Eluding ML-based Adblockers With Actionable Adversarial Examples

Shitong Zhu[†], Zhongjie Wang[†], Xun Chen[¶], Shasha Li[†], Keyu Man[†],
Umar Iqbal[$], Zhiyun Qian[†], Kevin S. Chan[#], Srikanth V. Krishnamurthy[†],
Zubair Shafiq[$], Yu Hao[†], Guoren Li[†], Zheng Zhang[†], Xiaochen Zou[†]

[†]University of California, Riverside, [¶]Samsung Research America, [$]University of Iowa,
[#]US Army Research Laboratory, [$]University of California, Davis

ABSTRACT

Online advertisers have been quite successful in circumventing traditional adblockers that rely on manually curated rules to detect ads. As a result, adblockers have started to use machine learning (ML) classifiers for more robust detection and blocking of ads. Among these, AdGraph which leverages rich contextual information to classify ads, is arguably, the state of the art ML-based adblocker. In this paper, we present A^4, a tool that intelligently crafts adversarial ads to evade AdGraph. Unlike traditional adversarial examples in the computer vision domain that can perturb any pixels (i.e., unconstrained), adversarial ads generated by A^4 are *actionable* in the sense that they preserve the application semantics of the web page. Through a series of experiments we show that A^4 can bypass AdGraph about 81% of the time, which surpasses the state-of-the-art attack by a significant margin of 145.5%, with an overhead of <20% and perturbations that are visually imperceptible in the rendered webpage. We envision that A^4's framework can be used to potentially launch adversarial attacks against other ML-based web applications.

KEYWORDS

adversarial examples, machine learning, adblockers

ACM Reference Format:
Shitong Zhu[†], Zhongjie Wang[†], Xun Chen[¶], Shasha Li[†], Keyu Man[†],, Umar Iqbal[$], Zhiyun Qian[†], Kevin S. Chan[#], Srikanth V. Krishnamurthy[†],, Zubair Shafiq[$], Yu Hao[†], Guoren Li[†], Zheng Zhang[†], Xiaochen Zou[†]. 2021. Eluding ML-based Adblockers With Actionable Adversarial Examples. In *Annual Computer Security Applications Conference (ACSAC '21), December 6–10, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3485832.3488008

1 INTRODUCTION

As adblockers have gained popularity in recent years [18], online advertisers have started fighting back. Specifically, many techniques have emerged to circumvent the current generation of adblockers. Notably, prior work [33] 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) [10, 17].

Given these limitations, several ML-based adblockers have recently emerged with the goal of improving the effectiveness and accuracy over signature-based adblockers [1, 11, 27] . Such adblockers can be categorized into "perceptual" and "non-perceptual" classes. Perceptual adblockers [1, 27] 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) [28] 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] [11]. 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. Furthermore, the use of non-visual features inhibit the applicability of traditional adversarial attack techniques from the unconstrained domain [28].

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

[1] [26] 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.

ACSAC '21, December 6–10, 2021, Virtual Event, USA
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8579-4/21/12.
https://doi.org/10.1145/3485832.3488008

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) [14], 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).

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 [3]. 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 [11], 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 §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 §3).

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) [4], 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 [7] 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 [14]. Essentially, PGD performs FGSM multiple times with a smaller step-size, or α. Formally, the search procedure can be expressed as:

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

where, for a given a given input vector A,

$$Clip(A_i, \epsilon_{min}, \epsilon_{max}) = \begin{cases} \epsilon_{min}, & \text{if } A_i < \epsilon_{min} \\ \epsilon_{max}, & \text{if } A_i > \epsilon_{max} \\ A_i, & otherwise. \end{cases} \tag{2}$$

A^4 is inspired by the iterative philosophy used in PGD, and extends its simple clipping mechanism to an extensive feedback loop (§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.

3.1 Threat Model

Before diving into the details of A^4's algorithm, we first define our threat model. As mentioned in §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 1. As shown, the objective of A^4 is to 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 §2, because even though AdGraph has been open-sourced, it is easy for other ML-based adblockers to hide their model internals.

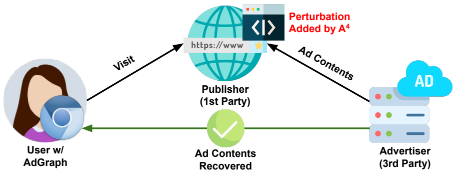


Figure 1: Different participants in A^4's threat model

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 §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.

