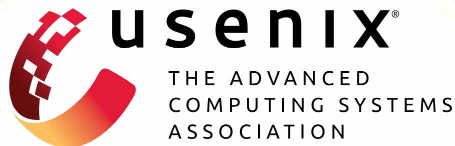

U Can't Debug This: Detecting JavaScript Anti-Debugging Techniques in the Wild

Marius Musch and Martin Johns, *TU Braunschweig*

https://www.usenix.org/conference/usenixsecurity21/presentation/musch

This paper is included in the Proceedings of the
30th USENIX Security Symposium.

August 11–13, 2021

978-1-939133-24-3

Open access to the Proceedings of the
30th USENIX Security Symposium
is sponsored by USENIX.

U Can't Debug This: Detecting JavaScript Anti-Debugging Techniques in the Wild

Marius Musch and Martin Johns
TU Braunschweig, Germany

Abstract

Through security contests such as Pwn2Own, we are constantly reminded that no complex piece of software should ever be considered secure. As we execute untrusted code in our browser every day, browser exploits written in JavaScript remain a constant threat to the security of our systems. In particular, evasive malware that detects analysis systems and then changes its behavior is a well-known problem.

However, there are also *anti-debugging* techniques that interfere with the *manual analysis* of a website *in a real browser*. These techniques try to prevent, or at least slow down, any attempts at manually inspecting and debugging the JavaScript code of a website. For example, such a technique could constantly trigger breakpoints at random locations to effectively hinder single-stepping while debugging the code. More cunningly, it could also find out whether the browser's integrated Developer Tools are open by using certain side-channels available in JavaScript. With this knowledge, it is possible to subtly alter or suppress any malicious behavior while under analysis.

In this paper, we systematically explore this phenomenon. To this end, we introduce 9 anti-debugging techniques and discuss their advantages and drawbacks. We then conduct a large-scale study on 6 of them, to investigate the prevalence of these techniques in the wild. We find that as many as 1 out of 550 websites contain severe anti-debugging measures, with multiple of these techniques active on the same site. Moreover, we present a novel approach based on a deterministic website replay and a comparison of JavaScript code coverage. The approach can automatically detect the remaining 3 timing-based anti-debugging techniques, which use side-channels to learn if the DevTools are open. In a targeted study on 2000 websites with anti-debugging techniques, we discover over 200 of them indeed execute different code when under analysis.

1 Introduction

In our modern, interconnected world, the Web platform is one of, if not the main way our computers interact with the outside world. We use our browsers to visit new websites almost every day, some of which might not be trustworthy at all. Nevertheless, we visit them and execute their JavaScript code on our computers, while relying on the browser to keep us safe. Yet browsers are incredibly complex applications, e.g., in 2020 the Chromium browser had over 25M lines of code [3]. Unsurprisingly, some of these lines have bugs that can have severe security implications [e.g., 8–11]. Therefore, detecting and analyzing JavaScript malware is a crucial task to maintain the security of the Web platform.

Heavy obfuscation and the ability to generate new code during runtime makes a fully static analysis of malicious JavaScript largely infeasible. Therefore, effective detection often relies on a dynamic analysis or a combination of both [e.g., 6, 7, 26, 43]. This then led to a shift towards *evasive* malware which abuses implementation differences between a real browser and dynamic analysis systems, leading in turn to new approaches to deal with such evasive techniques [25].

Yet one, so far, overlooked scenario is the manual analysis of websites using a normal browser, since we can only combat evasive malware deceiving our automated tools if we can manually inspect and learn from it. Unfortunately, this scenario opens up new paths for inventive attackers to interfere with the analysis by creating *anti-debugging techniques targeting humans using real browsers*.

Over the past few years, there were already a few reports of such techniques being used for malicious purposes. For example, in their 2018 paper on cryptojacking Konoth et al. [30] discovered one particular script that stops with the mining of cryptocurrency as soon as the browser's integrated Developer Tools are opened. More recently, in June 2020 the security company Sansec found online stores infected with a script that records credit card information while it is entered and then sends it to the attacker's servers. However, if at any point while visiting an affected domain someone opens the Dev-

Tools, the malicious script detects this, stops sending out its data, and sets a cookie to never activate the skimming for this particular user again [44]. These occurrences demonstrate that some attackers are aware of these anti-debugging techniques and already abuse them in the wild to thwart with manual analyses.

In this paper, we introduce 9 different anti-debugging techniques and present two studies on this phenomenon. In our large-scale study of the 1 million most popular websites, we investigate the prevalence of 6 *basic* techniques, like disabling the console or constantly triggering breakpoints to hinder an inspection. We find that their prevalence varies widely between the different techniques themselves, their aggressiveness (a few vs. 100 breakpoints), their distribution vectors (first vs. third-party code), and their presence on the site (front vs sub-page). Moreover, we also observe that these techniques are more prevalent on certain website categories related to suspicious, illegal, or outright malicious content.

We then follow up with a second study of the 2000 sites with the highest severity of these basic techniques. In this targeted study, we investigate the presence of 3 *sophisticated* techniques, which utilize timing side-channels to detect attempts at analyzing the website. To detect these elusive techniques, we use a generic approach that is based on measuring code coverage during multiple, deterministic replays of the same page. This approach of comparing executions recorded in multiple environments is a proven concept from the area of malware detection in native executables [e.g., 2, 28, 34]. However, we instead use this idea to replay a *whole website's code* to reveal anti-debugging techniques written in JavaScript which target a human analyst. In this study, we find that about 12% of these suspicious sites execute different code under analysis.

To summarize, we make the following contributions:

- Collection and systematization of 9 anti-debugging techniques
- Large-scale study of 6 basic techniques with our automated framework to measure their prevalence and severity in the wild
- Targeted study of 3 sophisticated techniques using a generic approach based on deterministic web page replay and code convergence

2 Background and Scenario

This section briefly describes how to inspect and debug JavaScript code in a browser, followed by the general scenario and what we consider in and out of scope for this paper.

2.1 Debugging JavaScript Code

While previously developers and malware analysts might have relied on browser extensions such as FireBug [41] to inspect a website, nowadays all browsers ship with powerful, integrated Developer Tools [17], or *DevTools* for short. At the time of writing the DevTools of Chromium shipped with 24 different tabs, each focusing on a different feature. In the following, we will briefly introduce the four most useful of these features.

The *elements* tab shows the DOM tree of the currently displayed page. It automatically updates all elements if JavaScript code manipulates them and all elements can also be changed by the user and directly affect the rendered page. The *sources* tab not only allows the inspection of the whole client-side code but also includes a full debugger. With it, the user can set breakpoints anywhere, step through the code, inspect the call stack and variable scopes, and even change the value of variables on the fly. The *console* tab acts like an interactive shell, which allows you to execute arbitrary JavaScript code in the top-level scope of the currently loaded page. If execution is currently suspended at a breakpoint, all code executed in the console will run in the scope of the breakpoint's location instead. The *network* tab, like the name suggests, allows full inspection of all network traffic including the headers and timing data. On top of that, the DevTools offer many advanced features like measuring site performance with a stack-based profiler, creating a heap snapshot to investigate memory leaks, and the ability to measure and inspect code coverage.

Using any other analysis tool that is not part of a browser, e.g., static analysis or executing a single script in isolation is usually *not* an option if one wants to obtain reliable results, due to multiple reasons: First of all, JavaScript code written for the Web expects many objects that are not part of the language specification, like `document` or `location`. Moreover, scripts often load additional code on the fly, e.g., one particular script might generate code for an iframe with a URL as the source and add that to the DOM. The browser then requests the content for that iframe over the network, which might contain additional script code which then again loads additional code via an `XMLHttpRequest`. Previous research has shown that such patterns of deep *causality trees* in script inclusions are a common occurrence today [31, 32]. Only a real browser is able to correctly handle the inherent complexity of modern Web applications and thus only a real browser can be used to accurately inspect and analyze JavaScript code on the Web.

2.2 Threat Model and Scope

Throughout this paper, we consider the following scenario: A user, also referred to as the analyst, manually visits a given website in a real browser to analyze and interact with the website's code. In particular, the user intends to browse the source code of that website, set breakpoints and step through

the code, and inspect variables and functions. On the other hand, the website does not want to be analyzed and contains evasive measures to detect and hinder or, at least, slow down and deter any attempts at inspection.

We consider the browser's integrated DevTools the tool of choice for the user to achieve their analysis goals. As previously outlined, the DevTools are not only full of useful features, but with their integration into the browser also the only way to correctly execute the JavaScript code in the first place. Moreover, using them also avoids the problem of evasive malware potentially detecting the inspection by noticing it does not run in a real browser.

