

MENG PROJECT

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

**Phishing Website Classification through
Behavioural Analysis**

Author: Thomas Szyszko

Supervisor: Dr. Sergio Maffei

Date: June 18, 2018

Abstract

Phishing is a form of social engineering that imitates trusted brands to trick users into giving away sensitive information, and has become one of the most common cyber threats on the planet. The internet community has fought back through adding blacklists to browsers, but with tens of thousands of new phishing sites being created every day, quick identification is a necessity, and the scale of the problem has led to the development of automated classification.

The criminals behind these attacks are aware of existing classifiers, and deploy a number of techniques to prevent detection. But whatever they do, a human must still be able to interact with a site, and so we propose a classifier that simulates a person interacting with a site, and generates a classification based on the sites behaviour.

Our innovative crawler appears to be the first to reliably solve validation on phishing pages, allowing us to access parts of a site that other classifiers never reach. In addition, we present a distributed classification platform using the latest cloud technologies that is able to scale to commercial classification rates.

Acknowledgements

I would like to thank Dr Sergio Maffeis for supervising this project. His brilliant Network and Web Security course is one the main reasons I wanted to pursue a security related project. I would like to thank Netcraft for providing me with the data needed for this project and for looking after me so well every time I came to Bath. On this front I would like to especially thank my Industrial Supervisor Charlie Hothersall-Thomas, James Long and Nic Prettejohn for their support and willingness to discuss this project at all hours of the day.

I thank my parents for always encouraging me to go further, along with my sisters Sophie, Anna and brother Richard. Finally, I would like to thank all the wonderful people I've met at Imperial over the last few years that have made it the life-changing experience it has been.

Contents

1	Introduction	5
1.1	Motivation	6
1.2	Objectives	6
1.3	Contributions	6
2	Background	8
2.1	Phishing	8
2.1.1	Targeted Phishing	8
2.2	Phishing Defence	9
2.2.1	User Training	9
2.2.2	Software Defences	10
2.3	Phishing Classification Approaches	11
2.3.1	Search Engine based	11
2.3.2	Feature based	11
2.3.3	URL based	12
2.3.4	Visual Similarity Based	12
2.3.5	Website Behaviour Based	13
2.4	Browser Automation	15
2.4.1	Headless Browsers	15
2.4.2	Browser Automation libraries	16
2.5	Measuring Classifier Performance	16
3	Initial Experimentation and Crawler development	18
3.1	Initial design choices	18
3.2	Phishing kits	18
3.2.1	Acquisition	18
3.2.2	Testing environment	19
3.2.3	Types of Login form encountered	20
3.2.4	Post Submission	23
3.2.5	Limitations	24
3.3	Live Phish	25
3.3.1	Acquisition	25
3.3.2	Target identification	25
3.4	Crawler Implementation	27
3.4.1	Form finding	27
3.4.2	Detecting visible elements	28
3.4.3	Entering values	28
3.4.4	Submitting Forms	28
3.4.5	Detecting effects of form submission	29
3.4.6	File uploads	30
3.4.7	JavaScript Alerts	31
3.4.8	DIVs as buttons	32
3.5	Defeating validation	32
3.5.1	Random data generation	32
3.5.2	Generic Matchers	33
3.5.3	Target-specific matchers	33
3.5.4	Speeding up matching	34
3.5.5	Handling non-existent or suspended pages	35
3.6	Target Identification	36
3.6.1	Overview	36
3.6.2	Domain to target mapping	36
3.6.3	Identifying pages based on network requests	37
3.6.4	Identifying targets based on links in the page	37

3.6.5	Identifying previously classified pages	37
3.7	Observations	38
3.7.1	Time per URL	38
3.7.2	Browser Resources	38
3.7.3	Concurrency with Puppeteer	40
3.7.4	Geoblocking	41
4	Large scale crawling and classification	42
4.1	Overview	42
4.2	Architecture	42
4.2.1	Amazon Web Services	42
4.2.2	AWS Lambda	43
4.2.3	AWS DynamoDB	43
4.2.4	AWS Simple Queue Service	44
4.2.5	AWS Simple Storage Service	44
4.2.6	Docker and Docker Swarm	44
4.2.7	Netcraft GeoProxy	45
4.3	Task Submission	46
4.4	Crawling	47
4.5	Classification	47
4.6	Web Interface	48
5	Evaluation	51
5.1	Per target testing for general URLs	51
5.1.1	Crawler success rates	51
5.1.2	Target Identification using redirects	52
5.1.3	Target Identification using extracted features	54
5.2	Testing using manual classification feed	55
5.2.1	Overview	55
5.2.2	Crawler success rates	55
5.2.3	Redirect to target results	55
5.2.4	Case studies	56
5.3	Scalable Distributed Classification	61
5.3.1	Resources and throughput	61
5.3.2	Cost analysis	64
5.3.3	Success by region	65
6	Conclusion	66
6.1	Objectives	66
6.1.1	Behaviour based classifier	66
6.1.2	Scalable distributed classification system	66
6.2	Future Work	66
6.2.1	Tool for easily creating matchers	66
6.2.2	Deal with IFrames	67
6.2.3	Detect which field has actually failed validation	67
6.2.4	Protect victims by flooding phish with fake data	67
6.2.5	Generate more insights into phishing campaigns	67
7	Bibliography	68
A	Target identification based on hyperlinks in the page	71
B	Target identification based on requests from a page	72

1 Introduction

You receive an email from your bank one morning. It warns you of suspicious activity on your account, and provides a link to take you to your online banking. Below it warns you, that failure to act now will result in your account being suspended. It's almost lunchtime, you need your bank card to pay for your meal. Following the instructions, you enter your login details. It seems to check out, the bank's logo is there, everything looks familiar. After hitting submit, it logs you in normally. There doesn't seem to be any strange transactions. You assume the bank's fraud department made a legitimate mistake and you return to work.

However, this is a scam. The email and the web-page it sent you to are fake. A cyber-criminal now has your banking credentials. You have been victim of a crime known as "phishing".

Phishing is defined as "a criminal mechanism employing both social engineering and technical subterfuge to steal consumers' personal identity data and financial account credentials" [1, p.2]. The social element here usually consists of spoofed emails purporting to be from a legitimate organisation to trick people into sharing personal information, such as usernames and passwords. It doesn't have to be by email, it can occur through many communication technologies, including phone calls, SMS messaging, and recently there has been a large increase through social media [2]. The fraudsters imitate brands that people will recognise, claiming the authority of that brand to psychologically manipulate victims into complying [3].

Since this is fundamentally an attack that plays on human instincts, the technical skills required by the attackers is minimal. In the case of phishing by email, which we will focus on since it is by far the most common type, an attacker is able to purchase large email lists on the black market. Thanks to the abundance of so-called "phishing kits" - tools that clone and replicate the sites of many popular targets, often available at low cost or free on the internet, all an attacker needs to be able to do is upload it to a host and send out emails containing links to it.

A typical attack could target hundreds of thousands or even millions of email addresses at once. Sending emails is cheap, and the criminal only needs a handful of those targeted to fall victim to make the enterprise profitable. Due to the globalised nature of the internet, they are further protected by difficulty in successful prosecution and extradition.

This low barrier to entry has led to phishing attacks continuing to grow, with the APWG (Anti-Phishing Work Group) reporting a 5,753% rise in monthly attacks between 2004 and 2016 [4, p.2].

In response, a number of defences have been developed. The primary shield is the use of blacklists - lists of confirmed phishing sites. Major desktop and mobile browsers (such as Google's Chrome [5] and Mozilla's Firefox[6]) utilise them to display warnings to users attempting to visit a known malicious link. It is vital that these lists are accurate, since if they report legitimate sites as suspicious, users may end up mistrusting and ignoring the warnings.

Initially, these blacklists were created by humans manually verifying user-submitted reports. However these manual-reviews are time-consuming and with around 50,000 new phishing sites created every month in 2017, can be quickly overwhelmed by the scale of the problem [7, p.3].

The criminals have also responded to this detection threat by conducting frequent, but short, campaigns, with a lifetime often of a few hours, further increasing pressure on blacklisting services. This volume of sites has led to the automation of phishing site classification.

1.1 Motivation

Today the vast majority of classifications are done automatically. For example, one of the classifiers behind Google's SafeBrowsing handled hundreds of millions of pages in 2009, with less than 1% passed on for manual review [8]. Current classification systems have improved on this statistic, with Netcraft's phishing site feed processing in the order of hundreds of thousands of URLs a day, with approximately 0.1% of classifications performed manually.

Most of these systems try to identify a reported URL as a phish by analysing the URL itself for suspicious patterns, and by analysing the page content, looking at features on the page itself, such as the presence of a brand's logo or frequency of commonly used phishing terms. If a suspected phish has enough so-called "phishy" features, it is automatically added to the blacklist. If the number of features falls below a certain threshold, it is marked benign. The sites currently in between are manually reviewed.

Many of the phishing sites that fall into this category may be visually identical to the legitimate site it is imitating. However, they often behave in a very different way when a user interacts with them. For example, a login form for a real service will usually respond with an "invalid credentials" error when presented with fake credentials. On the contrary, a phishing site could accept them, or issue some other message before redirecting the user to the targeted organisation's site.

Looking out for behaviours like this is something a human investigator may do when manually checking a page. By trying to automate these interactive methods, we should hopefully be able to significantly reduce the amount of pages requiring manual review.

1.2 Objectives

The aim of this project is to develop a classifier that will attempt to interact with a suspected phishing page, being able to identify and submit carefully crafted input into forms within the page, and monitor the result of these submissions, using this to determine whether the site is indeed phishing.

We aim to be able to perform at 100% precision, while achieving high recall rates. We also aim to achieve at least a minimum performance of around the rate of human classification - typically around one every ten seconds, but hopefully we will be able to significantly improve on this rate.

The secondary aim of this project is to develop a system that can slot into an existing classification system, receiving a URL submitted through an Application Programming Interface and returning a judgement whether it is indeed a phishing page.

Since a major bottleneck is likely to be the response time of a target site, especially since we may have to interact with multiple page loads with a target site, a key way to achieve scalability is through building a distributed platform, that will be able to interact with multiple sites concurrently. We hope to utilise current cloud technologies to build a scalable solution.

1.3 Contributions

In this document we present an innovative approach to phishing classification, by automating human interaction with a suspicious site, and using the pages response to that interaction to classify it.

We demonstrate our robust crawler design, the first we believe that is able to solve its way through multiple pages of input-validated forms on phishing sites.

We provide evidence for the observation that phishing sites tend to redirect their victims to their targeted brand.

Finally, we evaluate our approach as we attempt to apply it solve difficult targets which fail existing automated classification methods, and are currently solved by manual review.

