UNIVERSITY OF CALIFORNIA
RIVERSIDE

Understanding and Taming Adversarial Actions against Internet Content Blockers

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Shitong Zhu

December 2021

Dissertation Committee:

Dr. Zhiyun Qian, Chairperson
Dr. Srikanth V. Krishnamurthy
Dr. Heng Yin
Dr. Vagelis Papalexakis

The Dissertation of Shitong Zhu is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

First and foremost, I wish to express my heartiest thanks to both of my co-advisors, Dr. Zhiyun Qian and Dr. Srikanth V. Krishnamurthy. I would like to thank Zhiyun for choosing to hire me as one of his early doctoral students, despite my lack of computer science background from undergraduate studies. Besides mentoring and advising me on my research projects, Zhiyun has also spent an unparalleled amount of efforts in cultivating me into an independent researcher with visions and insights. I would like to thank Srikanth for working extensively with me in later years of my doctoral studies, which is vital in shaping my current and future research area. As a renowned and more senior scholar in the community, Srikanth has imparted a wide spectrum of his wisdom to me, including but not limited to how I should manage the entire cycle of research projects, and how a mature researcher should interact with his/her collaborators and funding agencies. I have honestly learned a ton from both of them. Thank you, Zhiyun and Srikanth! I will keep striving after graduation to be an advisee you are proud of.

I sincerely thank other members on my dissertation committee, Dr. Heng Yin and Dr. Vagelis Papalexakis. Being the top-notch scholars in their respective fields, my dissertation has enjoyed lots of insightful/shrewd feedback from them since its proposal. I genuinely feel lucky to have this opportunity to work with both of them at UCR.

I would like to thank my parents Hedong Long and Jiang Zhu, for their continued support throughout my doctoral studies. Moreover, ever since my birth, they have been devoting a significant portion of their energy and time to nothing but my upbringing and development. Without them, I could not have been here, writing my doctoral dissertation.

Special thanks also go to my other family members, including but not limited to Chuan Zhu, Mu Zhang, Jin Long and Yinchun Deng, for their special love that has never ceased to support me through all these years. I am utterly grateful to my beloved wife Jiajia Lin as well, for always being there for me. Without her, my later doctoral years would have been infinitely more stressful and duller. This dissertation is dedicated to my family.

I would specially thank my grandparents Xiufang Zhou, Kaifu Long, Li Xue and Dewei Chen. They, in their own ways, all significantly and positively influenced my formative years, which subtly, but also powerfully instilled the idea of embarking on the quest for knowledge and truth into my young mind. I still recall those hot and humid midsummer afternoons in my hometown, where I sat quietly by the windowsill in the small flat of Xiufang and Kaifu, being absorbed into the encyclopedia in my hands, while unconsciously hearing the way a sudden summer shower started falling on pavement on the outside. I sincerely hope I can do this again someday in the future with my grandparents around, just like old times, after I earn my doctorate, but before their declining health renders such an event impossible.

I bow down to my awesome fellow lab mates, both current and past: Dr. Zhongjie Wang, Dr. Shasha Li, Dr. Yue Cao, Dr. Daimeng Wang, Dr. Hang Zhang, Weiteng Chen, Yu Hao, Zheng Zhang, Yizhuo Zhai, Keyu Man, Xiaochen Zou, Guoren Li, Pengxiong Zhu, Xingyu Li, Xingyun Du, Zhutian Liu, Xin'an Zhou, Dr. Ahmad Darki, Abdulrahman Fahim and Dr. Kittipat Apicharttrisorn, for all the time, laughter and sleepless nights (mostly before deadlines) we have had together. Even though the COVID-19 pandemic prevents us from seeing each other in person again prior to my defense, I will forever miss

our time spent together at WCH 367. I am also indebted to my long-term collaborators Dr. Umar Iqbal, Dr. Zubair Shafiq and Dr. Xun Chen, for sharing their sharp and insightful comments and discussions. I just could not ask for co-workers better than them.

Last but not the least, my heartfelt thanks also go to my lovely friends since childhood. Xiaodi Xu, Yingxue Yang, Yunyi Wang, Ruifeng Chen, Ruijiang Zhao, Haoran Zhang and many more have not only been believing in me despite my good or tough times, but also constantly showering me with their unwavering support. They include Chunhao Shan, Hao Chen and Xiaoyu Wen who have been helping me and having fun together with me a lot since I started my graduate school in USA. In fact, I am now typing this last sentence of acknowledgments on a curved ultra-wide monitor that Chunhao generously lent to me, for me to write my dissertation as comfortably as possible :).

Bibliographical Notes. Chapter 2 is the reproduction of the paper titled "Measuring and Disrupting Anti-Adblockers Using Differential Execution Analysis" published at *The Network & Distributed System Security Symposium (NDSS)* 2018 [146]. Chapter 3 reproduces the paper titled "ShadowBlock: A Lightweight and Stealthy Adblocking Browser" published at *The Web Conference (WWW)* 2019 [147]. Chapter 4 replicates the paper titled "Eluding ML-based Adblockers With Actionable Adversarial Examples" that is accepted to appear at *Annual Computer Security Applications Conference (ACSAC)* 2021 [149]. Lastly, Chapter 5 is the reproduction of the paper titled "You Do (Not) Belong Here: Detecting DPI Evasion Attacks with Context Learning" published at *Conference on emerging Networking EXperiments and Technologies (CoNEXT)* 2020 [148]. These papers are all primarily authored by myself. In addition, the titles of chapters in this dissertation are slightly altered

from the original paper titles to reflect and summarize their central messages, and remove decorative phrases.

To my family.

ABSTRACT OF THE DISSERTATION

Understanding and Taming Adversarial Actions against Internet Content Blockers

by

Shitong Zhu

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, December 2021
Dr. Zhiyun Qian, Chairperson

The Internet has never ceased to be a central battle for adversarial actors to launch various attacks and abuses. More importantly, these actions have not received sufficient attention they deserve in the community, as most threat detection and defense works are devoted to tackle direct attacks/abuses instead of adversarial ones that enable and cloak direct threats. A significant portion of adversarial actions on the Internet target content blockers, which are primarily purposed to detect and overthrow unwanted and ill-intended Internet resources and traffic, because of their ever-growing popularity and dominance. Given the status quo, a more complete understanding of these threats is urgently needed, as well as effectual countermeasures and defenses against them.

In this dissertation, I aim at demystifying emerging adversarial actions against content blockers from different layers of the Internet, and design defenses for protecting user security and privacy accordingly. For discerning and analyzing these actions, I propose automated analysis, or learning-based methods as technical solutions, which range from program analysis, software instrumentation to machine learning. I then develop platforms

for measuring and detecting adversarial actions in the wild. At the core of my analysis, I observe that the fundamental nature of adversarial actions lies in **differential/conditional reaction**, which refers to how adversarial actors distinctly determine their behaviours depending on what they perceive from the target content blocker. This is because unlike other general attacks or abuses, adversarial actions, by definition, are specifically designed to evade deployed detection and protections (i.e., content blockers). Moreover, I find that such a unique differential behavioural pattern is best captured by examining and comparing **low-level/high-resolution contextual traces under different settings** of the deployed content blocking system. Therefore, I build platforms to define, extract and analyze such signals from Internet data and code, and use them to discover adversarial activities. Furthermore, in order to thwart unearthed adversarial actions, I design defenses that **conceal** distinguishable traces from the system protected by content blockers to prevent adversaries from activating their adversarial differential reactions.

In Chapter 2, 3 and 4 of this dissertation, I focus on the upper-level application layer of the Internet, and tackle adversarial threats there from web advertising practitioners. In Chapter 5, I turn to the lower-level infrastructure layer of the Internet, and detect/counter evasions that are designed for eluding intrusion detection systems based on packet inspections. In summary, by hardening layers of the Internet, I attempt to make it a more secure and private place overall, through learning and other automated analysis approaches. At the end of this dissertation (Chapter 6), I conclude my research contributions and highlight some open research questions that are worth investigating in the future.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Research Problem

Internet, or cyberspace in general, faces a wide range of emerging adversarial actions nowadays. Unlike general attacks and abuses, these threats are specifically designed to evade deployed detection and protections. In other words, they explicitly target vulnerable components of existing security measures to bypass them. Since this is not a direct attack on end users, but rather a paradigm to enable and cloak direct attacks and abuses, adversarial actions are generally considered understudied in the community. Among these adversarial threats, an increasingly present category is the ones that aim at subverting Internet content blockers; these blockers are designed to detect and thwart Internet resources and traffic that are considered unwanted and ill-intended. Adversarial actions are more and more motivated to elude them, because content blockers have been proven critically effective in curbing Internet attacks and abuses in various scenarios (e.g., adblocking and intrusion detection/prevention). Moreover, these actions are not limited to any particular layer of the

Internet. This is because as security and privacy gradually become an crucial aspect of the Internet, protection measures such as content blocking have been widely deployed across different layers, and adversaries are thus motivated to elude all of them for launching their end attacks. Therefore, I in this dissertation aim at both developing a more comprehensive understanding of adversarial actions/actors against Internet content blockers from different layers of the Internet, and exploring defenses to avert them effectually and efficiently.

1.2 Research Challenges

A grand challenge of analyzing and discerning adversarial actions lies in their stealthiness. Unlike direct or general-purpose attacks and abuses, adversarial actions are **conditional** and **differential**. That is, adversarial actors determine and adjust their behaviours depending on the status and feedback from the target content blocker (e.g., anti-adblockers only attempt to punish users when they detect that adblockers are installed and blocking ads). This unique characteristic prevents many nondiscriminatory analysis approaches from working properly, because they cannot observe the adversarial actions at all. In addition to the differential reaction, adversarial actions are stealthy also because they often only exhibit **minute differences** compared to benign actions (e.g., for evasion attacks against packet inspection infrastructures, there can be only a few specially-crafted packet headers), which is difficult to be captured by coarse-grained methods. Moreover, aligning with their adversarial nature, adversarial actors are highly motivated to adapt their actions and further evade analyses and defenses against them. Hence, my analyses and defenses must be reasonably **robust** against adversarial adaption and secondary evasion.

Besides the challenges in discerning and analyzing adversarial actions, I also need to take unique properties of the Internet system into account while building detection and defense platforms against them. Specifically, Internet handles a massive amount of data constantly, which renders analyses that primarily rely on manual inspection implausible. In other words, my analyses ought to be **automated** and **scalable** in order for being applied to large-scale Internet data and code. Additionally, my analyses and defenses need to be sufficiently **performant** to avoid disrupting functionalities of the Internet, if they are expected to be deployed in real-time environments.

1.3 Technical Approaches

To address all research challenges presented above, and meet desirable properties for solutions against adversarial actions, I leverage a wide spectrum of techniques. For capturing the signature differential behavioural pattern of adversarial actions, I propose to correspondingly apply **differential testing** [1], which essentially configures the protection measure that adversaries aim at evading differently, and examines and compares behaviours of the adversarial actor under different configurations. As explained in 1.1, this differential examination strategy can best reveal the differential behavioural pattern from adversarial actions themselves, as well as maximally expose signals that can be used to recognize them. However, for certain adversarial threats (e.g., evasion attacks against packet inspection systems, or packet content blockers), it is not immediately clear how one can coarsely configure the entire protection measure. Instead, the differential behavioral pattern of these

[1] Also known as A/B testing in some literature.

threats can be unraveled by analyzing **high-resolution interrelationships** among objects that appear in **low-level system traces**. This is because in these traces, there exist *micro* behavioral patterns where adversarial actors finely adjust their actions differentially (e.g., how to manipulate header fields of next packets), depending on what status and feedback (e.g., whether a disallowed keyword in packet payload has been previously blocked by the intrusion detector or not) they perceive from the target content blocker.

Collecting low-level system traces requires access to low-level internals inside Internet components (i.e., various software systems), which is not typically feasible with public software interfaces only (e.g., logging). Therefore, I leverage **software instrumentation** as the means to gather needed runtime information for understanding relationships among objects (e.g., packet or web element) in system traces. Additionally, in order to capture minute signals buried in the large amount of Internet data and code, I need to engage automated analyses that scale well. For both upper- and lower-layers of the Internet, I find that **program analysis** and **machine learning** can serve as two approaches to realize the goal of efficient automated analysis. Specifically, recall that I need to analyze both Internet data (e.g., webpages and network traffic) and code (e.g., JavaScript software), in which for the former my analysis should be able to automatically extract and learn meaningful statistical patterns from data samples, and for the latter the semantics of programs/code must be understood and assessed; evidently these analysis objectives can be achieved via machine learning and program analysis, respectively, with a sufficient level of efficiency.

1.4 Dissertation Contributions

Now I discuss the specific problems I solve in this dissertation, under the general theme of understanding and taming adversarial actions against Internet content blockers. At a high level, as aforementioned, I tackle threats against content blockers from both upper- and lower-layers of the Internet. Specifically, for the upper application layer of the Internet, I focus on the emerging ecosystem of web advertising, where adversarial actors aim to evade either manual blocklists or ML-based adblockers. For lower infrastructure layer of the Internet, I focus on evasion attempts that are purposed to subvert Deep Packet Inspection (DPI) systems. Note that although these two areas are only subsets of upper- and lower-layers of the Internet, they represent typical emerging thrusts of adversarial actions against the Internet and are thus considered generalizable to other threats in this dissertation. In the remainder of this section, I summarize the contributions of chapters in this dissertation in detail, categorized by the type of the Internet content blocker their analyses intend to defend against adversarial actions.

1.4.1 Web Content Blocking

Modern Web is largely ad-powered. Instead of selling contents to users (e.g., via subscriptions), many first-party content providers (e.g., news websites) turn themselves into ads publishers, cooperate with ads exchange networks (e.g., Google), and sell users' attention to third-party advertisers. Specifically, these publishers sell available slots on their pages to be delegated by exchange networks, which are later bid by the advertisers with highest offers. Moreover, these online advertisements heavily rely on web tracking technolo-

gies, which essentially build user profiles by collecting the user browsing and page interaction history across different sites and domains; these profiles are then used to personalize ads that are chosen to be displayed for particular end users. Such precise and ubiquitous tracking is often considered a major violation of user privacy. In addition, malvertising campaigns that spread malicious code (e.g., Trojan horse programs) through seemingly-benign advertisements have also been becoming a considerable security threat for Web users. Besides privacy and security concerns, removing ads also helps significantly speed up page loads, which brings about a faster web browsing experience.

Despite some regulatory or voluntary efforts that have never gone into meaningful effect (e.g., Do Not Track [113]), adblocking remains as the only viable means for users to resist excessive and intrusive web advertising on the Internet. Essentially, adblockers first engage various approaches for identifying web resources that relate to ads and trackers, then prevent these resources from loading (usually in the web browser). Next I will discuss different methods to classify web resources for blocking ads, and how my analyses and defenses can discern and hinder adversarial actions against such ads classification.

Blocklist-based Adblocking

The first, and more traditional thrust of recognizing and curbing ads resources is using manually-curated blocklists. Voluntary users, along with a small group of experts have founded and been maintaining a community where a set of lists that specify what web resources should be blocked are created and being updated; these lists are referred to as *blocklists*. Due to the rising popularity of blocklists, ads publishers have been increasingly motivated and acting to counter adblocker uses to recover their revenue, which leads to

adversarial actions against adblocking. One of the most noticeable types of such actions is *anti-adblocking*, which deploys JavaScript code that can detect and/or respond to the presence of adblockers at client-side (browser). For revealing anti-adblockers, I propose a differential testing strategy in Chapter 2 that instruments the web browser, collects its JavaScript execution traces under different adblocker settings (i.e., enabled and disabled), and applies program analysis techniques to discover divergences between these two sets of traces. With strictly controlled experimental variables (i.e., adblocker setting), any discovery of such execution trace divergences indicates the presence of anti-adblockers because they are the only possible remaining source to cause such bifurcation. Beyond detecting and analyzing anti-adblockers, Chapter 2 also presents a preliminary solution based on JavaScript API hooking to help adblocker users bypass anti-adblockers.

In seeking of stronger and more complete defenses against anti-adblocking, I find the root cause that enables anti-adblockers to effectively probe the use of adblockers is the changes of page states (e.g., presence/absence of blocked ads) introduced by adblockers. In other words, anti-adblockers also engages some form of differential testing to detect the existence of adblockers. Naturally, one of the most fundamental defenses against such a differential-testing-based adversarial action is to conceal the changes of page states. Specifically, in Chapter 3, I design a browser rendering framework that makes and maintains two copies of the loaded page, *adblocked-copy* and *shadow-copy*; where in the former ads are blocked, and in the latter ads are still preserved. The key in this design is that I only allow all JavaScript runtime APIs that can be used to probe page states to access states in the shadow-copy, but render the adblocked-copy to end users. This way, users avoid both

seeing ads in their viewpoints, and triggering anti-adblockers because they cannot perceive the presence of adblockers, at the same time. In the meantime, I limit/minimize the the scope of copied page states to ads elements only, to avoid prohibitive runtime overhead for maintaining a large number of live state copies. This requires accurate and efficient identifications of ads elements on webpages, which is extensively discussed along with other technical details in Chapter 3.

ML-based Adblocking

Despite the undeniable success of blocklist-based adblockers (and the fact they have become the de facto solution in the market), several inherent weaknesses of them have also been surfacing. First, the volunteer-run community that backs blocklists is inevitably prone to human errors (i.e., both misidentified benign web resources and missed ads), and difficult to keep pace with ever-emerging web advertisements. Second, as explained in the previous subsection, a number of adversarial actions against blocklist-based adblocking are being increasingly deployed in the wild by ads publishers, which significantly jeopardize the effectiveness of adblockers. To overcome these challenges, in recent years, some ads classifiers that rely on machine learning models have been proposed in the research community. Most notably, [80] presents a graph-based solution that translates the browser rendering events and relations (e.g., how HTML element A is connected to HTTP request B) into a graph representation; specifically, it instruments the browser core and collects/sequentializes HTML, HTTP(s) and JavaScript layers of the webpage, and eventually constructs the graph with events from different layers as nodes, and the causal relations (i.e., one event

causes another) as edges. Then [80] trains a tree-based classifier based on the labeling of existing blocklists to recognize network requests that relate to ads and trackers.

In Chapter 4, given the growing popularity of ML-based adblockers, I explore an interesting research question: Is it possible to craft *adversarial examples* that evade detection but preserve page functionalities (including ads themselves)? Unlike *adversarial machine learning* in less- or un-constrained domains (e.g., vision), webapges are inherently bounded by various constraints that are necessary to ensure they are grammatically parsable and semantically correct. These constraints can be easily violated if I blindly apply algorithms (e.g., standard gradient-based descent) from unconstrained domains, because the generated perturbations can be incompliant with domain constraints. Instead, Chapter 4 proposes a new framework that by iterative correction and adaption, progressively crafts actionable adversarial examples that can both evade ML-adblockers and avoid disrupting irrelevant page semantics. Through the lens of these examples, I simulate and analyze the possible routes that adversarial actors can take to overthrow learning-based adblockers, as well as distill insights into safety and robustness of content blockers on Web, in general.

1.4.2 Network Content Blocking [2]

Going deeper into the Internet hierarchy/stack, Deep Packet Inspection (DPI) middleboxes are widely deployed as part of modern network security infrastructures. They not only inspect individual packets, but also reassemble them to form stateful connections

[2]Note that even we use the term "content blocking" here, we are aware that technically, DPI itself is exclusively responsible for detecting unwanted network traffic instead of blocking it. However, we believe that in the context of network security and intrusion detection, because content detection is often the necessary preceding step that leads to subsequent blocking/removal actions, it is generally acceptable to interchangeably use the terms "blocking" and "detection".

defined by a network protocol (e.g., TCP). In recent years, a series of adversarial attacks have emerged with the goal of evading such middleboxes; these DPI evasion attacks largely involve subtle manipulations of packets to cause different behaviours at DPI and end hosts, to cloak malicious network traffic that is otherwise detectable. As discussed previously, these adversarial actions do not depend on coarse-grained differential reactions that are conditioned on the overall DPI configuration (i.e., whether it is on or off); rather, adversaries manipulate packet headers based on what previous packets they observe. To decipher and analyze such stealthy manipulations, in Chapter 5 I propose a fully-automated, unsupervised ML solution that leverages *context learning* models for extracting *packet context* that is used to discover evasion packets. At a high level, ML models are trained to amplify and learn minute interrelationships among packet headers in connections, and rely on these relationships to discriminate adversarial packets from benign ones. I present more details and how one can generally tackle adversarial actions against network infrastructures in Chapter 5.

Chapter 2

Measuring and Disrupting Anti-Adblockers Using Differential Execution Analysis

2.1 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 [125, 143]. Advertisers track users across the Web without providing any transparency or control to users [68, 72, 91, 100, 122]. The popularity of adblockers is also rising because they provide a clean and faster browsing experience by removing excessive and intrusive ads.

Millions of users worldwide now use adblockers [130] 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 [28], 11% of the global Internet population is blocking ads as of December 2016. A recent study by comScore [96] 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 [14]. 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 [104] 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 [126] 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.

2.2 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 [17]. 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 [22].

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 [28], 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 [14, 133] 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 [71] 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 [14]). 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 [127] 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 [13]) and industry self-regulations (e.g., AdChoices [30]). Researchers [127] 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 [108]. 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 [104] 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 [12] and Adblock warning removal list [19]. 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 [77] 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 [104]. 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 [104].

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

[127]. 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 [108] 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 [65, 84, 121]. Prior work has borrowed these techniques to automatically extract signatures for tracker [76] and anti-adblock [77] detection. Ikram et al. [76] proposed syntactic and semantic JavaScript static code features with one-class SVMs to detect tracking JavaScript programs. Iqbal et al. [77] 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 [104] 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.

2.3 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 [77, 104, 105, 108] 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 [104] 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 §2.5.2). In this section, we formulate the anti-adblock detection problem. We then present the blueprint for a 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 2.1. 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 §2.4.2). 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.

```
<div class="banner_ads"> </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>
```

Figure 2.1: A simple anti-adblocking example from a real website

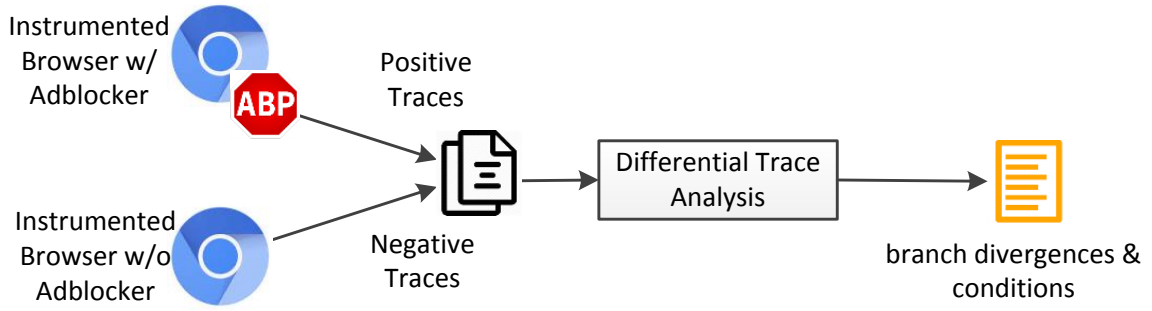


Figure 2.2: System overview

Given two traces, we define two types of execution differences [83]: *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 [104] 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 2.2 illustrates the overview of our proposed system of differential trace analysis. First, we instrument the open-source Chromium [27] 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 §2.4). We discuss other details of the instrumentation later in §2.4.1. 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 §2.5, we conduct both large-scale and small-scale evaluation and analysis of the identified anti-adblocking scripts. Finally, in §2.6, we show how one can apply the learned knowledge and use it against anti-adblockers.

2.4 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 checked that they yield similar results from our differential execution analysis.

2.4.1 Chromium Instrumentation

Prior work has proposed several instrumentation approaches to collect JavaScript execution traces. These include JavaScript rewriting [142], JavaScript debugger interface [75], 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 [24]). 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 [104]. For trace alignment (see §2.4.2), we also record all function call/ret statements and call stack information for all branch statements to include their calling context.

2.4.2 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 [16] 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 [104]). 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 [104], 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 [83, 139]. 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 [106]. 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 2.1, 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 [133], 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!
    }
});
```

2.5 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.

2.5.1 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 [104]. 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 differ-

ent 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* [83] 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).

```
1 if( window.advertsAvailable ===
    undefined ){
2   //adblocker detected, show
      fallback
3   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"/>');
4   jQuery('#testreplace').css('
      display','block');
5 }
```

(a) Switching ad sources upon detecting any adblocker

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

(b) Reporting adblocker usage through Google Analytics

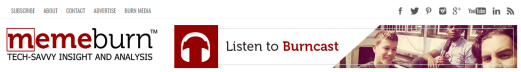
Figure 2.3: Silent anti-adblocker (Left: `memeburn.com`. Right: `englishforum.ch`)

As mentioned earlier in §2.4.2, 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 §2.7.

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

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



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

Figure 2.4: Ad switching behavior on `memeburn.com`

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 2.3.

1. Switching ad sources: As shown in Figure 2.3(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 2.4 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 2.3(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.

2.5.2 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 [104, 108]. An earlier study [108] 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 [104] 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 [104] 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 [108] relied on manual analysis and may miss

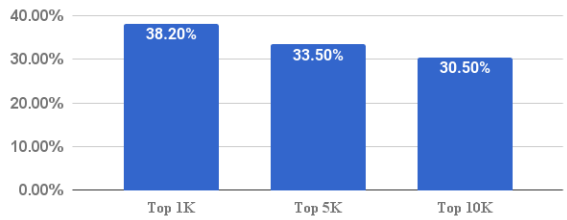


Figure 2.5: Popularity of anti-adblockers by website ranking

some anti-adblockers that do not have obvious keyword in the scripts (e.g., obfuscated). Moreover, the anti-adblock prevalence has likely increased [77] since more than a year ago [108] 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.

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 §2.5.1, 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 2.5 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 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 [104, 108] 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 [26]. To better understand the small set of third-party anti-adblocking scripts, we aggregate their sources using the domain and URL information. Table 2.1 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 2.1 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 §2.6.2 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 [11, 15]. In particular, PageFair [104] 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

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

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 }
```

Figure 2.6: Taboola's anti-adblocking script snippet

Next, we illustrate the anti-adblocking logic for Taboola, another big play in the field, in Figure 2.6. 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 as baits. Upon the detection of them being blocked, `isBlockDetectedOnClass-Names()` 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)
```

Figure 2.7: Outbrain's anti-adblocking script snippet

Finally, `Outbrain`'s anti-adblocking script is illustrated in Figure 2.7. 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 [66], 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 [18].

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 [104]. 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.

```
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);
```

Figure 2.8: First-party anti-adblocking script in `www.businessinisder.com`

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 2.8. 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.

```
1 BlockAdBlock.prototype.on = function(detected, fn) {
2   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}
```

Figure 2.9: First-party anti-adblocking script in nytimes.com

```
1 var n=document.getElementById(t);
2 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()}
```

Figure 2.10: Third-party anti-adblocking script in aol.com

nytimes.com has a first-party script that logs adblocker usage (see Figure 2.9. Similarly, aol.com includes a third-party script from blogsmithmedia.com that fires an

```
1 if (!window.isAdblockerDisabled) {
2   define('expads', function () {
3     var displayFallbackImage = function (slotConfig) {
4         ...
```

Figure 2.11: First-party anti-adblocking script in `expedia.com`

application event when adblocker is detected (see Figure 2.10). Finally, `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.

2.6 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 §2.2, 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.

2.6.1 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

```
1 if (hasBlock()) {
2   $('.notification').
        show();
3 }
```

(a) Original code

```
1 if (false) {
2   $('.notification').
        show();
3 }
```

(b) Condition rewrite (replacement)

```
1 if (hasBlock() &&
      false) {
2   $('.notification').
        show();
3 }
```

(c) Condition rewrite (append-only)

Figure 2.12: Choices of condition rewrite

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 2.1 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 [86]). 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 2.12 illustrates the differences. Figure 2.12(a) shows the original JavaScript code that attempts to detect adblockers. Figure 2.12(b) and Figure 2.12(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 2.12:

```
1 if (hasBlock() && matches(StackTrace.getSync(), recorded_stacktrace) &&
      false) {
2   $('.notification').show();
3 }
```

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 [21]. Thus, this approach may not always work

```
1  function checkAdVisible() {
2    if(isVisible(ad)) {
3        // pass
4      } else {
5        // penalizing user
6      }
7  }
8  function isVisible(obj) {
9    return (obj.offsetWidth == 0 && obj.offsetHeight == 0)? False: True;
10 }
```

Figure 2.13: Nested branch divergence example

even though most modern browsers such as Chrome and Firefox have some support for it [23]. 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 2.13, 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 [104]. It is possible that sophisticated anti-adblocking scripts (such as PageFair) will conduct multiple rounds of such checks.