In scope In general, the underlying problem in this scenario is that the analyst can not fully trust the capabilities used during a live inspection, e.g., any logged output during execution, as the website might have manipulated the logging functionality on-the-fly. Furthermore, if the website is able to detect the presence of the inspection, it could also alter or completely suppress any malicious activity to appear benign during analysis. In this paper, we investigate all these techniques that affect the *dynamic analysis* of a website, like altering built-in functions or detecting the presence of a debugger. We refer to such techniques as *anti-debugging techniques* from now on.

Out of scope Since we only focus on techniques that are affecting the code at runtime, all *static code transformation* techniques, in particular obfuscation, are out of scope for this paper. While these can certainly be a powerful tool to greatly slow down manual analysis, especially when combined with some of the anti-debugging techniques introduced in the following, these static techniques have already been extensively studied in the past [e.g., 4, 14, 61, 62]. Similarly, all techniques that do not affect a real browser but rather aim to break sandboxes or other analysis systems, e.g., by intentionally using new features or syntax not yet supported by these systems, are out of scope as well.

3 Basic Anti-Debugging

In this section, we will introduce 6 *basic anti-debugging techniques (BADTs)* with three different goals: Either to outright impede the analysis, or to subtly alter its results, or to just detect its presence. During its introduction, we will give each technique a short name, e.g., *ModBuilt*, by which it will be referenced throughout the remainder of the paper and will also provide a link to a mention of this technique on the Web. Additionally, we provide a testbed[1] with one or two exemplary implementations for each technique so that the interested reader can experiment with each technique while reading this chapter. Finally, we will also briefly describe possible countermeasures for each BADT to give a better impression of how effective they are.

[1] Available at https://js-antidebug.github.io/

3.1 Impeding the analysis

These first three techniques all just try to impede attempts at debugging the website. They are generally not very effective but still might cause an unsuspecting user to give up in frustration.

Preventing Shortcuts (ShortCut) Before any meaningful work can begin, the analyst first needs access to the full client-side code of the website and thus the following BADT simply tries to prevent anyone from accessing that source code. The quickest way to open the DevTools is by using a keyboard shortcut. Depending on the browser and platform there are multiple slightly different combinations to consider, e.g., for Chrome on Windows `F12`, `Ctrl+Shift+I`, and `Ctrl+Shift+J` all work. As JavaScript has the ability to intercept all keyboard and mouse events as long as the website has the focus, these actions can be prevented by listening for the respective events and then canceling them, as shown in Figure 1 [52]. This obviously can not prevent someone from opening the DevTools by using the browser's menu bar.

Besides the advanced DevTools, common browsers also have a simple way to just show the plain HTTP response of the main document. This can usually be accessed by right-clicking and selecting "View page source" from the context menu, or directly with the `Ctrl+U` shortcut. Again, both these actions can be prevented by listening for these events and then canceling them. There are many ways to easily bypass this, e.g., by prefixing the URL with the `view-source:` protocol or opening the sources panel of the DevTools.

```
window.addEventListener("keydown", function(event){
    if (event.key == "F12") {
        event.preventDefault(); return false;
}});
```

Figure 1: Disabling the F12 shortcut

Triggering breakpoints (TrigBreak) The `debugger` statement is a keyword in JavaScript that has the same effect as manually setting a breakpoint [12]. As long as no debugger is attached, i.e., the DevTools are closed, the statement has no effect at all. This behavior makes the statement a perfect tool to only interfere with debugging attempts. The technique can be as simple as just calling the debugger in a fast loop over and over again. As a simple measure to counter this technique, the DevTools of popular browsers have the option to "Never stop here", effectively disabling only the `debugger` statements while still allowing breakpoints in general. However, many variations exist which make it harder to reliably block it, e.g., constantly creating new anonymous functions on the fly instead of always hitting the breakpoint at the same location [46]. On the other hand, this can still be countered by specific code snippets that remove all `debugger` statements on the fly, like the *Anti Anti-debugger* script [60] for the Greasemonkey browser extension [1].

Clearing the Console (ConClear) While the sources panel for the DevTools offers the ability to inspect and change variables in the scopes of the current breakpoint, the console can be very useful in this regard as well. For example, it allows one to easily compare two objects or to run a simple statement at the current location of the suspended execution. However, it is possible to make the console unusable by constantly calling the `console.clear` function [51]. If done fast enough, this makes it near impossible to inspect the output and thus the value of variables during runtime without setting breakpoints with the debugger. However, this technique can be circumvented by enabling "Preserve log" in the DevTools options or by disabling the clear function by redefining it to an empty function.

3.2 Altering the analysis

Instead of only blatantly trying to impede the analysis, the following technique can also subtly alter what an analyst observes during debugging attempts.

Modifying Built-ins (ModBuilt) As JavaScript allows monkey patching, all built-in functions can be arbitrarily redefined. For instance, a popular music streaming service for a while had modified the `alert` function, which many bug bounty hunters use to test for XSS, to secretly leak all client-side attempts to trigger an XSS attack to their back-end, as shown in Figure 2.

```
// Wrapping funcs in a naive attempt to catch
↪ externally found XSS vulns
(function(fn) {
  window.alert = function() {
    var args = Array.prototype.slice.call(arguments);
    _doLog('alert', args);
    return fn.apply(window, args);
  };
}(window.alert));
```

Figure 2: This code including the comment was found on spotify.com in 2018 [59]. The `_doLog` function reports the current URL along with a full stack trace to their backend any time the `alert` function is called.

As this example demonstrates, the possibilities to redefine built-in functions and objects to make them behave differently are basically endless. Furthermore, there are many legitimate use cases, like polyfills that provide a shim for an API not supported by older browsers. Since we are only interested in functions that a human analyst is likely to use in the DevTools console, we focus our search on modifications to the `console`, `String` and `JSON` objects, and their respective functions. Figure 3 shows a somewhat contrived example of how malicious code could hide itself [16]. Note that this technique can also be used to impede the analysis instead, e.g., by redefining all functions like `log` and `info` to an empty function [46, 49]. A possible countermeasure is to save a reference to every native function one intends to use before executing any of the malicious code, a tactic popular in JavaScript rewriting and sandboxing literature [e.g., 39, 42].

```
let original = console.log;
console.log = function(arg) {
  if (arg == "shellcode") { arg = "benign code"; }
  original(arg); }
```

Figure 3: Redefining the log function to hide malicious code

3.3 Detecting the analysis

Finally, the most subtle of all techniques only try to detect the presence of the analysis. In contrast to the previous technique which directly altered the behavior of built-in functions an analyst would use, these techniques instead aim to alter the control flow of their own code. This way, attackers could suppress executing malicious code for any user that opens the DevTools or had them previously open on the same domain.

Inner vs. OuterWidth (WidthDiff) By default, opening the DevTools either splits the browser window horizontally or vertically. In JavaScript, it is possible to obtain both the size of the whole browser window including all toolbars (outer size) and the size of the content area without any toolbars (inner size). Thus by constantly monitoring the `outerWidth` and `innerWidth` properties of the window object, we can check if the DevTools are currently open on the right-hand side. The same works if the DevTools are attached to the bottom, by comparing the height instead, as shown in Figure 4. This is the method used by the popular *devtools-detect* package [47] that, at the time of writing, already had over 1000 stars on Github and is thus probably often used in the wild. This is also the technique used by the credit card skimming case [44] from the introduction.

```
setInterval(() => {
    if (outerWidth - innerWidth > threshold ||
        outerHeight - innerHeight > threshold) {
        //DevTools are open!
    }
}, 500);
```

Figure 4: Monitoring the window size to detect the DevTools

However, this technique does not work if the DevTools are undocked, i.e., open in a separate, detached window. Additionally, this technique will report a false positive if any other kind of sidebar is opened in the browser.

Log Custom Getter (LogGet) Exactly because of the just described drawbacks of the WidthDiff technique, some developers are interested in more reliable alternatives. A StackOverflow question titled "Find out whether Chrome console is open" [50] back from 2011 so far received 130 upvotes

and 14 answers. While many of the suggested approaches have stopped working over the years, some answers are still regularly updated and present working alternatives.

In particular, for at least the last three years, some working variations of what we call the LOGGET technique existed. The technique works by creating an object with a special getter that monitors its `id` property and then calling `console.log` on it. If the DevTools are open, its internals cause it to access the id property of every logged object, but if they are closed, the property is not accessed. Therefore, this getter was a reliable way to determine if the DevTools are open. While the original approach stopped working sometime in 2019, someone created a variation of it that uses `requestAnimationFrame` to log the element with the custom getter which still works as of time of writing. As an alternative, it is also possible to overwrite the `toString` function of an arbitrary function and then log that function, as shown in Figure 5. Since the DevTools internally also use toString to create the printed output, we know that the DevTools are opened whenever this toString function is called.

```
var logme = function(){};
logme.toString = function() {
    //DevTools are open!
}
console.log('\%c', logme);
```

Figure 5: Approach from 2018 to detect the DevTools

As long as one of these variations continues to work, this method is a very reliable way to detect if the DevTools are open, as it also works if they are detached or already open when the website is loaded. There is no real countermeasure except to remove all logging functions of the console object, an invasive step which by itself also might get detected.

