Measuring and Disrupting Anti-Adblockers Using Differential Execution Analysis

Shitong Zhu*, Xunchao Hu†, Zhiyun Qian*, Zubair Shafiq‡ and Heng Yin*
*University of California, Riverside
Email: shitong.zhu@email.ucr.edu, zhiyunq@cs.ucr.edu, heng@cs.ucr.edu
†Syracuse University, Email: xhu31@syr.edu
‡University of Iowa, Email: zubair-shafiq@uiowa.edu

***Abstract*—Millions of people use adblockers to remove intrusive and malicious ads as well as protect themselves against tracking and pervasive surveillance. Online publishers consider adblockers a major threat to the ad-powered "free" Web. They have started to retaliate against adblockers by employing anti-adblockers which can detect and stop adblock users. To counter this retaliation, adblockers in turn try to detect and filter anti-adblocking scripts. This back and forth has prompted an escalating arms race between adblockers and anti-adblockers.**

We want to develop a comprehensive understanding of anti-adblockers, with the ultimate aim of enabling adblockers to bypass state-of-the-art anti-adblockers. In this paper, we present a differential execution analysis to automatically detect and analyze anti-adblockers. At a high level, we collect execution traces by visiting a website with and without adblockers. Through differential execution analysis, we are able to pinpoint the conditions that lead to the differences caused by anti-adblocking code. Using our system, we detect anti-adblockers on 30.5% of the Alexa top-10K websites which is 5-52 times more than reported in prior literature. Unlike prior work which is limited to detecting visible reactions (*e.g.,* warning messages) by anti-adblockers, our system can discover attempts to detect adblockers even when there is no visible reaction. From manually checking one third of the detected websites, we find that the websites that have no visible reactions constitute over 90% of the cases, completely dominating the ones that have visible warning messages. Finally, based on our findings, we further develop JavaScript rewriting and API hooking based solutions (the latter implemented as a Chrome extension) to help adblockers bypass state-of-the-art anti-adblockers.

I. Introduction

The ad-powered Web keeps most online content and services "free". However, online advertising has raised serious security and privacy concerns. Attackers have repeatedly exploited ads to target malware on a large number of users [48, 55]. Advertisers track users across the Web without providing any transparency or control to users [30, 32, 39, 41, 47]. The popularity of adblockers is also rising because they provide a clean and faster browsing experience by removing excessive and intrusive ads.

Network and Distributed Systems Security (NDSS) Symposium 2018
18-21 February 2018, San Diego, CA, USA
ISBN 1-1891562-49-5
http://dx.doi.org/10.14722/ndss.2018.23331
www.ndss-symposium.org

Millions of users worldwide now use adblockers [51] that are available as browser extensions (e.g., Adblock, Adblock Plus, and uBlock) and full-fledged browsers (e.g., Brave and Cliqz). Even Chrome has now included a built-in adblocker in its experimental version — Chrome Canary [7]. According to PageFair [26], 11% of the global Internet population is blocking ads as of December 2016. A recent study by comScore [40] reported that 18% of Internet users in the United States use adblockers. Moreover, the prevalence of adblockers is much higher for certain locations and demographics. For instance, approximately half of 18-34 year old males in Germany use adblockers.

The online advertising industry considers adblockers a serious threat to their business model. Advertisers and publishers have started using different countermeasures against adblockers. First, some publishers such as Microsoft and Google have enrolled in the so-called *acceptable ads program* which whitelists their ads. While small publishers can enroll in the program for free, medium- and large-sized publishers have to pay a significant cut of their ad revenue to enroll. Second, some publishers — most notably Facebook — are manipulating ads that are harder to remove by adblockers [13]. However, adblockers have been reasonably quick to catch up and adapt their filtering rules to block these ads as well [3]. Third, many publishers have implemented anti-adblockers — JavaScript code that can detect and/or respond to the presence of adblockers at client-side. While Facebook is the only reported large publisher that has tried to use the second approach, a recent measurement study of Alexa top-100K websites [42] reported that the third strategy of anti-adblocking is more widely employed. Common anti-adblockers force users to whitelist the website or disable their adblocker altogether.

We want to develop a comprehensive understanding of anti-adblockers, with the ultimate aim of enabling adblockers to be resistant against anti-adblockers. To this end, we propose a system based on differential execution analysis to automatically detect anti-adblockers. Our key idea is that when a website is visited with and without adblocker, the difference between the two JavaScript execution traces can be safely attributed to anti-adblockers. We use differential execution analysis to precisely identify the condition(s) used by anti-adblockers to detect adblockers which helps us understand how they operate. The experimental evaluation against a ground-truth labeled dataset shows that our system achieves 87% detection rate with no false positives.

We employ our system on Alexa top-10K list and are able to detect anti-adblockers on 30.5% websites. From manually checking one third (1000) of these detected websites, we find that the number of websites that have no visible reactions versus is an order of magnitude higher than the ones that have visible warning messages. We not only discover anti-adblocking walls (warnings) invoked after adblockers are detected, but also websites that silently detect adblockers and subsequently either switch ads [49] or report adblock statistics to their back-end servers. Our ability to detect visible as well as silent anti-adblockers allows us to detect 5-52 times more anti-adblockers than reported in prior literature.

We further leverage our systematic detection of anti-adblockers using differential execution analysis to help adblockers evade state-of-the-art anti-adblockers. First, since we can precisely identify the branch divergence causing adblock detection, we propose to use JavaScript rewriting to force the outcome of a branch statement for avoiding anti-adblocking logic. Second, we propose to use API hooking in a browser extension to intercept and modify responses to hide adblockers. The evaluation shows that our current proof-of-concept implementations, which still have room for engineering improvements, are able to successfully evade a vast majority of the state-of-the-art anti-adblockers.

II. Background and Related Work

Adblockers rely on manually curated filter lists to block ads and/or trackers. EasyList and EasyPrivacy are the two most widely used filter lists to block ads and trackers, respectively. The filter lists used by adblockers contain two types of rules in the form of regular expressions. First, HTTP filter rules generally block HTTP requests to fetch ads from known third-party ad domains. For example, the first filter rule below blocks all third-party HTTP requests to doubleclick.com. Second, HTML filter rules generally hide HTML elements that contain ads. For example, the second filter rule below hides the HTML element with ID `banner_div` on `aol.com`.

```
||doubleclick.com^$third-party
aol.com###banner_div
```

It is noteworthy that filter lists may contain tens of thousands of filter rules that together block ads/trackers on different websites. At the time of writing, EasyList contains more than 63K filter rules and EasyPrivacy contains more than 13K filter rules. The filter lists are maintained by a group of volunteers through informal crowdsourced feedback from users [16]. As expected, adding new rules or removing redundant rules in the filter lists is a laborious manual process and is prone to errors that often result in site breakage [20].

Adblocking browser extensions (e.g., Adblock, AdBlock Plus, uBlock) and full-fledged browsers (e.g., Brave, Cliqz) are used by millions of mobile and desktop users around the world. According to PageFair [26], 11% of the global Internet population is blocking ads as of December 2016. Adblocking results in billions of dollars worth of lost advertising revenue for online publishers. Therefore, online publishers are fast adopting different technical measures to counter adblockers.

First, publishers can manipulate ad delivery to evade filter lists. For example, publishers can keep changing domains that serve ads or HTML element identifiers [13, 52] to bypass filter list rules. Such manipulation forces filter list authors to update filter list rules very frequently, making the laborious process even more challenging. To address this problem, researchers [31] proposed a method based on network traffic analysis (e.g., identify ad-serving domains) for updating HTTP filter list rules automatically. This method, however, does not address HTML manipulation by publishers (like recently done by Facebook [13]). While adblockers have updated their filter rules to block Facebook ads for now [3], Facebook can continuously manipulate their HTML to circumvent new filter rules. To address this challenge, researchers [50] proposed a perceptual adblocking method for visually identifying and blocking ads based on optical character recognition and fuzzy image matching techniques. The key idea behind the perceptual adblocking method is that ads are distinguishable from organic content due to government regulations (e.g., FTC [12]) and industry self-regulations (e.g., AdChoices [27]). Researchers [50] reported that perceptual adblocking fully addresses the ad delivery manipulation problem.

Second, publishers try to detect and stop adblockers using anti-adblocking scripts. At a high level, anti-adblockers check whether ads are correctly loaded to detect the presence of adblockers [45]. After detecting adblockers, publishers typically ask users to disable adblockers altogether or whitelist the website. Some publishers also ask users for donation or paid subscription to support their operation. Prior work [42] showed that 686 out Alexa top-100K websites detect and visibly react to adblockers on their homepages.

Adblockers try to circumvent anti-adblockers by removing JavaScript code snippets or by hiding intrusive adblock detection notifications. To this end, adblockers again rely on crowdsourced filter lists such as Anti-Adblock Killer [11] and Adblock warning removal list [18]. First, HTTP filter list rules block HTTP requests to download anti-adblock scripts. For example, the first filter rule below blocks URLs to download `blockadblock.js`. Second, HTML filter rules hide HTML elements that contain adblock detection notifications. For example, the second filter rule below hides the HTML element with ID `ad_block_msg` on `zerozero.pt`.

```
/blockadblock.js$script
zerozero.pt###ad_block_msg
```

Prior work [35] showed that these filter lists targeting anti-adblockers are maintained in an ad-hoc manner and are always playing catchup. Publishers can evade these filter lists by either serving anti-adblocking scripts from first-party or by incorporating them in the base HTML. Moreover, some third-party anti-adblocker services deploy fairly sophisticated hard-to-defeat techniques to detect adblockers [42]. In sum, adblockers currently are simply not effective against anti-adblocking. For example, prior work showed that adblockers remove less than 20% of the adblock detection warning messages shown by anti-adblockers [42].

