SpacePhish: The Evasion-space of Adversarial Attacks against Phishing Website Detectors using Machine Learning [Artifact]

Giovanni Apruzzese*
giovanni.apruzzese@uni.li
Institute of Information Systems
University of Liechtenstein

Mauro Conti†
conti@unipd.it
Department of Computer Science
Delft University of Technology, NL

Ying Yuan*
ying.yuan@math.unipd.it
Department of Mathematics
† University of Padua, Italy

ABSTRACT

Existing literature on adversarial Machine Learning (ML) focuses either on showing attacks that break every ML model, or defenses that withstand most attacks. Unfortunately, little consideration is given to the actual *cost* of the attack or the defense. Moreover, adversarial samples are often crafted in the "feature-space", making the corresponding evaluations of questionable value. Simply put, the current situation does not allow to estimate the actual threat posed by adversarial attacks, leading to a lack of secure ML systems.

We aim to clarify such confusion in this paper. By considering the application of ML for Phishing Website Detection (PWD), we formalize the "evasion-space" in which an adversarial perturbation can be introduced to fool a ML-PWD—demonstrating that even perturbations in the "feature-space" are useful. Then, we propose a realistic threat model describing evasion attacks against ML-PWD that are cheap to stage, and hence intrinsically more attractive for real phishers. Finally, we perform the first statistically validated assessment of state-of-the-art ML-PWD against 12 evasion attacks. Our evaluation shows (i) the true efficacy of evasion attempts that are more likely to occur; and (ii) the impact of perturbations crafted in different evasion-spaces. Our realistic evasion attempts induce a statistically significant degradation (3–10% at p <0.05), and their cheap cost makes them a subtle threat. Notably, however, some ML-PWD are immune to our most realistic attacks (p=0.22). Our contribution paves the way for a much needed re-assessment of adversarial attacks against ML systems for cybersecurity.

Our resources are publicly available: https://spacephish.github.io

CCS CONCEPTS

• **Security and privacy**; • **Computing methodologies** → **Machine learning**;

ACM Reference Format:
Giovanni Apruzzese, Mauro Conti, and Ying Yuan. 2022. SpacePhish: The Evasion-space of Adversarial Attacks against Phishing Website Detectors using Machine Learning [Artifact]. In *Annual Computer Security Applications Conference (ACSAC '22), December 5–9, 2022, Austin, TX, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3564625.3567980

1 FEATURE EXTRACTOR

An important part of our evaluation is represented by the feature extractor, for which we rely on the established guidelines provided in [8, 9] and still widely employed in recent literature (e.g., [6]). The underlying principle of such guidelines is to analyze several elements of a webpage (e.g., the length of its URL), and then use threshold-based mechanisms to determine whether such element is 'benign' or 'phishing' (e.g., a short URL is likely benign, whereas a long one is likely phishing). Any feature can have a value within [-1, 1], where -1 is 'benign' and 1 is 'phishing'. Our extractor generates all the features reported in Table 1. We explain some of them.

- (#1) *URL_length*. We compute the amount of character composing the entire URL. Strings shorter than 53 characters correspond to -1 (likely 'benign'), whereas longer strings correspond to +1 (likely 'phishing').
- (#4) *URL_short*. If the URL starts[1] with keywords related to popular shortening services (bit.ly, goo.gl, tinyurl, ad.fly) then this feature is set to +1, and to -1 otherwise.
- (#28) *URL_pageRank*. We use Open PageRank API to query the URL's domain. The response shows the page ranks from 0 to 10: the corresponding feature is normalized between -1 (if the rank is 10) and +1 (if the rank is 0).
- (#37) *HTML_objectRatio*. We capture all the objects embedded in the webpage, and compute the ratio of internal-to-external objects. An internal object either has its link starting with *../* or with the same 'root' as the website's URL. If the ratio is less than 0.15, then the value of this feature is -1 (likely benign), and +1 otherwise (likely phishing).
- (#38) *HTML_metaScripts*. Same as #37, but for scripts, links and metas. If the ratio is more than 0.61, the feature value is +1 (likely phishing); if the ratio is less than 0.52, the feature value is -1 (likely benign); otherwise, the value is set to 0.
- (#45) *HTML_nullLnkWeb*. We check how many links are useless, i.e., they point to the exact same page (e.g., *href=#*). The count of useless can be normalized between +1 (high number of useless links) and -1 (no useless links).
- (#51) *HTML_hiddenInput*. We check if there are any hidden *input* tags in the webpage. If there are, the feature value is +1 (likely phishing), and -1 otherwise (likely benign).
- (#52) *HTML_URLBrand*. We check (in the HTML) if the webpage *title* includes the brand name in the URL. If included, the feature value is -1 (benign); otherwise, is +1 (phishing).

[1] Our feature extractor is 'stateless'. Once it receives a sample, the only queries performed are those to some third-party services (e.g., PageRank API, DNS servers), which can be cached to save time. Our extractor, however, does not 'update' a sample: if, e.g., a URL uses a shortening service, then the extractor uses such 'shortened' URL as basis, and if the HTML changes (due to some automatic script) then such change will not be captured. Such choice makes sense because ML-PWD must be fast: a user does not want to wait seconds before visiting each website just because a phishing check is made. Moreover, our decision makes our extractor suitable also to to ML-PWD that analyze *only* the URL, because the webpage will not be opened in the first place (which is common for phishing email filters) due to the high overhead.