3.4 Systematization I

To put the BADTs seen so far into context, we examine them based on four properties: Effectiveness, stealth, versatility, and resilience. An *effective* technique has a high likelihood of activation and thus causing an impact on the analyst. As such, LOGGET is an effective technique while SHORTCUT might never really affect anyone. A *stealthy* technique wants to remain unnoticed, i.e.,WIDTHDIFF is a stealthy technique (although the measures it takes upon detection of the DevTools might be not so stealthy) while TRIGBREAK is the very opposite of stealthy. A *versatile* technique can be used to achieve many different outcomes, as opposed to something very specific. Therefore, MODBUILT is a versatile technique as it can redefine a built-in function to anything else and LOGGET can react in many different ways if it detects the DevTools. A *resilient* technique is not easily circumvented, even if the user is aware of its existence. For example, LOGGET is a resilient technique because there is no good countermeasure, while DEVCUT was easily bypassed by using the menu bar. Table 1 shows the full results of our systematization for each technique. As all four properties are desirable from the perspective of an attacker, the techniques WIDTHDIFF, LOGGET, and MODBUILT offer the most potential.

Table 1: Systematization of BADTs. The goals are **I**mpede, **A**lter, and **D**etect. A filled circle means the property fully applies, a half-filled circle means it applies with limitations.

Technique	Goal	Effective	Stealthy	Versatile	Resilient
SHORTCUT	I	○	○	○	○
TRIGBREAK	I	●	○	○	◐
CONCLEAR	I	◐	○	○	○
MODBUILT	A/I	◐	◐	●	◐
WIDTHDIFF	D	◐	●	●	○
LOGGET	D	●	◐	●	●

4 Large-Scale Study of BADTs

The previously mentioned *devtools-detect* package and also the question on StackOverflow already indicated a certain interest in anti-debugging techniques, in particular in detecting whether the DevTools are opened. However, so far, there has not been a comprehensive study on the prevalence of these techniques in the wild. In this section, we will therefore present a fully automatic methodology to detect each of the BADTs from the previous section and report on the results of our measurement on 1 million web sites.

4.1 Study I – Methodology

In the following, we will briefly outline how we can detect the presence of each technique during a single, short visit to the website. For this, we use the fact that all basic techniques have an obvious "signature" that is easy to detect, e.g., logging an object with special properties. While the detection methodology presented in this section is specifically tailored to each technique and only able to detect exactly them, this methodology is simple, effective, and scales very well. Note that in contrast to this approach, we will also introduce a more generic approach to detect sophisticated anti-debugging techniques in the second half of the paper.

In general, we are using a real Chromium browser for our experiments which is controlled from Node.js via the DevTools Protocol [5]. This means we have many advanced capabilities, e.g., injecting JavaScript into each context before the execution of any other code occurs or programmatically controlling the behavior of the debugger. Yet for any loaded website, we still appear and behave like a normal browser.

ShortCut To detect intercepted key presses, we first collect all event listeners via the `getEventListeners` function. For each collected `keydown` or `contextmenu` listener, we create an

artificial keyboard or mouse event to imitate the shortcut or right click. We pass this event to the listener and then check if the `defaultPrevented` property of the event was set, i.e., the respective normal behavior was blocked by this listener.

TrigBreak By registering the `Debugger.paused` event of the DevTools protocol, we can observe the location of each triggered breakpoint. We log this data and immediately resume execution, to not reveal the presence of the debugger itself.

ConClear To check for attempts at constantly clearing the console, we first register a callback to the `Runtime.consoleAPICalled` event of the DevTools protocol. This API notify us of all invocations of functions of the `console` object and thus allows us to observe how often `console.clear` is called.

ModBuilt We inject JavaScript code into each website which is guaranteed to execute before any of the website's code. Our injected code then creates a wrapper around each object and all of its properties we want to observe. This wrapper will notify our back-end if someone overwrites them or one of their properties. We ignore code that only adds new properties that do not overwrite existing functionality, e.g., a polyfill that adds a new function like `String.replaceAll` to browsers that do not yet support this feature.

WidthDiff We use a similar wrapper as described in MODBUILT, only this time we monitor for read accesses instead of writes to the property `innerWidth` and its siblings. Since we expect that tracking and fingerprinting scripts, in particular, might be interested in some of these values to determine the screen resolution of all visitors, we only flag scripts that access all four properties.

LogGet Similarly to the CONCLEAR technique, we observe all interactions via the console APIs. As the technique requires one to log some specifically crafted objects that are unlikely to be logged during normal operations of a website, we can look out for those. Thus, if we observe a format string logged together with a function that has a custom toString function like in Figure 5, we flag the page. The same applies if we observe the logging of an object that has an `id` property which is a function instead of a value.

Triggering breakpoints or clearing the console once or twice is rather harmless, they only become a problem if they happen constantly. Therefore, for all these 6 BADTs we not only detect *if* they happen but also *how often* per script. One disabled shortcut could be a coincidence, but disabling all five within the same piece of code is most likely a deliberate attempt at preventing access to the source. For this, we only count occurrences in the main frame of the loaded page, since (usually rotating) advertisements should not influence the numbers. Moreover, many techniques lose their effectiveness in iframes, e.g., SHORTCUT would only prevent the shortcut while the iframe is focused. We aggregate all numbers by site, i.e., if a given technique is present on multiple (sub-)pages of the same site, we only count it once. The same applies if one site has multiple different scripts that trigger the same technique. In all cases, we only use the most significant occurrence of each technique within a site for further analysis, e.g., the script that cleared the console most often.

4.2 Study I – Experiment Setup

For our large-scale study, we visited the 1 million most popular websites according to the Tranco list [33] generated on 21 Dec 2020. We started 80 parallel crawlers using Chromium 87.0.4280 on 22 Dec and finished the crawl three days later.

On each page, our crawler waits up to 30 seconds for the load event to trigger, otherwise we flag the site as failed and move on. After the load event, we wait up to 3 more seconds for pending network requests to resolve to better handle pages which dynamically load additional content. Finally, we then stay for an additional 5 seconds on each loaded page, so that techniques that take repeated actions like TRIGBREAK or WIDTHDIFF have enough time to trigger multiple times.

Of all the sites of the initial 1 million, about 15% could not be visited at all, despite having used the most recent Tranco list. Of these, about 8% were due to network errors, in particular, the DNS lookup often failed to resolve. In another 4%, the server returned an HTTP error code and the remaining 3% failed to load before our 30 seconds timeout hit. In total, we successfully visited around 2.8M pages on about 846k sites, where site refers to an entry in the Tranco list which then consists of one or more pages. We did not only visit the front page because research on the *cryptojacking* phenomenon has shown that a common evasive technique is to not run any malicious code on the front page to avoid detection during brief inspections. In line with previous research [30, 40], we therefore additionally selected three random links to an internal subpage and visited these as well.

4.3 Study I – Prevalence

First of all, we are interested in the general prevalence of BADTs in the wild. As can be seen in Table 2, we can find indicators of behavior resembling the six BADTs on over 200k sites. The overwhelming majority of these are caused by MODBUILT and WIDTHDIFF, which, judging from the high numbers, seem to be common behavior also in benign code. Moreover, we can see that visiting subpages did indeed significantly increase the prevalence by about 17% compared to only crawling the front pages. Interestingly, indicators of the more desirable techniques (using the properties from our systematization in Table 1) are also more often hidden in subpages. Specifically, TRIGBREAK is a clear outlier here and breakpoints occurred a lot more often only on subpages.

These results in Table 2 should only be seen as indicators for behavior resembling those of the six BADTs. Next, we analyze how *confident* we are for each occurrence that it is

Table 2: Number of sites with indicators for each technique and the increase from also visiting subpages.

Technique	# Websites	% Total	# Subpages only
ShortCut	4525	0.53	818 (+22%)
TrigBreak	1128	0.13	502 (+80%)
ConClear	3061	0.36	981 (+47%)
ModBuilt	101587	12.00	15345 (+18%)
WidthDiff	114154	13.49	18615 (+19%)
LogGet	3044	0.36	756 (+33%)
Total	206676	24.42	30494 (+17%)

used in an intentional and malicious manner. As previously stated, there is a huge difference between clearing the console once and clearing it 50 times within a few seconds. On the other hand, it makes little difference anymore if it is cleared 20, 50, or even 1000 times which are all highly unusual and hard to cause by accident. In between those two extremes, there is a window of values that are suspicious but not definitely malicious, e.g., clearing it 5 times. As Figure 6 shows, for many techniques about 50% of all detections were caused by just a single occurrence. Looking at ConClear, we can see that of all sites that cleared the console at least once, only about 4% cleared it between 6 and 10 times and only 1% cleared it more than 10 times.