2 Background

2.1 Phishing

The fraudulent practice of sending emails purporting to be from reputable companies in order to induce individuals to reveal personal information, such as passwords and credit card numbers.

- Oxford Dictionary [9]

A phishing site is a website that aims to trick users into entering confidential details, and then transmitting these details to the controller behind the “phish”. It usually tries to fool its victims by masquerading as the target site.

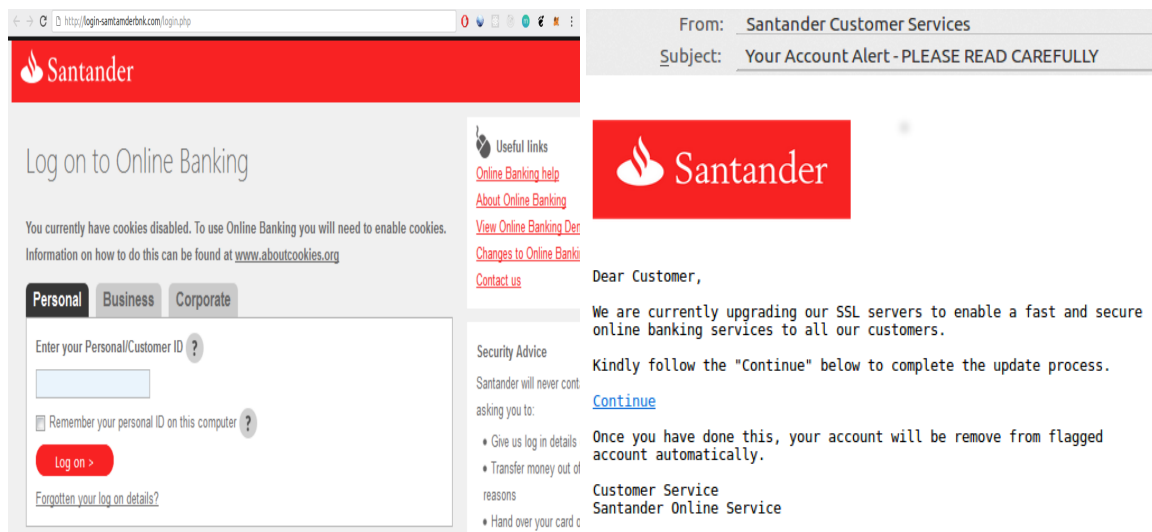


Figure 1: Example of a phishing site and email targeting customers of Santander

The victim usually receives the link to this page by email, such as the one shown in Figure 1. It contains the Santander logo to give the email the feel of legitimacy, and presents the potential victim with a seemingly credible warning, claiming that they need to login to secure their account. The page that the hyperlink points to is almost exactly mimicking the usual Santander login, ironically including the security advice .

The URL (<http://login-samtamderbnk.com/login.php>) at first glance vaguely resembles the legitimate Santander URL. Under pressure to avoid being locked out of their bank account, a user may overlook the spelling mistakes and the lack of a padlock in the URL bar (indicating a trusted Secure Sockets Layer [SSL certificate) and become a victim.

2.1.1 Targeted Phishing

The most common type of phishing is of the mass market emailing variety. This is where an attacker sends out the same or similar messages to a large group of recipients, typically thousands or even millions of email addresses at once. The general assumption is that a subset of the would-be victims will recognise or be customers of the organisation they are impersonating, while others will simply ignore the email.

However, there are some variants of phishing attacks that include personal information about the victim within the message. These are known as “spear-phishing” attacks. Here the attacker focuses on employees of a particular organisation, or even individuals themselves, using publicly available information and contextual knowledge to better entice someone to open the message. For example,

they may spoof the email address of someone high up in the company, leading workers further down the chain to feel obligated not to question the instructions in the message. This is known as a “business email compromise”.

A well known specialisation of a spear attack is known as “whaling”. This is when high-ranking individuals within an organisation are targeted, such as CEOs or board members. The account details obtained from these individuals are likely to be much more valuable for an attacker, due to the information and financial resources they will have access to.

Another variant is so-called “clone phishing”. This is where the attacker has previously seen an email being sent within an organisation containing an attachment or link . They then resend an almost identical copy of the email, switching out the attachment or link with a malicious payload. They may spoof the address of the original sender, and attach claims that the email had to be resent or that the attachment was an “updated” copy. Since this attack is based on a previously seen message, it is more likely a user will fall victim to the attack.

2.2 Phishing Defence

2.2.1 User Training

One approach that organisations take to defend themselves from phishing attacks is to try to make users more aware on how to spot phishing websites and phishing attacks. A technology research firm estimated that the security awareness training market was worth \$1 billion dollars in 2014, and appeared to be growing at 13% a year[10]. Another research firm announced that it expects this market to grow to \$10 billion by 2027[11].

However, despite the fact that some in the security community believe that “Security user education is a myth”[12], there is strong evidence that provided with effective material, users’ susceptibility to attacks does decrease. A research study that tested users ability to recognise these sites before and after studying relevant teaching materials for 15 minutes showed that the number of undetected phishing sites fell from 38% to just 12 % [13].

The issue is how to get users to focus on this material. Users are often not motivated to learn about security - it is a secondary task over which their core work takes priority. Security notices and emails are regularly ignored.

Some solutions for this problem have included turning phishing detection education into a game, with an example being “Anti-Phishing Phil” (See Figure 2). The game tests recognition skills in a scenario where users control an in-game fish, and swim into suspicious looking URLs while avoiding sharks and other obstacles. If they make a mistake, a quick lesson in the form of a hint is displayed. Another solution is for an organisation to test its staff by sending itself “fake” phishing emails. If a user fails to identify it as fake by opening it and not marking it as spam, they are presented with a short “learning moment”, teaching them how to better identify messages. Well-known products that automate this include Phish Guru and PhishMe.

Researchers comparing Phish Guru and Anti-Phishing Phil with the study focussing on traditional materials suggests both are more effective at helping users retain knowledge, with the game environment proving to be most fruitful at improving identification. Unfortunately with increased education, comes an increased rate of false reports, where users mark non-harmful legitimate emails as spam [13].

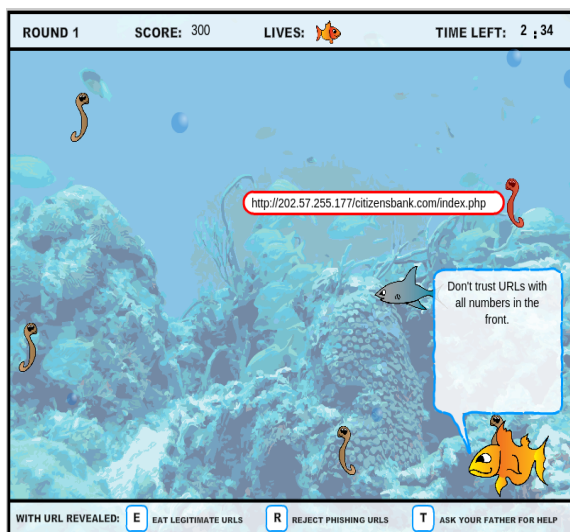


Figure 2: Anti-Phishing Phil - Developed by Carnegie Mellon University [14]

2.2.2 Software Defences

A number of software tools have been developed to try and prevent users from navigating to malicious sites. In the past browser extensions and toolbars that warned the user of suspicious sites were widespread. Often they would implement heuristics similar to checks taught to humans, such as links that contain IP addresses rather than domain names, and using http instead of https. Those that relied mainly on heuristics alone were shown by researchers to have high numbers of false alerts, displaying warnings on legitimate websites [15]. This would often lead to users simply ignoring the warnings.

Gradually these extensions have fallen out of use, mainly since many of their features have now become built into browsers and email clients. Also, the increasing complexity of websites today can lead to significant slow downs as these extensions analyse them on load.

Blacklists help solve this problem by offloading the task of identifying and verifying malicious URLs to an external provider. A blacklist is a list of confirmed phishing sites, which a browser or extension can quickly lookup a requested URL against. This is usually done on a local copy of this list, which is frequently updated to ensure users are protected when the feed providers detect a new threat. The blacklists used by common desktop browsers, such as Google's Safe Browsing which is built into Google's own Chrome browser [5] as well as Mozilla's Firefox [6], often source URLs from multiple trusted URL feeds, aiming to achieve as wide a coverage as possible. These could combine URLs identified through user reports, those found automatically by crawling the internet, to those found by setting up "honeypots", where accounts and machines are set-up deliberately insecure to encourage criminals to attack them.

Some mobile browsers also utilise these blacklists to protect their users. Chrome on Android and Safari on IOS as well as popular applications like SnapChat make use of Safe Browsing [16]. This is important since recent statistics show that mobile browsers now have a larger share of web traffic than desktop browsers [17].

2.3 Phishing Classification Approaches

2.3.1 Search Engine based

Search Engine based techniques try and extract features from a suspected phishing site, such as text, images and URLs, and then search for them in one or more search engines. They then look for the site's URL amongst the search results, and return a judgement based on the page ranking.

This is based on the assumption that legitimate sites are likely to be ranked highly due to traffic, and by existing longer than phishing sites, which tend to have very short lifespans.

This sort of analysis can be quick to perform, and in practice has been highly accurate. Varshney et al [18] reported an accuracy rate of 99.5% just by analysing the page titles and domain names of target sites. This solution also scales very well since the inspected page does not need to fully load to extract this information - the page title tends to be defined at the start of the HTML of a page. However, it operates on the assumption that all legitimate pages will have page titles, and those that don't are automatically marked as phishing. This is likely to increase the False Positive Rate (FPR), the proportion of legitimate pages incorrectly marked as phish. An increased FPR affects amateur pages that may not have set a title. Hung et al [19] used the fact that most phishing sites will contain the logos of the brands they are appropriating to design a classifier that extracts logos from screen-shots of a suspected phishing page, and then using Google's image search to determine the legitimate site's domain name, achieving a reported accuracy of 92.5%.

A large issue with this technique is that it relies on the probability of a search engine's ranking algorithm to rank legitimate sites over phishing pages. If a criminal can get their page to rank higher (perhaps by including lots of relevant keywords and linking to their page from multiple sites) this approach could mark the legitimate site as a phishing page. Also, if the phisher creates a type of page that does not exist on the legitimate site which is being impersonated, the phishing site may be the top result for many elements in the page, which depending on the algorithm used may overpower the result of the branding in the page.

2.3.2 Feature based

This is a vast category that can be broadly defined as classifiers that extract features from a suspected page's content, and then apply machine learning or algorithms that implement a number of weighted heuristics.