ACSAC '22, December 5–9, 2022, Austin, TX, USA

ACM ISBN 978-1-4503-9759-9/22/12.
https://doi.org/10.1145/3564625.3567980

(Our repository includes the source-code of our feature extractor.) We use similar thresholds as those by Mohammad et al. [8, 9], and are the same used to create the popular UCI dataset [1]. To validate our choice of using the same thresholds (which play a crucial role in our evaluation), we find instructive to report the length of URLs contained in our chosen datasets, i.e., Zenodo and δphish. The results are as follows: for Zenodo, there are 1500 URLs (out of 4000) which are *longer* than 54 characters; for δphish, there are 1909 URLs (out of 6523) which are *longer* than 54 characters. Hence, such a threshold is still sensible for more recent datasets.

Table 1: Features F of the considered ML-PWD.

#	Feature Name	#	Feature Name	#	Feature Name
1	URL_length	20	URL_shrtWordPath	39	HTML_commPage
2	URL_hasIPaddr	21	URL_lngWordURL	40	HTML_commPageFoot
3	URL_redirect	22	URL_DNS	41	HTML_SFH
4	URL_short	23	URL_domAge	42	HTML_popUp
5	URL_subdomains	24	URL_abnormal	43	HTML_rightClick
6	URL_atSymbol	25	URL_ports	44	HTML_domCopyright
7	URL_fakeHTTPS	26	URL_SSL	45	HTML_nullLnkWeb
8	URL_dash	27	URL_statisticRe	46	HTML_nullLnkFooter
9	URL_dataURI	28	URL_pageRank	47	HTML_brokenLnk
10	URL_commonTerms	29	URL_regLen	48	HTML_loginForm
11	URL_numerical	30	URL_checkGI	49	HTML_hiddenDiv
12	URL_pathExtend	31	URL_avgWordPath	50	HTML_hiddenButton
13	URL_punyCode	32	URL_avgWordHost	51	HTML_hiddenInput
14	URL_sensitiveWrd	33	URL_avgWordURL	52	HTML_URLBrand
15	URL_TLDinPath	34	URL_lngWordPath	53	HTML_iframe
16	URL_TLDinSub	35	URL_lngWordHost	54	HTML_favicon
17	URL_totalWords	36	HTML_freqDom	55	HTML_statBar
18	URL_shrtWordURL	37	HTML_objectRatio	56	HTML_css
19	URL_shrtWordHost	38	HTML_metaScripts	57	HTML_anchors

2 EXPERIMENTAL WORKFLOW

We follow the standard evaluation protocol for adversarial attacks at inference time: We first develop the targeted systems, and then assess their robustness against our adversarial attacks.

2.1 Development of the ML-PWD

We provide a schematic of the operations for our ML-PWD in Fig. 1. Such operations involve three phases (dotted squares in Fig. 1).

(1) *Setup.* The first phase is choosing a given source dataset (i.e., Zenodo or δphish) and partition its samples into *benign* and *phishing* (B and P, respectively). Then, we perform a *random split* (to avoid bias) on each of these partitions by using a 80:20 ratio (common in related literature [3, 4]). In other words, we randomly select 80% of the samples in both B and P (i.e., B_t and P_t respectively), which will be used to train the ML model. The leftout samples, B_i and P_i (corresponding to 20% of B and P, respectively), are used to assess the inference performance of the resulting ML model. We will also use P_i as basis to craft our adversarial samples.

(2) *Training.* To train $\mathcal{M}$, we recall that the source data is in raw format. Hence, before obtaining the training dataset $\mathcal{D}$, the corresponding *training* partitions B_t and P_t must be transformed into their feature representation. Hence, we develop a feature extractor (described in §1 of this Artifact) that is based on a given feature set F (either F^u, F^r, or F^c). Then, we preprocess both B_t and P_t to obtain the actual training data $\mathcal{D}$. At this point, we apply a given ML algorithm $\mathcal{A}$ (either *RF*, *LR* or *CN*) to such $\mathcal{D}$; the resulting ML model $\mathcal{M}$ (a binary classifier, which we fine tune via grid-search) will be the detection component of the considered ML-PWD.

(3) *Testing.* The last phase is measuring the performance of $\mathcal{M}$. In our case, a ML-PWD must exhibit both a high detection rate and a low false positive rate: indeed, no one is interested in detectors that block legitimate websites due to excessive false alarms. Hence, we preprocess the *inference* partitions B_i and P_i (by considering the proper F) and measure the fpr and tpr—in the absence of adversarial attacks.

The topmost priority is ensuring that $\mathcal{M}$ analyzing F^c achieve optimal performance: indeed, models using either F^u or F^r are expected to exhibit a lower performance as they are provided with less information; however, using F^u or F^r is expected to yield a superior robustness in the presence of evasion attacks. (Our repository includes the best parameter configurations of each ML algorithm.)