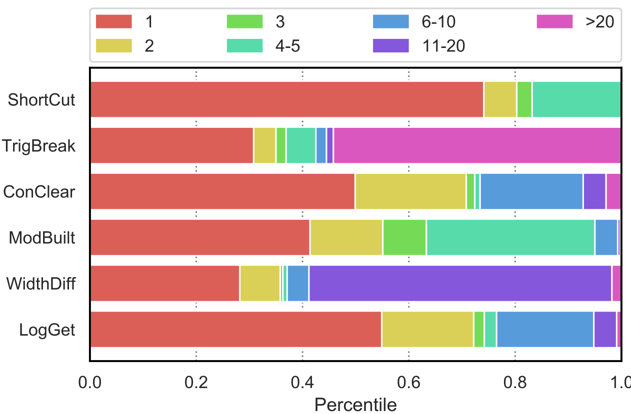

Figure 6: Occurrences within each BADTs grouped into 7 bins, e.g., all sites on which a technique triggered 11-20 times share the same bin. The bins are only used for data visualization and not for further analysis.

To compare indicators of different techniques, we first need a normalized value that incorporates these insights from Figure 6. Therefore, we calculate the *confidence score* by taking the squared value of the percentile within that technique. For example, if we visit a site and see one script that clears the console twice, we would assign a confidence score of $0.6^2 = 0.36$ to this script. On the other hand, if the same script would trigger 30 times, we would assign a confidence score of $0.95^2 = 0.9025$ to it. The rationale behind this formula is that the percentile encodes how often the number of occurrences was observed compared to observations of the same technique on other websites. Squaring this value then puts more weight on the unusually high occurrences, e.g., when the console is cleared dozens of times, resulting in a higher confidence that this usage is intentional and resembles anti-debugging efforts.

Yet, we still have to consider that clearing the console is by itself an uncommon occurrence, with only about 0.53% of all sites behaving this way. A ConClear event with low confidence can still be more significant than e.g.,ModBuilt with a higher confidence score. Thus, we next calculate a *severity score*, which combines the confidence score with the inverse frequency of the techniques, i.e., the more common a technique the less it increases this score. For this, we use the *Inverse Document Frequency* (IDF) from the domain of information retrieval and adapt it to count techniques instead of word terms. Thus, the weights for each technique are calculated as follows: *ln*(number of sites with any technique / number of sites with given technique). This means that the presence of ConClear has a weight of 4.21 while ModBuilt only has 0.71. We then multiply the confidence score with these weights and build the sum over all techniques on the site to obtain the final severity score. Overall, this score considers that 1) some techniques are rarer than others, 2) some sites use these techniques more aggressively than others, and 3) combining different techniques on the same site is more effective.

4.4 Study I – Results

Based on our severity score, we can now analyze the most significant cases of anti-debugging in more detail. In this and the following sections, we focus on the 2,000 sites with the highest severity score, which represents approximately the top 1% of all sites with any indicators. These sites all had a severity score of 3 or higher, as shown in Figure 7. Moreover, the same figure shows that more than two-thirds of these sites had multiple BADTs on the same site, with a few sites as many as 5 simultaneously. On average, the severity score on these 2,000 sites was 4.63 and the average amount of techniques on the same site was 2.28, as the raw numbers in Figure 8 show.

First, we wanted to see if there is a correlation between the popularity of a website and the prevalence of BADTs. We investigated this separately for each technique, to account for their high variance in the total number of occurrences. As shown in Figure 9, BADTs were slightly more prevalent in the higher ranking and thus more popular websites, with the notable exception of ShortCut.

Next, we analyzed the *code provenance* of the scripts we found to be responsible for executing the BADT by distinguishing between first- and third-party scripts. However, it should be noted that the following analysis based on the

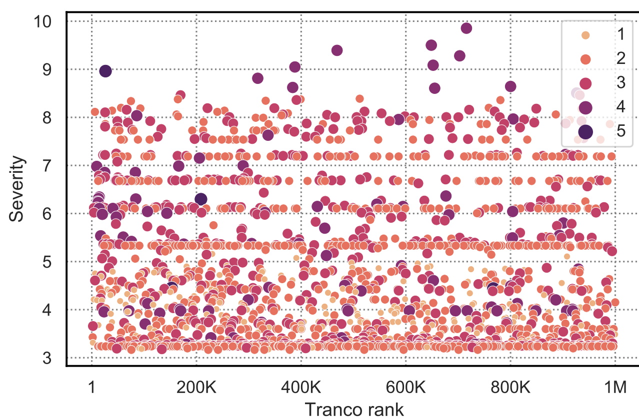

Figure 7: Scatter plot showing the distribution of the severity scores over the Tranco ranks. Size and color both indicate the number of simultaneous techniques on the website.

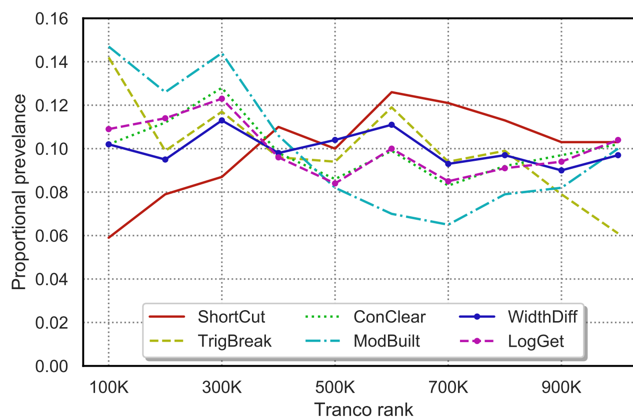

Figure 9: Normalized correlation between website rank and prevalence of each technique in 100k buckets. The most popular sites are on the left.

Figure 8: Severity scores on the left and sites with multiple techniques on the right.

Severity	# Sites
3-4	1095 (54.75%)
5-6	563 (28.15%)
7-8	330 (16.50%)
9-10	12 (0.60%)

(a) Severity scores

Combo	# Sites
1	201 (10.05%)
2	1142 (57.10%)
3	565 (28.25%)
4	88 (4.40%)
5	4 (0.20%)

(b) Combinations of BADTs

eTLD+1[2] is only a rough estimation. For example, a third-party library could also be hosted on first-party servers or first-party code on another domain like a CDN which then would appear to be third-party code. In general, it is rather complex to correctly determine if multiple domains belong to the same owner, as previous research has shown [e.g., 31, 36, 53, 57].

Table 3: BADT occurrence by first- and third-party code.

Technique	# First-party	# Third-party
SHORTCUT	283 (73%)	103 (27%)
TRIGBREAK	282 (81%)	68 (19%)
CONCLEAR	221 (17%)	1084 (83%)
MODBUILT	145 (43%)	195 (57%)
WIDTHDIFF	19 (3%)	707 (97%)
LOGGET	197 (16%)	1059 (84%)
TOTAL	1147 (26%)	3216 (74%)

Now as Table 3 shows, we get a very different picture depending on the technique: SHORTCUT was mainly caused by first-party code, while MODBUILT was more balanced. On the other hand, WIDTHDIFF showed the exact opposite and was with an overwhelming majority present in third-party code. But even if a technique was triggered by third-party code, it still can very well be the first party's intent to interfere with an analysis by including their code. For example, the most prevalent script for causing both SHORTCUT and TRIGBREAK in third-party code is a plugin for the popular e-commerce platform Shopify called *Vault AntiTheft Protection App* [13], which promises to protect the website from competitors that might want to steal one's content.

Now we next want to know if the number of third-party inclusions is caused by relatively few popular scripts or not. In Figure 10 we can see that, e.g., for WIDTHDIFF the most popular script is already responsible for about 51% of all cases in third-party code and the top 5 together cover already 77%. This means that only a very small number of scripts is responsible for the high prevalence of this technique, while for other BADTs this behavior is less pronounced. Moreover, LOGGET and CONCLEAR almost perfectly overlap each other, as the most popular implementations also try to hide the suspicious logged elements by clearing the console immediately afterward each time.

To further investigate this, we performed a manual analysis of the 10 most prevalent third-party scripts for each of the 6 BADTs. We found that many of these scripts are related to advertisements, bot detection, content protection, and crypto-jacking. Moreover, many of them were not just minified but completely obfuscated. In total, 35 of the 60 most prevalent scripts and in particular 9 of the 10 most common scripts causing LOGGET were obfuscated, indicating that these scripts would rather not be analyzed and might even be related to malicious activities.

Finally, we also investigated what types of categories these 2,000 sites belong to. To this end, we used the *WebPulse Site Review* service [55] operated by the security company Symantec. As Table 4 shows, these sites are often related to pornography, piracy, and suspicious activity in general.