Moghimi and Varjani [20] claimed an accuracy of 99.14% with looking at the relationship between a page's URL and its content using a string matching algorithm. Mohammad et al [21] suggest in their paper that since phishing sites are becoming more complex and are constantly evolving, algorithms must be continually updated to maintain accuracy and low FPR. They suggest neural networks as a solution that can adapt as the phishing environment changes. While they report a lower accuracy rate - 92.18% after 1000 epochs (number of times the neural network iterated through the dataset), they suggest that their approach is quicker to apply in a general case, since traditional modelling is more dependant on "trial and error".

Machine learning over website features was an important part of Google's Safe Browsing classifier according to Whittaker, Ryner and Nazif in 2009 [8]. They state that by looking at frequencies of terms on a page, they are able to identify phishing sites even if they have "legitimate looking URLs" that are hosted on reputable hosting sites, but they were limited in the number of features they could analyse due to the computational expense of running neural networks. They claim an accuracy of over 90 % on the millions of pages they analysed every day.

2.3.3 URL based

Some classifiers do not visit or inspect a phishing site at all. They make a judgement based solely on a URL, whether focusing on lexical features (i.e. how it looks) or by extracting information about the website host itself. Not having to crawl and parse pages significantly reduces the time and computational power needed to make a classification, making this an ideal first filter in large scale classification systems.

McGrath and Gupta found that the URL lengths of phishing links tend to be longer than legitimate links, while the domain names within them are usually shorter. They found that legitimate URLs tend to have character frequencies similar to English, while phishing URLs do not. Phishing URLs also often contain the name of the brand they are targeting.

They flagged a number of observations that can be made from the domain names of these sites. Many links were using free hosting sites and URL shortening services. In 2018, use of URL-shortening has become wide-spread across the Internet. For example, on the popular social network Twitter [22], all URLs contained in tweets are automatically shortened to a `t.co` address [23]. This means that users are likely to be less suspicious of URLs containing short domain names and combinations of random characters than before since they are more likely to encounter them on a daily basis.

Criminals behind phishing sites have been using them to avoid blacklists by generating many unique URLs for the same page, and trying to avoid detection by chaining multiple URLs together in redirect chains [24]. This could also potentially happen after a user has logged into a form on a phishing site to break direct links back to the targeted brand's domain.

A significant proportion of McGrath and Gupta's phishing URLs were found to be hosted on multiple machines in multiple countries, with many on residential computers, suggesting many phishing sites are hosted on botnets.

Ma et al [25] used a variety of machine-learning techniques to produce models based on number of these observations, reporting an extremely high accuracy of 95 % - 97 % on datasets of a size between 20,000 to 30,000 URLs, with around half of those URLs being confirmed phishing addresses. The source of these phishing URLs was either the PhishTank [26] blacklist or a University of California tool called Spamsscatter that detects and analyses spam emails and extracts URLs from contained links [27].

They claimed very low FPRs, but didn't provide any exact statistics other than one of their datasets using the Phishtank blacklist had 60 False positives. Assuming the dataset contains 15,000 URLs, this achieves a FPR of around 0.4%.

2.3.4 Visual Similarity Based

Visual similarity classifiers try to compare visual features of phishing sites with corresponding legitimate sites. While they may use different features and algorithms, there are generally two categories, Document Object Model (DOM) and screen-shot based classifiers.

DOM-based classifiers inspect visual features by analysing the HTML, CSS and JavaScript code that generates elements, whereas screen-shot classifiers render a page in a browser, take a screen-shot, and then extract features at a pixel level. Rendering and subsequent analysis can be very computationally expensive.

An example of a DOM-based classifier is BaitAlarm [28]. It compares the similarity of the CSS structure of a page. The authors claim a 100 % accuracy, however the testing dataset only consisted of confirmed phishing sites, making this figure a bit misleading. An issue with comparing CSS structure is that a phishing site could replicate the look and feel of a site by taking a screen-shot and setting it as a background image, overlaying it with form fields for entering details. The pages could look

identical, but the CSS structure would suggest they are completely different.

A screen-shot based solution would not suffer from this deficiency, since it considers the final look of the page, regardless of the underlying web page elements that generated it. Teh-Chung et al system based on screen-shots claims a very low FPR of less than 0.01 %. [29].

However since both types of classifiers are comparing visual elements, they need a pre-built set of images/ signatures to compare against. This makes them only really suitable for identifying phishing pages of a select few brands, rather than identifying phishing pages in general.

2.3.5 Website Behaviour Based

In 2008, a paper by Salikar et al [30], when introducing the concept of Public Key embedded Graphic CAPTCHAs¹ suggested the possibility that phishing websites could be identified by the fact that they often would accept any credentials supplied. They then describe a scenario where an intelligent attacker actually attempts to sign in to the legitimate website using them, thereby verifying their correctness in real time. He suggested that more phishing sites in the future are likely to incorporate elements of Man in the Middle (MITM) attacks, and proposed a defence mechanism based on specially encoded CAPTCHAS.

Later in the year, they were co-authors on a paper [31] that stated since the majority of phishing attacks were not MITM, the idea would be worth exploring further. They developed a Mozilla Firefox browser plug-in that attempted to identify phishing sites by hijacking the browser's auto-fill feature to extract a user's inputted credentials. When the user clicked submit, it would randomise the password field, and attempt to login to the site using the generated fake password. If the website returned a Hypertext Transfer Protocol (HTTP) status code 200, indicating a successful login, then the plug-in concluded that the page was indeed a phishing site and did not submit the user's real credentials.

If the site returned a HTTP code 401, indicating an authentication failure, it would try submitting different fake credentials several more times. If the website returned some other status other than failure, it would conclude the site was randomising its response and classify it as a phish.

Finally if the site continued to return HTTP 401, it would submit the real credentials. If the site returned HTTP 200, the plug-in would conclude the site was legitimate. If it continued to return HTTP 401, it would assume that the site was a phishing site that was simply always returning an authentication failure. In order to prevent the phisher from identifying the legitimate password by using the last one submitted, it would post several more fake password login attempts in order to bury the legitimate password and make it less likely an attacker would try it.

This plug-in makes an assumption that all sites use HTTP Digest as an authentication method, ignoring all other available methods (including HTTP basic auth!). HTTP Digest is a process that sends a MD5 cryptographic hash of the user's credentials using a nonce value sent by the server. Since HTTP Digest doesn't send a users credentials in plain-text, while there are known weaknesses in it, it is unlikely an attacker would use it. Also, most authentication these days is done using HTTP forms over HTTPS. Modern sites (or phishing sites defending from this sort of analysis) may rate-limit the number of login attempts in a period of time, and may redirect to a login page (with a HTTP 302) instead of returning 401. The paper does not state the results of any testing.

At the same time Yue et al [32] proposed a similar method for defending user's credentials, resulting in an extension called Bogus-Biter. They identified that sending multiple incorrect logins to a legitimate site may result in a user's account being temporarily suspended, but also suggested that legitimate sites may present CAPTCHAS after an unsuccessful login. In a similar manner, a phishing

¹Completely Automated Public Turing test to tell Computers and Humans Apart - a challenge-response test that determines whether a user is human or not.

site could evade possible automated classification by implementing its own CAPTCHA system.

Bogus-biter was purely a way of defending a user's credentials. It did not try to classify websites itself, instead relying on a hit from an external blacklist to trigger it. However, it did extract user-name and password fields from the DOM of a web-page directly, making it useful for the vast majority of web-based logins.

In 2017, Rao et al [33] presented a classifier that would try to login to a potential phishing site and apply a filtering process to determine whether it was indeed a phish. The classifier was a standalone Java application that would interact with a Firefox browser controlled using the Selenium browser automation framework.

First of all, it would attempt to find a login box on the page. The authors identified three potential areas to search in - the Home page of the URL (i.e. the first page encountered when the URL was visited), a modal window² or Iframes³ embedded in the page.

Login modals are identified by looking for keywords like "sign-in" or "login". They state that attackers may use Iframes to hide from classifiers that analysis the HTML source of the Home page. To solve this problem the classifier iterates through every Iframe it finds until it finds a login form. It identifies the presence of a login form by locating a password field in the source (See Fig 3). This seems like a reasonable assumption since marking a field with a type "password" causes input into it to become shielded. User's would be quick to spot passwords appearing in plain text, and while possible, it seems unlikely an attacker would implement this using a text input with JavaScript to spoof the shielding.

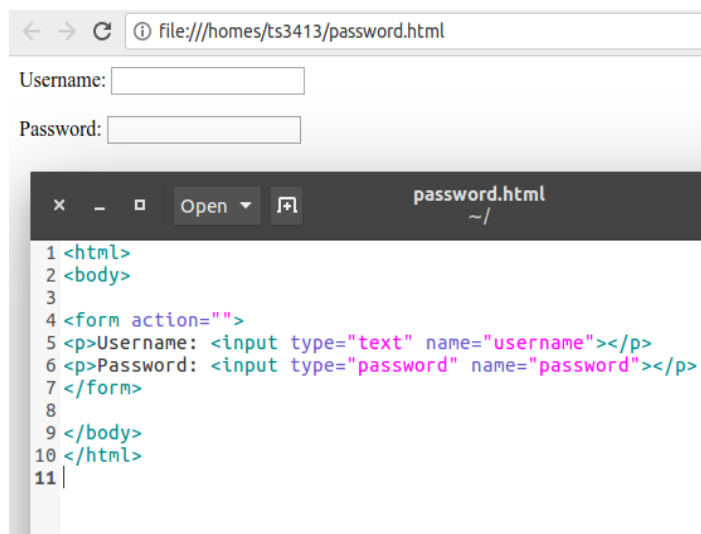


Figure 3: Example of password field in HTML source

With the login form identified, the classifier inputs fake data into the form fields. It makes no attempt to try and please any potential client-side validation of values (i.e. The attacker may embed JavaScript within the page that verifies that input to a field labelled "email" is in fact in a valid email form). They state an assumption that this validation code is likely linked to the "click" method of the submit button rather than to the "submit" action of the entire form. The authors acknowledge that phishing sites with better validation will escape this filter.

With the form submitted, the classifier looks for the presence of another password field on the subsequent page, the reasoning being that an unsuccessful sign-in on a legitimate site would return the

²a mode where the main window is disabled, but kept visible, with the user being required to interact with window displayed over it before being able to return back to the main window.

³Iframes allow a HTML page to be split into segments containing separate, independent documents.

user back to the login form again. Identifying a password field is analogous to receiving a HTTP 401 in the Firefox classifier described earlier [31]. This lets them avoid having to analyse potential error messages returned by websites, however many legitimate sites exist that display a warning on an unsuccessful login, and then ask a user to click to go back to the login form. These sites would be marked incorrectly as phishing sites using this scheme.

If the classifier identifies an unsuccessful login, it then checks a number of heuristics. It has a check that see if a page has zero links in it. A typical login page will have some links present (forgot password button, home page link, etc). When introducing Visual classifiers we presented a phishing technique where the phisher is able to mock a legitimate site with a simple snapshot of it with some HTML forms placed in front of the picture. This check targets these sites, and marks them automatically as phishing.