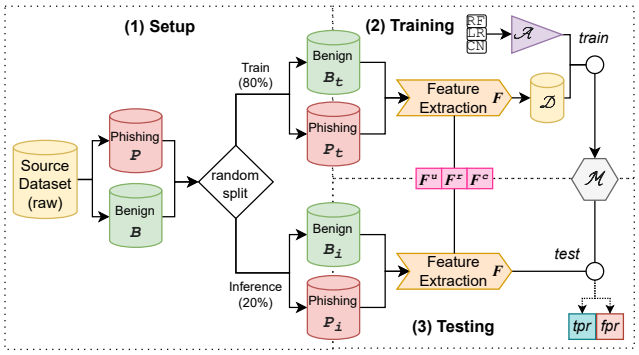


Figure 1: Workflow to develop our baseline ML-PWD. Each source dataset (containing benign, B, and phishing, P, samples) is randomly split into the training (B_t and P_t) and inference (B_i and P_i) partitions, used to train and test each ML-PWD. We use P_i as basis for our adversarial samples.

2.2 Attack Simulation (from Appendix D)

The procedure to assess the adversarial attacks involves three steps.

(1) *Isolate.* Our threat model envisions evasion attacks that occur during inference, hence our adversarial samples are generated from those in P_i. Furthermore, we recall that the attacker expects the ML-PWD to be effective against 'regular' malicious samples. To meet such condition, we isolate 100 samples from P_i that are detected successfully by the best[2] ML-PWD (typically using F^c) during one of our runs. Such samples are then used as basis to craft the adversarial samples corresponding to each of the 12 considered types of evasion attacks.

(2) *Perturb.* We apply the perturbations as follows. For WA and $\widehat{\text{WA}}$, we craft the corresponding WsP, apply them to each of the 100 samples from P_i, and then preprocess such samples by using the feature extractor. For PA and MA, we first

[2]This ensures that all ML-PWD are assessed against the *same* adversarial samples. We provide such sampmles in the source-code.

preprocess the 100 samples with the feature extractor, and then apply the corresponding PsP or MsP. Overall, these operations result in 1200 adversarial samples (given by 12 attacks, each using 100 samples).

(3) *Evade.* The 1200 adversarial samples are then sent to all the 9 ML-PWD (for each dataset), and we measure the tpr again.

The expected result it that the tpr obtained on the adversarial samples (generated as a result of any of the 12 considered attacks) will be lower than the tpr on the original 100 phishing samples.

3 ATTACKS IMPLEMENTATION

Let us discuss how we implement our perturbations, and provide some insight as to which features are influenced as a result of our attacks. We recall that each attack family presents three variants, depending on which features the attacker is 'consciously' trying to affect. Namely: u, r and c, i.e., features involving the URL, the representation (HTML) or a combination thereof. All attacks are created by manipulating (phishing) samples taken from P_i. In particular, during our first trial we isolate 100 samples from P_i that are correctly detected by the best ML-PWD: such samples are then used as basis for all their adversarial variants (to ensure consistency). For simplicity, we will denote any of such samples as p.

We start by describing MA which are the easiest to implement. Then, we describe WA and $\widehat{\text{WA}}$. Finally, we describe PA, which are the most complex to implement because they must consider several implications (e.g., inter-feature dependencies). (Our repository includes the exact implementation of MA and PA, and also all the pre-processed variant of the samples generated via WA and $\widehat{\text{WA}}$.)

3.1 ML-space Attacks (MA)

These attacks are the easiest to implement. Indeed, we simply follow the same procedure as done by most prior works (e.g., [5, 7]) that directly manipulate the feature representation F_p of a sample p right before it is analyzed by the ML-PWD. We do this without taking into account any inter-dependency between features and/or any physical property that the actual webpage must preserve: this is compliant with our assumption that the attacker has access to the ML-space. Specifically, for each MA we apply the following MsP:

- MA^u: The attacker targets URL-related features. Hence, we manipulate F_p by setting features based on F^u equal to -1, which denotes a value that is more likely associated with a benign sample. In particular, we set to -1 the features in Table 1 with the following numbers: (1-17,19-21,27,30-35)
- MA^r: Same as above, but the targeted features are within F^r. Hence, we set to -1 the features in Table 1 with the following numbers: (36-40,42-52,54-57)
- MA^c: We set to -1 all features involved in MA^u and MA^r.

We remark that the attacker is not aware of the feature importance (because it would require knowledge of $\mathcal{M}$). Hence, although some manipulations will likely 'move' F_p towards a benign webpage, it is not guaranteed that $\mathcal{M}$ will actually classify such F_p as benign: if the manipulated features are not important, then even MsP may have no effect (and such phenomenon *does* happen in our evaluation, e.g., the ML-PWD using *RF* with F^c on Zenodo against MA^r).

