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 [10]. 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 executing some statement, entering or leaving a control structure etc.
- Execution trace – a series of execution events collected during program execution. It is dependent both on program structure and it's input (also implicit such as randomly generated numbers)
- Execution index – a concept formally introduced by Xin et al. [22]. For our purposes it is any function that uniquely identifies execution points. In our case it will be a statement source map information 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 [21]

1.2. Adblockers

TODO: References

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 ads:

- 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 frustrating it becomes.
- Ads increase webpage payload size, which generates higher cost in case of mobile connections.
- Some ads track users, which raises privacy concerns
- Ads can be used to spread malware

One of the solutions is to use special browser extensions, called ad blockers, 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.

The removal depends on the type of ad. Whole page overlay can simply be not shown without disrupting website UI. On the other hand, after removing banners, there remain 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 loading ads can be blocked, thus saving the bandwidth and 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 users should have control over what content should be accepted in their browser [16].

1.3. Anti-adblockers

As mentioned earlier, ads are the main source of income for numerous websites, including some news services. Users using AdBlock deprive such websites of their rightful income. 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.

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 run on any site.

One rather simple example is presented by company offering "Adblock Analytics" service [1]. In their example they add file named *ads.js* with a short JavaScript code that adds a hidden div with 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 DOM tree.

TODO: more examples

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. [11]

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 the converge again. The exact algorithm is discussed in section 4.2.

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

1.5. Detecting anti-adblockers

Zhu et al. in their paper "Measuring and Disrupting Anti-Adblockers Using Differential Execution Analysis" [23] introduced a novel method of detecting anti-adblockers using Differ-

ential 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 AdBlock 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 there is interest not just in function entry and exit events. The authors instrument V8 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 AdBlock 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" [14]. 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 processing is done when the stack is 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 and why it requires at least a bit of thought.

Each iframe and browser tab has it's own 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 analyzed script. The last kind modify the engine to produce the desired data.

2.2. Dynamic in-JS 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 [15].

```
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);
                }
            })();
        }
    }
}

const o = {
    f(w) {
        /* function body */
    }
};

instrument(o, console.log)

o.f("hello");
```

Listing 2.1: Dynamic instrumentation in JavaScript

Listing 2.1 is an example of a code that instruments all functions in selected object to log their name and arguments when they are called. Function *instrument* simply goes trough all

properties of an object and replaces each function with 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's 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.

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 [9]. 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 see them in a special app.

Having to instrument all source code is cumbersome, especially when we try to analyze code on some arbitrary website. For this reason Web Tracing Framework also offers an extension and proxy server that cooperate to instrument all JavaScript code when it is loaded into 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 [7] 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 one 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 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 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. [23] 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, but utilize on a more specialized program called JavaScript engine. It is often embedded.

V8 is an engine used by the most popular browser, Chrome [19], which in June 2019 had over 80% market share [20].

TODO: Isolate, Context, Snapshots

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 is, however, important when we try to add instrumentation code.

At some point during execution, functions that are called very often, and with the same argument types, can be compiled into native code by its TurboFan Just In Time compiler [3].