Prior research has proposed several solutions to detect and circumvent anti-adblockers. One solution is to fingerprint third-party anti-adblocking scripts using static code signatures [50]. However, JavaScript fingerprinting is not scalable if code signatures are not automatically derived. Manual JavaScript code analysis is much more challenging compared to identifying ad-related URL or HTML elements. So even if the effort is crowdsourced, it is unlikely to catch up with the quickly changing landscape of anti-adblockers. Worse, even simple obfuscation techniques such as code minimization will likely substantially increase the manual effort to rebuild these signatures. Similarly, the approach in [45] also bears the above limitations even when they attempt to reduce the amount of manual work by analyzing only commonly appeared scripts in clusters.

Researchers have proposed automated static JavaScript code analysis techniques (based on syntactic and structural analysis) for malicious JavaScript detection [28, 37, 46]. Prior work has borrowed these techniques to automatically extract signatures for tracker [34] and anti-adblock [35] detection. Ikram et al. [34] proposed syntactic and semantic JavaScript static code features with one-class SVMs to detect tracking JavaScript programs. Iqbal et al. [35] proposed syntactic JavaScript static code features with SVMs to detect anti-adblocking scripts. However, it is challenging for static code analysis techniques to truly capture JavaScript behavior, which is dynamic and can be easily obfuscated.

To aid future research on the arms race between adblockers and anti-adblockers, in this paper we propose a dynamic code analysis approach to systematically characterize anti-adblock behavior on a large scale. Our key idea is that differential execution analysis (i.e., with and without adblocker) will reveal anti-adblockers trying to detect adblockers. Our proposed approach has three major advantages over prior work. First, it allows us to detect anti-adblockers without prior knowledge about their behavior. Second, it is robust against simple (e.g., code minimization) and more advanced (e.g., runtime code generation) code obfuscation techniques. Finally, unlike prior work [42] that only detects visible reactions by anti-adblockers, it can identify whether there is an attempt to detect adblockers even if there is no visible reaction.

III. Problem Formulation & System Overview

An anti-adblocker script consists of two main components: (1) *trigger*, which detects the presence of adblockers (*e.g.,* by checking the absence of an ad); (2) *reaction*, which can display the adblock detection message and/or simply report the results to a backend server. As discussed earlier, prior work [35, 42, 43, 45] has reported a wide range of strategies used by different publishers to detect and react to adblockers. Some websites use simple anti-adblock scripts, served from first-party domains, to check display-related attributes of ads for detecting adblockers. Others use more sophisticated third-party anti-adblocking services that may employ active baits, do continuous detection, or use cookies track users' adblock detection status across different visits. Therefore, it is challenging to automatically detect diverse anti-adblocking behaviors used by different publishers and third-party anti-adblocking services. The state-of-the-art solution in prior literature [42] uses machine learning to automatically detect anti-adblockers that exhibit visible reactions. However, this solution can largely underestimate the ubiquitous of anti-adblockers because it cannot detect silent anti-adblockers or subtle reactions (more details in §V-B). In this section, we formulate the anti-adblock detection problem. We then present the blueprint for a

```
<div class="banner_ads">&nbsp;</div>
<script>
var ads = document.getElementsByClassName('
    banner_ads'),
ad  = ads[ads.length - 1];
if (!ad || ad.innerHTML.length == 0 || ad.
    clientHeight === 0)
  alert("We've detected an ad blocker running on
      your browser." + ...);
}
</script>
```

Fig. 1: A simple anti-adblocking example from a real website

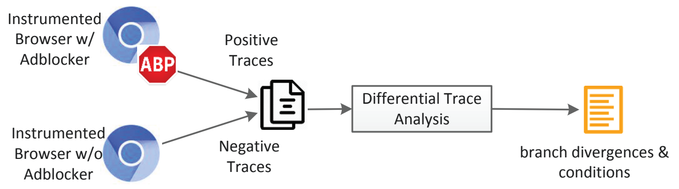

Instrumented Browser w/ Adblocker
ABP
Positive Traces
Differential Trace Analysis
Instrumented Browser w/o Adblocker
Negative Traces
branch divergences & conditions

Fig. 2: System overview

program analysis based approach to automatically detect anti-adblockers.

We start by providing a motivating example to introduce our key ideas. Our key observation is that a website employing anti-adblocking would have a different JavaScript execution trace if it is loaded with adblocker (positive trace) as compared to without adblocker (negative trace). To illustrate the point, we consider a simple real-world anti-adblocking example in Figure 1. We note that the anti-adblocking script embeds an empty `div` whose class is set to `banner_ads` which is a known class type that will trigger blocking. The code then simply checks whether the ad frame is blocked to determine the presence of adblockers. Specifically, when an adblocker is used, the `ad` frame will become undefined, and its `length` and `height` values will be zero. The `if` condition in the anti-adblocking script checks the values of these attributes. If the script detects that the value of either of these attributes is zero, it detects adblocker and reacts by displaying an alert and subsequently redirecting the user to a subscription page (code omitted for brevity). Without adblocker, the `if` condition will not be satisfied.

It is noteworthy that the JavaScript execution trace will differ under A/B testing. More specifically, since we are able to control two execution environments (*i.e.,* browser instances) where the only difference is the presence/absence of an adblocker, the execution trace difference can be safely attributed to adblocking (barring some noise issues which are discussed §IV-B). Equipped with the knowledge of the trace difference, we can then track the origin of the objects that are checked in the branch statements (*e.g.,* which objects/variables and what attributes), as well as understand the subsequent reactions. Next, we provide more details of the differential execution analysis approach.

Given two traces, we define two types of execution differences [36]: *flow differences* and *value differences*. A flow difference is caused by control flow divergence in the two executions (i.e., with and without adblocker). A value difference is caused when a variable in any statement has different values in the two executions. Note that anti-adblockers have to execute some additional statements (after an adblocker is detected) such as displaying warning messages or sending statistics to their backend servers. Thus, we can rely on control flow differences to detect anti-adblockers without needing to track value differences. A recent study [42] also reported that most anti-adblockers manifest themselves through conditional branches. Therefore, in this work, we consider only the *flow differences* in our differential trace analysis and leave *value differences* (which may be required for more advanced anti-adblockers) for future work.

Figure 2 illustrates the overview of our proposed system of differential trace analysis. First, we instrument the open-source Chromium [25] browser to collect execution traces. Since we focus on the control flow of JavaScript, we collect traces for all branch statements along with the call stack information which is needed for trace alignment (discussed later in §IV). We discuss other details of the instrumentation later in §IV-A. After we collect the execution traces, we feed them to the *differential execution analysis* to identify the diverging branches between the positive trace (with adblocker) and the negative trace (without adblocker). The differential execution analysis outputs a list of branch divergences and the conditions checked in those branch statements.

The produced result not only allows one to affirm the presence of anti-adblocking logic but also helps us understand how they operate. As we will show later in §V, we conduct both large-scale and small-scale evaluation and analysis of the identified anti-adblocking scripts. Finally, in §VI, we show how one can apply the learned knowledge and use it against anti-adblockers.

IV. Differential Execution Analysis

In this section, we describe the framework, building blocks, as well the methodology for differential execution analysis (branch divergence discovery). Overall, we need to select one or more adblocker extensions for A/B testing, instrument Chromium, and conduct the differential execution analysis.

Adblocker choice. As the A/B testing requires the collection of the Javascript execution trace with and without an adblocker, we need to select an adblocker extension. We choose Adblock as it is one of the most popular. It is also possible to use Adblock Plus or uBlock, as the way they operate is exactly the same — HTTP filters and HTML element hiding. In fact, they share the same basic set of filter lists and we confirm that they yield the same result from our differential execution analysis.

A. Chromium Instrumentation

Prior work has proposed several instrumentation approaches to collect JavaScript execution traces. These include JavaScript rewriting [54], JavaScript debugger interface [33], and JavaScript engine-based approaches. In this paper, we use the last approach which modifies the JavaScript engine to output the execution traces. We prefer this approach because it does not require any change to JavaScript code itself. Moreover, our approach is transparent which makes it much more challenging for anti-adblocking scripts to detect that they are being instrumented and possibly change their behavior.

We instrument the JavaScript engine for Chromium (V8 [22]). V8 generates an abstract syntax tree (AST) for every function. The ASTs are then compiled into native code (also called Just-In-Time code). Our instrumentation is embedded into the native code generation process. Our instrumentation collects the source map information (*e.g.,* the offset of the statement located within a script) as well as the JavaScript statement information (*e.g.,* whether a true/false branch is taken) for every statement of interest. As we discuss later, we instrument only a subset of statements that are pertinent to anti-adblockers. The information is stored into inlined variables. Before emitting the native code for the statements, we modify the JIT engine to emit stub code, which at runtime will access the inlined variables to record the executed JavaScript statements. The source map information is used as the ID of the executed statement (which is later required for trace alignment). The JavaScript statement information simply records the branch outcomes, so that we can perform the differential trace analysis to identify any branch divergence or flip between two different execution runs (with and without adblocker).

Since we only monitor control flow differences to detect anti-adblockers, we monitor all branch statements to record the control flow part of the execution trace. In JavaScript, the branch statements include `if/else` (including `else if`), `switch/case`, and conditional/ternary operators (`condition?expr1:expr2`), `for/while` loop, and `try/catch` for exception handling. In addition, there are implicit branching expressions such as `A && B;` where the outcome of `A` in fact determines whether `B` will be executed (*i.e.,* if A is false, B will not be executed). We currently monitor only the `if/else` and conditional/ternary operators, which are reported to be most commonly used by anti-adblockers [42]. For trace alignment (see §IV-B), we also record all function call/ret statements and call stack information for all branch statements to include their calling context.