Of course, we could set *all* features to -1 (e.g., all F^u and F^r). Doing this, however, would obviously result in a perfect misclassification (and hence not interesting to show). Moreover, it would not be sensible *even for the attacker*. Indeed, MA assume no knowledge of $\mathcal{M}$ and of $\mathcal{D}$, meaning that an attacker may suspect the existence of a honeypot [10]. For instance, $\mathcal{D}$ may contain some samples with all features set to -1 (i.e., benign) that are labelled as phishing—for the sole purpose of defeating similar attacks in the ML-space. Hence, it is realistic to assume that even an attacker capable of MA would not exaggerate with their perturbations.

3.2 Website Attacks (WA and $\widehat{\text{WA}}$)

We recall that we perform two families of attacks in the website-space: WA and $\widehat{\text{WA}}$. The peculiarity of both of these attacks (both relying on WsP) is that the attacker does not have access to the ML-PWD. Hence, they are not able to manipulate F_p, and they are not even able to *observe* F_p.

3.2.1 WA. These attacks resemble the pragmatic example provided in Appendix B of the main paper.

- WA^u: We set the URL to a random string starting with "www.bit.ly/", followed by 7 randomly chosen characters (which what this popular URL shortener does).
- WA^r: For δPhish, we change the HTML by adding 50 invisible internal links (i.e., having the same root domain of the website);[3] for Zenodo, we wrap all links within an "onclick", i.e., we change: *<a href='link'>* into *<a onclick="this.href='link'">*.[4]
- WA^c: We do both of the above for each dataset.

3.2.2 $\widehat{\text{WA}}$. These attacks envision an attacker that knows how the feature extractor within the ML-PWD operates (see 1). Such knowledge can be acquired, e.g., if the attacker has (or is) an insider that provided them with such intelligence. However, the attacker is still confined in the website-space, and hence can only apply WsP (to generate $\bar{p}$). For a meaningful comparison, we assume an attacker who is aware of how the features targeted in WA are "extracted" within the ML-PWD. Hence, we craft each $\widehat{\text{WA}}$ as follows:

- $\widehat{\text{WA}^u}$: The attacker, having knowledge of the extractor, knows that by using an URL shortener they will affect all features related to the URL (i.e., F^u); furthermore, they know the threshold (53) that makes an URL to be considered as 'benign'. Such length is well above that of an URL generated via any shortening service. As such, these attacks are an exact replica as $\widehat{\text{WA}^u}$ (the only difference is that the attacker of $\widehat{\text{WA}^u}$ is more confident than the one in WA^u).
- $\widehat{\text{WA}^r}$: The attacker manipulates the HTML in the same was as in WA^c. However, the attacker also knows the threshold (0.15) of internal-to-external links that yields a benign value of the *HTML_objectRatio* feature. Hence, the WsP manipulate the HTML of each p by introducing as many links (or wrappings) as necessary to meet such threshold.
- $\widehat{\text{WA}^c}$: The attacker does both of the above.

We stress that the attacker cannot observe $F_{\bar{p}}$. Indeed, doing this would require the attacker to completely replicate the feature extractor, which is costly, and may not even be possible (some third-party

[3] The exact string we inject is: "* can not see*", which is the second string shown in our pragmatic example (Appendix B in the paper).

[4] Such WsP, if applied to textual link, would remove the underline of such a link, therefore being visible to a user; however, it is possible to make it invisible by editing the CSS properties. Our feature extractor is agnostic of such properties, so we do not do this: the results would be equivalent.

services may require subscriptions to be used). As such, the attacker is aware of how to craft WsP that are more likely noticed by the ML-PWD, but evasion is not guaranteed.

3.3 Preprocessing Attacks

These attacks are the most complex to realize *from a research perspective* and *in a fair way*. Let us explain.

Challenges. The underlying principle of PsP (the backbone of PA) is affecting the preprocessing space of the ML-PWD. Technically, since we are the developers of our own feature-extractor (i.e., the component of the ML-PWD devoted to data preprocessing), we could simply directly manipulate our own extractor, i.e., by introducing a 'backdoor'. However, doing this would prevent a fair generalization of our results: for instance, it is possible to develop another feature extractor, having the same functionality but whose operations are executed in a different order. Hence, to ensure a more fair evaluation, we adopt a different approach: we apply the perturbations *at the end* of the preprocessing phase, but we do so by anticipating how a perturbation in the website-space (a WsP) could affect the preprocessing-space, thereby turning a WsP into a "physically realizable" PsP. To this purpose, we *assume the viewpoint of an attacker*. For instance, we ask ourselves: "if an attacker wants to affect URL features by using an URL shortener, how would the feature extractor react?".