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 [8]. 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 albe 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 8) 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> 0x162d4f41eafa @    0 : a5                StackCheck
6     79 S> 0x162d4f41eafb @    1 : 0c 01             LdaSmi [1]
7     88 E> 0x162d4f41eafd @    3 : 6b 02 00          TestLessThanOrEqual a0,
      [0]
8           0x162d4f41eb00 @    6 : 99 06             JumpIfFalse [6] (0
      x162d4f41eb06 @ 12)
9           0x162d4f41eb02 @    8 : 0c 01             LdaSmi [1]
10          0x162d4f41eb04 @   10 : 8b 15             Jump [21] (0x162d4f41eb19
      @ 31)
11    99 E> 0x162d4f41eb06 @   12 : 13 00 02          LdaGlobal [0], [2]
12          0x162d4f41eb09 @   15 : 26 fa             Star r1
13          0x162d4f41eb0b @   17 : 25 02             Ldar a0
14   115 E> 0x162d4f41eb0d @   19 : 41 01 04          SubSmi [1], [4]
15          0x162d4f41eb10 @   22 : 26 f9             Star r2
16   103 E> 0x162d4f41eb12 @   24 : 5d fa f9 05       CallUndefinedReceiver1 r1,
       r2, [5]
17   101 E> 0x162d4f41eb16 @   28 : 36 02 01          Mul a0, [1]
18   120 S> 0x162d4f41eb19 @   31 : a9                Return
19 Constant pool (size = 1)
20 0x162d4f41eaa9: [FixedArray] in OldSpace
21  - map: 0x3f9cc96807b1 <Map>
22  - length: 1
23            0: 0x162d4f41e741 <String[#9]: factorial>
24 Handler Table (size = 0)
```

Listing 3.2: Ignition bytecode for function *factorial*

The listing starts with prividing 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, but we have to remember that all JavaScript functions also take implicit arguments *this*.

After the header, the actual bytecode is listed. Each line starts with offset (in characters) in the source file. It is followed by the code's address in memory, offset (in bytes) from function start and the code itself in 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.

TODO: Rephrase Reading the code is easy, knowing that most codes use accumulating register, which is reflected in their 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

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

register. Next, argument 0 ($a0$) 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 instructions which returns from the function. The accumulating register stores the return value, so 1 is returned. If the conditional was true, and we 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 [17], 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 as abstraction called CodeStubAssembly (CSA). There post, however, is slighty dated. Since then, new abstraction – Torque [18] – 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 [2]. For us, the most important decision was made after discovery of Spectre [12] and Meltdown [13] vulnerabilities. To increase protection against attacks based on those two vulnerabilites, Chrome's team decided to make Site Isolation enabled by default. [6]

What it means is that each website is put in its own process and, as a consequence, has its own instance of V8.

3.3. Chrome's extensions architecture

In the current model, Chrome's extensions may consist of the following components [5]:

- 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 registered callback happens. It runs in its own content, 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 communicates 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. [4]

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.

Let's see what this flag can do. First, we will reuse the example from section 3.1.1 (Listing 3.1). 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           0x199a2c61eb32 @    0 : 61 a7 01 fb 00    CallRuntime [TraceEnter],
       r0-r0
6     18 E> 0x199a2c61eb37 @    5 : a5                StackCheck
7     28 S> 0x199a2c61eb38 @    6 : 0c 01             LdaSmi [1]
8     37 E> 0x199a2c61eb3a @    8 : 6b 02 00          TestLessThanOrEqual a0,
       [0]
9           0x199a2c61eb3d @   11 : 99 06             JumpIfFalse [6] (0
       x199a2c61eb43 @ 17)
10          0x199a2c61eb3f @   13 : 0c 01             LdaSmi [1]
11          0x199a2c61eb41 @   15 : 8b 15             Jump [21] (0x199a2c61eb56
       @ 36)
12    48 E> 0x199a2c61eb43 @   17 : 13 00 02          LdaGlobal [0], [2]
13          0x199a2c61eb46 @   20 : 26 fa             Star r1
14          0x199a2c61eb48 @   22 : 25 02             Ldar a0
15    64 E> 0x199a2c61eb4a @   24 : 41 01 04          SubSmi [1], [4]
16          0x199a2c61eb4d @   27 : 26 f9             Star r2
17    52 E> 0x199a2c61eb4f @   29 : 5d fa f9 05       CallUndefinedReceiver1 r1,
        r2, [5]
18    50 E> 0x199a2c61eb53 @   33 : 36 02 01          Mul a0, [1]
19          0x199a2c61eb56 @   36 : 26 fb             Star r0
```

```
20          0x199a2c61eb58 @   38 : 61 a8 01 fb 01    CallRuntime [TraceExit],
    r0-r0
21    69 S> 0x199a2c61eb5d @   43 : a9                 Return
22 Constant pool (size = 1)
23 0x199a2c61eae1: [FixedArray] in OldSpace
24  - map: 0x1b10a60007b1 <Map>
25  - length: 1
26            0: 0x199a2c61e741 <String[#9]: factorial>
27 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 and our own tracing.

The entire implementation required a few steps:

1. Adding two new flags – one for turning on our tracing, 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 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 file locations. However, it turned out to be too slow for use in Chromium, as there was a noticeable slowdown and the browser displayed an alert that the page is loading too long. 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, 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 in the analysis, 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)