[2]The *eTLD* is the effective top-level domain, e.g., for foo.example.co.jp the eTLD is .co.jp and the eTLD+1 is example.co.jp

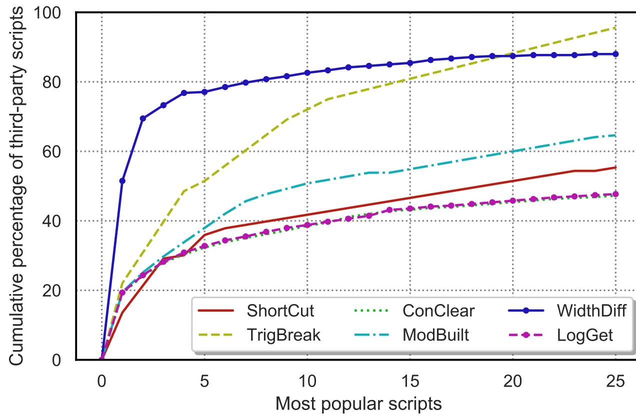

Figure 10: The 25 most common scripts for each technique and their cumulative share of sites

Table 4: Website categories according to Symantec.

Category	# Sites	% Total
Entertainment	183	9.15%
Finance	64	3.20%
Malware	85	4.25%
News	54	2.70%
Other	188	9.40%
Piracy	104	5.20%
Pornography	602	30.10%
Shopping	50	2.50%
Suspicious	232	11.60%
Technology	78	3.90%
Uncategorized	477	23.85%

5 Sophisticated Anti-Debugging

In contrast to the BADTs seen so far, the following *sophisticated anti-debugging techniques (SADTs)* in this chapter are much more elusive. They use side-channels to become aware of an ongoing analysis and then subtly alter the behavior of a website only if they are triggered and otherwise stay dormant.

5.1 Timing-Based Techniques

The following three *timing-based* SADTs are based on the fact that certain operations become slower as long as the DevTools are open. On a high level, these techniques get the current time, e.g., via `Date.now` or `performance.now`, perform some action and then check how much time has passed. If that time is above a specified threshold or changes significantly at one point, then the DevTools were likely opened. These techniques thus use the time between operations as a side-channel about the state of the DevTools. Firefox, for example, lowers the resolution of timers due to privacy concerns and to mitigate side-channel attacks like Spectre [37]. Yet a precision in the range of milliseconds is still more than enough for these techniques to work.

Monitor existing Breakpoint (MonBreak) As the debugger statement only halts the execution if a debugger is attached, we can simply compare the time directly before and after that statement. If it took longer than, e.g., 100ms then we can be sure that the DevTools are open [16]. Figure 11 shows how this technique can be implemented in a few lines of JavaScript code. The main difference to TrigBreak is that the goal here is not to disrupt the user but rather to infer the state of the DevTools. So, in this case, triggering the breakpoint only once is already enough to know somebody is analyzing the website and there is no need to trigger additional breakpoints afterward.

```
function measure() {
    const start = performance.now();
    debugger;
    const time = performance.now() - start;
    if (time > 100) { /*DevTools are open!*/ }
}
setInterval(measure, 1000);
```

Figure 11: Detecting the DevTools by checking for an attached debugger

Wait for new Breakpoint (NewBreak) A more stealthy variation of the MonBreak technique does not trigger breakpoints by itself, but rather detects when the analyst is adding a new breakpoint anywhere. As soon as this new breakpoint is hit, we can again observe this through timing information. If we call a function repeatedly in the same interval and suddenly it took way longer to execute again, there is a good chance that a breakpoint was hit. While this approach is more stealthy, it obviously has no effect as long as someone uses the DevTools without setting a breakpoint at all. Also, note that `setInterval` and similar functions are throttled if the user switches to another tab. Therefore, an additional check with `hasFocus` is needed to confirm that this page is currently in the foreground, as shown in Figure 12.

```
function measure() {
    const diff = performance.now() - timeSinceLast;
    if (document.hasFocus() && diff > threshold) {
        //DevTools are open!
    }
    timeSinceLast = performance.now();
}
setInterval(measure, 300);
```

Figure 12: Detecting the DevTools by checking the time between multiple executions

Console spamming (ConSpam) While the debugger statement is a useful tool to implement anti-debugging measures, it still has the drawback that halting at breakpoints can easily be disabled in the DevTools. The following technique

instead abuses the fact that certain functions of the browser-provided `window` object run slower while the DevTools are open. Historically, this worked by creating many text elements with long content and quickly adding and removing them to the DOM over and over again [50]. This caused a noticeable slowdown, as the elements tab of the DevTools tries to highlight all changes to the DOM in real-time. However, this approach no longer works in both Firefox and Chrome. What still works, at the time of writing, is to write lots of output to the console and check how long this took [19]. As the browser needs to do more work if the console is actually visible, this is a useful side-channel about the state of the DevTools. Conveniently, this technique also works regardless of which tab in DevTools currently has the focus.

Figure 13 shows a possible implementation of this CONSPAM technique. An alternative is to first measure the time a few rounds in the beginning and then always compare to that baseline. This has the advantage that a visitor with slow hardware does not trigger a false positive, as there is no fixed threshold. However, this approach then assumes the DevTools are going to be opened after the page has loaded and not right from the start.

```
function measure() {
    const start = performance.now();
    for (let i = 0; i < 100; i++) {
        console.log(i);
        console.clear();
    }
    const time = performance.now() - start;
    if (time > threshold) { /*DevTools open!*/ }
}
setInterval(measure, 1000);
```

Figure 13: Detecting the DevTools by repeatedly calling functions of the console

5.2 Systematization II

Using the same properties as in our previous systematization of the basic techniques, we now take a look at these newly introduced sophisticated techniques in Table 5. All of them are versatile since they only detect the presence of the analysis and do not prevent the use of certain features. NEWBREAK is stealthier but less effective since, depending on the user's actions, it might not be triggered at all. While MONBREAK stops working if breakpoints are disabled, the other techniques are rather resilient since they are hard to disarm unless one finds their exact location in the code.

6 Targeted Study of SADTs

Now that we have taken a closer look at these SADTs, we also want to find them in the wild. The main challenge in detecting them is that they are a lot more flexible and thus

Table 5: Systematization of SADTs. The goals are **I**mpede, **A**lter, and **D**etect. A filled circle means the property fully applies, a half-filled circle means it applies with limitations.

Technique	Goal	Effective	Stealthy	Versatile	Resilient
MONBREAK	D	●	○	●	○
NEWBREAK	D	◐	●	●	◐
CONSPAM	D	●	◐	●	●

not as easy to detect as the basic techniques. In particular, we can not identify them by just monitoring a few specific function calls and property accesses. While all SADTs rely on timing information, they do not necessarily need access to the `Date` or `performance` objects, as they could also get a clock from a remote source, e.g., via WebSockets. Therefore, we need a more general approach to reliably detect sophisticated techniques in the wild. In the following, we will describe how we address this challenge and then report on our findings.

6.1 Study II – Methodology

While these SADTs can differ in how they are implemented, they still have something in common: They try to figure out whether they are currently analyzed or not and then behave accordingly. Therefore, code execution must diverge from the default, benign case as soon as the analysis is detected. If we somehow could monitor the executed code twice, once with the DevTools open and once with them closed, and then compare those two executions, we would be able to isolate the SADT. Thus, our methodology is based on two concepts: *deterministic website replay* and *code convergence*.

Deterministic website replay To obtain meaningful results when visiting the same website multiple times, we first need a way to reliably load it exactly the same way. In particular, this means we do not want the server-side logic to have any influence on the response and we also do not want dynamic content like different ads on every page load. Therefore, we must load the website only once from the remote server and cache all content on a local proxy. Afterward, we ensure that our browser can not connect to the outside world and loads the page only from our proxy to avoid any interaction with the remote server. However, we also must disable all ways to obtain randomness on the client-side. Otherwise, if a URL parameter contains a random id, the proxy will not have seen this request before and be unable to answer as expected. Thus we replace `Math.random` with a PRNG implementation with a fixed seed and use a fixed timestamp as a starting point for all clock information in `Date` and `performance`.

In theory, without any external logic or randomness, the page should behave entirely deterministic every time we load it, which is exactly what we need for our analysis. Unfortunately, the replays are not entirely perfect. Since in the browser and also the underlying operating system many ac-

tions are executed in parallel, the exact order of events is not always deterministic. For example, consider a website with multiple iframes which all send a postmessage to the main frame upon completion. The main frame could execute different code depending on which frame loaded first. So even if our replay system otherwise works perfectly, we can not prevent that small performance differences in this multi-process system sometimes cause one iframe to load faster than another, leading to different behavior in the main frame in the end. Getting rid of these performance fluctuations is unrealistic, as it would require immense changes to both browser architecture and the underlying operating system's scheduler. Therefore, we instead rely on the concept of *code convergence* to deal with this problem.