B. Branch Divergence Discovery

We now explain our approach to discover branch divergences. We visit a website to collect two sets of execution traces with (positive trace) and without (negative trace) the adblocker. We then analyze their *flow differences* between the positive and negative traces.

Adblockers can be used in different ways based on their filter list configurations. The default configuration on Adblock [15] includes two blacklists (EasyList to remove ads and Adblock Warning Removal List to remove adblock detection responses) and one whitelist (Acceptable Ads List to allow some ads). We choose a configuration that can maximize the likelihood of detecting anti-adblockers (which is also what was used in [42]). We first disable the Acceptable Ads List, not allowing websites to show even the whitelisted ads. Then we disable the Adblock Warning Removal List, allowing intrusive notifications by some anti-adblockers. Finally we further remove the sections of rules in Easylist that are specifically crafted against anti-adblockers. It is worth mentioning that Adblock, as shown in a recent study [42], does not actually do a good job in defending against anti-adblockers, but we disable these capabilities nevertheless to get a more complete picture of anti-adblockers in the wild.

We need to align a positive trace and a negative trace to discover branch divergences. Trace alignment is a well-research issue for comparing different execution traces of the same program [36, 53]. To accurately align two execution traces, we can assign each execution point an execution index while taking into account the program's nesting structure and the caller/callee relationship. We opt for the use of call stack information as the calling context for each recorded statement. More specifically, two execution traces are said to be aligned only when the following two conditions hold simultaneously:

1) the call stacks of all statements of both traces match perfectly; and
2) the identifiers of all statements (represented by their offset) of both traces match perfectly.

The key challenge in aligning JavaScript execution traces is that JavaScript runtime has a unique concurrency model [44]. More specifically, standard JavaScript execution for each web page is single-threaded and event driven. This means that each event is processed independently and completely before any other event is processed. Instead of generating a single sequence of trace for each page visit, we are now forced to consider the code executed to handle all events (*e.g.,* on successfully loading an external resource) on different sub-traces because they all start from the beginning of the event loop.

We address this challenge by aligning sub-traces in a disjoint manner. More specifically, we slice a trace into sub-traces by recording all the function call/ret statements. Whenever a ret statement is encountered where its call stack is empty (*i.e.,* an iteration of the event loop is about to end), we know that it is the end of a sub-trace. We align two sets of positive and negative sub-traces separately in a pairwise manner. The number of alignments is on the order of O($n \times m$) where n is the number of positive sub-traces and m is the number of negative sub-traces.

Given a pair of aligned positive and negative traces, we next attempt to discover and locate branch divergences. Basically, given a positive and a negative trace, we record all encountered branches (with the same call stack at the same offset) with opposite outcomes. In the example shown in Figure 1, the positive and negative traces will simply be (3, 4, 5-true, 6) and (3, 4, 5-false) respectively — the numbers here are statement identifiers. We can therefore confirm the branch divergence at statement 5. The key technical challenge we need to address is that a script can generate different execution traces due to external factors (e.g., time) or other sources of randomness. We need to cater for this to avoid mistakenly attributing branch divergences to adblockers which are actually completely unrelated.

Handling execution noises. The following example illustrates this problem. Two different runs of the same code can possibly produce two different execution traces – one with coinFlip() returning true and the other with coinFlip() returning false. If the two runs happen to occur when an adblocker is on and off respectively, we will mistakenly think that the branch divergence is due to an adblocker.

```
function coinFlip() {
    return Math.floor(Math.random() * 2);
}
if (coinFlip()) {
    // displayDynamicContent
}
```

To counter such "noises", we generate redundant positive and negative traces with the goal of identifying unrelated divergences. Based on our pilot experiments, we decide to generate *three* runs of positive traces and *three* runs of negative traces to detect and eliminate unrelated divergences.

Note that even though not incorporated yet, our system can generate these traces from the same webpage (by simply forcing the same exact webpage and scripts to be reloaded), thus avoiding the case where different runs encounter two different versions of webpages or scripts. This means that even if a website intentionally tries to deliver a different webpage or script every time [52], we are still able to analyze one specific version and determine if anti-adblocker is present.

Due to the nature of JavaScript runtime, we are unable to handle implicit branching caused by callbacks. The following example illustrates implicit branching. Depending on whether the URL is successfully loaded, `success()` and `error()` will be invoked respectively. It is important to note that the URL is pointing to an advertisement; therefore, if it fails to load, it is indicative that an adblocker is present (and `error()` will be invoked accordingly in reaction to it). However, since `success()` and `error()` are both invoked at the beginning of the event loop (*i.e.,* their call stack is empty), we are unable to correctly align the two sub-traces and therefore will not discover the branch divergence. To address this issue, we will need to consider all callback functions for a same event (URL fetch) as implicit branch statements. This means that if we see `success()` in a positive run but `error()` in a negative run, we can attribute it to a branch divergence. Our system currently does not support this uncommon special case. We plan to address this limitation in our future revisions.

```
$.ajax({
    type: "GET",
    url: "some_ads_url",
    success: function(){  ## display ads   },
    error: function(XMLHttpRequest, textStatus,
        errorThrown) {
        ## adblocker detected!
    }
});
```

V. Evaluation

We first evaluate the timing requirements of our differential execution approach. Recall that we visit each website three times with adblocker and three times without adblocker. For each visit, we wait for 20 seconds before we stop the trace collection to ensure the website finishes loading (Chromium loads websites slower with our instrumentation). In addition, it takes less than a minute to perform the differential trace analysis. Overall we need about 3 minutes per website on average to run our differential execution analysis. Given a server with 32 cores, we need a little over 14 hours to process ten thousand websites (assuming we schedule one Chromium instance per core). Therefore, our current implementation is efficient enough to analyze Alexa top-10K websites on a daily basis using only one server.

We next evaluate the accuracy of anti-adblocker detection on a small and large scale data set. We have constructed an anonymous project website at https://sites.google.com/view/antiadb-proj/ to show some detailed cases studies of anti-adblocking websites and scripts.

A. Small-Scale Ground Truth Analysis

For positive examples, we pick the list of 686 websites that were reported to use anti-adblockers in February 2017 [42]. Since some websites may no longer be using anti-adblockers now (August 2017), we manually re-analyze these 686 websites and shortlist 428 websites which still visibly react to adblockers. During manual screening, we at each loaded mainpage manually for around 30s each without any clicking (but scrolling down is also performed to be able to catch minor warning messages inserted in the middle of the page). For negative examples, we manually select 100 websites (*e.g.,* Wikipedia, academic, and non-profit websites) that do not contain any ads. Thus, these websites do not trigger adblockers and also do not contain anti-adblockers.

We evaluate the accuracy of our system in detecting anti-adblockers on the aforementioned manually labeled set of websites. Our system achieves 86.9% (372/428) true positive rate and 0% false positive rate. For the 100 labeled negative websites, our system did not mistakenly detect any as using anti-adblockers. For the 56 false negative cases, we identify that three main reasons: (1) incomplete instrumentation of branch statements; (2) inherent randomness in website loading, and (3) incomplete blocking of ads by the adblocker. We elaborate on these reasons below.

First, we note that websites can implement anti-adblock detection logic in JavaScript (or any other Turing-complete programming language for that matter) using many different constructs that may not be covered by our current implementation. For example, the presence of a bait object can be checked using

```
if (bait_is_absent()) {
  reaction_func()
}
```

ternary operators,

```
bait_is_present() ? do_nothing() : reaction_func()
```

callbacks,

```
<script onerror="reaction_func()" src="/bait.js"></
    script>
```

and `&&` operator.

```
bait_is_absent() && reaction_func()
```

Among these, our prototype implementation currently only covers the `if/then/else` clause and ternary operators while leaving out callbacks. Moreover, some solutions (*e.g.,* BlockAdblock) utilize `eval` to wrap their anti-adblocking logic represented as a string, which is not currently instrumented in our implementation. Finally, one website (expats.cz) implements anti-adblocking logic using non-control-flow paradigms (*e.g.,* using an array element to decide whether to trigger anti-adblock reaction). To tackle this issue, our system needs to trace *value difference* [36] as well.

Second, several websites seem to be impacted by different sources of randomness that can bypass our current implementation. These include (1) behavioral randomization and (2) content randomization. In behavioral randomization, a website randomly activates its anti-adblocking module which results in inconsistent positive/negative traces. Our system rules out such inconsistencies as noise. In content randomization, a website may change various page elements (*e.g.,* DOM/variable/bait names) across multiple runs, thereby breaking our trace alignment (e.g., our current implementation requires the same variable name). As mentioned earlier in §IV-B, this is not a fundamental limitation of differential trace analysis as we can force the same exact page to be loaded across multiple runs. However, it is much more challenging to deal with behavioral randomization, which we discuss in §VII.

Third, we also note a few special cases. For instance, one interesting case is that an anti-adblock warning message (initially invisible) is placed behind the real ad, and becomes visible when the ad in the front is blocked. In this case, no extra code is executed to conduct anti-adblocking but its effect is still preserved (it's arguable whether it should be considered an anti-adblocker). The best solution is probably to let the adblocker block the warning message as well.