```
1  $(window).load(function() {
2      if(typeof canRunAds != 'undefined'){
           // check level 1
3          var adFilled = $('.adsbygoogle')
               .find('ins').length;
4          if(adFilled!=0){ // check level
               2
5              // return okay
6          } else {
7              $('.layerPubAdBg').fadeIn();
8          }
9      } else {
10         $('.layerPubAdBg').fadeIn();
11     }
12 });
```

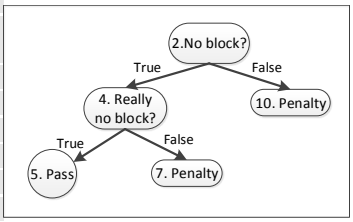


Figure 2.14: An example anti-adblocker with two levels of adblock detection

A simplified real example from `www.pandajogosgratis.com` is illustrated in Figure 2.14. 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 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 [25]. 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 §2.5.1), 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 §2.5.1. The failed cases are mostly due to the same reasons as outlined in §2.5.1. Only one website is found to have broken functionality where the JavaScript is mistakenly considered to be disabled.

2.6.2 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, 127]. 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:

```
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:

```
1 // intercept access to window.adblockV1, and always return true;
2 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 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.

[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`.

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 §2.5.2). 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 2.3(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,

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

since our solution is readily deployable in a standard browser extension, it has potentials to influence the future design of adblockers.

2.7 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 §2.4.1). 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 §2.6.2, 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 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.

2.8 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.

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 §2.6.2

Table 2.1: Top origins of anti-adblocker scripts based on different sources

Chapter 3

Lightweight and Stealthy Adblocking

3.1 Introduction

The deployment of adblocking technology has been steadily increasing over the past few years. PageFair reports that more than 600 million devices globally use adblockers as of December 2016 [28]. Many reasons contribute to the popularity of adblocking. First, lots of websites show flashy and intrusive online ads that negatively impact user experience. Second, the pervasiveness of targeted or personalized ads has incentivized a global ecosystem of online trackers and data brokers, which in turn raises concerns for user privacy. Third, the inclusion of numerous advertising and tracking scripts causes excessive website bloat resulting in slower page loads. The rise of adblocking has jeopardized the ad-powered

business model of many online publishers. For example, U.K. publishers lose nearly 3 billion GBP in revenue annually due to adblocking [29].

In response to adblocking, many publishers have deployed JavaScript-based, client-side anti-adblockers to detect and circumvent adblockers. An anti-adblocker typically consists of two components: detection and reaction. For adblocker detection, common practices include checking the absence of ad elements and proactively injecting bait ad elements [103]. Both practices exploit the fact that adblockers make observable changes to the DOM by either blocking relevant requests or hiding DOM elements directly [128]. As a result, these DOM changes can be perceived by the detection part of anti-adblockers through invocation of JavaScript APIs such as `getElementById()`. After adblocker detection, the reaction component can perform different subsequent operations. It can be aggressive paywalls that prevent users from accessing the content or even switching ad sources.

Adblockers have addressed anti-adblockers in one of the following three ways: (i) blocking the JavaScript code of anti-adblockers using filter lists [78], (ii) disrupting anti-adblocker code based on program analysis [146], and (iii) hiding the trace of adblocking to fool anti-adblockers [128]. The first countermeasure is currently adopted by the adblocking community using filter lists such as Anti-Adblock Killer and Adblock Warning Removal [78]. However, the coverage and accuracy of these filter list is lacking. Our manual evaluation on 207 websites using anti-adblockers with visible reactions (i.e. warning message or paywall), only less than 30% of them are correctly identified by Adblock Warning Removal [32] or Anti-Adblock Killer [33]. This is likely due to the manual nature of filter list curation and maintenance which is cumbersome and error-prone. The second countermeasure of

rewriting JavaScript to deactivate anti-adblockers is prone to false positives causing site breakage with unacceptable user experience degradation [146]. The third countermeasure of hiding the trace of adblocking, as implemented in prior work [128], is not stealthy because it injects new JavaScript which is easily detectable by anti-adblockers.

In this paper, we aim to completely hide the traces of adblocking in a stealthy manner by going deep into the browser core. This is analogous to the rootkits in the OS kernel where user applications are unable to detect the presence of a malicious process [123]. Since the browser core is at a lower level (more privileged), it is in theory capable of hiding the states of adblockers from the anti-adblockers while presenting an ad-free view to the user. Specifically, since anti-adblocker is implemented as client-side JavaScript, it can only access web-page states through a number of predefined Web APIs including the ones used to probe the the presence/absence of ad elements. These APIs are standardized by W3C and implemented eventually by web browsers. ShadowBlock hooks any JavaScript API that can potentially distinguish the difference before and after hiding ad elements, and assures no information about the element hiding can be leaked through such API.

We tackle three major challenges in designing and implementing ShadowBlock. First, existing adblockers' model of blocking ad-related URLs (e.g., scripts, iframes, images) does not fit well in our requirement of presenting the same exact DOM view as if no adblocker is employed. For example, if a DOM element is not even retrieved, ShadowBlock would have no way to fake its size, dimension, and other properties. Second, given that the Web APIs suite and the rendering process implemented in modern web browsers are highly complex and intertwined, there may exist unexpected channels that leak information about

adblocking deployment. To achieve stealthy adblocking, we need to ensure that no such channel discloses differentiable information about our element hiding action in a conclusive way. Third, modern web browsers make significant efforts in improving their page loading and rendering performance. As we develop ShadowBlock on open-sourced Chromium, we need to minimize the overhead it incurs during the page load process.

Contributions. We summarize our key contributions as follows.

1. We design a well-reasoned solution where we present two different views to anti-adblockers and users. On one hand, ad elements are never directly blocked (so they remain visible to anti-adblockers); on the other hand, these ads are stealthily hidden from users.
2. We reuse the rules from filter lists used by adblocking browser extensions to element hiding decisions. On top of existing lists that are community-backed and have been widely adopted, we replicate 98.3% of their ad coverage according to manual inspection over Alexa Top 1K websites, with less than 0.6% breakage rate.
3. We design and implement a fully functional prototype which is open-sourced at the time of publication. We evaluate the effectiveness of ShadowBlock prototype on 207 websites with visible anti-adblockers. We pick real anti-adblockers with different trigger mechanisms and of different complexity. All of them are successfully evaded by our new adblocker design.
4. Our performance evaluation of ShadowBlock shows that it loads pages comparably fast as Adblock Plus on average, in terms of Page Load Time and SpeedIndex.

3.2 Background and Related Work

3.2.1 Adblockers And Filter Lists

Mechanisms. Adblockers rely on manually curated filter lists to identify ads on web pages. EasyList and EasyPrivacy are two most widely used filter lists to block online ads and trackers, with about 71000 and 15000 rules [20], respectively. These lists consist of two types of rules which are basically regular expressions. One of them is HTTP rules that block HTTP requests to URL addresses that are known to serve ads. For example, the first filter rule below blocks all third-party HTTP requests to `amazon-adsystem.com`, preventing any resource on this domain from being downloaded. The other type is HTML rules, which generally hides HTML elements that are identified as ads. For example, the second filter below hides all HTML elements with ID `promo_container` on `wsj.com`.

```
||amazon-adsystem.com^$third-party
wsj.com###promo_container
```

It is noteworthy that HTML rules are mostly only introduced to complement HTTP rules while dealing with first-party text ads. This is because these text ads are directly embedded into the HTML itself with no associated additional resource loads, making it inevitable to be included while the browser downloads the web page. Otherwise, HTTP rules are preferred as they prevent ad resources from being loaded in the first place, saving unnecessary network traffic and avoiding execution of ad-related scripts to speed up page loading and rendering.

Limitations. A group of volunteers that maintain filter lists carry out the manual process to add new rules, correct and remove erroneous or redundant rules all based on

informal feedback from users [35]. Due to its crowd-sourced nature, this laborious effort faces challenges from both the completeness and soundness. In the context of adblocking, the former results in missed ads and the latter often translates to site breakage or malfunction [49]. At the same time, as adblockers gain their popularity rapidly (11% of global Internet population is blocking ads as of December 2016 [28]), online publishers are also fast adopting countermeasures against adblockers that we summarize below.

3.2.2 Countermeasures Against Adblockers

Concealing ad signatures. First, online advertisers can bypass rules on filter lists by concealing the signatures these lists use to identify ads. At a high level, this line of countermeasures attempts include first-party advertising and rotation of ad serving domains. First-party advertising exploits the fact that many rules on filter lists are designed to block ads from being loaded from third-party servers. Instead, ads are served from the same domain of the web page hosting them and their nature as ads is concealed as normal content [38]. However, adblockers can easily hide any HTML element on a web page by applying HTML rules that are crafted to target elements based on any combination of their CSS/HTML properties. In other words, any CSS selectors used to create and/or locate ads elements, can also let adblockers identify and hide these elements in turn.

Domain name rotation is another tactic for obfuscating advertising content. It relies on ad rotation networks that serve ads from frequently-changing, or even automatically generated [134] domain names, which overwhelms volunteers that maintain the filter lists so they are hard to keep pace with the rule updates. This can result in, however, the indifferent

blocking of all third-party resources on websites that show such ads [36]. By only whitelisting legitimate scripts that support core functionalities, any possible ads or tracking JavaScript are prevented from running, leaving no chance for loading domain rotating ads. Moreover, AdBlock Plus recently launched its Anti-Circumvention Filter List [31] that specifically counter "circumvention ads", including ads adapting the two countermeasures above.

Deploying anti-adblockers. Second, many publishers choose to deploy anti-adblocker JavaScript code to battle the rise of adblocking. Specifically, such client-side scripts consist of two main components, *trigger* that detects the presence of adblockers by checking whether ads or bait elements are still present, and *reaction* that can display warning messages and/or simply report the results to a remote server [103, 146]. Prior work [78] showed that 686 out of Alexa top-100K websites detect and visibly react to adblockers on their homepages. Even worse, these visible ones only account for less than 10% of all anti-adblockers [146]. Zhu et al. [146] showed that among Alexa top-10k websites, 30.5% are countering adblockers in some form, with most of them being silent reporting. In summary, because of their flexibility and ease of deployment, anti-adblockers are considered the most widely used countermeasure against adblockers adopted by online publishers.

3.2.3 Countermeasures Against Anti-adblockers

Dedicated anti-adblocking filter lists. As a response, adblockers attempt to circumvent anti-adblockers by blocking their JavaScript code snippets, whitelisting bait scripts/elements, or hiding warning notifications. To this end, adblockers once again rely on manually curated filter lists such as Anti-Adblock Killer [33] and Adblock Warning Removal [32].

These lists either trick anti-adblockers' trigger so they cannot detect adblockers, or mute their reaction component to prevent responses after successful detection.

```
/kill-adblock/js/function.js$script
@@||removeadblock.com/js/show_ads.js$script
ilix.in,urlink.at,priva.us###blockMsg
```

For example, the first HTTP rule above blocks the code snippets containing implementation of an anti-adblocking library `KillAdBlock`, and the second whitelists a bait script file named `show_ads.js` that is used to detect adblockers. The third HTML rule hides the warning message with ID `blockMsg` issued by the associated anti-adblocker. However, our manual evaluation (§4) shows that these lists targeting anti-adblockers are generally ineffective. Only less than 30% of the anti-adblocking warning messages can be removed by the state-of-the-art filter lists. This is again partly because of the crowdsourcing nature of these lists, and also the rising popularity of third-party anti-adblocking services that deploy sophisticated techniques dedicated for detecting/circumventing adblockers [103].

Disrupting anti-adblocker code. Other than the filter lists that have been officially adopted by adblockers, there are also research efforts for detecting and evading anti-adblockers. One solution to measure the anti-adblockers is to perform program analysis techniques that automatically determine if a script functions for anti-adblocking purposes. Such analysis can be static that is based on syntactic and structural features extracted from JavaScript code, and utilizes machine learning approaches to classify the code from ground-truth-labeled training data [78]. It can also be dynamic that captures JavaScript behavior at runtime by collecting and analyzing differential execution trace with the adblocker turned on and off [146]. After successfully pinpointing the critical conditions that are used

by anti-adblockers to assert/react against the presence of adblockers, one can choose to rewrite these conditions to prevent the anti-adblockers from functioning. This approach is generally intrusive (patching Javascript can be tricky and cause breakage) and easy to evade. Indeed, the overall success rate of this strategy is shown to be only less than 80%.

Hiding adblockers. Besides disrupting the functionalities of anti-adblockers, researchers also have proposed a way to hide the trace of using adblockers, or known as *stealthy adblocking*. In [128], Storey et al. created a shadow copy of the DOM that anti-adblockers operate on before any adblocking actions take place, and then redirects all JavaScript APIs (e.g. `getElementById()`) that can be used to detect the presence of ad elements to the copy instead of the original DOM. However, this so-called rootkit-style stealthy adblocker has inherent drawbacks. First, unless it lives in browser core and with significant engineering efforts, the underlying DOM mirroring and propagation are difficult to be complete in all cases. This is especially problematic in the context of web browsing as any site breakage causes unacceptable user experience degradation. Second, even with a perfect implementation, maintaining a live copy of complicated data structures such as DOM poses a prohibitively high overhead onto the rendering performance of modern web browsers. Given that modern browsers place significant emphasis on performance, heavy operations like such at runtime are generally not acceptable.

3.3 ShadowBlock

In this section, we first provide an overview of ShadowBlock's architecture. We then discuss ShadowBlock's two building blocks: (1) the identification of ad elements by

translating filter list rules to per-element hiding decisions and (2) the concealment of our hiding actions. Finally, we summarize the modifications we make in the relevant modules of Chromium.

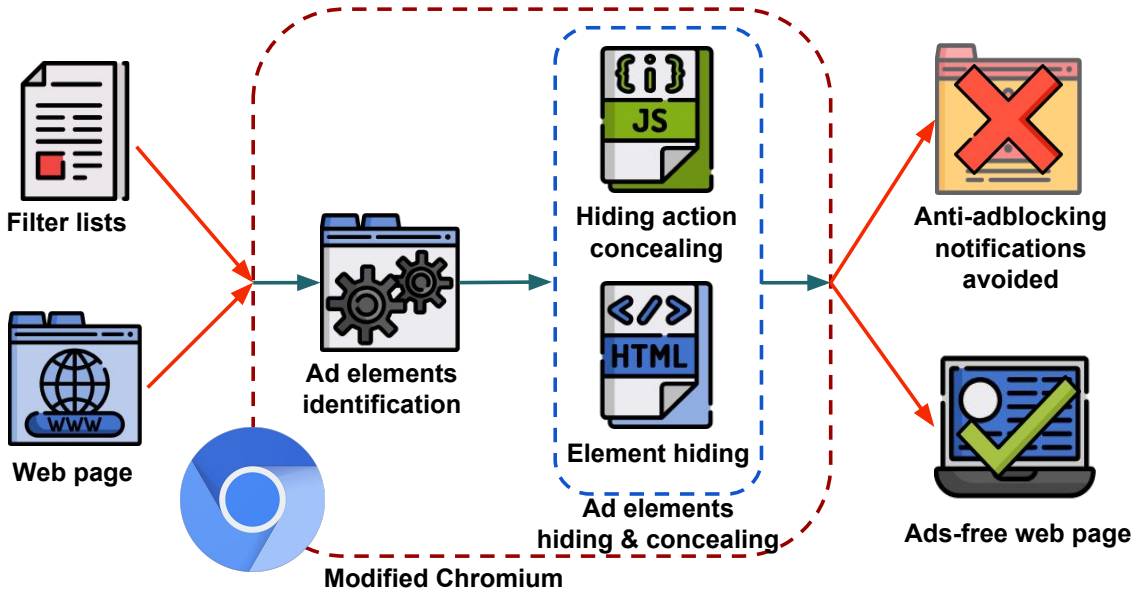


Figure 3.1: Architectural overview of ShadowBlock

3.3.1 ShadowBlock Overview

Figure 3.1 illustrates ShadowBlock's architecture. It consists of two sub-systems: one translates rules from filter lists and use them for identifying ad elements in DOM to hide; the other hooks necessary points in Chromium to ensure that the hiding actions are transparent to the trigger/detection component of anti-adblockers. Recall from Section 3.2.1 that filter lists contain tens of thousands of rules that either block HTTP requests to fetch ad resources or hide HTML ad elements. To prevent exposing adblocking actions to anti-adblockers, we need to hide the changes in DOM or other states (e.g. resource loads)

introduced by adblocking because these changes can be detected by anti-adblockers through JavaScript APIs such as `getElementById()`. In order to do so, we first allow all HTTP requests to proceed, then mark any element that results in ads, and subsequently hide the marked elements. It is important that we allow these elements to be retrieved so when the anti-adblocking script queries the state of the element, ShadowBlock will be able to generate valid responses (e.g., dimension of the element).

3.3.2 Identifying Ad Elements

Next, we explain our approaches for marking different types of ad elements. In general, there are two types of elements: (1) those that are statically embedded in the HTML, and (2) those that are dynamically created by JavaScript. First, some ad elements are statically embedded in the HTML, including ad images, iframes, media files (i.e. video and audio). Such ad elements can be typically identified by matching against the HTML rules on filter lists. Second, ad scripts usually *create* new ad elements that display advertising content. These dynamically created elements should be identified, marked, and hidden.

Ad elements loaded statically. ShadowBlock does not need to do anything special for such elements. Ad filter lists already contain extensive rules that cover them. For example, the rule `||googlesyndication.com/safeframe/` in EasyList blocks the HTML file from `https://tpc.googlesyndication.com/safeframe/1-0-23/html/container.html` on site `cnbc.com`, which prevents a container frame used by Google Ads from being loaded. To hide such an ad element, ShadowBlock needs to first identify the iframe element with the blocked URL as its source attribute and then hide it.

Alternatively, Easylist rules may hide ads based on element properties (e.g., element id). We simply reuse these rules to match elements in the page, and mark them accordingly.

Ad elements generated dynamically by ad scripts. There are generally two cases. The easy and the hard. For the easy case, the dynamically generated ads may contain URLs or ids that already show up on Easylist. This allows us to directly mark them as ads using very much the same strategy as mentioned above.

For the hard case, the dynamically generated elements are not on Easylist. This is because it is assumed that ad scripts are blocked upfront and therefore there is no need to block the elements generated by them. In ShadowBlock, in contrast, we need to allow ad scripts to load and execute, which mandates us to track such elements.

To identify elements dynamically created by ad scripts, we need to attribute each element to the script that created it. More formally, we can define this process of attribution as tracking the *data provenance* of HTML elements using taint analysis. Note that there are in general two types of element creation that can be initiated by a script: (1) control-flow-based creation, in which the script directly invokes JavaScript API (e.g. `createElement(tagName)`) and only propagates data from itself into the new element; and (2) data-flow-based creation, in which the script uses data from sources other than itself into the new element (e.g. `createElement(fetchTagNameFromServer())`). Dynamic taint analysis [85] can accurately track the data provenance through taint propagation for both types. Simply put, taint analysis involves *taint source* (where data comes from), *taint sink* (where data ends), and *propagation policies* that define how that tainted data are propa-

gated through the program execution. In our case, all data derived from an identified ad script as "tainted" (i.e., data downloaded through an ad URL, or retrieved/generated by an ad script), then such data can be tracked standard taint analysis propagation policies for JavaScript (e.g., [60]). Finally we will hide any tainted HTML elements (i.e. taint sinks).

Unfortunately, such dynamic taint analysis at runtime usually incurs significant overhead for web browsing [85]. In addition, we argue that the cases where taint analysis will be required are limited. Specifically, even if ad elements take dynamically fetched data, e.g., `createElement(fetchTagNameFromServer())`, they are most likely created by ad scripts. This is sufficient for us to mark the element and hide it (irrespective of what data are actually fetched from the server). The reasoning is that if we consider extension-based adblockers as our baseline, the dynamically created element wouldn't even exist in the first place (as the script as a whole would have been blocked). Based on the above, we devise a simple technique which we call *execution projection* using the call stack information extensively (which are used for various other purposes as well [79, 92]). At a high level, we maintain an "execution stack" that keeps track of what scripts are being executed at any given time point, and mark an element as ad if there is *any* ad script in the stack when the element is being created. For example, consider a simple ad script `ad_loader.js` in Code 3.1.

```
1 var ad_img = document.createElement("img");
2 ad_img.src = "https://some_ad_publisher.com/ad.jpg";
3 document.body.appendChild(ad_img);
```

Code 3.1: Example ad script `ad_loader.js`

In this example, at the time of the image element creation, `ad_loader.js` would be at the top of the execution stack because it is being executed.

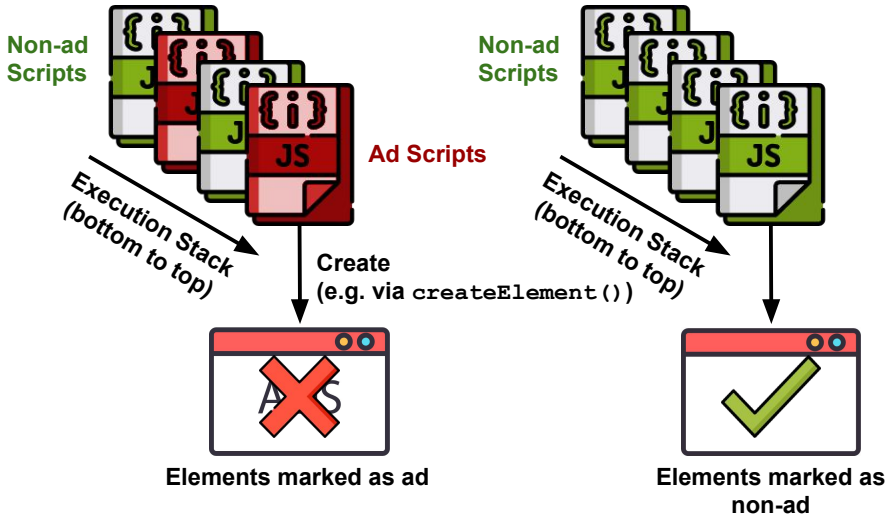


Figure 3.2: Execution projection for marking script-created ad elements

After attributing elements to ad scripts, which are identified using filter lists, we can mark all ad elements and hide them accordingly. We illustrate this projection/marking process in Figure 3.2. In more complex cases, there can be many scripts in the executing stack, because code in one script can invoke functions in other scripts, and so on. More importantly, ad scripts can invoke non-ad libraries (e.g. jQuery) to create ad elements so the top script in stack at the time of element creation is not necessarily ad script. To tackle this challenge, we need to scan the entire execution stack. If there is any ad script in the execution/invocation chain, ShadowBlock should mark the element being created as ad. This is because any script (ad-related or not) invoked by a known ad script should never have been executed in the first place, given that adblocking extensions simply block the whole ad script altogether.

Note that our approximate solution does not handle a special case when an element is created via a JavaScript API that gets overridden (hooked) by an ad scripts . Since client-side JavaScript is allowed to override (i.e. injecting code containing a callback to its own)

arbitrary JavaScript API functions (e.g. `createElement()`) with its own version, we would see ad script in the stack when an element is being created via the API overridden by an ad script. In this case, we may mistakenly mark a non-ad element as ad when the overriding ad script does not propagate any data into the newly created element. To tackle this issue, at the time of element creation, we would need to further check whether the code injected by the overriding ad script alters the element. If it alters the element then we mark the element as ad, and non-ad otherwise.

```
var original = document.createElement;
document.createElement = function (tag) {
  new_elem = original.call(document, tag);
  report_statistic_to_server(new_elem);
  return new_elem;
};
```

Code 3.2: API overriding with the element intact

```
var original = document.createElement;
document.createElement = function (tag) {
  new_elem = original.call(document, tag);
  ad_elem = change_to_ad_elem(new_elem);
  return ad_elem;
};
```

Code 3.3: API overriding with the element altered

3.3.3 Stealthily Hiding Ad Elements

After identifying ad elements, ShadowBlock needs to stealthily hide them so as to not leaving its trace to anti-adblockers. Next, we discuss how we realize such stealthiness through CSS property access API hooking.

Choice of hiding mechanism. We first need to decide how to hide ad elements within Chromium. Given the complexity of modern web browsers and API standards, we can hide an HTML element in several different ways. To better understand different hiding mechanisms, we illustrate Blink's rendering process in Figure 3.3 [43]. Blink's rendering

path, from parsing an HTML file to the pixel display on user's screen, can be summarized in the following phases.

1. **Parse** flat HTML and CSS in plain-text to DOM and CSS Object Model (CSSOM) in tree structure.

2. **Combine** DOM and CSSOM to Render Tree, which captures all the visible DOM content and all the CSSOM style information for each node.

3. **Paint** the rendered pixels to user's display according to Render Tree.

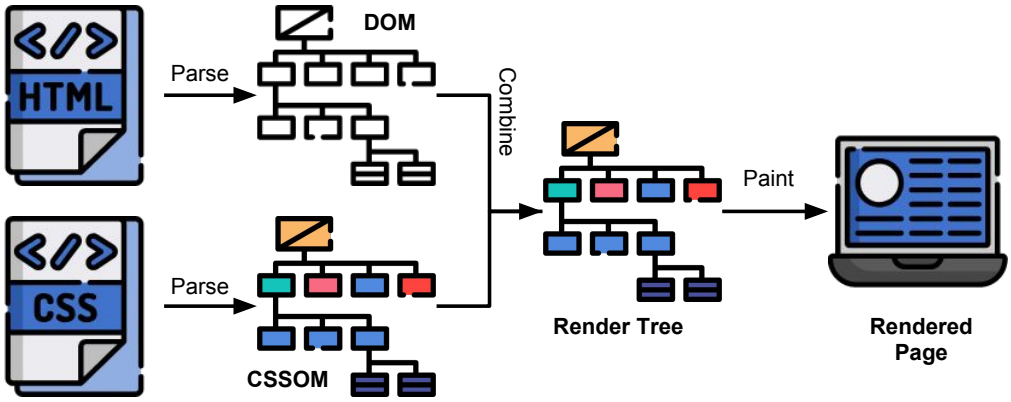


Figure 3.3: Rendering Path for Blink

In theory, each of these 3 phases contain APIs/modules that we can leverage for hiding an HTML element (or in other words, preventing it from being rendered). We outline different possible strategies to hide HTML elements below:

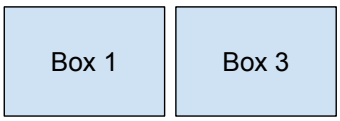
1. **DOM/CSS layer**: (i) remove the element from DOM; (ii) set the element's style to `display:none`, `visibility:hidden` or `opacity:0`

2. **Render Tree layer**: remove the `LayoutObject` (i.e. a styled node) from Render Tree

3. **Paint layer**: prevent the region in pixels associated with the element from being painted

Note that generally, the higher-layer (i.e. closer to DOM/CSS layer) we tweak around in the hierarchy, more engineering effort is required for ensuring its stealthiness against anti-adblockers, while narrower gap we have to bridge for translating the identified ad HTML elements to the data structure corresponding to that layer (e.g. CSS properties for DOM/CSS layer, `PaintBlock` for Paint layer etc.). In contrary, the lower-layer (i.e. closer to Paint layer) our modifications reside, fewer unexpected channels there are that can potentially leak the hiding activities, but at the same time more engineering efforts are required for translating the identified ad elements to that layer's data structure.

After investigating different possibilities, we find the CSS property `visibility:hidden` achieves the most suitable trade-off for our objective. It persists in phase (1), functions in phase (2) of the rendering path, and eventually affects both Render Tree and painted/rendered page in phase (2) and (3). On one hand, `visibility` is a property associated with every HTML element so we can easily identify after marking its effective element as ad, without the need of further tracing. On the other hand, unlike `display:none`, it by design preserves the space taken up by the hidden element so it causes no side-effect to the layout of the document, minimizing the impacted points that need to be hooked for covering the hiding action. Figure 3.4 shows the visual difference between their respective effects. Compared to `opacity:0` that also preserves the occupied space, `visibility` is a categorical CSS property instead of numerical as `opacity` so its implementation in Chromium is considerably less complex, simplifying our hooking logic as well.



(a) `display:none`

(b) `visibility:hidden`

Figure 3.4: Comparison of toggling different CSS properties (Box 2 is hidden)

Hooking for concealing hiding actions. After hiding the target ad element by setting its `visibility` CSS property value [47] to `hidden`, we next need to cover any traces that result from this change of value and can be detected by anti-adblockers. Since anti-adblocking scripts are client-side JavaScript code, which can only access the change of states happened in the page through JavaScript Web APIs [48], we search through the source code of Chromium, analyze relevant modules and locate the following three categories of APIs that are impacted by the `visibility` CSS property:

- **CSS/Style-related**: changing a CSS property value immediately affects its own return value to JavaScript APIs via `getComputedStyle()`. We hook this value to `visible` to fool anti-adblockers. Fortunately, for our case, setting `visibility:hidden` preserves the space the element occupies, so it does not collapse the page layout nor affect other relevant CSS properties such as `offsetHeight/offsetWidth`.

- **Event-related**: flipping the `visibility` property prevents an element from receiving some DOM events, such as `onfocus`. Since anti-adblockers can leverage these events

as a side-channel to infer the real visibility of an element, we hook relevant modules in Chromium so hidden elements can receive these events just like visible elements.

- **Hit-testing-related**: another effect of setting `visibility: hidden` is Blink treats elements with it as inapplicable to Hit Testing, which checks if an element is clickable by users in their viewpoints. This removes the hidden element from return values to APIs like `elementFromPoint()`, which can potentially be used by anti-adblockers to differentiate hidden elements from visible ones. We hook relevant modules in Blink to cover it.

Since all JavaScript APIs, to our best knowledge, directly or indirectly rely on the modules above to determine an element's visibility, we ensure the completeness of our hooking against potential information leakage to client-side anti-adblockers. It is worth noting that to avoid affecting existing `visibility:hidden` CSS property value, in Chromium we create a new `visibility:fake-visible` value that completely mimics what `visibility:hidden` behaves, except for the points that are intentionally hooked. Moreover, since `visibility` is a CSS property that inherits from parent node to child nodes, we can make the identified ad elements invisible to user's display even though we only identify the top-most element as ad according to filter lists.

3.3.4 Chromium Modification

Next, we describe our modifications to Chromium for implementing ShadowBlock. We start with a brief introduction of Chromium's architecture, then move to the instrumentation we use for identifying ad elements, and lastly discuss the modules we lever-

age for hiding identified ad elements and hook for eliminating the traces resulted from the hiding action. We implement the prototype of ShadowBlock with 1307 LOC (1265 LOC addition and 42 LOC deletion) in C++ on top of Chromium [1]. Note that we re-use `SubresourceFilter` [45] and `libadblockplus` [37] for parsing filter lists with production-level robustness.

Chromium Architecture. Chromium's rendering engine is called Blink and its JavaScript engine is called V8. Blink [34] is responsible for fulfilling the rendering path shown in Figure 3.3, in which its core module renders all HTML elements and handles their `visibility` CSS properties we use for hiding ad elements. V8 [24] handles the compilation and execution of all JavaScript code, including the ad scripts ShadowBlock needs to identify and anti-adblocker scripts that intend to detect our hiding action over ad elements. Blink has a bindings module to handle interactions between rendering and JavaScript execution. Rendering related script tasks are passed through bindings module. For example, JavaScript API calls such as `getComputedStyle()` are handled through the bindings module.

Instrumentation for identifying ad elements. As discussed in Section 3.3.2, there are three types of ad elements ShadowBlock needs to identify.

First, for ad elements generated by ad scripts, we rely on execution projection. In Chromium, we leverage `blink::SourceLocation::Capture` to capture the full `v8::v8_inspector::V8StackTrace` that includes the entire JavaScript call stack[2] at any given time

[1] We open source our implementation at `https://github.com/seclab-ucr/ShadowBlock` to allow reproducibility as well as help future extensions by the research community.

[2] We admit that there are cases where asynchronous tasks are not correctly captured by the default V8 call stack trace. However, we argue this incompleteness only translates to very limited number of missing ads according to our manual evaluation in Section 3.4.2.

point. It serves as the underlying stack tracing mechanism for V8's debugger/inspector, and has therefore been optimized with low overhead [46].

Second, we instrument the constructor of `blink::Element` class in Blink, which captures the earliest point of creation event for all HTML elements. Because of V8's single-threading nature, we can safely associate the current stack trace to the element creation event, and scan the stack to match the scripts in it against filter lists. If it is a match, we then mark the element as ad. Additionally, we instrument the `DispatchWillSendRequest` event in both `blink::FrameFetchContext` and `blink::WorkerFetchContext` to intercept the point when an ad script loads another script, and mark the loaded script as ad script. By adding such loaded ad scripts to a set, we match the stack trace against them as well at element creation, ensuring we can mark all ad elements.

Third, for elements loaded with resources that match rules in the filter lists, we intercept the `AttributeChanged` event in `blink::Element` and match the URL against filter lists, if it is a match we mark this element as ad. For element hiding rules, we adopt `libadblockplus` [37], a C++ wrapper library around the core functionality of Adblock Plus to parse filter lists and generate the CSS selectors for matching ad elements for a particular domain. Then, we mark the ad elements that match the generated CSS selectors by calling `ContainerNode::QuerySelectorAll()`.

Since many web pages are dynamic due to JavaScript execution over time, we also need to monitor attribute changes of each element. For this purpose, we instrument the `AttributeChanged` event and match any element with newly changed attributes against CSS selectors from HTML rules. We mark an element as ad if it is a match, or un-mark the

element if this element has been marked but it is not matched this time. Note that in order for minimizing the number of matches needed to perform, we conduct the first batch match (via `QuerySelectorAll()`) after the `load` event of DOM is fired, and then match elements upon their attribute changes. This design choice leaves a short period of time (few milliseconds) between page navigation and `load` DOM event in which ads are displayed. We make this trade off to reduce the overhead incurred by `QuerySelectorAll()`. In comparison, adblocking extensions such as Adblock Plus inject CSS rules when `document.readyState` turns `interactive` [52], which happens before the `load` event. However, it is important to note that most ads in current web ecosystem are loaded in an asynchronous manner and are unlikely to appear before the `load` event in first place.

Stealthy modifications for hiding ad elements. As mentioned earlier, we leverage `visibility` CSS property to hide identified ad elements by creating a new `fake-visible` enumerate and visually hide elements with this enumerate, as if it behaves as `hidden`. In the meantime, we hook relevant modules in both Blink and its bindings with V8 to ensure the stealthiness of our hiding action. More specifically, for eliminating traces accessible by CSS/Style-related APIs, we hook `CSSComputedStyleDeclaration::GetPropertyCSSValue` in Blink and force return `visible` to queries about hidden elements. For event-related APIs, we hook `Element::IsFocusableStyle()` and other conditions that determine if an element can receive events. Lastly, we hook `ComputedStyle::VisibleToHitTesting()` so ad elements are still regarded "visible" from the viewpoint perspective of Blink. In principle, our hooking guarantees that the identified ad

elements are invisible to user's display as pixels on screen but appear as visible to APIs accessible to client-side JavaScript.

3.4 Evaluation

We evaluate ShadowBlock in terms of its (1) stealthiness against anti-adblockers, (2) ad coverage, and (3) performance as as compared to adblocking extensions.

3.4.1 Stealthiness Analysis

Takeaway: ShadowBlock has 100% success rate against anti-adblockers whereas state-of-the-art anti-adblocking filter lists have only 29% success rate.

Experimental Setup. To evaluate the stealthiness of ShadowBlock, we use previously reported [146] 682 websites with visual anti-adblockers. We manually analyze these websites and find that 207 of them still use visible anti-adblockers. For each website, we perform stealthiness comparison as follows.

1. Open a website with four Chromium instances simultaneously. Each instance has a different profile configuration: (i) no modification or extension; (ii) Adblock Plus extension with EasyList only; (iii) Adblock Plus extension with EasyList, Anti-Adblock Killer list, and Adblock Warning Removal list; and (iv) ShadowBlock using EasyList.

2. Scroll the page down to the bottom and wait for 30 seconds after the `load` event has fired to ensure complete page load.

Tool	Notification	Ad switching	Crypto-mining
Total	201	5	1
ShadowBlock	201 (100%)	5 (100%)	1 (100%)
Filter lists	59 (29%)	1 (20%)	0 (0%)

Table 3.1: Breakdown of stealthiness analysis

3. Capture the full-page screenshots (including content after scrolldown) for all browser instances.

4. Manually inspect the screenshots: compare (i) and (ii) to determine if the page has visual anti-adblocker. If so, further compare (ii) and (iii) to check whether anti-adblocking filter lists evade the anti-adblocker, compare (iii) and (iv) to check whether ShadowBlock achieves the evasion.

Results. In addition to visible anti-adblocking notifications, we also consider ad switching and crypto-mining reactions from websites. Table 3.1 compares ShadowBlock with anti-adblocking filter lists for each of these anti-adblocking reactions. "Notification" refers to websites that show anti-adblocking notifications such as paywalls. "Ad switching" refers to websites that switch their ad sources upon detection of adblockers. "Crypto-mining" refers to the websites that load crypto-mining scripts to mine crypto-currencies on detection of adblockers [74]. We note that ShadowBlock has 100% success rate as compared to 29% success rate of anti-adblocking filter lists.

(a) Original ad

(b) Replaced ad

Figure 3.5: Ad switching behavior on `golem.de`

Case Studies. Below we discuss a few interesting examples of anti-adblockers that ShadowBlock successfully handles but filter lists do not. Note that besides visible anti-adblocking notifications, we also include one example discovered in the wild that uses non-visual countermeasure against adblocker users.

Ad source switching. On detecting adblockers, some websites switch their ad sources to sources that are currently not blocked by filter lists. Figure 3.5a and 3.5b show an example from `golem.de`. Since ShadowBlock stealthily hides the original ads, the ad source switching script is never triggered. Therefore, unlike what Figure 3.5b shows in which adblocking extensions fail to remove the replaced ad, ShadowBlock successfully hides it.

Silent reporting. Besides visible reaction, anti-adblockers can also choose to silently report the adblocking status to back-end servers to collect adblocking statistics. For example, `varmatin.com` uses Code 3.4 to place a bait with keywords on EasyList to track adblocking

users and report the status to back-end server. ShadowBlock handles such cases and these statistics are never reported.