Code convergence The idea here is that the more often we replay the same website, the lower the likelihood becomes that we will discover any new execution paths caused by small timing differences. Or to describe it more briefly: The executed code converges over time. We thus replay each page multiple times and always measure the code coverage, i.e., we track which statements in a script are executed and which are not, across all scripts on the page. By merging all seen code from the previous replays, we can check if the current replay introduced any new statements. In the same way, we can also build the intersection of all previously executed code and check if some parts were not executed, which always had been executed before. If now, for multiple replays, no new code is added nor always executed code missing, we likely have executed until convergence.

Detection Methodology By combining these two concepts, we can now replay any website in the same environment until convergence. We can then inject analysis artifacts into the page, like attaching a debugger or adding a breakpoint. As long as we do not make any changes to the website's code, it should behave like during the previous replays. This means we should not see any completely new code, nor should code be missing that previously was always executed. If, however, we reliably observe different code execution only when our analysis artifacts are present, then these differences are most likely caused by an anti-debugging technique.

6.2 Study II – Implementation

We implemented our approach as a tool that can detect SADTs in a fully automated fashion. As in our first study, we control the browser from Node.js by using the *Chrome DevTools Protocol (CDP)*. This protocol exposes all features of the DevTools for programmatic access and gives us low-level information and callbacks for many useful events. In particular, the CDP gives us fine-grained code coverage data with the `Profiler.takePreciseCoverage` command. Moreover, the protocol lets us control the debugger, so we can programmatically enable breakpoints and set them at specific locations, which we need to detect MONBREAK and NEWBREAK. Since the CDP does not include a way to open the DevTools on demand, we instead cause an artificial slowdown of the console to detect the CONSPAM technique. We implemented this by wrapping all functions of the `console` object to first execute a busy loop for a short time, which approximates the slowdown normally caused by an open DevTools window.

For the replaying part, we use a modified version of *Web Page Replay (WPR)* [18], a tool written in Go that is developed and used by Google to benchmark their browser. The tool is designed to record the loading of a website and creates an archive file with all requests and responses, including the headers. This archive file can then be used to create a deterministic replay of the previously recorded page. WPR also tries to make the replays as deterministic as possible, by injecting a script that wraps common sources of client-side randomness like `Math.random` and `Date` to always use the same seed values. Additionally, we improved the accuracy of the replays by extending WPR to always answer with the same delay as the real server during the recording. By combining our Node.js browser instrumentation and this modified Go proxy, we can now automatically detect anti-debugging techniques in the wild.

6.3 Study II – Experiment Setup

To get accurate results, it is important to replay each page multiple times to ensure we have reached code convergence. Therefore, we record each website once and replay it until we get 10 consecutive measurements without any changes in coverage. If after 50 replays this still did not happen, we discard the website as being incompatible with our replaying infrastructure. After convergence, we test each technique 5 times. We only count a technique as present if it caused differences in at least 3 of the replays, to ensure their effect on the code coverage is reproducible.

Replaying this many times is a costly process, especially since we need to restart the browser with a new profile between each replay. Otherwise, stored state in cookies, local storage, and other places could lead to different execution branches. Therefore, in this second study, we only target the 2000 websites with the highest severity score according to our previous study on BADTs in Section 4.3. In the following, we will investigate whether this score is also a good indicator for the presence of sophisticated techniques.

6.4 Study II – Prevalence

While we started this study directly after the first had finished, nevertheless 33 out of the 2000 selected sites were no longer reachable. Another 6 sites did not converge even after 50 replays. On 229 out of the remaining 1961 sites, we could find behavior similar to one or more of the three SADTs. Thus,

about 12% of these sites executed different code when under analysis.

As Table 6 shows, the MONBREAK technique was the most common of the three and present on around 14% of the investigated sites. On the other hand, CONSPAM was rather uncommon with less than 1% prevalence. The technique MONBREAK was mostly seen in first-party code, while NEWBREAK was a bit more often seen in third-party code. However, any difference in third-party code execution might also cause differences in first-party code and vice versa. Thus, there is some overlap between first- and third-party code detections.

Table 6: Sites with SADTs in first- and third-party code.

Technique	# All	# First-party	# Third-party
MONBREAK	138	124	24
NEWBREAK	85	38	54
CONSPAM	8	5	3
TOTAL	229	165	81

When comparing these results to another sample of 100 randomly selected sites, we only found 1 site with a SADT, in this case NEWBREAK. We see this low false positive rate as evidence that our approach to detect sophisticated techniques is reliable. Furthermore, we can see that BADTs are indeed a good indicator for the presence of further sophisticated techniques.

7 Discussion

In the following, we will discuss reasons to employ anti-debugging techniques, some limitations of our presented approach, and how we envision future work to build and improve on this.

7.1 Anti-Debugging and Maliciousness

As with so many other technologies and techniques, the very same thing can be used for both good and evil. On one hand, the anti-debugging techniques presented in this paper obviously can be used to make it more difficult to detect and subsequently analyze malicious JavaScript code. On the other hand, the same techniques can also be used in legitimate ways, e.g., to protect intellectual property by making it harder to extract the content of a website and to discourage reverse-engineering attempts of client-side code. Discerning between these two use cases, however, depends a lot on the context, i.e., what other content and scripts a website is serving. In this regard, anti-debugging techniques share many characteristics with code obfuscation techniques, which can also be used in an attempt to protect intellectual property, as well as to better hide malicious code [45]. Moreover, both do not prevent the analysis in itself, but rather deter by complicating any attempts at it. Thus, any malicious code that makes use of obfuscation and/or anti-debugging techniques has an advantage over code that does not use these techniques, by increasing the chances that an attack can remain undetected for longer.

Previous research has shown that while the obfuscation of JavaScript code does not necessarily imply maliciousness, the majority of malware samples are nevertheless obfuscated [15, 23]. Thus, discerning minified from obfuscated code is important, as the presence of obfuscated code can serve as a useful feature for malware scanners [48]. Due to their similar characteristics, we argue that these findings on code obfuscation likely apply to anti-debugging techniques as well. That is to say, their presence should not be taken as the sole reason to classify a website as malicious. Yet, similar to the presence of obfuscation, their presence can serve as a useful feature for a malware scanner and thus should be taken into account accordingly.

7.2 Limitations

Our approach is essentially a detector for anti-debugging techniques. As such, it struggles with three properties that affect virtually every detector: *completeness*, *false positives*, and *false negatives*.

Completeness First of all, we can not be certain that we have included all existing anti-debugging techniques in this work. However, due to our extensive study of previous publications, blog posts, and Q&A sites on the Web, we are confident that our research investigates the most common and well-known techniques. Moreover, we are certain that all anti-debugging techniques must have at least one of the three goals described in Section 3: Either outright impede the analysis, or subtly alter its results, or just detect its presence. While it is possible that we have missed one particular implementation to achieve one of the three goals, we argue that we are complete in the sense that no entirely new technique with completely different goals does exist.

False positives Many of our measurements are highly accurate, e.g., the code to trigger the techniques DEVCUT and LOGGET is so specific that they are obviously and undeniably anti-debugging techniques and nothing else. However, especially our results on the sophisticated techniques report on websites that would interfere with an analysis, yet their behavior might not necessarily be malicious or intentional. As a backdoor could always be cleverly disguised as a "bugdoor" [56], i.e., look like an innocent programming mistake, we will never know the true intentions behind any suspicious piece of code. Nevertheless, these websites behave differently in an analysis environment. We show that just attaching a debugger or setting a breakpoint during analysis can already have dangerous effects on the outcome of the analysis. Under these circumstances, any derived results should be considered inconclusive at best and deceiving at worst.

False negatives Some actions are only significantly hindering an ongoing analysis if they are happening *constantly* like clearing the console or breakpoints on every function invoke. Therefore, we introduced the confidence and severity scores, to focus on the most severe cases of anti-debugging attempts. Naturally, this means that some sites might have escaped our attention if they trigger the technique only very rarely, but on the other hand then also means their techniques are less effective. Moreover, self-inspecting scripts could become aware of our modifications to built-in functions during the replay and then interfere with our data collection, as we will discuss in the next section. Therefore, our results should be seen as merely the lower bound of active anti-debugging techniques in the wild. To make certain we definitely detect anti-debugging attempts from known implementations, we created a testbed with code snippets found on the Web a well as generated by a JavaScript obfuscator with anti-debugging features [46] to validate our detection methodology.

7.3 Future Work

We see our paper as the first foray into the world of anti-debugging on the Web, where we quantify the problem and raise awareness for this phenomenon. Yet, there is still more to be done, in particular concerning reliably detecting *advanced self-inspection* and deploying effective *countermeasures*.

Advanced self-inspection In this paper, we worked under the assumption that attackers only try to interfere with debugging attempts, but not with our attempts to detect their anti-debugging. Our replaying approach for sophisticated techniques in particular needs to modify built-ins like `Date` and `Math`, which could be detected by self-inspecting scripts. Therefore, the sensible next step is to move these modifications from the JavaScript environment to the C++ realm, where they can not be inspected directly by an attacker and could only be observed through side-effects. Projects like VisibleV8 [22] seem to offer a promising route for researchers to achieve this without a deep understanding of the browser's code.