We mentioned earlier that there are 428 out of 686 websites that have visible anti-adblockers. We are curious about the remaining 258 websites — could it be that they are simply performing anti-adblocking with no visible reactions? After running our system on these websites, it turns out that most of them are actually flagged as positive (with branch divergences under A/B testing). To understand if our results are correct, we conduct small-scale manual verification. Specifically, we look at the branch divergence and the triggered logic when an adblocker is detected. Interestingly, we indeed find many websites that perform adblocker detection with minimal or no visible reactions. We illustrate two interesting types of them from two websites memeburn.com and englishforum.ch in Figure 3.

1. Switching ad sources: As shown in Figure 3(a), `memeburn.com` uses a first-party anti-adblock script. Upon detecting an adblocker, it immediately replaces one of its banner ads with a gif image (see Figure 4 for the visual differences). Interestingly, instead of switching to external ads, in this case the website has chosen to advertise its own services (about its tech news podcasts).

2. Collecting adblocker usage statistics: We find `englishforum.ch` has a first-party script that detects adblockers yet does not exhibit any visible reaction. As shown in Figure 3(b), the variable `blockStatus` indicates the presence of adblockers, which is set to true as soon as an ad frame is found missing. The `ga()` function is of Google Analytics API that reports events back to the website owner. As we can see in this case, there are no additional reactions besides the silent recording of adblocker usage.

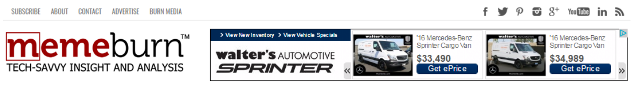
```
if( window.advertsAvailable === undefined ){
  //adblocker detected, show fallback
  jQuery('.replace-me').html('<img style="width:auto;height:auto;max-width:728px" id="testreplace" src="/images/Burn_sad.gif" width="728" height="90" alt="Please support us by allowing ads on our site"/>');
  jQuery('#testreplace').css('display','block');
}
```

(a) Switching ad sources upon detecting any adblocker

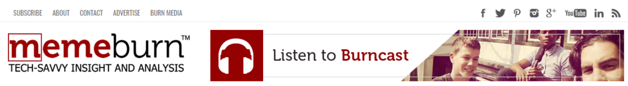
```
var blockStatus = 'Unblocked';
var ad = $('#adsense')[0];
if (!ad || ad.innerHTML.length == 0 || ad.clientHeight === 0) blockStatus = 'Blocked';
ga('send', 'event', 'Ad block JavaScript', blockStatus, 'Desktop', {nonInteraction: true});
ga('theLocalNetwork.send', 'event', 'Ad block JavaScript', blockStatus, 'Desktop', {nonInteraction: true});
```

(b) Reporting adblocker usage through Google Analytics

Fig. 3: Silent anti-adblocker (Left: memeburn.com. Right: englishforum.ch)


SUBSCRIBE ABOUT CONTACT ADVERTISE BURN MEDIA
memeburn
TECH-SAVVY INSIGHT AND ANALYSIS
walter's AUTOMOTIVE
SPRINTER
$33,490
$34,989
Get ePrice

(a) Original banner ads (shown when there is no adblockers)


SUBSCRIBE ABOUT CONTACT ADVERTISE BURN MEDIA
memeburn
TECH-SAVVY INSIGHT AND ANALYSIS
Listen to Burncast

(b) Replaced banner ads (shown when the original is blocked)

Fig. 4: Ad switching behavior on memeburn.com

B. *Large-Scale Analysis of Alexa Top-10K Websites*

We now conduct a large-scale in the wild evaluation of our system. Surprisingly, we are able to detect anti-adblockers on 30.5% on the Alexa top-10K websites. Our results point out that roughly one-third of the most popular websites are equipped with anti-adblockers. Our investigation shows that 1238 websites use only if/else type of branch divergences, 473 use only ternary-type divergences, and the remaining 1344 are detected to contain both divergences. This finding highlights that anti-adblockers implement their detection logic in several different ways. Thus, as we expand support for other types of branch statements in our implementation in future, our anti-adblock detection rate may further increase.

It is noteworthy that our results show that anti-adblockers are much more pervasive than previously reported in prior work [42, 45]. An earlier study [45] published in May 2016 reported that 6.7% of Alexa top-5K websites use anti-adblockers. Our anti-adblock detection results are approximately 5× more than theirs. Another study [42] conducted in February 2017 reported that 0.7% of Alexa top-100K websites use anti-adblockers. Our anti-adblock detection results are approximately 52× more than theirs. To better understand the discrepancy, we should reiterate that an anti-adblocker has at least two components: (1) adblocker detection and (2) subsequent reaction. The solution in [42] also leverages A/B testing but it aims to detect HTML changes caused by anti-adblockers. It makes the assumption that there will be a significant reaction (visible at the HTML level) after an adblocker is detected. However, as we have shown earlier, this assumption may not hold as many websites can have subtle or no visible changes at all while still having the ability to detect adblockers. The authors in [45] relied on manual analysis and may miss some anti-adblockers that do not have obvious keyword in the scripts (e.g., obfuscated). Moreover, the anti-adblock prevalence has likely increased [35] since more than a year ago [45] when the study was conducted. In contrast to prior work, our approach is *oblivious* to the reactions by anti-adblockers; instead, it essentially relies on catching the *adblocker detection* logic that is evident by the discovered branch divergence. Later we will sample a number of popular websites and scripts with manual inspection to validate our results.

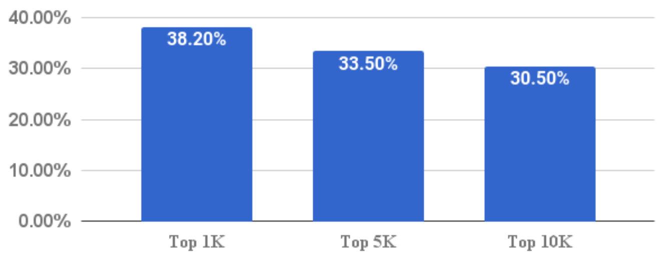


Fig. 5: Popularity of anti-adblockers by website ranking

In summary, our hypothesis is that a much larger fraction of websites than previously reported are "worried" about adblockers but many are not employing retaliatory actions against adblocking users yet. To verify the hypothesis, we manually inspected 1000 websites out of the 3000+ detected websites. Following the same inspection methodology described in §V-A, we find that there are only 66 (10 of them simply switch sources of the ads) websites that do have visible reactions and 934 that do not, which indeed represents a huge disparity. While it may be useful to conduct automated analysis of subsequent reactions (*e.g.,* whether they invoke APIs that have visual impact, or whether they send data over to network to log adblocker usage), we leave this as future work.

We now attempt to categorize the websites that use anti-adblockers in the following aspects. First, we are curious to see whether there's any correlation between their popularity and the likelihood of them deploying anti-adblockers. Figure 5 shows that the higher ranked websites are more likely to use anti-adblockers. This is somewhat counter-intuitive as most top websites do not actually have any visible reactions to adblocker users, leaving users the impression that they are not doing

Rank	Script Source	Detection Trigger[1]	Reaction	Count[2]
1	Google Analytics	unknown	unknown	614
2	Google DoubleClick	real ads	silent reporting	403
3	YouTube	real ads	silent reporting	311
4	Taboola	baits	silent reporting	144
5	PageFair	mixed baits + extension probing	silent reporting + local storage	95
6	Chartbeat	unknown	unknown	82
7	Mail.ru	baits	silent reporting	72
8	Addthis	unknown	unknown	61
9	Yandex	baits	silent reporting	57
10	Cloudflare	baits	silent reporting	51
11	Twitter	unknown	unknown	45
12	Criteo	baits	silent reporting + ads switching	32
13	Outbrain	baits	silent reporting + local storage	32

[1] All scripts check attributes of DOM elements as opposed to others.
[2] The number of websites that contain the script.
See cases for checks against other types of variables in §VI-B

TABLE I: Top origins of anti-adblocker scripts based on different sources

anything about adblockers. We find that many popular websites are passively collecting statistics (to evaluate what they should do). Second, our analysis of website categorization corroborate results reported in prior work [42, 45] that "news and media" websites are much more likely to use anti-adblockers. This is expected because online advertising is a key source of income for news and media websites.

We investigate the source of anti-adblocking scripts used by websites. More specifically, are they first-party vs. third-party scripts? Are there a small number of third-party scripts that are widely deployed by many websites? Our results show that there are 422 websites that use only first-party anti-adblocking scripts while 2219 websites use only third-party scripts (414 websites use both). This discrepancy shows that most websites choose to outsource anti-adblocking to dedicated third-party anti-adblocking service providers such as PageFair [24]. To better understand the small set of third-party anti-adblocking scripts, we aggregate their sources using the domain and URL information. Table I reports the third-party sources of the most popular anti-adblocking scripts. We note that analytics and ads scripts by Google are the most popular source of anti-adblocking scripts. As expected, we also note several other online advertising services such as Taboola and Outbrain using anti-adblockers.

To better understand popular third-party anti-adblocking scripts, we next investigate them using several different analysis approaches such as code base, network traffic, cookie content, probing etc. For instance, if silent reporting exists, the network traffic would contain at least some difference in the payload during A/B testing. Similarly, a different cookie value set during A/B testing can also support silent reporting. These anti-adblocking scripts are fairly challenging to analyze because they are large, complex, and often use obfuscation. It is also challenging to analyze some of them because they do not have visible reactions to adblocker detection. For example, 9 out of 13 most popular anti-adblocking scripts, which account for almost one-third of the Alexa top-10K websites that use anti-adblockers, detect adblockers silently.