```
function checkAds() {
  if ($(document.getElementById('adsense')).css('display') !== 'none' && $('
      #myadsblock').length === 1) {
    dataLayer.push({
      'DimAdBlock': 'Unblocked'
    });
    window.adblockdetected = false;
  } else {
    dataLayer.push({
      'DimAdBlock': 'Blocked'
    });
    window.adblockdetected = true;
  }
}
```

Code 3.4: Code snippet on `varmatin.com` of silent anti-adblocker

Crypto-currency mining. Some websites have started to employ cryptojacking as a response to adblocking [74]. To this end, websites use anti-adblockers to detect use of adblockers and load scripts to mine a crypto-currency on user's browser. Mining crypto-currencies consumes processing power on user's machine. For example, `knowlet3389.blogspot.com` blocks organic content on detection of adblockers and asks users of allow crypto-currency mining for monetization instead. Code 3.5 shows the crypto-currency mining script on `knowlet3389.blogspot.com`.

```
setInterval(function() {
  try {
    var a3 = document.getElementById('AdSense3');
    if (a3.offsetHeight < 33 || a3.clientHeight < 33) {
      throw "Fuck U AdBlock!";
    }
  } catch (err) {
    miner.start(CoinHive.IF_EXCLUSIVE_TAB);
  }
}, 5487);
```

Code 3.5: Code snippet on `knowlet3389.blogspot.com` for mining crypto-currency

3.4.2 Ad Coverage Analysis

Takeaway: ShadowBlock achieves 97.7% accuracy, with 98.2% recall and 99.5% precision in blocking ads on Alex top-1K websites.

Experimental Setup. For ad coverage analysis, we use ShadowBlock on Alexa top-1K sites and measure its accuracy in terms of true positive (TP), false negative (FN), true negative (TN), and false positive (FP). We define TP, FN, TN, and FP as:

TP: All ad elements on a page are correctly hidden.

FN: At least one ad element on a page is not hidden.

TN: No non-ad element on a page is incorrectly hidden.

FP: At least one non-ad element on a page is incorrectly hidden.

For each website, we perform ad coverage comparison as follows.

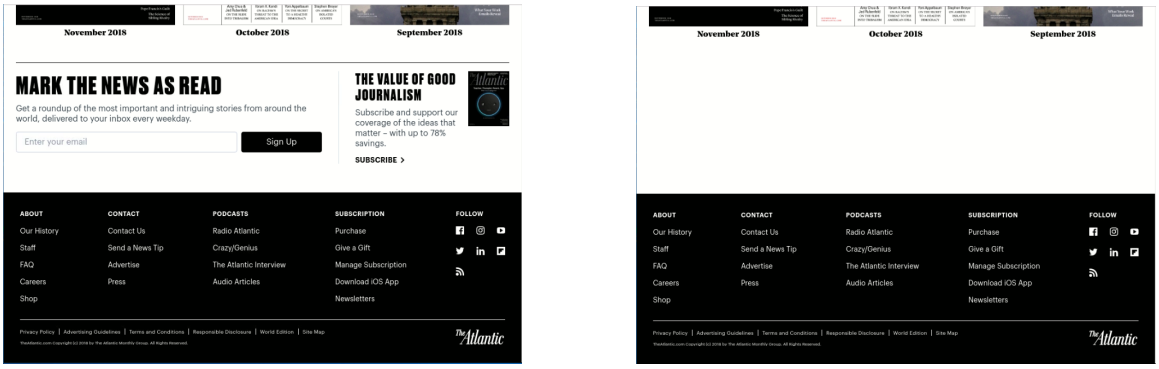
1. Open a website with three Chromium instances simultaneously. Each profile has a different profile configuration: (i) no modification or extension; (ii) Adblock Plus extension with EasyList; and (iii) ShadowBlock using EasyList.

2. Scroll the page down to the bottom and wait for 30 seconds after the `load` event has fired to ensure complete page load.

3. Capture the full-page screenshots (including content after scrolldown) for all browser instances.

4. Manually inspect the screenshots: compare (i), (ii) and (iii) to determine if the website has any FPs or FNs.

Results. Table 3.2 shows the breakdown of our manual analysis. We evaluate results in terms of TPs, FNs, TNs, and FPs. Note that we are able to perform our analysis on 943



(a) Adblock Plus (b) ShadowBlock

Figure 3.6: Minor visual breakage caused by ShadowBlock

out of Alexa top-1000 websites. The remaining websites failed to properly load primarily due to server-side errors (e.g., 404).

False Positive Analysis. From Table 3.2, we note that ShadowBlock has only 0.5% false positive rate. However, they are still critical as they might lead to user experience degradation. We further investigate false positives to diagnose their root cause.

`theatlantic.com` is an example of false positive. Figure 3.6 shows that ShadowBlock incorrectly hides organic content at the bottom of the page. On further investigation, we find that an ad script `ads.min.js` loads another script `script.js` that hooks JavaScript API methods. In this case, even when a non-ad element is being processed it would go through same hooked JavaScript API methods. Since our ad marking heuristics

Event	TP	FN	TN	FP
Count	926 (98.2%)	17 (2.8%)	938 (99.5%)	5 (0.5%)

Table 3.2: Breakdown of ad coverage analysis

check for the presence of ad scripts on execution stack it will incorrectly mark such elements as ads. In comparison, extension-based adblockers block the request for downloading `ads.min.js` in the first place, so the hooking script never gets executed. As discussed in Section 3.3.2, we can deal with this issue by checking whether or not the overridden API alters the elements and hiding the elements altered by ad scripts.

False Negative Analysis. From Table 3.2, it can be seen that ShadowBlock has only 2.8% FNs. We further investigate FNs and identify that they are again caused by corner cases not covered by ShadowBlock and that they can be handled by performing taint analysis.

`sohu.com` is an example of false negative. On further investigation, we find that `sohu.com` uses a non-ad script (not on Easylist) to load both ads and non-ad content on the page. Since ShadowBlock only attributes elements created by ad scripts as ads, it misses dual-purpose scripts. It's noteworthy that this should be a rare case, as it is contrary to the common practice of using dedicated third-party scripts to create and load ad elements that most ad publishers exercise today. These publishers normally deploy third-party ad scripts because they have a complex bidding system and prefer dominant control over their ad modules