Countermeasures Some of the presented techniques are trivial to bypass, e.g., DEVCUT just prevents the use of certain hotkeys but not the menu bar to open the DevTools. However, something like preventing the executed JavaScript code from learning that a breakpoint was hit is a much harder problem, as we saw with the MONBREAK technique. This would only be possible to achieve by modifying the browser and its underlying JavaScript engine itself. And even then, freezing the time is an especially difficult feat since a script could also get time information from a remote server and thus easily detect any gaps or clock drifts. Therefore, we would like to see a special *forensic browser* with countermeasures in place to enable safe and reliable debugging of client-side code in an adversarial setting.

8 Related Work

In this section, we will first present works on anti-debugging in native malware, followed by publications on the topic of malicious JavaScript in general and conclude with the most closely related papers about evasive malware on the Web.

8.1 Anti-debugging in General

Anti-debugging techniques are a well-known concept from the area of native x86 malware. Back in 2006, Vasudevan and Yerraballi [58] proposed the first analysis system that focused on mitigations for anti-debugging techniques. Their system called Cobra can, in particular, deal with self-modifying and self-checking code and thus counters many anti-analysis tricks. In 2010, Balzarotti et al. [2] proposed a technique to detect if a malware sample behaves differently in an emulated environment when compared to a reference host. Their main challenge was to achieve a deterministic execution of the malware in both environments so that a robust comparison of behavior becomes possible. Therefore, they first record all interaction of the malware with the operating system to exactly replay the results of the system calls in the second run. One year later Lindorfer et al. [34] extended on this idea with their system called Disarm, by not only comparing the behavior between the emulation and a real system, but instead comparing behavior between four different emulation systems.

Kirat et al. [28] improved on these previous works by creating an analysis platform called *BareCloud* which runs the malware in a transparent bare-metal environment without in-guest monitoring. However, the cat and mouse game continued by finding new techniques to detect and evade even these bare-metal analysis systems. In 2017, Miramirkhani et al. [38] presented their work on "wear and tear" artifacts, i.e., detecting the analysis system because typical artifacts of human interaction with the system in the past are missing.

To summarize, we can see that the deterministic execution of malware in multiple environments and then comparing differences in execution is a well-established approach to analyze malware binaries. However, we are, to the best of our knowledge, the first to apply this concept for JavaScript code running in browsers and to provide insights into how wide-spread these techniques are in the wild.

8.2 Malicious JavaScript

Over the years, there have been many publications on malicious JavaScript in general without any particular focus on evasive measures or anti-debugging. Multiple works focused on drive-by attacks, e.g., *JSAND* by Cova et al. [6] uses anomaly detection combined with an emulated execution to generate detection signatures, while *Cujo* by Rieck et al. [43] use static and dynamic code features to learn malicious pat-

terns and detect them on-the-fly via a web proxy. Similarly, *Zozzle* by Curtsinger et al. [7], uses mostly static features from the AST together with a Bayes classifier to detect malicious code. Targeting drive-by exploit kits, Stock et al. [54] presented their work on *Kizzle*. Their approach is based on the fact that while the obfuscated code of such attacks changes frequently, the underlying unpacked code evolves much more slowly, which aids the detection process. As a more general defense that is not based on a detector, Maisuradze et al. [35] proposed *Dachshund*, which removes all attacker-controlled constants from JavaScript code, rendering JIT-ROP attacks infeasible. Other works focused on malicious browser extensions [26], discovering evil websites [21], and creating fast pre-filters to aid the large-scale detection of malware [4, 15].

8.3 Evasive Malware on the Web

A few publications also specifically focused on evasive JavaScript malware, which actively tries to avoid being detected. In 2011, Kapravelos et al. [24] showed how they can detect the presence of a high-interaction honeyclient and subsequently evade detection. One year later, Kolbitsch et al. [29] created Rozzle, an approach to trigger environment-specific malware via JavaScript multi-execution. This way, they can observe malicious code paths without actually satisfying checks for browser or plugin versions. Improving on this, Kim et al. [27] presented their work on forced execution to reveal malicious behavior, with a focus on preventing crashes. To detect evasive JavaScript malware samples that evolve over time, Kapravelos et al. [25] designed Revolver, which utilizes similarities in samples compared to older versions of the same malware. Their rationale is that malware authors react to detections by anti-virus software and iteratively mutate their code to regain their stealthiness. In their work called *Tick Tock*, Ho et al. [20] investigated the feasibility of browser-based red pills, which can detect if the browser is running in a virtual machine from JavaScript code by using timing side-channels.

However, while these previous publications worked on the phenomenon of evasive Web malware, they all assume the malware is analyzed as part of an *automated system* and tries to detect differences in this analysis environment. On the other hand, our threat model instead considers *anti-debugging* measures to hinder or avoid detection by a *human analyst using a real browser*.

9 Conclusion

In this paper, we systematically explored the phenomenon of anti-debugging techniques targeting human analysts using a real browser. We first introduced 6 basic techniques and conducted a large-scale study to investigate the prevalence of these techniques in the wild. We found that as many as 1 out of 550 sites make use of severe anti-debugging, with multiple techniques active on the same website. Furthermore, we presented a novel approach to detect 3 sophisticated techniques, which is based on web page replaying and code convergence. We used this approach to conduct a second, targeted study on the websites with the most severe anti-debugging measures from the first study. In this study, we could identify over 200 sites that behave differently when under analysis.

While many of these techniques are simple to detect and counter if their presence is known, they can still be quite effective if multiple of them are used together. This is especially true if the code is also additionally obfuscated so that they can not be easily identified in the source code and subsequently removed. As these techniques allow a website to completely change its behavior under analysis, they are a threat to the security of the Web and its users. We, therefore, see the need for a forensic browser with effective and robust inspection capabilities, which can not be detected or interfered with by the website's JavaScript code.

Acknowledgments

We would like to thank our shepherd Nick Nikiforakis and all anonymous reviewers for their valuable comments and suggestions. Moreover, we gratefully acknowledge funding by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972.

References

[1] Anthony Lieuallen. Greasemonkey. Online https://addons.mozilla.org/en-US/firefox/addon/greasemonkey/, June 2019.

[2] D. Balzarotti, M. Cova, C. Karlberger, E. Kirda, C. Kruegel, and G. Vigna. Efficient detection of split personalities in malware. In *Proc. of Network and Distributed System Security Symposium (NDSS)*, 2010.

[3] Black Duck Open Hub. Chromium open source project. Online https://www.openhub.net/p/chrome/analyses/latest/languages_summary, May 2020.

[4] D. Canali, M. Cova, G. Vigna, and C. Kruegel. Prophiler: a fast filter for the large-scale detection of malicious web pages. In *Proc. of the International World Wide Web Conference (WWW)*, 2011.

[5] ChromeDevTools. Chrome devtools protocol. Online https://chromedevtools.github.io/devtools-protocol/, May 2020.

[6] M. Cova, C. Kruegel, and G. Vigna. Detection and analysis of drive-by-download attacks and malicious javascript code. In *Proc. of the International World Wide Web Conference (WWW)*, 2010.

[7] C. Curtsinger, B. Livshits, B. G. Zorn, and C. Seifert. Zozzle: Fast and precise in-browser javascript malware detection. In *Proc. of USENIX Security Symposium*, 2011.

[8] CVE Details. CVE-2018-6140. Online https://www.cvedetails.com/cve/CVE-2018-6140/, Jan. 2019.

[9] CVE Details. CVE-2019-11708. Online https://www.cvedetails.com/cve/CVE-2019-11708/, July 2019.

[10] CVE Details. CVE-2019-11752. Online https://www.cvedetails.com/cve/CVE-2019-11752/, Sept. 2019.

[11] CVE Details. CVE-2019-5789. Online https://www.cvedetails.com/cve/CVE-2019-5789/, May 2019.

[12] ECMA International. Ecmascript 2019 language specification. Edition 10, 2019.

[13] Electric Apps. Vault antitheft. Online https://apps.shopify.com/vault-antitheft-protection-app, May 2020.

[14] A. Fass, R. P. Krawczyk, M. Backes, and B. Stock. Jast: Fully syntactic detection of malicious (obfuscated) javascript. In *Proc. of Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2018.

[15] A. Fass, M. Backes, and B. Stock. Jstap: a static pre-filter for malicious javascript detection. In *Proc. of Annual Computer Security Applications Conference (ACSAC)*, 2019.

[16] J. M. Fernández. JavaScript AntiDebugging Tricks. Online https://x-c3ll.github.io/posts/javascript-antidebugging/, Feb. 2018.