Table I reports the detection mechanism and subsequent reactions of popular anti-adblockers. They all check attributes of DOM elements in certain way (as opposed to alternatives which are more common in less popular scripts. See §VI-B for details). Most of the checked DOM elements are baits and a number of them are real ads. Specifically, we are able to confirm `DoubleClick` detects adblockers by checking the height and length of ad-related objects and sends view-status ad requests to the back-end server. Scripts from `PageFair` and `Taboola` are also performing anti-adblocking, which is expected since both are known to provide anti-adblocking services [10, 14]. In particular, PageFair [42] attempts to craft a diverse set of baits and even probes into the extension folder [1]. The probing methodology is deprecated in newer browsers but still effective against older versions of Chrome [9] to detect adblockers. Both scripts from PageFair and Taboola silently report adblocking statistics. It is noteworthy that PageFair has two types of scripts: one is analytics which only collects adblocker usage; the other one has the ability to switch ad sources [1]. In our study, the analytics script is the one that showed up as top scripts, likely because it is a free service while the other is not [1].

Being the biggest adblock-analytics-tech player in the market, PageFair meticulously crafts a diverse set of baits to maximize its chances of detecting adblockers. Its analytics script `measure.min.js` creates six different baits in total, with two of them being the `<div>` elements and the rest as images/scripts. Then the blocking status of these six baits will be monitored and stored independently. Finally the script saves the status into a local cookie for future use, and also sends it to back-end server with multiple fields indicating the state of each bait and other statistics.

```
1 TRC.blocker.blockedState = TRC.blocker.
      getBlockedState(this.global["abp-detection-class
      -names"] || ["banner_ad", "sponsored_ad"])
2
3 getBlockedState: function(a) {
4   return a && this.isBlockDetectedOnClassNames(a) ?
        this.states.ABP_DETECTED : this.states.
        ABP_NOT_DETECTED
5 }
```

Fig. 6: Taboola's anti-adblocking script snippet

Next, we illustrate the anti-adblocking logic for Taboola, another big play in the field, in Figure 6. We can see that the key function is `getBlockedState()` on line 4. As we can see, multiple strings are passed as arguments. Notably, the "banner_ad" and "sponsored_ad" are two known element ids that are filtered by Easylist. `isBlockDetectedOnClassNames()` will create a DOM element for each string in the list. These elements serve

[1] Every extension in Chrome is organized in a folder with a globally unique extension ID assigned by Google as the folder name

as baits. Upon the detection of them being blocked, `isBlockDetectedOnClassNames()` returns true.

```
setTimeout(function() {
  c.tj(a);
  c.ba = (0 < c.gd).toString();
  d.h.log("AdBlock - finish long status check.
      adBlock = " + c.ba);
  c.af = !0;
  d.b.yh("OB-AD-BLOCKER-STAT", c.ba);
  c.cd.o("onAdBlockStatusReady", [c.ba])
}, e)
```

Fig. 7: Outbrain's anti-adblocking script snippet

Finally, `Outbrain`'s anti-adblocking script is illustrated in Figure 7. We can see the code is minified and the key check here is `(0 < c.gd)` that checks on variable `c.gd` which stores the concatenated and encoded value of all deployed baits to determine if they are still present. To validate our analysis we manually remove the filter rules associated with `DIV` ids `Ads_4`, `AD_area`, `ADBox` and `AdsRec`, and can indeed successfully flip the relevant adblock status field in its reporting request. `Outbrain` also chooses to save the status in browser's cookie (`OB-AD-BLOCKER-STAT`).

Unlike the multiple-bait strategies used in PageFair and Taboola, some scripts use the "pixel" technique [29], which loads a small, unobtrusive piece of image (*i.e.,* pixel) and then drops a browser cookie for future inter-domain ad re-targeting. This ad re-targeting technique allows publishers to ensure that their ads are served only to people who have previously visited their site. The pixel often contains ads-related keywords in its URL path, and therefore can be used as a bait object to detect adblockers. Most of these scripts also silently report adblocking statistics by using a query string (*e.g.,* adblock=0/1) in the HTTP request to load the next pixel. `Yandex` and `Criteo` also leverage the same "pixel" technique. `mail.ru`, `Outbrain` and `Cloudflare` instead create regular DOM baits and check their presence to detect adblockers. It is noteworthy that `Criteo`, besides silent adblocker reporting, also switches ads to acceptable ads [17].

Since many popular third-party anti-adblocking scripts are obfuscated and challenging to manually analyze, we randomly sample a few popular websites that use non-obfuscated anti-adblocking scripts. Our goal is to (1) confirm that the identified websites are not false positives, and (2) understand their detection approach and reaction to adblockers. We select these websites from the Alexa top-1K list: businessinsider.com, nytimes.com, cnn.com, aol.com, cnet.com, gmx.net, reddit.com, sourceforge.net, nba.com, glassdoor.com, expedia.com, iqiyi.com, thefreedictionary.com, ria.ru, jeuxvideo.com, gamespot.com, intel.com, nfl.com, myanimelist.net, kizlarsoruyor.com. All of these websites detect adblockers and some even have visible reactions that were not reported in prior work [42]. This demonstrates the usefulness of differential execution analysis in accurately pinpointing the adblocker detection logic used on any website. Next we discuss in detail the the anti-adblocking logic of a few interesting examples.

The homepage of `businessinsider.com` is flagged by our system to have multiple branch divergences. Surprisingly, we do not see any warning messages and the page seems to be completely ad-free. Upon a closer look, we realize that the website has a first-party script that silently detects the presence of adblocking and records the information into the cookie. The code snippet is illustrated in Figure 8. We note that the website injects a bait script at `www.npttech.com/advertising.js` and invokes the pre-defined callbacks either `onload()` or `onerror()`, depending on whether the bait scripts gets blocked by adblockers. As mentioned earlier, our instrumentation currently does not support callback-based implicit branching which means this may be a false negative case. Fortunately, as we can see inside `setAdblockerCookie()` (which is the registered callback in correspondence with `onload()` and `onerror()`), there is a ternary operator that checks the value of variable `adblocker` which allows us to correctly detect the branch divergence.

```
var setAdblockerCookie = function(adblocker) {
  var d = new Date();
    d.setTime(d.getTime() + 60 * 60 * 24 * 30 *
        1000);
    document.cookie = "__adblocker=" + (adblocker ?
        "true" : "false") + "; expires=" + d.
        toUTCString() + "; path=/";
}
var s = document.createElement("script");

s.setAttribute("src","//www.npttech.com/advertising.
    js");
s.setAttribute("onerror","setAdblockerCookie(true);"
    );
s.setAttribute("onload","setAdblockerCookie(false);"
    );
document.getElementsByTagName("head")[0].appendChild
    (s);
```

Fig. 8: First-party anti-adblocking script in `www.businessinisder.com`

```
BlockAdBlock.prototype.on = function(detected, fn) {
  return this._var.event[detected === !0 ? "detected
      " : "notDetected"].push(fn), this._options.
      debug === !0 && this._log("on", 'A type of
      event "' + (detected === !0 ? "detected" : "
      notDetected") + '" was added'), this}
```

Fig. 9: First-party anti-adblocking script in `nytimes.com`

```
var n=document.getElementById(t);
n&&0!=n.innerHTML.length&&0!==n.clientHeight&&0!==n.
    clientWidth&&0!==n.offsetWidth?e.application.
    fire("adblock:detect",{enabled:!1}):e.
    application.fire("adblock:detect",{enabled:!0}),
    $(i).empty()}
```

Fig. 10: Third-party anti-adblocking script in `aol.com`

`nytimes.com` has a first-party script that logs adblocker usage (see Figure 9. Similarly, `aol.com` includes a third-party script from `blogsmithmedia.com` that fires an application event when adblocker is detected (see Figure 10). Finally,

```
if (!window.isAdblockerDisabled) {
  define('expads', function () {
    var displayFallbackImage = function (slotConfig)
        {
      ...
```

Fig. 11: First-party anti-adblocking script in `expedia.com`

`expedia.com` has a first-party script that attempts to load a fallback image. Interestingly, we are unable to see any fallback image (because even the fallback image is blocked by EasyList) when we manually inspect the page.

VI. Towards Improving Ad-blockers

In addition to leveraging differential execution analysis to detect anti-adblockers, we are interested in understanding how this knowledge can help strength adblockers, making them more resistant against anti-adblockers. As we mentioned earlier in §II, adblockers are currently struggling to keep up with anti-adblockers due to the challenges in manually analyzing the anti-adblocking Javscript (which we find to be extremely diverse and complex).

In this section, we attempt two such directions to help adblockers, with the help of the comprehensive anti-adblocking knowledge. We describe our solutions, implementations, and preliminary results.

A. *Avoiding Anti-adblockers with JavaScript Rewriting*

The differential execution analysis enables us to understand which branches are entered because of the presence/absence of adblockers. This knowledge can also naturally help adblockers to evade anti-adblockers. The idea is to force the outcome of a branch statement towards the one corresponding to the absence of adblockers, effectively avoiding any anti-adblocking logic. However, forcing the outcome of a branch statement may also cause unexpected side effects. Fortunately, since the execution path we are attempting to force already occurs in the negative trace (without adblocker), it is unlikely the anti-adblocking code we avoid will cause any breakage. In other words, we expect to not cause any program inconsistency because the rest of the functionality on a web page is unlikely to depend on the missed anti-adblocking code (as the example in Figure 1 illustrated). Note that our Javascript rewrite is targeting specific branches, as opposed to systematically exploring all possible program paths (which is sometimes desired for malware analysis purposes [38]). Much more care has to be given to ensure the reliability of such an exhaustive program exploration (e.g., checkpointing and rollback are commonly required). In comparison, our solution is much more lightweight and easier to implement.

