Fighting Phishing at the Website Host

Niclas van Eyk

February 15, 2023

Abstract

The Anti Phishing Working Group (APWG) regularly reports an increase in phishing attacks. Recent events, such as the COVID-19 pandemic, forced more people to use the internet, making phishing more interesting and amplifying the aforementioned trend. Users can protect themselves client-side by relying on built-in browser security mechanisms or installing additional extensions to identify dangerous websites. However, this shifts the responsibility to each end-user. This work focuses on finding ways to detect phishing at the website host instead. All users are protected by such mechanisms, making it more efficient than client-side detection. The website host also has access to data not commonly available to clients, which could help to improve the classification process. CodeSandbox's platform regularly is exploited by phishers as a free website host. Their data is used to create a machine learning system that helps to classify phishing before it is able to fool victims. With the XGBoost classifier achieving a balanced accuracy of 90.64%, the results are comparable to academic literature.

Contents

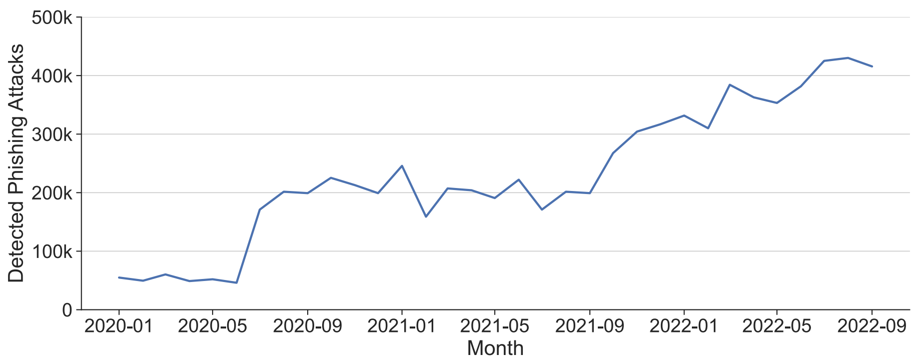


Figure 1.1: Detected phishing websites since 2020 as reported by the APWG [2].

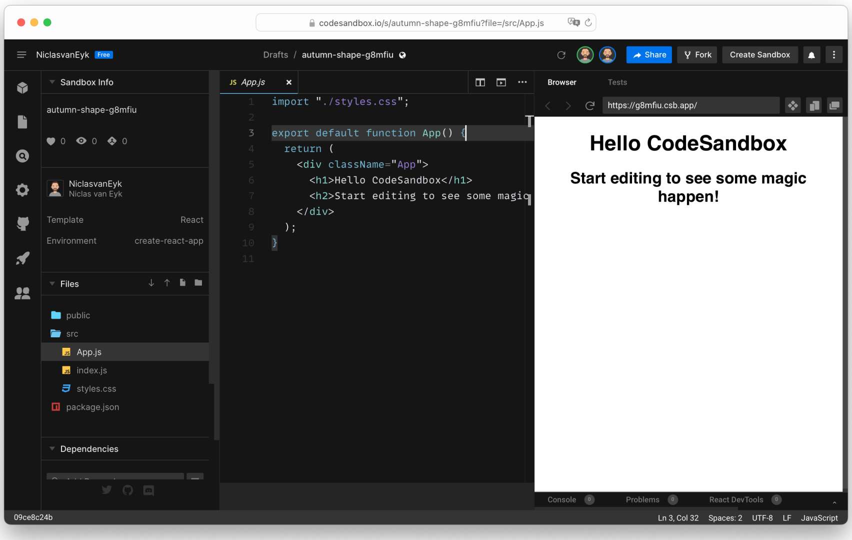
1 Introduction

Phishing is a social cyber-attack where the attacker tricks the victim into trusting them by impersonating another authority. Common examples include stealing user credentials by rebuilding a popular website's login form but hosting it under a different name. The victim thinks it logs into their account, but in reality, they provide sensitive information to the attacker. Technically literate or trained staff *can* identify such attacks by e.g. noticing that the website's URL is different. However, research suggests that even after being made aware that a website may not be legitimate, many people are not able to reliably distinguish them from a fake one [1].

1.1 Motivation

As the recent Covid-19 pandemic forced many people to work from home, a large number of employees is now required to use the internet, even if they lack technical literacy. This rise in internet users also sparked an increase in the number of phishing attacks [3]. Since the first lockdowns during the first half of 2020, the Anti Phishing Working Group (APWG) regularly reports new all-time highs for the number of detected phishing attacks [2], as can be seen in Figure 1.1. Detecting and warning users about them is therefore more important than ever.

To fight phishing, one can try to educate users or install software that scans websites visited by them and either runs detection algorithms [4, 5] or looks up known phishing incidents reported by third parties [6]. All of these measures work but require additional steps by the user of the computer, who is interested in not being a victim of a phishing attack. However, platforms hosting content on the internet also might be interested in preventing phishing, as it hurts their users and reputation. A user, who visits a link on a social media website and falls for a phishing attack, is likely associating it with their negative experience, driving them away from the platform. Therefore, actively fighting phishing at the host level benefits the platform *and* its users.



```
import "./styles.css";

export default function App() {
  return (
    <div className="App">
      <h1>Hello CodeSandbox</h1>
      <h2>Start editing to see some magic
    </div>
  );
}
```

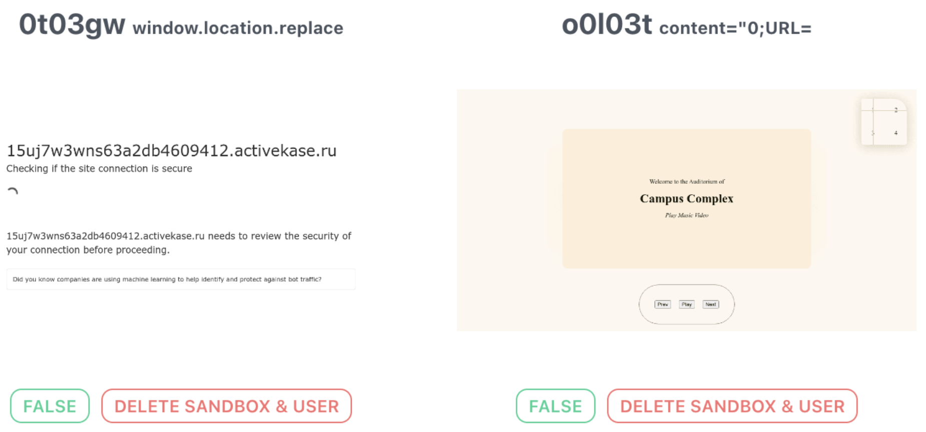
Figure 1.2: The CodeSandbox online editor.

1.2 CodeSandbox

CodeSandbox[1] provides web developers with a tool for collaboratively creating websites. Their virtualized in-browser *sandboxes* are set up for common web development tasks. This is shown in Figure 1.2, where the code can be edited on the left, and the resulting website is displayed on the right in an automatically refreshing website preview. The preview can also be accessed without the editor user interface via a short link (e.g. `https://g8mfiu.csb.app`), essentially making CodeSandbox a free website host. According to literature [7], such short links are attractive to phishers since people are used to them due to the widespread usage of URL-shortener services on social media. These two features make it an attractive target for phishers to exploit. As the host of potential phishing websites, they are in a unique position to automatically detect and remove them, thereby saving users from getting their credentials stolen. However, phishing detection on a platform aimed at helping developers to create websites faces challenges not present in traditional phishing detection. Users often try to rebuild popular websites as a training exercise. Falsely flagging such harmless replicas as phishing hurts the user experience and may drive them away from the platform. Additionally, the difference between a harmful replica and a dangerous one is hard to detect since the web provides multiple ways of transferring data to an external source.

Currently, CodeSandbox runs a number of checks when a preview is opened to detect phishing sandboxes. This saves computing resources compared to running them on every change, since a phishing website that has not been accessed yet can not do much harm. Each Hypertext Markup Language (HTML) file in the sandbox is evaluated using a JavaScript based Document Object Model (DOM) implementation, on which a set of predefined rules compute a final

[1]`https://codesandbox.io`


0t03gw window.location.replace

15uj7w3wns63a2db4609412.activekase.ru

Checking if the site connection is secure

15uj7w3wns63a2db4609412.activekase.ru needs to review the security of your connection before proceeding.

Did you know companies are using machine learning to help identify and protect against bot traffic?

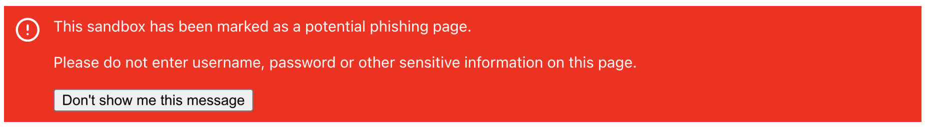
o0l03t content="0;URL=

Welcome to the Auditorium of

Campus Complex

Play Music Video

Figure 1.3: Two entries on the review dashboard.


This sandbox has been marked as a potential phishing page.

Please do not enter username, password or other sensitive information on this page.

Figure 1.4: The red banner is displayed to users until a human reviewer marks it as a false-positive.

score. This score is a natural number between 0 and 999 and tries to quantify the likelihood of a sandbox being used for phishing. By dividing the score by 10, one can think of it as a probability, where a score of 999 represents the detection mechanism being 100% certain that the sandbox is used for phishing. The number of points awarded for each matching rule is set based on experience and estimation. An example rule scans the generated HTML content and increases the score by 100 points for each form element with an "action" attribute containing the string "http". This indicates that the sandbox issues a request to an external website, which could potentially send credentials typed into the form to a third party. If the final score does not exceed a predefined threshold, the sandbox is marked as harmless and otherwise as possibly containing phishing.

Harmless sandboxes are discarded, while the potentially harmful ones show up on an internal review dashboard, where an employee manually inspects the sandbox and decides whether the classification was correct or represents a false-positive. In the former case, the employee is able to directly take action against the phisher by deleting their account and thereby all associated sandboxes. Two example entries of the dashboard are displayed in Figure 1.3. To support CodeSandbox's own detection mechanism, they collaborate with another company that can report phishing sandboxes detected by their proprietary mechanism, which also creates entries on the review dashboard. While a sandbox is pending human review, the banner shown in Figure 1.4 is displayed on the preview page alerting viewers that the site they are currently looking at possibly tries to deceive them.

However, not all sandboxes detected by the current mechanism are indeed used for phishing. Out of the 13,000 entries that were at one point listed on the review dashboard, only around 600 were actually deleted by a human reviewer for containing phishing.

If we treat an entry showing up on the review dashboard as it being classified as phishing and the decision of the human reviewer as the ground truth, the system has a False Positive Rate (FPR) of 95%. At the same time, users still report phishing incidents. The company supporting CodeSandbox in their fight against phishing reports on average three previously undetected and later confirmed phishing sandboxes per day. In a recent incident[2], Google found phishing activities on of the subdomains used by CodeSandbox to display their sandbox previews. This led to users visiting *any* preview on the subdomain through Google's Chrome browser seeing a red warning page instead, labeling it as insecure and potentially containing harmful content. Since Chrome has 60% market share, this leads to the majority of visitors of that subdomain needing to dismiss the warning in order to visit their previews. To summarize, the current detection mechanism misclassifies many harmless sandboxes as phishing, leading to more work for human reviewers. User reports and findings from external entities prove that it misses multiple sandboxes per day.

1.3 Problem Statement

This work is concerned with creating a server-side phishing detection system that utilizes machine learning (ML) technologies, manually designed features, and is operated by the website host. Given this context, the following research questions need to be answered:

RQ1 **How can the FPR be reduced?** The current method adds too many irrelevant entries to the review dashboard, likely because of its simple rule-based approach. This work should find a new ML-based approach better suited for this complex problem domain. Additionally, the new method should use higher quality information since the current one only utilizes data obtained through simulated execution of source code. By actually running sandboxes in a headless browser and incorporating metadata about the sandboxes, the algorithm has access to more and higher quality information to issue more informed predictions. Together, these improvements aim to improve the current system's high FPR.

RQ2 **What identifies a phishing sandbox?** The new features obtained through answering RQ1 are unlikely to contribute equally to the final prediction. While new and anonymously created sandboxes have an increased probability of containing phishing, a sandbox containing a login form is more likely to identify it as being used for phishing. Due to the unique position of being the website host, this could reveal influential features that were previously not used for phishing detection. Therefore this work explores the influences of the new features on the classification result.

RQ3 **How can the new mechanism still be fast enough?** Although the current method has a high FPR, it is fast and does not use many system resources. As there are close to 30,000,000 sandboxes on the platform, a new detection run is triggered about 24 times per minute. If the result of the phishing scan takes too long to complete, a user might already have

[2]`https://twitter.com/csbstatus/status/1559834311838388225`

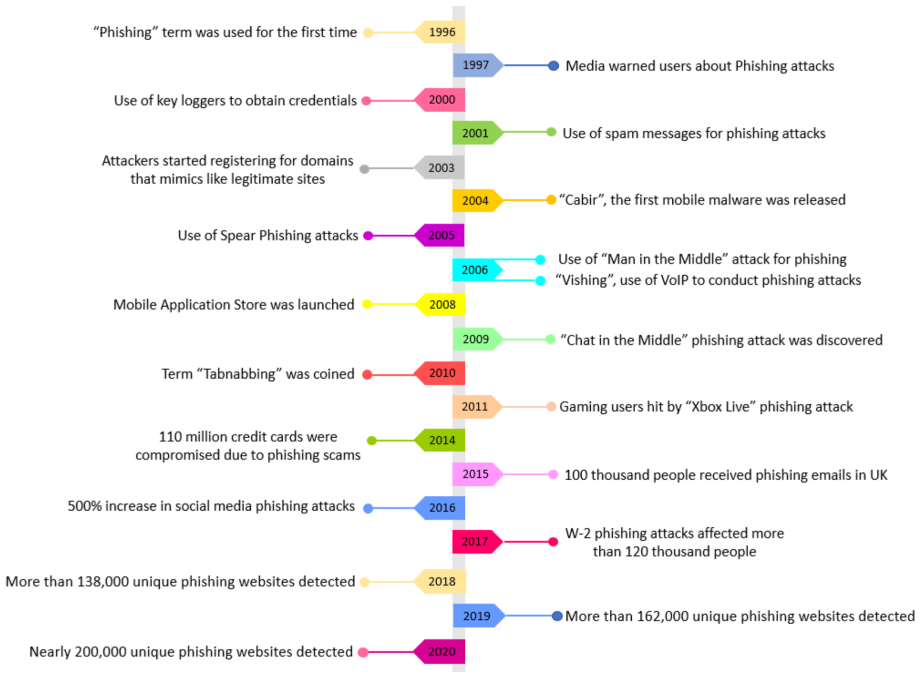
typed their credentials into a fake login form. It is therefore crucial that a new mechanism needs to be scalable and reasonably fast to execute. However, the newly introduced methods and features described in RQ1 will increase the complexity and computational cost of server-side phishing detection. An important question is therefore how to ensure that a sandbox can still be analyzed in under 10 seconds. This work tries to find a system design that realizes this goal.

RQ4 **How can the design of the review system be changed to make better use of the reviewers' effort?** Human reviewers detecting phishing need expert knowledge about web technologies in order to e.g. distinguish between a harmless replica of a popular website's login page and a dangerous one used for phishing. Because of these requirements, the developers of CodeSandbox handle the review, whose time should be used in the most effective way possible. Currently, this is not the case, as they need to invest time into reviewing obviously harmless sandboxes with no apparent signs for why they showed up on the dashboard. While RQ1 should improve this situation, there are fundamental issues with the current design of the review system that hinder its usefulness when using machine learning. Supervised ML algorithms require training data but classifying a sandbox as phishing deletes data associated with it. Since users also modify and delete sandboxes, part of the annotated labels that make up the training data are lost over time. The user interface of the review dashboard neither reflects the "danger" of a sandbox nor why it should be considered dangerous, making it hard for the reviewers to provide feedback on the automated decisions. Due to all these shortcomings, this work explores an improved design for the review system, which combats training data loss and makes the reasoning behind automated decisions more transparent to the reviewers.

In addition to these research questions, there are aspects that are intentionally left out in order to maintain a reasonable scope. Sandboxes can be used for many undesirable purposes, but this work solely focuses on detecting phishing. Other problems, like detecting spambots, sandboxes that just redirect to Google searches to gain popularity, or explicit content are outside the scope of this research.

1.4 Structure

The rest of this document is structured as follows: section 2 describes relevant background topics, such as the type of attacks used by phishers. Section 3 then lists several related works and categorizes them by their used features and methods. Section 4 describes the data model of CodeSandbox and the available data used to detect phishing. A selection of features and methods used by this work to answer the research questions are defined in section 5. The achieved results are listed in section 6 and discussed in section 7. Finally, this work is summarized in section 8, including the mention of potential future work.