Finally, they check to see how many links to itself a page has. The authors reason that phishers tend to leave most hyper-links within a page as “null” links (which creates a link to the same page).

One issue that the paper doesn’t seem to resolve is what happens if a phishing site redirects to a legitimate site on login, and this page also contains a login field. This will be marked as an unsuccessful login, resulting the page being subject to the heuristics filter. It isn’t clear if the home page is then analysed or the redirected page. If it is the redirected page, then this would incorrectly mark the phishing site in this case as legitimate.

The classifier was tested against 2342 websites, of which 1459 were phishing sites sourced from PhishTank and 883 legitimate sites sourced randomly from the Alexa web list. A very high recall of 97.53 % was reported, but with a relatively high FPR of 5.63 %.

2.4 Browser Automation

2.4.1 Headless Browsers

In the past, if someone wanted to analyse programmatically the content of a website, they would simply download the HTML content of a page (using a command line tool such as curl), which they would then run through a HTML parser to extract elements such as form fields. However, to defend against this form of static analysis, many phishing sites will dynamically generate the contents of a page using (possibly obfuscated) JavaScript. In order to view the elements on a page like this, we need to be able to execute the JavaScript and allow it to render. We then need to be able to explore the page elements that have been created to identify the features we are looking for.

Popular browsers such as Mozilla’s Firefox and Google’s Chrome have been difficult to use in the past for this purpose since they previously could not be run without generating a Graphic User Interface [GUI] window, which comes with a high resource and start-up time cost. While it is possible to “fake” a screen, such as running an X Server with a virtual framebuffer on Linux using XVFB[34], this introduces a significant resource and software dependency overhead. Worse still, the browsers did not provide easy APIs to control the browser’s actions. There are some projects that use the development interfaces to provide automation frameworks, such as Selenium.

The recent growth of the use of Continuous Integration (CI) amongst web developers has resulted in a number of lightweight scriptable browser engines for the purposes of running user interface (UI) tests, such as PhantomJS, which up till recently was a de facto standard for running CI tests. However, despite implementing the majority of current browser standards, it is not running identical code to any of the major browsers (one of the major complaints being that it runs a much older version of the WebKit engine that many modern browsers are based on), meaning there can be differences in how sites are rendered. Worse still, some sites might serve different versions of the page depending on the browser accessing it. While a simple user-agent check can be easily spoofed, fingerprints based on the actual browser environment a page is loaded in can be much harder to trick.

In response to the huge demand for automated testing within real, commonly used browsers, Google released Headless Chrome as part of Chrome 59 in June 2017[35], with Mozilla quickly following with headless mode in Firefox 56[36], released in September 2017. Since the most used browser in the world in 2017 was Google Chrome [37], Headless Chrome is quickly becoming the new standard, with many other industry standard libraries quickly being discontinued by their maintainers, including PhantomJS[38].

2.4.2 Browser Automation libraries

There are two popular automation libraries for Google Chrome: Selenium [39] and Puppeteer[40]. Selenium is a long established cross-browser automation tool. It has a well-documented API and allows scripting in 10 languages. However, due to the need for it to support multi-browsers, it is not highly optimised against any of them, and is known to be relatively slow.

Puppeteer is a Google supported project which implements a very similar API to Selenium. It only supports Google Chrome, however it is much more optimised for it. This is likely to be important when designing an efficient and scalable cloud-based classifier. It currently only supports Node.JS.

2.5 Measuring Classifier Performance

To test the accuracy of classifiers, researchers use training sets of known phishing sites and legitimate sites, and check the rate of correct classifications. These sets can come be constructed from a number of sources, for example one could use the URL feed from a blacklist provider such as PhishTank or Netcraft, and for legitimate sites extract URLs from popular web services from a service such as Amazon's Alexa Top Sites Service.

The measure commonly used for classification to compare accuracy is a coincidence matrix [41, p.137]:

		True Class	
		Positive	Negative
Predicted Class	Positive	# of True Positives	# of False Positives
	Negative	# of False Negatives	# of True Negatives

Table 1: Coincidence matrix, adapted from [41, p.137]

- **The number of True Positives (TP)**
This is the number of known phishing sites that the classifier correctly marked as phishing.
- **The number of True Negatives (TN)**
This is the number of known legitimate sites that the classifier correctly marked as legitimate.
- **The number of False Positives (FP)**
This is the number of known legitimate sites that the classifier incorrectly marked as phishing.
- **The number of False Negatives (FN)**
This is the number of known phishing sites that the classifier incorrectly marked as legitimate.

From these parameters we can extract the following metrics:

$$\text{True Positive Rate (TPR)} = \frac{\text{\# of True Positives}}{\text{\# of True Positives} + \text{\# of False Negatives}} \quad (1)$$

$$\text{True Negative Rate (TNR)} = \frac{\# \text{ of True Negatives}}{\# \text{ of True Negatives} + \# \text{ of False Positives}} \quad (2)$$

$$\text{False Positive Rate (FPR)} = \frac{\# \text{ of False Positives}}{\# \text{ of False Positives} + \# \text{ of True Negatives}} \quad (3)$$

$$\text{False Negative Rate (FNR)} = \frac{\# \text{ of False Negatives}}{\# \text{ of False Negatives} + \# \text{ of True Positives}} \quad (4)$$

$$\text{Accuracy} = \frac{\# \text{ of True Positives} + \# \text{ of True Negatives}}{\# \text{ of True Positives} + \# \text{ of True Negatives} + \# \text{ of False Positives} + \# \text{ of False Negatives}} \quad (5)$$

$$\text{Precision} = \frac{\# \text{ of True Positives}}{\# \text{ of True Positives} + \# \text{ of False Positives}} \quad (6)$$

$$\text{Recall} = \frac{\# \text{ of True Positives}}{\# \text{ of True Positives} + \# \text{ of False Negatives}} \quad (7)$$

For the purposes of comparing phishing classifiers, the important metrics are the recall (i.e. the proportion of phishing sites that are correctly identified by the classifier) and the False Positive Rate (FPR), which is the proportion of legitimate sites incorrectly identified as phishing by the classifier). The FPR is arguably the more important metric of the two; research has shown that if a user is presented with multiple false warnings on sites they know are legitimate, they will trust them less and start to ignore the warnings, defeating the purpose of classification in the first place [42, p.7].

However, a high recall rate is also important, since if many threats are not detected, users will not be adequately protected.

3 Initial Experimentation and Crawler development

In this section we will present our initial design choices; how to automate the interaction with phishing sites, and outline the design of a "crawler", the program that controls interaction with a suspect site, and returns the observations necessary to perform a classification. We will use the results from the interactions with the phishing sites to iteratively improve our initial design.

3.1 Initial design choices

An initial starting point for this project was to attempt to implement a similar mechanism as described by Rao et al [33] of controlling a browser using an automation framework, but instead of using the Selenium automation framework together with a full version of Firefox, we chose to use Headless Chrome(ium) with Puppeteer. Aside from having a more mature support for headless operation, Chrome has a significant majority in current world-wide browser rankings (58 % compared to Firefox's 5 % in May 2018 [43]), making Chrome support the de facto standard for web-site designers, including the phishers.

Puppeteer's official support by Google means that breaking changes in the Devtools API used to control Chrome (which occur with surprising frequency) are less likely to be an issue. Each version of Puppeteer ships with a specific version of Chromium so we don't need to actively track the change logs with every update.

Unlike the previously cited example, we chose to use JavaScript instead of Java for all of our project code. While the main reason for this is that Puppeteer only provides a JavaScript API, we can also standardise the entire code-base since any direct interaction with a web-page must be done in JavaScript, and any supporting libraries are likely to provide a NodeJS client.

3.2 Phishing kits

3.2.1 Acquisition

Since the lifespan of a typical phishing URL is in the order of days, if not hours, and there is a need for a stable environment for initial development, we started by attempting to solve the password forms on self-hosted phishing kits.

We were able to obtain a number of phishing kits from Netcraft for a number of common targets. Users of these kits often don't delete the zips of the kits they have used when deploying them, and if directory listing is enabled or by trying common file-names, ready to use kits can be found.

A fair number of the kits we received required tweaking of the environment they were running in to operate correctly while some completely refused to work. The fact that many of these kits are run on compromised web-pages where the attacker has limited control of the execution environment might help explain why when later testing on live phish, we experienced so many partially working pages.

Many of the working kits we received contained login forms targeting a number of different brands, giving a sizeable set of login forms to practice on.

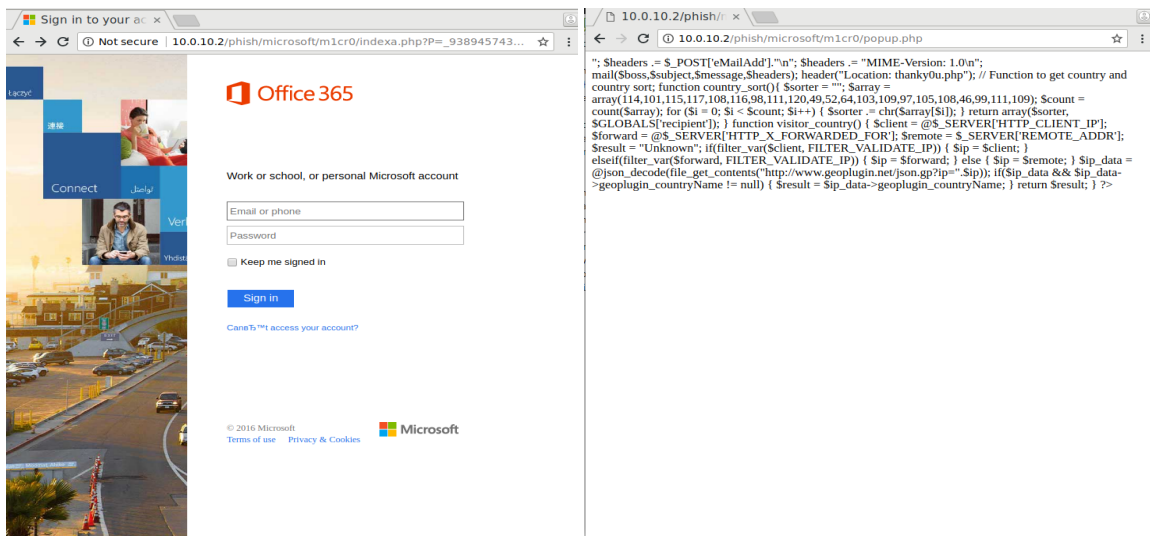


Figure 4: Example of a locally hosted phishing kit that fails on submission due to PHP environmental issues

Target	Number of Logins
Microsoft (including live and outlook)	9
Generic WebMail	7
Google (including gmail)	6
Yahoo	5
AOL	4
Paypal	5
Dropbox	2
Fedex	1
ASB Bank	1
Alibaba	1

Table 2: Breakdown of working logins by target

3.2.2 Testing environment

Since most of the phishing kits contain server-side code which could potentially be malicious, we created a sand-boxed environment for testing. This consisted of two Virtual Machines (VMs), with one running a standard LAMP⁴ stack on current release of the Debian Linux distribution hosting all the provided phishing kits, and the other running a current version of Ubuntu Linux, containing our crawler code as well as a copy of the WebStorm IDE to allow quick prototyping and modifications without copying code into the VM.