There are two options for rewriting a condition in a branch: (1) we replace the original condition completely with the desired branch outcome directly; or (2) we keep the original condition but still force the outcome by appending true or false at the end. Figure 12 illustrates the differences. Figure 12(a) shows the original JavaScript code that attempts to detect adblockers. Figure 12(b) and Figure 12(c) correspond to the two rewrite choices above respectively (both can force branch outcome to false successfully). The difference is that the first option prevents any original code in the condition to be executed (*i.e.,* `hasBlock()`), while the second option does allow the original function to be invoked. For the first option of not allowing the adblocker detection code (`hasBlock()`) to execute, it can potentially have negative impact on the remaining code. For instance, a variable may be defined only inside the function. Without invoking the function, the subsequent access to the variable may become undefined and cause site breakage. The second option avoids this issue and we therefore prefer it.

We can even perform more fine-grained rewrite management, *i.e.,* perform the rewrite only when the call stack matches the ones collected in the trace. For instance, if function A and B both call C, and a divergence is discovered in C only when A calls C. Then the rewrite should rewrite the condition only when A calls C as well. The rewritten code would look like the following for the same example as in Figure 12:

```
if (hasBlock() && matches(StackTrace.getSync(),
    recorded_stacktrace) && false) {
  $('.notification').show();
}
```

This allows condition rewrite to operate with more precision and is less likely to affect other execution paths that happen to also depend on the same code block (and may be incorrectly forced to either true or false all the time). Unfortunately, without instrumenting the JavaScript execution engine, we cannot get call stack (or stack trace) in JavaScript without relying on non-standard features [19]. Thus, this approach may not always work even though most modern browsers such as Chrome and Firefox have some support for it [21]. Therefore, we opt not to use it in our current implementation.

When two aligned traces are found to have multiple nested branch divergences, it is important to decide whether to rewrite all of the branch outcomes or only a subset of them. Taking the example in Figure 13, we can force either the return of `isVisible()` call or condition of `Width == 0 && Height == 0` to false. In general, we prefer to rewrite the condition at the outer level, meaning `isVisible(obj)` will be rewritten to `isVisible(obj) || true`. This is because rewriting at lower level can potentially cause functionality breakage as low-level functions tend to be reused for various purposes (potentially beyond adblocker detection).

As mentioned earlier, a typical anti-adblocker requires conditional statements to test whether the desirable ad-related elements are still present in page, and trigger anti-adblocking behaviors accordingly. These elements can either be real ads, or sometimes baits intentionally placed for adblocker detection [42]. It is possible that sophisticated anti-adblocking scripts (such as PageFair) will conduct multiple rounds of such checks.

A simplified real example from www.pandajogosgratis.com is illustrated in Figure 14. We note that the first check considers whether the canRunAds element is blocked or not ('undefined' means that it is blocked). If no blocking is detected, it continues with a secondary check which looks at the length of an element. It is not hard to tell that the positive trace (with adblocker) represented on the control flow graph would be (1:false) and the negative trace (without adblocker) would

```
if (hasBlock()) {
  $('.notification').show();
}
```

(a) Original code

```
if (false) {
  $('.notification').show();
}
```

(b) Condition rewrite (replacement)

```
if (hasBlock() && false) {
  $('.notification').show();
}
```

(c) Condition rewrite (append-only)

Fig. 12: Choices of condition rewrite

```
function checkAdVisible() {
  if(isVisible(ad)) {
      // pass
    } else {
      // penalizing user
    }
}
function isVisible(obj) {
  return (obj.offsetWidth == 0 && obj.offsetHeight
      == 0)? False: True;
}
```

Fig. 13: Nested branch divergence example

be (1:true, 2:true). Now when we analyze the two traces differentially, only the first branch divergence can be detected. Unfortunately it is not enough to force only the first branch to be true, as the second branch will still turn out to be false, resulting in adblock detection.

Our solution to this problem is to iteratively collect such nested divergences. More specifically, once we finish one round of differential trace analysis (with the corresponding JavaScript rewrite rules being produced), we deploy the rule and continue a new round of instrumentation and differential analysis. This way, we will be able to capture the second branch divergence and incrementally include it in our rewrite rules. The iteration stops when no more new divergences are detected.

Implementation and preliminary results. Ideally, we should be able to implement JavaScript rewrite using a browser extension. Unfortunately, JavaScript rewrite is not natively supported by most browser extension APIs. In lieu of that, we implement the rewrite system using the mitmproxy [23]. The downside is that a user needs to install an external program (and certificate) as opposed to only an extension. The benefit is that this proxy-based solution is browser-independent and can be deployed across different platforms. As mitmproxy already provides nice abstractions for HTTP(S) request and response manipulation, our implementation of JavaScript rewrite is only less than 200 lines of python code. The whole system is completely automated in rewriting the right conditions without any human intervention.

To evaluate the effectiveness of the Javascript rewrite, we choose to test the anti-adblocking websites that are known to have visible reactions. After the Javascript rewrite, if the visible reactions are eliminated, we consider it a success. In addition, we will check for any functionality breakage by interacting with the website with modified Javascript.

Overall, for the 428 detected positive websites with visible reactions (from §V-A), we find that the JavaScript rewrite can successfully evade 352 websites (82.2%), evident by the lack of warning or popup messages after the rewrite. Here we follow the same manual inspection methodology in §V-A. The failed cases are mostly due to the same reasons as outlined in §V-A. Only one website is found to have broken functionality where the Javascript is mistakenly considered to be disabled.

B. Hiding Adblockers with API Hooking

While Javascript rewriting is a promising direction, it has several drawbacks and limitations. First, it requires a MITM proxy (or with browser modification) and cannot be implemented as a browser extension. Second, it is more intrusive and likely to cause breakage of site functionality. We next consider an alternative solution that aims to address the above shortcomings.

Our key observation is that all API calls used by publisher scripts to examine the state of the page (e.g., whether an ad is visible) can be intercepted and modified by a browser extension [8, 50]. In Chrome extensions, for example, a content script can run before the page is loaded (no other script can run yet), i.e., `document_start`. This allows one to inject script in the page which can define wrapper functions for existing objects. However, many objects are created on demand and therefore are not available for interception in the beginning. Unfortunately, it may become too late to inject any script after a page is loaded, i.e., `document_end` as other scripts might have already executed (and race conditions may occur). This makes API call interception a challenging task.

To understand how this problem can be overcome, we observe that there are generally two sources of variables/objects that are checked for adblocker detection: (1) DOM elements which are either statically or dynamically created; (2) variables unrelated to DOM elements (defined elsewhere and potentially nested in other objects).

For (1) — DOM element checks, all objects are in fact retrieved through API calls from the browser built-in object `document` such as `document.getElementByName(arg)`. This allows our injected script to intercept the element retrieval. If the object is deemed a bait or a real ad (based on its name, id, or source, etc.), we can simply return a fake object prepared ahead of time. Later when the object is checked for size, visibility, and other attributes, we can simply return the values according to what we have learned during the analysis of anti-adblocking scripts. If the object is dynamically created, it is more challenging to decide if it is an ad object (as its **div id**, **class** and other properties are all dynamically assigned), and therefore may require more monitoring at runtime. An example is shown below:

```
$(window).load(function() {
    if(typeof canRunAds != 'undefined'){ // check level 1
        var adFilled = $('.adsbygoogle').find('ins').
            length;
        if(adFilled!=0){ // check level 2
            // return okay
        } else {
            $('.layerPubAdBg').fadeIn();
        }
    } else {
        $('.layerPubAdBg').fadeIn();
    }
});
```

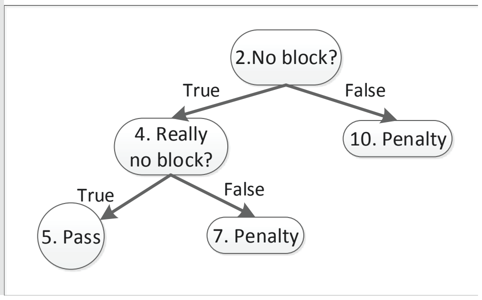


Fig. 14: An example anti-adblocker with two levels of adblock detection

```
var bait = document.createElement ('div') ;
bait.setAttribute ('class', this.options.baitClass);
bait = body.appendChild(bait);
if (bait == undefined || bait.height == 0) { /*
    adblock detected */ }
```

In this case, bait may become null and therefore trigger the adblock detection. Unfortunately, we don't know at the creation time whether the `div` will be blocked by adblocker and cannot simply return a fake object (as it could be a useful `div` not related to ads). However, it is possible that we still instrument `document.createElement()` and inside of it we can add additional hooks to intercept future method invocations on this object (*e.g.,* `setAttribute`), which will allow us to determine the true class of the object. Below is the code snippet to illustrate the instrumentation logic.

```
var old_createElement = document.createElement();
document.createElement = function(type) {
  var temp = old_createElement(type);
    var old_setAttribute = temp.setAttribute();
    temp.setAttribute = function(key, value) {
      // check if it is an ad-related element by
          consulting the adblocker filter list
    }
}
```

This allows us to keep monitoring the future development of a newly created element. If it does turn out to become an ad-related element, we will mark it so. In the future, when the element is checked for its height or other attributes, we can similarly return expected results from the offline knowledge.

Note that an ideal solution would require us to link the object used for adblock detection to its name, id, or classname, etc. This way we will know precisely what values to return when their properties are checked. For instance, if the condition is `obj.height <= 20`, then we need to fake a number that is larger than 20 for the specific `obj`. Such analysis is more complex and will likely involve symbolic execution. We leave implementation of this approach for future work.