"Phishing" term was used for the first time
1996
1997
Media warned users about Phishing attacks
Use of key loggers to obtain credentials
2000
2001
Use of spam messages for phishing attacks
Attackers started registering for domains that mimics like legitimate sites
2003
2004
"Cabir", the first mobile malware was released
Use of Spear Phishing attacks
2005
2006
Use of "Man in the Middle" attack for phishing
"Vishing", use of VoIP to conduct phishing attacks
Mobile Application Store was launched
2008
2009
"Chat in the Middle" phishing attack was discovered
Term "Tabnabbing" was coined
2010
2011
Gaming users hit by "Xbox Live" phishing attack
110 million credit cards were compromised due to phishing scams
2014
2015
100 thousand people received phishing emails in UK
500% increase in social media phishing attacks
2016
2017
W-2 phishing attacks affected more than 120 thousand people
More than 138,000 unique phishing websites detected
2018
2019
More than 162,000 unique phishing websites detected
Nearly 200,000 unique phishing websites detected
2020

Figure 2.1: Historic overview of events related to phishing from 1996 to 2020 [3].

2 Background

This section includes relevant background knowledge about topics related to phishing and its prevention. First, phishing is defined and examples are shown of common and clever attacks. Then the problem of classifying websites is defined. The last subsection describes techniques for incorporating humans into machine learning processes and how to use their work in the most effective way.

2.1 Phishing

As seen in Figure 2.1, phishing has historically been a problem since the creation of email and the internet. With technological advancements, phishing strategies also got more creative and new record were set highs. In general, all phishing attacks work by making the user think they visit a trusted web page or communicate with a person they trust. Email, SMS, and more recently social networks like Facebook, Instagram, or YouTube are used for phishing. The attacker impersonates another authority, prompting the user to send them money, credentials or otherwise doing harmful actions.

2.1.1 Types of Phishing Attacks

Since phishing is a social attack, it can be executed in various technological ways. A phishing attack should deceive many users with a high probability to maximize its impact. Therefore, *spear phishing* or *whaling* [8] attacks focus on increasing the probability of a single person being deceived, at the cost of the target audience being smaller. This necessitates the attacks being more personalized to the victims to appear more credible.

An example is presented by Siadati, Nguyen, and Memon [8], who showed X-Platform Phishing, where they used the ability to trigger transactional emails like social media friend requests with customizable content containing hyperlinks leading to potentially harmful websites. As these emails are sent by an authority trusted by the user, they are more likely to visit the linked website. In their experiments using the GitHub platform, all emails containing phishing links were delivered to the victims inbox, sometimes even forwarded to others and not caught by any spam or phishing filters. As the results demonstrate, this attack is hard to detect, but requires the attacker to address the users directly, making it harder to quickly distribute the emails to a large number of users.

An important part of most phishing attacks are fake websites, which replicate the look of a popular one, but are controlled by the attacker. In these *website forgery* [9] attacks the phisher makes the user think they visit familiar site. In reality, they are served a visually similar copy that sends their data to the attacker. According to the APWG [2] and phishing datasets [10], popular targets for such attacks are financial websites such as PayPal or online banking providers. In general, a website forgery attack has the following lifecycle [9, 11]:

1. The attacker sets up a website similar to an existing one.

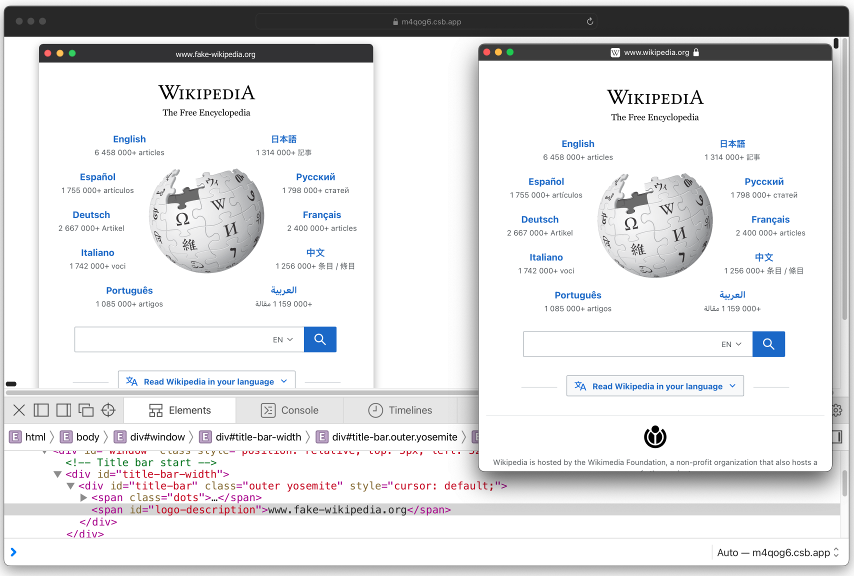
2. The attacker somehow makes the victims aware of the phishing website. This can be done in a variety of ways, such as linking to it from a mail that is sent out to the victims, a comment on a social media website or an SMS.

3. Victims fooled by the phishing website enter their credentials, thinking they are interacting with its original counterpart. In reality, the credentials are transferred to the attacker, thereby compromising them.

4. The phishing website gets detected by entities. This could be the hosting provider, or another third-party, such as Google, who wants to warn the users of their Chrome browser of potentially harmful websites. A recent study [11] states that it takes nine hours on average for these third-parties to detect a phishing website from the point where the first victim is deceived. During this timeframe, the majority of the damage has been done, since 62.73% of the victims have been deceived.

5. The website is finally taken down by the hosting provider as it was detected as harmful. Alternatively the attacker could be motivated to remove the site themselves, since the cost of hosting is not worth it, as external entities already marked is as phishing, thereby lowering its efficiency.

The URL is an important part of a webpage, that an attacker cannot just reuse to appear legitimate. However, there are techniques that allow them to look official, such as slightly altering a commonly known one. A user might not catch a missing “e” in `appl.com`, if the website looks convincing enough. Similarly, “vv” can look very similar to a “w”, depending on the kerning, tracking and the chosen font of the browser. These techniques are called *homoglyph* or *homograph* attacks [12]. They abuse the fact, that there are similar looking characters in different alphabets. As an example, the letter “a” (U+0061) has a very similar



```
    <!-- Title bar start -->
  ▼ <div id="title-bar-width">
    ▼ <div id="title-bar" class="outer yosemite" style="cursor: default;">
      ▶ <span class="dots">…</span>
        <span id="logo-description">www.fake-wikipedia.org</span>
      </div>
    </div>
```

Figure 2.2: A fake popup window (left) built with HTML[3] to look and feel like a native macOS one (right). While there are some subtle differences, phishers could add trust signifiers like the currently missing padlock or change the URL.

looking counterpart (U+0430) in the *Cyrillic* Unicode block. Unicode characters from blocks other than the Basic Latin one need be encoded in URLs, so `apple.com` using the Cyrillic "a" would become `xn--pple-43d.com`. However, browsers choose to display the Cyrillic letters instead, as it is better readable to speakers of languages using such alphabets. Even though measures against this have been implemented, to e.g. display the escaped version if a URL uses letters from multiple Unicode blocks, this attack vector has been abused in the past and continues to represent a tradeoff between security and user experience.

These types of attacks can be detected by comparing the similarity of the current URL to a list of known ones. This is not the case, if the browser window is entirely rendered by the web-page and controlled by the phisher. This is the case for *Picture-in-Picture* (or *Browser-in-the-Browser*) attacks [13] where a popup window is replicated using HTML and JavaScript. Based on the browser's user agent string, the attacker can infer the operating system and therefore make the replicated user interface look like a native one. As a consequence, the URL or other trust signifiers can be arbitrarily controlled by the attacker. An example is displayed in Figure 2.2, where the fake popup on the left looks very similar to the native one on the right. The developer tools at the bottom show the HTML structure used to build the fake one and reveal that the displayed URL can be set to an arbitrary string chosen by the phisher. The user can still notice the difference by e.g. trying to drag the fake popup outside the original browser window, which is only possible for real ones. A study by Jackson et al. [13] showed, that these types of attack are as effective

[3]`https://github.com/mrd0x/BITB`

as homograph ones.

2.2 Classification

Detecting phishing attacks is a binary classification problem, where a non-phishing site is associated with the *negative* class and a phishing one with the *positive*. Some related works are able to achieve high accuracy scores of more than 95% [14, 15, 16]. However, as the web is constantly changing, so do phishing attacks and relevant features. Methods and features that achieve high accuracy now, may not do so as time passes, and browser introduce new APIs. Menon and Gressel [17] showed, that this *concept drift* needs to be taken into account, otherwise model performance will worsen over time.

When a phishing classifier is built, it needs to be determined when and where it should run. Most implementations from related works run on the client-side, in response to a user opening a webpage. Some browsers support this out of the box, e.g. Google's Chrome browser uses their Safe Browsing service [18] to detect when a potentially harmful website will be visited. Other techniques from research [4, 5] can be installed as a browser extension, to enhance the built-in detection mechanisms. However, client-side protection only works for users who installed it. In an ideal scenario, fraudulent websites could be detected, *before* they can harm any users. To realize this, Maurya and Jain proposed shifting the detection to the Internet Service Provider (ISP) [19]. CodeSandbox, as a website host, is in a similar position to implement server-side detection on behalf of their users.

2.3 Cloaking

Phishers use techniques referred to as *cloaking* to evade detection. A generic example is making their websites look harmless when scanned for phishing but still displaying a fake login form when a victim visits the website. The authors of the CrawlPhish [20] paper analyzed different cloaking techniques and their usage in the wild. Suppose a server-side phishing scanner uses the Chrome browser to analyze the website. In that case, phishers can analyze the user agent header to block all requests by Chrome users or serve them a harmless website instead. Similar server-side cloaking techniques can be implemented based on the requests' IP or included cookies. They also describe client-side detection techniques, such as requiring the user to move their mouse to display the real page. Humans express such subconscious behavior but not by phishing crawlers. The authors of the PhishFarm [21] project described similar patterns and showed that a phishing site using cloaking takes almost twice as long to be detected by popular blacklists than one without.

```javascript
const parameters = parseQueryParametersToObject();
const email = parameters["email"];

if (email) {
    window.location.replace(`https://phishing.com/?e=${email}`);
}
```

Listing 1: JavaScript code that only redirects the user if a specific query parameter is present. The value of the query parameter is then passed on to the phishing site to pre-fill the email field in the displayed login form, making it seem more trustworthy.

The most prominent example of cloaking on CodeSandbox is through query parameters. As displayed in Listing 1, the sandbox redirects the visitor if a query parameter is present or has a specific value. Sometimes this is done explicitly using an if-statement and sometimes as a byproduct of using the `atob` function to decode a base64-encoded string, which fails if the passed value is `undefined`. More advanced methods redirect the user, but the target page validates a token passed through a query parameter. If that token does not have a valid value only known by the phisher and the victim, the target website displays a harmless version. This effectively blocks human reviewers from adequately accessing the danger of the website. Other forms of cloaking are placing a static phishing HTML file inside an otherwise compiled web app. The phishers then deep-link to the phishing file, which has to be found by human reviewers, making it harder to detect. Finally, CAPTCHAs are used in a few cases to prevent automated access to phishing sandboxes. While this is less common than the other phishing patterns, it prevents automated phishing scans.

2.4 Human-in-the-loop Machine Learning

While machine learning models should replace manual human processes, humans are not completely eliminated from the task. Many algorithms require labeled data points created by humans to work and tasks like quality control or feedback on classification results also require human effort. This section describes related research that deal with human-machine-learning interaction.

2.4.1 Human Guided and Interactive Machine Learning

While machine learning models can help to automate manual tasks, no classifier achieves a perfect accuracy of 100%. Sometimes, they are wrong. During the training process, the model can adjust its weights based on the correct label provided by a human expert. However, after the model is deployed to production, it is expected to make a decision on its own regularly. But if the model is only 50% certain that a website is used for phishing, should it be deleted automatically or left as is? Another option would be to delegate the final decision to a human expert. *Human Guided Machine Learning (HGML)* [22] (or simply *Guided Machine Learning* [23]) and *Interactive Machine Learning* are research areas for interactively incorporating humans into the machine learning lifecycle.

Examples from other domains than phishing include search engines, where the end-user can provide feedback to the search results [24]. The model can then use this feedback to rank the search results and improve over time. Human reviewers already vote on the results of the phishing detector at CodeSandbox, the feedback is just not incorporated back into the model. Another example is ManiMatrix [25], a system where users interactively adjust parameters of a machine learning process based on the resulting confusion matrix. Thereby users can control their decision preferences, which can vary between use cases. Classifying a regular email as spam is likely to have worse consequences than the inverse [24].

2.4.2 Explainability

When using a machine learning algorithm to solve a classification problem, it can be hard to explain the result. Simple methods like linear regression can use the coefficients to derive an explanation for how much each feature contributed to the final result. Decision trees can be visualized and, depending on the depth, make it easy to follow their decision as a human. The same holds for rule-based systems, like CodeSandbox's current approach, where one could simply list the contributions of each rule to the final score. More complex methods like neural networks or an ensemble of decision trees can lead to more accurate results but are not as easy to explain and are often treated as black boxes. This can lead to models with good classification performance, which may focus on unintended details such as the presence of text instead of the actual subject of the image [26]. Such mistakes are hard to spot without transparency about what the model focuses on and how specific feature values influence its decision.

Post-hoc explanation methods [27] are created after the model has been trained. They provide an idea of a feature's influence without requiring the classification algorithm to be inherently explainable, such as a decision tree. Permutation importance [28] can be computed by simulating the absence of a feature and measuring how much the accuracy decreases. This covers the overall influence but fails to explain a particular decision. Local Interpretable Model-agnostic Explanations (LIME) [29] tries to simplify a complex model by assuming that a local part of the decision boundary can be approximated by a simple linear model. These much simpler models can then be used to derive an explanation. SHapley Additive exPlanation (SHAP) [30] and its adaptation for tree-based algorithms TreeSHAP [31] use an additive model to represent the attributions of each feature to the final classification result. For regression models, this has the additional effect that the sum of the expected value and the SHAP values of each feature is equal to the regression result. When computing a risk score for a sandbox containing phishing, one can see numerical attributions of each feature on the final score.

2.4.3 Sampling

When incorporating humans into machine learning processes, one must decide which data points should be handled automatically, and which ones need to be manually reviewed. If reviewers were presented with a random sample of the data to classify, it is likely, that the model could have classified many of those data points correctly [32].

Uncertainty sampling [34] represents a family of sampling algorithms that select the data points that a model has the most trouble identifying correctly.

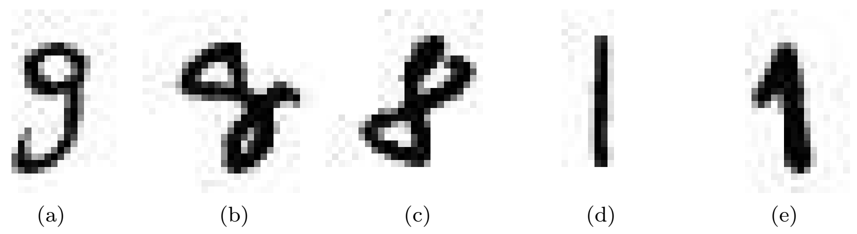


Figure 2.3: Samples from the MNIST dataset [33]. While (a) and (c) are easy to classify as either 9 or 8, the results for (b) will likely be less certain. Images (d) and (e) show different styles of writing the digit 1. All images were inverted to improve legibility.

The digit displayed in Figure 2.3 (b) could either be classified as an 8 or as a 9, making a classification result made by a model less certain. When employing uncertainty sampling, the model would try to detect the displayed number in each image while recording a measure for the certainty of the decision. A human would then manually label a number of uncertain data points and retrain the model. Das Bhattacharjee et al. [35] applied this technique to phishing detection based on URL features. They gradually increased the number of manually annotated samples, each time tracking the classifiers' accuracy. At each step, the newly introduced samples were chosen through uncertainty sampling. The results show that the classifier's performance stabilizes after approximately 1% of the dataset is annotated.

Another challenge when only looking at a subset of the overall dataset, is to obtain a diverse sample. Otherwise, the model might overfit on the chosen data points [36], thereby lowering its accuracy on unseen data. Imagine a sample from the MNIST dataset [33], where the digit "1" is written as a straight line, as seen in Figure 2.3 (d). A classifier trained on this sample is likely to mistake a "1" in the style of (e) as a "7". To obtain a more diverse sample, the general strategy is to find groups of data points that are similar. Concrete implementations leverage techniques such as outlier detection [36], clustering [37] or regression [32]. Diversity sampling can also be combined with uncertainty sampling, to obtain a subset that helps the model learn fast, while still generalizing well [38].

	Method	Setting	Features	Screenshot	Popularity	Text	DOM	URL	Phish.	Leg.	Acc.	FPR
[39]	Rule-Based	C	8			✓	✓	✓	100	100	95.0%	10.0%
[40]		C	2	✓		✓			1,000	200	96.1%	1.4%
[41]		E	6		✓			✓	9,661	1,000	97.0%	2.0%
[42]		C	1				✓		-	-	-	16.9%
[43]		E	1	✓	✓				100	100	99.0%	0.0%
[44]	ML	E	15		✓			✓	4,883	8,118	-	0.4%
[45]		E	30	✓	✓		✓	✓	615	489	98.3%	-
[46]		E	212	✓			✓	✓	1,216	100,000	95.6%	0.0%
[47]		E	17				✓	✓	800	600	92.1%	-
[48]		E	12				✓		1,428	1,121	98.4%	1.5%
[49]		S	1	✓					193	347	95.2%	-
[4]		S	22		✓			✓	1,473	1,500	92.5%	5,4%
[50]	DL	E	9					✓	24,539	24,502	97.0%	2.6%
[51]		E	1					✓	8,796	9,000	98.3%	1.7%

Table 1: Overview of the method, features, and results of related works implementing phishing website classifiers. The *Setting* column describes whether the method is intended to be run on the server (S), the client (C), or was of experimental nature (E). In an experimental setting, the detection can run on the server side, but the authors neither mention this explicitly nor describe real-world challenges, such as time or resource constraints. If known, the amount of phishing (Phish.) and legitimate (Leg.) samples in the training set is displayed, as well as the accuracy (Acc.) and FPR. When a work compares multiple methods, only the best metrics are listed.