```
"resource": {
  "type": "text",
  "text": "Guangzhou, Audi TT 82.2K RMB off",
  "md5": "",
  "click": "http://dealer.auto.sohu.com/882054/promotion/article?id=7360579"
      ,
  "imp": [],
  "clkm": [],
  "adcode": "Guangzhou, Audi TT 82.2K RMB off",
  "itemspaceid": "15770"
}
```

Code 3.6: JSON snippet on `sohu.com` for loading ads (translated from Chinese)

We can tackle this issue by implementing the taint analysis approach discussed in Section 3.3.2. Specifically, Code 3.6 shows the snippet of a JSON file on `sohu.com` containing parameters required to create ad elements. In this case, we will need to first mark the JSON object as ad-related, or tainted, and whenever any piece of the data derived from it propagates to any element field (e.g. the URL in JSON's `click` field is used to set an element's `src` attribute), we mark the element as ad. In comparison, extension-based adblockers intercept the network request to load such ad JSON based on its URL in the first place, which effectively prevents the resulting ad HTML element from being created.

Similarly, we observe FNs on `youtube.com` where ShadowBlock is unable to hide all video ads. Our manual analysis shows that `youtube.com` leverages the Media Source Extensions (MSE) API [51] to load video segments through AJAX requests as byte streams. Unlike the standard HTML `video` tag that loads videos as HTTP requests, `youtube.com` loads ad videos in `Blob` objects [50] which are downloaded by JavaScript on the fly. ShadowBlock cannot identify video ads loaded as `Blob` objects, because both ad and non-ad objects are generated by the same non-ad script and assigned to a single HTML `video` element. Unlike our strategy that relies on differentiating ad scripts, extension-based adblockers block the AJAX requests to fetch ad video segments based on their URLs, which achieves the goal of ad removal. As discussed earlier, taint tracking can be used to address this challenge.

Even through we show that tainting is the ultimate solution to the FN cases encountered during our evaluation, we argue that it a comprehensive taint engine poses prohibitively high runtime overhead in the context of web browsing [60, 85]. More im-

portantly, our evaluations have shown the sufficient accuracy of ShadowBlock with the lightweight stack-based execution approximation, as discussed in Section 3.3.2.

3.4.3 Performance

Takeaway: we use two web performance metrics: Page Load Time (PLT) and SpeedIndex. ShadowBlock speeds up page loads by 5.96% in terms of median PTL and 6.37% in terms of median SpeedIndex, on Alexa top-1000 websites.

Page Load Time (PLT). PLT has been the de-facto standard metric for measuring web performance. PLT can be computed by timing the difference between certain browser events using the Navigation Timing API [39]. In order to minimize variations introduced by the initial network setup (e.g., establishing TCP connection with server), we measure the time between `responseStart` [42] and `loadEventStart` [41] events.

SpeedIndex. PLT does not capture a real user's visual perception of webpage rendering process. For example, two pages A and B can have exactly the same PLTs, but page A can have 95% of its visual content rendered by a certain time point while page B has only rendered 30%. From the user perception perspective, page A outperforms page B but they are equally good in terms of PLT. To address this issue, SpeedIndex [44] was proposed to capture the visual progress of *above-the-fold* content, i.e., content in the viewport without scrolling. Unlike PLT, SpeedIndex measures how visually complete a webpage looks at different points during its loading process. Specifically, the page loading process is recorded as a video and each frame is compared to the final frame, for measuring completeness. SpeedIndex is computed using the following formula:

$$SpeedIndex = \int_{t_{begin}}^{t_{end}} 1 - \frac{VisualCompleteness}{100}$$

, where t_{begin} and t_{end} represent the time points of the start (i.e. `responseStart` event in our case) and end (i.e. `loadEventStart` event in our case) of video recording, respectively. $VisualCompleteness$ measures the difference of the color histogram for each frame in the video versus the histogram at frame t_{begin}, and compares it to the baseline (difference of histogram at t_{begin} and t_{end}) to determine how "complete" that video frame is.

We emulate DSL network condition by throttling Chromium [40] to 4 Mbps downlink bandwidth and 5ms RTT latency for all responses to best mitigate measurement volatility across different browser instances. [3] For each site, we first load the webpage to generate its resource cache, then we re-load the webpage 10 times and average the measured PLT and SpeedIndex for each page load. Note that our warm-up strategy ensures most of the static non-ad resources are cached, while ad resources dynamically generated by JavaScript execution are not. This is intended, because we want to minimize the variability introduced by irrelevant factors such as processing non-ad network traffic.

We compute relative ratio of PLT/SpeedIndex across two different configuration pairs (A) ShadowBlock and vanilla Chromium; and (B) ShadowBlock and Adblock Plus (EasyList + Anti-blocking lists), which are denoted as $Conf_A$ and $Conf_B$, respectively.

$$\frac{PLT_{Group_A} - PLT_{Group_B}}{PLT_{Group_B}}$$

[3] We also run another configuration with 750 Kbps downlink bandwidth and 100ms RTT latency to emulate a regular 3G condition [40] and observe similar median trends for both PLT (DSL -5.96% vs 3G +0.30%) and SpeedIndex (DSL -6.37% vs 3G -7.07%) with respect to Adblock Plus.

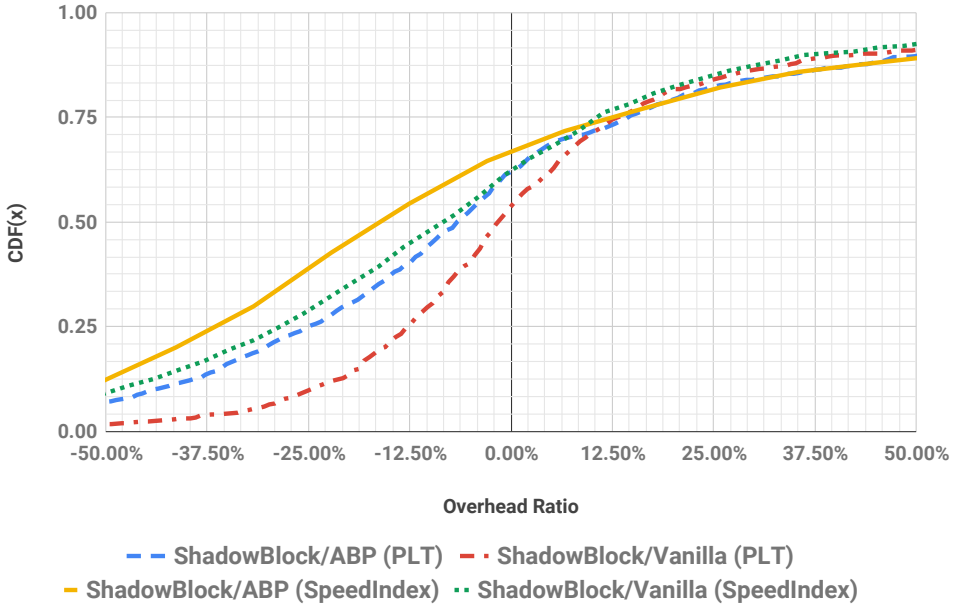


Figure 3.7: CDF for performance metrics

$$\frac{SpeedIndex_{Group_A} - SpeedIndex_{Group_B}}{SpeedIndex_{Group_B}}$$

Overall, both PLT and SpeedIndex show that ShadowBlock speeds up page load time as compared to not only Adblock Plus but also vanilla Chromium. In comparison to Adblock Plus, ShadowBlock speeds up page loads by 5.96% in terms of median PTL and 6.37% in terms of median SpeedIndex. In comparison to vanilla Chromium, ShadowBlock speeds up page loads by 1.03% in terms of median PTL and 5.22% in terms of median SpeedIndex.

The distributions of PLT and SpeedIndex in Figure 3.7, also confirm this trend. We surmise that ShadowBlock's speed up with respect to Adblock Plus is because ShadowBlock's in-browser modifications, versus Adblock Plus's JavaScript-level implementation,

inherently incur less overhead without the necessity of communications between browser core and extension code. For the speed up with respect to vanilla Chromium, it can be explained because ShadowBlock avoids the rendering and painting work for hidden ad elements.

3.5 Discussions and Limitations

Hiding instead of blocking. As discussed in Section 3.3.3, ShadowBlock is designed to visually hide ad elements. Compared to extension-based adblockers that prevent ad resources from loading, ShadowBlock's hiding strategy is inherently limited in two ways. First, ShadowBlock loads the ads and then hides them, thus does not save any network bandwidth. Second, it allows ad resources to load and ad-related scripts to execute, thus exposes users to online tracking. However, we argue that ShadowBlock can complement other tracker blocking approaches that obfuscate and anonymize user-identifiable data [107, 109] which do not require blocking ad requests or stopping script execution.

Completeness of implementation. As discussed in Section 3.3.2, a sufficiently complete yet lightweight taint analysis engine is required to tackle all the FN cases we encounter during ad element identification. However, given the adequate accuracy and practical runtime overhead level, we consider our execution projection technique a sufficient and necessary simplification of taint tracking conceptually.

Adversary from publishers. Modern websites enforce strong isolation among different scripts running in the same document. Violating such policies would normally raise substantial awareness to owners of other scripts or the website itself. This isolation also helps

establish the assumption that ad scripts should never interact with non-ad elements in the same page. However, As soon as the publishers become aware of our approach, they might attack ShadowBlock by pro-actively breaking this assumption to cause collateral damage. For example, an ad script can intentionally modify an attribute of a non-ad element without changing its semantics (e.g. by changing the text encoding). In this case, if we blindly follow the taint tracking and mark the element with taint as ads, we might end up hiding benign elements. To address this challenge, we will need an equivalence test on the semantics of the cases with and without tainting.

Alternatively, an adversary may attempt to detect ShadowBlock. Even though we have closed all normal channels (JavaScript APIs) from leaking information about the presence of ShadowBlock. The adversary may still use more extreme means such as side channels. For instance, if we conduct taint analysis, we slow down the JavaScript execution and therefore they can potentially detect ShadowBlock by timing. However, we argue that this will be extremely challenging if not impossible, because there exist many browsers with different versions of JavaScript engines. There are simply too many possibilities if an adversary observes that the execution is slightly slower (it can even be just a slow machine).

3.6 Conclusions

In this paper, we propose ShadowBlock— a Chromium based stealthy adblocking browser. In addition to blocking ads it hides the traces of adblocking, making it insusceptible to anti-adblocking. Compared to the current state-of-the-art adblocking extensions,

that block resources, ShadowBlock allows resources to load and keeps track of them. Later it hides the loaded resources and fakes their states to JavaScript APIs used by anti-adblockers. Through manual evaluation, we find that ShadowBlock (i) achieves 100% success rate in evading visual anti-adblockers; (ii) replicates 98.2% of ads coverage achieved by adblocking extensions; and (iii) causes minor visual breakage on less than 0.6% of the tested websites. In addition, we evaluate ShadowBlock's performance and find that it loads web pages as fast as adblocking extensions in terms of SpeedIndex and Page Load Time, on average. In summary, ShadowBlock constitutes a substantial advancement for building adblockers invisible to anti-adblockers and presents an important advancement in the rapidly escalating adblocking arms race.

Chapter 4

Eluding ML-based Adblockers With Actionable Adversarial Examples

4.1 Introduction

As adblockers have gained popularity in recent years [28], online advertisers have started fighting back. Specifically, many techniques have emerged to circumvent the current generation of adblockers. Notably, prior work [146] has shown that from among Alexa's top 10K websites, more than 30% have JavaScript code that serve as countermeasures against adblocker use.

Conventionally, adblockers rely on manually curated filter lists, with rules/signatures that are matched against resource request URLs sent from the browser and the

elements rendered in a web page. Unfortunately, manual maintenance of such filter lists does not scale and is error-prone. Moreover, they are fairly easy to subvert (just as antivirus signatures) [78, 103].

Given these limitations, several ML-based adblockers have recently emerged with the goal of improving the effectiveness and accuracy over signature-based adblockers [53, 80, 128] . Such adblockers can be categorized into "perceptual" and "non-perceptual" classes. Perceptual adblockers [53, 128] block ads by recognizing visual cues (e.g. "sponsored" or other marketing keywords) in the web page. It is claimed that these are more robust because some regulators (e.g. FTC) require publishers to disclose the ads and sponsored content. However, recent research has shown that these vision-based adblockers can be easily fooled by adversarial examples; this is a result of recent advances in adversarial machine learning (AML) [131] where ML classifiers can be fooled with human-imperceptible perturbation to the ad images.

In contrast, non-perceptual adblockers detect ads based on non-visual features such as the URL contents and page structure. The state-of-the-art in non-perceptual ML-based adblockers, arguably, is AdGraph [1] [80]. Improving on existing works that simply analyze information in request URLs or HTML/JavaScript code, AdGraph builds a graph representation of a web page load combining all this contextual information, and extracts features from this graph structure to detect ad requests. AdGraph's use of this contextual information supposedly makes it robust because an advertiser needs to make non-trivial changes to a web page, to in turn suitably perturb its graph structure, for circumvention.

[1] [124] extends AdGraph by combining visual and non-visual features. We believe that one can draw similar conclusions as in this work, on attacking its non-visual features.

Furthermore, the use of non-visual features inhibit the applicability of traditional adversarial attack techniques from the unconstrained domain [131].

The main contribution of our work is that we show that it is in fact still possible to craft adversarial ads to circumvent AdGraph. The feasibility of crafting adversarial inputs in domains with stringent constraints (e.g., web pages) remains largely unexplored. The main challenge is to preserve application semantics, which in this case is the visual rendering of the web page. Since web pages are processed by the web browser prior to user exposition (unlike images), rather than the magnitude of the perturbation being the most important criterion, what matters is whether the rendered web page after applying the perturbation presents the same look-and-feel and functionality. Thus, to make so-called *actionable* perturbations, we need to principally rethink the constraints that must be enforced while crafting adversarial samples.

Extending this insight to ML-based adblockers, given the goal of perturbing an ad resource request to bypass the ML classifier: (i) the adversarial example should be actionable in that it must be "mapped back" to the appropriate valid webpage and (ii) the modified request must preserve its original functionality of directing the requester to the remote ad server i.e., this requires the "functional" parts of the page to be equivalent before and after modification. Our goal is to operationalize this insight by crafting such *actionable* perturbations that can circumvent AdGraph. To this end, our challenge is to realize the following properties.

Feature-space actionability: First, any perturbation in the feature-space (including features selected by the ML-based adblocker) must be bounded by domain-specific constraints (e.g., number of child nodes of a DOM node cannot be negative).

Application-space actionability: Second, upon mapping the feature-space perturbations back to the application space (web page), the computed modifications may not be perfectly translated, thererby requiring extra constraints to be satisfied.

As our primary contribution, we present A^4: **A**ctionable **A**d **A**dversarial **A**ttack to craft perturbations that are actionable in both the feature- and the application-space. A^4 needs minimal domain knowledge for providing a set of *seed* features that can be mapped from the feature space back to the input or application space. Specifically, it has the following desirable characteristics.

- **Efficiency:** Inspired by the widely used gradient-based attack, Projected Gradient Descent (PGD) [89], A^4 *iteratively* searches for an adversarial example while accounting for the unique constraints of the web domain. Our evaluations show that A^4 achieves a success rate of about 81% (evading AdGraph's ad detection). In comparison, a naive baseline cannot generate any viable example while two stronger baselines can achieve success rates of only 33% or lower.
- **Actionability:** All perturbed web resources are guaranteed to comply with both the feature and application-space constraints. This compliance makes these examples practical, i.e., they still retain their ad/tracker functionalities.

- **Stealthiness:** A^4 generates perturbations with low detectability (by adblockers) since the perturbations are bounded and concealed with respect to the corresponding web page; further, they are imperceptible to users (except for displaying the ads).

4.2 Background

In this section, we provide a brief background on adblockers and AML, and discuss relevant related work.

Non-perceptual ML-based Adblocking. Because rule-based adblockers are plagued by scale/errors and demonstrable attacks, ML-based adblockers are emerging. Previous works leverage URL strings and JavaScript code as features to represent web resources in ML models [57]. However, these attempts have low accuracy because the representations used are incomplete in capturing the distinguishing characteristics of ad and non-ad resources. This led to AdGraph [80], a recent work on identifying ad resources using a more comprehensive set of features and is considered the state-of-the-art in this field, and also our target in this paper.

AdGraph. By instrumenting the browser core, AdGraph collects a comprehensive set of browser-internal events to stitch together a graph that represents the interactions among the HTML page elements, network requests, and JavaScript executions (e.g., web element A is dynamically created by script B). This representation is then used to train a classifier for identifying advertising and tracking resources. With support from this rich loading context, AdGraph extracts 65 features from a resource load, and classifies the request based on these features. These features can be categorized into two types: structural and content-based. Content-based features include (but not limited to) certain susceptible ad-related keywords

in the URL and the requested resource type (e.g. image, iframe). AdGraph's classifier uses Random Forest as the underlying model, which is non-differentiable. As discussed later in §4.3, this choice hinders traditional AML based attacks as they require the use of gradient to guide the adversarial example generation. Moreover, from the 65 features AdGraph uses, 5 of them are categorical, i.e., will be converted into more than 250 sparse one-hot-encoded features. Such sparsity not only poses new challenges for existing adversarial attacks that expect dense data, but also requires additional constraints to ensure the validity of the one-hot vectors (we discuss how A^4 overcomes these in §4.3).

4.3 A^4: Actionable Ad Adversarial Attack

Adversarial attacks on ML models. Formally, suppose a classifier defined by its prediction function P_{model} and an input x with its malicious label l_{mal}; an attacker needs to find an adversarial transformation T_{adv} such that $P_{model}(T_{adv}(x_{input})) \neq l_{mal}$. The AML community defines different levels of model transparency to describe the knowledge that an attacker possesses with regards to the target classifier:

- With **White-box attacks**, an attacker is assumed to know all the information about the model, including but not limited to the model internals (e.g., the classifier model type, parameters), the training dataset and feature definitions.
- With **Grey-box attacks**, the attackers do not know the internals of the model, but know the training dataset and feature definitions. Further, the attacker can query the target classifier about the label for a specific input.

Gradient-based attacks. One popular attack is based on the Fast Gradient Sign Method (FGSM) [59], which leverages the gradients derived from the target classifier to compute the perturbation that maximizes its loss function with respect to the particular malicious input. Given the loss function of the target model L_{model}, FGSM computes its perturbation η as $\eta = \epsilon \cdot sign(\nabla_x L_{model})$, where ϵ is the norm constraint specified by the attacker. There are also other variants [67] that follow the "loss-maximizing" philosophy used in FGSM. They are generally referred to as gradient-based attacks. Since these attacks all use the gradient information from the target model, they should be considered as white-box attacks.

Gradient-based attacks generate perturbations that are bounded based on different L_p (ϵ above) norms (e.g. L_0, L_2 or L_{inf}). These traditional norms, bounds, or thresholds (referred to as norms in the paper) measure the magnitude of the perturbation, and are thus primarily suitable for visual domain applications (lower norms generally mean less visually-detectable changes) wherein human imperceptibility is the auxiliary characteristic desired in a perturbation. In the web space however, the perturbed page has complex structures and is processed by the browser which parses and renders the page. Thus, the norms can no longer capture what is a "desirable perturbation". In other words, new metrics are needed to effectively capture the properties of functionality preservation and stealthiness of the perturbed web page.

Projected Gradient Descent. Being a single-step attack, FGSM suffers from low success rates, especially when gradients cannot provide sufficiently accurate guidance (usually the case for non-white-box attacks). One can improve the success rate by applying FGSM iteratively; this is known as the Basic Iterative Method, or Projected Gradient Descent [89].

Essentially, PGD performs FGSM multiple times with a smaller step-size, or α. Formally, the search procedure can be expressed as:

$$\begin{aligned} x_0 &= x_{input} \\ x_{n+1} &= Clip_{\epsilon}(x_n + \alpha \cdot sign(\nabla_x L_{model})) \end{aligned} \tag{4.1}$$

where, for a given a given input vector A,

$$Clip(A_i, \epsilon_{min}, \epsilon_{max}) = \begin{cases} \epsilon_{min}, & \text{if } A_i < \epsilon_{min} \\ \epsilon_{max}, & \text{if } A_i > \epsilon_{max} \\ A_i, & otherwise. \end{cases} \tag{4.2}$$

A^4 is inspired by the iterative philosophy used in PGD, and extends its simple clipping mechanism to an extensive feedback loop (§4.3), which seeks to produce actionable perturbations targeting ML-based adblockers.

In this section, we describe how A^4 crafts actionable and stealthy adversarial examples. Being actionable in the both feature and application spaces refers to the following (i) in the feature space, explicit numerical constraints defined based on domain knowledge (to maintain the validity, functionality and stealthiness of the ad request) must be complied with, by its perturbed adversarial feature vector; and (ii) in the application space, the perturbed feature vector must be successfully mappable back to the original web page. Note that actionable perturbations in the feature-space are not *naturally* actionable in the application-space; implicit/unpredictable side-effects that occur when the feature-space perturbations are mapped back to application-space (e.g., a feature space perturbation of adding nodes to a page changes other features such as the average connection degree) must be considered when crafting perturbations.

4.3.1 Threat Model

Before diving into the details of A^4's algorithm, we first define our threat model. As mentioned in §4.2, AdGraph is a full-fledged web browser with custom modifications for blocking ad/tracker resources. Generally, there are three participants when a user visits a website using AdGraph: a user, an ad publisher (1st party) website and an advertiser (3rd party); their relationships are depicted in Figure 4.1. As shown, the objective of A^4 is to

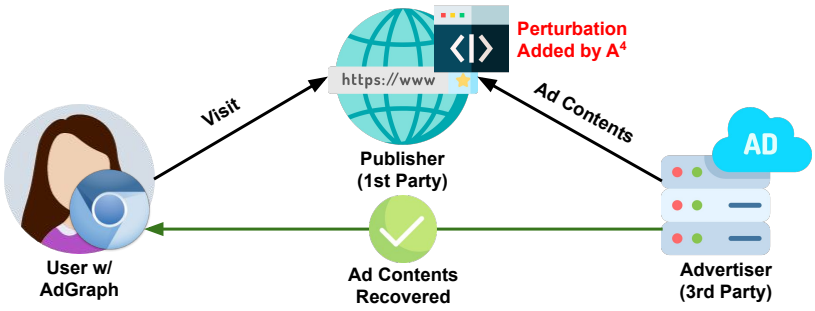


Figure 4.1: Different participants in A^4's threat model

help the website recover its ad revenue lost due to ads getting blocked by AdGraph. A^4 achieves this by adding perturbations to contents generated by the publisher, so that the classifier used by AdGraph is fooled into mis-classifying the ad resource as a non-ad. We assume a grey-box attack setup as elaborated in §4.2, because even though AdGraph has been open-sourced, it is easy for other ML-based adblockers to hide their model internals.

Recall that A^4 is a gradient-based attack which requires the knowledge of model internals, which we do not assume to have. Thus, we make A^4 a transfer-based attack where the attacker is only aware of the training dataset and feature definitions (see §4.2) such that we can reconstruct a surrogate model. Based on the above, our perturbations should meet the following requirements from the practicality/usability perspective:

- **Easily deployable by advertisers:** As a third-party who pays the publisher website for displaying its ad contents, an advertiser is generally reluctant to drastically change the way they operate their services.
- **Easily deployable at the ad publisher:** From the perspective of the publisher, the process of injecting perturbations into the target page should be mostly automatic and convenient. We envision an additional procedure in website deployment via which the web pages will go through to make changes to the page.

4.3.2 Overview

Optimization problem formulation. Formally, consider the optimization problem:

$$\begin{aligned} \underset{x_{adv}}{\text{minimize}} \quad & \text{Dist}(x_{adv} - x_{input}) \\ \text{subject to} \quad & P_{model}(x_{adv}) \neq P_{model}(x_{input}), \\ & x_{adv} \in \mathcal{H}_{feature-space}, \\ & x_{adv} \in \mathcal{H}_{application-space}, \end{aligned} \tag{4.3}$$

where, $Dist()$ measures the cost of adding the generated perturbation, $\mathcal{H}_{feature-space}$ and $\mathcal{H}_{application-space}$ denote the hyperspace that actionable examples can exist in the feature space and application space, respectively. Note that as discussed in §4.2, it is hard for conventional L_p norms to capture the real cost of adding a perturbation. For this, we also modify the L_{inf} norm to take the domain uniqueness into account, as discussed in the next subsection. We also point out that it is impractical to directly apply standard combinatorial optimization techniques (e.g., mixed-integer programming) to exactly solve Equation 4.3 due to computational inefficiency [59], especially in presence of complex target

models (e.g., neural networks). Therefore, as discussed in §4.2, gradient-based solutions are necessary and we build A^4 on top of the state-of-the-art among them (i.e., PDG).

Iterative search. Since the optimization problem defined in Equation 4.3 does not have an analytic solution [59], we instead approximate one iteratively through a search procedure as captured in the pseudo-code in Algorithm 1. The search process not only enforces

Algorithm 1: A^4: **A**ctionable **A**d **A**dversarial **A**ttack

Input : target model M, ad request x_{input}, maximum iterations max_iter, maximum perturbation magnitude ϵ

Output: actionable adversarial example x_{adv}

$success \longleftarrow False$
$curr\_iter \longleftarrow 0$
$x_{curr} \longleftarrow x_{input}$
while $curr\_iter < max\_iter$ **and** $success \neq True$ **do**
  $curr\_iter \longleftarrow curr\_iter + 1$
  $pert_{curr\_iter} \longleftarrow$ `GenerateFeatureSpacePerturbation`(M, x_{curr}, ϵ)
  $x_{curr} \longleftarrow x_{input} + pert_{curr\_iter}$
  $x_{curr} \longleftarrow$ `EnforceFeatureSpaceConstraints`(x_{curr})
  $page_{perturbed} \longleftarrow$ `MapBackToWebPage`$(x_{curr} - x_{input})$
  $x_{curr} \longleftarrow$ `ExtractFeatureValues`$(page_{perturbed})$
  $success \longleftarrow$ `VerifyIfAdversarialOnTargetModel`(x_{curr})
end
return x_{curr}

the feature-space constraints, but also incorporates corrections to address application-space side-effects that occur when mapping feature-space perturbations back to the web application domain. The key guiding principle is to take small steps (in *each iteration*) and corrective actions so that we are always on the right path. (details in the next subsection).

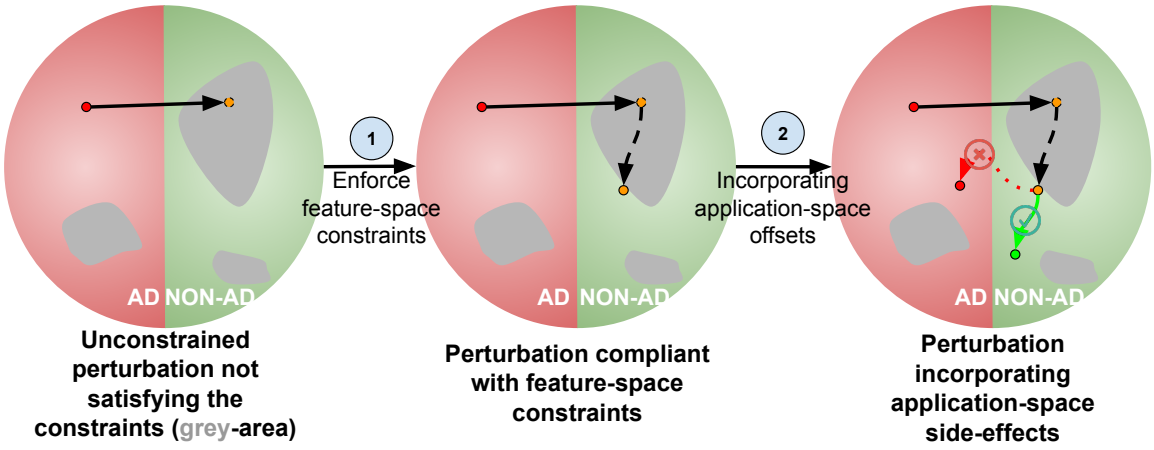


Figure 4.2: Perturbation trajectory in hyperspace for a search iteration; ● refers to positions in non-adversarial hyperspace; ● refers to positions in adversarial hyperspace that do not satisfy constraints; ● refers to positions in adversarial hyperspace that satisfy constraints.

To better illustrate the framework, we show for each iteration, how the generated original perturbation is moved in hyperspace to ensure its actionability in Figure 4.2. The intuition is that errors may accumulate across multiple iterations and can mislead us if we do not correct them at every step (and we take smaller steps for the same reason). Inspired by the iterative philosophy that underpins PGD, A^4 also divides the overall optimization problem into multiple iterations. We would also like to point out that A^4 is not a simple "trial-and-error" framework, because at each iteration of the algorithm, gradients from the target model provide feedback that guides Algorithm 1 what direction it should explore next.

Transfer-based attack. To craft a successful grey-box attack, we need to use the dataset for training AdGraph to train a local *surrogate* model that is differentiable, and then use this model to estimate the gradients and craft adversarial examples accordingly. These type of attacks are considered "transfer-based" because the successful adversarial examples

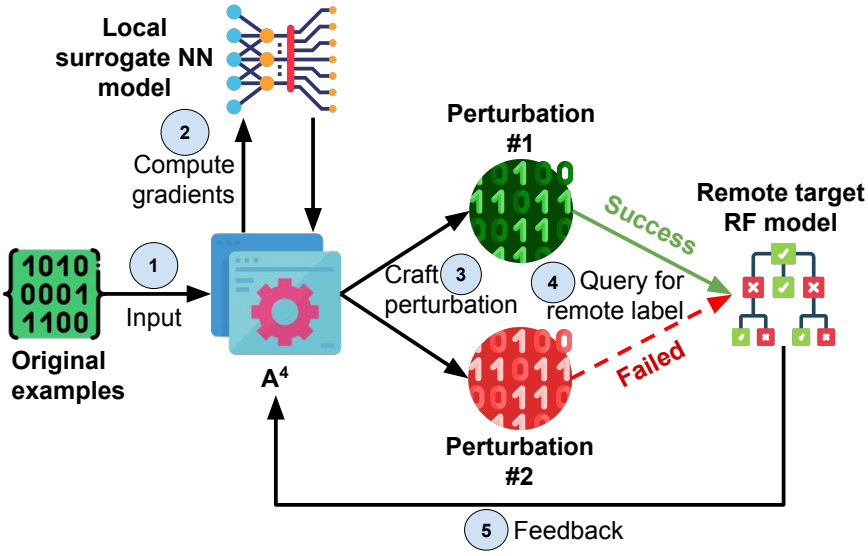


Figure 4.3: Transfer-based attack paradigm; steps ③, ④, and ⑤ are executed in a loop.

crafted locally need to be adversarial on a remote target model that is different and possibly unknown. We depict A^4's transfer-based attack generation in Figure 4.3. Prior research [110] has shown that this so-called inter-model transferability exists in almost all modern ML models (including non-differentiable ones such as Random Forest).

Perturbable feature selection. Before delving into the constraints, we need to first manually identify what features to perturb. These features must be perturbable, i.e., the attacker must know how to map the perturbations from the feature space back to concrete changes in the application space, i.e., the web page. As mentioned in §4.2, AdGraph has two categories of features: structural and content-based (URL-related). After systematically analyzing all the 65 features used in AdGraph, we identify 19 seed features from both (8 URL and 11 structural) categories that the publisher of the ad request can control and perturb in practice. Table A.5 in the appendix shows their semantics, data types (i.e., integer, binary or float) and categories (i.e., structural and URL).

4.3.3 Feature-Space Constraint Enforcement

In this subsection, we describe how A^4 imbibes explicit numerical constraints in the feature space. Since the constraints defined in this space could be for three different purposes — validity, functionality and stealthiness, we need to enforce them differently. This step corresponds to `EnforceFeatureSpaceConstraints()` in Algorithm 1.

Validity constraints. These constraints keep the perturbed features numerically valid i.e., they guarantee that they fall within meaningful *domains of definitions*. For instance, features #1 and #2 are counts of nodes and characters, and cannot be negative or non-integers; binary features #4 to #5 should always take values of a 0 or a 1. These constraints are enforced by projecting any perturbed value falling outside the meaningful domain of definition back to the domain. Concretely, we define three projection operations for binary and numerical types of perturbable features in `EnforceFeatureSpaceConstraints()` in Algorithm 1:

$$x_{proj} = \begin{cases} \max(\min(1, x_{pert}), 0), \text{ if } x_{pert} \in S_{numerical} \\ 0, \text{ if } \|x_{pert} - 0\| \leq \|x_{pert} - 1\| \\ \quad \text{and } x_{pert} \in S_{binary} \\ 1, \text{ if } \|x_{pert} - 0\| > \|x_{pert} - 1\| \\ \quad \text{and } x_{pert} \in S_{binary} \end{cases} \tag{4.4}$$

Through these operations, the perturbed features of our choice are guaranteed to be valid in the feature space.

Functionality constraints. Besides validity, A^4 also needs to ensure that the generated feature-space perturbations won't break any functionality of the original ad request. Hence, we also enforce functionality constraints onto the perturbed features. To do so, we follow two principles which we refer to as *non-decreasing* and *semantic equivalence*. For counter-like features like #1 and #2 (Type "I"/"F" in Table A.5), we limit their perturbed values to be greater than or equal to the original values; otherwise, we project the modified value back to its original value. We refer to this as the "non-decreasing principle." This projection reflects our assumption that adding information to the web page should not break any existing functionality, but removing existing items might harm the semantics in an unpredictable fashion. In Figure A.1 (in the appendix), we show one example DOM snippet before and after introducing the structural perturbations. Note that the inserted DOM nodes can be disguised as pertaining to regular content with random properties/text, to avoid being detected by simple rule-based scans. We describe possible obfuscation strategies to disguise inserted DOM nodes in more details in §A.1 (in the appendix).

For URL features (Category "U" in Table A.5), we need to ensure that after perturbation, the original functionalities/semantics of the request are preserved. Specifically, features that detect predefined keywords/characters from the URL string (e.g., feature #5 and #6), can be simply replaced with unmarked sub-strings. Since AdGraph hardcodes these keywords/characters, our URL manipulations can effectively bypass its detection over all URL-related features. For feature #3, we choose to append random characters to increase its value, and place the appended string as an unused query, which best avoids disrupting other functional parts of the URL.

We are aware that the above URL manipulations introduce changes to the request received by 3rd-party advertisers, and therefore require cooperation from them. As discussed in §4.3.1, A^4 is expected to enable easy deployment for both 1st-party publisher websites and 3rd-party advertisers. Towards this, we believe the simplest solution is a reverse proxy employed at advertiser servers, which translates perturbed URLs to their unmodified version based on pre-negotiated protocols between publishers and advertisers. In order to do so, we preserve the basic components (i.e., scheme and host name) in the URL and only perturb the remaining parts (i.e., path and query string) as guided by A^4. This way, the advertiser server is guaranteed to receive the requests, and can then easily translate these perturbed URLs internally. We point out that similar setups have been already practiced by publishers/advertisers and adblocking circumvention services to successfully evade rule-based adblockers [64, 90]. We are also aware that despite available deployment strategies, some publishers/advertisers might still be reluctant to altering their system configurations; we anticipate though, given the gigantic revenue loss to publishers due to adblocking ($16 billion to $78 billion in the year of 2020, as projected in [81]), a considerable number of publishers/advertisers would be willing to do so in exchange of ad revenue recovery.

Stealthiness constraints. Besides validity and functionality, the generated perturbations should also achieve a high level of stealthiness. Specifically, the perturbations that A^4 applies on features will have to be limited by a threshold. Conventionally, the perturbation size is measured via the use of L_p norms. However, these norms are unsuitable for AdGraph's feature set. First, with many binary/categorical features, use of L_p norms blindly treats all

features as having the same scale, which is not the case in reality. For example, changing a binary feature from 0 to 1 means that the status it represents has flipped. This is fundamentally different from an integer feature changing by the same amount; for the latter, it could indicate that its real value has changed from a minimum to a maximum value (due to data normalization happened in dataset pre-processing). Thus, if we set a threshold L_{inf} to 0.3, binary features can never be flipped (as the flipping threshold is 0.5), whereas integer values can still change even if a normalization is applied. To account for such differences, we propose a customized L_{inf} norm which is defined as follows, we propose a customized L_{inf} norm which focuses on numerical features as defined in Equation 4.5. The examples generated by A^4 are bounded by this norm.

$$L_{custom\_inf}(pert) = \max(|pert_i| : i = 1, 2, \ldots, n \text{ if } pert_i \in S_{numerical}) \quad (4.5)$$

Besides customizing the norm, we also slightly modify the operation for clipping a perturbation within the norm. Specifically, conventional clipping functions (e.g., the one used by PGD) regard the global range of a particular feature across the whole dataset as the base of the clipping threshold, for conventional features. For web pages, such clippings can easily lead to overly large perturbations as the ranges of many numerical features can vary drastically from website to website. Therefore, we change the clipping from relying on a global range to a local per-webpage range, as formally defined in Equation 4.6.

$$Clip'_{global\_local\_mixin}(pert, \epsilon_g, \epsilon_l) = \begin{cases} Clip(pert, 0, \epsilon_g \cdot r_i), & \text{if } \epsilon_g \cdot r_i < \epsilon_l \cdot pert \\ Clip(pert, 0, \epsilon_l \cdot pert), & otherwise \end{cases} \quad (4.6)$$

where ϵ_g is the global threshold, ϵ_l is the local threshold, r_i is the global range of x_i with respect to this particular feature in the training dataset, given by $x_i^{max} - x_i^{min}$, and

$Clip(x_i, \epsilon_{min}, \epsilon_{max})$ is the standard clipping operation defined in Equation 4.2. As shown in §4.5, our customized norm along with the localized clipping operation, helps limit the effective size of generated perturbations, and thus improves the stealthiness significantly compared to the traditional setup of L_{inf} norms and global clipping.

4.3.4 Application-Space Side-Effect Incorporation

Now that we have generated feature-space adversarial perturbations that comply with manually-defined domain constraints, we need to map them back to concrete changes in the web page representations. As discussed previously, ideally these perturbed feature values should all be reflected accurately in the page. This can be overt if we can re-extract the feature vector from the perturbed web page and verify that it matches the expected one. This step corresponds to the combination of `MapBackToWebPage()`, `ExtractFeatureValues()` and `VerifyIfAdversarialOnTargetModel` in Algorithm 1.

However, introducing changes (e.g., total number of nodes) to the web page can cause unpredictable offsets to values of other features not included in the feature-space perturbations. Specifically, there are several inter-dependent features considered by AdGraph such as feature #19. As we add perturbations nodes to the page to perturb the feature counting the total number of nodes in the graph, feature #19 might also *nondeterministically* change as the maximum per-node connection degree is raised, which might end up turning an adversarial perturbation into non-adversarial. More critically, such feature value offsets/drifts are impossible to be predicted, and therefore cannot be pre-computed in closed-loop formulas, which motivates our design of executing the feedback loop.