[17] Google Developers. Chrome devtools. Online https://developers.google.com/web/tools/chrome-devtools, Sept. 2019.

[18] Google Git. Web page replay. Online https://chromium.googlesource.com/catapult/+/HEAD/web_page_replay_go/, May 2020.

[19] guya. How to know when chrome console is open. Online https://blog.guya.net/2014/06/20/how-to-know-when-chrome-console-is-open/, June 2014.

[20] G. Ho, D. Boneh, L. Ballard, and N. Provos. Tick tock: building browser red pills from timing side channels. In *Proc. of USENIX Workshop on Offensive Technologies (WOOT)*, 2014.

[21] L. Invernizzi, P. M. Comparetti, S. Benvenuti, C. Kruegel, M. Cova, and G. Vigna. Evilseed: A guided approach to finding malicious web pages. In *Proc. of IEEE Symposium on Security and Privacy*, 2012.

[22] J. Jueckstock and A. Kapravelos. Visiblev8: In-browser monitoring of javascript in the wild. In *Proc. of Internet Measurement Conference (IMC)*, 2019.

[23] S. Kaplan, B. Livshits, B. Zorn, C. Siefert, and C. Curtsinger. "nofus: Automatically detecting"+ string. fromcharcode (32)+" obfuscated". tolowercase ()+" javascript code. *Technical report, Technical Report MSR-TR 2011–57, Microsoft Research*, 2011.

[24] A. Kapravelos, M. Cova, C. Kruegel, and G. Vigna. Escape from monkey island: Evading high-interaction honeyclients. In *Proc. of Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2011.

[25] A. Kapravelos, Y. Shoshitaishvili, M. Cova, C. Kruegel, and G. Vigna. Revolver: An automated approach to the detection of evasive web-based malware. In *Proc. of USENIX Security Symposium*, 2013.

[26] A. Kapravelos, C. Grier, N. Chachra, C. Kruegel, G. Vigna, and V. Paxson. Hulk: Eliciting malicious behavior in browser extensions. In *Proc. of USENIX Security Symposium*, 2014.

[27] K. Kim, I. L. Kim, C. H. Kim, Y. Kwon, Y. Zheng, X. Zhang, and D. Xu. J-force: Forced execution on javascript. In *Proc. of the International World Wide Web Conference (WWW)*, 2017.

[28] D. Kirat, G. Vigna, and C. Kruegel. Barecloud: bare-metal analysis-based evasive malware detection. In *Proc. of USENIX Security Symposium*, 2014.

[29] C. Kolbitsch, B. Livshits, B. Zorn, and C. Seifert. Rozzle: De-cloaking internet malware. In *Proc. of IEEE Symposium on Security and Privacy*, 2012.

[30] R. K. Konoth, E. Vineti, V. Moonsamy, M. Lindorfer, C. Kruegel, H. Bos, and G. Vigna. Minesweeper: An in-depth look into drive-by cryptocurrency mining and its defense. In *Proc. of ACM Conference on Computer and Communications Security (CCS)*, 2018.

[31] D. Kumar, Z. Ma, Z. Durumeric, A. Mirian, J. Mason, J. A. Halderman, and M. Bailey. Security challenges in an increasingly tangled web. In *Proc. of the International World Wide Web Conference (WWW)*, 2017.

[32] T. Lauinger, A. Chaabane, S. Arshad, W. Robertson, C. Wilson, and E. Kirda. Thou shalt not depend on me: Analysing the use of outdated javascript libraries on the web. In *Proc. of Network and Distributed System Security Symposium (NDSS)*, 2017.

[33] V. Le Pochat, T. Van Goethem, S. Tajalizadehkhoob, M. Korczyński, and W. Joosen. Tranco: A research-oriented top sites ranking hardened against manipulation. In *Proc. of Network and Distributed System Security Symposium (NDSS)*, 2019.

[34] M. Lindorfer, C. Kolbitsch, and P. M. Comparetti. Detecting environment-sensitive malware. In *Proc. of International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2011.

[35] G. Maisuradze, M. Backes, and C. Rossow. Dachshund: digging for and securing against (non-) blinded constants in jit code. In *Proc. of Network and Distributed System Security Symposium (NDSS)*, 2017.

[36] S. Matic, G. Tyson, and G. Stringhini. Pythia: a framework for the automated analysis of web hosting environments. In *Proc. of the International World Wide Web Conference (WWW)*, 2019.

[37] MDN Web Docs. performance.now(). Online https://developer.mozilla.org/en-US/docs/Web/API/Performance/now, May 2020.

[38] N. Miramirkhani, M. P. Appini, N. Nikiforakis, and M. Polychronakis. Spotless sandboxes: Evading malware analysis systems using wear-and-tear artifacts. In *Proc. of IEEE Symposium on Security and Privacy*, 2017.

[39] M. Musch, M. Steffens, S. Roth, B. Stock, and M. Johns. Scriptprotect: mitigating unsafe third-party javascript practices. In *Proc. of ACM Asia Conference on Computer and Communications Security (ASIA CCS)*, 2019.

[40] M. Musch, C. Wressnegger, M. Johns, and K. Rieck. New kid on the web: A study on the prevalence of webassembly in the wild. In *Proc. of Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2019.

[41] J. H. Odvarko. Saying goodbye to firebug. Online https://hacks.mozilla.org/2017/10/saying-goodbye-to-firebug/, Oct. 2017.

[42] P. H. Phung, D. Sands, and A. Chudnov. Lightweight self-protecting javascript. In *Proc. of ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2009.

[43] K. Rieck, T. Krueger, and A. Dewald. Cujo: efficient detection and prevention of drive-by-download attacks. In *Proc. of Annual Computer Security Applications Conference (ACSAC)*, 2010.

[44] Sansec. Digital skimmer runs entirely on Google, defeats CSP. Online https://sansec.io/research/skimming-google-defeats-csp, June 2020.

[45] S. Sarker, J. Jueckstock, and A. Kapravelos. Hiding in plain site: Detecting javascript obfuscation through concealed browser api usage. In *Proc. of Internet Measurement Conference (IMC)*, 2020.

[46] T. Serafim and T. Kachalov. JavaScript Obfuscator Tool. Online https://obfuscator.io/, Dec. 2020.

[47] Sindresorhus. devtools-detect. Online https://github.com/sindresorhus/devtools-detect, July 2020.

[48] P. Skolka, C.-A. Staicu, and M. Pradel. Anything to hide? studying minified and obfuscated code in the web. In *Proc. of the International World Wide Web Conference (WWW)*, 2019.

[49] StackOverflow. How to quickly and conveniently disable all console.log statements in my code? Online https://stackoverflow.com/questions/1215392/, July 2009.

[50] StackOverflow. Find out whether chrome console is open. Online https://stackoverflow.com/questions/7798748/, Oct. 2011.

[51] StackOverflow. How does Facebook disable the browser's integrated Developer Tools? Online https://stackoverflow.com/a/50674852, Feb. 2014.

[52] StackOverflow. How can I block F12 keyboard key. Online https://stackoverflow.com/questions/28575722/, Feb. 2015.

[53] M. Steffens, M. Musch, M. Johns, and B. Stock. Who's hosting the block party? studying third-party blockage of csp and sri. In *Proc. of Network and Distributed System Security Symposium (NDSS)*, 2021.

[54] B. Stock, B. Livshits, and B. Zorn. Kizzle: a signature compiler for detecting exploit kits. In *Proc. of Conference on Dependable Systems and Networks (DSN)*, 2016.

[55] Symantec. Webpulse site review. Online https://sitereview.bluecoat.com/, Dec. 2020.

[56] S. J. Tan, S. Bratus, and T. Goodspeed. Interrupt-oriented bugdoor programming: a minimalist approach to bugdooring embedded systems firmware. In *Proc. of Annual Computer Security Applications Conference (ACSAC)*, 2014.

[57] T. Urban, M. Degeling, T. Holz, and N. Pohlmann. Beyond the front page: Measuring third party dynamics in the field. In *Proc. of the International World Wide Web Conference (WWW)*, 2020.

[58] A. Vasudevan and R. Yerraballi. Cobra: Fine-grained malware analysis using stealth localized-executions. In *Proc. of IEEE Symposium on Security and Privacy*, 2006.

[59] Wayback Machine. Spotify. Online https://web.archive.org/web/20180301010204/https://www.spotify.com/us/, Mar. 2018.

[60] ww. Anti anti-debugger. Online https://greasyfork.org/en/scripts/32015-anti-anti-debugger/code, Aug. 2017.

[61] W. Xu, F. Zhang, and S. Zhu. The power of obfuscation techniques in malicious javascript code: A measurement study. In *2012 7th International Conference on Malicious and Unwanted Software*, 2012.

[62] W. Xu, F. Zhang, and S. Zhu. Jstill: mostly static detection of obfuscated malicious javascript code. In *Proc. of ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2013.