Scenario. In PA the attacker *knows* and *can interfere* (through PsP) with the feature extraction process of the targeted ML-PWD. However, the attacker is *not* aware of what happens next: the ML-space and the output-space are both inaccessible by the attacker (from both a *read* and *write* perspective). Hence, once the PsP has been applied and $\overline{F_p}$ is generated, the attacker cannot influence $\overline{F_p}$ any longer. For each PA we do the following:

- PA^u: we anticipate an attack that targets URL features, and specifically *URL_length*, by using an URL shortener. Hence, we can foresee that operations (in the website-space) can lead to alterations of *all* the features involved with the URL (i.e., F^u). For instance, doing this would make weird characters (if present) to disappear from the URL. However, doing this would induce to alterations also to F^r. For instance, some objects originally considered to be 'internal' would become 'external'. Hence, we implement PA^u by setting the following features (from Table 1) to -1: (1-3,5,6,8,10-16,22,23,25,26,28-30), whereas the following features are set to +1: (4,27,36-38,41,44,48,52,54,56).
- PA^r: we anticipate an attack that targets features related to the representation of a website—in our case the HTML, and specifically the *HTML_objectRatio* feature. We foresee that an attacker can interfere with such feature in many ways, for instance by removing links, adding new ones, or changing those already contained in the webpage. All such changes will affect many features, such as the *HTML_freqDom*: because populating the HTML with (fake) internal links would change the 'frequent domains' included in the HTML. Such changes can also affect the links in the footer of the webpage (*HTML_nullLnkFooter*); or the anchors (*HTML_anchors*); but also others. We implement PA^r by setting the following features (from Table 1) to -1: (36–38,41,51,54,56,57); whereas we set (39,40) to 1 and 46 to 0.
- PA^c: they are a combination of the two above. We expect the attacker to use a URL shortener, and also infterfer with the HTML_objectRatio. However, we cannot simply set the features to the same values as PA^r and PA^u, because one of the two will prevail. In our case, shortening the URL will be 'stronger', because the URL will change (to that of the URL shortener) and hence the internal objects will become 'external'. Hence, we implement PA^c by setting the following features (from Table 1) to -1: (1-3,5,6,8,10-16,22,23,25,26,28-30), whereas the following features are set to +1: (4,27,36-38,41,44,48,52,54,56).

Nevertheless, we remark that such PsP may not yield an $\overline{F_p}$ that is a perfect match with a $F_{\bar{p}}$ generated via WsP (i.e., those of $\widehat{\text{WA}}$). Indeed, some inconsistencies may be present—likely due to 'inaccurate' anticipations from our (i.e., the attacker's) side. Such inconsistencies are sensible. An attacker with access to the preprocessing-space can theoretically *replicate* the entire feature extractor, and use it to exactly pinpoint how to generate PsP that are an exact match with WsP (i.e., $\overline{F_p}$=$F_{\bar{p}}$). However, doing this would be *very expensive*. Furthermore, it would *defeat the purpose* of using PsP: the attacker does not want that $\overline{F_p}$=$F_{\bar{p}}$, rather, they want a PsP that is 'stronger'; otherwise, why use PsP in the first place?

4 PROOF-OF-CONCEPT: ATTACKS AGAINST A COMPETITION-GRADE ML-PWD

To further prove the impact of our 'cheap' attacks (i.e., WA), we tested them on a real ML-PWD that is used in a (currently ongoing at the time of writing the paper) well-known Machine Learning Security Evasion Competition (MLSEC [2]). Such competition is held yearly, and is organized by leading tech-companies that provide cybersecurity services reliant on ML methods. The 2022 edition of MLSEC envisions a challenge in which participants are asked to *evade* ML-PWD. We took this opportunity to assess whether our attacks had any impact against such 'competition-grade' ML-PWD. Short story: they do. A demonstrative video can be found at the homepage of our website (which also includes the source-code).

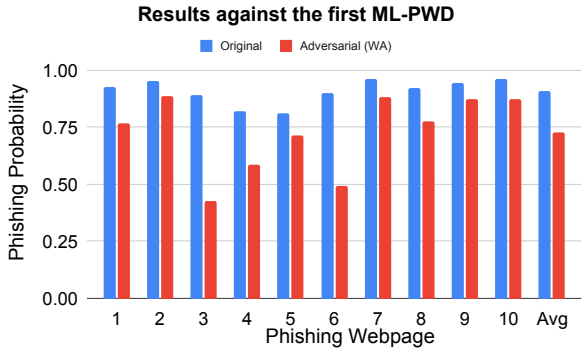
4.1 Challenge

Participants of the phishing evasion challenge are given 10 'phishing' webpages, which are provided in their raw HTML form. The purpose of the challenge is to manipulate such webpages so that (i) they render exactly as the originals, and (ii) they evade a ML-PWD. Specifically, the organizers provide 8 different ML-PWD, which the participants can use as a black-box: by sending an input (i.e., the HTML of a phishing webpage), they are given an output (i.e., the probability that such webpage is malicious—according to the specific ML-PWD). Such ML-PWDs only analyze the HTML of the webpage (which must render exactly as the original).

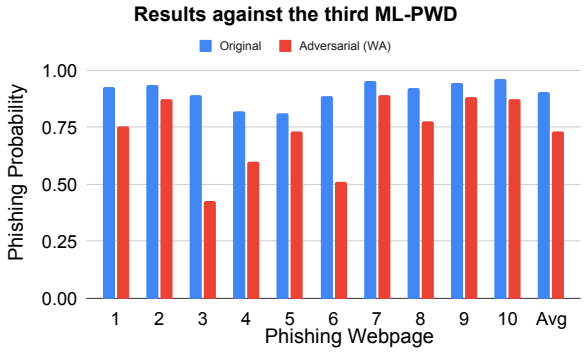
Put simply: the objective of the challenge is to tweak the HTML of the 10 webpages with imperceptible modifications that decrease the confidence of the 8 ML-PWD.