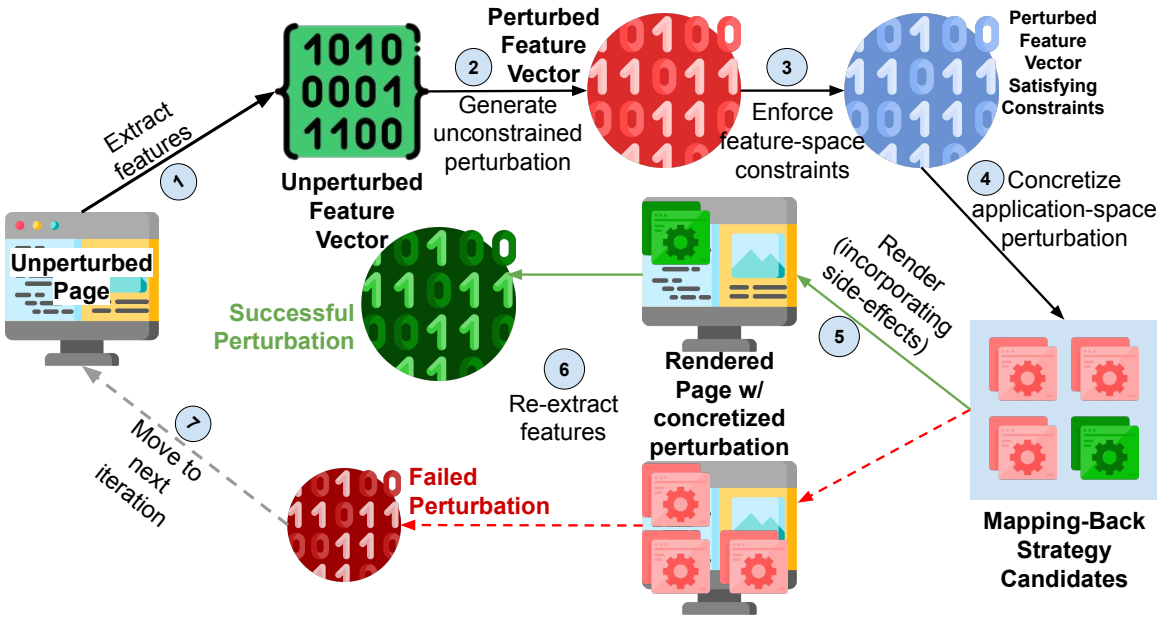


Figure 4.4: Proposed feedback loop in A^4's each search iteration

Feedback loop. To incorporate such unpredictable side-effects, we *passively observe* how changes in one feature causes changes in others. Specifically, we first map the controllable feature-space perturbations back to the web pages by rendering the page and then re-extract all the features to *capture* the side-effects. We verify if the final perturbation (with side-effects) can still evade detection. If so, we are done; else, we continue the iterative search procedure to find another candidate perturbation (we enlarge the current step size by a step size to generate a new gradient). Effectively, we have created an automated feedback loop as illustrated in Figure 4.4.

Diversified mapping-back strategies. For some features, there are multiple ways to concretize the feature-space perturbations as changes to web pages (step 4 in Figure 4.4). For instance, there are multiple ways to increase the total number of nodes in a page (feature #1). We can choose to place these nodes either as the children of a single existing node

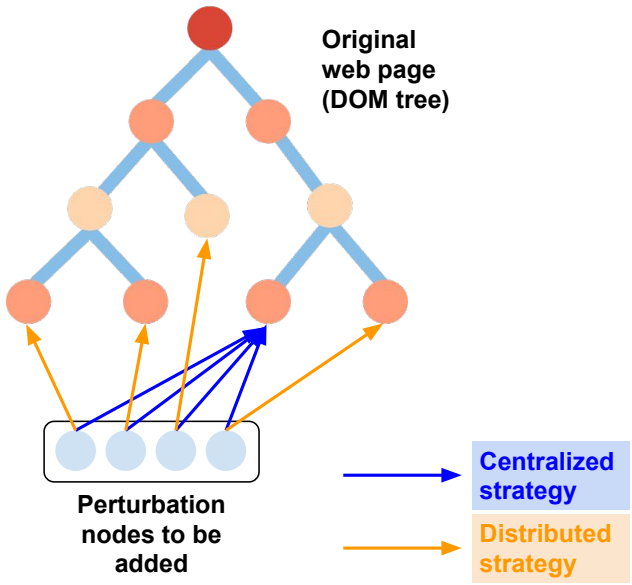


Figure 4.5: Different mapping-back strategies on adding perturbation nodes

(*centralized* strategy), or as the children of multiple existing nodes (*distributed* strategy), as shown in Figure 4.5. These different mapping-back strategies introduce different side-effects to the feature values, and can hence affect the effectiveness of the final adversarial example (as depicted by the red and green points in Figure 4.2). One example is, again, that for the feature #19 (average degree of connectivity), the centralized strategy is likely to lower the feature value significantly after the map-back as the added nodes cause crowding and thus, raise the current maximum number of connections per node in the graph; this is the denominator in the formula that computes the average degree of connectivity for the page. In contrast, the distributed strategy tends to have negligible side-effects with respect to this feature. In order to maximize the chance of finding a successful adversarial example, we apply all feasible mapping-back strategies in the feedback loop, and then verify their results. These two diversified strategies help A^4 discover as many green point cases as possible (Figure 4.2).

4.4 Implementation

We use the library `Foolbox` [117, 118] to compute the adversarial perturbations numerically. Per our design in §4.3, we implement A^4 as described in Algorithm 1, by augmenting the standard PGD attack in `Foolbox`. We iteratively enforce the constraints in both feature and application spaces with the feedback loop, as demonstrated in Figure 4.4. For step ④, ⑤ and ⑥ in Figure 4.4, we implement an HTML manipulator for reflecting the computed numerical adversarial examples in the webpage representation (HTML), based on the commonly used Python library `BeautifulSoup` [120]. As will be discussed in §4.5, the current implementation of A^4 handles static ad requests only, and therefore acts as a proxy that can be deployed at publisher websites hosting ads. Specifically, A^4 parses the HTML containing ads, computes and inserts corresponding adversarial perturbations into it, and delivers the perturbed HTML to users [2].

4.4.1 Model Training

We need to reproduce the classifier used in [80] following its revealed hyper-parameters, on the newly collected dataset, since [80] did not release their trained models. We use the popular open-source machine learning library `scikit-learn` to train a Random Forest (RF) model based on the crawled training dataset. This is then used as the target model that A^4 queries, with each perturbed example, to verify the attack result. We show the model's hyper-parameters and classification accuracy metrics over the partitioned testing set in Table A.1 in the appendix. As shown, the accuracy metrics with our reproduced

[2] We open source the implementation of A^4 and its dataset at `https://github.com/seclab-ucr/A4`, for reproducibility and future research extensions.

RF model are close enough to the ones reported in [80], which validates our replication effort.

We then need a local surrogate model that is differentiable (recall §4.3), to drive our gradient-based attack. To this end, we use the Python-based deep learning library `Keras` [63] to train a 3-layer Neural Network (NN) as the surrogate. NN is considered to have the best model capacity [110] for imitating the decision boundaries of other models. The hyper-parameters and accuracy metrics of this NN are in Table A.2 in the appendix. Note that in order to best mimic the remote decision boundary, we use the dataset that trains the target RF classifier and labels given by the *target model*, instead of ground-truth labels, to train the local NN. Hence, the accuracy in Table A.2 represents the agreement rate between two models.

4.4.2 Active Learning

Given that A^4 trains a local surrogate model to imitate the decision boundary of the target classifier in [80] and provide necessary gradients. Even though in §4.5 we show the surrogate agrees with the classification outcome with the target model on testing set sufficiently well (90% of the time), there can still be cases where two models disagree with each other, which indicate local divergences of their decision boundaries. To further align the decision boundaries and boost the effectiveness of A^4, we apply active learning [111]. Specifically, upon encountering any ad request that is classified differently by the target model and the surrogate, we train the local surrogate with the request and the label from the target model. If one iteration of training (i.e., one pass of back-propagation) does not address the divergence, we repeat 10 times at maximum.

4.4.3 Hyper-parameters

Table 4.1 shows the hyper-parameters used in our implementation. Note that the parameter enforcement interval here refers to the number of steps we take in the gradient-based search before we enforce the constraints and consider these steps together as one iteration of the feedback loop. We operate at units of intervals instead of steps, because multiple steps are often required for binary features to be pushed across the flipping threshold of 0.5, as discussed in §4.3.

Note that to avoid confusion, we use the term "iteration" to refer to an enforcement interval. These parameters are empirically chosen, and we have varied and tuned them to pick the best parameter set that are shown to yield best performances. We also would like to point out that out of the different combinations of parameters, the improvement achieved by A^4 (as will be shown in §4.5) over baseline attacks are generally consistent.

Iterations (*max_iteration* in Algorithm 1)	20
Step-size	0.07
Maximum *global* perturbation threshold (ϵ_g in Equation 4.2)	0.3
Maximum *local* perturbation threshold (ϵ_l in Equation 4.2)	0.5
Enforcement interval	15

Table 4.1: Hyper-parameters used for attacks

4.5 Evaluation

4.5.1 Setup

Dataset. Since A^4 primarily targets the current version of AdGraph [80], we first reproduce its pipeline using its open source implementation. Given that the web crawl conducted in [80] was in early 2018, and is hence outdated, we carried out a new crawl on December 3, 2020 to collect the graph representation of the landing pages of Alexa's top 10k websites. Then, we processed these graphs and extracted 65 features to form the dataset ready for ML tasks. Table A.3 lists some basic statistics from the crawled dataset.

From the 503,526 request records, we randomly pick 50,000 as the test set (i.e., the remaining 453,526 records are used as a training set). These are used to test the accuracy of trained classifiers compared to the original AdGraph model. Other than these 50,000 samples, we additionally randomly pick 2,000 unique ad requests as the target ad resources to be perturbed for evaluating the effectiveness of A^4 in flipping classifications for ad requests.

Attack variants. As discussed in §4.3, A^4 picks 19 features to perturb in total. To understand what roles these features play in achieving evasion, we conduct an ablational analysis by grouping features into three subsets: (1) "All" includes all 19 features; (2) "Only URL" includes 8 features in category "U" in Table A.5; (3) "Only Structural" includes 11 features in category "S" in Table A.5. We refer to these three variants as `All`, `Only URL` and `Only Structural` hereon.

Baseline attacks. For comparison, we consider two baseline attacks we describe below (the descriptions also illustrate the necessity of our proposed solution).

- **Baseline 1:** In this attack, we apply the standard PGD without enforcing any feature-space constraints, other than the basic perturbation size limit (ϵ in Equations 4.1 and 4.2) and a basic domain of definition (i.e., $0.0 < x_i < 1.0$).
- **Baseline 2:** In this attack, we generate uniformly random perturbations (i.e., using $pert_i \in \mathcal{U}(-\epsilon_g, \epsilon_g)$ instead of ② in Figure 4.4) without relying on any guidance such as gradients from the surrogate model. We enforce constraints and execute the remaining feedback loop once on the random perturbation. The purpose of this setup is to assess the effectiveness of one-step correction without the guidance of gradients. We anticipate subpar success rates as it is highly challenging to relocate random perturbations to constrained adversarial space without iterative corrections.
- **Baseline 3:** In this attack, we apply A^4 for one complete iteration (i.e., enforce constraints and execute the full feedback loop once), instead of performing iterative repetitions. The purpose of this setup is to comprehensively validate the benefits/advantages of the proposed iterative search framework, in presence of estimated gradients. Again, without multi-iteration corrections, we anticipate difficulty in achieving high success rates, mainly because even with some guidance of initial gradients, the single-step approach can lead to application-space offsets that disrupt the adversarial nature of the example and cannot be corrected in one iteration.

4.5.2 Experimental Results

Ad coverage. Currently A^4 handles only static requests that are embedded into the HTML file of each webpage, which correspond to about 60% of all ad requests detected by AdGraph (it operates at top of iframes and thus cannot detect/handle ad resources inside these iframes). This is because of the implementation of current prototype. Due to its prohibitively high overhead in computing adversarial examples (882 seconds on average for generating one example, as will be shown in overhead evaluations), it is challenging to envision a runtime solution (e.g., via injected JavaScript) that dynamically intercepts ad requests and computes perturbations for them on the fly. Instead, perturbations need to be computed offline by publisher websites and then statically inserted into webpages at the moment. We will discuss possible future improvements for covering more ad requests in §4.6. Note that all success rates in the following subsections are calculated with respect to the covered ad requests (i.e., ∼60% of all ad requests).

Success rate. We summarize the success rates achieved by the three attacks in terms of finding actionable adversarial examples from the 2,000 ad requests in the testing set, in Table 4.2. We see that A^4 achieves the highest success rate in generating mis-classified examples while guaranteeing their actionability in all attack variants. In `All` and `Only URL` attack variants, it is over twice as successful (145.5% improvement) compared to Baseline 2 [3] and 3. Even using only parts of available features (i.e., `Only URL` and `Only Structural`), A^4 can achieve over or close to 60% success rates. From an ablation perspective, these results also

[3]Note that it is not a surprising finding that random perturbations can evade ML classifiers in cases. Prior research [70, 73] shows that widely used and state-of-the-art image classification models (e.g., `ResNet` and `InceptionV3`) are vulnerable to perturbations of additive random noises, and observe accuracy degradation of over 30% if they are naturally trained with clean samples only.

Feature Set/ Success Rate	All	Only URL	Only Structural
Baseline 1	0.00%	0.00%	0.00%
Baseline 2	32.67%	18.98%	32.65%
Baseline 3	33.02%	24.76%	32.42%
A^4	**81.18%**	**65.74%**	**58.62%**
Partial Dataset	73.06%	55.91%	53.78%

Table 4.2: Breakdown of attack results; **bold** numbers indicate the best success rate in that feature set

suggest that A^4 relies more on URL perturbations than structural ones in evading AdGraph. Supporting this observation, we provide the information gain (i.e., feature importance [115]) ranking of top-5 perturbed features used in AdGraph's random forest classifier in Table A.4 in the appendix, 4 of which are URL-related.

The large margin of improvement shows the power of the iterative search adopted by A^4. In comparison, Baseline 1 fails to produce any valid perturbation because if none of the constraints is enforced, features of data types like binary or categorical, can be changed into meaningless values (e.g. in a one-hot-encoded vector, more than one feature becomes 1). This makes it impossible for these perturbed examples to be rendered in the browser at all. Therefore, we do not consider Baseline 1 in the further evaluations, and refer to Baseline 3 as the only baseline hereon.

Partial training dataset. As explained in §4.2, we assume a Grey-box threat model where the attacker knows the entire training dataset. Even though we believe this is practical as the training dataset [80] uses is from crawling open webpages (i.e., can be easily

Feature Set/Strategy	**Centralized**	**Distributed**	**Both**
All	13.94%	19.28%	66.78%
Only Structural	20.45%	32.24%	47.30%

Table 4.3: Mapping-back strategy significance analysis

replicated by crawling the top websites again), one might wonder what if attackers perform their crawls at different time points and thus only have limited access to the original dataset for building their surrogate NN models. We therefore evaluate A^4 with its NN trained on only *one-third* (i.e., 151,175 vs 453,526) of the dataset that is used for training the target RF model. We report the success rates for this experiment in all three attack variants in Table 4.2, and find that they only degrade insignificantly (<10 percentage points), which suggests that even with partial knowledge of the training dataset, A^4 remains sufficiently effective in crafting adversarial samples in practice.

Mapping-back strategy significance. Table 4.3 reports the significance of different mapping-back strategies that A^4 tried in its feedback loop. Note we only evaluate strategies for `All` and `Only Structural` variants, as `Only URL` does not involve structural manipulations. As can be seen, in close to 50% and 70% of the successful perturbations, both centralized and distributed mapping-back strategies are attempted (to a similar degree) to craft actionable adversarial examples. However, for a significant portion of the test cases in both variants, only one strategy succeeds. These cases show the advantage of applying multiple strategies to cope with the unpredictable application-space side-effects into the

iterative search procedure used in A^4. Essentially, more valid green points (as depicted in Figure 4.2) can be discovered with additional strategies.

Attack convergence. In Figure 4.6, we show a histogram of the number of iterations needed to reach convergence with all the successful adversarial perturbations generated by A^4, We see that most (>80%) of cases converge within 5 iterations; this shows that the iterative feedback loop in A^4 is extremely efficient in generating the adversarial examples.

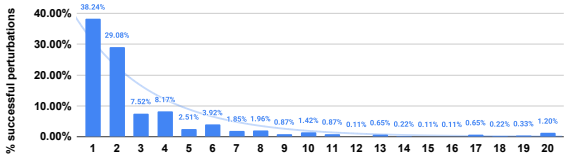


Figure 4.6: Attack convergence analysis

Node additions. Table 4.4 lists the average size for successful perturbations in terms of their number of added DOM nodes. Recall that the perturbations A^4 generates, for manipulating structural features, primarily rely on adding/inserting DOM nodes into webpages. We statistically analyze the node additions, and find that (1) they obey the local perturbation threshold (i.e., 0.5) as defined in Table 4.1, confirming its effectiveness; and (2) over 70% of the time A^4 only needs to inject <50% additional DOM nodes (as depicted in Figure 4.7c) to successfully evade the adblocker, indicating its economy/efficiency in leveraging DOM node addition as the primary underlying manipulation mechanism. Note that the perturbation size here does not translate to performance overhead A^4 incurs, as perturbed webpages need to be loaded and rendered.

Performance overhead. We evaluate the runtime performance overhead that A^4 imposes on web browsing, in terms of page size increases (due to perturbations) and user-perceived

All	Only URL	Only Structural
+49.00% +5.41%	- -	+55.21% +13.54%

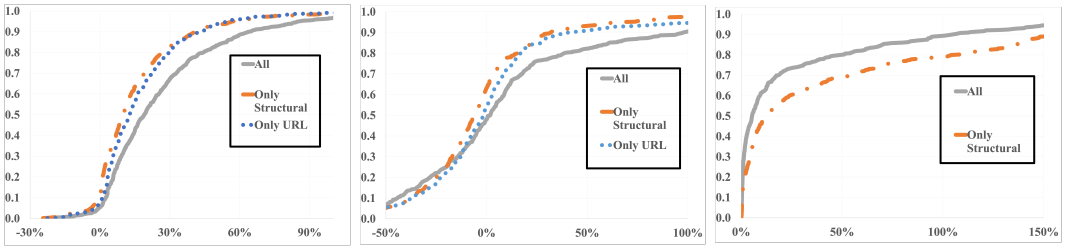
Table 4.4: Perturbation size in terms of # of additional/inserted DOM nodes (first row in each cell is the mean over all tested webpages; second row is the median); "-" means not applicable as `Only URL` does not involve node additions.

Feature Set/Overhead Metric	All (Baseline)	All (A^4)	Only URL (Baseline)	Only URL (A^4)	Only Structural (Baseline)	Only Structural (A^4)
File Size	+22.11% +13.61%	+27.64% +18.59%	**+12.19%** +10.09%	+17.97% +12.78%	+12.51% **+7.65%**	+14.32% +8.85%
Page Load Time	-16.61% -5.26%	+19.20% +1.83%	**-5.37%** -4.09%	+10.85% -1.71%	+18.16% **-6.12%**	+3.16% -6.01%

Table 4.5: Breakdown of overhead analysis results (first row in each cell is the mean over all tested webpages; second row is the median); **bold** numbers indicate the lowest overhead in that metric.

Page Load Time (PLT) (following the standard method as used in [80]). Table 4.5 shows the breakdown of the overhead results across different attack setups. Both metrics suggest that A^4 only incurs acceptable/insignificant performance overheads (i.e., <30% in all attack variants) in web browsing. Figure 4.7c further shows the cumulative distribution function of PLT measurements for different attack variants. It suggests that, for close to or more than 90% of successful perturbations, A^4 incurs <50% overhead in PLT.

We also evaluate the offline overhead for generating/computing the perturbations. For `All` attack variant that achieves the highest success rate among all variants, A^4 spends



(a) CDF for file size overhead (b) CDF for page load time overhead (c) CDF for # of additional nodes

Figure 4.7: Cumulative Distribution Function (CDF) for different measurements

882 seconds on average for computing a successful adversarial perturbation (on a single core Xeon 6248 server). Note this computation is only needed once for a given ad request, and is therefore considered acceptable for recovering its associated ad revenue.

Breakage analysis. As discussed in §4.3, A^4 minimizes the risks of disrupting legitimate webpage functionalities by selecting perturbation primitives that preserve functionality. To validate this is indeed the case, we conduct manual inspection for breakages caused by the perturbations. Specifically, we randomly sample 100 successfully perturbed (i.e., the classification of target ad request has been flipped to a non-ad due to the perturbations) webpages from the 2,000 tested ad requests in `All` attack variant. We then have four student evaluators browse the original and perturbed versions of the same webpages, and record their judgements of how A^4 impacts the site's functionality.

The following are the classification criteria to define different levels of breakages and their respective results, in accordance to the breakage analysis methodology in [80].

4

[4]Note in this evaluation, we only account for *differential breakages*, which are breakages that exhibit on the perturbed webpage only. There are common breakages on both versions in some cases due to the time lag between web crawling and manual inspection, which are irrelevant to A^4.

- **No breakage (92%)**: There is no perceptual difference between the perturbed and original versions of the webpage;
- **Minor breakage (8.0%)**: The browsing experience is impacted, but the main objective of the visit can still be accomplished;
- **Major breakage (0.0%)**: The main objective of the visit is severely impacted and cannot be completed;
- **Crash (0.0%)**: The webpage cannot be opened/rendered.

Above shows that A^4 achieves considerably low breakage. Note that A^4 does not cause any major breakage or crash in the evaluation. Upon a closer analysis, we find these minor breakages are mostly cosmetic. To better demonstrate the breakages that A^4 possibly introduces, Figure A.2 in the appendix shows an example with minor breakages after adding the perturbations, from the 100 successfully perturbed webpages used in the breakage analysis.

4.6 Discussions and Limitations

Deployability. As discussed in §4.5, A^4's current implementation can only cloak ad requests that are statically embedded into the main HTML DOM; requests that are dynamically generated by JavaScript are unprotected. Even though A^4 already covers more than 60% of all ad requests detected by AdGraph, we plan to extend the coverage further to dynamic ad requests in the future, following three routes. First, publishers can render webpages at server-end so that A^4 can access dynamically-generated ad requests and compute perturbations for them. Note that in this case we believe the amount of page dynamics

(i.e., page differences caused by distinct JavaScript execution outcomes across page loads) is insignificant and should not reverse the adversarial-ness of pre-computed perturbations. This is because as discussed in §4.5, AdGraph does not rely on contents inside iframes for making adblocking decisions and a significant portion of 3rd-party web contents causing page dynamics are loaded inside iframes [132]. Second, one can implement and mount runtime solutions (e.g., via injected JavaScript) to perturb URL features only. This would result in significantly less time in computing perturbations as compared to including expensive structural features for realizing on-the-fly perturbations, and still successfully evade AdGraph ∼65% of times (as shown in §4.5). Third, if advertisers would fully collude with publishers, they can act as adversaries and compute perturbations for themselves. Again, we acknowledge such collusion requires additional efforts to implement and might hence appear unattractive to some publishers/advertisers, but still anticipate voluntary adoptions by those who prefer ad revenue over deployment cost (as discussed in §4.3.3).

Another challenge A^4 faces is its current per-request perturbation generation. One webpage can contain multiple ad requests whose features are co-dependent, mandating dependencies across their adversarial perturbations. This will require a joint optimization in generating per-page perturbations over multiple requests and is thus is left as a future research direction.

Generalizability. Although A^4 primarily targets an ML-based adblocker at this point, we argue that its iterative search procedure and feedback loop are general enough to be applied to other AML scenarios in the web domain, or even other domains. At a high level, any ML task that requires (1) the generated examples be actionable in both the feature and

application spaces, (2) is constrained in the feature-space due to the functionality requirements as discussed in §4.3, and (3) has associated side-effect offsets upon mapping from the feature-space to application-space (as shown in Figure 4.2), can be viewed as compatible with A^4's methodology. Examples include classifiers that operate in non-traditional representations such as programs, network traffic [148] etc.

As noted in §4.3, currently A^4 is designed to perturb at most 19 features from the possible 65. Although this conservative limit makes our attack stealthier in practice, we plan to explore perturbations on more features in the future and analyze their effectiveness. Further, our current A^4 implementation only has two mapping-back strategies (centralized and distributed) as discussed in §4.3. While even with these two strategies, we can already showcase the power of our proposed feedback loop, we will explore additional strategies in the future to expand our search space. Also as mentioned in §4.3, A^4 leverages HTML DOM node addition as its underlying manipulation mechanism. We believe this is a general solution to other web-based ML classifiers and even graph models.

Arms race between adblockers and advertisers/publishers. Our ability to subvert AdGraph suggests that ML-based adblockers should be more careful in designing their feature sets. During our analysis of AdGraph's features, for example, we find that there are several global features (e.g. #1/#2 in Table A.5) that encode the overall size of the constructed graph. Since these features do not describe anything local with respect to the request node being classified, they are of less importance, but leave perturbation space for adversarial attacks. There is also rising trend of research on improving adversarial robust-

ness of models through blending adversarial samples into training datasets (i.e., adversarial training) [70], and detecting them [94] from ML perspectives to consider.

From the adversarial perturbation generation perspective, our investigations suggest that it is crucial to provide the necessary setups (e.g. proxy/rotating servers) to facilitate perturbations on URL-related features. As shown in §4.5, URL features are crucial in helping A^4 achieve high success rates. Proxy deployments are adopted by some websites that counter rule based adblockers [146].

4.7 Related Work

Constrained adversarial examples. Existing research on AML has been dominantly focused on domains where the adversary can exploit the unconstrained nature of the input representation such as images – each feature (e.g., pixel) is fully under control of the adversary. However, as ML models are increasingly being deployed in many constrained domains (e.g., network intrusion and malware detection), it is important to explore their robustness in such systems against adversarial attacks. In addition to the attack that is designed for mainly webpages in this paper, we summarize some early explorations in other constrained domains as follows.

[112] proposes adversarial samples against Android malware detectors. It codifies the space of permissible adversarial examples, and then transplants code snippets from benign Android software into given Android malware. Additionally, [112] creates opaque predicates in perturbed malware so that the transplanted gadgets won't be actually executed, to preserve malware semantics. Following similar routes, [144] and [141] aim to craft adversar-

ial examples for general ML-based code models. They apply different semantic-preserving code transformations with the guidance of model gradients, and evade code models for various purposes. Beyond source code, [87] presents an attack against binary-based malware detectors, by injecting unused bytes.

Adblocking and anti-adblocking. As briefly mentioned in §4.1, there has been a fierce arms race between adblockers and their countermeasures. Besides concealing ad signatures to combat rule-based adblockers, ad publishers have also been widely deploying anti-adblockers [146] and other aggressive obstacles to avoid being detected. Recent research [90] shows that a number of publishers/advertisers are actively circumventing adblockers through various techniques including ad cloaking (e.g., unmonitored web APIs [56]), obfuscation (e.g., ad element randomization [134]), and etc. We envision A^4 advances the state-of-the-art in this arms race and substitutes the first attempt for bypassing learning-based adblockers.

4.8 Conclusions

In this paper, we present the design and implementation of A^4 (**A**ctionable **A**dversarial **A**d **A**ttack), a new adversarial attack targeting the state-of-the-art learning-based adblocker AdGraph. Unlike previous work on generating adversarial samples on unconstrained domains, A^4, explicitly accounts for constraints that arise in the context of the web domain. We show promising results in this unique domain which can have substantive implications in online advertising and other future ML-based web applications.

Chapter 5

Detecting DPI Evasion Attacks with Context Learning

5.1 Introduction

Deep Packet Inspection (DPI) middleboxes are widely deployed as part of modern network security infrastructures [140]. They are stateful , i.e., they not only inspect individual packets, but also reassemble them to form stateful connections defined by a network protocol (e.g., TCP), based on a predefined state machine. To do so, they (for ease of exposition we refer to these as DPIs) need to include a custom network protocol implementation that is often simplified compared to its counterpart on end point platforms (e.g. the OS kernel) due to scarce computation capability, prohibitive overhead and sometimes the need to provide generality in the presence of ambiguous network protocol specifications.

Adversarial Packets. Since the advent of DPI, there have been multiple attacks to circumvent them. These attacks exploit the discrepancies between the DPI's and the OS-level network protocol implementations, to craft subtle yet powerful packets which trigger completely different behaviours on the DPI and at the endpoint (e.g., server). For example, such a packet may be crafted so as to cause the DPI to ignore it, while it still reaches and is accepted by the server. Such packets can potentially contain malicious payloads, and yet successfully bypass the detection of the DPI (because their contents would not be inspected at all). Even worse, prior work shows that in some cases, using only one such packet could cause the DPI to disengage from monitoring of the associated connection, thereby allowing follow-up packets in the connection to pass through without triggering alarms.

Automated Discovery of Evasion Packets. In recent years, as network protocol stacks have become increasingly complex with newly added features [116], a growing number of vendor-specific implementations, and continually evolving specifications, there is a surge in research [58, 93, 136] on automating the discovery of *adversarial packets* as described above, to evade DPIs. By applying principled search [93], genetic mutation [58], or symbolic execution [136], a vast assortment of adversarial packets can be found with little to no manual intervention. While this works towards understanding attackers, it poses a hard challenge from the defense perspective. As protocol stacks evolve and become more complex, the potential discrepancies that may be discovered with automation can grow in theory, and it becomes prohibitive to manually analyze these subtle implementation issues and patch all of them in the code base; in addition it is hard to generate new hardcoded DPI

policies to keep up with new discrepancies that may arise given the pace of rapidly evolving implementations and protocol standards.

Existing Defenses. Given its difficulty, only a few limited countermeasures have been proposed against adversarial packets. Notably, [88], Kreibich et al., apply *traffic normalization* to mitigate the threats of adversarial packets. Specifically, a so-called traffic normalizer is proposed, which acts as a network forwarding element preceding DPI middleboxes, and alters/drops the packets going through the latter as per a predefined set of rules. These rules are manually curated to describe the "normal traffic" (e.g., the IP header length field must never be smaller or greater than the actual header length). This countermeasure unfortunately, cannot scale in presence of automation – the achievable search space of all possible adversarial packets can become large enough to make timely manual curation prohibitive. Moreover, the ever-evolving protocol standards introduce false alarms – a previously correct rule can later cause incorrect decisions (e.g. normal packets being dropped) because of updated implementations. It is also worth noting that as a normalizer, or more generally a *traffic shaper*, [88] provides no detection ability; rather, it blindly alters the traffic stream.

Our Approach. Our goal in this paper is to design a practical defense to effectively detect evasion attempts on DPI middleboxes. Instead of relying on significant manual curation/analysis, we propose a novel, fully-automated (with minimum feature engineering) Machine Learning (ML)-based approach called CLAP (**C**ontext **L**earning based **A**dversarial **P**rotection). CLAP learns the benign (what we call) *context* from only normal traffic traces (i.e., unsupervised), and uses this learning to detect adversarial packets. In other words, CLAP asserts whether the context of the unseen packets "fit" in the associated connection

or not. Specifically, the context of a packet is composed of two types of sub-contexts to describe the aforementioned "fitness" of a given packet:

- **Inter-packet context**, which captures the inter-relationships among different packets in the connection in terms of how their header fields change/evolve over the trace; these changes generally relate to the transitioning of states for a stateful network protocol (e.g., TCP);
- **Intra-packet context**, which captures the inter-relationships among different header fields in a given packet (in terms of the combinations of their values).

By learning the joint distribution (i.e., the *packet context*) of the two (sub-)contexts from benign traffic, CLAP automatically finds violations thereof, caused by adversarial packets.

Motivating Example. To showcase the intuition behind how CLAP works, we present a concrete attack and its detection with CLAP. Bad-Checksum-RST is an attack that has been reported in [58, 136] and shown to be effective against Great Fire Wall (GFW), a state-of-the-art DPI-based censorship system. It injects an ill-formed RST packet with a garbled TCP checksum value after the three-way handshake. Since common endhost TCP implementations perform a rigorous checksum verification but the GFW does not, this injected RST packet is dropped by the endhost but not GFW. As a result, upon seeing a RST packet, GFW would disengage its monitoring of the connection, while the communications between two endhosts would be allowed to continue.

By learning the benign context from a large set of clean network traces, CLAP knows what requirements (inter- and intra-packet contexts) must be met by a packet at its position in the connection (e.g., can it be a RST and if so, should its checksum be correct?);

then, CLAP checks whether the packet conforms (fits in) to a benign packet context. In this case, the RST packet with a bad checksum value that appears after three-way handshake is asserted as violating both the inter- (RST should not take place at this point) and intra-packet (checksum of RST packet should be correct) contexts (based on training) and thus, the evasion attempt is detected.

Contributions. In brief, our contributions in this paper are:

1. We are the first to propose a fully-automated unsupervised learning approach to detect adversarial packets that are crafted to elude DPI middleboxes.

2. Our evaluations on 73 state-of-the-art DPI evasion attacks over realistic backbone traffic captures, show that by only learning from benign traffic, CLAP achieves an overall detection AUC-ROC of over **0.963**, with an average EER of only **0.061** (the two most commonly used evaluation metrics for ML-based IDSs [54, 97, 98, 102]).

3. Our evaluations show that beyond detection, CLAP achieves an average Top-5 localization accuracy (identifying a train of five packets most likely to contain the attack vector) of **94.6%** (Top-3 of **91.0%**).

4. Our performance analysis shows that our pipeline can process over 2,100 packets per second on a single CPU core, with linear scalability.

5. We will open source both our implementation of CLAP, and the used datasets, at https://github.com/seclab-ucr/CLAP to allow reproducibility and future extensions.

5.2 Related Work

ML-based Intrusion Detection System (IDS). As a critical and commonly deployed network infrastructure, intrusion detection systems (IDSs) are designed to detect suspicious traffic and flag them accordingly. Traditionally, IDSs are signature-based and catch malicious traffic that violates a predefined set of rules. This type of IDSs face challenges because their manually curated signatures cannot keep up with emerging threats. In contrast, anomaly-based IDSs do not rely on priori-created signatures; an anomaly-based IDS inspects and classifies traffic by asserting whether it aligns with patterns observed in normal traffic. More recently, ML techniques have been used for anomaly detection, as they are efficient in learning patterns from benign traffic and then distinguishing between normal and suspicious instances on unseen traffic [54, 97, 98, 102]. While with a similar general goal (detecting suspicious traffic), CLAP is fundamentally different because it (1) spots attacks that specifically seek to evade DPI detection, not for general malicious purposes (e.g. DDoS); and (2) uniquely considers packet context that is critical for detecting DPI evasion attacks, while context-agnostic ML-based IDS is unable to do so as will be shown in our evaluations.

[82] is the only attempt towards using ML to discover DPI evasion attacks to our best knowledge. However, the approach proposed in [82] relies on training its model on both benign and malicious traffic traces, which renders a different threat model as compared to what we assume in this paper, and is therefore not comparable directly. In fact, one of the key attributes of CLAP is its ability to detect subtle evasion attacks without a priori knowing about them. For an apples-to-apples comparison, we use a state-of-the-art

unsupervised IDS [102] as one of the baselines in Section 5.4, and show that the state of the art general-purpose IDS is ineffective/inaccurate in detecting DPI evasion attacks because it does not consider context.

Context Learning. The general notion of using context has been explored in computer vision for improving classifier performance [95, 129]. Context refers to co-occurrences of objects that commonly appear together in the same scene, and aids detection/segmentation tasks where certain objects lack inherent patterns to be recognized, but can be inferred by occurrences of other easily-recognizable objects. Note that while these approaches, by mainly characterizing spacial inter-relationships among different objects as context, are generally appropriate for vision applications (i.e., objects in same scene), it cannot be applied directly to network domain applications. Context inconsistency has also been very recently considered for thwarting adversarial examples in computer vision [94] but has never before considered in the network intrusion detection context. Remotely inspired by these efforts, for the first time, we define *packet context* for network traffic data, and propose CLAP to automatically learn and apply this to detect DPI evasion attacks. We consider the unique characteristics of network traffic, and design our system accordingly.