3.2 Overview

Optimization problem formulation. Formally, consider the optimization problem:

$$\begin{aligned} \underset{x_{adv}}{\text{minimize}} \quad & \text{Dist}(x_{adv} - x_{input}) \\ \text{subject to} \quad & P_{model}(x_{adv}) \neq P_{model}(x_{input}), \\ & x_{adv} \in \mathcal{H}_{feature-space}, \\ & x_{adv} \in \mathcal{H}_{application-space}, \end{aligned} \tag{3}$$

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 §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 3 due to computational inefficiency [4], especially in presence of complex target models (e.g., neural networks). Therefore, as discussed in §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 3 does not have an analytic solution [4], 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: Actionable Ad Adversarial Attack

```
Input  : target model $M$, ad request $x_{input}$, maximum iterations
         $max\_iter$, maximum perturbation magnitude $\epsilon$
Output : actionable adversarial example $x_{adv}$
success ⟵ False
curr_iter ⟵ 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($M, x_{curr}, \epsilon$)
    $x_{curr} \longleftarrow x_{input} + pert_{curr\_iter}$
    $x_{curr} \longleftarrow$ EnforceFeatureSpaceConstraints($x_{curr}$)
    $page_{perturbed} \longleftarrow$ MapBackToWebPage($x_{curr} - x_{input}$)
    $x_{curr} \longleftarrow$ ExtractFeatureValues($page_{perturbed}$)
    $success \longleftarrow$ VerifyIfAdversarialOnTargetModel($x_{curr}$)
end
return $x_{curr}$
```

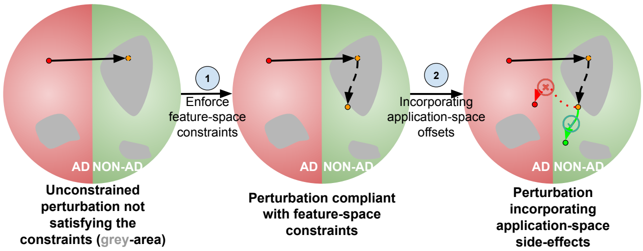

AD
NON-AD
1
Enforce feature-space constraints
2
Incorporating application-space offsets
Unconstrained perturbation not satisfying the constraints (grey-area)
Perturbation compliant with feature-space constraints
Perturbation incorporating application-space side-effects

Figure 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.

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). 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 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 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 3. Prior research [19] has shown that this so-called inter-model transferability exists in almost all modern ML models (including non-differentiable ones such as Random Forest).

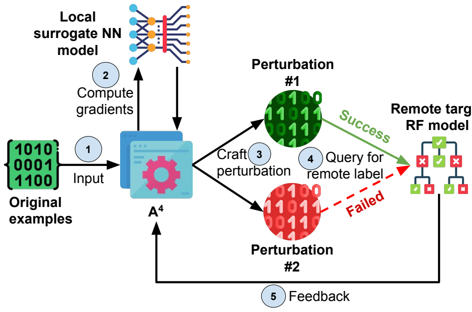


Figure 3: Transfer-based attack paradigm; steps ③, ④, and ⑤ are executed in a loop.

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 §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 10 in the appendix shows their semantics, data types (i.e., integer, binary or float) and categories (i.e., structural and URL).

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:

$$x_{proj} = \begin{cases} \max(\min(1, x_{pert}), 0), \text{ if } x_{pert} \in S_{numerical} \\ 0, \text{ if } \|x_{pert} - 0\| \leq \|x_{pert} - 1\| \\ \quad \text{and } x_{pert} \in S_{binary} \\ 1, \text{ if } \|x_{pert} - 0\| > \|x_{pert} - 1\| \\ \quad \text{and } x_{pert} \in S_{binary} \end{cases} \tag{4}$$

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 10), 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 8 (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 10), 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 §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 [6, 15]. 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 [12]), 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 5. The examples generated by A^4 are bounded by this norm.

$$L_{custom\_inf}(pert) = \max(|pert_i| : i = 1, 2, \ldots, n \text{ if } pert_i \in S_{numerical}) \quad (5)$$

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 6.

$$Clip'_{global\_local\_mixin}(pert, \epsilon_g, \epsilon_l) = \begin{cases} Clip(pert, 0, \epsilon_g \cdot r_i), & \text{if } \epsilon_g \cdot r_i < \epsilon_l \cdot pert \\ Clip(pert, 0, \epsilon_l \cdot pert), & otherwise \end{cases} \quad (6)$$

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 2. As shown in §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.