The two VMs were networked together using a private network within VMWare, and then outgoing traffic from the Ubuntu VM was firewalled using IPTables. A dummy version of sendmail was added which captured all calls from the PHP engine in Apache and logged them, allowing us to see what a phisher would receive from kits that rely on email rather than POSTing form results directly from the webpage.

Interestingly around half the kits were completely self contained in terms of resources such as images, whereas the other half relied on external resources to display correctly.

⁴Linux, Apache, MySQL, PHP

3.2.3 Types of Login form encountered

From the login forms found in these phishing kits we were able to solve a number of different styles of login.

Simple Form

The simplest case involves the page serving a simple HTML form containing username and password inputs. The password field in seen cases has a type of password set, meaning the browser masks any input, making the form seem more legitimate. Since there are only two text input elements on the page, the other input can be assumed to always be a username or email input (the only other example seen when viewing real live sites is bank account numbers or customer ids).

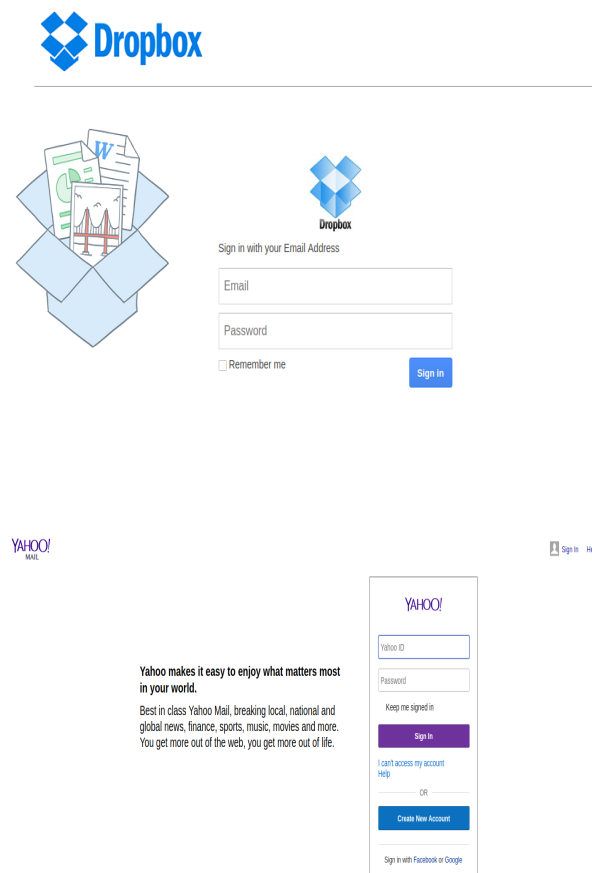


Figure 5: Examples of simple forms

Multiple account logins/ SSO

In order to obtain credentials from as large an audience as possible, phishing kit creators often target multiple accounts at once. These are sometimes known as Single Sign-On (SSO) logins. In the

absence of any visible inputs on a page, we click on all the links on a page to see if they lead to a login field.

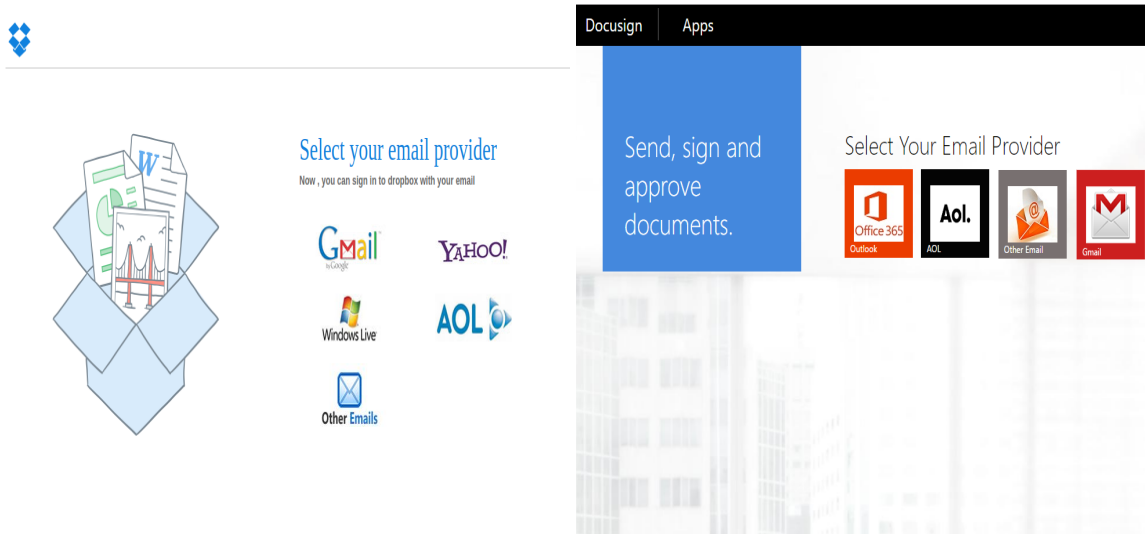


Figure 6: Examples targeting multiple providers

Pop-up login

While some of the SSO examples redirect to a separate page per target login, the vast majority use JavaScript to produce a different login window depending on the target login chosen by the victim. A simple method a phishing kit can use to present realistic logins for multiple targets at once is to simply clone the HTML of the genuine login forms, attach their own code to handle form submissions and display it in a pop-up window. We are able to deal with these easily by simply taking the last created page off the stack of open windows when searching for forms.

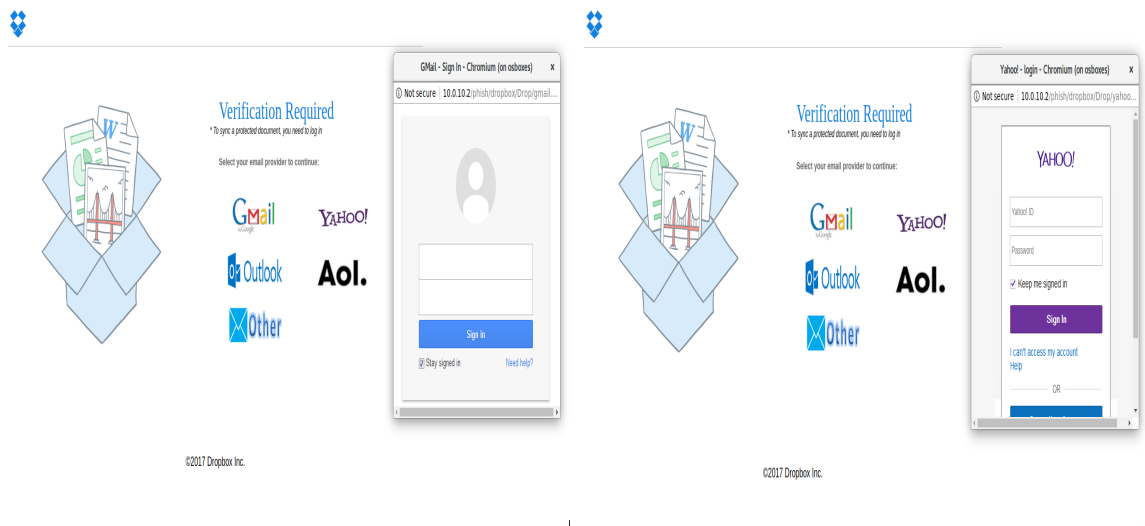


Figure 7: Example of a kit displaying different pop-ups depending on target login chosen

JavaScript modal login

Sometimes the JavaScript generates a modal instead of creating a pop-up. This can make deployment easier, since a phishing kit can be entirely self contained within a single HTML file. When a victim

clicks on a particular brand's login button, the kit can make a modal appear, displaying that brand's logo next to it. Often the form is identical for all targeted brands, with only the logo changing. The crawler described in Rao et al's [33] paper is unable to handle these types of sites, citing an increased "complexity of the detection process". This is probably due the fact that they extract the HTML of a page using Selenium and then parse it with JSoup [44], a separate Java based HTML parser. By using Puppeteer, we are able to query the Document Object Model (DOM) of a page directly using injected JavaScript to see if more inputs appear after clicking a button.

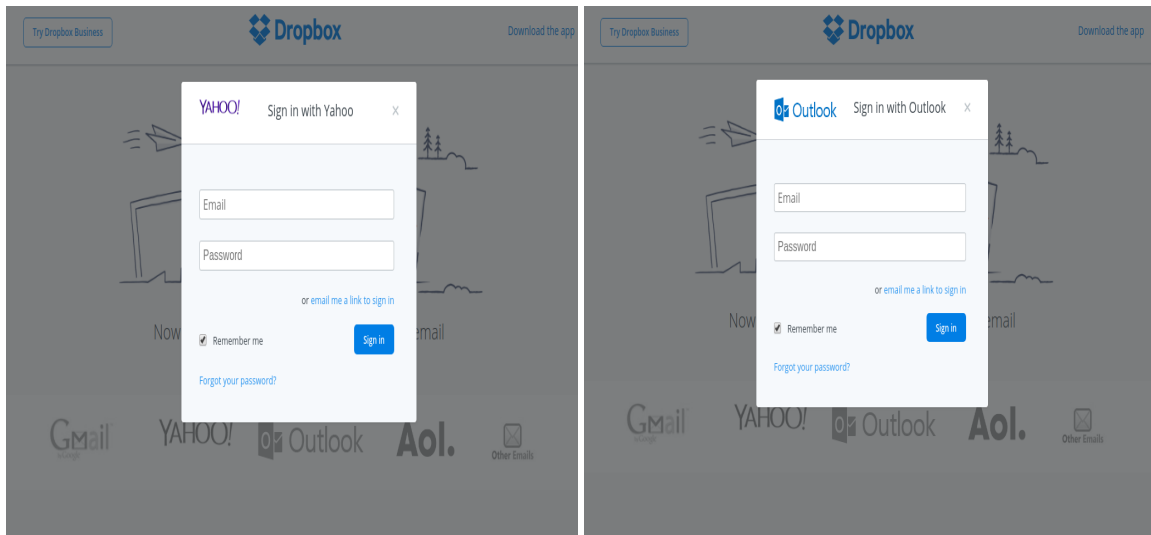


Figure 8: Example of a kit using JavaScript to change the logo and title depending on brand chosen

Multi-part login

As more online service providers provide the option of two factor authentication (2FA) login that does not require password entry but instead relies on a user having an application on a mobile device, many login forms do not contain a password field. Often they prompt the user for a username or email address, and then after checking the supplied credential against their database, either present a password form or a 2FA confirmation depending on the authentication options enabled on that account. Both Microsoft and Google authenticate users in this way. As a result, phishers mimicking these forms may split username and password inputs onto separate pages. None of the examples of previous work attempt to deal with these forms. Rao et al's crawler [33] uses the existence of a HTML input with a type attribute set to "password" to detect a login form. Since the password will tend to be requested on subsequent pages from the username, they would not be able to classify such phishing sites.