5.3 System Design

5.3.1 Overview

As discussed earlier, CLAP draws on the fact that there are co-occurence relationships between packet fields (intra-packet), and across packets (inter-packet) in a single TCP flow. To re-iterate, these relationships form the packet context. An evasion attack is

likely to tamper with these relationships to confuse the DPI middlebox and CLAP seeks to most efficiently learn and use the packet context to detect such tampering. Our design of CLAP consists of 4 stages:

- (a) Learning benign inter-packet context by training a RNN model whose task is to predict the transitions across a set of connection states (not only high-level TCP states but also subtle verdicts/states, as described later), on benign traffic traces;
- (b) Fusing/concatenating the benign inter-packet context (i.e., as discussed later gate weights from (a)'s RNN model represent inter-relationships among different packets) and intra-packet context (i.e., combinations of packet header fields) to generate benign *context profiles*;
- (c) Learning benign holistic packet context by characterizing the distribution of context profiles generated in (b);
- (d) Detecting DPI evasion attacks by verifying whether the context profile of unseen packet trains violates the distribution of benign context profiles as learned in (c).

We show diagrams that depict Stage (a)/(b)/(c) of CLAP in Figure 5.2, and (d) in Figure 5.3.

5.3.2 Threat Model

Before diving into the technical details of the design of CLAP, we establish the threat model we consider. As previously mentioned, CLAP is designed to protect stateful DPI middleboxes against evasion attacks. We capture the threat model associated with

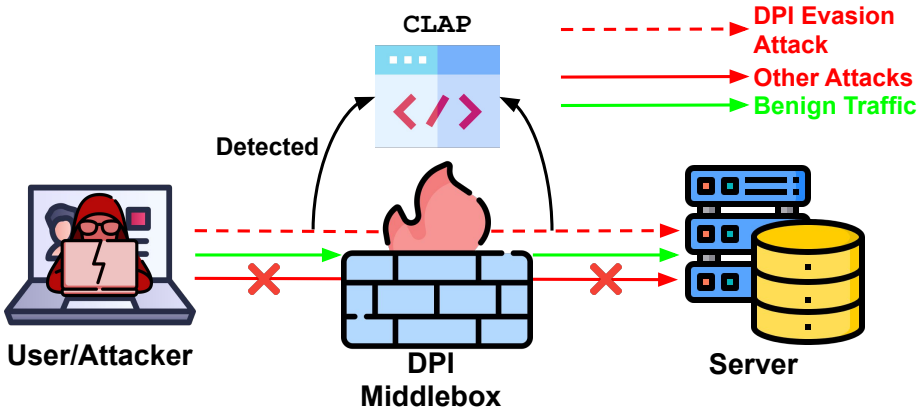


Figure 5.1: Threat model of DPI with CLAP

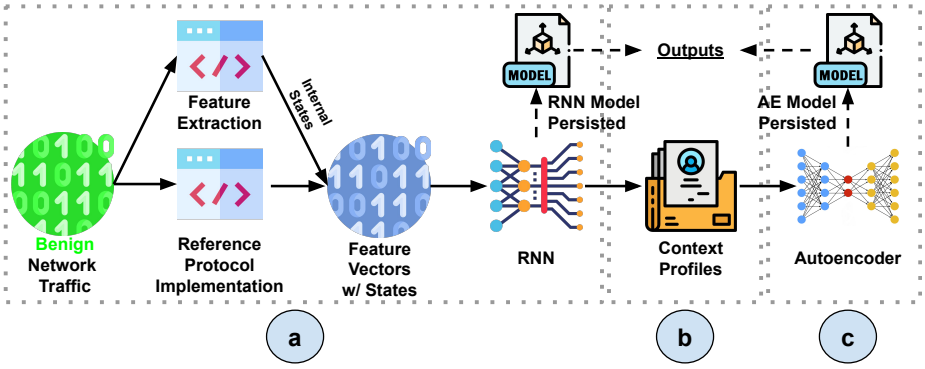


Figure 5.2: Training phase of CLAP

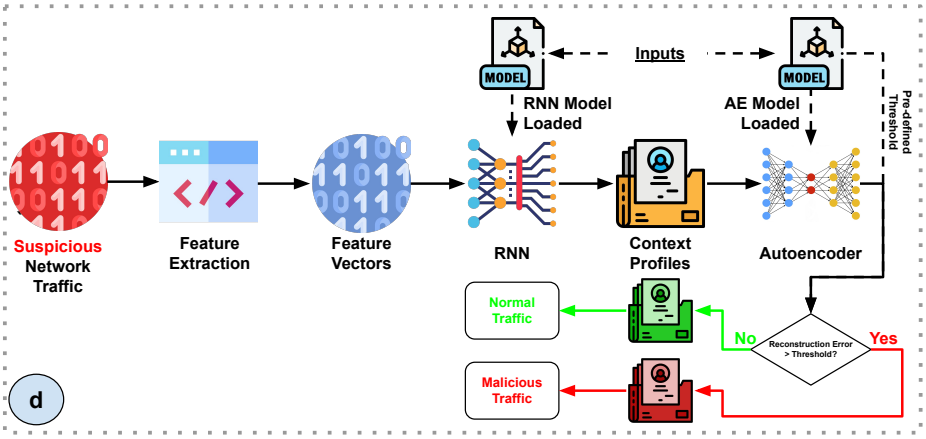


Figure 5.3: Testing phase of CLAP

common DPI middleboxes relating to our work, and the role of CLAP in Figure 5.1. We assume both the DPI middlebox and CLAP are located in between the client and server, and capable of reading all packets going through. Commonly, DPI inspects payloads of these packets and detects malicious contents in those payloads (i.e., "Other Attacks" in Figure 5.1). When the attacker launches the DPI evasion attack from client side, he/she injects specially crafted packets (i.e., adversarial packets) to cause discrepancies in behaviours between the DPI and the server. The discrepancies cause the follow-up malicious traffic to not be inspected by the DPI (but can still reach the server). CLAP is designed to specifically detect such adversarial packets. Note that we limit the scope of our target evasion attacks to those that only involve header fields manipulations, due to (1) scarcity of payload-related evasion strategies; and (2) prohibitive overheads in training for benign payloads. We only focus on the TCP protocol in this work since it is arguably the most popular transport layer protocol and is most commonly targeted in DPI evasion attacks. Since CLAP does not depend on the DPI itself, it can function interdependently (for forensic analysis) and does not rely on anything besides raw traffic that traverses the DPI middlebox (i.e., PCAP captures). Hence, CLAP can not only be deployed as an online detector complementing existing DPIs to detect evasion attacks (with affordable overhead as shown in Section 5.4.4), but also be used as a forensic tool to analyze the traffic captures offline.

5.3.3 CLAP Design

Next, we describe the different components in CLAP that help achieve its overarching goal.

(a) Learning Inter-packet Context.

Goal. The first stage of CLAP involves training a RNN model to drive it towards learning the inter-packet context as defined previously. As established earlier, the inter-packet context essentially captures the relationships among the header fields (co-occurrence) in a train of packets. We reiterate that this context strictly describes the relations across *different* packets, and therefore does not concern itself with any relations/combinations of header fields within the same packet.

In order to capture these inter-packet relationships with a ML model, we need to design a corresponding learning task. Since the transitions of a TCP state machine depend on such relationships across a sequence of given packets, we use a ML classifier that predicts these transitions; such a classifier will need to learn the temporal structure across packets. The best option among various neural network choices for doing so is a GRU-based RNN model. Based on this general architecture, we design a customized model that takes the header fields of each packet as inputs, and predicts the corresponding states to which the reference TCP state machine will transition as a result of this packet, as model output. Note that we are not interested in the classification result of the RNN, but rather require the distribution of the benign inter-packet contexts which we draw from the weights of the gates in the neural network architecture, as discussed in the next subsection. This requires exceptionally high performance for the task we design for RNN (connection state prediction), which from our evaluations (detailed in Table A.7), is achieved with an overall prediction accuracy over 0.99).

RNN/GRU. RNNs are a class of neural networks that are widely used to model temporal data. In brief, they contain a recurrent cell that processes one element in the input sequence a time, considers both the current input and a memory state from the previous unit to output a new memory state. Beyond accomplishing classification tasks, the chaining of these repeated units in RNN models have also been highly successful [95, 129] in generally modeling and encoding the interrelationships across the input sequence. Among RNN cells, the Long Short-Term Memory Unit (LSTM) and the Gated Recurrent Unit (GRU) are considered the state-of-the-art architectures. Both of them consist of a "gating" mechanism, wherein gates that are in the cells update weights that represent the relationships across input sequence. We pick GRU in this work as it provides performance that is comparable with LSTM but incurs considerably lower overhead, and note that LSTM is also a viable option.

Input Feature Set. In order to cover (1) all header fields that influence the transitions of the TCP state machine and (2) header fields that can possibly be manipulated by DPI evasion strategies (i.e., all non-tuple-related, non-optional IP/TCP headers, and a few common TCP option headers, as specified in [114, 119]), we include 37 header field values as features (7 for IP and 25 for TCP, full list in Table A.9) from the IP and TCP headers. To reiterate, we do not consider payload-related features here because (1) they are irrelevant to TCP state transitions; (2) there are only a very small fraction (7 out of 80 from [58, 93, 136]) of DPI evasion attacks that involve manipulating packet payloads; (3) it is prohibitively challenging to learn distributions of unstructured data such as benign payloads; we leave this as an open question to examine in the future; and (4) most public traffic archives

(including what we use in the evaluations) strip payloads in the traces for privacy concerns. We alto follow the general principle of using these fields in the raw form to the extent possible to avoid heavy feature engineering (i.e., only need minimum pre-processing such as validating checksum, as detailed in Table A.9), and show that the RNN is capable of inferring the connection states without requiring extensive domain knowledge.

Labeling. According to the IETF standard [119], TCP defines 11 different states (e.g. `SYN_SENT/SYN_RECV/ESTABLISHED`) that we refer to as *master TCP states*. In addition to these states, in order to enrich the learned packet context, especially in the rather coarse grained ESTABLISHED state, we also include the more subtle, in-/out-of-window states (i.e., whether an incoming packet is within the recipient's receive window) collected from the reference TCP implementation to be part of the label. Therefore, the label we use to train RNN is the concatenation of the master TCP state and the in-/out-of-window subtle *state* [1], resulting in a total of $11 * 2 = 22$ potential classes (an in-/out-of-window possibility is included for each of the 11 master states); these are listed in Table A.9. In order to collect reliable and accurate states for labeling the training data of our RNN, we resort to OS-level (e.g. Linux) TCP stack implementations and instrument their relevant modules to expose our desirable states (i.e., master TCP state and in/out-of-window subtle state). We replay the benign training traffic captured on the instrumented platform to harvest their corresponding states as labels (the details of the setup are in Section 5.4.1).

[1] Although, we are slightly misusing the term, when we refer to state from hereon, we not only include the traditionally defined TCP states, but also a packet classification/verdict (in-window or out-of-window) that strongly relates to inter-packet relationships, to better drive the RNN model to learn bengin inter-packet contexts.

Training. To put things together, we now formally describe how we train the RNN model. Consider a benign network connection consisting of n packets $P_{1...n}$, where $P_i = [F^1_{IP}, ..., F^8_{IP}, F^1_{TCP}, ..., F^{29}_{TCP}]$ (i.e., F^i_{IP} represent the features from the corresponding packet's IP header fields, and F^i_{TCP}, those from the TCP header fields). We train a RNN model M_{GRU}, with GRU as its cell architecture, by feeding $P_{1...n}$ and their corresponding ground-truth labels (i.e., states from TCP state machine) $L_{1...n}$, and executing standard error back-propagation [62]. The standard multi-class cross entropy loss function in Equation 5.1, where L_i and $\hat{L}_i$ are the probabilities relating to class i of the ground-truth label vector and the predicted label vector, respectively, is used as the loss function for training the RNN. It measures the error between the current RNN model output (i.e., a vector of probabilities) and the ground-truth label (i.e., a vector of values of 0 except that the index of expected/ground-truth state is 1). The error is propagated back to the prior layers of the RNN to update its parameters until the model produces the correct output. In the next subsection, we describe how the intermediate/latent gate states of the training GRU are used to build context profiles.

$$Loss_{CrossEntropy}(L, \hat{L}) = -\sum_i L_i log(\hat{L}_i), \quad \text{where } \hat{L} = M_{GRU}(P) \tag{5.1}$$

(b) Fusing Inter- and Intra-packet Contexts.

Goal. Once the RNN model is trained, its gates contain the inter-packet context learned from benign traffic traces (described in more detail in the next paragraph). Next, we need to extract the intra-packet context and fuse it with the inter-packet context to form the *context profiles* of a packet. Recall that the intra-packet context is defined by the

relationship/combinations across header field features within the packet. We fuse the two contexts by simply concatenating them (i.e., gate weights and header field values) into a unified feature vector which is referred to as the context profile for that packet. This fusion strategy enables the autoencoder in Stage (c) (to be described) to learn the joint distribution of both contexts. By examining this joint distribution, CLAP is capable of exposing attacks that violate (1) only the inter-packet context; (2) only the intra-packet context; and (3) both sub-contexts simultaneously.

Gate Weights. Figure 5.4 provides a closer look at the internals of GRU cells. For one such cell, besides the input (i.e., x_t in Figure 5.4) and the output state (i.e., h_t in Figure 5.4), there are also "gates" for optionally letting information through [62] (i.e., reset and update gates as marked in Figure 5.4). In order words, these gates explicitly control whether the output classification (i.e., the TCP state described earlier) of a given packet strongly relates to its previous packets, or not. Specifically, if at the current time step $t1$ the gate weights with respect to a previous packet P_{t2} at time step $t2$ are large, it means the features of P_{t2} contribute greatly to the classification of the next output at $t1$ (and vice versa). As one can see, the gate weights are ideal means to characterize the dependencies or inter-relationships across the different packets in a connection, or in other words, capture the inter-packet context.

Chain Graph. To visualize the context profile of a packet, we can represent the same as a graph-based model wherein the packet header features are the graph nodes, and gate weights are the edges connecting packets (nodes). Thus, the train of packets can be represented as a chain-shaped graph. The packets header field features are the node features, and the edges

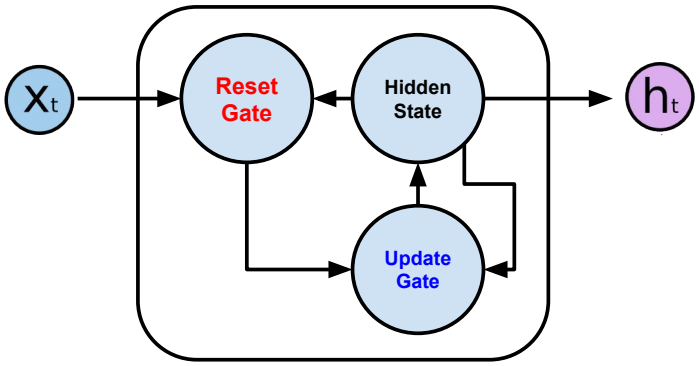


Figure 5.4: Internals of GRU cell

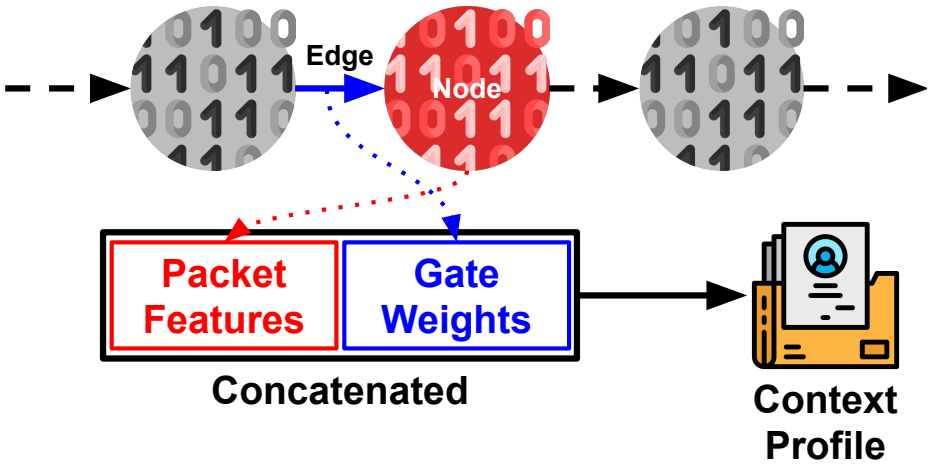


Figure 5.5: Representation of context profile as chain graph

between two adjacent nodes (consecutive packets) are the GRU gates that describe the inter-relationship between the two packets. The resulting connected chain graph, models the network trace. The gate weights (edges) in the graph, (which are learnt during training) control the information propagation between the adjacent packets (nodes). We depict this chain like graph model in Figure 5.5.

Amplification Features. While we have included all necessary packet header fields as the feature set for learning the intra-packet context, we discover that some subtle context vi-

olations are challenging to capture in practice with just these features. These relate to extremely small perturbations in terms of the associated changes of certain features; these help in successfully evading the DPI, but without amplification seem too insignificant for the ML classifier (specifically the autoencoder) to recognize. To address this, we augment our feature set with two types of *amplification features*, that are incorporated into the context profile; these features, crafted based on domain knowledge, amplify the aforementioned subtle context violations such that the distribution discrepancy they cause are easily captured by the autoencoder that is described later when we discuss Stage (c). In particular, the amplification features that we design are: (1) *out-of-range features*, which indicate whether the packet's associated numerical header field value is out of the range of what has been observed in the benign training traces; and the (2) *equivalence relation feature*, which indicates whether an expected equivalence relationship with respect to certain header field values is maintained (e.g., TCP payload length = IP total length - IP header length - TCP data offset) or not. We provide the full list of the 15 amplification features (all belonging to the two types) with their semantics in Table A.9. We refer to the concatenation of the raw header field value features and the amplification features as *packet features*.

Concatenation. We next provide details on "how exactly" we generate the context profile. Consider a GRU taking n input features with its hidden state size also being n-dimensional; in such a case, the size of its gates should be equal to the hidden state (i.e., n) [62]. As previously discussed, the context profile is the concatenation of the packet features and the gate weights, as defined in Equation 5.2. We provide a full list of all packet features in Table A.9. Upon generating the context profiles for all the packets in a train, we can

concatenate consecutive packet profiles to form a "stacked profile." This design aggregates multiple context profiles from consecutive packets, and therefore explicitly embeds inter-packet relationships into the stacked profile (in addition to existing gate weights that are also designed to capture inter-packet context); this in turn provides profiles with richer inter-packet context encoded. As one might expect, we find that this helps improve performance and eventually use it in our evaluations.

$$\begin{aligned} CxtProf &= [P_{IP}, P_{TCP}, P_{amp}, G_{forget}, G_{update}] \\ StackedCxtProf &= [..., CxtProf_{t-1}, CxtProf_t, CxtProf_{t+1}, ...] \end{aligned} \tag{5.2}$$

(c) Learning the Joint (Context) Distribution.

Goal. Once the benign context profiles that contain both inter- and intra-packet contexts are generated as described in (b), CLAP needs to learn their joint distribution so that it can later detect suspicious packets that violate the benign context (i.e., joint distribution).

Autoencoder. Antoencoders are a class of neural networks that characterize the distribution of given training samples by forcing the networks to reproduce the inputs themselves as model outputs. In other words, they are tasked to encode a given input to a compressed latent space (bottleneck layer), and decode it to recover the input as much as possible. During this process, the autoencoder learns the compressed representation from the training data (i.e., benign context profiles in our case) distribution as the features of its bottleneck layer, and the reconstruction error between the real input and recovered input (model output) is considered an ideal metric to characterize an input (context profiles) in terms of how close it is to the learned distribution. If a context profile traversing an autoencoder

trained on benign context profiles, exhibits a high reconstruction error, we infer that it deviates from, or violates the benign context. Autoencoders are considered as the state-of-the-art means for anomaly-detection tasks [145]. We use an autoencoder here to learn the distribution of benign context profiles, and then determine if the context of unseen packets is consistent with what is observed in benign packet traces in (d).

Training. We train the autoencoder with benign context profiles obtained. The L1 loss function is used to measure the reconstruction error and is the sum of the all the absolute differences between the true value (ground-truth context profile) and the predicted value (reconstructed context profile). During training, by minimizing the L1 loss, the autoencoder learns to characterize the distribution of benign context profiles. The choice of the L1 loss function also relates to its excellent performance in handling dense input data, meaning data wherein there is little to no sparsity (i.e., features with zeroes as values). This suits the properties of the context profiles generated in Stage (b), because both the packet features and the gate weights are dense and rarely have 0 as values.

$$Loss_{L1}(X_{input}, X_{output}) = \frac{1}{n}\sum_{i=1}^{n} |X^i_{output} - X^i_{input}| \quad \text{where } \{X_{input}, X_{output}\} \in \mathcal{R}^n \tag{5.3}$$

(d) Verification.

Goal. Lastly, CLAP with its trained RNN (to generate gate weights for context profiles) and autoencoder (trained using those profiles), is to be deployed online (i.e., it encounters previously unseen packets) to detect possible DPI evasion attacks. Recall that the autoencoder from Stage (c) can only compute the reconstruction error with respect to each context profile. Given that the input to CLAP would be connections/sequences of packets

and our goal is to provide a connection-level detection conclusion, we need to first determine a strategy for CLAP to compute a score (which captures the likelihood of whether there are evasion packets within the connection for a given connection that contains sequence of reconstruction errors produced by the trained autoencoder. Then, CLAP must analyze the context profiles for unseen packets (their conformance to benign profiles) and use the chosen score (discussed below) to assert if this connection is adversarial.

Adversarial Score. Once the autoencoder is trained, it produces a numeric reconstruction error for a given (stacked or non-stacked) context profile. Consider a connection that consists of n packets in total. Let us assume that stacked context profiles containing t packets, are generated in a sliding-window fashion (i.e., concatenation of every t consecutive single-packet profiles from the beginning to the end of a unidirectional traffic sequence; these profiles are overlapping) are obtained. Specifically, with this approach, CLAP generates $n - t + 1$ such stacked profiles (i.e., leading to $n - t + 1$ tests and thereby, reconstruction errors at the autoencoder stage) for the connection. This sliding window enumerates all possibilities of temporal contexts, and can be expected to provide the best coverage. CLAP now needs to capture a "summarized" value that characterizes the "overall profile" of the connection, by means of these enumerated profiles. There is a spectrum of approaches for characterizing a group of observations in general, ranging from basic statistical quantities such as maximum/minimum, variance, mean, median, to more advanced methods such as training a separate autoencoder for the summarization [99].

We examine the reconstruction error trends from adversarial connections, towards empirically choosing a proper metric. An example is shown in Figure 5.6, where an ad-

versarial packet introduces a spike in the error value (around the time it is encountered) and then, the error level falls off to get closer to that with benign profiles. Based on this observation (which is expected since the sliding windows closest to the adversarial packets are likely to show the highest error), we propose a *localize-and-estimate* approach to choose a metric; this maximizes our odds of distinguishing adversarial from benign connections in the testing phase. Specifically, we first localize (identify the position within the sliding window) the profile with the maximum reconstruction error in the connection; and (2) sample the mean reconstruction error over the window (of 5 profiles in this work) by choosing the profile with the maximum reconstruction error as center. We refer to this mean as the *adversarial score* for the connection. We believe this newly designed metric best captures the most distinguishing (the spike) part of the reconstruction error sequence in a connection, provides the best detection performance. In addition to the score, to make a Boolean classification (attack or no attack), we also need a threshold to determine at what level of the adversarial score, do we consider the connection to be an attack. The choice of this threshold will provide a trade-off between true and false positive rates. We leave the freedom of choosing the threshold to the deployer of CLAP towards achieving the appropriate trade-off; however, our results in Section 5.4.2 offer possible choices for these thresholds (by means of ROC curves).

Deployment. Lastly, with the 3-tuple $\{M_{GRU}, M_{AE}, TH_{Adv}\}$ (i.e., trained GRU-based RNN model, trained autoencoder model and selected threshold for adversarial score) generated in the previous steps and an incoming (suspicious) connection C_{Sus}, the deployer of CLAP first executes the pipeline in Figure 5.3 to compute the adversarial score for C_{Sus},

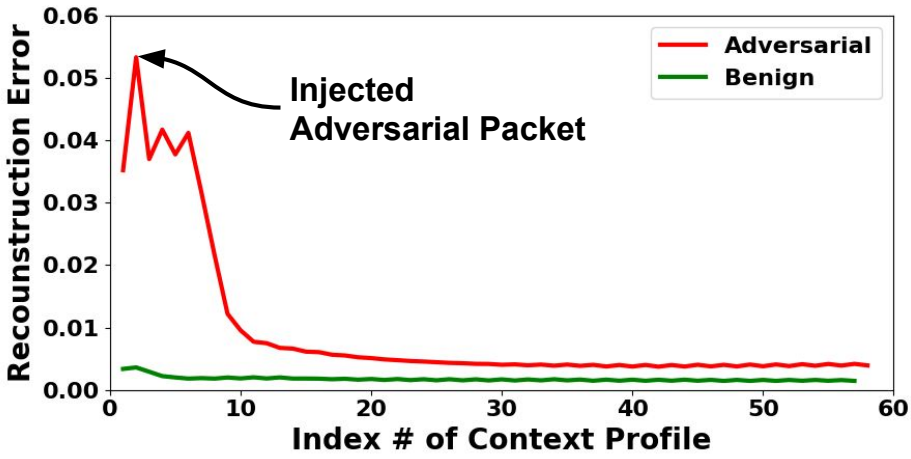


Figure 5.6: Typical trend of reconstruction errors across a connection

and then compares it with TH_{Adv} to determine if the connection is adversarial or not. Note that beyond binary classification (i.e., adversarial or benign), CLAP can also pinpoint the most "suspicious" packets in a connection, by localizing the ones that bear the highest reconstruction error (i.e., the first step of "localize-and-estimate" approach). We will evaluate both the detection and localization accuracies in the next section.

5.4 Evaluations

In this section, we provide comprehensive evaluation results with regards to different aspects of CLAP, viz.,: (1) effectiveness analysis in terms of how well it detects and localizes adversarial packets, compared to other baselines; (2) case studies with in-depth analysis on its performance on certain attacks; and (3) performance analysis in terms of how fast it can process network traffic in practice.

5.4.1 Setup

Dataset. We first describe the dataset used in our evaluations. For learning benign packet contexts that are as complete and sound as possible, the dataset of benign traffic traces must guarantee that it (1) only consists of benign (i.e., no DPI evasion attempts) flows; and (2) is adequately representative i.e., non-adversarial packets including those with uncommon but legitimate patterns are well-covered. Given these requirements, we select the MAWI Traffic Archive [10] that provides PCAP captures of a backbone network in Japan. MAWI traffic traces are considered one of the state-of-the-art long-term measurements on wide-area/global Internet and have been used in prior networking/security research [54, 61, 69]. Note that the payloads of MAWI traces (and most other public datasets) are all stripped (payloads are not visible) for privacy concerns, making it impossible to use for detecting payload-related attacks (e.g. segmentation-based attacks) – we adjust the corpus of evaluated attacks accordingly. Table A.6 (in Appendix due to space limitation) lists basic statistics of the traffic capture we use.

Recall that Stage 1 of CLAP needs a reference TCP implementation to generate the internal states required for training the RNN model. We select the `conntrack-tools` module [55] in Linux's `Netfilter` sub-system that provides standard and reliable infrastructures for packet inspection. We instrument relevant code in Linux kernel version 5.6.3 to expose the required states, and use it as the traffic "replayer" for collecting labels for RNN training.

Simulated Attacks. For generating the the traffic that seeks to evade the DPI middlebox, and to include these to form the dataset, we adopt a simulation-based approach. We

integrate these traffic flows into the benign MAWI traffic traces. Specifically, we thoroughly analyze the evaluated 73 DPI evasion attacks proposed in [58, 93, 136], and implement a simulator that injects the modifications incorporated in these to evade the DPI middlebox, into the MAWI traffic traces (i.e., PCAP files). We acknowledge that this PCAP-level simulation might potentially introduce some slight divergence in behaviors, compared to what happens in live DPI evasion attacks (e.g. the timings associated with the injected packets is estimated in our simulation, which could differ from the live case due to unpredictable congestion effects). However, we argue that (1) these differences do not disrupt the underlying mechanisms of DPI evasion attacks and thus, do not affect our evaluation results/conclusions; and (2) the open-source implementations released by [58, 93, 136] do not support a direct replay of the attack traces and, thus we are unable to verify those directly; however, the simulations are a faithful reproduction of those attack behaviors and we assume them to yield similar success against DPI middleboxes.

Baselines. We compare CLAP's performance with that of 2 baselines to showcase its effectiveness. Baseline #1 reuses the same pipeline as CLAP, but (1) removes all gate weight features from its context profiles, (2) limits the length of the profile to single packet. In other words, only intra-packet context features are considered to eliminate inter-packet context information. One can think of this as a "temporal context" agnostic version of CLAP. Baseline #2 is the faithful reproduction of a state-of-the-art anomaly-detection-based IDS [102]. This IDS also leverages autoencoders as its underlying model and the paper claims that it targets a broad spectrum of attacks. We will show the results with these baselines, in comparison with those with CLAP's in Figures 5.7, 5.8 and 5.9.

5.4.2 Effectiveness Analysis

Takeaway: CLAP achieves an average AUC-ROC score of **0.963** (vs. 0.846 [-12.1%] for Baseline #1 and 0.498 [-48.3%] for #2), Equal Error Rate (EER) of **0.061** (vs. 0.198 [+224.6%] for Baseline #1 and 0.502 [+723.0%] for #2) in detecting attacks; it also achieves a **94.6%** Top-5 (meaning the localized packet is within a window of five packets from the identified point), **91.0%** Top-3 (the localized packet is within a window of 3 packets), and **76.8%** Top-1 (exactly identify the position of the adversarial packet) accuracy in localizing the positions of injected adversarial packets.

Evaluation Metrics. For our evaluations of CLAP, we use the following, most commonly adopted metrics in the state-of-the-art ML-based networked systems research [54, 97, 98, 102]. "Area Under of the Receiver Operating Characteristic Curve" (AUC-ROC) is our first metric of interest; it captures the trade-off between the True Positive Rate (TPR) and the False Positive Rate (FPR) by considering a comprehensive set of reconstruction error thresholds, and is considered a comprehensive metric for evaluating binary classifiers. The higher the AUC-ROC, better the classifier. Complementary to AUC-ROC, EER is the point on ROC curve where the False Positive Rate is equal to the False Negative Rate; this metric is sometimes considered to be a more balanced metric to evaluate a classifier compared to AUC-ROC. The lower the ERR, the more accurate the detection. Lastly, *Top-N Hit Rate* is defined by the percentage of the connections where the context profiles with the N-highest reconstruction errors produced by CLAP, intersect with actual injected adversarial packets from among all tested connections. In other words, it measures the accuracy of the localization step in CLAP's "localize-and-estimate" adversarial score

Results/ Approach	Mean AUC-ROC for [136]	Mean EER for [136]	Mean AUC-ROC for [93]	Mean EER for [93]	Mean AUC-ROC for [58]	Mean EER for [58]
CLAP	**0.953**	**0.072**	**0.952**	**0.082**	**0.988**	**0.024**
Baseline #1	0.829 (-13.0%)	0.218 (+202.8%)	0.805 (-15.4%)	0.232 (+182.9%)	0.913 (-5.9%)	0.133 (+454.2%)
Baseline #2	0.501 (-47.4%)	0.501 (+595.8%)	0.500 (-47.5%)	0.500 (+509.8%)	0.491 (-50.3%)	0.504 (+2000%)

Table 5.1: Breakdown of average detection performance for strategies in [58, 93, 136]

generation approach, in terms of how well top candidates marked by CLAP capture the real adversarial packets. With accurate localization, CLAP can pinpoint the exact positions of adversarial packets for the purposes of forensic analyses. The higher the hit rate, the more accurate the localization.

Detection Performance. For evaluating CLAP's detection accuracy, we must first inject the attacks into the benign traffic dataset to form its adversarial counterpart for the testing samples. As previously discussed, DPI evasion attacks all function by either injecting new packets, or modifying existing packets in a connection. However, the exact injection and modification methods differ from attack to attack. We therefore provide a breakdown/taxonomy of the evaluated attacks and the projects they were discovered from.

Figure 5.7 shows the evaluation results of attacks presented in [136]. Since all of these attacks work at TCP layer by modifying the TCP header fields of injected or existing packets, they can be categorized based on (1) the type (i.e., TCP flags) of the packets that are injected or altered, and (2) the exact header modifications. In Figure 5.7, for each attack (i.e., the title of each bar plot), the first line is the key TCP flag (e.g., SYN) in the injected/altered packets, and the second line indicates how header fields are modified (e.g., Bad SEQ means changing the SEQ number to an invalid value). As shown in Table 5.1,

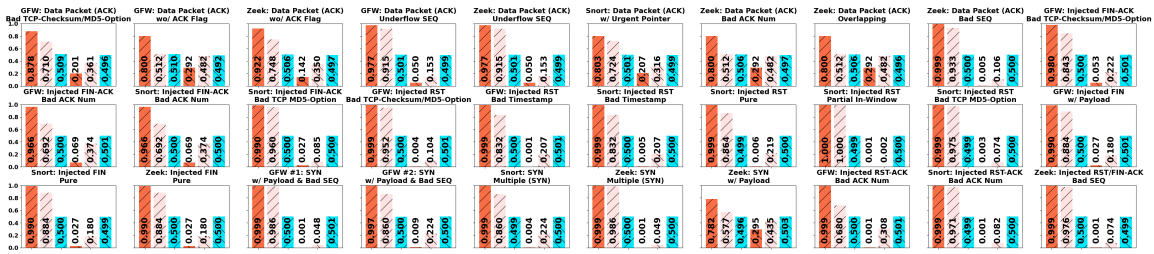


Figure 5.7: Per-strategy detection accuracy of CLAP in detecting the different attacks (shown in title) from [136]

CLAP outperforms both baselines in detecting evasion strategies from [136] by a significant margin.

Figure 5.8 shows the results relating to attacks proposed in [93]. Unlike [58, 136], these attacks target DPI-based traffic classification systems (e.g., is it YouTube traffic or not?) by manipulating header fields across the TCP/IP layers in certain packets in a connection; these classifiers make decisions by examining an arbitrarily long subset of data packets called the matching packets, which are transferred after the initial TCP handshake. To evade the classifier, evasion packets are inserted in front of all of these matching packets. However, without knowing what classification possibilities the attacker is trying to evade we cannot know the exact number of evasion packets that are inserted (i.e., they vary depending on the content that requires evasion). Hence, we simulate two extreme cases wherein there is (a) a single matching packet *min* and (b) where there are a $max = 5$ matching packets as considered in the [93] paper. Note that these packets are sent after the connection transitions into the `ESTABLISHED` state. With the above strategies we show the

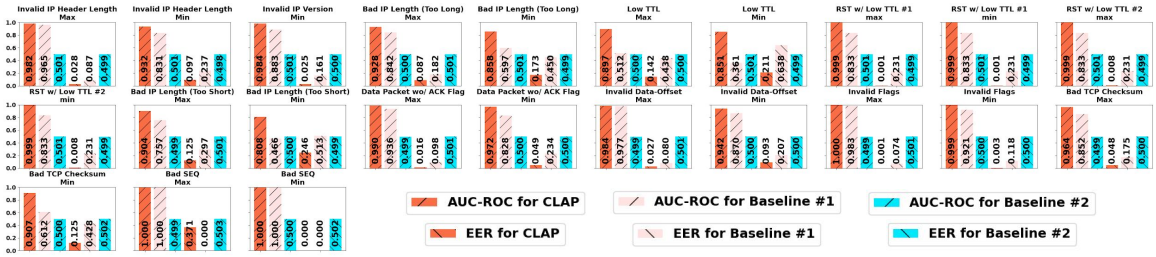


Figure 5.8: Per-strategy detection accuracy of CLAP in detecting the different attacks (shown in title) from [93]

corresponding detection accuracies in Figure 5.8. Again, CLAP outperforms both baselines in detecting evasion strategies from [93] by a significant margin, as reported in Table 5.1

Lastly, Figure 5.7 reports the detection accuracies with respect to attacks from [58]. Unlike [136] and [93], these attacks (1) are executed/injected rather blindly and all data packets (i.e., packets transferred in `ESTABLISHED` TCP state after completing the handshake) are altered; and (2) consist up to 2 different modifications in one attack strategy. Therefore, in Figure 5.9, when labeling each attack, the first and second lines describe the first and second modification type ("/" means only one modification for that strategy), respectively. We again observe significant gains in terms of detection performance, with CLAP over the other baselines. Once again, CLAP wins over both baselines in terms of the detection accuracy against attack strategies from [58] by a considerable margin, as shown in Table 5.1.

Analysis. From the detection performance results, alluded to above, across different evasion strategies, we have the following observations: (1) CLAP consistently outperforms both Baselines #1 and #2, in terms of all the considered metrics and across all attack strategies; (2) all evaluation metrics indicate that CLAP tends to perform exceptionally well on all

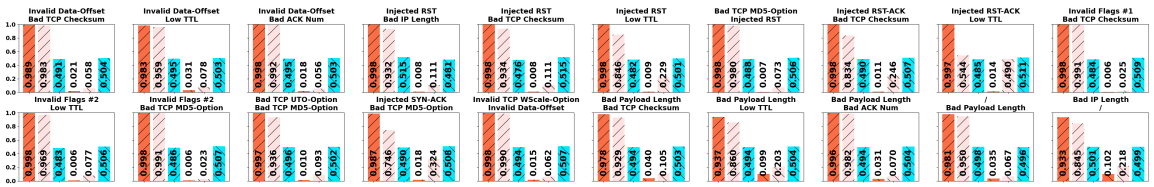


Figure 5.9: Per-strategy detection accuracy of CLAP in detecting the different attacks (shown in title) from [58]

evasion strategies that involve inter-packet context violations (e.g., Pure RST strategy from [136] with an AUC-ROC of 0.999, positioned at row #2 column #7 in Figure 5.7) and most strategies that involve intra-packet context violations (e.g., Invalid Data Offset/Bad TCP Checksum from [58], positioned at row #1, column #1 in Figure 5.9); however, the performance is not as good on a small fraction of strategies involving intra-packet context violations (e.g., SYN w/ Payload attack from [136] with an AUC-ROC of 0.782, positioned at row #3, column #7 in Figure 5.7 and, Low TTL attack (min) from [93] with an AUC-ROC of 0.851, positioned at row #1, column #7 in Figure 5.8). We believe that this is because, for these intra-packet context violations, even with amplification features, the quantum of the adversarial perturbation/modification imposed onto the packet is still considered too insignificant by the autoencoder in CLAP, to be spotted. Note however that, even so, CLAP still (1) detects a large fraction of evasion attempts of these types (an AUC-ROC ¿ 0.75); and (2) provides considerable improvements (¿ 30%) over both baselines.

Localization Accuracy. Next, we evaluate the accuracy of CLAP in localizing the injected adversarial packets; this is the first step of the "localize-and-estimate" approach in computing the adversarial score for a given connection (see Section 5.3.3). Here, our goal is: (1) evaluating the localization ability of CLAP, and (2) providing an analysis of

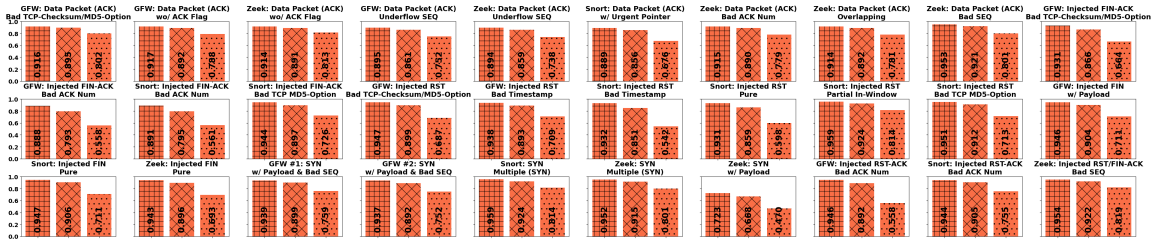


Figure 5.10: Per-strategy localization accuracy of CLAP in detecting the different attacks (shown in title) from [136]

Figure 5.11: Per-strategy localization accuracy of CLAP in detecting the different attacks from [93]

the design choice made in CLAP in association with the computation of adversarial score. We do not consider baseline approaches for localization evaluations, because neither of the baselines are sufficiently accurate in detecting adversarial packets in the first place and furthermore, do not consider the temporal dependencies across a train of packets (indicating their inadequacy in terms of localization abilities). As previously discussed, we use the Top-N hit rate as the metric to measure the localization accuracy. We report the Top-5, Top-3 and Top-1 rates in Figure 5.10, 5.11 and 5.12 with the same categorizations adopted in the detection accuracy evaluations. Specifically, the three bars in each plot from left to right correspond to Top-5, Top-3 and Top-1 rates, respectively.

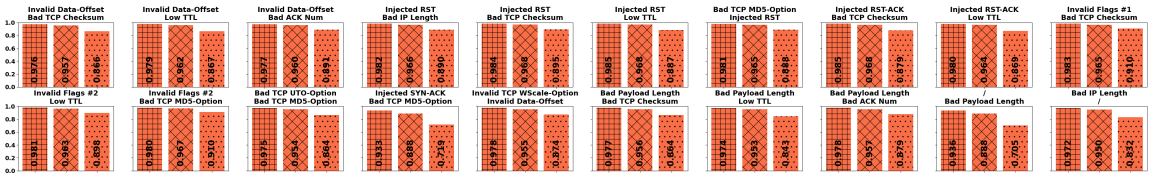


Figure 5.12: Per-strategy localization accuracy of CLAP in detecting the different attacks from [58]

Analysis. Our general observations with regards to the localization results are as follows: (1) as one might expect, the accuracy varies when we require different levels of localization; specifically, achieving Top-1 (with an average accuracy across all strategies of 76.8%) localization (as expected) is much more challenging than Top-5 (average accuracy 94.6%) and Top-3 (average accuracy 91.0%) localization. This in a way highlights the importance of our design choice with regards to using the "mean across a sliding window" surrounding the context profile with the maximum reconstruction error as the adversarial score described previously; (2) the better the localization, the better the detection performance, as more accurate localization yields richer and more directive context information for CLAP to utilize.

5.4.3 Case Studies

Having provided a holistic picture of the performance of CLAP across a large set of DPI evasion attacks, we now pick two concrete attacks to exemplify why CLAP can detect them accurately. Specifically, we pick an attack that is discovered due to violations of the inter-packet context and an attack that is caught for violating the intra-packet context. These case studies also serve to validate the core design of CLAP.

Note here that we have undertaken a careful analysis of all the attacks reported and are able to reason about them; we believe that the two attacks that we choose to magnify in this section are representative of the entire set. We also provide a summarized categorization of all 73 evaluated strategies from [58, 93, 136] into the two context violation types, in Appendix A.10. Recall that our Baseline #1 approach does not account for inter-packet context since it only considers single-packet features (gate weights from RNN are ignored). In other words, it only captures intra-packet context towards detecting DPI evasion attacks; thus, any evasion strategy that exhibits disparity between CLAP and Baseline #1, is considered as one that invokes a inter-packet violation for evasion. Following this principle, if the AUC-ROC disparity between CLAP and Baseline #1 is greater than a chosen threshold TH_{inter} (we pick 0.15 here), we categorize the attack as one invoking an inter-packet context violation; otherwise, the attack is considered as causing an intra-packet context violation. Based on this rule of thumb, we categorize 27 out of the 73 evasion attack strategies as primarily inter-packet violations, and 49 as primarily intra-packet violations. Note that this does not mean these two categories are strictly disjoint. While some strategies violate both contexts, we categorize them based on the main violation (manifested by the significant accuracy discrepancy).