4.2 Method

Of course, the setting described above perfectly describes the black-box scenarios envisioned in adversarial ML papers: query the detector, and use the response as a guide to craft a more evasive phishing



(a) Impact of WA^r on the *first* ML-PWD.



(b) Impact of WA^r on the *third* ML-PWD.

Figure 2: Effectiveness of the most likely attacks (WA^r on δPhish) against the ML-PWD provided by the organizers of MLSEC [2].

webpage. Our primary attacks (WA), however, are query-less. Because we are aware that the target ML-PWD analyzes the HTML (recall that this is an assumption of our threat model), we then craft our 'adversarial' phishing webpages by using exactly the same WA^r used in our paper for δPhish: we add 50 invisible internal links. We apply these WsP to all the 10 webpages *provided by the organizers of the challenge*, and then test whether they had any impact to the *real* ML-PWD involved in the challenge.

4.3 Results

By taking into account *all* webpages against *all* ML-PWD, our attacks induced a drop of 3.4% in the confidence of the ML-PWD, indicating that our WsP had some effect. However, while some ML-PWD were not very affected, others incurred a significant drop.

Specifically, we focus our attention on the first and third ML-PWD provided by the organizers of MLSEC. The results of our proof-of-concept experiments are shown in Figs. 2. These graphs show phishing probability (y-axis) given as output by the corresponding ML-PWD for each of the 10 webpages of the challenge (x-axis). We report two bars: the blue bar are the results of the original webpages, whereas the red bars are the results after applying our WsP.

4.4 Discussion

These two detectors were significantly less certain after our WsP, with an average confidence drop of 17.5%. We observe that in most cases, the confidences were still above 0.5 (i.e., the webpages would still be classified as 'phishing'). A more detailed look, however, reveals that **these detectors were completely fooled by some webpages** (i.e., their confidence dropped to below 0.5). We report:

- Page #3: from 0.90 down to 0.43 for the 1st and 3rd detectors.
- Page #6: from 0.90 down to 0.49 for the 1st detector.

We also attempted the same WA^r by changing the number of fake links, and also by considering a different string[5]. When applied to, e.g., webpage #3, adding 280 links dropped the confidence to below 0.2; whereas adding a slightly different string (the first one shown in our pragmatic example in Appendix B) 280 times, the confidence dropped to 0.2 for the first and third detector, and to 0.49 for the seventh detector. The seventh detector was also fooled by adding such alternative string 50 times to webpage #4, causing a confidence of 0.46 (down from 0.68). The source-code is available in our Artifact, and the experiments are entirely reproducible.

Interestingly, these results align with those shown in our main paper: our *query-less* WA attacks cannot bypass any ML-PWD, but in some cases *they can induce a miss-classification.*

[5]We also considered the 'wrapping' WsP for Zenodo: the effects were negligible—probably because these ML-PWD factored such links into their 'count' (i.e., the attacker made a wrong guess).

5 COMPLETE BENCHMARK TABLES

We carry out our experiments by developing original software tools, all written in Python3 by leveraging well-known libraries (e.g., scikit-learn, Tensorflow). The ML-PWD using *RF* and *LR* are assessed on a system mounting an Intel Xeon W-2223@3.6GHz with 32GB RAM. For the *CN*, we use an nVidia P100 GPU. (Our results have been reproduced during the ACSAC artifact evaluation.)

Baseline. Table 2 shows the average (and std. dev.) fpr and tpr of all 18 ML-PWD across the 50 trials when no attack occurs. Boldface denotes the best ML-PWD on each dataset. We highlight the high standard deviation of some detectors (e.g., the *CN* analyzing F^r on Zenodo), which justifies our decision to perform many trials.

Table 2: Performance in non-adversarial settings, reported as the average (and std. dev.) tpr and fpr over the 50 trials.

$\mathcal{A}$	F	Zenodo		δphish	
		tpr	fpr	tpr	fpr
CN	F^u	0.96±0.008	0.021±0.0077	0.55±0.030	0.037±0.0076
	F^r	0.88±0.018	0.155±0.0165	0.81±0.019	0.008±0.0020
	F^c	0.97±0.006	0.018±0.0088	0.93±0.013	0.005±0.0025
RF	F^u	0.98±0.004	0.007±0.0055	0.45±0.022	0.003±0.0014
	F^r	0.93±0.013	0.025±0.0118	0.94±0.016	0.006±0.0025
	F^c	**0.98±0.006**	**0.007±0.0046**	**0.97±0.007**	**0.001±0.0011**
LR	F^u	0.95±0.009	0.037±0.0100	0.24±0.017	0.011±0.0026
	F^r	0.82±0.017	0.144±0.0171	0.74±0.025	0.018±0.0036
	F^c	0.96±0.007	0.025±0.0077	0.81±0.020	0.013±0.0037

Table 3: Evasion Robustness of the ML-PWD on the Zenodo dataset. The cells report the average (and std. dev.) *tpr* over the 50 reiterations. Lines correspond to the ML-PWD, while rows correspond to a specific attack.