3 Related Work

This section describes existing phishing classifiers from academic literature. Their methods, features and datasets are described and broadly categorized. A summary is displayed in Table 1.

3.1 Methods

The methods used to detect phishing evolved over time from being simple and requiring manual effort, towards being more complex, resilient to changes and automated. They can be broadly categorized into blacklists, rule-based, machine learning and deep learning approaches, which will be described in this section. These categories are not exclusive and can be combined to leverage the advantages of another method.

3.1.1 Blacklists

The first thing a user interacts with before they visit a web page is usually its URL. It uniquely identifies a website, can not be controlled by an attacker and is therefore valuable for detecting phishing websites. In the most simple way, a URL is checked against a *blacklist* of sites known to contain phishing. Examples include PhishTank [52], OpenPhish [53] and Google Safe Browsing [18]. How sites get blacklisted differs between providers. PhishTank allows its users to submit entries, which are then voted on by others. The other two do not publicly state how the detection works, but both expose ways to report false positives or suspicious websites.

A study from 2007 suggests that list-based methods are able to detect up to 90% of phishing websites [1]. It is to be noted that they sourced their phishing URLs from PhishTank, which is available for free. The evaluated lists might use PhishTank as a source, thereby possibly inflating the measured accuracy. More recent studies [54, 55] show similar results, with Google Safe Browsing being reported as the largest and most precise list. However, when using more advanced evasion techniques can be circumvented leading to worse performance. Oest et al. [21] showed that Google was able to detect 97% of non-evasive phishing sites. When they implement techniques to prevent detection, such as blocking US traffic or IPs known to be used by Google's crawlers, only 1–3% of websites were detected. This illustrates that while using a blacklist may be simple to implement, they still have their drawbacks. Their effectiveness is dependent on how frequently they are kept up to date and literature suggests that it takes hours rather than minutes for new sites to be detected [21]. This means they are less effective against zero-day attacks, compared to other methods which actively scan for phishing content.

3.1.2 Rule-based

In order to be independent of third parties, one can manually implement rules that if satisfied to a certain degree, classify a website as phishing. These rules or heuristics are similar to features in machine learning. The impact of each rule on the classification result is defined manually, as it is done by CodeSandbox's current system described in section 1.2. This is a clear drawback of such systems, since the influence of a feature can change over time [17], requiring continuous adjustments in order to maintain a good classification accuracy. An example from academic literature is given by Nguyen et al. [41], who were able to achieve 97% accuracy and a FPR of 2% with only six rules. They queried search engines for the domain and path of the website and measured the Levenshtein distance to the returned spelling suggestion. This way they intended to detect phishers using homograph attacks. They also used Alexa to determine scores for popularity and reputation, since phishing sites are less frequently visited than their counterparts. All these metrics were combined into a weighted sum that if it exceeds a threshold leads to the site being classified as phishing.

3.1.3 Machine Learning

As an evolution to the manually created rule-based systems, ML techniques are used in combination with features derived from the URL or from the pages content. This way, one does not need to manually create rules and let the algorithm decide how to best utilize the features. Shahrivari, Darabi, and Izadi compared 12 commonly used algorithms throughout phishing detection research [45, 3]. Those include Logistic Regression, Decision Tree, Random Forest, K Nearest

Neighbor (KNN), Support Vector Machine (SVM), Neural Network and various boosting algorithms such as XGBoost or Gradient Boosting. The resulting accuracy scores range from 92% to 98% with the ensemble methods XGBoost and random forest achieving the overall best performance. As all classifiers were able to detect phishing websites with high confidence, one can choose one depending on the specific problem. If explainability is important for the use case, one can sacrifice a bit of accuracy, and implement a decision tree classifier, whose predictions can be easily visualized as a diagram. Similarly good results were achieved by Jain and Gupta [48] who also tried several different ML classifiers. They used 12 different content-based features mostly focusing on URLs contained in attributes of HTML like the ones from anchor tags, forms or loaded Cascading Style Sheets (CSS). Using a balanced test set of phishing and legitimate sites and a logistic regression classifier, they were able to achieve an accuracy of 98.4% while maintaining a low FPR of 1.5%. This approach sounds promising for this work, since it is proven to work well with a reasonable amount of features and computational cost.

3.1.4 Deep Learning

Through the use of specialized machine and deep learning methods, several authors have tried to reduce the maintenance burden induced by having to constantly respond to new types of phishing attacks. While traditional machine learning algorithms improve upon rule-based systems, they still require the effort of manual feature engineering. *Deep learning* based methods try to circumvent this, by inferring important features from raw data. This way, they are also more resilient towards change and are able to deal with concept drift and the introduction of new browser APIs [17]. For example, the introduction of the global `fetch` function to JavaScript supplied phishers with a new way of issuing requests to external servers. Traditional machine learning models created previously need to be made aware of this fact, or otherwise their false negative rate will likely increase over time, as they miss phishing attempts using this new API. Deep learning based methods can be retrained on phishing pages abusing the new API, thereby circumventing the need for new feature engineering. The gained advantages come at the expense of requiring larger datasets and more resources while training [3].

A recent implementation leveraging deep learning is PDRCNN [50]. Based solely on URL features extracted by a Long Short-Term Memory (LSTM) network, which are then passed to a Convolutional Neural Network (CNN), they achieve 97% accuracy on 500,000 websites crawled from Alexa and PhishTank. The approach followed by CNN-MHSA [51] also works solely based on the URL, but requires no explicit feature engineering. Each letter of the URL is one-hot-encoded based on the valid 84 characters it can contain. The resulting matrix is then passed through a CNN containing a multi-head self-attention (MHSA) layer that should discover relationships between the characters. This method yields an accuracy of 98.3% with a FPR of 1.7%.

3.2 Features

There are numerous signifiers indicating that a website is used for phishing. Marchal et al. [46] alone use 212 different features in their implementation. In

$$\underbrace{\text{https}}_{\text{protocol}}://\underbrace{\text{accounts}}_{\text{subdomain}}.\underbrace{\text{paypal}}_{\text{domain}}\underbrace{\text{.com}}_{\text{TLD}}\underbrace{\text{/secure/login}}_{\text{path}}\underbrace{\text{?key=secret}}_{\text{query}}$$

Figure 3.1: Selected parts of a URL.

general, the features are derived from the URL, the content of the page or other metadata about both. This section lists popular examples found in literature and describes them in more detail.

3.2.1 URL-based

Since URLs uniquely identify a website, data extracted from them is prominently used in research. Some signifiers, such as popularity, can also be derived from other features making them not exclusive to the URL. To put URL-based works into perspective, Althobaiti, Meng, and Vaniea conducted a study [56] to find out how well humans can detect phishing when presented with URL-based features. They used 7 different features derived from the URL and the subjects achieved on average an accuracy of 91.6% with a FPR of 12%.

The *protocol*, so whether the URL begins with “http://” or “https://”, signifies whether the traffic is transferred in plain text or encrypted. In the former case, all credentials entered by a user will be visible to anyone monitoring the networks traffic. This is not the case for the latter, which requires the website host to obtain a possibly paid Secure Sockets Layer (SSL) certificate. As this means extra effort for phishers, it could be used as a detection mechanism [1, 47]. However, over time browsers started marking websites making use of unencrypted forms as insecure rendering this feature less reliable [11]. Advances in technology and the widespread availability of free SSL certificates amplified this effect.

Other features are derived from the text of the URL. Unusual lengths, such as very short or very long ones are used by phishers to confuse users or hide parts of it [50]. For similar reasons, an unusual number of dots or other special characters is seen as suspicious. By utilizing Natural Language Processing (NLP) techniques such as word embeddings, the used language can also be processed by machine and deep learning models. [47, 45]

Some features can be derived from the metadata about the URL. Algorithms like PageRank determine the popularity of a website based on its incoming and outgoing links. A phishing website imitating PayPal’s login form is likely to have a drastically smaller number and less popular sites that link to it compared to the real one [44]. Due to the short lifespan of a phishing website, the domain ownership duration can also be used as an indicator [4, 56].

3.2.2 Content-based

While using URL-based features works well, solely relying on them has its downsides. If the phishing site is hosted on a free provider, which make up a significant portion of phishing attacks [57], the URL is determined by the host and therefore not informative. Users also have trouble judging the legitimacy of a website based on the URL [56]: the user is required to know its canonical version and link shorteners such as `bit.ly` make it impossible to know the real

```html
<script>
  // This redirects the current page to other-website.com
  window.location = "https://other-website.com";

  // This has the same effect, but will not be detected by
  // statically scanning for "window.location ="
  const w = window;
  const l = "location";
  w[l] = "https://other-website.com";
</script>
```

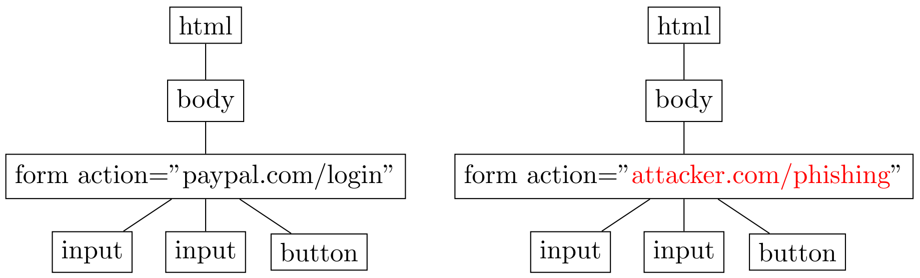
Listing 2: Two different ways of redirecting the current page using JavaScript.

destination of a link. It is therefore also important, to search the *content* of the page for phishing signifiers.

One of the first projects to detect phishing websites based on their content is Cantina [39]. They used the term frequency-inverse document frequency (TF-IDF) algorithm to identify signature terms of the website, then search for them on Google to find its canonical URL, and finally compare the URLs from the search results to the one from the website in question. Additionally they incorporated URL-based rules such as checking whether it contains special characters or IP-adresses to reduce the number of false positives. Once these rules are evaluated, a final classification decision is made. This way Cantina is able to achieve 95% accuracy and a 10% FPR. This work has been iterated upon since, for example Cantina+ [44] lowering the FPR to 0.4% by using more features, such as how likely it is that the website represents a login form and using a machine learning based method.

As the source code of a website can easily be read, it is also often used to derive phishing features. For example, a phishing website is likely to load its favicon, a little image displayed next to the page's title in a browser tab, from the canonical version of the website. This way it is always up-to-date, making it more resilient to changes. Therefore, if the favicon is loaded from a domain different from the one of the website, it could mean that the website could be used for phishing [48]. Similarly, JavaScript can be used to e.g. hide the target of a link, change the URL in the address bar without actually navigating away, or directly redirect the page [49]. In the past these have been used as features by scanning the page's source code, but due to the dynamic nature of JavaScript, these scanners can easily be avoided. An example is illustrated in Listing 2. While the first example is caught by static code scanners, the second one needs to be evaluated to detect a redirect and is not able to be caught by static analysis.

The overall DOM, the tree of HTML tags that make up the page, can also be used to detect phishing replicas. The easiest way to copy the style of a website is to just copy its HTML code and make some adjustments to send credentials to the phisher. By comparing the nodes and attribute values of the DOM tree to ones of popular websites, Rosiello et al. [42] were able to identify almost all phishing websites, with a false-positive rate of 16%. This falls under the rule-based category, with only one rule and a threshold for the calculated similarity


html
body
form action="paypal.com/login"
input
input
button
html
body
form action="attacker.com/phishing"
input
input
button
Figure 3.2: Parsed DOM tree of a legitimate website (left) that has been copied and modified by an attacker (right).

to the similar page. Figure 3.2 shows a simplified example that is detected by this method.

However, only relying on the textual content extracted from the HTML can be exploited by attackers by using images of text instead of actual text in their phishing websites. While this may be detected by users and make them question the legitimacy of the website, it is not immediately obvious without any interaction. To combat this, implementations such as Goldphish [43] use website screenshots and extract its text through Optical Character Recognition (OCR) algorithms. This way, the signature terms are based on what the real users see instead of the HTML, which may contain text that is not actually visible. Prominent regions, such as the websites' logo, are treated as more significant, similar to how a user perceives them.

The image data of website screenshots can also be utilized to create a measure of *visual similarity* to existing popular websites. Implementations following this approach usually keep track of a set of canonical websites with an accompanying screenshot. This introduces the overhead of needing to maintain a continuously updated set of canonical websites and their screenshots. By spending this additional effort, one gains the ability to quickly react to zero-day attacks against a specific website, by simply adding it to the set. If a website under question looks similar to one in this set, it gets flagged as phishing, as it is assumed the similarity is intended to fool users. Afroz and Greenstadt [40], stored a feature vector in addition to the screenshot, measured visual similarity through the Scale-Invariant Feature Transform (SIFT) algorithm and achieved an accuracy of 96.1% with a FPR of 1.4%. Another more recent implementation leveraging deep learning is VisualPhishNet by Abdelnabi, Krombholz, and Fritz [58]. They were able to match 81% of phishing pages to their canonical counterpart, solely based on visual similarity computed by a triplet CNN.

3.2.3 Other

In traditional website phishing detection, the authors of the phishing pages are typically not known to the public. However, this is not the case when detecting phishing on social media platforms. Aggarwal, Rajadesingan, and Kumaraguru detected phishing tweets in their Phishari paper [4] using a mix of URL based, tweet content, and metadata features. Besides URL-based features, metadata such as the age of the account or the ratio of accounts followed by the account and the number of its followers were amongst the most informative features. Similar results were obtained by Lee, Caverlee, and Webb [59], who also observed other platforms than Twitter. They came to similar conclusions, but showed that the account age is far more informative than social features.

3.3 Datasets

The UCI Website Phishing Data Set [60] is used by many works. As it contains features of phishing, as well as legitimate websites. Since this dataset provides derived feature data, implementations using it can be compared to each other. However, the dataset was created in 2014. Since the web evolves at a rapid rate, the importance of some contained features can diminish over time [17], making the dataset less useful.

Instead of using an off-the-shelf dataset, many researchers create their own by crawling the previously mentioned public blacklists. PhishTank [52] is publicly accessible and seems to be more popular than the lesser used OpenPhish [53]. This has the advantage that researchers can derive their own features. At the same time, this makes it harder to compare implementations, since they are using different datasets. As a compromise, the PhishMonger [10] project created a dataset by crawling active phishing websites from PhishTank from 2015 to 2018. It contains the raw HTML files, as well as other files requested by the documents, such as the CSS, JavaScript or image files. Because the source code is openly available, it can be used to create similar, more recent versions without needing to reimplement the scraping logic.

3.4 Contributions of This Work

The main contribution of this work is that it shows how to proactively detect phishing on the website *host* instead of experimental settings or the client side. This approach works without any action required by the users and utilizes features only known to the website host. Knowing when a website owner pushes new changes enables immediately triggering a phishing scan. Detailed account information can help distinguish regular users of the platform from suspicious ones, something that historically relied on publicly available information. Controlling the incoming and outgoing network traffic enables deferring the phishing scan until a user visits the website. Academic literature does not describe this approach, even though it has unique advantages compared to e.g. running a browser extension on the end-users system. This work also discusses the challenges faced with this type of detection, such as balancing computational costs with security benefits and the vulnerability to cloaking. Finally, it uses a holistic approach and describes the phishing detection from the end-user to internal review systems that enable continuous improvements to the automated detection process.

4 Dataset

The dataset is built up from several sources and consists of real-world data from CodeSandbox. The feature data is extracted from the operational PostgreSQL database, which stores data used to run their primary user-facing application. The existing phishing detection stores the results in a MongoDB. Out of the 30 million public sandboxes, 600 contain detected phishing websites. While the actual number will likely be larger, it shows that the dataset is intrinsically imbalanced towards harmless sandboxes.

4.1 Operational Data

Data in the operational database is distributed across several tables connected by foreign keys. The main entity for the phishing context is the *sandbox.* It stores the primary identifier, a user defined title and description, the foreign keys to other tables and metadata such as creation timestamps and whether the sandbox was created in a manual or automated manner. In order to uniquely identify a sandbox together with its contents, a version field is incremented on every edit of the sandbox.

Unauthenticated users can create up to three sandboxes before they are required to create an *account.* These are linked to sandboxes and contain valuable feature data. Users can sign up either through their GitHub or Google account. Next to the usual username, email and name fields, CodeSandbox stores the creation date and the time the user was last active on the platform.

Tables required for social features connect both previous entities. It is known which user liked which sandbox, how many times a sandbox was shared or forked, and who created comments. Similarly, the number of times a sandbox was viewed is tracked for a given timeframe.

The files in the sandbox are also stored in the database, along with their contents. The directory path of a HTML file corresponds to its path on the website, thereby controlling parts of the URL it will be served at.