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.

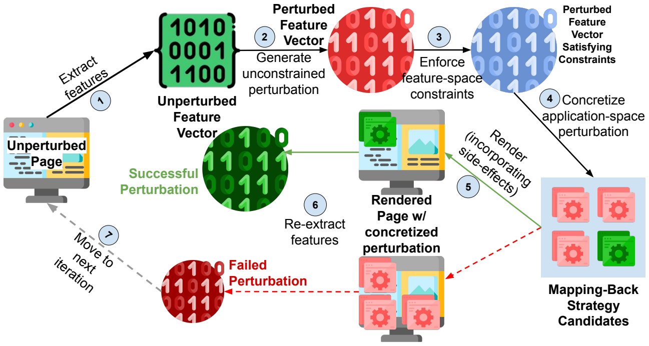

Unperturbed Page
1 Extract features
1010
0001
1100
Unperturbed Feature Vector
2 Generate unconstrained perturbation
Perturbed Feature Vector
3 Enforce feature-space constraints
Perturbed Feature Vector Satisfying Constraints
4 Concretize application-space perturbation
Mapping-Back Strategy Candidates
5 Render (incorporating side-effects)
Rendered Page w/ concretized perturbation
6 Re-extract features
Successful Perturbation
Failed Perturbation
7 Move to next iteration

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

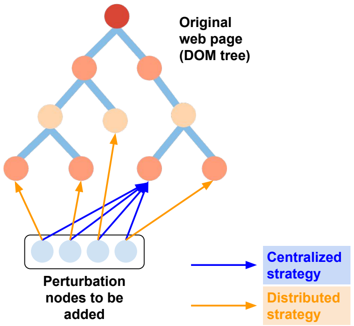


Figure 5: Different mapping-back strategies on adding perturbation nodes

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.

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.

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). 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 (*centralized* strategy), or as the children of multiple existing nodes (*distributed* strategy), as shown in Figure 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 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 2).

4 IMPLEMENTATION

We use the library `Foolbox` [23, 24] to compute the adversarial perturbations numerically. Per our design in §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. For step ④, ⑤ and ⑥ in Figure 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` [25]. As will be discussed in §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].

4.1 Model Training

We need to reproduce the classifier used in [11] following its revealed hyper-parameters, on the newly collected dataset, since [11] 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 6 in the appendix. As shown, the accuracy metrics with our reproduced RF model are close enough to the ones reported in [11], which validates our replication effort.

We then need a local surrogate model that is differentiable (recall §3), to drive our gradient-based attack. To this end, we use the Python-based deep learning library `Keras` [5] to train a 3-layer Neural Network (NN) as the surrogate. NN is considered to have the best model capacity [19] for imitating the decision boundaries of other models. The hyper-parameters and accuracy metrics of this NN are in Table 7 in the appendix. Note that in order to best mimic the remote decision boundary, we use the dataset that trains

[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.

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 7 represents the agreement rate between two models.

4.2 Active Learning

Given that A^4 trains a local surrogate model to imitate the decision boundary of the target classifier in [11] and provide necessary gradients. Even though in §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 [20]. 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.

4.3 Hyper-parameters

Table 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 §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 §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 2)	0.3
Maximum *local* perturbation threshold (ϵ_l in Equation 2)	0.5
Enforcement interval	15

Table 1: Hyper-parameters used for attacks

5 EVALUATION

5.1 Setup

Dataset. Since A^4 primarily targets the current version of AdGraph [11], we first reproduce its pipeline using its open source implementation. Given that the web crawl conducted in [11] 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 8 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 §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 10; (3) "Only Structural" includes 11 features in category "S" in Table 10. 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 1 and 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) 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.

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 §6. Note that all success rates in the

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

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 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 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 [22]) ranking of top-5 perturbed features used in AdGraph's random forest classifier in Table 9 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 §2, we assume a Greybox threat model where the attacker knows the entire training dataset. Even though we believe this is practical as the training dataset [11] uses is from crawling open webpages (i.e., can be easily 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 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.