For (2) — non-DOM element checks, if the checked variables are not related to DOM elements, the only possibility we have observed is related to Javascript blocking. In such cases, there is typically a global variable (or a variable nested in other global objects) defined in an ad-related script. If the script is blocked, then the variable becomes undefined and therefore trips the adblocker detection. Fortunately, if the variable is a global one and directly accessible from the browser built-in `window` object, we can intercept it and return any expected result to pass the detection check with the following single line of code:

```
// intercept access to window.adblockV1, and always
    return true;
window.__defineGetter__('adblockV1', function() {
    return true; });
```

However, if it is a nested variable defined in other objects, as mentioned earlier we will not be able to intercept its accesses. As a workaround, we propose to let the ad-related script load (instead of blocking it) and rely other adblocking filter rules to remove any injected ads. After all, ads have to be inserted into the DOM tree in order to be rendered (and trigger adblockers to block them). If the ad-related script is not injecting any ads and instead only serving as a bait to define some variables, not blocking the script itself actually can already successfully avoid anti-adblocking. Interestingly, we find many bait scripts such as the one at https://tags.news.com.au/prod/adblock/adblock.js that do exactly this. In the more general case though, this transforms the problem into DOM element checks which we already have a solution for.

Implementation and preliminary results. Without loss of generality, we have implemented a proof-of-concept Chrome extension that works for a randomly selected subset of websites and third-party scripts for which we have ground truth (able to manually analyze the script and confirm their behaviors). We have picked 15 websites, 5 with popular third-party scripts (silent reporting), 5 with less popular or custom scripts (silent reporting), and 5 with visible reactions (ad switching or warning messages).[2] Our solution works well against all of these websites, *i.e.,* it successfully avoids the anti-adblockers. Specifically, we can always successfully either avoid the warning messages or change the reporting messages (e.g., from `adp = 1` to `adp = 0`). We find that 8 websites (and their corresponding scripts) check attributes of DOM elements

[2] They are: *popular third-party scripts:* https://mc.yandex.ru/metrika/watch.js, http://static.criteo.net/js/ld/publishertag.js, http://widgets.outbrain.com/outbrain.js, https://cdn.taboola.com/libtrc/impl.254-8-RELEASE.js, http://asset.pagefair.com/measure.min.js; and *websites with less popular or custom scripts:* philly.com, foxsports.com.au, cda.pl, bt.dk, boredomtherapy.com; and *websites with visible anti-adblockers (first with ad-switching and others with warning messages):* memburn.com, pasty.link, exspresiku.blogspot.co.id, ani-short.net, gta.com.ua.

and 7 websites check values of variables other than DOM elements (e.g., defined or not). Out of the 7, 6 check a global variable such as `window.adblockV1` and therefore can be easily intercepted and tricked. One website, however, checks a nested variable `window.utag_data.no_adblocker`. Interestingly, both `utag_data` and `no_adblocker` are defined in a bait script. Simply allowing the script to execute can trick the adblock detector without any other implications. We analyze a few more scripts below as case studies.

One of the most popular anti-adblocking third-party scripts from Taboola has a complex logic of adblock detection spanning several functions (the simplified code snippet already explained in §V-B). Specifically, the script is written generically so that it can load a list of bait DOM elements dynamically by iterating through a list of predefined element ids (strings).[3] Despite this, as soon as we can track the origin of the element ids in the array, the rest can follow our procedure as described earlier (about how to deal with DOM element checking).

As an interesting example, we show that memburn.com's ad switching behavior (described in Figure 3(a)) is now completely disabled as they are unable to detect the failure of loading the initial ad through the simple check `window.advertsAvailable === undefined`. This is because we can intercept all accesses to `window.advertsAvailable` and simply fake any arbitrary value. The page will now simply contain an empty white space in place of the original ad frame.

Finally, we revisit the website bild.de for which Javascript rewrite has caused functionality breakage. By manual inspection, we have found that Javascript rewrite targeted a wrong function which is general and used by legitimate part of the website. By applying the analysis procedure outlined in this section, we simply hook the access to `window._art` and provide a fixed constant to solve this issue.

Fundamentally, the API hooking based solution operates on the source of the problem — DOM elements or ad-related scripts that get blocked by adblockers; it is therefore more precise and less likely to cause side effects, compared to Javascript rewrite. In addition, since our solution is readily deployable in a standard browser extension, it has potentials to influence the future design of adblockers.

VII. Limitations and Discussion

As we have seen, applying differential trace analysis to detect and analyze anti-adblockers is a promising direction, and it has validated our idea to a large extent. Future directions include improving the differential execution analysis by considering the value differences, as well as investigating the feasibility of the techniques to hide adblockers. Below we discuss the limitations of our solution at the implementation level and design level.

Completeness of instrumentation. Our system is as good as the capability of the instrumentation. At the moment, we do not cover all branch statements. It is especially challenging to catch implicit branching operations such as callbacks (as mentioned in §IV-A). To overcome this, one strategy is to catch the registration of callbacks (*e.g.,* `onsuccess()` and `onerror()` that are associated to the same event. This way, we will be aware of which one of the callbacks is taken in the A/B testing and catch the implicit branch divergence. Nevertheless, in theory an our system becomes popular and the instrumentation details are made known to the websites, they could easily counteract by hiding the adblocker detection logic in the form that we do not capture. In addition, we also acknowledge that in theory both flow differences and value differences need to be considered, as anti-adblockers in theory can hide its logic without changing control flows. One other issue is the dynamically generated code through `eval()`, which can be addressed with improvement of instrumentation as well — after all, we are instrumenting the Javascript engine.

Robustness of differential execution analysis. Assuming a perfect instrumentation capability, we should be able to catch most state-of-the-art anti-adblockers. However, we point out three different cases where randomness can interfere or defeat our differential analysis. First, we find that fluctuations in the network speed and system load can affect the load time of the ads. Since adblocker detection in many cases is triggered by a timeout callback (1s or 2s), an ad may or may not be completely loaded when the detection logic is triggered, introducing randomness its execution trace. To mitigate this unintentional randomness, we can force objects to be loaded from cache. Next, the randomization can happen at two levels: (1) behavioral randomization — same script, different behavior; (2) content randomization — different script, different behavior. For content randomization, as we discussed, can be addressed by forcing the same exact webpage/scripts to be reloaded during A/B testing. For behavioral randomization where multiple anti-adblocking modules exist and one of them will be randomly selected in each run, we envision that it can be addressed based on the following observation: the random selection of modules has to be guided by some underlying source of randomness (*e.g.,* system clocks, external network packets). If we can force a random source in every run, then the random selection becomes deterministic (*e.g.,* a random coin flip becomes deterministic). This is very much similar to virtual machine replay where all external non-determinism are recorded and replayed to ensure the deterministic behavior of the VM. In summary, we argue that the two kinds of randomization does not pose a fundamental threat to our differential analysis.

Robustness of anti-adblocker evasion. Equipped with the result of differential analysis, we have demonstrated the power of the JavaScript rewrite and API hooking based solutions. They are subject to the ongoing arms race between adblockers and anti-adblockers. For Javascript rewrite especially, it is in general hard to estimate and contain the effect of any code change, and therefore can hinder real-world deployment. Even worse, rewriting Javascript cannot be conducted in a browser extension and therefore further limits its uses. For API hooking, as we discussed in §VI-B, it is much more precise, close to the root, and therefore much less likely to induce undesired side effects. The challenge of this approach though is the reliance on the discovery of the exact DOM elements that are checked in adblock detection (which may require further program analysis), and we leave as future work.

In addition, both approaches share a fundamental limitation

[3]More details can be found in our project website at https://sites.google.com/view/antiadb-proj/.

of webpage or Javascript content randomization. Unlike the task of anti-adblocker detection conducted in a controlled environment, for which we can force not only the same page/script to be used but also the execution to be deterministic (see discussion earlier), the task of anti-adblocker evasion happens in real users' browsers where we may not be able to contain randomization. For instance, for content randomization (different pages/scripts are loaded in each visit), there is no fixed page or script to learn from offline and every user will potentially obtain a unique version that has never been observed in the past. Such frequent randomizations, however, will likely hurt the web performance by effectively disabling caching. Users who are not using adblockers will also be unnecessarily penalized.

It is slightly easier to deal with behavior randomization where the same exact script randomly selects anti-adblocking modules during different runs. In this case, since different users will obtain the same copy of script, it is possible to learn which part of the code is related to the random selection of anti-adblocking modules, and simply force the outcome of that random outcome to eliminate this particular source of non-determinism.

VIII. Conclusions

We presented a differential execution analysis approach to discover anti-adblockers. Our insight is that websites equipped with anti-adblockers will exhibit different execution traces when they are visited by a browser with and without an adblocker. Based on this, our system enables us to unveil many more (up to 52×) anti-adblocking websites and scripts than reported in prior literature. Moreover, since our approach enables us to pinpoint the exact branch statements and conditions involved in adblocker detection, we can steer execution away from the anti-adblocking code through JavaScript rewriting or hide the presence of adblockers through API hooking. Our system can bypass a vast majority of anti-adblockers without causing any site functionality breakage (except one with Javascript rewriting).

We anticipate escalation of the technological battle between adblockers and anti-adblockers — at least in the short term. From the perspective of security and privacy conscious users, it is crucial that adblockers are able to keep up with anti-adblockers. Moreover, the increasing popularity of adblocking has already led to various reform efforts within the online advertising industry to improve ads (*e.g.,* Coalition for Better Ads [5], Acceptable Ads Committee [2]) and even alternate monetization models (*e.g.,* Google Contributor [6], Brave Payments [4]). However, to keep up the pressure on publishers and advertisers in the long term, we believe it is crucial that adblockers keep pace with anti-adblockers in the rapidly escalating technological arms race. Our work represents an important step in this direction.