```
[generated bytecode for function: factorial]
Parameter count 2
Register count 4
Frame size 32
         0x3dd338d9eb3a @    0 : 61 a9 01 fb 00    CallRuntime [TraceDeaEnter
   ], r0-r0
   18 E> 0x3dd338d9eb3f @    5 : a5                StackCheck
   28 S> 0x3dd338d9eb40 @    6 : 12 00             LdaConstant [0]
         0x3dd338d9eb42 @    8 : 26 fb             Star r0
         0x3dd338d9eb44 @   10 : 0c 01             LdaSmi [1]
   37 E> 0x3dd338d9eb46 @   12 : 6b 02 00          TestLessThanOrEqual a0,
    [0]
         0x3dd338d9eb49 @   15 : 99 0b             JumpIfFalse [11] (0
    x3dd338d9eb54 @ 26)
         0x3dd338d9eb4b @   17 : 61 af 01 fb 01    CallRuntime [
    TraceDeaIfStmtThen], r0-r0
         0x3dd338d9eb50 @   22 : 0c 01             LdaSmi [1]
         0x3dd338d9eb52 @   24 : 8b 1a             Jump [26] (0x3dd338d9eb6c
    @ 50)
         0x3dd338d9eb54 @   26 : 61 b0 01 fb 01    CallRuntime [
    TraceDeaIfStmtElse], r0-r0
   48 E> 0x3dd338d9eb59 @   31 : 13 01 02          LdaGlobal [1], [2]
         0x3dd338d9eb5c @   34 : 26 f9             Star r2
         0x3dd338d9eb5e @   36 : 25 02             Ldar a0
   64 E> 0x3dd338d9eb60 @   38 : 41 01 04          SubSmi [1], [4]
         0x3dd338d9eb63 @   41 : 26 f8             Star r3
   52 E> 0x3dd338d9eb65 @   43 : 5d f9 f8 05       CallUndefinedReceiver1 r2,
     r3, [5]
   50 E> 0x3dd338d9eb69 @   47 : 36 02 01          Mul a0, [1]
         0x3dd338d9eb6c @   50 : 26 fb             Star r0
         0x3dd338d9eb6e @   52 : 61 aa 01 fb 01    CallRuntime [TraceDeaExit
   ], r0-r0
   69 S> 0x3dd338d9eb73 @   57 : a9                Return
Constant pool (size = 2)
0x3dd338d9eae1: [FixedArray] in OldSpace
 - map: 0x0f45366007b1 <Map>
 - length: 2
           0: 35
           1: 0x3dd338d9e741 <String[#9]: factorial>
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's not used later in the pipeline. Stack is represented as 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)
- If statement then/else paths
- Ternary expression truthy/falsy value

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.

```
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 v8-builtins) are not logged. It is certainly possible to add logging to each one of them by modyfing its 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.

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 leaving . An easy workaround is to create a new file for each process 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.

TODO: Blocking uBlock from logging events

3.6. Controlling Chrome programatically

TODO: Describe why Selenium is needed and how it works

Chapter 4

Trace analysis

4.1. Trace untangling using execution index

4.1.1. Optimizations

4.2. Trace alignment

4.3. Trace matching using SMP

4.4. Noise filtering

Chapter 5

Evaluation

5.1. Evaluated websites

5.2. Detected anti-adblockers

Listings

Bibliography

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

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

[3] 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].

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

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

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

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

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

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

[10] HostingFacts Team. Internet Stats and Facts for 2019. https://hostingfacts.com/internet-facts-stats, 2019. [Online; accessed 27-July-2019].

[11] 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.

[12] 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.

[13] 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.

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

[15] 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].

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

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

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

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

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

[21] 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].

[22] 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.

[23] 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.