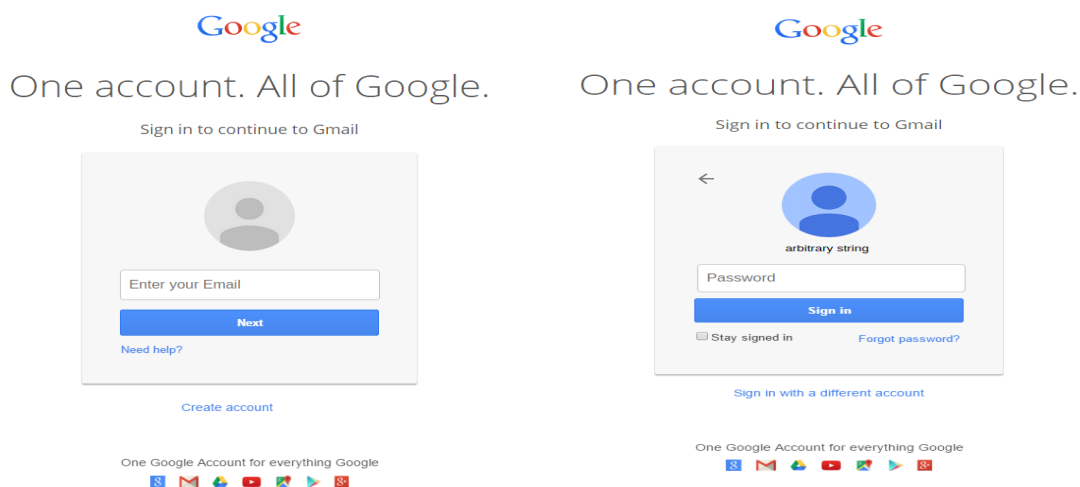


Figure 9: Example of a login spread over two pages. Notice that any input is accepted as a username

Client-side validation

While many of the kits accepted any data entered, a number performed client-side validation on the username field. The most basic validation which is present on almost every phish simply ensures that all fields have data entered into them. This is either achieved by adding the HTML "required" property (which means the browser will not submit the form if nothing was entered), or explicitly implementing this check in JavaScript by attaching a listener to the form submission event and checking the values entered. The checks were rather simple, either checking to see if an input was greater than a certain length, or that it follows a certain pattern. Interestingly, one of the kits would reject any email address entered that had capital letters in it, due to a poorly written regular expression⁵ that only included lower-case letters. This sort of validation was easy to solve by simply resubmitting the forms with different types of common credentials until the page accepted the value. However, this approach would prove to be much less effective when dealing with many live phishing sites.

3.2.4 Post Submission

Unfortunately determining the post-login behaviour of these phishing kits was made difficult due to a large percentage of them returning PHP errors post form submission. However, by analysing the source code of the kits we received, we were able to produce the results shown in table 3. 8 of the 21 tested kits (40 %) instantly redirected to a domain owned by the brand the kit was targeting. A further 8 requested additional information from the victim, after which the majority seemed to redirect to the target (the 2 kits we are unsure about seemed to contain redirect code, but it was not immediately obvious where they were redirecting to). The rest displayed error messages, or redirected to unrelated content.

⁵A special text string for describing a search pattern

Target	Num of login forms	Post-login behavior	Second page contained URLs pointing to target?	Final URL on target domain?
Alibaba	1	Redirect to Alibaba	yes (post redirect)	yes
ASB	1	Additional forms (possible redirect)	yes	unknown
Dropbox	5	Redirect to Dropbox	yes (post redirect)	yes
Dropbox	1	Redirect to externally hosted PDF document	no	no
Dropbox	5	Redirect to Dropbox	yes (post redirect)	yes
Dropbox	4	Additional forms and redirect	no	yes
Dropbox	5	Display error, then redirect to Dropbox after 6 seconds	yes (post redirect)	yes
Dropbox	5	Redirect to PDF hosted on Dropbox	no	yes
Dropbox	1	Local page full of images of houses	no	no
Dropbox	5	Redirect to Dropbox	yes (post redirect)	yes
DocuSign	4	Display error message	no	no
Fedex	1	Additional forms (possible redirect)	yes	unknown
Generic	1	Display error message	n/a	n/a
Generic	1	Display error message	n/a	n/a
Microsoft	1	Redirect to Microsoft	yes (post redirect)	yes
Microsoft	1	Redirect to Microsoft	yes (post redirect)	yes
Microsoft	6	Redirect to PDF hosted on Google Drive	no	no
Paypal	1	Additional forms and redirect	no	yes
Paypal	1	Additional forms and redirect	no	yes
Paypal	1	Additional forms and redirect	no	yes
Paypal	1	Additional forms and redirect	no	yes
Paypal	1	Additional forms and redirect	no	yes
Yahoo	1	Unknown – seems like a file is missing	unknown	unknown

Table 3: Analysis of post-login behaviours of tested phishing kits

3.2.5 Limitations

While obtaining and self-hosting phishing kits provides a steady and reliable environment in which to develop a crawler in, the manual set-up involved (obtaining, installing and manually testing to check it is operating properly) means that the sample size is small. This means it is likely not representative of the sites currently live on the web, bearing in mind the evolutionary rate of these pages.

Full kits tend to be obtained from sites that have left compressed versions of source files publically accessible, mistakes which more sophisticated phishers are less likely to make. Once the source code of a kit is known, it becomes easy for security companies to write automated methods to detect them, meaning sites using these kits are much less likely to appear in a manual classification queue.

3.3 Live Phish

3.3.1 Acquisition

In order to be able to interact with live phishing sites, we would need a reliable source of live URLs. We initially tried the PhishTank [26] feed as used in many of the papers referenced in the background reading. However, we quickly realised that this is a poor source for live phish for the many commonly targeted brands. For example, we decided to visit all URLs targeting Amazon.com from the "online and valid" feed. Out of the 50 URLs returned from the API, only three presented a classic login form. Four were of a prize scam variety, where after clicking on a simulated roulette wheel, the user is informed they have "won" a prize, and prompted to enter their account details. However, the vast majority of links (17) served blank or error pages, and 13 links pointed at seemingly unrelated content. A full breakdown can be seen in table 4.

Page type	Count
Unrelated	13
PHP / Database/ Blank page	12
404 Not Found	5
Survey	5
Claim your prize	4
Login form	3
Suspended	3
Legitimate (looking) blog	2
Instant redirect to Amazon	2
Image	1
Total	50

Table 4: Survey of PhishTank URLs for Amazon.com

A likely reason for this is that many of the most targeted brands employ security firms to actively remove phishing sites targeting their brand, and since PhishTank requires a few contributors to confirm a valid identification, the links may be down by the time they are entered into PhishTank's feed.

To gain a more reliable source of URLs, Netcraft provided access to confirmed URLs from their phishing feed. We were also able to filter the list to ensure only actual phishing sites were tested (e.g. avoiding scams, fake surveys etc.).

Page type	Count
Login form	295
PHP / Database / Blank page / 503	38
Payment form	34
Instant redirect to Amazon	13
404 Not Found	11
Suspended	2
Total	393

Table 5: Survey of Netcraft's take-down URLs for Amazon.com

3.3.2 Target identification

Initially we tried to follow the method suggested by Rao et al [33] to see if we could correctly identify a URL as an Amazon phish. Their method looks at all the hyperlinks on the page returned after filling out a form containing a password field. However, we found that that out of 295 successful form submissions, only 27 contained hyperlinks pointing to a domain owned by Amazon, less than 10%. Also, 34 pages had hyperlinks pointing to other targets, which could potentially mean target

misidentification, especially since some of those had no links pointing to Amazon. A breakdown of our findings can be seen in table 6.

Hyperlinks post login	Count
Contains only '#' and '' and null hyperlinks	193
Contains non Amazon external links	34
Contains links to Amazon	27
Contains relative hyperlinks	19
No '#' or '' or null hyperlinks	15
Contains local query string	5
Contains no hyperlinks	5

Table 6: Breakdown of href values of hyperlinks post filling in password

However, we found that the vast majority of the sites would actually redirect to Amazon after filling out all the forms on a page. Unlike Rao et al's method, this also holds true for the sites where the initial form did not contain a password field. Out of all the links that did not return an error page, 274 out of 342 (over 80 % !) redirected to another domain, with only 2 of them redirecting somewhere other than to an Amazon owned domain (they both linked to another Amazon phish).

Redirects to Amazon after Page	Count
0 (instant redirect)	13
1	13
2	16
3	45
4	16
5	166
6	2
7	1
Total	272

Table 7: Average number of form pages filled until redirect to Amazon

Interestingly, the vast majority of the phishing sites presented multiple forms and button presses to the user before redirecting to Amazon, with the most common containing 5 pages of forms. These sites don't just request login credentials, they ask for billing information, credit card numbers, social security numbers, and two factor authentication (2FA) pins, with most of the forms containing JavaScript or server-side validation of these inputs (e.g. a credit card field will only accept 16 digit numbers). The data suggests that if we can reliably fill these forms out, we should be able to reliably identify the target of a phishing page through page URLs alone.

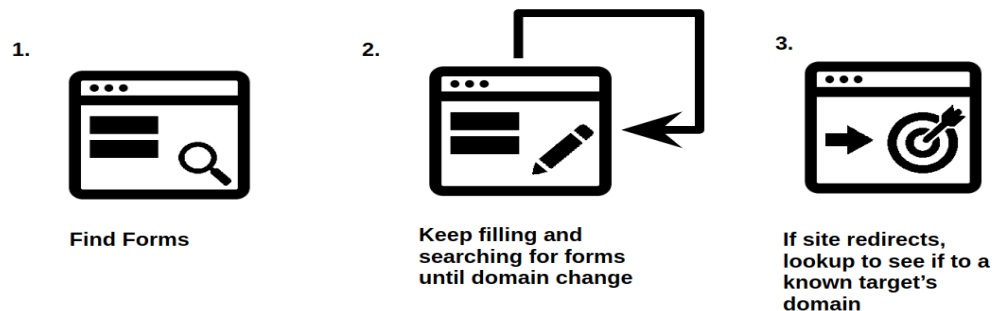


Figure 10: Target identification plan

3.4 Crawler Implementation

Here we describe the technical implementation of our crawler code used to produce the previously presented analysis, and that was subsequently added to the Feature Extraction module of our distributed classification system presented in Section 5.