$\mathcal{A}$	F	no-atk	WA^u	WA^r	WA^c	$\widehat{\text{WA}^u}$	$\widehat{\text{WA}^r}$	$\widehat{\text{WA}^c}$	PA^u	PA^r	PA^c	MA^u	MA^r	MA^c
CN	F^u	0.96±0.007	1.00±0.000	0.93±0.020	1.00±0.000	1.00±0.000	0.95±0.018	1.00±0.000	1.00±0.017	0.95±0.018	1.00±0.017	0.18±0.222	0.95±0.018	0.18±0.222
	F^r	0.86±0.013	0.88±0.013	0.87±0.056	0.87±0.055	0.88±0.013	0.44±0.153	0.83±0.051	0.54±0.108	0.29±0.120	0.31±0.118	0.88±0.013	0.02±0.095	0.02±0.095
	F^c	0.97±0.009	0.92±0.036	0.93±0.020	0.94±0.063	0.92±0.036	0.92±0.016	0.83±0.115	1.00±0.011	0.90±0.031	0.99±0.017	0.51±0.131	0.92±0.036	0.15±0.211
RF	F^u	0.96±0.007	1.00±0.000	0.96±0.008	1.00±0.000	1.00±0.000	0.96±0.008	1.00±0.000	0.54±0.183	0.96±0.007	0.54±0.183	0.04±0.098	0.96±0.007	0.04±0.098
	F^r	0.90±0.013	0.90±0.013	0.88±0.024	0.88±0.025	0.90±0.013	0.71±0.053	0.80±0.025	0.59±0.086	0.47±0.082	0.30±0.088	0.90±0.013	0.04±0.155	0.04±0.155
	F^c	0.97±0.009	0.98±0.064	0.94±0.012	0.94±0.171	0.98±0.063	0.94±0.010	0.94±0.191	0.65±0.101	0.94±0.010	0.21±0.134	0.07±0.115	0.92±0.012	0.03±0.158
LR	F^u	0.97±0.005	1.00±0.000	0.95±0.005	1.00±0.000	1.00±0.000	0.96±0.005	1.00±0.000	0.73±0.071	0.96±0.006	0.73±0.071	0.00±0.000	0.96±0.006	0.00±0.000
	F^r	0.80±0.013	0.80±0.013	0.65±0.043	0.64±0.040	0.80±0.013	0.54±0.027	0.56±0.022	0.61±0.007	0.08±0.013	0.01±0.010	0.80±0.013	0.00±0.000	0.00±0.000
	F^c	0.98±0.005	0.82±0.035	0.95±0.015	0.32±0.079	0.80±0.038	0.93±0.014	0.32±0.132	0.46±0.053	0.91±0.032	0.06±0.025	0.00±0.000	0.76±0.036	0.00±0.000

Table 4: Evasion Robustness of the ML-PWD on the δphish dataset. The cells report the average (and std. dev.) *tpr* over the 50 reiterations. Lines correspond to the ML-PWD, while rows correspond to a specific attack.

$\mathcal{A}$	F	no-atk	WA^u	WA^r	WA^c	$\widehat{\text{WA}^u}$	$\widehat{\text{WA}^r}$	$\widehat{\text{WA}^c}$	PA^u	PA^r	PA^c	MA^u	MA^r	MA^c
CN	F^u	0.65±0.028	0.91±0.276	0.65±0.029	0.91±0.275	0.90±0.299	0.65±0.029	0.90±0.300	0.60±0.165	0.65±0.028	0.60±0.165	0.14±0.346	0.65±0.028	0.14±0.346
	F^r	0.79±0.013	0.80±0.013	0.35±0.018	0.34±0.017	0.80±0.013	0.86±0.033	0.88±0.020	0.46±0.065	0.69±0.038	0.46±0.064	0.81±0.013	0.00±0.000	0.00±0.000
	F^c	0.95±0.010	0.88±0.066	0.93±0.012	0.84±0.113	0.89±0.046	0.89±0.020	0.87±0.058	0.90±0.107	0.58±0.059	0.82±0.163	0.04±0.198	0.01±0.011	0.04±0.196
RF	F^u	0.56±0.037	0.84±0.330	0.56±0.036	0.84±0.330	0.84±0.330	0.56±0.034	0.84±0.331	0.57±0.238	0.56±0.037	0.57±0.238	0.01±0.053	0.56±0.037	0.01±0.053
	F^r	0.95±0.008	0.95±0.009	0.84±0.003	0.84±0.043	0.95±0.009	0.80±0.038	0.94±0.009	0.84±0.049	0.55±0.090	0.95±0.055	0.95±0.008	0.00±0.000	0.00±0.000
	F^c	0.95±0.009	0.90±0.020	0.92±0.006	0.77±0.047	0.90±0.017	0.86±0.018	0.92±0.015	0.90±0.065	0.68±0.013	0.86±0.097	0.88±0.026	0.00±0.001	0.00±0.000
LR	F^u	0.30±0.014	0.21±0.332	0.30±0.015	0.22±0.341	0.26±0.364	0.30±0.015	0.24±0.359	0.64±0.256	0.30±0.014	0.64±0.256	0.00±0.000	0.30±0.014	0.00±0.000
	F^r	0.78±0.011	0.78±0.011	0.57±0.014	0.56±0.047	0.78±0.011	0.60±0.030	0.63±0.010	0.80±0.029	0.04±0.006	0.45±0.068	0.78±0.011	0.00±0.000	0.00±0.000
	F^c	0.86±0.014	0.47±0.094	0.81±0.011	0.36±0.102	0.73±0.126	0.73±0.018	0.63±0.150	0.65±0.157	0.23±0.014	0.32±0.109	0.00±0.000	0.00±0.000	0.00±0.000