Inter-packet context violation. Recall that the inter-packet context refers to the inter-relationships across different packets in a connection. Among all evaluated attacks, there are many that evade DPI detection by injecting manipulated packets that cause a behavioural discrepancy between the DPI and endhost only when the TCP state machine is in a specific state. For example, the RST with Bad Timestamp strategy from [136]

Category/ Metric	Inter-packet Context Violation (24 Strategies)		Intra-packet Context Violation (49 Strategies)	
	CLAP	Basline #1	CLAP	Baseline #1
Mean AUC-ROC	**0.925 (+37.6%)**	0.672	**0.980 (+6.2%)**	0.923
Mean EER	**0.109 (-70.0%)**	0.364	**0.039 (-68.3%)**	0.123

Table 5.2: Breakdown of inter vs intra-packet violation detections

injects a RST packet with an invalid TCP timestamp option value, specifically when the target connection is in the `SYN_RECV` state. Since this evasion packet is then only dropped by endhost, but accepted by target DPIs (e.g. Zeek and GFW), it tricks the DPI into disengaging the monitoring subsequently. The end host, which is in exactly `SYN_RECV` TCP state, receives the follow-up packets that effectively bypass DPI detection. We determine that this strategy primarily violates the inter-packet context because it (1) it is successful only when the current TCP state is in a specific state (i.e., a Bad Timestamp RST packet would not cause behavioural discrepancy between DPI and endhost in `ESTABLISHED` state) and, (2) the validity of the TCP timestamp option is determined by the timestamps in previous packets. In other words, in order to detect this strategy accurately, CLAP must utilize the inter-packet context/relationship. It asserts if a given packet (1) bears a bad timestamp relative to previous packets, and (2) if the current TCP state is `SYN_RECV`. As

expected, in this case, Figure 5.2 shows that CLAP performs much better than Baseline #1 (¿ 35%).

Intra-packet context violation. Recall that the intra-packet context captures the relationships across different header fields in the same packet. Several evaluated attacks inject shadow packets in front of legal data packets. In these shadow packets, certain TCP/IP header field values are altered such that they are rejected by end point's rigorous TCP implementation but accepted by DPI's simplified version. This way, the data packets that follow the injected shadow packets are cloaked and evade DPI detection. For example, the *Invalid IP Version* attack from [93] injects packets with an incorrect IP Version header value (e.g 5 as there is no IPv5) to trigger the discrepancy between the DPI and endhost. CLAP detects this attack by learning the intra-packet context i.e., legitimate packets should not have a IP Version of 5; it is thus able to flag any packet that violates this requirement. The aggregate accuracies in Table 5.2 show that CLAP performs considerably better than Baseline #1.

5.4.4 Runtime Overhead Analysis

Finally, we evaluate how efficiently CLAP processes network traffic streams with our current implementation. Given that the training phase of CLAP is completely offline, the critical overhead would be its runtime overhead, i.e., the model processing efficiency of CLAP's testing phase (Stage (d)). For reference, we also measure the runtime model inference overhead of the open-source version released by [102], which is considered to be the state-of-the-art autoencoder-based IDS, and show that CLAP achieves a significantly

higher inference/processing speed. We use the default hyper-parameters to train the model from [102], as detailed in Table A.8.

Setup. We run the pipelines of both CLAP and [102] on a desktop with Intel Xeon E3-1225 Processor (3.2Ghz, 4 cores), 20GB RAM and disabled GPU support. We constrain both pipelines to only use one logical core for fair comparison. We note that [102] claims a higher processing throughput using a C++ implementation, but find that its released Python version [101] is much slower. Furthermore, it is fair to compare our Python-based prototype implementation with its Python-based baseline counterpart – we leave the task of re-implementing CLAP in more efficient languages (e.g. C/C++) for improving its overhead performance as future work. Our test adversarial traffic corpus consists of 92,262 packets and 6,424 connections.

Throughput. We compare the model processing throughputs of CLAP and [102] in Table 5.3. We see that CLAP outperforms [102] by a margin of $\approx 50\%$, while detecting DPI evasion attacks with much higher accuracies, as shown in previous subsections. However, we acknowledge that CLAP is not designed for detecting other network attacks, that are captured by [102]. We believe that the performance disparity is because [102] uses an ensemble (forest) of 10 small autoencoders to process subsets of different features pertaining to different attacks (details in Table A.8); it merges the separate reconstruction errors into an aggregate error with another autoencoder, making it overall a heavy procedure i.e., it cannot process packets as fast as CLAP (with only one autoencoder).

Model/ Metric	CLAP	[102]
Packets/Second	**2,162.2 (+49.7%)**	1,444.5
Connections/Second	**97.0 (+49.7%)**	64.8

Table 5.3: Model processing throughput

5.5 Discussion

Ethical Implications. In recent years, DPI evasion attacks have been considered as a means to bypass censorship systems [58, 93, 135, 136]. This is because, censors are essentially DPI middleboxes. We acknowledge that malicious DPI evasion attacks cannot be fully separated from censorship circumvention practices (since they rely on DPI evasion attacks). However, while CLAP can inherently enable censorship i.e., potentially expose censorship circumvention, we do not advocate censorship and our work is completely inspired by research questions.

Feature Completeness. Currently, CLAP covers all non-optional, non-tuple-related header fields. One might wonder if new evasion strategies that are outside our current feature set are viable. Since CLAP currently does not consider (1) uncommon IP and TCP option fields that are variable-sized (i.e., hard to encode as fix-sized features); and (2) payloads as they are unstructured data (i.e., with no fixed formats), we envision that these are the most likely surfaces that future emerging evasions can manipulate. To tackle this challenge, we plan to explore the remaining headers (TCP/IP options) and payload contents,

possibly via embedding techniques [138] that can map unstructured/variable-sized inputs to fix-sized vectors in future work.

Generalizability. In this work, we focus on evasions via attacks of TCP/IP protocols as these protocols are the most popular/common parts of most network protocol suites used today, and thus attract the vast majority of evasion efforts. However, we believe that the pipeline and core design of CLAP can be easily transferred and applied to other stateful protocols (e.g., HTTP) with minimum effort. As long as one can define the internal states inside the protocol implementation, CLAP is expected to learn packet contexts that help enable adversarial evasions.

Deployability. As shown in Section 5.4.4, the model inference time for CLAP is considerably faster than the prototype released by a state-of-the-art ML-based IDS, making CLAP potentially deployable on middleboxes on low-workload networks. We acknowledge that compared to signature-based general-purpose IDSs such as Zeek, our overall processing speed is considerably slower, but argue that (1) we target specifically DPI evasion attacks that cannot be captured by general-purpose IDS; (2) a re-implementation of CLAP in more efficient languages (e.g. C/C++) can greatly help reduce its overhead (this is left to future work); and (3) the rising trend of GPU-based ML acceleration can also be of great help in boosting the runtime performance of CLAP, which is again, part of our future plan.

We acknowledge that even though we have demonstrated that CLAP comes with very low error rates in both detection and localization tasks in Section 5.4, there can still be a number of false alarms that might be generated. However, we emphasize that CLAP can be tuned using a pre-defined adversarial score threshold (see Section 5.3.3), which would allow

the deployers of CLAP the flexibility to freely choose the desired trade-off between true positive and false positive rates. In addition, one can down sample the alarms generated by CLAP, and only selectively inspect/analyze those packets that have the highest associated adversarial scores. We believe these two strategies can significantly control the number/ratio of false alarms.

Attacker Adaptation. We are aware of the emerging research that exploits inherent vulnerabilities against ML models such that specifically crafted inputs, known as adversarial examples, can bypass ML models with minimum differences compared to its benign counterparts. We envision the possibility that attackers could generate such examples (packets) to hide from CLAP's detection. However, we point out that most of the adversarial example attacks from the adversarial ML community, consider end-to-end models so that it is rather easy to obtain the gradients from such end-to-end models to guide the generation of adversarial examples. In CLAP our design choices of using a multi-step pipeline (i.e., RNN + autoencoder) makes it extremely challenging, if possible at all.

5.6 Conclusions

In this paper, we design and implement a framework (called CLAP) for detecting DPI evasion attacks that have emerged recently. The key observation that drives the design of CLAP is that these attacks often violate either legitimate relationships between the headers within a packet, or relationships across headers in a train of packets. We construct a packet context-profile that captures these relationships and train a set of appropriate ML models to learn the legitimate (benign) distributions of such profiles. During test time,

CLAP checks if encountered packets conform with these distributions and flags those that do not as evasion attempts. Our comprehensive evaluations on a large variety of attacks from three different recent efforts show that CLAP (a) is extremely effective in detecting evasion patterns, and (b) significantly outperforms ML methods agnostic to packet context-profiles.

Chapter 6

Conclusion

6.1 Future Work

I discuss concrete future research directions in Chapter 2, 3 4 and 5 for each specific research thrust, respectively. In this section, I summarize some overarching future work routes to offer insights and suggestions towards endgame solutions.

Deep and Hybrid Adblocking. As shown previously, for identifying and resisting adversarial actions from the web advertising ecosystem, defenders need higher privileges to be able to intercept necessary information and carry out necessary countermeasures, before adversaries can intervene and manipulate. Since ads publishers and advertisers exist as JavaScript code and other web resources, their privilege level is limited to the web browser runtime. In this case, to be in the advantageous position, defenders need to go deeper into the browser core, and complete their instrumentation and modifications there. Fortunately, users with privacy and security concerns are motivated to allow such deep modifications to their web

browsers. In fact, this is well supported by the rapidly growing number of privacy-oriented (e.g., Brave and Firefox) browser (with deep instrumentation and modifications) users in recent years.

Besides going deeper, adopting hybrid solutions to identify ads and tracker resources is also a future research avenue to consider. I notice that both blocklist- and ML-based adblockers have their own pros and cons. For instance, blocklists are prone to human errors, but are considerably low-overhead to deploy; in comparison, ML models are strong in avoiding human mistakes, but vulnerable to carefully-crafted adversarial samples. Therefore, an intuitive improvement over using them individually is to combine these two approaches. For example, one can turn current binary (i.e., allowed/disallowed) blocklists into multi-class ones, so that they can be first used as quick filters that identify and remove ads only with very high confidence; then a potent ML model kicks in and classifies remaining web resources. Using this hybrid strategy, the entire pipeline of adblocking can be more effective, efficient and robust at the same time.

Regulatory Resolutions. Beyond technical solutions presented in this dissertation, I believe a greater space for improving the current web advertising ecosystem resides in regulatory and legislative efforts. Specifically, cyber privacy should be seen as one of the basic human rights and protected by laws, and mass web surveillance and tracking should be hence regulated in a stricter fashion. There is no doubt pursuing these regulatory resolutions requires considerable efforts and cooperation from multiple parties. I look forward to seeing more success in this space in the near future.

Automated DPI Patching. Even though my ML-based DPI evasion detector and localizer achieve sufficient accuracy performance, their runtime overhead is yet ideal for real-time applications that require ultra-low latency. Given this, I believe a more practical solution to address adversarial actions against DPI systems that suit real-time applications, for the time being, is through automated discovery and patching. Essentially, DPI code can be analyzed by program analysis and other techniques to discover ambiguities compared with endhost implementations; then, according to these ambiguities, code patches can be automatically generated. I am happy to have been participating in a series of works along this research vision [136, 137].

6.2 Concluding Remarks

This dissertation shows that, adversarial actions on the Internet launched to bypass deployed content blockers, are growingly pervasive and sophisticated. More importantly, through appropriate and efficient modeling and analyses (as demonstrated in this dissertation), these threats can be accurately detected and effectually prevented. The guiding principles for doing so are carefully designing differential tests to trigger conditional behaviours of adversarial actors, and analyzing minute, low-level and contextual interrelationships from objects in system traces to discern them; such analyses should be fully-automated, and are generally realized through learning signals and patterns from Internet data and code. For defending against these actions, it is needful to build platforms that conceal traces left by content blockers which adversarial actions target, so that these actions cannot be activated

anymore. I hope the vision and insights in this dissertation can be applied to and help other domains that also face the pressing threat of adversarial actions.

Bibliography

[1] A Publisher's 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] Google's 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] Mawi working group traffic archive. http://mawi.wide.ad.jp/mawi/. Accessed: 2020-06-26.

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

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

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

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

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

[16] AdBlock. https://getadblock.com/, 2017.

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

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

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

[20] EasyList. https://easylist-downloads.adblockplus.org/easylist.txt, 2017.

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

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

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

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

[25] mitmproxy. https://mitmproxy.org/, 2017.

[26] PageFair. https://pagefair.com/, 2017.

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

[28] The state of the blocked web 2017 Global Adblock Report. PageFair. https://pagefair.com/downloads/2017/01/PageFair-2017-Adblock-Report.pdf, 2017.

[29] Uk publishers lose nearly 3bn in revenue annually due to adblocking, 2017.

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

[31] Abp anti-circumvention filter list. https://github.com/abp-filters/abp-filters-anti-cv, 2018.

[32] Adblock warning removal list. https://easylist-downloads.adblockplus.org/antiadblockfilters.txt, 2018.

[33] Anti-adblock killer: Don't touch my adblocker! https://reek.github.io/anti-adblock-killer/, 2018.

[34] Blink - the chromium projects. https://www.chromium.org/blink, 2018.

[35] Easylist forum. https://forums.lanik.us/, 2018.

[36] Issues with yavli advertising. https://easylist.to/2015/08/19/issues-with-yavli-advertising.html, 2018.

[37] libadblockplus: A c++ library offering the core functionality of adblock plus. https://github.com/adblockplus/libadblockplus, 2018.

[38] Native advertising: A guide for businesses. https://www.ftc.gov/tips-advice/business-center/guidance/native-advertising-guide-businesses, 2018.

[39] Navigation timing api - web apis — mdn. https://developer.mozilla.org/en-US/docs/Web/API/Navigation_timing_API, 2018.

[40] Optimize performance under varying network conditions — tools for web developers — google developers. https://developers.google.com/web/tools/chrome-devtools/network-performance/network-conditions, 2018.

[41] Performancetiming.loadeventstart - web apis — mdn. https://developer.mozilla.org/en-US/docs/Web/API/PerformanceTiming/loadEventStart, 2018.

[42] Performancetiming.responsestart - web apis — mdn. https://developer.mozilla.org/en-US/docs/Web/API/PerformanceTiming/responseStart, 2018.

[43] Render-tree construction, layout, and paint. https://developers.google.com/web/fundamentals/performance/critical-rendering-path/render-tree-construction, 2018.

[44] Speed index - webpagetest documentation. https://sites.google.com/a/webpagetest.org/docs/using-webpagetest/metrics/speed-index, 2018.

[45] Subresourcefilter in chromium source code. https://cs.chromium.org/chromium/src/components/subresource_filter/, 2018.

[46] V8stacktraceimpl in chromium source code. https://cs.chromium.org/chromium/src/v8/src/inspector/v8-stack-trace-impl.h, 2018.

[47] visibility - css: Cascading style sheets — mdn. https://developer.mozilla.org/en-US/docs/Web/CSS/visibility, 2018.

[48] Web apis — mdn. https://developer.mozilla.org/en-US/docs/Web/API, 2018.

[49] Yavli filters issues - easylist forum. https://forums.lanik.us/viewtopic.php?f=64&t=36091, 2018.

[50] Blob - web apis — mdn. https://developer.mozilla.org/en-US/docs/Web/API/Blob, 2019.

[51] Media source extensions api - web apis — mdn. https://developer.mozilla.org/en-US/docs/Web/API/Media_Source_Extensions_API, 2019.

[52] preload.js in adblock plus extension that injects css selectors into web pages. https://github.com/adblockplus/adblockpluschrome/blob/c742bcc37b459c03bd564aea941ef6f05834e7fd/include.preload.js#L259, 2019.

[53] Zainul Abi Din, Panagiotis Tigas, Samuel T King, and Benjamin Livshits. {PERCIVAL}: Making in-browser perceptual ad blocking practical with deep learning. In *2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20)*, pages 387–400, 2020.

[54] Azeem Aqil, Karim Khalil, Ahmed OF Atya, Evangelos E Papalexakis, Srikanth V Krishnamurthy, Trent Jaeger, KK Ramakrishnan, Paul Yu, and Ananthram Swami. Jaal: Towards network intrusion detection at isp scale. In *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies*, pages 134–146, 2017.

[55] P Neira Ayuso. The conntrack-tools user manual, 2008.

[56] Muhammad Ahmad Bashir, Sajjad Arshad, Engin Kirda, William Robertson, and Christo Wilson. How tracking companies circumvented ad blockers using websockets. In *Proceedings of the Internet Measurement Conference 2018*, pages 471–477, 2018.

[57] Sruti Bhagavatula, Christopher Dunn, Chris Kanich, Minaxi Gupta, and Brian Ziebart. Leveraging machine learning to improve unwanted resource filtering. In *Proceedings of the 2014 Workshop on Artificial Intelligent and Security Workshop*, pages 95–102. ACM, 2014.

[58] Kevin Bock, George Hughey, Xiao Qiang, and Dave Levin. Geneva: Evolving censorship evasion strategies. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 2199–2214, 2019.

[59] Joan Bruna, Christian Szegedy, Ilya Sutskever, Ian Goodfellow, Wojciech Zaremba, Rob Fergus, and Dumitru Erhan. Intriguing properties of neural networks. 2013.

[60] Quan Chen and Alexandros Kapravelos. Mystique: Uncovering information leakage from browser extensions. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1687–1700. ACM, 2018.

[61] Kenjiro Cho. Recursive lattice search: hierarchical heavy hitters revisited. In *Proceedings of the 2017 Internet Measurement Conference*, pages 283–289, 2017.

[62] Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734, 2014.

[63] François Chollet et al. Keras. `https://keras.io`, 2015.

[64] Catalin Cimpanu. Ad network uses dga algorithm to bypass ad blockers and deploy in-browser miners. https://www.bleepingcomputer.com/news/security/ad-network-uses-dga-algorithm-to-bypass-ad-blockers-and-deploy-in-browser-miners/, 2018.

[65] Charlie Curtsinger, Benjamin Livshits, Benjamin Zorn, and Christian Seifert. ZOZZLE: Fast and Precise In-Browser JavaScript Malware Detection. In *USENIX Security Symposium*, 2011.

[66] 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.

[67] Yinpeng Dong, Fangzhou Liao, Tianyu Pang, Hang Su, Jun Zhu, Xiaolin Hu, and Jianguo Li. Boosting adversarial attacks with momentum. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 9185–9193, 2018.

[68] Steven Englehardt and Arvind Narayanan. Online tracking: A 1-million-site measurement and analysis. In *ACM Conference on Computer and Communications Security (CCS)*. ACM, 2016.

[69] Romain Fontugne, Patrice Abry, Kensuke Fukuda, Darryl Veitch, Kenjiro Cho, Pierre Borgnat, and Herwig Wendt. Scaling in internet traffic: a 14 year and 3 day longitudinal study, with multiscale analyses and random projections. *IEEE/ACM Transactions on Networking*, 25(4):2152–2165, 2017.

[70] Justin Gilmer, Nicolas Ford, Nicholas Carlini, and Ekin Cubuk. Adversarial examples are a natural consequence of test error in noise. In *International Conference on Machine Learning*, pages 2280–2289. PMLR, 2019.

[71] David Gugelmann, Markus Happe, Bernhard Ager, and Vincent Lenders. An automated approach for complementing ad blockers blacklists. *Proceedings on Privacy Enhancing Technology (PETS) 2015*, 2015.

[72] Saikat Guha, Bin Cheng, and Paul Francis. Privad: Practical privacy in online advertising. In *NSDI*, 2011.

[73] Dan Hendrycks and Thomas Dietterich. Benchmarking neural network robustness to common corruptions and perturbations. *Proceedings of the International Conference on Learning Representations*, 2019.

[74] Geng Hong, Zhemin Yang, Sen Yang, Lei Zhang, Yuhong Nan, Zhibo Zhang, Min Yang, Yuan Zhang, Zhiyun Qian, and Haixin Duan. How you get shot in the back: A systematical study about cryptojacking in the real world. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1701–1713. ACM, 2018.

[75] Xunchao Hu, Aravind Prakash, Jinghan Wang, Rundong Zhou, Yao Cheng, and Heng 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)*, September 2016.

[76] Muhammad Ikram, Hassan Jameel Asghar, Mohamed Ali Kaafar, Anirban Mahanti, and Balachander Krishanmurthy. Towards seamless tracking-free web: Improved detection of trackers via one-class learning. In *Proceedings on Privacy Enhancing Technology (PETS)*, 2017.

[77] Umar Iqbal, Zubair Shafiq, and Zhiyun Qian. The ad wars: Retrospective measurement and analysis of anti-adblock filter lists. In *Proceedings of the ACM Internet Measurement Conference (IMC)*, 2017.

[78] Umar Iqbal, Zubair Shafiq, and Zhiyun Qian. The ad wars: retrospective measurement and analysis of anti-adblock filter lists. In *Proceedings of the 2017 Internet Measurement Conference*, pages 171–183. ACM, 2017.

[79] Umar Iqbal, Zubair Shafiq, Peter Snyder, Shitong Zhu, Zhiyun Qian, and Benjamin Livshits. Adgraph: A machine learning approach to automatic and effective adblocking. *arXiv preprint arXiv:1805.09155*, 2018.

[80] Umar Iqbal, Peter Snyder, Shitong Zhu, Benjamin Livshits, Zhiyun Qian, and Zubair Shafiq. Adgraph: A graph-based approach to ad and tracker blocking. In *Proc. of IEEE Symposium on Security and Privacy*, 2020.

[81] Vishveshwar Jatain. Countering the revenue loss caused by ad blockers. `https://digitalcontentnext.org/blog/2020/08/12/countering-the-revenue-loss-caused-by-ad-blockers/`, 2020.

[82] Jia Jingping, Chen Kehua, Chen Jia, Zhou Dengwen, and Ma Wei. Detection and recognition of atomic evasions against network intrusion detection/prevention systems. *IEEE Access*, 7:87816–87826, 2019.

[83] Noah M. Johnson, Juan Caballero, Kevin Zhijie Chen, Stephen McCamant, Pongsin Poosankam, Daniel Reynaud, and Dawn Song. Differential slicing: Identifying causal

execution differences for security applications. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11, 2011.

[84] Alexandros Kapravelos, Yan Shoshitaishvili, Marco Cova, Christopher Kruegel, and Giovanni Vigna. Revolver: An Automated Approach to the Detection of Evasive Web-based Malware. In *USENIX Security Symposium*, 2013.

[85] Rezwana Karim, Frank Tip, Alena Sochurkova, and Koushik Sen. Platform-independent dynamic taint analysis for javascript. *IEEE Transactions on Software Engineering*, 2018.

[86] Kyungtae Kim, I Luk Kim, Chung Hwan Kim, Yonghwi Kwon, Yunhui Zheng, Xiangyu Zhang, and Dongyan Xu. J-force: Forced execution on javascript. In *Proceedings of the 26th International Conference on World Wide Web*, WWW '17, 2017.

[87] Bojan Kolosnjaji, Ambra Demontis, Battista Biggio, Davide Maiorca, Giorgio Giacinto, Claudia Eckert, and Fabio Roli. Adversarial malware binaries: Evading deep learning for malware detection in executables. In *2018 26th European signal processing conference (EUSIPCO)*, pages 533–537. IEEE, 2018.

[88] Christian Kreibich, Mark Handley, and V Paxson. Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics. In *Proc. USENIX Security Symposium*, volume 2001, 2001.

[89] Alexey Kurakin, Ian Goodfellow, and Samy Bengio. Adversarial machine learning at scale. *arXiv preprint arXiv:1611.01236*, 2016.

[90] Hieu Le, Athina Markopoulou, and Zubair Shafiq. CV-Inspector: Towards Automating Detection of Adblock Circumvention. In *The Network and Distributed System Security Symposium (NDSS)*, 2021.

[91] Adam Lerner, Anna Kornfeld Simpson, Tadayoshi Kohno, and Franziska 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.

[92] Bo Li, Phani Vadrevu, Kyu Hyung Lee, and Roberto Perdisci. Jsgraph: Enabling reconstruction of web attacks via efficient tracking of live in-browser javascript executions. In *25th Annual Network and Distributed System Security Symposium*, 2018.

[93] Fangfan Li, Abbas Razaghpanah, Arash Molavi Kakhki, Arian Akhavan Niaki, David Choffnes, Phillipa Gill, and Alan Mislove. lib• erate,(n) a library for exposing (traffic-classification) rules and avoiding them efficiently. In *Proceedings of the 2017 Internet Measurement Conference*, pages 128–141, 2017.

[94] Shasha Li, Shitong Zhu, Sudipta Paul, Amit Roy-Chowdhury, Chengyu Song, Srikanth Krishnamurthy, Ananthram Swami, and Kevin S Chan. Connecting the dots: Detecting adversarial perturbations using context inconsistency. In *European Conference on Computer Vision*, pages 396–413. Springer, 2020.

[95] Yong Liu, Ruiping Wang, Shiguang Shan, and Xilin Chen. Structure inference net: Object detection using scene-level context and instance-level relationships. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 6985–6994, 2018.

[96] Matthew Malloy, Mark McNamara, Aaron Cahn, and Paul Barford. Ad blockers: Global prevalence and impact. In *ACM Internet Measurement Conference (IMC)*, 2016.

[97] Gonzalo Marín, Pedro Casas, and Germán Capdehourat. Rawpower: Deep learning based anomaly detection from raw network traffic measurements. In *Proceedings of the ACM SIGCOMM 2018 Conference on Posters and Demos*, pages 75–77, 2018.

[98] Gonzalo Marín, Pedro Casas, and Germán Capdehourat. Deepsec meets rawpower-deep learning for detection of network attacks using raw representations. *ACM SIGMETRICS Performance Evaluation Review*, 46(3):147–150, 2019.

[99] William Mendenhall, Robert J Beaver, and Barbara M Beaver. *Introduction to probability and statistics*. Cengage Learning, 2012.

[100] Hassan Metwalley, Stefano Traverso, Marco Mellia, Stanislav Miskovic, and Mario Baldi. The Online Tracking Horde: A View from Passive Measurements. In *Traffic Monitoring and Analysis*. 2015.

[101] Yisroel Mirsky. Kitsune-py. `https://github.com/ymirsky/Kitsune-py`, 2020.

[102] Yisroel Mirsky, Tomer Doitshman, Yuval Elovici, and Asaf Shabtai. Kitsune: An ensemble of autoencoders for online network intrusion detection. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society, 2018.

[103] Muhammad Haris Mughees, Zhiyun Qian, and Zubair Shafiq. Detecting anti ad-blockers in the wild. *Proceedings on Privacy Enhancing Technologies*, 2017(3):130–146, 2017.

[104] Muhammad Haris Mughees, Zhiyun Qian, and Zubair Shafiq. A first look at ad-block detection: A new arms race on the web. *Proceedings on Privacy Enhancing Technology (PETS)*, 2017.

[105] Arvind Narayanan and Dillon Reisman. The princeton web transparency and accountability project.

[106] Mozilla Developer Network. Concurrency model and Event Loop. `https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop`, 2017.

[107] Nick Nikiforakis, Wouter Joosen, and Benjamin Livshits. Privaricator: Deceiving fingerprinters with little white lies. In *Proceedings of the 24th International Conference on World Wide Web*, pages 820–830. International World Wide Web Conferences Steering Committee, 2015.

[108] Rishab Nithyanand, Sheharbano Khattak, Mobin Javed, Narseo Vallina-Rodriguez, Marjan Falahrastegar, Julia E. Powles, Emiliano De Cristofaro, Hamed Haddadi, and Steven 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, 2016. USENIX Association.

[109] Xiang Pan, Yinzhi Cao, and Yan Chen. I do not know what you visited last summer: Protecting users from third-party web tracking with trackingfree browser. In

Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA, 2015.

[110] Nicolas Papernot, Patrick McDaniel, and Ian Goodfellow. Transferability in machine learning: from phenomena to black-box attacks using adversarial samples. *arXiv preprint arXiv:1605.07277*, 2016.

[111] Li Pengcheng, Jinfeng Yi, and Lijun Zhang. Query-efficient black-box attack by active learning. In *2018 IEEE International Conference on Data Mining (ICDM)*, pages 1200–1205. IEEE, 2018.

[112] Fabio Pierazzi, Feargus Pendlebury, Jacopo Cortellazzi, and Lorenzo Cavallaro. Intriguing properties of adversarial ml attacks in the problem space. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1332–1349. IEEE, 2020.

[113] Alexander Pons, Andrew De La Rosa, Silvia Vidaurre, Luis Vargas, and Eugene Pons. Security and privacy implications of'do not track'. *International Journal of Information Privacy, Security and Integrity*, 3(2):117–133, 2017.

[114] Jon Postel. Rfc0791: Internet protocol, 1981.

[115] J. Ross Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986.

[116] Anantha Ramaiah, R Stewart, and Mitesh Dalal. Improving tcp's robustness to blind in-window attacks. *Internet Engineering Task Force, RFC*, 5961, 2010.

[117] Jonas Rauber, Wieland Brendel, and Matthias Bethge. Foolbox: A python toolbox to benchmark the robustness of machine learning models. *arXiv preprint arXiv:1707.04131*, 2017.

[118] Jonas Rauber, Roland Zimmermann, Matthias Bethge, and Wieland Brendel. Foolbox native: Fast adversarial attacks to benchmark the robustness of machine learning models in pytorch, tensorflow, and jax. *Journal of Open Source Software*, 5(53):2607, 2020.

[119] IETF RFC793. Transmission control protocol. *J. Postel. September*, 981, 1981.

[120] Leonard Richardson. Beautiful soup documentation. *April*, 2007.

[121] Konrad Rieck, Tammo Krueger, and Andreas Dewald. Cujo: Efficient detection and prevention of drive-by-download attacks. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, 2010.

[122] Franziska Roesner, Tadayoshi Kohno, and David Wetherall. Detecting and Defending Against Third-Party Tracking on the Web. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

[123] Ethan Rudd, Andras Rozsa, Manuel Gunther, and Terrance Boult. A survey of stealth malware: Attacks, mitigation measures, and steps toward autonomous open world solutions. *IEEE Communications Surveys & Tutorials*, 19(2):1145–1172, 2017.

[124] Alexander Sjösten, Peter Snyder, Antonio Pastor, Panagiotis Papadopoulos, and Benjamin Livshits. Filter list generation for underserved regions. In *Proceedings of The Web Conference 2020*, pages 1682–1692, 2020.

[125] Aditya K Sood and Richard J Enbody. Malvertising–exploiting web advertising. *Computer Fraud & Security*, 2011(4):11–16, 2011.

[126] 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.

[127] Grant Storey, Dillon Reisman, Jonathan Mayer, and Arvind Narayanan. The future of ad blocking: An analytical framework and new techniques. *Technical Report*, 2017.

[128] Grant Storey, Dillon Reisman, Jonathan Mayer, and Arvind Narayanan. The future of ad blocking: An analytical framework and new techniques. *arXiv preprint arXiv:1705.08568*, 2017.

[129] Kaihua Tang, Hanwang Zhang, Baoyuan Wu, Wenhan Luo, and Wei Liu. Learning to compose dynamic tree structures for visual contexts. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 6619–6628, 2019.

[130] The PageFair Team. 2017 Adblocking Report. `https://pagefair.com/blog/2017/adblockreport/`, 2017.

[131] Florian Tramèr, Pascal Dupré, Gili Rusak, Giancarlo Pellegrino, and Dan Boneh. Adversarial: Perceptual ad blocking meets adversarial machine learning. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 2005–2021, 2019.

[132] Tobias Urban, Martin Degeling, Thorsten Holz, and Norbert Pohlmann. Beyond the front page: Measuring third party dynamics in the field. In *Proceedings of The Web Conference 2020*, pages 1275–1286, 2020.

[133] Weihang Wang, Yunhui Zheng, Xinyu Xing, Yonghwi Kwon, Xiangyu Zhang, and Patrick 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*, FSE 2016, 2016.

[134] Weihang Wang, Yunhui Zheng, Xinyu Xing, Yonghwi Kwon, Xiangyu Zhang, and Patrick 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*, pages 205–216. ACM, 2016.

[135] Zhongjie Wang, Yue Cao, Zhiyun Qian, Chengyu Song, and Srikanth V Krishnamurthy. Your state is not mine: a closer look at evading stateful internet censorship. In *Proceedings of the 2017 Internet Measurement Conference*, pages 114–127, 2017.

[136] Zhongjie Wang, Shitong Zhu, Yue Cao, Zhiyun Qian, Chengyu Song, Srikanth V Krishnamurthy, Kevin S Chan, and Tracy D Braun. Symtcp: Eluding stateful deep packet inspection with automated discrepancy discovery. In *Network and Distributed System Security Symposium (NDSS)*, 2020.

[137] Zhongjie Wang, Shitong Zhu, Keyu Man, Pengxiong Zhu, Yu Hao, Zhiyun Qian, Srikanth V Krishnamurthy, Tom La Porta, and Michael J De Lucia. Themis:

Ambiguity-aware network intrusion detection based on symbolic model comparison. In *CCS*, 2021.

[138] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 2020.

[139] Bin Xin, William N. Sumner, and Xiangyu Zhang. Efficient program execution indexing. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, 2008.

[140] Chengcheng Xu, Shuhui Chen, Jinshu Su, Siu-Ming Yiu, and Lucas CK Hui. A survey on regular expression matching for deep packet inspection: Applications, algorithms, and hardware platforms. *IEEE Communications Surveys & Tutorials*, 18(4):2991–3029, 2016.

[141] Noam Yefet, Uri Alon, and Eran Yahav. Adversarial examples for models of code. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–30, 2020.

[142] Dachuan Yu, Ajay Chander, Nayeem Islam, and Igor Serikov. Javascript instrumentation for browser security. In *ACM SIGPLAN Notices*, volume 42, pages 237–249. ACM, 2007.

[143] Apostolis Zarras, Alexandros Kapravelos, Gianluca Stringhini, Thorsten Holz, Christopher Kruegel, and Giovanni Vigna. The Dark Alleys of Madison Avenue: Understanding Malicious Advertisements. In *ACM Internet Measurement Conference (IMC)*, 2014.

[144] Huangzhao Zhang, Zhuo Li, Ge Li, Lei Ma, Yang Liu, and Zhi Jin. Generating adversarial examples for holding robustness of source code processing models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 1169–1176, 2020.

[145] Chong Zhou and Randy C Paffenroth. Anomaly detection with robust deep autoencoders. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 665–674, 2017.

[146] Shitong Zhu, Xunchao Hu, Zhiyun Qian, Zubair Shafiq, and Heng Yin. Measuring and disrupting anti-adblockers using differential execution analysis. In *The Network and Distributed System Security Symposium (NDSS)*, 2018.

[147] Shitong Zhu, Umar Iqbal, Zhongjie Wang, Zhiyun Qian, Zubair Shafiq, and Weiteng Chen. Shadowblock: A lightweight and stealthy adblocking browser. In *The World Wide Web Conference*, pages 2483–2493, 2019.

[148] Shitong Zhu, Shasha Li, Zhongjie Wang, Xun Chen, Zhiyun Qian, Srikanth V Krishnamurthy, Kevin S Chan, and Ananthram Swami. You do (not) belong here: detecting dpi evasion attacks with context learning. In *Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies*, pages 183–197, 2020.

[149] Shitong Zhu, Zhongjie Wang, Xun Chen, Shasha Li, Keyu Man, Umar Iqbal, Zhiyun Qian, Kevin S Chan, Srikanth V Krishnamurthy, Zubair Shafiq, et al. Eluding ml-

based adblockers with actionable adversarial examples. In *Annual Computer Security Applications Conference*, 2021.

Appendix A

Appendix

A.1 Disguising DOM Perturbations for Chapter 4

As discussed in §4.3, A^4 can disguise its DOM manipulations to avoid being detected by simple rule-based checks and perceived by users. Specifically, we propose the following strategies to obfuscate and conceal inserted DOM nodes.

- **Randomized node types**: Inserted nodes can be of any DOM element type that supports the `visibility` property (e.g., `<p>`, `<div>`, `<table>`).
- **Randomized node properties**: Inserted nodes can have random property keys and values that do not conflict with functional ones (e.g., `rand_prop_Aft4A: "rand_value_SzJd2"`). This also includes random node text because inserted nodes are set to be invisible.
- **Randomized local node placement**: Besides the two mapping-back strategies in §4.3, inserted nodes can be organized arbitrarily inside each insertion site (e.g., for

centralized strategy, inserted nodes can be cascaded in randomized topologies as a sub-tree and then attached to the insertion site).

Note that the first and second DOM perturbation strategies above can also be written and hidden into existing CSS style sheets (via `class` or `id` selectors) so that simple scans over inserted DOM nodes themselves would not raise suspicion.

A.2 Additional Tables and Figures for Chapter 4