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

Table 3: Mapping-back strategy significance analysis

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

Table 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.

Mapping-back strategy significance. Table 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 2) can be discovered with additional strategies.

Attack convergence. In Figure 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.

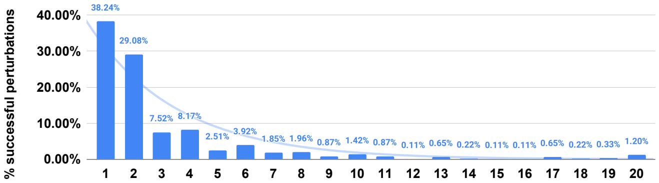


Figure 6: Attack convergence analysis

Node additions. Table 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 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 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.

[3] Note that it is not a surprising finding that random perturbations can evade ML classifiers in cases. Prior research [8, 9] 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.

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 Page Load Time (PLT) (following the standard method as used in [11]). Table 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 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 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 §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 [11]. [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 9 in the appendix shows an example with minor breakages after adding the perturbations, from the 100 successfully perturbed webpages used in the breakage analysis.

6 DISCUSSIONS AND LIMITATIONS

Deployability. As discussed in §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 §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 [29]. 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 §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 §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 §3, and (3) has associated side-effect offsets upon mapping from the feature-space to application-space (as shown in Figure 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 [34] etc.

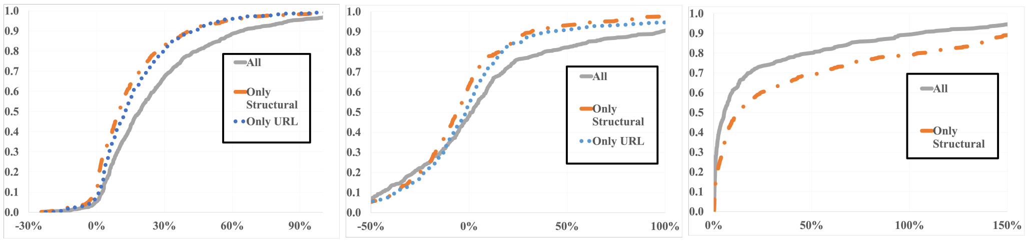
As noted in §3, currently A^4 is designed to perturb at most 19 features from the possible 65. Although this conservative limit makes our attack stealthier in practice, we plan to explore perturbations on more features in the future and analyze their effectiveness. Further, our current A^4 implementation only has two mapping-back strategies (centralized and distributed) as discussed in §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 §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

[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.

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 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.



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

Figure 7: Cumulative Distribution Function (CDF) for different measurements

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 10) 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 robustness of models through blending adversarial samples into training datasets (i.e., adversarial training) [8], and detecting them [16] 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 §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 [33].

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.

[21] 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, [21] creates opaque predicates in perturbed malware so that the transplanted gadgets won't be actually executed, to preserve malware semantics. Following similar routes, [32] and [31] aim to craft adversarial 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, [13] presents an attack against binary-based malware detectors, by injecting unused bytes.

Adblocking and anti-adblocking. As briefly mentioned in §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 [33] and other aggressive obstacles to avoid being detected. Recent research [15] shows that a number of publishers/advertisers are actively circumventing adblockers through various techniques including ad cloaking (e.g., unmonitored web APIs [2]), obfuscation (e.g., ad element randomization [30]), 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.

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.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers and our shepherd, Roberto Perdisci, for their insightful comments. This research was partially sponsored by the U.S. Army Combat Capabilities Development Command Army Research Laboratory and was accomplished under Cooperative Agreement Number W911NF-13-2-0045 (ARL Cyber Security CRA). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Combat Capabilities Development Command Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on. This work was also partially supported by NSF grants #1652954, #2051592, #2103439, #2102347 and #1719147.

REFERENCES

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

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

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

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

[5] François Chollet et al. 2015. Keras. https://keras.io.

[6] Catalin Cimpanu. 2018. 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/.

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

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

[9] Dan Hendrycks and Thomas Dietterich. 2019. Benchmarking Neural Network Robustness to Common Corruptions and Perturbations. *Proceedings of the International Conference on Learning Representations* (2019).