Evasion Performance We report the complete results of all the 12 considered evasion attacks against all the 18 considered ML-PWD in Table 3 (for Zenodo) and Table 4 (for δphish). These tables also include the performance in non-adversarial settings computed on the 100 phishing samples (drawn from P_i that are used as base for the adversarial samples). We remark that we chose such 100 samples by randomly selecting 100 samples which were correctly detected by the best ML-PWD on each dataset. As such, the *tpr* reported in the *no-atk* column can slightly differ from the one in Table 2 (which is computed on the entire P_i).

Runtime. We report in Table 5 the runtime for training and testing all our ML-PWD in non-adversarial scenarios. The values denote the average runtime (and standard deviation) across the 50 trials. Training the *RF* and *LR* uses all cores/threads of our CPU.

Table 5: Execution Times for training (on $\mathcal{D}$) and testing (on both P_i and B_i) the ML models used by our ML-PWD.

$\mathcal{A}$	F	Zenodo		δphish	
		Train (s)	Test (ms)	Train (s)	Test (ms)
CN	F^u	110.88±15.318	178.13±9.661	201.314±21.753	301.91±46.133
	F^r	76.61±4.562	171.95±10.577	167.74±25.197	273.4±43.99
	F^c	152.325±13.183	222.696±86.618	165.486±23.367	274.84±47.975
RF	F^u	0.152±0.0052	7.59±0.208	0.583±0.0181	28.09±0.402
	F^r	0.146±0.0037	7.85±0.07	0.369±0.0181	22.39±0.151
	F^c	0.179±0.0035	9.39±0.312	0.44±0.0062	23.6±0.205
LR	F^u	0.045±0.019	0.1±0.005	0.185±0.0285	0.45±0.895
	F^r	0.055±0.0182	0.09±0.003	0.083±0.0509	0.74±1.161
	F^c	0.063±0.0179	0.17±0.014	0.301±0.0678	0.36±0.678

REFERENCES

[1] 2015. UCI Phishing Websites Dataset. https://archive.ics.uci.edu/ml/datasets/phishing+websites.

[2] 2022. Machine Learning Security Evasion Competition. https://mlsec.io/.

[3] Rayah Al-Qurashi, Ahmed AlEroud, Ahmad A Saifan, Mohammad Alsmadi, and Izzat Alsmadi. 2021. Generating Optimal Attack Paths in Generative Adversarial Phishing. In *Proc. IEEE Int. Conf. Intell. Secur. Inf.*

[4] Trinh Nguyen Bac, Phan The Duy, and Van-Hau Pham. 2021. PWDGAN: Generating Adversarial Malicious URL Examples for Deceiving Black-Box Phishing Website Detector using GANs. In *Proc. IEEE Int. Conf. Machin. Learn. Appl. Netw.*

[5] Igino Corona, Battista Biggio, Matteo Contini, Luca Piras, Roberto Corda, Mauro Mereu, Guido Mureddu, Davide Ariu, and Fabio Roli. 2017. Deltaphish: Detecting phishing webpages in compromised websites. In *Proc. Springer Europ. Symp. Res. Comput. Secur.* 370–388.

[6] Ankit Kumar Jain and Brij B Gupta. 2018. Towards detection of phishing websites on client-side using machine learning based approach. *Telecom. Syst.* (2018).

[7] Jehyun Lee, Pingxiao Ye, Ruofan Liu, Dinil Mon Divakaran, and Mun Choon Chan. 2020. Building robust phishing detection system: an empirical analysis. *Proc. Netw. Distrib. Syst. Symp. – MADWeb Workshop* (2020).

[8] Rami M Mohammad, Fadi Thabtah, and Lee McCluskey. 2014. Intelligent rule-based phishing websites classification. *IET Inf. Secur.* (2014).

[9] Rami M Mohammad, Fadi Thabtah, and Lee McCluskey. 2014. Predicting phishing websites based on self-structuring neural network. *Neur. Comp. Appl.* (2014).

[10] Shawn Shan, Emily Wenger, Bolun Wang, Bo Li, Haitao Zheng, and Ben Y. Zhao. 2020. Gotta Catch'Em All: Using Honeypots to Catch Adversarial Attacks on Neural Networks. In *Proc. ACM SIGSAC Conf. Comp. Commun. Secur.* 67–83.