```
1 <script async="" src="https://
    publisher.com/adscript.js
    ></script>
```

Code A.1: Original DOM snippet

```
1 <script async="" src="https://
    publisher.com/adscript.js></
    script>
2 <p hidden="" rand_prop_1="1">
    RANDOM TEXT 1</p>
3 <p hidden="" rand_prop_2="2">
    RANDOM TEXT 2</p>
4 <p hidden="" rand_prop_3="3">
    RANDOM TEXT 3</p>
```

Code A.2: Perturbed DOM snippet

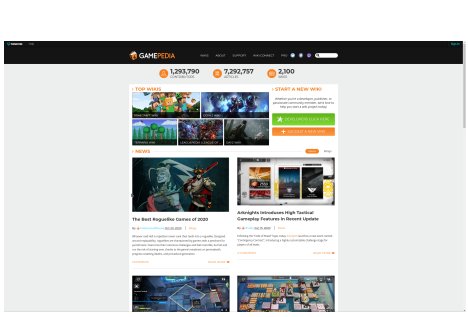
Figure A.1: Example DOM snippet with structural perturbations (inserted invisible sibling nodes)

# trees	100
Split criterion	entropy
Maximum tree depth	unlimited
Precision	0.87
Recall	0.88
Accuracy	0.93

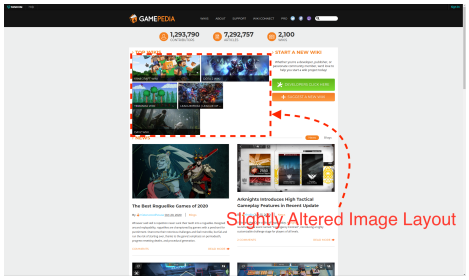
Table A.1: Reproduced target RF's hyper-parameters and accuracy metrics

# hidden layers	3
# neurons	(1024, 512, 128)
# epochs	30
Dropout rate	0.1
Accuracy (agreement rate)	0.90

Table A.2: Local surrogate NN's hyper-parameters and agreement rate



(a) Unperturbed `gamepedia.com`



(b) Perturbed `gamepedia.com` (w/ minor breakage)

Figure A.2: Example webpage with minor breakage

A.3 MAWI Traffic Dataset Statistics Chapter 5

As described in Section 5.4.1, here we list the basic statistics of the dataset used in our evaluations in Table A.6.

A.4 RNN Prediction Accuracy for Chapter 5

We present the per-label accuracy breakdown along with the number of samples (packets) in Table A.7. Overall, the RNN model in CLAP achieves an accuracy of 0.995 on its testing set.

A.5 Feature Set for Chapter 5

Here we list the features we used in building context profiles in Table A.9. As previously described, only TCP and IP layer header features (i.e., #1-#32) are used for training the RNN model.

A.6 Per-context Categorization of Evasion Strategies for Chapter 5

We categorize the 73 evasion strategies that were evaluated in Section 5.4.2 in Table A.10 according to the context they primarily violate, using the categorization scheme proposed previously.

A.7 Model Hyper-parameters for Chapter 5

We list hyper-parameters used in training the RNN/autoencooder models in CLAP, and the autoencoder models used in Baseline #1 and #2 in evaluations, in Table A.8.

# successfully crawled records (distinct requests)	503,526
# successfully crawled websites (distinct domains)	8,121
# features before one-hot encoding	65
# features after one-hot encoding	312
# categorical features	5
# binary features	36
# numeric features	25
# records to perturb (distinct requests)	2,000

Table A.3: Dataset statistics

Rank	**Feature No. # (from Table A.5)**	**Category**
1	19	Structural
2	3	URL
3	5	URL
4	7	URL
5	4	URL

Table A.4: Information gain ranking of top-5 perturbed features

No. #	Meaning	Type	Category	No. #	Meaning	Type	Category
1	Total number of nodes in the graph at the time of classification	I	S	10	Presence of ad keyword attributes in ascendant nodes	B	S
2	Total number of edges in the graph at the time of classification	I	S	11	Presence of screen dimension keywords in query string	B	U
3	Length of request URL	I	U	12	Presence of ad dimension keywords in full URL	B	U
4	Presence of ad keywords	B	U	13	Number of siblings of current node	I	S
5	Presence of special characters	B	U	14	Number of siblings of parent node	I	S
6	Presence of semicolons	B	U	15	Presence of siblings of parent node with ad keyword attributes	B	S
7	Presence of base domain in query string	B	U	16	Number of inbound connections of parent node	I	S
8	Presence of ad dimension keywords in query string	B	U	17	Number of outbound connections of parent node	I	S
9	Number of inbound and outbound connections	I	S	18	Number of inbound and outbound connections of parent node	I	S
				19	Average degree of connectivity for all nodes in current page	F	S

Table A.5: List of perturbed features; **Type** includes – I: integer, B: binary, F: float; **Category** includes – S: structural, U: URL

Original Trafffc Archive			
Capture Timestamp	04/07/2020 14:00 JPT	# Packets	111,851,572
# TCP/IPv4 Packets	51,692,562	/	
Sampled (Used) Dataset			
# TCP/IPv4 Packets	540,353	# TCP/IPv4 Connections	37,622
# TCP/IPv4 Packets (Training)	448,091	TCP/IPv4 Connections (Training)	31,198
# TCP/IPv4 Packets (Testing)	92,262	# TCP/IPv4 Connections (Testing)	6,424

Table A.6: Statistics of used MAWI dataset

TCP State/ Packet Window Classification	SYN_SENT	SYN_RECV	ESTABLISHED	FIN_WAIT	CLOSE_WAIT	LAST_ACK	TIME_WAIT	CLOSE
In-Window	0.999678 (6166)	1.0 (18906)	0.994733 (47664)	0.996628 (2966)	0.988205 (3117)	0.993641 (2988)	0.992513 (2805)	0.987467 (2314)
Out-of-Window	0.0 (2)	/	0.953246 (1155)	0.911764 (34)	0.694444 (36)	0.814818 (27)	0.95 (20)	0.956521 (23)

Table A.7: Per-label breakdown of RNN accuracy

	RNN (GRU-based) in CLAP				Autoencoder in CLAP			
Model Parameter	# Layer(s)	Input Size	Hidden Size (Gate Size)	# Epochs	# Layer(s)	Input Size	Length of Context Profile Stacking	Bottleneck Layer Size	# Epochs
Value	1	32	32	30	7	345	3	40	1,000
	Autoencoder in Baseline #1				Ensembled Autoencoders in Baseline #2				
Model Parameter	# Layer(s)	Input Size	Bottleneck Layer Size	# Epochs	# Layer(s)	Ensemble Size (# Autoencoders)	Total Input Size	Average Input Size (Per Autoencoder)	Average Bottleneck Layer Size	# Epochs
Value	3	51	5	1,000	1	16	100	6.25	4.68	1

Table A.8: Hyper-parameters used in the paper

Index	Type	Semantic	Index	Type	Semantic	Index	Type	Semantic	Index	Type	Semantic
TCP Layer Features			17	Integer	Payload Length	**IP Layer Features**			**Amplification Features (not included for training RNN)**		
1	Binary	Packet direction	18	Integer	Option: Maximum Segment Size	26	Integer	Length	33-45	Binary	Out-of-Range indicators for numeric TCP features
2	Integer	SEQ number (incremental)	19	Integer	Option: Timestamp Value (TSVal)	27	Integer	Time-To-Live	46-50	Binary	Out-of-Range indicators for numeric IP features
3	Integer	ACK number (incremental)	20	Integer	Option: Timestamp Echo Reply (TSecr)	28	Integer	Header Length	51	Binary	TCP Payload Length correctness (#17 = #26 - #28 - #4)
4	Integer	Data Offset	21	Integer	Option: Window Scale	29	Binary	Checksum validity	**Gate Weights from GRU**		
5-13	Categorical	Flags (one-hot encoded)	22	Integer	Option: User Timeout	30	Integer	IP Version	52-83	Float	Update Gates
14	Integer	Window Size	23	Binary	Option: MD5 Header Validity	31	Integer	Type of Service	84-115	Float	Reset Gates
15	Binary	Checksum vadility	24	Integer	TCP Timestamp	32	Binary	Existence of non-standard IP options			
16	Integer	Urgent Pointer	25	Integer	Frame Timestamp						

Table A.9: List of features in context profile

From	Strategy Name	From	Strategy Name	From	Strategy Name
	Inter-packet Context Violation		Zeek: Data Packet (ACK) Bad SEQ		Bad SEQ (Min)
	GFW: Data Packet (ACK) Bad TCP-Checksum/MD5-Option		GFW: Injected FIN-ACK Bad TCP-Checksum/MD5-Option		Data Packet wo/ ACK Flag (Max)
	GFW: Data Packet (ACK) wo/ ACK Flag		Snort: Injected FIN-ACK Bad TCP MD5-Option	[93]	Data Packet wo/ ACK Flag (Min)
	Zeek: Data Packet (ACK) wo/ ACK Flag		GFW: Injected RST Bad TCP-Checksum/MD5-Option		Invalid Data-Offset (Max)
[136]	Zeek: Data Packet (ACK) Bad ACK Num	[136]	Snort: Injected RST Pure		Invalid Data-Offset (Min)
	Zeek: Data Packet (ACK) Overlapping		Snort: Injected RST Partial In-Window		Invalid Flags (Max)
	GFW: Injected FIN-ACK Bad ACK Num		Snort: Injected RST Bad TCP MD5-Option		Invalid Flags (Min)
	Snort: Injected FIN-ACK Bad ACK Num		GFW: Injected FIN w/ Payload		Bad TCP Checksum (Max)
	GFW: Injected RST Bad Timestamp		Snort: Injected FIN Pure		Bad SEQ (Max)
	Snort: Injected RST Bad Timestamp		Zeek: Injected FIN Pure		Bad SEQ (Min)
	Zeek: SYN w/ Payload		GFW #1: SYN w/ Payload & Bad SEQ		Invalid Data-Offset Bad TCP Checksum
	GFW: Injected RST-ACK Bad ACK Num		GFW #2: SYN w/ Payload & Bad SEQ		Invalid Data-Offset Low TTL
	Bad IP Length (Too Long) (Min)		Snort: SYN Multiple (SYN)		Invalid Data-Offset Bad ACK Num
[93]	Low TTL (Max)	[93]	Zeek: SYN Multiple (SYN)	[58]	Injected RST Bad IP Length
	Low TTL (Min)		Snort: Injected RST-ACK Bad ACK Num		Injected RST Bad TCP Checksum
	RST w/ Low TTL #1 (Max)		Zeek: Injected RST/FIN-ACK Bad SEQ		Bad TCP MD5-Option Injected RST
	RST w/ Low TTL #1 (Min)		Invalid IP Header Length (Max)		Invalid Flags #1 Bad TCP Checksum
	RST w/ Low TTL #2 (Max)		Invalid IP Header Length (Min)		Invalid Flags #2 Low TTL
	RST w/ Low TTL #2 (Min)		Invalid IP Version (Min)		Invalid Flags #2 Bad TCP MD5-Option
	Bad IP Length (Too Short) (Min)		Bad IP Length (Too Long) (Max)		Bad TCP UTO-Option Bad TCP MD5-Option
	Bad TCP Checksum (Min)		Bad IP Length (Too Short) (Max)		Invalid TCP WScale-Option Invalid Data-Offset
[58]	Injected RST Low TTL		Data Packet wo/ ACK Flag (Max)		Bad Payload Length Bad TCP Checksum
	Injected RST-ACK Bad TCP Checksum		Data Packet wo/ ACK Flag (Min)		Bad Payload Length Low TTL
	Injected RST-ACK Low TTL		Invalid Data-Offset (Max)		Bad Payload Length Bad ACK Num
	Injected SYN-ACK Bad TCP MD5-Option		Invalid Data-Offset (Min)		/ Bad Payload Length
	Intra-packet Context Violation		Invalid Flags (Max)		Bad IP Length /
	GFW: Data Packet (ACK) Underflow SEQ	[136]	Invalid Flags (Min)		
[136]	Zeek: Data Packet (ACK) Underflow SEQ		Bad TCP Checksum (Max)		
	Snort: Data Packet (ACK) w/ Urgent Pointer		Bad SEQ (Max)		

Table A.10: Per-context categorization of evasion strategies from [58, 93, 136] (with $TH_{inter} = 0.15$)