3.4.1 Form finding

In order to extract elements from a page we use Puppeteer's `evaluate()` method to inject JavaScript into the browser context, similar to if we were using a developer console in a browser. This is implemented by converting the supplied function and arguments to a string [45], and executing it in the browser, returning any results in a similar manner. This means that we can only pass absolute values and objects that can be stringified between the two environments - any external referencing such as closures will not work.

We can then use the browser's native `document.querySelectorAll` method to return references to all inputs on a page, as well as other potentially clickable elements such as buttons, hyperlinks, images, divs and select elements. If a page does not contain any text or numerical inputs, we perform a 1 level deep search by clicking on each element and seeing if any input forms appear, either using JavaScript or by loading a new page. We found that many phishing sites displayed fake loading spirals between forms, sometimes as long as 20 -30 seconds. After much trial and error, we found forcing a wait of between 15 - 20 seconds after each click let us detect the majority of delayed forms or redirects. However, this does significantly slow down the crawling process.

If no form is detected, we want to navigate the browser to the URL of the previous page, and try clicking on the next object in the array. However, this is not as easy as it sounds, since we cannot pass DOM references using `evaluate()`. Puppeteer tries to solve this problem by providing `Element Handles`, a sort of mapping between DOM references in the browser context and objects in Puppeteer's context. It means you can extract references from a web page and they use them in commands such as `click()`, which simulates a mouse click on that object. But since we modify the DOM by clicking on links (and reset it completely by navigating to other pages), those mappings are lost with almost every simulated click!

As well as `Element Handles`, most methods in Puppeteer's API accept CSS selector strings in order to select and interact with objects in a page. CSS selectors are patterns used to identify elements in a web-page, often used for styling purposes. They enable a user to reference an object by class type, id attribute, HTML tag type or a combination of all three. Since the DOM stores all objects in a tree-like structure, we can use the previous rules to match with a known object, and then look a certain number of objects below it to find the one we are targeting. This can be important to generate unique selectors, since there are often many elements on a page with the same class or tag type, (and many phishing sites contain invalid HTML, often reusing id tags even though they should be unique identifiers). Neither Puppeteer nor Chromium provide a built in way of generating selectors (there is a CSS selector generator in Chromium's DevTool console but it can only be accessed through the GUI).

Generating unique but efficient and reliable selectors is a research problem in itself, and there are a number of JavaScript libraries that each offer their own balance of speed, uniqueness and reliability. While trialling these libraries, reliability would prove to be an issue with a number of them, with the badly formed HTML of a number of phishing sites causing them to crash or return junk selectors. After a few weeks of trial and error, we settled on `SimmerJS`[46], a mature CSS selector engine maintained on GitHub since 2014. Even when set to traverse the DOM to a very deep level (100 levels) which the authors warn could make it more breakable, it has proven reliable and stable.

In order to inject the library into the page, we use Puppeteer's `addScriptTag()` method, which copies the minified JavaScript code directly into the DOM of a page. When `Simmer` initialises, it can be accessed through a page's window object at `window.Simmer`, which we can then use together with `querySelectorAll()` to generate selectors for any object on a page. To ensure that `Simmer` is always

available, before calling `Simmer` we check to see if the object is set, and re-inject if necessary.

One severe limitation of this method is that sites with strict Content Security Policies (CSP) set that block scripts added from a local source will prevent `Simmer` from running. We have not encountered any phishing sites where it has actually been blocked, but a number of legitimate sites have. `Puppeteer` recently introduced a `setBypassCSP()` method that works by intercepting page loads and removing the CSP policy before it is loaded by the browser, however in practice it proved unreliable and we have disabled it for now.

3.4.2 Detecting visible elements

As seen in the phishing kit examples earlier, some sites may contain multiple forms on a single page, with JavaScript listeners making a particular form visible depending on what options a user clicked on. Therefore, in order to know that we are filling in the correct form, we need to filter out the extracted inputs so that we only interact with those that are visible. We use `Puppeteer`'s `waitForSelector()` method to do this. It is usually used to prevent a script executing until a particular element has been created on a page, and it has an additional extra parameter to wait until that object is visible from a users perspective. By setting an extremely short timeout (20ms), if it returns true then the object is visible, but if it times out then it is not.

3.4.3 Entering values

We are able to interact with a number of HTML input types. The simplest are ones which produce a text box (e.g. `type = 'text', 'tel', 'email', 'password'`). First we clear the text box of any placeholder text or previously entered value by setting its `value` attribute to an empty string. We then use `Puppeteer`'s `click()` method to simulate a mouse click in the centre of the text box, and then use `Puppeteer`'s virtual keyboard to simulate a user typing characters. Sometimes the way a page has rendered can cause the `click()` method to fail to select a box, so we check whether the `value` attribute of a box is still empty. If it is, we set the `value` attribute to match the text we wanted to enter. This is the less preferred method since some sites perform validation as text is being entered, and we can use the difference between the string we virtually typed in and the resulting value of the text box as one of our methods to detect failed validation.

The next type is the `Select` input type. This generates a drop-down menu with a number of defined options for a user to select. `Puppeteer` provides a `select()` method that provided with the selector of a `Select` input and a value, will search for that value in the list and click it. Currently we use `evaluate()` to extract an array containing all the values and randomly choose one.

We are also able to deal with `Date` inputs. While `Puppeteer` does not provide a method to interact with the browser's date picker, we can set the `value` attribute of the input directly since it follows a standard format ('YYYY-MM-DD'). However, interacting with it in this way means any code on a target page listening for form interaction will not be triggered.

3.4.4 Submitting Forms

Occasionally phishing sites have multiple forms on a single page. The most common cases are either having a separate "search" bar, or a fake "sign-up" form. Since we cannot discount all single input forms (due to multi-part logins), our current solution is to try and match on all inputs on a page. Originally we just emulated a return key press to force a form submission, but that submits the form that the last filled in input belong to. In a hypothetical page containing a login form and a search bar, if we happened to fill out the search bar last, we would end up submitting that form instead of the login form, potentially throwing our crawler off course.

Since most phishing sites that emulate enough of a target site to have multiple forms are unlikely to omit actual submission buttons on a page (rather than a simple phish with just inputs and a background image faking the rest), we try to identify the most important form on the page by assuming that the form containing the most inputs is the most important one on the page. However, we can't rely on the phisher structuring their inputs into segregated forms. Custom JavaScript on the page may cause differing behavior depending on which submit button was pressed, for example. To have a generalised method that can be applied to many different page structures, we compute the mean of the (X, Y) coordinates of all input fields on the page (omitting any buttons). We then try to submit using the closest submit button to this mean coordinate. In case that submit button doesn't do anything useful, we also emulate a return key press (since sometimes a page's structure can cause a click to fail). This has proven to be effective against most phishing sites encountered.

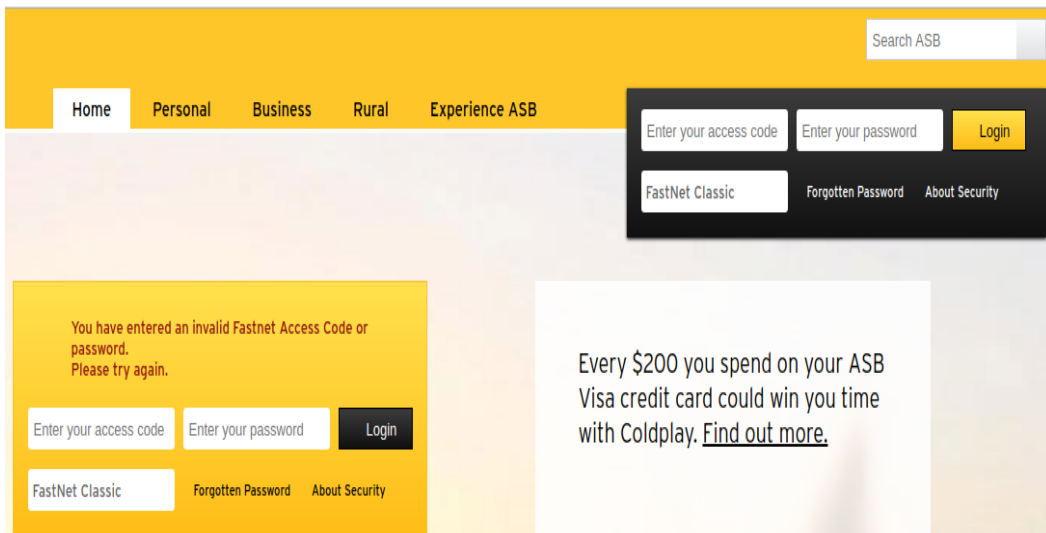


Figure 11: Example of ASB phishing site, with a search box and two login boxes

3.4.5 Detecting effects of form submission

After filling in a form on a suspected phishing site, we assume there are four possible outcomes. The first is that the page accepts the inputs and redirects to a new domain. This is the easiest to detect since polling the page for its current URL will reveal a change. We then use the parse-domain library[47] to extract the domain from the URL and check it against a list of known domains to see if it is owned by a known target. We don't currently lookup redirects to IP addresses. While we could compile a list of known IP ranges by target, this is likely to be unreliable due to many companies not hosting their own infrastructure and relying on cloud providers and we haven't yet seen an example of a phishing site that redirects to an IP address of a target brand. It wouldn't make a huge amount of sense for them to do this since an IP address in the URL bar looks suspicious, and the main reason that phishers redirect to target sites is to reduce the likelihood of a victim realising they have had their credentials stolen. Sometimes a phishing site will display a loading spiral or display an error message before redirecting to "justify" a reason why a user is being sent to the home page. We add a timeout of 10 seconds after submitting a form to be able to detect this.

The second outcome is that a page accepts the input, and navigates to a new page on the same domain. We expect legitimate sites carrying surveys or sign-ups to exhibit this behaviour after requesting data from a user. They are much less likely to redirect away immediately, unlike a phishing site, which will want its victims to spend as little time as possible on their page after handing over their credentials to reduce the chance of detection. Of course, at the same time, phishers have an incentive to extract as much information as they can get away with before detection, so many include multiple pages of forms to request additional data. Making a general assumption that any textual

inputs detected on subsequent pages are part of the same form, we traverse and fill in these inputs until we reach a page with no more, or have redirected to a new domain.