4.2 Phishing Data

The current phishing detection process described in section 1.2 stores all its data in a single MongoDB collection. Each item in this collection stores a sandbox ID, a score calculated from the rules, a string describing which rules were matched and a checksum of the rules available at the time. The last field is needed in order to invalidate an existing score, after new rules were introduced into the system. Reviewing entries on the dashboard then sets flags whether the sandbox was falsely flagged, or indeed phishing.

Since February 2022, 5.8 million scans were triggered. Most of the scans receive a score of zero, meaning none of the configured rules were matched. Only 250,000 scans yielded a non-zero score, with an average 500. As Figure 4.1 shows, the number of sandboxes with a non-zero score decreased over time, which can be attributed to less phishing activity or adjustments made to reduce the number of false positives. On an average day, 150 non-zero scores were computed, out of which 73 exceeded the configured threshold, leading them to be displayed on the review dashboard. Around 36% of the reviewed dashboard entries were marked as a false-positive. However, only 8% of the dashboard

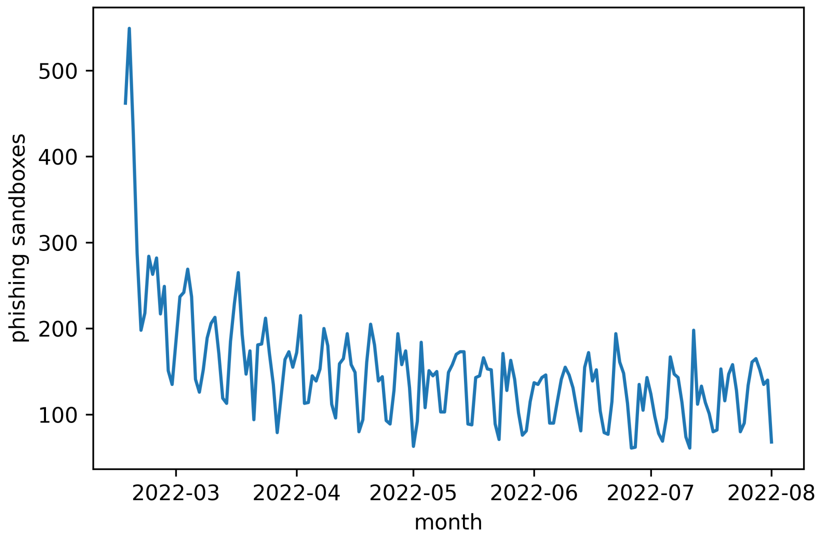


Figure 4.1: Number of daily detected sandboxes with a non-zero score over time.

entries were actually voted on. As the sandboxes on the review dashboard are sorted by date and score, the most recent and most likely dangerous sandboxes are displayed first. This seems to lead to most harmless entries just being ignored, thereby being pushed to the end of the review queue. Currently, there are roughly 11,000 unreviewed entries in the phishing database with a high enough score to be displayed on the dashboard.

4.3 Training Data

In order to detect phishing with supervised machine learning techniques, a dataset is required containing labeled examples for harmless and phishing sandboxes. The data in MongoDB is used to obtain an initial set of labeled sandboxes. Each label belongs to a sandbox id, its version, and the path it was opened from by the user to uniquely identify its contents as seen by the user at the time of labeling.

It contains around 1000 entries deleted for phishing activities. Unfortunately, one can not distinguish between instances deleted by the regex-based filter and ones manually annotated by a human reviewer, so the dataset will contain some amount of false positives. Entries with a low score are selected as examples for the harmless class. While they could contain undetected phishing due to shortcomings of the current detection mechanism, there are far less phishing sandboxes than harmless ones. Therefore the probability of false negatives in the training data influencing the learning process is assumed to be low enough that this source of errors can be neglected. Using this method, around 5800 harmless examples are randomly sampled.

In addition to the randomly sampled instances, a few edge cases are included to combat bias in the phishing samples. Since phishing is commonly used to harvest user credentials, the number of phishing samples containing login forms is higher than the one of harmless ones. While the presence of a login form increases the possibility of the sandbox being used for phishing, it is not a sound proof. However, if all samples containing login forms in the training set are labeled as phishing, the model is likely to overfit on this correlation. To prevent this, harmless login form replicas are included as well. Utilizing the search feature of CodeSandbox, the first 200 results matching the query “login form” are included as examples of the harmless class.

All in all, applying the described sampling techniques leads to a dataset consisting of 6995 samples.

5 Methodology

This section describes a phishing detection system that answers the research questions. The described system differs from related works since it runs on the website host instead of the client's device. Therefore it can utilize different features that have not been used before and protect end users without any actions needed on their part.

5.1 Improving the FPR

5.1.1 New Scanning Approach

When running detection client-side, the detector only needs to scan the website the user is visiting. From the website host's point of view, a user could visit an unknown number of paths on the website, all of which could potentially contain phishing. Maybe visiting `/`, `/about` and `/terms-of-use` all return harmless content, but the phisher sends out links leading to a specific phishing sub-page at `/login` collecting user credentials.

Even if the server knows all possible paths to all sub-pages, their content can still change through JavaScript. Consider the example of a minimal page containing a button labeled "Login". A human user will likely press the button, leading to a login form being built up dynamically through JavaScript. This intuitive behavior represents a challenge to automate. How can an algorithm detect that pressing the button changes the website's state and displays a phishing login form? It would either need to try every possible interaction with the website, which, given the amount of APIs available in a modern browser, is too large to try practically, or one could train an algorithm to behave like a human visiting the website. Either way represents a challenge needing to be solved since static analysis alone can not find all possible states of a web presence.

The current scanner solves this by scanning and evaluating all HTML files proactively in one run. However, phishing is not detected if the code does not reside in a HTML file. It also results in the system being vulnerable to server-side cloaking techniques, as described in the CrawlPhish paper [20]. Since it only uses a simulated DOM, it does not request images or execute JavaScript.

The new approach solves this by passing more information about the user's visit to the scanner. It enables the scanner to simulate a user's visit more closely, thereby being less vulnerable to cloaking. It also reproduces the user's request in a headless browser which should lead to more realistic results than simply parsing the HTML. In addition to the id and version of the sandbox, the request to the phishing detector also includes the path on which the user opened the sandbox. This affects performance since it increases the number of necessary scans. In CodeSandbox's case, the server access logs revealed that most users visit the root path of a sandbox, so the performance impact should be negligible. As a consequence, a scan is now identified by the id, version, and path.

5.1.2 Features

Related works show multiple possible approaches for lowering the FPR and answering RQ1. The current system mainly relies on a sandbox's source code and resulting HTML to compute the final score. As shown in section 4, CodeSandbox has much more information about sandboxes than its source files. A phisher

ID	Feature	Description
f_1	Has author	Whether the sandbox author has an account
f_2	Age of the author	Account age in days
f_3	Relative sandbox age	Age difference in days between author and sandbox
f_4	Number of views	
f_5	Number of likes	
f_6	Sandbox version	Number of changes made to the sandbox after its creation
f_7	Method of creation	If the sandbox was created automatically using the CLI
f_8	Number of files	Total number of files in the sandbox
f_9	Redirection	Whether the sandbox immediately redirects the user to a different domain
f_{10}	Required compilation	Whether the sandbox had to be compiled before any HTML was rendered
f_{11}	Compiler Error	Whether an error occurred while compiling the sandbox
f_{12}	Login form similarity	Binary classification result of a model detecting login form screenshots
f_{13}	Number of external URLs	Number of links pointing to domain other than CodeSandbox

Table 2: Overview of the classification features.

may be able to trick the current system by not using a form to send the user data to their server, but if the classifier also checks if the website visually looks like a login form, the system will be a lot harder to circumvent. Similar hints can be obtained from all the metadata, such as how many users liked or viewed the sandbox. The remainder of this section discusses the features used by the new system and why they were selected. All of them are summarized in Table 2.

Metadata Currently, platform users can create up to three sandboxes until they are required to create an account. These represent a good source for metadata features not present in traditional phishing website detection scenarios. Academic literature shows that attackers tend to avoid additional effort needed to create phishing, such as obtaining SSL certificates in the past [1]. Whether the author of the sandbox created an account or not (f_1) should therefore be relevant. If available, the age of the sandbox author (f_2) is tracked to prevent phishers from appearing more credible simply by creating an account. Related works concerned with phishing on social media platforms validate this fact, as they list the account age as one of the most informative features [4, 59]. Figure 5.1 also shows a difference between the author ages of harmless and phishing sandboxes. Additionally, the time between the account and sandbox creation (f_3) is computed to surface accounts intended to appear more credible.

Phishing detection research commonly uses the popularity of a website to determine if it is used for phishing. This work makes similar assumptions for sandboxes. Phishing sandboxes mainly abuse the preview feature, so they are

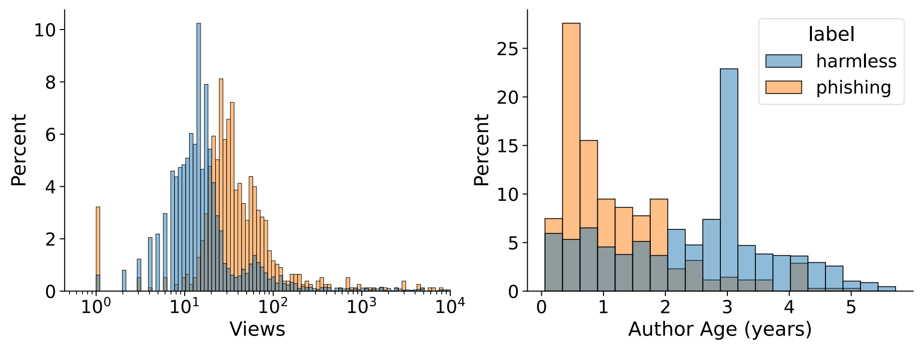


Figure 5.1: Per label distributions for the metadata features f_4 and f_2. Note that the x-axis of the "views" distribution has been cut off at 10^4 to improve legibility.

unlikely to get many views (f_4), likes (f_5), shares, or other social signifiers of popularity. While it is still possible to artificially increase these metrics by e.g. creating lots of accounts, it requires additional effort, making the platform less attractive to phishers. Mass account creation is also detrimental to the platform in general, so it is assumed to be a solved problem.

Since creating a website takes time, regular users will likely make many edits to their sandboxes. Phishers, on the other hand, are interested in creating their sites quickly since they are short-lived [11]. This might be a reason for the version column in the current phishing detection database having a median value of 1 for the version field for confirmed phishing sandboxes. Therefore, the sandbox version (f_6) is a good indicator since it reflects how often the contents of a sandbox have changed. Similar time-savings can be achieved by creating a sandbox via automated methods such as the CodeSandbox CLI (f_7). The current rule-based approach punishes large projects since the more HTML files it has, the higher the probability of one matching a rule. Since creating more files requires time and effort, the number of files in the sandbox (f_8) is included as a feature intended to prevent false positives.

All metadata features are simple to retrieve since they are stored directly in the operational database. Some values are computed at query time (e.g. the number of files (f_8) and the author age (f_2)), but the majority is selected from existing columns after joining the necessary tables together. This feature extraction part should be fast and finish after a few milliseconds.

Content-based The content-based features are available only after looking at the HTML served to the user and the behavior of the website. Especially the latter is something that the previous detection mechanism can not analyze, as it does not *run* the website. The previously mentioned new approach of evaluating the website in a headless browser enables the collection of more content-based features described in this section.

A typical phishing pattern on CodeSandbox is redirecting the user to a phishing website hosted elsewhere. The authors of such sandboxes likely exploit the short preview URLs and CodeSandbox's credibility to evade phishing detection on other platforms. As shown in Figure 5.2, almost all sandboxes in the training

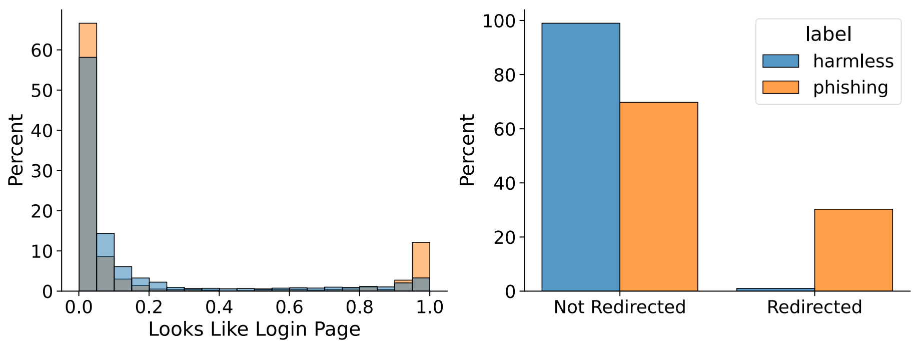


Figure 5.2: Per label distributions for the content-based features f_{12} and f_9.

set that redirected the users were also considered phishing. Therefore whether the website redirects to a host not controlled by CodeSandbox (f_9) is a reliable feature for identifying true positives. A listener is attached when running inside the headless browser to obtain this information. Once a navigation attempt is detected, the host value of the target URL is compared to the original one when opening the sandbox. If the host changes, a list of allowed ones is checked to see if the new host is owned by CodeSandbox. This step is necessary since sometimes redirection happens internally for backward compatibility. If this is not the case, the sandbox is considered to be redirected externally and f_9 is set to `true`. The distribution shown in 5.2 suggests that this is a promising indicator for finding true positives.

Since CodeSandbox is a developer-focused platform and modern web development commonly includes compiling the source code before serving it to the user, many sandboxes include a compilation step before being served to the user. When visiting a preview of a sandbox requiring compilation, the visitor sees a loading screen, which might seem suspicious to potential victims and drive them away. Phishers prefer using static files served immediately to the user, skipping the compilation and loading screen. While some harmless sandboxes also use static files, they are the minority. Therefore requiring compilation is used as a feature (f_{10}), as it is likely to reduce the number of false positives. Since the compilation step is managed by injected code, the result of it is known. If an error arises during compilation, a big red popup containing a stack trace covers the preview. It helps developers but is likely to alarm phishing victims or drive them away completely. This makes an error during the potential compilation process (f_{11}), another suitable feature to reduce the false positive rate.

Phishers often use login forms to harvest user credentials, which is why a sandbox containing one is more dangerous than others. To incorporate this risk into the model's decisions, the likelihood of it showing a login form is used as a feature (f_{12}). A screenshot of the sandbox is taken and passed to a deep learning model, which computes a number between 0 and 1. It is implemented using Tensorflow [61] and an Xception [62] network, which was pre-trained on ImageNet. The output layer consists of two nodes representing the classes "login page" and "other" using a sigmoid activation function. The value of the "login page" node is then used as the value of f_{12}.

In order to adjust it to analyze screenshots, it is re-trained on a modified

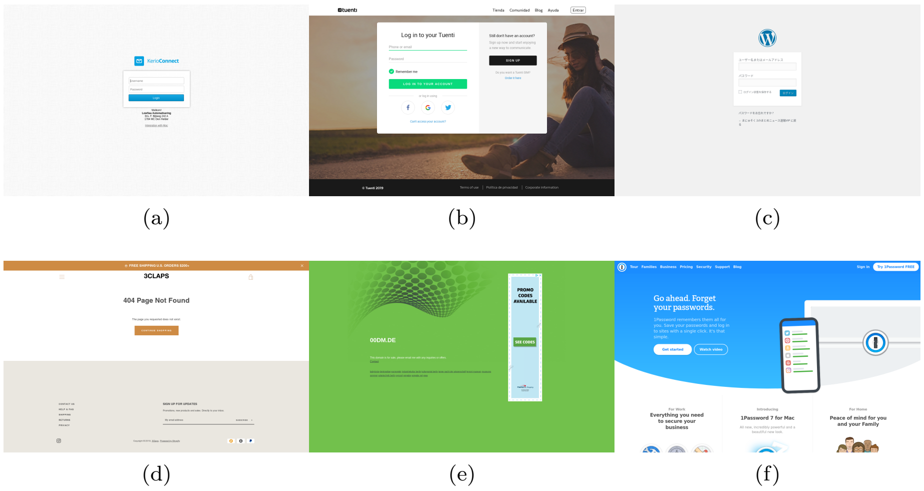


Figure 5.3: Screenshots included in the Pentest dataset. (a)–(c) belong to the "login page" category, (d) to "custom 404 page", (e) to "parked domain" and (f) to "web application".

version of the Pentest Screenshots dataset [63] originally intended to surface websites that are interesting for security penetration testing. The dataset consists of 14,000 images labeled as either "web application", "old-looking", "login page", "custom 404 page" and "parked domain". Figure 5.3 shows examples for each category. Out of the six available features, only the "login" column is used, which states whether the screenshot contains a login form. Finally, the dataset is balanced to contain the same amount of login forms and other screenshots.

As Jain and Gupta [48] achieved good results with their features, this work will also use a similar one. CodeSandbox hosts most of the content on their servers, so external URLs in either form actions, links, or images indicate phishing. Phishers often reference e.g. the image file hosted on the original websites' server to not have to update it when it changes. Therefore the number of external URLs (f_{13}) is included as a feature. When the sandbox has finished loading in the headless browser, the `href` attribute of all `a`-tags, the `src` attribute of all `img`- and `iframe`-tags and the `action` attribute of all `form`-tags on the page are collected and the number of them pointing to external domains are counted.