Acknowledgment

We would like to thank the anonymous reviewers for their valuable feedback on this paper. This work is supported in part by the National Science Foundation under grant numbers 1719147, 1715152, 1664315, and by a seed grant from the Data Transparency Lab (DTL).

References

[1] "A Publishers Guide To Counter-Ad Blocking Technology," https://adexchanger.com/platforms/a-publishers-guide-to-counter-ad-blocking-technology/.

[2] "Acceptable Ads Committee," https://acceptableads.com/en/committee/.

[3] "AdBlock for Chrome Now Hides Facebook Ads and Blocks More Ads On More Sites," https://blog.getadblock.com/adblock-for-chrome-now-hides-facebook-ads-and-blocks-more-ads-on-more-sites-f5918ebc43c6.

[4] "Brave Payments," https://brave.com/publishers/.

[5] "Coalition for Better Ads," https://www.betterads.org/.

[6] "Google Contributor," https://contributor.google.com/v/beta.

[7] "Googles Inbuilt Ad-Blocker Comes To Chrome Canary," https://techviral.net/googles-inbuilt-ad-blocker-comes-chrome/.

[8] "Introducing ES2015 Proxies," https://developers.google.com/web/updates/2016/02/es2015-proxies.

[9] "Manifest - Web Accessible Resources," https://developer.chrome.com/extensions/manifest/web_accessible_resources.

[10] "Blocking Taboola ads," https://adblockplus.org/forum/viewtopic.php?t=20747, 2014.

[11] "Anti-Adblock Killer List," https://github.com/reek/anti-adblock-killer/blob/master/anti-adblock-killer-filters.txt, 2015.

[12] "Native Advertising: A Guide for Businesses," https://www.ftc.gov/tips-advice/business-center/guidance/native-advertising-guide-businesses, 2015.

[13] "Facebook Will Force Advertising on Ad-Blocking Users," https://www.wsj.com/articles/facebook-will-force-advertising-on-ad-blocking-users-1470751204, 2016.

[14] "The Rise of the Anti-Ad Blockers," https://www.wsj.com/articles/the-rise-of-the-anti-ad-blockers-1465805039, 2016.

[15] "AdBlock," https://getadblock.com/, 2017.

[16] "Adblock forum," https://forums.lanik.us, 2017.

[17] "Adblock Plus 2.0, Allow non-intrusive advertising," https://easylist-downloads.adblockplus.org/exceptionrules.txt, 2017.

[18] "Adblock Warning Removal List," https://easylist-downloads.adblockplus.org/antiadblockfilters.txt, 2017.

[19] "Error.prototype.stack," https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Error/Stack, 2017.

[20] "Forum section about false content blocking," https://forums.lanik.us/viewforum.php?f=64&sid=acd4dfc10ed86e1bc7e29d5f482fd8c7, 2017.

[21] "Generate, parse, and enhance JavaScript stack traces in all web browsers," https://github.com/stacktracejs/stacktrace.js, 2017.

[22] "Google v8," https://developers.google.com/v8/, 2017.

[23] "mitmproxy," https://mitmproxy.org/, 2017.

[24] "PageFair," https://pagefair.com/, 2017.

[25] "The Chromium Projects," https://www.chromium.org/Home, 2017.

[26] "The state of the blocked web 2017 Global Adblock Report. PageFair,"

https://pagefair.com/downloads/2017/01/PageFair-2017-Adblock-Report.pdf, 2017.

[27] "YourAdChoices Gives You Control," http://youradchoices.com/, 2017.

[28] C. Curtsinger, B. Livshits, B. Zorn, and C. Seifert, "ZOZZLE: Fast and Precise In-Browser JavaScript Malware Detection," in *USENIX Security Symposium*, 2011.

[29] Digital-Advertising-Blog, "What tracking pixels are and why they matter to your next digital ad campaign," http://www.digitaland.tv/blog/what-is-tracking-pixel-ht/, 2017.

[30] S. Englehardt and A. Narayanan, "Online tracking: A 1-million-site measurement and analysis," in *ACM Conference on Computer and Communications Security (CCS)*. ACM, 2016.

[31] D. Gugelmann, M. Happe, B. Ager, and V. Lenders, "An automated approach for complementing ad blockers blacklists," *Proceedings on Privacy Enhancing Technology (PETS) 2015*, 2015.

[32] S. Guha, B. Cheng, and P. Francis, "Privad: Practical privacy in online advertising," in *NSDI*, 2011.

[33] X. Hu, A. Prakash, J. Wang, R. Zhou, Y. Cheng, and H. Yin, "Semantics-preserving dissection of javascript exploits via dynamic js-binary analysis," in *Proceedings of the 19th Symposium on Research in Attacks, Intrusions and Defense (RAID'16)*, Sep. 2016.

[34] M. Ikram, H. J. Asghar, M. A. Kaafar, A. Mahanti, and B. Krishanmurthy, "Towards seamless tracking-free web: Improved detection of trackers via one-class learning," in *Proceedings on Privacy Enhancing Technology (PETS)*, 2017.

[35] U. Iqbal, Z. Shafiq, and Z. Qian, "The ad wars: Retrospective measurement and analysis of anti-adblock filter lists," in *Proceedings of the ACM Internet Measurement Conference (IMC)*, 2017.

[36] N. M. Johnson, J. Caballero, K. Z. Chen, S. McCamant, P. Poosankam, D. Reynaud, and D. Song, "Differential slicing: Identifying causal execution differences for security applications," in *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, ser. SP '11, 2011.

[37] A. Kapravelos, Y. Shoshitaishvili, M. Cova, C. Kruegel, and G. Vigna, "Revolver: An Automated Approach to the Detection of Evasive Web-based Malware," in *USENIX Security Symposium*, 2013.

[38] K. Kim, I. L. Kim, C. H. Kim, Y. Kwon, Y. Zheng, X. Zhang, and D. Xu, "J-force: Forced execution on javascript," in *Proceedings of the 26th International Conference on World Wide Web*, ser. WWW '17, 2017.

[39] A. Lerner, A. K. Simpson, T. Kohno, and F. Roesner, "Internet Jones and the Raiders of the Lost Trackers: An Archaeological Study of Web Tracking from 1996 to 2016," in *Proceedings of USENIX Security*, 2016.

[40] M. Malloy, M. McNamara, A. Cahn, and P. Barford, "Ad blockers: Global prevalence and impact," in *ACM Internet Measurement Conference (IMC)*, 2016.

[41] H. Metwalley, S. Traverso, M. Mellia, S. Miskovic, and M. Baldi, "The Online Tracking Horde: A View from Passive Measurements," in *Traffic Monitoring and Analysis*, 2015.

[42] M. H. Mughees, Z. Qian, and Z. Shafiq, "A first look at ad-block detection: A new arms race on the web," *Proceedings on Privacy Enhancing Technology (PETS)*, 2017.

[43] A. Narayanan and D. Reisman, "The princeton web transparency and accountability project."

[44] M. D. Network, "Concurrency model and Event Loop," https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop, 2017.

[45] R. Nithyanand, S. Khattak, M. Javed, N. Vallina-Rodriguez, M. Falahrastegar, J. E. Powles, E. D. Cristofaro, H. Haddadi, and S. J. Murdoch, "Adblocking and counter blocking: A slice of the arms race," in *6th USENIX Workshop on Free and Open Communications on the Internet (FOCI 16)*. Austin, TX: USENIX Association, 2016. [Online]. Available: https://www.usenix.org/conference/foci16/workshop-program/presentation/nithyanand

[46] K. Rieck, T. Krueger, and A. Dewald, "Cujo: Efficient detection and prevention of drive-by-download attacks," in *Proceedings of the 26th Annual Computer Security Applications Conference*, ser. ACSAC '10, 2010.

[47] F. Roesner, T. Kohno, and D. Wetherall, "Detecting and Defending Against Third-Party Tracking on the Web," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

[48] A. K. Sood and R. J. Enbody, "Malvertising–exploiting web advertising," *Computer Fraud & Security*, vol. 2011, no. 4, pp. 11–16, 2011.

[49] SOPHOS, "Adblocker blockers move to a whole new level," https://nakedsecurity.sophos.com/2016/02/01/adblocker-blockers-move-to-a-whole-new-level/, 2016.

[50] G. Storey, D. Reisman, J. Mayer, and A. Narayanan, "The future of ad blocking: An analytical framework and new techniques," *Technical Report*, 2017.

[51] T. P. Team, "2017 Adblocking Report," https://pagefair.com/blog/2017/adblockreport/, 2017.

[52] W. Wang, Y. Zheng, X. Xing, Y. Kwon, X. Zhang, and P. Eugster, "Webranz: Web page randomization for better advertisement delivery and web-bot prevention," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016, 2016.

[53] B. Xin, W. N. Sumner, and X. Zhang, "Efficient program execution indexing," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '08, 2008.

[54] D. Yu, A. Chander, N. Islam, and I. Serikov, "Javascript instrumentation for browser security," in *ACM SIGPLAN Notices*, vol. 42, no. 1. ACM, 2007, pp. 237–249.

[55] A. Zarras, A. Kapravelos, G. Stringhini, T. Holz, C. Kruegel, and G. Vigna, "The Dark Alleys of Madison Avenue: Understanding Malicious Advertisements," in *ACM Internet Measurement Conference (IMC)*, 2014.