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

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

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

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

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

[15] Hieu Le, Athina Markopoulou, and Zubair Shafiq. 2021. CV-Inspector: Towards Automating Detection of Adblock Circumvention. In *The Network and Distributed System Security Symposium (NDSS)*. https://doi.org/10.14722/ndss.2021.24055

[16] Shasha Li, Shitong Zhu, Sudipta Paul, Amit Roy-Chowdhury, Chengyu Song, Srikanth Krishnamurthy, Ananthram Swami, and Kevin S Chan. 2020. Connecting the Dots: Detecting Adversarial Perturbations Using Context Inconsistency. In *European Conference on Computer Vision*. Springer, 396–413.

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

[18] PageFair. 2017. 2017 Global Adblock Report. PageFair. https://pagefair.com/downloads/2017/01/PageFair-2017-Adblock-Report.pdf.

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

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

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

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

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

[24] Jonas Rauber, Roland Zimmermann, Matthias Bethge, and Wieland Brendel. 2020. 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 (2020), 2607. https://doi.org/10.21105/joss.02607

[25] Leonard Richardson. 2007. Beautiful soup documentation. *April* (2007).

[26] Alexander Sjösten, Peter Snyder, Antonio Pastor, Panagiotis Papadopoulos, and Benjamin Livshits. 2020. Filter List Generation for Underserved Regions. In *Proceedings of The Web Conference 2020*. 1682–1692.

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

[28] Florian Tramèr, Pascal Dupré, Gili Rusak, Giancarlo Pellegrino, and Dan Boneh. 2019. AdVersarial: Perceptual Ad Blocking meets Adversarial Machine Learning. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 2005–2021.

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

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

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

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

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

[34] Shitong Zhu, Shasha Li, Zhongjie Wang, Xun Chen, Zhiyun Qian, Srikanth V Krishnamurthy, Kevin S Chan, and Ananthram Swami. 2020. 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*. 183–197.

A APPENDICES

A.1 Disguising DOM Perturbations

As discussed in §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 §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

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

Table 6: 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 7: Local surrogate NN's hyper-parameters and agreement rate

# 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 8: Dataset statistics

Rank	Feature No. # (from Table 10)	Category
1	19	Structural
2	3	URL
3	5	URL
4	7	URL
5	4	URL

Table 9: Information gain ranking of top-5 perturbed features

Code 1: Original DOM snippet

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

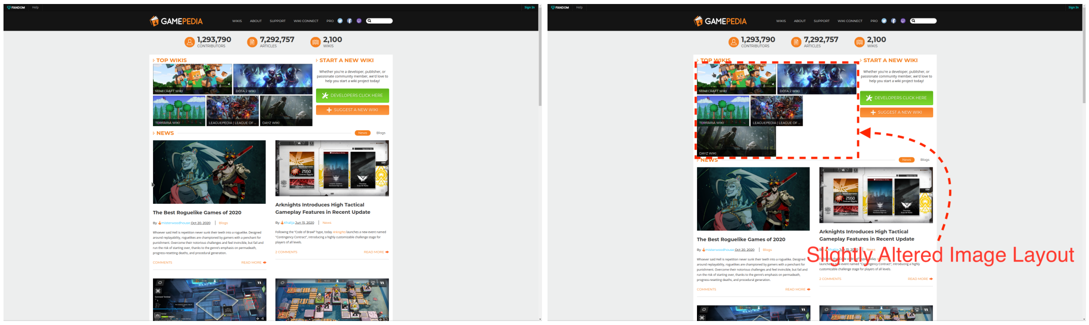
Code 2: Perturbed DOM snippet

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

Figure 8: Example DOM snippet with structural perturbations (inserted invisible sibling nodes)

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 10: List of perturbed features; Type includes – I: integer, B: binary, F: float; Category includes – S: structural, U: URL


GAMEPEDIA
1,293,790
7,292,757
2,100
TOP WIKIS
START A NEW WIKI
NEWS
The Best Roguelike Games of 2020
Arknights Introduces High Tactical Gameplay Features in Recent Update
GAMEPEDIA
1,293,790
7,292,757
2,100
START A NEW WIKI
The Best Roguelike Games of 2020
Arknights Introduces High Tactical Gameplay Features in Recent Update
Slightly Altered Image Layout

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

Figure 9: Example webpage with minor breakage