The compilation process demonstrates the challenge of extracting content-based features for server-side phishing detection, which is *when* to run the extraction. For traditional static websites, this is usually realized by waiting until all requests have settled and the browser has painted something to the screen (e.g. by listing for the `domcontentloaded` event using JavaScript). If the screenshot for f_{12} is taken while the compilation has not finished, it will have a low value, most likely leading to a false negative. To solve this, the feature extraction process checks if the sandbox requires compilation and waits until it has terminated. Another problem occurs when the user is redirected to an external site, but the redirection includes two steps. The first one redirects to a waiting page, where the user is finally redirected to the intended website after somewhere between 1–10 seconds. After all, requests have settled on the first page, it is considered ready, and a screenshot is taken. While f_9 will be `true`, the value of f_{12} is likely to influence the final analysis result to be less dangerous. Waiting for the page to be ready before being able to extract features also makes it harder to solve RQ3 and scale the system to prevent multiple simultaneous scans from overloading the servers. As a solution, temporary placeholder pages found in the training data are detected based on the presence of CSS classes and HTML structures. However, these do not generalize well and could lead to false positives, so finding a generic solution remains an open problem.

5.1.3 Model Training

Together with the labels described in section 4.3, the features enable the training of machine learning models for phishing detection.

Some features are not always present, while others can contain non-numeric datatypes such as periods of time. Before the model can operate on these features, they must be pre-processed and converted into a number. According to literature [64], one option to deal with missing data is to discard the entries, which would not be sensible when running in production. The alternative is to compute placeholder values instead, such as setting the sandbox and author age to zero for anonymously created sandboxes. Another commonly used technique is using the mean or another statistically computed value. Both approaches will be tested to see the influence on classification accuracy.

After preprocessing, the data is ready to be interpreted by a machine learning algorithm. A binary classifier would be sufficient to answer RQ1 and distinguish phishing sandboxes from harmless ones. However, a probabilistic classifier is able to quantify how likely a sandbox contains phishing, enabling further improvements to the review process. When including the probability on the review dashboard, two sandboxes classified as phishing now do not look equally dangerous anymore. Reviewers can either focus on confirming the sandboxes most likely to contain phishing or support the model in cases where the two class probabilities are similar.

For the implementation, Decision Tree, Random Forest, Linear Regression, and Gradient Boosting were selected from scikit-learn [65] in addition to Extreme Gradient Boosting (XGBoost) [66], based on prior usage in academic literature. Each algorithm is evaluated using different hyperparameters, which will be listed next to the results. For each parameter combination, the metrics of 5 different splits are averaged, where 80% is used for training and the remainder for computing metrics on unseen data. The ratio of phishing to harmless samples is preserved in each split to prevent it from influencing the resulting metrics. Since only 12% of the samples belong to the phishing class, the models are using the class weight parameter exposed by the selected algorithms.

This research aims to improve the existing detection mechanism's high FPR. To achieve a more natural metric where a higher number indicates better performance, its inverse, the True Positive Rate (TPR), is computed instead. However, solely optimizing for a low FPR might lead to the model developing a high threshold for classifying a sandbox as phishing, leading to many undetected phishing sandboxes. This is why the True Negative Rate (TNR), the portion of correctly detected phishing sandboxes, is also important. To combine the two, Balanced Accuracy (BA) is used, which for binary classification is defined as the arithmetic mean of the TPR and TNR [67]. The formulas to compute these metrics are shown in equation 1–3 where TP, FP, TN and FN represent the number of true positives, false positives, true negatives, and false negatives.

$$TPR = \frac{TP}{TP + FN} = 1 - FPR \tag{1}$$

$$TNR = \frac{TN}{TN + FP} \tag{2}$$

$$BA = \frac{TPR + TNR}{2} \tag{3}$$

Since the labels from section 4 are partially based on automated classifications, they may include misclassifications that the newly trained model should not learn. During the development of the models, the intermediate results of the validation dataset are used to surface such errors in the training data. Since manually re-labeling all sandboxes in the dataset would consume too much time, uncertainty sampling is used to surface anomalies or otherwise interesting samples.

The absolute difference between the class probabilities is computed to obtain a measure of the model's confidence. A value close to zero indicates that the model has difficulty deciding between the two classes, while a score near one represents the model being certain. Intuitively, the classification accuracy should be higher when the model is confident and lower when it is less so. A misclassification with high confidence can either indicate an error in the underlying data or that the model needs to be revisited. Similarly, predictions with low confidence scores are manually checked for wrong labels. If one is found, it is adjusted, which increases the model's accuracy and data quality.

5.2 Impact Analysis

To answer RQ2, the influence of each feature on the final classification result needs to be quantified. When training the classifier, it constantly issues predictions for samples of the training dataset. By computing SHAP values for these predictions, each feature's contribution to the final phishing probability can be approximated.

The hypothetical samples shown in Table 3 demonstrate the gained information. Each sample's predicted phishing probability is represented as the sum of the SHAP values of its features and the overall probability of it containing phishing without any information about its features. Note that the latter is derived from the whole training set and omitted from the table. Positive SHAP values make the sample more likely to be classified as phishing, while negative ones have the opposite effect. Therefore the fact that sample 1 has an author makes it seem harmless, but the low author age increases its probability of containing phishing.

While the per-sample SHAP values provide local explanations for a single prediction, the statistical properties of all SHAP values need to be analyzed in order to derive a global one. To find the overall most influential features, their mean absolute SHAP value is computed. Additionally, the relation between the

#	Features							Prediction	
	Has Author		Author Age		…	Ext. URLs			
	Value	SHAP	Value	SHAP		Value	SHAP	H	P
1	`true`	-0.03	2 days	0.23	…	20	0.23	0.24	0.76
2	`false`	0.25	*missing*	0.25	…	3	-0.10	0.50	0.50
					…				
N	`true`	-0.02	674 days	-0.03	…	0	-0.04	0.99	0.01

Table 3: Hypothetical values for the resulting data when computing SHAP values while training. Note that the sum of the general probability of a sample containing phishing (0.10 in this example) and all of its SHAP values always result in its predicted probability of belonging to the phishing class (P). The predicted probability for the sample being harmless (H) is $1 - P$.

value of a feature and its SHAP value can reveal interesting insights, such as whether a low author age generally makes the classifier lean towards classifying a sample as phishing.

Together, these two measures should provide a good idea about what features provide the most value for phishing detection.

5.3 Designing The System To Be Scalable

The deployed system consists of a Python server (phishing API) responsible for phishing detection, a NodeJS server responsible for launching browsers and extracting content-based features (browser feature extractor), as well as databases for persistent storage. Figure 5.4 shows the communication between the components of the system, which is as follows: When a user visits a sandbox preview, an injected script issues a request to the CodeSandbox server. Since the CodeSandbox server is the central system responsible for serving any frontend requests, it also acts as a proxy for the phishing API. It gathers the sandbox id, version, and the path at which the user opened the sandbox. All this information is then forwarded to the phishing API, triggering a phishing scan.

These processes already existed to support the rule-based phishing detector. This work added the path to the information communicated to the phishing API. Additionally, all following parts of the system have been newly implemented to enable the feature extraction needs.

The arrival of a request at the phishing API starts the extraction of the features described in section 5.1.2. Retrieving the meta- and content-based data are independent processes executed in parallel to save time. The latter requires a request to the NodeJS server, which spawns a headless browser and extracts content-based features. After the sandbox is ready, the DOM is analyzed to retrieve all content-based features except the login form similarity (f_{12}). The server creates a screenshot of the website and uploads it to a cloud storage bucket. Then the NodeJS server sends the response containing the values of each extracted feature and a URL of the screenshot. The phishing API then downloads the and uses the deep learning model to compute f_{12}. All features are now available, stored in a database for retraining purposes, and finally passed to the phishing detection model.

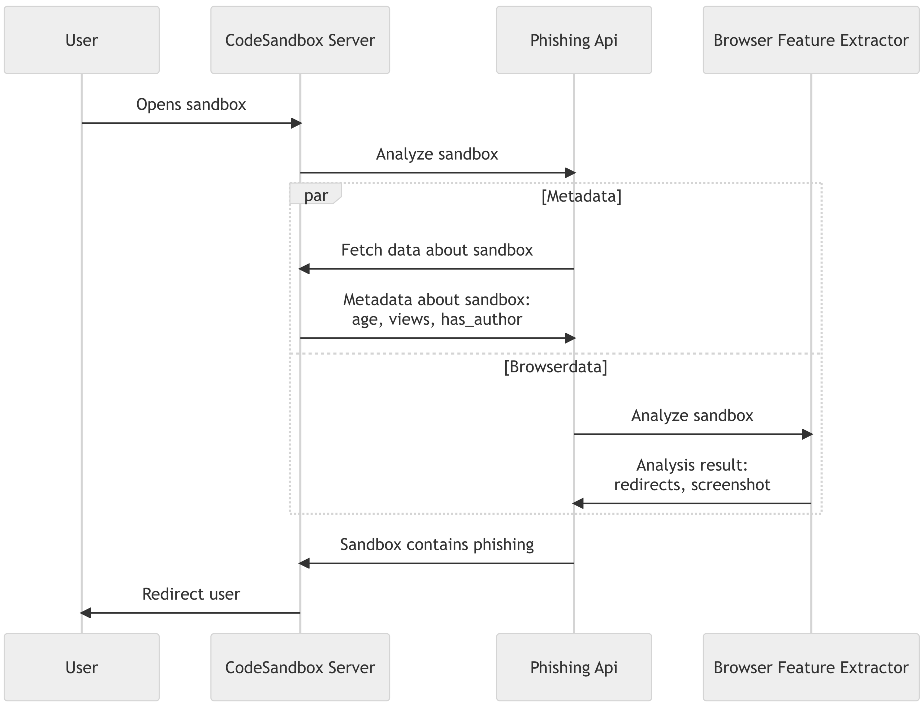


Figure 5.4: Communication between the sub-systems when a sandbox contains phishing.

The model then computes the probabilities for each class from which the final classification result is derived, depending on which class is more probable. All of these values are stored in the database alongside the features and metadata, such as the time taken when extracting features. These enable a new version of the review dashboard to show more rich and relevant results to the employees. The phishing API returns one of three responses to the CodeSandbox server, which forwards it directly to the injected script on the sandbox preview. Similar to the existing logic described in section 1.2, two thresholds exist for the phishing class probability. The sandbox is harmless if it does not exceed the first one. If it does exceed it, the phishing banner previously shown in Figure 1.4 is displayed. If it also exceeds the second one, the sandbox is considered dangerous, and the injected script redirects the user to a safe website, telling them about what just happened.

5.3.1 Performance

The new feature extraction process yields more information about a sandbox, but it is also more complex, takes longer to terminate and requires more computational resources than the existing one. Therefore measures are necessary to answer RQ3 and ensure that the phishing API responds to request on time. The most significant contributor to the increase in time and complexity is opening the sandbox in a headless browser. Since the browser feature extraction server uploads the screenshots to a cloud bucket, it does not require any local state. This enables the deployment of multiple instances behind a load balancer. If all instances are occupied, additional ones can be spawned on demand to cope with the current load. Once the demand decreases again, the number of instances can be reduced to save resources. Overall this guarantees that the feature extraction process only takes a similar amount of time as the compilation process on the user's machine.

The previous system cached the scan results based on the sandbox id and version. The new one has similar logic but adds the path visited by the user to the cache key. Before triggering a scan, the phishing API fetches all existing entries for the sandbox version from the database. If a reviewer has labeled any as phishing, the API does not trigger a new scan and immediately returns the phishing response to the CodeSandbox server. Similar logic exists for the automated classifications made by the model. Even if the currently visited path turns out harmless, the user should be warned or redirected if a dangerous or suspicious path exists elsewhere in the sandbox. If any scan yielded a score exceeding the previously mentioned second threshold, no new scan is triggered since it would not change the response.

A final step towards better scalability is to reduce the load on the phishing API overall. The system is less likely to be overloaded when fewer scans are triggered. Regular users of CodeSandbox are typically making many edits and review the results in the preview window. Each time they save their progress, the sandbox version is incremented and the preview window refreshes. The refresh triggers a request to the phishing detector, which needs to re-scan the sandbox since the version changed. It is unlikely that a user other than the sandbox author will visit the preview in between these version updates. When the CodeSandbox server receives the request for a phishing scan, it knows if the author issued the request or an external unauthenticated visitor. Since authors can not phish themselves, many unnecessary phishing scans can be prevented by not passing their own requests to the phishing API. This is an optimization that would need to be implemented in the main CodeSandbox server application, which at the time of writing, has not been completed yet. Even though its implementation is not part of this work and no results can be presented, it provides an example of how work can be avoided in server-side phishing detection.

5.4 Redesigning The Review Process

The review system described in section 1.2 enables the human reviewers to either remove the phishing banner to correct a false positive or to delete a sandbox if it contains phishing. These two actions effectively generate labels, which are currently not used to improve the detection system. By deleting phishing sandboxes, the reviewers actually do the phishers a favor since it makes it harder to find examples of patterns used by them to evade detection. Some phishers delete their sandboxes by themselves after they e.g. have been detected by popular blacklists or have tricked enough users.

While the new system also allows the reviewers to label the sandbox as harmless or phishing, the decision is stored in a database and overrides the one issued by the model. The overridden decision is then used to hide the phishing banner or redirect users to a safe page instead of the phishing sandbox preview. Since the system stores the computed features alongside the reviewers' decisions, it now contains new data for training future iterations of the phishing detection

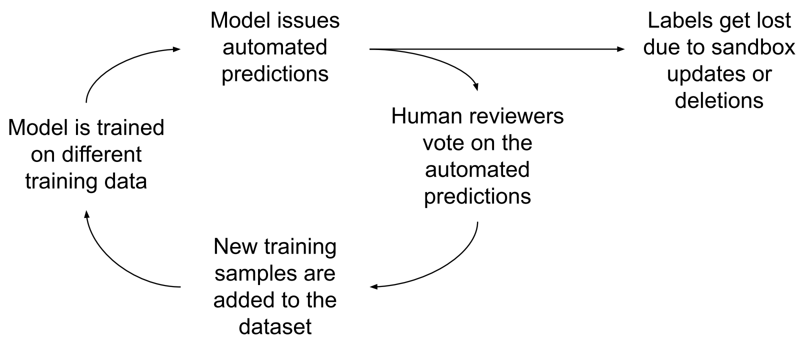

Model issues automated predictions
Labels get lost due to sandbox updates or deletions
Human reviewers vote on the automated predictions
New training samples are added to the dataset
Model is trained on different training data

Figure 5.5: Envisioned lifecycle of the training data and resulting models.

model. Every label created by a human reviewer increases the available training data, hopefully leading to a more accurate model and less manual work in the future. Figure 5.5 shows this lifecycle.

It also includes the fact that some entries in the training dataset will expire over time. If a sandbox is edited, its label may not be accurate anymore. A phisher could have updated it to prevent their patterns from being detected as phishing. Similarly, they could have turned a harmless sandbox into a phishing one. If a user chooses to delete their sandbox, it will not be available after re-running the feature extraction for the training dataset. Recomputing the features might be necessary when the way a feature is computed changes or new features are added. While the improvements due to more training data being available will most likely stagnate at some point, they are necessary to combat the described loss of training samples.

The current phishing dashboard entries shown in Figure 1.3 neither include the final score nor the exact numerical contributions of each rule. To a reviewer, all entries on the dashboard look equally dangerous, even though their scores might suggest otherwise. In an internal conversation, a CodeSandbox employee asked why a particular sandbox was classified as phishing, and another employee answered: “No one really knows exactly how it works”. While this answer was mostly intended as a joke, it highlights that the existing system could be more transparent. This lack of transparency also makes it harder for the reviewers to provide helpful feedback on the scoring system. Consider a rule adding a high value to the score for each form tag in a sandbox. A detailed breakdown of the final score enables the reviewer to suggest lowering the value added by said rule or even outright removing it if it mostly leads to false positives. Without knowing the exact contributions, reviewers can just complain about irrelevant results.

The new system includes transparency where possible. As a first step, it shows the reviewers the models’ scores for the sandbox being harmless or containing phishing. Additionally, they can request a detailed breakdown of each feature’s contribution to the final classification result. This would have been easy to compute for the previous rule-based system but can still be approximated using the new machine learning-based approach. By computing SHAP values, the system obtains a post-hoc explanation for each decision. This way, the probability computed by the classifier can be represented as a sum of the overall chance of a sandbox containing phishing and the contribution by each feature. It is comparable to the score computed by the previous system, but this time it is visible to the reviewers. Each feature has a "score" that pushes the probability value closer to either 0 or 1. The larger a value, the higher its influence on the final probability.

A reviewer can now immediately spot predictions that have been made based on questionable feature values. Consider a hypothetical example of a sandbox that looks harmless but was classified as phishing. Computing the SHAP values reveals that the most influential is the author age (f_2) having a SHAP value of 0.6. Moreover, the author's age is high, which intuitively would make them more trustworthy. On top of reporting a missclassification, the reviewer can now provide more granular feedback about the intended weights the model should assign to each feature. If high author ages often lead to misclassifications, it indicates a flaw in the training data or the model requiring deeper analysis. Even though the analysis needs to be carried out by data scientists, the example illustrates how the SHAP values enable the reviewers to provide more insightful feedback on the predictions without requiring any domain expertise in machine learning.

