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 [5]. 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. [13]. 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 [12]

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 [9].

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. [6]

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" [14] 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" [7]. 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 [8].

```
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 [4]. 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 [3] 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. [14] 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. Most of them rely on a separate module called JavaScript engine.

V8 is an engine used by the most popular browser, Chrome [10], which in June 2019 had over 80% market share [11].

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 [2].

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

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 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].

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

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

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

[6] 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.

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

[8] 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].

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

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

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

[12] 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].

[13] 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.

[14] 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.