Noah Johnson, Juan Caballero, Kevin Zhijie Chen, Stephen McCamant, Pongsin Poosankam, Daniel Reynaud, and Dawn Song. Differential slicing: Identifying causal execution differences for security applications. In *Proceedings of the 32nd IEEE Symposium on Security and Privacy*, pages pp. 347–362. IEEE, May 2011.

[20] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[21] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[22] Mark Karpov. Short ByteString and Text. https://markkarpov.com/post/short-bs-and-text.html, 2017. [Online; accessed 5-August-2019].

[23] Martin Brinkmann. Integrate Nano Defender with uBlock Origin to block Anti-Adblocker. https://www.ghacks.net/2019/02/15/integrate-nano-defender-with-ublock-origin-to-block-anti-adblocker/, 2019. [Online; accessed 6-August-2019].

[24] MDN. Concurrency model and event loop. https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop, 2019. [Online; accessed 26-July-2019].

[25] PageFair. 2017 Adblock Report. https://pagefair.com/blog/2017/adblockreport/, 2017. [Online; accessed 6-August-2019].

[26] Reek. Anti-Adblock Killer. https://reek.github.io/anti-adblock-killer/, 2019. [Online; accessed 6-August-2019].

[27] sitexw. BlockAdBlock. https://github.com/sitexw/BlockAdBlock, 2018. [Online; accessed 6-August-2019].

[28] StackOverflow users. Adding console.log to every function automatically. https://stackoverflow.com/questions/5033836/adding-console-log-to-every-function-automatically, 2017. [Online; accessed 25-July-2019].

[29] uBlock Origin Contributors. This is uBlock's manifesto. https://github.com/gorhill/uBlock/blob/master/MANIFESTO.md, 2015. [Online; accessed 27-July-2019].

[30] V8 Team. Built-in functions. https://v8.dev/docs/builtin-functions, 2019. [Online; accessed 2-August-2019].

[31] V8 team. Design of V8 bindings. https://chromium.googlesource.com/chromium/src/+/master/third_party/blink/renderer/bindings/core/v8/V8BindingDesign.md, 2019. [Online; accessed 6-August-2019].

[32] V8 Team. V8 Torque user manual. https://v8.dev/docs/torque, 2019. [Online; accessed 2-August-2019].

[33] V8 Team. What is V8? https://v8.dev, 2019. [Online; accessed 29-July-2019].

[34] W3Schools. Browser Statistics. https://www.w3schools.com/browsers/, 2019. [Online; accessed 29-July-2019].

[35] Wikipedia contributors. Stable marriage problem — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Stable_marriage_problem&oldid=907225019, 2019. [Online; accessed 25-July-2019].

[36] Wikipedia contributors. Wordpress — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=WordPress&oldid=909584536, 2019. [Online; accessed 6-August-2019].

[37] Bin Xin, William N. Sumner, and Xiangyu Zhang. Efficient program execution indexing. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 238–248, New York, NY, USA, 2008. ACM.

[38] Shitong Zhu, Xunchao Hu, Zhiyun Qian, Zubair Shafiq, and Heng Yin. Measuring and disrupting anti-adblockers using differential execution analysis. In *NDSS*. The Internet Society, 2018.