5.5 Comparison Through Proactive Scanning

A proactive scanner is built to evaluate the model and overall system on out-of-distribution samples. This also enables comparing the new and the existing phishing detection systems. The scanner periodically samples recently accessed sandboxes from the production database. After this step, the filesystem structure of each sandbox is obtained from the CodeSandbox server to find potential paths to scan. This way not, only the root path `/` will be scanned, but also all statically served HTML files in subfolders such as `/public/login.html`. Once all interesting paths are collected, the scanner posts a scanning request for each one to the new phishing detector, triggering the process shown in Figure 5.4. Local versions of the phishing detection API, the browser feature extractor, and the review dashboard run locally on the same system. The results will show how long the feature extraction and classification processes will take and how much phishing is detected by the new one compared to the old one.

Algorithm	TPR	TNR	BA	Parameters
XGBoost	88.93%	92.35%	**90.64%**	lr: 0.1, md: 4, ne: 220
	91.73%	87.79%	89.76%	lr: 0.05, md: 5, ne: 60
	81.66%	**94.50%**	88.08%	lr: 0.01, md: 9, ne: 220
Gradient Boosting	83.68%	91.38%	**87.53%**	lr: 0.2, md: 4, ne: 100
	84.57%	89.30%	86.94%	lr: 0.1, md: 4, ne: 100
	76.96%	**94.56%**	85.76%	lr: 0.2, md: 5, ne: 500
Random Forest	**74.36%**	97.14%	**85.75%**	cw: bs, c: e, mf: none
	72.04%	**97.65%**	84.85%	cw: none, c: g, mf: log2
Decision Tree	**72.83%**	95.94%	**84.38%**	cw: none, c: e, mf: sqrt
	59.40%	**97.71%**	78.56%	cw: b, c: e, mf: log2
Logistic Regression	**60.29%**	87.24%	**73.76%**	cw: b, mi: 500
	12.87%	**98.94%**	55.90%	cw: none, mi 1000

Table 4: Best metrics for each classification algorithm on the phishing sandboxes dataset. The highest values for a metric per classifier are highlighted in bold text. Parameters in the last column are shortened versions of the ones used by scikit-learn. The ones used are learning rate (lr), maximum tree depth (md), number of estimators (ne), class weight (cw), criterion (c) which can be either "entropy" (e) or "gini" (g), the number of features to consider for splits (mf), and the maximum number of iterations (mi). For XGBoost the objective is set to "binary:logistic" and the class weight was balanced by setting the `scale_pos_weight` parameter to the ratio of negative and positive samples.

6 Results

This section describes model training results such as metrics, error analysis, and feature importance. It also describes the feature importances by analyzing the SHAP values computed from the training data. Finally, it lists the improvements made to the review dashboard and shows how it makes it easier for the reviewers to find phishing.

6.1 Classification

On the training set, the classifier achieves a balanced accuracy of close to 93% using the XGBoost algorithm. An overview of the top results per metric for different algorithms is displayed in Table 4. As can be seen, the parameters can be used to fine-tune the classifier to focus on detecting more phishing sandboxes or yielding fewer false positives. The imputation method used to replace missing values does not influence the results in a significant way and is therefore not listed.

6.1.1 Error Analysis

The most common misclassifications surfaced by repeatedly running the model using the best parameters on different training and validation splits. Figure 6.1 shows the result of running this process 250 times. Each histogram shows the

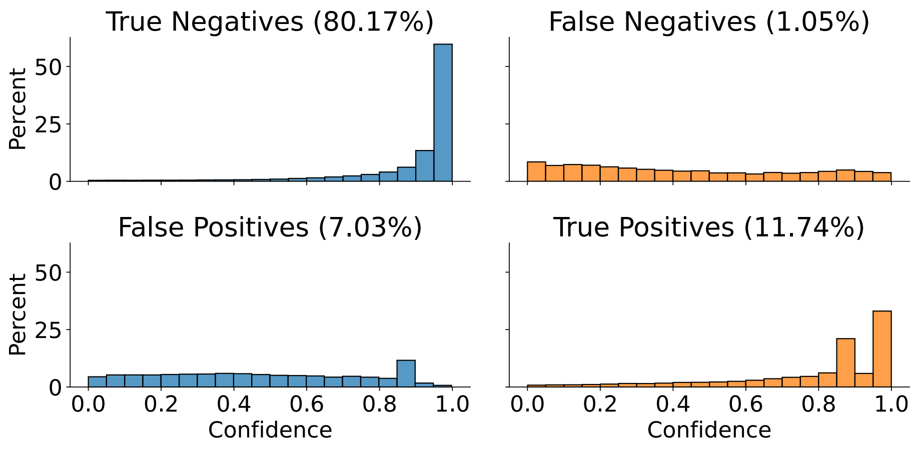
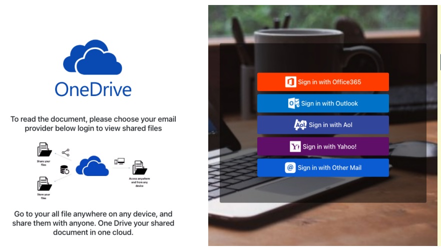


Figure 6.1: Confusion matrix and prediction confidence distribution per condition. The true class labels are color coded where blue means harmless and orange phishing.

distribution of the prediction confidence per predicted and true label. As can be seen, most of the correct predictions are issued with high confidence. These graphs looked different during the first iterations of the error analysis processes. The models made a few highly confident but ultimately wrong predictions. After manually verifying the sandboxes, they turned out to be false positives made by the old classifier. After 40 of these instances were corrected, the remaining misclassifications can be attributed to the model and wrong labels.

The false negatives mainly consist of phishing, which does not work directly via login forms. They try to trick the victim into thinking they have won a price and need to enter their credit card details to claim it. In these cases, the login form similarity is of no use and even makes the sandbox look more harmless than it really is. This is also the case when the sandboxes use effective cloaking techniques, such as displaying a CAPTCHA before revealing the real content. Other false negative samples revealed that some of the sandboxes labeled as phishing reference external websites that were taken down. A sandbox containing an iframe that once embedded an active phishing page inside a sandbox now only consists of the text “404 — Not Found”. In this case, there is neither something that looks like a login form nor any external links. Interestingly the model identifies sandboxes that redirect the visitor to a taken-down phishing website as phishing, which suggests that f_9 has a high impact on the overall classification result.

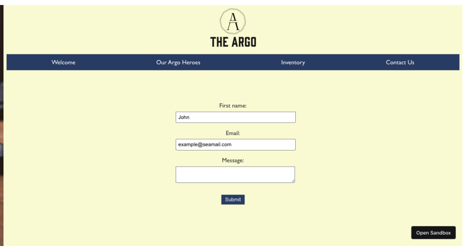
The false positives often contained non-login forms created by anonymous or young accounts. This shows that the login form detection can still be improved or that more training data is required. Other sandboxes containing forms were correctly classified as harmless. There was no apparent difference between a true negative and a false positive sandbox containing a form.



To read the document, please choose your email provider below login to view shared files

Go to your all file anywhere on any device, and share them with anyone. One Drive your shared document in one cloud.

(a)



First name: John

Email: example@seamail.com

Message:

(b)

Figure 6.2: Two screenshots of misclassified login forms. The login similarity score (f_{12}) of (a) is close to 0, even though it depicts the old login screen for OneDrive. For (b) it has a value of 1, even though it shows a contact form instead of a login page.

6.1.2 Login Form Detection

After training the model for 25 epochs, it achieves an accuracy of 90,6% on the validation data. The login form detection model works well, even though it was not explicitly trained on sandbox screenshots. After manually looking at a randomly sampled set of 500 screenshots with a login form similarity (f_{12}) larger than 0.9, 86% of the samples indeed depicted login forms. Most were traditionally looking and contained two rectangular input fields and a submit button. The remaining ones still displayed forms, but they were intended for purposes other than authenticating a user, such as Figure 6.2 (b), showing a comment form for a blog article. Such false positives were also commonly unstyled or otherwise not imitating the look of popular websites, making them uninteresting for phishing detection.

In contrast, less traditionally looking login forms including Single Sign-on (SSO) capabilities, such as shown in Figure 6.2 (a), were not detected. This specific example originates from a phishing sandbox.

The results suggest that the model focuses on traditional login forms containing two or more input fields. To lower the FPR, one could add more examples of non-login forms to the training dataset.

6.2 Feature Impact

Computing the SHAP values for the dataset provides an approximation of how important each feature is and helps to answer RQ2. The mean absolute SHAP values are displayed in Figure 6.3 and provide an idea of what features influenced the decision results the most. If the model is trained without the number of views, which on average contributes the most based on the SHAP values, the BA drops to 87% and the highest TPR to 84%. This indicates that a high mean absolute SHAP value is correlated to the feature being important for the model's predicitons.

The plot in Figure 6.4 shows how some features have a larger influence on a few classification results than the averages in Figure 6.3 suggest. It is constructed based on the data of the feature columns as described in Table 3. Each row in the diagram represents a feature. The points represent samples,

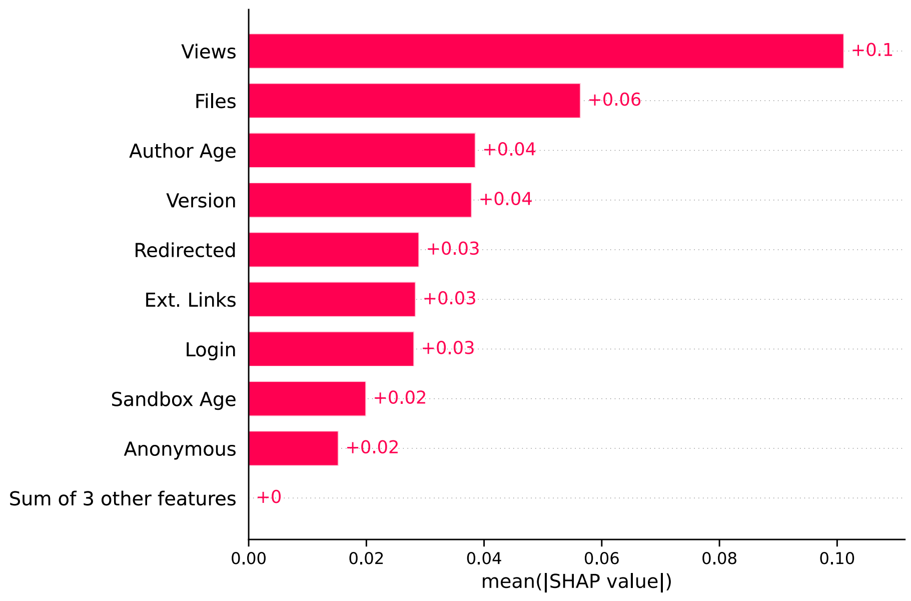


Figure 6.3: Mean SHAP values for each feature in descending order.

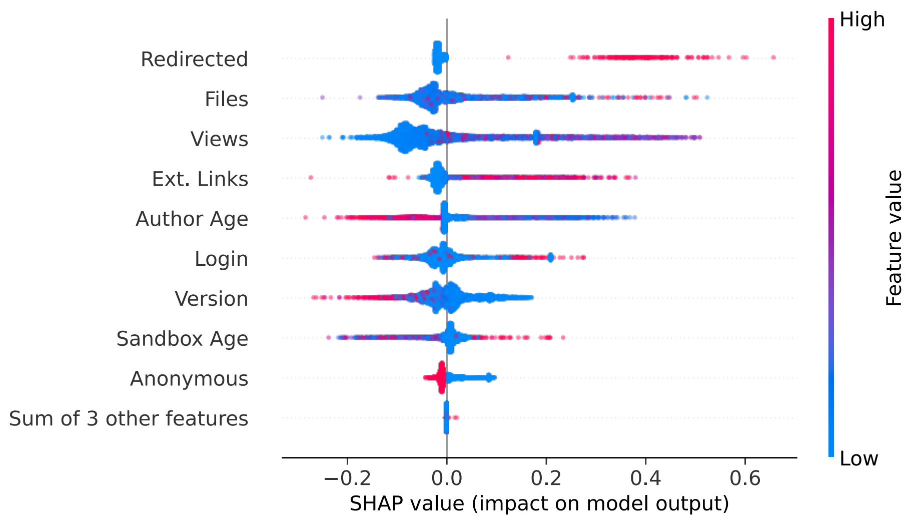


Figure 6.4: Approximated contributions (SHAP values) to the final classification result per feature colored by value. The y value is randomized to get a qualitative idea of the distribution.

where each one is mapped onto the x-axis based on its SHAP value, while its feature value after preprocessing determines its color. Therefore, the position of each point provides an idea of its influence on the predicted probability. The further it is to the right, the more it influences the phishing probability to be higher. Inversely, points on the left side make it seem more harmless.

This uncovers relations between the feature value and its influence on the predicted phishing probability. Most sandboxes in the dataset do not redirect the user, as shown by the training data distribution in 5.2. Therefore, f_9 does not have a huge influence on the result when it is `false`, leading to a comparatively low mean absolute SHAP value. However, when redirection occurs, it is usually enough for the sandbox to be classified as phishing. The plot also confirms some assumptions made when selecting the features. A sandbox having no author or one that was recently created makes the model more likely to classify a sandbox as phishing. Another less reliable hint for phishing is a high number of external links. A high author age seems to be associated with less phishing activity, thereby confirming its initial inclusion as a measure to lower the false positive rate. The inverse can be observed with anonymous sandboxes, which clearly make the model lean more towards phishing, though with a negligible impact.

An unexpected result is obtained for the version feature. If the version number contributed to the sandbox being classified as phishing, its value was commonly high. This relation is the opposite of why this feature was included. Similarly, the method of creation (f_7), compiler error f_{11} and the number of likes (f_5) did not influence the results in any meaningful way.

6.3 Performance

The new system is computationally more expensive than the previous system since the sandboxes are opened in an actual browser when extracting content-based features. This means that the overall time from a user opening the sandbox until the classification result is ready is longer. The median classification took 21 seconds. When waiting longer than 2 minutes for a sandbox to be ready, content-based features were extracted regardless of current compilation processes to unblock the extraction queue. Since 95% of scans terminated in under 60 seconds, this time can be cut in half to get faster classification results in these edge cases.

While the median classification time is larger than the 10-second goal defined in RQ3, one can argue that the system is still fast enough. The time t_δ between the visitor seeing the webpage and the phishing API responding can therefore be expressed as equation 4. To illustrate this, consider the example of a user visiting a sandbox, which is visualized in Figure 6.5. They visit the sandbox, the compilation process downloads the necessary dependencies, and after a short delay d, the request to the phishing detector is triggered. If all browser feature extraction workers are busy, the request has to wait for some time w until it can be processed. Therefore the compilation process on the server begins up to a few seconds after the one on the client. Depending on the server's hardware h, the compilation will be faster or slower than the one on the users' machine t_c. Additional constant time b is spent consulting the cache, storing the features, and creating and storing the classification result.

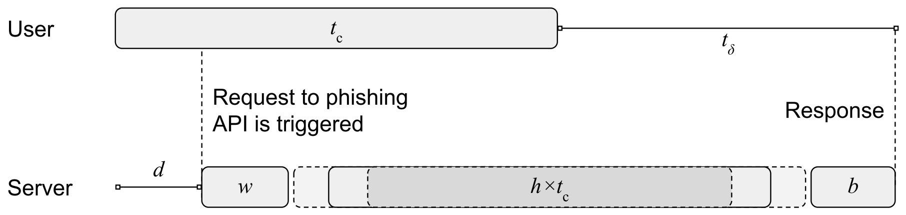


Figure 6.5: Timeline of the subprocesses from equation 4.

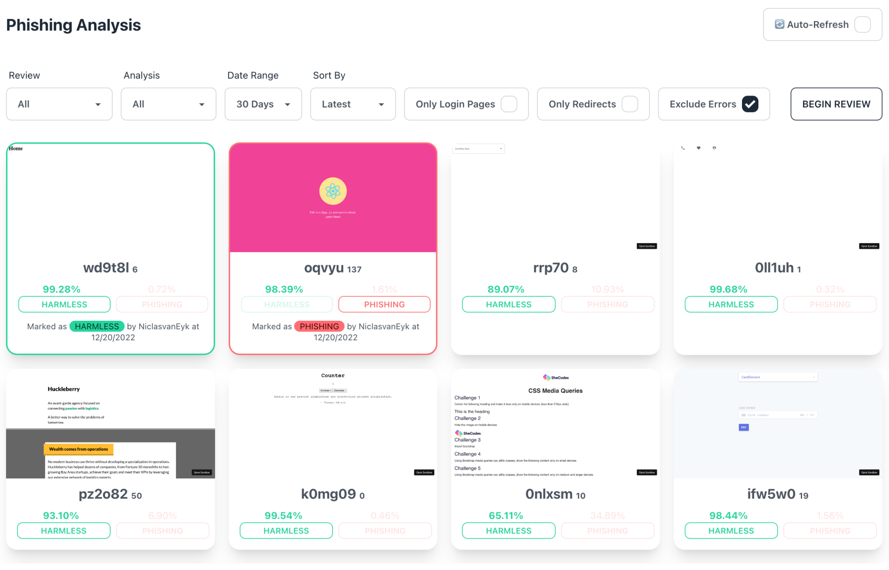
$$t_\delta = d + w + h \times t_c + b - t_c \tag{4}$$

To optimize the t_δ, b is unlikely to take more than one second and is unlikely to have much room for improvement besides using proper database indices when looking up past classification results. Similarly, d can be kept below one second by injecting the phishing detection script at the top of the HTML document, making it the first request that the page issues. In an ideal case, a headless browser is ready, the request can be served immediately, and the server hardware is equally powerful as the client one, making their compile times equally long. This results in t_δ being equal to $d + b = 2$, which meets the 10-second goal defined in research question RQ3.

As mentioned during the definition of the research questions, the current phishing detector starts 24 scans per minute on average. While running the proactive scanner, the locally running instance of the browser feature extractor could handle up to 10 sandboxes simultaneously while maintaining similar compile times to those when opening one sandbox. This illustrates that only a few instances are required to run in production to maintain the load and prevent w from significantly influencing t_δ. If the user has more powerful hardware and compiles the sandboxes more quickly, it is likely to exceed the 10-seconds goal. Assuming that due to high server load, the user can compile the sandbox twice as fast, even with the median compile time t_c of 21 seconds, the clients' compilation process finishes 10 seconds faster. However, phishing sandboxes generally take less time to compile since the compilation screen makes their websites look less legitimate and drives potential victims away. In the case of shorter or even no compile time, the influence of h becomes less significant. All in all, the system can achieve the intended 10-second goal with a reasonable amount of computing resources, which also can be easily scaled up or down based on demand.

6.4 Review

The new iteration of the review dashboard is displayed in Figure 6.6 and provides a view into *all* decisions made by the model. Currently, the reviewers are only exposed to unreviewed predictions, which are expected to contain phishing. This means that reviewers are not able to detect false negatives, and no harmless samples could be generated for retraining. Additionally, there was no way to track the review decisions, to e.g. manually check whether some phishing pattern were already reviewed in the past. Displaying all scanned sandboxes addresses these shortcomings, but new problems arise due to the now larger result set.


Phishing Analysis

Auto-Refresh

Review: All
Analysis: All
Date Range: 30 Days
Sort By: Latest
Only Login Pages
Only Redirects
Exclude Errors ✓
BEGIN REVIEW

Figure 6.6: The new review dashboard.

A filter bar at the top is introduced to limit the displayed result set. Sandboxes can be filtered on their review status ("all", "unreviewed", "reviewed", "harmless", "phishing"), the prediction made by the model ("all", "harmless", "phishing"), and the date at which the sandbox was scanned ("all", "last 30 days", "last 7 days", "today").

Additionally, three feature-based filters are introduced to make the displayed results more relevant. Sandboxes that threw a compiler error while extracting features can be excluded, which is active by default. Since a red overlay is shown and the probability of them being functional is low, excluding them is unlikely to increase the danger for users. Around 20% of 5000 randomly sampled sandboxes do not compile properly, so this filter helps to reduce noise for the reviewers. The other two feature-based filters can be used to either only show sandboxes looking like login pages or ones that immediately redirect the users to another website not controlled by CodeSandbox. While both indicators are not proof of phishing, they represent relevant cases for reviewers to look at. As the distributions shown earlier in Figure 5.2, the overwhelming majority of sandboxes redirecting their users are used for phishing. Also, fast redirection circumvents measures like the phishing banner from being effective since the sandbox can redirect the user before the response from the phishing detector is received.

By default, the entries are listed by the order they were scanned in. For the phishing review, there are additional sorting options "danger" and "uncertainty". The first one sorts by the phishing class score in descending order and is intended to provide the reviewers with the most likely phishing entries first. As the distributions in Figure 6.1 show, the models also make fewer errors in high-confidence regions, so the reviewers should also see fewer false positives. The second sorting option sorts by confidence in ascending order. It enables uncertainty sampling, which generates valuable training data for entries the model

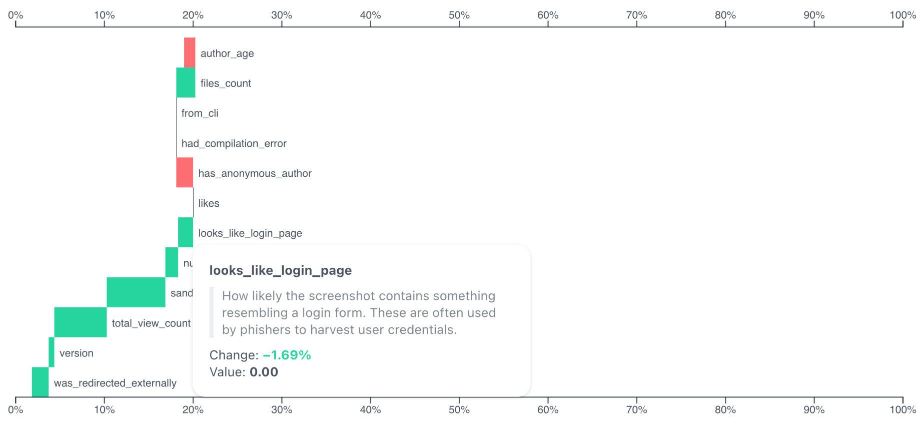


Figure 6.7: Waterfall chart displaying the influence of each feature based on their computed SHAP values. The reviewers can view detailed values and feature descriptions by hovering with their mouse.

has difficulty classifying correctly.

The intended way of using the dashboard is as follows: A reviewer visits the dashboard and presses the button at the top right to apply the filters for the "review mode". This limits the result set to unreviewed sandboxes predicted to contain phishing, sorted by the likelihood of them containing phishing in descending order. If the list of sandboxes to be reviewed is still too large to review manually in a reasonable amount of time, it can be further limited using feature-based filters. These have already been helpful during development since they revealed a bug in the feature extraction process. The listeners running inside the headless browser tracked redirects inside iframes, which resulted in false positives for sandboxes embedding YouTube videos.

An entry on the dashboard displays the sandboxes' id alongside its version and the path that was scanned. Additionally, the screenshot from the scan is shown, which was used to determine the likelihood of it depicting a login form. The reviewer is then presented with two buttons to label the entry as either harmless or phishing. Additionally, the prediction scores for each class are displayed above each button. The one representing the more likely class is displayed in bold, while the other one has lower opacity. If an entry was reviewed, the reviewer's decision is appended below the one of the model.

When the reviewers disagree with the model's decision, they can open the detail view of an entry. Next to a larger version of the screenshot and the raw feature values, this view contains a waterfall chart like the one shown in Figure 6.7 explaining the decision result. This is based on SHAP values that are computed on-demand using the raw features from the database and the stored model.

7 Discussion

7.1 Classification

The resulting (balanced) accuracy is comparable to the lower end of related works shown in Table 1. Possible reasons are the more complicated domain of developer tooling. Distinguishing an educational login form and a phishing one is hard to get right. The number of features is also low compared to other ML-based approaches. This is amplified by three of the 13 features not having a meaningful influence on the classification results. The inclusion of even more features might help, as well as more accurate data.

The results of the proactive scanning show that the model still yields more false than true positives. A potential reason for this could be the ratio of phishing samples in the training data being higher than in the real world. If the model is influenced by this distribution, it is more likely to classify a sample as phishing than if it was trained with more harmless examples than currently used.

While the new ML-based approach decreases the high FPR compared to the rule-based approach, the number of false positives still remains a problem.

7.2 Features

The feature importance results are close to those of Phishari [4]. Similar to their results, the account age and redirects significantly influence the classification decision. Redirects are more important for this specific scenario, probably because the attractive short preview links make CodeSandbox a more popular target to redirect from.

When observing the SHAP values of manually sampled false positives, the metadata features like the number of views have a huge influence on the predicted probabilities. While the metadata features were intended to surface suspicious sandboxes for reviewers to look at, automatically classifying one mostly based on its number of views and the age of its author seems unreasonable. They also lead to the model learning unwanted patterns. As stated previously, a sandbox having a high version number leads to a smaller probability of containing phishing. This is not something that the model should learn and might point to flaws in the training data. A regular user of CodeSandbox should not be penalized and classified as phishing because they made more edits to their code than others.

The content-based features seem to be more reliable. Their SHAP values show a clear correlation to e.g. a redirect happening and the sandbox being classified as phishing. This is less pronounced for the login form feature, however the explicit inclusion of harmless sandboxes containing login forms might have influenced this. If there are the same number of harmless and harmful sandboxes that contain login forms, the model is less likely to rely on this feature.

To answer RQ2, the content-based features are the most helpful for identifying phishing sandboxes. While the metadata ones have a high impact on average, they are more likely to lead to false positives.

7.3 Performance

The current system triggers 24 scans per minute on average. As the results of local scanning presented in section 6.3 show, non-server hardware can handle 10 concurrent scans while maintaining a median compile time of 21 seconds. When using more powerful hardware and more than one instance, this shows that a real-world deployment can handle the current load of the system without an unreasonable amount of computational cost.

However, most requests to the phishing API can currently be served from the cache and trigger no new scans. A requirement for caching to work correctly is to capture all parameters that influence the computation. If the path were excluded from the cache key in the new system, phishing hidden in a sub-path would potentially never be found. When the visit that triggered the first scan was to the root path, all subsequent requests, even for other paths, will be retrieved from the cache.

Also, passing the visits' query parameters to the phishing API would be an effective measure to fight redirect cloaking. At the same time, it would increase the number of scans required. Besides the query parameter, modern browsers offer many parameters that can determine the page's contents. Examples include any header value of the request, but also client-side features such as the presence of a gyroscope sensor on the device. Including them all in the cache key would render the cache effectively useless.

Even if we assume the existence of a perfect phishing detector, implementing caching opens up new attack vectors. Since JavaScript can read the current time, a phisher is able to create a website that displays harmless content up until a chosen point in time, after which it displays a phishing form instead. Once a phishing scanner evaluates the site, it finds only harmless content and caches the result. If the site is not scanned again for further visits, the phishing version of the site will not be detected. The system can mitigate this attack by expiring entries in the cache after a certain amount of time. The host therefore needs to balance the security of their users and the resulting increase in resource usage. The larger this timeframe, the more time a phisher has to attract victims. Lowering this timeframe increases the number of scans and computational costs.

To answer RQ3, the system can be fast enough by using a stateless architecture for the most computationally expensive part, the headless browser analysis. The current load could be handled without unreasonable amounts of additional compute, but when extending the system in the future, passing more information to the phishing API might affect its performance.

7.4 Review Dashboard

As stated at the end of section 7.1, the number of false positives remains a problem for the automated classifications. However, the inclusion of filters based on content-based features on the review dashboard makes this problem more manageable. Especially filtering based on redirects helps with finding true positives. If two entries have a score of 99% and one redirects the user, it is more likely to contain phishing than the other one. The same holds for login forms, though this filter still contains some amount of false positives.

The explanations provided by SHAP and the waterfall chart shown in Figure 6.7 generally work well. However, due to their post-hoc nature, they only describe the features' influence, not *why* a particular feature value is a sign of phishing. As can also be seen in the beeswarm plot in Figure 6.4, a high author age generally makes the sandbox look more harmless, but for some instances, it also makes the model think it contains phishing. This is most likely due to other features having a specific value, which is not reflected in the explanation.

The continuous collection of training data based on the reviewers' decisions also proves to be necessary. After manually confirming 18 phishing sandboxes found the described active scanning approach, only 12 of them were still present after two weeks after their discovery. While another approach might be to prevent deletion after a sandbox is confirmed to contain phishing, adding new samples to the training data.

All in all, the implemented measures answer RQ4. The review system provides the reviewers with more transparency, explainability, and uses their decisions to fight data loss and improve the automated predictions in the future.

7.5 Security

The error analysis shows that the feature extraction phase is vulnerable to the influence of phishers. For content-based features, this is mainly represented by cloaking. As already mentioned, one of the most common types of phishing is redirection to external phishing websites. If a phishing login form is hidden from the browser feature extractor, it will be missing on the screenshot, and f_{12} will be lowered. Similarly, the detected number of external links f_{13} can be decreased.

The query parameter cloaking can be prevented by passing more information from the initial request to the phishing scanner. To prevent phishers from detecting the automated visit, one needs to try to appear as convincing as possible. Addons for headless browsers exist that make their detection harder and even enable them to solve CAPTCHAs[4]. While simple detection mechanisms like the user agent can easily be circumvented, more advanced ones like moving the mouse as described in CrawlPhish [20] are harder to solve. Not being able to detect bots also creates problems in other areas, as it is e.g. vital for limiting the automated creation of user accounts, something that could harm other features such as the sandbox having an author (f_1).

The metadata features are mostly resilient to the influence of attackers. While they can change the sandbox version with little effort, creating new accounts or gaining access to old ones takes more effort. Similarly, the view count could be incremented through automation, again requiring effort on the phisher's side. The increase in effort required by phishers to game the features can also be seen as an additional security mechanism. If a phisher needs to create automation scripts specific to a single provider, it will likely drive them away to a different host.

[4]`https://github.com/berstend/puppeteer-extra/tree/master/packages/puppeteer-extra-plugin-recaptcha`

8 Conclusion

8.1 Summary

All in all, the results show that phishing detection at the website host can be effective. While hosting providers with a free plan represent an attractive target for phishers, they have access to unique information, enabling them to detect phishing more effectively. This work focused on CodeSandbox, but the ideas and techniques can be generalized to other hosting platforms such as Amazon or DigitalOcean.

This work provides answers to four research questions. The new system improves upon the current one's high FPR (RQ1) by incorporating more information into the classification process and using a more advanced method. SHAP values have been computed to obtain a post-hoc breakdown of the contribution per feature. This revealed the number of views (f_4) and files (f_8) to be the most influential features on average (RQ2), while redirects (f_9) or the author age (f_2) have a more significant influence for a few specific instances. While the new features require a more complicated and resource-intensive extraction process, the classification process is still fast enough (RQ3) by utilizing a flexibly scalable architecture, caching classification results, and requiring fewer phishing scans overall. Finally, it makes the automated decision process easier to understand and fills the review dashboard with more relevant entries. It also provides the reviewers with tools to filter the resultset if it gets too large. The reviewers' decisions are used to continuously improve the automated classification results, thereby using their manual work more effectively (RQ4).

The experiments also showed up unique challenges involved in server-side phishing detection. Since users of the platform can constantly change or delete their websites, additional steps are necessary to ensure a large and up-to-date dataset. The host is also required to balance the effectiveness of the system and its associated cost.

8.2 Future Work

The results show that the login form detection can still be improved by expanding the functionality of the review dashboard. It could let the reviewers also label the screenshots of sandboxes as login forms or not, similar to how the phishing voting system works. This would help them to receive more relevant results when using the login form filter and will likely improve the relevance of f_{12}.

Cloaking remains another crucial area for future research. The more information is passed from the user's visit to the phishing detector, the more closely it can be replicated. However, this also makes caching harder since the website could be different depending on the query parameter, cookies, and other request-specific parameters. This increases the computational load, potentially making the whole system more expensive.

Another possible future research would be to explore the effectiveness of user feedback. This way, the user base can support the employees with their review duties. While users may not be as trustworthy as paid workers, community websites such as PhishTank use such a crowd-based system and are widely used for phishing detection. It would also help an author whose sandbox was falsely flagged as phishing to report this mistake and make them feel less helpless in such a situation.

A Appendix

References

[1] Christian Ludl et al. "On the Effectiveness of Techniques to Detect Phishing Sites". In: *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment.* Springer. 2007, pp. 20–39.

[2] *Phishing Activity Trends Reports.* URL: `https://apwg.org/trendsreports` (visited on 06/20/2022).

[3] Nguyet Quang Do et al. "Deep Learning for Phishing Detection: Taxonomy, Current Challenges and Future Directions". In: *IEEE access : practical innovations, open solutions* (2022), pp. 36429–36463. DOI: `10.1109/ACCESS.2022.3151903`.

[4] Anupama Aggarwal, Ashwin Rajadesingan, and Ponnurangam Kumaraguru. "PhishAri: Automatic Realtime Phishing Detection on Twitter". In: *2012 eCrime Researchers Summit.* IEEE. 2012, pp. 1–12.

[5] Mahmood Moghimi and Ali Yazdian Varjani. "New Rule-Based Phishing Detection Method". In: *Expert Systems with Applications* 53 (July 1, 2016), pp. 231–242. ISSN: 0957-4174. DOI: `10.1016/j.eswa.2016.01.028`.

[6] Simon Bell and Peter Komisarczuk. "An Analysis of Phishing Blacklists: Google Safe Browsing, OpenPhish, and PhishTank". In: *Proceedings of the Australasian Computer Science Week Multiconference.* ACSW '20. New York, NY, USA: Association for Computing Machinery, 2020. ISBN: 978-1-4503-7697-6. DOI: `10.1145/3373017.3373020`.

[7] Neha Gupta, Anupama Aggarwal, and Ponnurangam Kumaraguru. "Bit.Ly/Malicious: Deep Dive into Short URL Based e-Crime Detection". In: *2014 APWG Symposium on Electronic Crime Research (eCrime).* 2014 APWG Symposium on Electronic Crime Research (eCrime). Sept. 2014, pp. 14–24. DOI: `10.1109/ECRIME.2014.6963161`.

[8] Hossein Siadati, Toan Nguyen, and Nasir Memon. "X-Platform Phishing: Abusing Trust for Targeted Attacks Short Paper". In: *Financial Cryptography and Data Security.* Ed. by Michael Brenner et al. Cham: Springer International Publishing, 2017, pp. 587–596. ISBN: 978-3-319-70278-0.

[9] Ibrahim Waziri. "Website Forgery: Understanding Phishing Attacks and Nontechnical Countermeasures". In: *2015 IEEE 2nd International Conference on Cyber Security and Cloud Computing.* 2015 IEEE 2nd International Conference on Cyber Security and Cloud Computing. Nov. 2015, pp. 445–450. DOI: `10.1109/CSCloud.2015.77`.

[10] David G. Dobolyi and Ahmed Abbasi. "PhishMonger: A Free and Open Source Public Archive of Real-World Phishing Websites". In: *2016 IEEE Conference on Intelligence and Security Informatics (ISI).* 2016 IEEE Conference on Intelligence and Security Informatics (ISI). Sept. 2016, pp. 31–36. DOI: `10.1109/ISI.2016.7745439`.

[11] Adam Oest et al. "Sunrise to Sunset: Analyzing the End-to-end Life Cycle and Effectiveness of Phishing Attacks at Scale". In: 29th USENIX Security Symposium (USENIX Security 20). 2020, pp. 361–377. ISBN: 978-1-939133-17-5.

[12] Jonathan Woodbridge et al. “Detecting Homoglyph Attacks with a Siamese Neural Network”. In: *2018 IEEE Security and Privacy Workshops (SPW)*. 2018 IEEE Security and Privacy Workshops (SPW). May 2018, pp. 22–28. DOI: 10.1109/SPW.2018.00012.

[13] Collin Jackson et al. “An Evaluation of Extended Validation and Picture-in-Picture Phishing Attacks”. In: *Financial Cryptography and Data Security*. Ed. by Sven Dietrich and Rachna Dhamija. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2007, pp. 281–293. ISBN: 978-3-540-77366-5. DOI: 10.1007/978-3-540-77366-5_27.

[14] Jhen-Hao Li and Sheng-De Wang. “PhishBox: An Approach for Phishing Validation and Detection”. In: *2017 IEEE 15th Intl Conf on Dependable, Autonomic and Secure Computing, 15th Intl Conf on Pervasive Intelligence and Computing, 3rd Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress(DASC / PiCom / DataCom / CyberSciTech)*. Nov. 2017, pp. 557–564. DOI: 10.1109/DASC-PICom-DataCom-CyberSciTec.2017.101.

[15] Yi-Shin Chen et al. “Detect Phishing by Checking Content Consistency”. In: *Proceedings of the 2014 IEEE 15th International Conference on Information Reuse and Integration (IEEE IRI 2014)*. 2014, pp. 109–119. DOI: 10.1109/IRI.2014.7051880.

[16] Mahmood Moghimi and Ali Yazdian Varjani. “New Rule-Based Phishing Detection Method”. In: *Expert Systems with Applications* (2016), pp. 231–242. ISSN: 0957-4174. DOI: 10.1016/j.eswa.2016.01.028.

[17] Aditya Gopal Menon and Gilad Gressel. “Concept Drift Detection in Phishing Using Autoencoders”. In: *Machine Learning and Metaheuristics Algorithms, and Applications*. Ed. by Sabu M. Thampi et al. Communications in Computer and Information Science. Singapore: Springer, 2021, pp. 208–220. ISBN: 9789811604195. DOI: 10.1007/978-981-16-0419-5_17.

[18] *Safe Browsing – Google Safe Browsing*. URL: https://safebrowsing.google.com.

[19] Swati Maurya and Anurag Jain. “Deep Learning to Combat Phishing”. In: *Journal of Statistics and Management Systems* 6 (2020), pp. 945–957.

[20] Penghui Zhang et al. “CrawlPhish: Large-scale Analysis of Client-side Cloaking Techniques in Phishing”. In: *2021 IEEE Symposium on Security and Privacy (SP)*. 2021 IEEE Symposium on Security and Privacy (SP). May 2021, pp. 1109–1124. DOI: 10.1109/SP40001.2021.00021.

[21] Adam Oest et al. “PhishFarm: A Scalable Framework for Measuring the Effectiveness of Evasion Techniques against Browser Phishing Blacklists”. In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019 IEEE Symposium on Security and Privacy (SP). May 2019, pp. 1344–1361. DOI: 10.1109/SP.2019.00049.

[22] Yolanda Gil et al. “Towards Human-Guided Machine Learning”. In: *Proceedings of the 24th International Conference on Intelligent User Interfaces*. IUI ’19. New York, NY, USA: Association for Computing Machinery, Mar. 17, 2019, pp. 614–624. ISBN: 978-1-4503-6272-6. DOI: 10.1145/3301275.3302324.

[23] Florian Westphal, Niklas Lavesson, and Håkan Grahn. “A Case for Guided Machine Learning”. In: *Machine Learning and Knowledge Extraction.* Lecture Notes in Computer Science (2019). Ed. by Andreas Holzinger et al., pp. 353–361. DOI: 10.1007/978-3-030-29726-8_22.

[24] Saleema Amershi et al. “Power to the People: The Role of Humans in Interactive Machine Learning”. In: *AI Magazine* 35.4 (Dec. 22, 2014), pp. 105–120. ISSN: 2371-9621. DOI: 10.1609/aimag.v35i4.2513.

[25] Ashish Kapoor et al. “Interactive Optimization for Steering Machine Classification”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems.* CHI ’10. New York, NY, USA: Association for Computing Machinery, Apr. 10, 2010, pp. 1343–1352. ISBN: 978-1-60558-929-9. DOI: 10.1145/1753326.1753529.

[26] Sebastian Lapuschkin et al. “Unmasking Clever Hans Predictors and Assessing What Machines Really Learn”. In: *Nature Communications* 10.1 (1 Mar. 11, 2019), p. 1096. ISSN: 2041-1723. DOI: 10.1038/s41467-019-08987-4.

[27] Alejandro Barredo Arrieta et al. “Explainable Artificial Intelligence (XAI): Concepts, Taxonomies, Opportunities and Challenges toward Responsible AI”. In: *Information Fusion* 58 (June 1, 2020), pp. 82–115. ISSN: 1566-2535. DOI: 10.1016/j.inffus.2019.12.012.

[28] Leo Breiman. “Random Forests”. In: *Machine Learning* 45.1 (Oct. 1, 2001), pp. 5–32. ISSN: 1573-0565. DOI: 10.1023/A:1010933404324.

[29] *”Why Should I Trust You?” — Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining.* URL: https://dl-acm-org.ezproxy2.utwente.nl/doi/abs/10.1145/2939672.2939778 (visited on 12/30/2022).

[30] Scott M Lundberg and Su-In Lee. “A Unified Approach to Interpreting Model Predictions”. In: *Advances in Neural Information Processing Systems 30.* Ed. by I. Guyon et al. Curran Associates, Inc., 2017, pp. 4765–4774.

[31] Scott M. Lundberg et al. “From Local Explanations to Global Understanding with Explainable AI for Trees”. In: *Nature Machine Intelligence* 2.1 (2020), pp. 2522–5839.

[32] Ksenia Konyushkova, Raphael Sznitman, and Pascal Fua. “Learning Active Learning from Data”. In: *Advances in Neural Information Processing Systems.* Vol. 30. Curran Associates, Inc., 2017.

[33] Li Deng. “The MNIST Database of Handwritten Digit Images for Machine Learning Research [Best of the Web]”. In: *IEEE Signal Processing Magazine* 29.6 (Nov. 2012), pp. 141–142. ISSN: 1558-0792. DOI: 10.1109/MSP.2012.2211477.

[34] David D. Lewis and Jason Catlett. “Heterogeneous Uncertainty Sampling for Supervised Learning”. In: *Machine Learning Proceedings 1994.* Ed. by William W. Cohen and Haym Hirsh. San Francisco (CA): Morgan Kaufmann, Jan. 1, 1994, pp. 148–156. ISBN: 978-1-55860-335-6.

[35] Sreyasee Das Bhattacharjee et al. “Prioritized Active Learning for Malicious URL Detection Using Weighted Text-Based Features”. In: *2017 IEEE International Conference on Intelligence and Security Informatics (ISI)*. 2017 IEEE International Conference on Intelligence and Security Informatics (ISI). July 2017, pp. 107–112. DOI: 10.1109/ISI.2017.8004883.

[36] Robert (Munro) Monarch. *Human-in-the-Loop Machine Learning*. Manning, June 2021. ISBN: 978-1-61729-674-1.

[37] Hieu T. Nguyen and Arnold Smeulders. “Active Learning Using Pre-Clustering”. In: *Proceedings of the Twenty-First International Conference on Machine Learning*. ICML ’04. New York, NY, USA: Association for Computing Machinery, July 4, 2004, p. 79. ISBN: 978-1-58113-838-2. DOI: 10.1145/1015330.1015349.

[38] Yi Yang et al. “Multi-Class Active Learning by Uncertainty Sampling with Diversity Maximization”. In: *International Journal of Computer Vision* 113.2 (June 1, 2015), pp. 113–127. ISSN: 1573-1405. DOI: 10.1007/s11263-014-0781-x.

[39] Yue Zhang, Jason I. Hong, and Lorrie F. Cranor. “Cantina: A Content-Based Approach to Detecting Phishing Web Sites”. In: *Proceedings of the 16th International Conference on World Wide Web*. WWW ’07. New York, NY, USA: Association for Computing Machinery, 2007, pp. 639–648. ISBN: 978-1-59593-654-7. DOI: 10.1145/1242572.1242659.

[40] Sadia Afroz and Rachel Greenstadt. “PhishZoo: Detecting Phishing Websites by Looking at Them”. In: *2011 IEEE Fifth International Conference on Semantic Computing*. 2011 IEEE Fifth International Conference on Semantic Computing. Sept. 2011, pp. 368–375. DOI: 10.1109/ICSC.2011.52.

[41] Luong Anh Tuan Nguyen et al. “Detecting Phishing Web Sites: A Heuristic URL-based Approach”. In: *2013 International Conference on Advanced Technologies for Communications (ATC 2013)*. 2013 International Conference on Advanced Technologies for Communications (ATC 2013). Oct. 2013, pp. 597–602. DOI: 10.1109/ATC.2013.6698185.

[42] Angelo P. E. Rosiello et al. “A Layout-Similarity-Based Approach for Detecting Phishing Pages”. In: *2007 Third International Conference on Security and Privacy in Communications Networks and the Workshops - SecureComm 2007*. 2007 Third International Conference on Security and Privacy in Communications Networks and the Workshops - SecureComm 2007. Sept. 2007, pp. 454–463. DOI: 10.1109/SECCOM.2007.4550367.

[43] Matthew Dunlop, Stephen Groat, and David Shelly. “GoldPhish: Using Images for Content-Based Phishing Analysis”. In: *2010 Fifth International Conference on Internet Monitoring and Protection*. 2010, pp. 123–128. DOI: 10.1109/ICIMP.2010.24.

[44] XiangGuang et al. “CANTINA+: A Feature-Rich Machine Learning Framework for Detecting Phishing Web Sites”. In: *ACM Transactions on Information and System Security (TISSEC)* (Sept. 1, 2011). DOI: 10.1145/2019599.2019606.

[45] Vahid Shahrivari, Mohammad Mahdi Darabi, and Mohammad Izadi. "Phishing Detection Using Machine Learning Techniques". 2020. arXiv: 2009.11116.

[46] Samuel Marchal et al. "Know Your Phish: Novel Techniques for Detecting Phishing Sites and Their Targets". In: *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS).* 2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS). June 2016, pp. 323–333. DOI: 10.1109/ICDCS.2016.10.

[47] Rami M. Mohammad, Fadi Thabtah, and Lee McCluskey. "Predicting Phishing Websites Based on Self-Structuring Neural Network". In: *Neural Computing and Applications* 2 (Aug. 1, 2014), pp. 443–458. ISSN: 1433-3058. DOI: 10.1007/s00521-013-1490-z.

[48] Ankit Kumar Jain and B. B. Gupta. "A Machine Learning Based Approach for Phishing Detection Using Hyperlinks Information". In: *Journal of Ambient Intelligence and Humanized Computing* 10.5 (May 1, 2019), pp. 2015–2028. ISSN: 1868-5145. DOI: 10.1007/s12652-018-0798-z.

[49] Dongjie Liu et al. "Malicious Websites Detection via CNN Based Screenshot Recognition". In: *2019 International Conference on Intelligent Computing and Its Emerging Applications (ICEA).* 2019 International Conference on Intelligent Computing and Its Emerging Applications (ICEA). Aug. 2019, pp. 115–119. DOI: 10.1109/ICEA.2019.8858300.

[50] Weiping Wang et al. "PDRCNN: Precise Phishing Detection with Recurrent Convolutional Neural Networks". In: *Security and Communication Networks* 2019 (Oct. 29, 2019), e2595794. ISSN: 1939-0114. DOI: 10.1155/2019/2595794.

[51] Xi Xiao et al. "CNN–MHSA: A Convolutional Neural Network and Multi-Head Self-Attention Combined Approach for Detecting Phishing Websites". In: *Neural Networks* 125 (May 1, 2020), pp. 303–312. ISSN: 0893-6080. DOI: 10.1016/j.neunet.2020.02.013.

[52] *PhishTank — Join the fight against phishing.* URL: https://phishtank.org.

[53] *OpenPhish - Phishing Intelligence.* URL: https://openphish.com.

[54] Adam Oest et al. "PhishTime: Continuous Longitudinal Measurement of the Effectiveness of Anti-phishing Blacklists". In: 29th USENIX Security Symposium (USENIX Security 20). 2020, pp. 379–396. ISBN: 978-1-939133-17-5.

[55] Simon Bell and Peter Komisarczuk. "An Analysis of Phishing Blacklists: Google Safe Browsing, OpenPhish, and PhishTank". In: *Proceedings of the Australasian Computer Science Week Multiconference.* ACSW '20. New York, NY, USA: Association for Computing Machinery, Feb. 4, 2020, pp. 1–11. ISBN: 978-1-4503-7697-6. DOI: 10.1145/3373017.3373020.

[56] Kholoud Althobaiti, Nicole Meng, and Kami Vaniea. "I Don't Need an Expert! Making URL Phishing Features Human Comprehensible". In: *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems.* New York, NY, USA: Association for Computing Machinery, 2021. ISBN: 978-1-4503-8096-6. DOI: 10.1145/3411764.3445574.

[57] Tyler Moore and Richard Clayton. “Evil Searching: Compromise and Recompromise of Internet Hosts for Phishing”. In: *Financial Cryptography and Data Security*. Ed. by Roger Dingledine and Philippe Golle. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2009, pp. 256–272. ISBN: 978-3-642-03549-4. DOI: 10.1007/978-3-642-03549-4_16.

[58] Sahar Abdelnabi, Katharina Krombholz, and Mario Fritz. “VisualPhishNet: Zero-Day Phishing Website Detection by Visual Similarity”. In: *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. New York, NY, USA: Association for Computing Machinery, Oct. 30, 2020, pp. 1681–1698. ISBN: 978-1-4503-7089-9.

[59] Kyumin Lee, James Caverlee, and Steve Webb. “Uncovering Social Spammers: Social Honeypots + Machine Learning”. In: *Proceedings of the 33rd International ACM SIGIR Conference on Research and Development in Information Retrieval*. SIGIR ’10. New York, NY, USA: Association for Computing Machinery, July 19, 2010, pp. 435–442. ISBN: 978-1-4503-0153-4. DOI: 10.1145/1835449.1835522.

[60] Neda Abdelhamid, Aladdin Ayesh, and Fadi Thabtah. “Phishing Detection Based Associative Classification Data Mining”. In: *Expert Systems with Applications* 41.13 (Oct. 1, 2014), pp. 5948–5959. ISSN: 0957-4174. DOI: 10.1016/j.eswa.2014.03.019.

[61] Martín Abadi et al. *TensorFlow: Large-scale Machine Learning on Heterogeneous Systems*. 2015.

[62] Francois Chollet. “Xception: Deep Learning With Depthwise Separable Convolutions”. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 2017, pp. 1251–1258.

[63] Dan Petro. *Pentest Screenshots*. 2021-06-10, 2021.

[64] Tlamelo Emmanuel et al. “A Survey on Missing Data in Machine Learning”. In: *Journal of Big Data* 8.1 (Oct. 27, 2021), p. 140. ISSN: 2196-1115. DOI: 10.1186/s40537-021-00516-9.

[65] F. Pedregosa et al. “Scikit-Learn: Machine Learning in Python”. In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[66] Tianqi Chen and Carlos Guestrin. “Xgboost: A Scalable Tree Boosting System”. In: *Proceedings of the 22nd Acm Sigkdd International Conference on Knowledge Discovery and Data Mining*. 2016, pp. 785–794.

[67] Isabelle Guyon et al. “Design of the 2015 ChaLearn AutoML Challenge”. In: *2015 International Joint Conference on Neural Networks (IJCNN)*. 2015 International Joint Conference on Neural Networks (IJCNN). July 2015, pp. 1–8. DOI: 10.1109/IJCNN.2015.7280767.

List of Figures

List of Tables