The third outcome is that a page doesn't navigate away after entering inputs and submitting, it either refreshes the page or it presents an identical page at a different URL. We assume a legitimate site would do this when supplied with incorrect credentials, as would a phishing site when form inputs supplied don't match client or server-side validation. There are also phishing pages which try to evade detection by faking validation or authentication errors on any input but still send any credentials entered to the phisher. We need to be able to detect seeing the same page to be able to try different credentials, or to reliably timeout in the latter case. After waiting for a fixed period of time after entering credentials (to see if the page navigates elsewhere or to deal with JavaScript waits), we extract the list of visible inputs, and hash their attributes. We then compare this hash to the hash of the page where we entered details. If they match, we assume the forms are asking for identical inputs. This logic could fail in the case where a new input is added to the form (but it is still asking for the same credentials, e.g. a Completely Automated Public Turing test to tell Computers and Humans Apart (CAPTCHA) test included in the same form after entering incorrect credentials), but we suspect that it won't impact on successfully completing phishing forms since very few phish currently feature CAPCHAs.

The final outcome is where a page fails and causes the browser to crash, or navigates to a non-existent page causing the browser to display its own error page. This can be caused by the phishing site being temporarily overloaded with traffic, or due to misconfiguration by the phisher. Currently, we treat a navigation to the browser's error page as a redirect to a new domain and stop crawling any further, while we treat browser crash as an absolute failure.

3.4.6 File uploads

A number of sites include file upload inputs, usually to gain photos of government identification such as passports or driving licenses. There are two mechanisms through which a user can upload a file using a browser. One is the traditional file picker input, created using a HTML Input tag with `type='file'`, and the other is through "drag-and-drop", with the browser triggering a "dropped" event when a file is dropped on to a DIV on the page. Puppeteer provides an `uploadFile()` method on Element Handles of file inputs, so we get the Element Handle for the selector, and choose a random image from a bank of royalty-free photos to upload.

We don't currently handle "drag-and-drop" uploads, since we haven't seen an example of a phishing site using one. Also, due to limited browser support for drag-and-drop, most sites tend to still have a traditional file input as a backup.

Initially we had problems solving file uploads since often the file input has a "hidden" attribute set, meaning it is not displayed on the page, and we drop such not-visible inputs from consideration when filling forms. The reason for this is that legitimate site designers often consider the default browser upload button to look archaic and ugly, so they hide it and display a more stylish custom button (an example can be seen in figure 12). When a user clicks on this button, JavaScript calls the `click()` method on the hidden file input element to trigger the file picker. Normally this button will contain an attribute with the word "file" or "browse", so if we encounter such a button, we search the page for a hidden file input, and if one exists, trigger it in the same way as if it were visible.

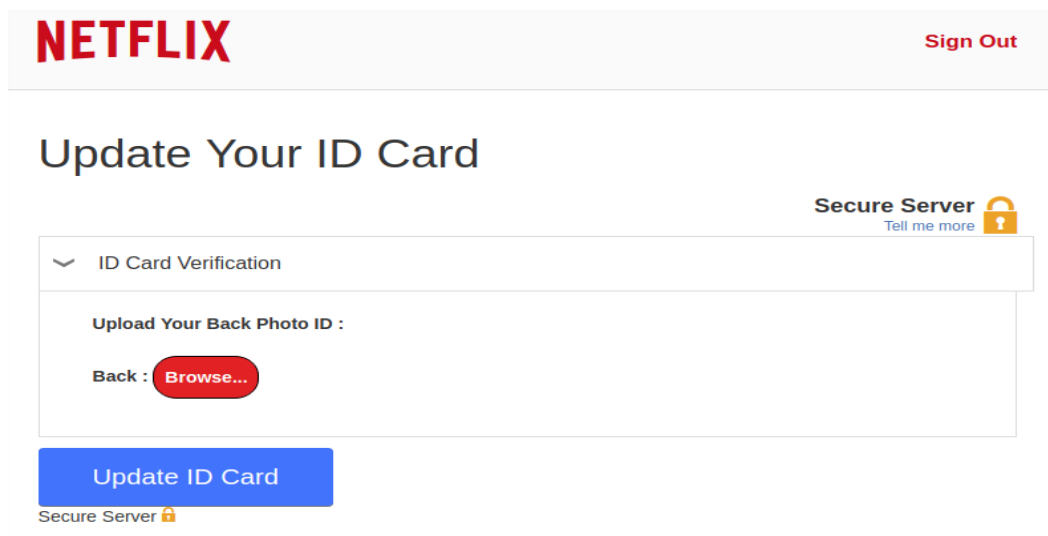


Figure 12: Example of a styled upload button that triggers a hidden file input

3.4.7 JavaScript Alerts

JavaScript alerts are a built in feature of most browsers. They allow a website designer to trigger a pop-up with a single line of code. However, they cannot be styled, and so are often associated with pop-up ads and scams (see figure 13) . This is due to the fact that they pause JavaScript execution , meaning a user cannot interact with the browser until an alert has been dismissed. As a result most legitimate sites tend to use modal pop-ups instead. Phishing sites however do sometimes still use them to display various messages, and we need to dismiss them to continue interacting with a page.

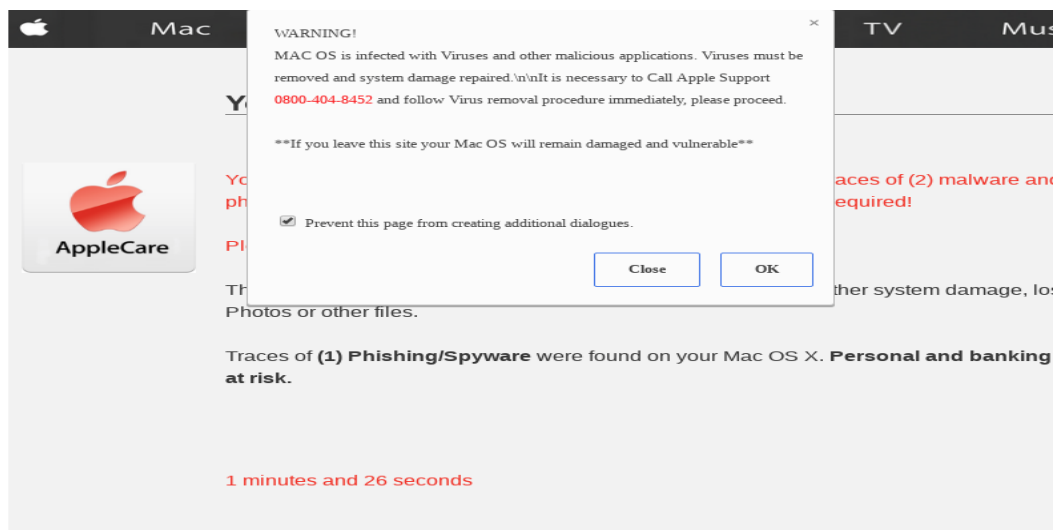


Figure 13: Example of a pop-up alert

We can solve this by attaching an event listener to every page we load for a 'dialog' event. When triggered, we call the dismiss() method of the alert box, instantly closing it.

3.4.8 DIVs as buttons

Sometimes a phisher will use DIV elements to form the buttons they want a victim to click on. Usually, an onclick method will be added which calls the appropriate logic when actually clicked. However, since a DIV element will not modify a victims mouse pointer while being rolled over, phishers usually enclose the DIV inside of a A tag, or place numerous hyper-links behind the DIV. Our current approach of clicking on all hyperlinks on a page usually works to solve these, but sometimes due to the way a page has rendered, the central point of the hyperlink tag does not intersect with the fake button. This means that using Puppeteer's click method will not trigger it. To catch this scenario, we add all DIVs with the onclick attribute set to our list of links to try.

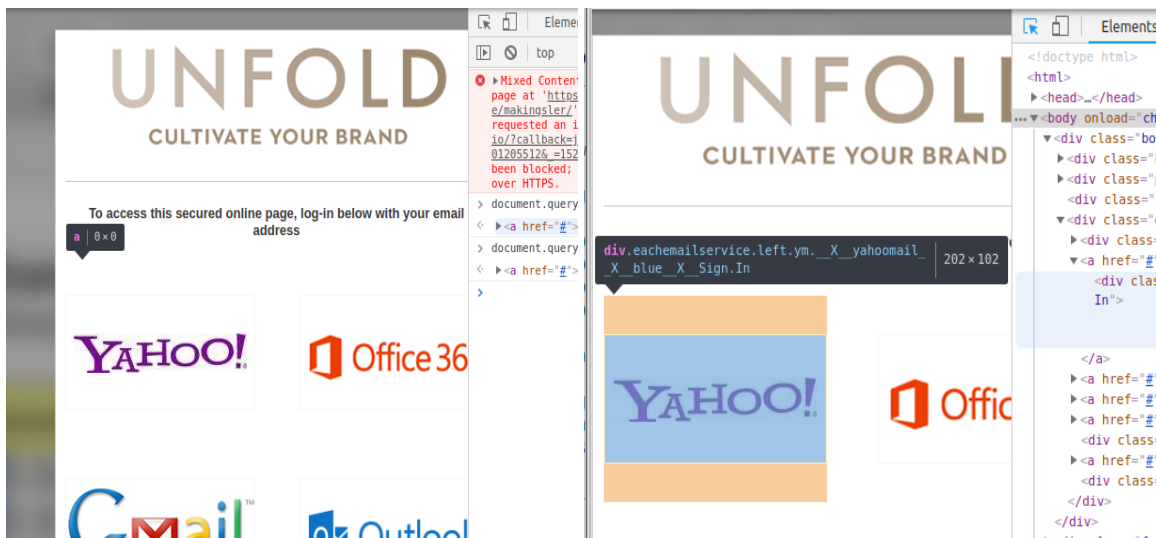


Figure 14: Example of a DIV within an A tag. Notice that selecting the A tag does not highlight the DIV's contents

3.5 Defeating validation

Earlier we simply submitted credentials randomly from a list of common types of details, including emails, usernames, account numbers etc. However, we often encountered sites with multiple pages of forms, each requiring different types of values, with most pages implementing some form of input validation, our small list of hand-generated credentials often prevented us from going further than the first page.

3.5.1 Random data generation

We first tried to solve the problem by creating a wider pool of values to use with our trial and error method. Using the faker.js [48] library, we can produce a full random profile for a user, with many details a phishing site is likely to ask for eg., address, date of birth, account numbers, etc. However, since we have to allow time for a page to respond after submitting (10-20 seconds), completing forms in this manner can take a while. We tried to better match input types with a type of credential, for example only matching usernames and emails for inputs of type "email", or only matching numerical values for inputs of type "tel", but this actually lowered our success rate somewhat, since sometimes phish will use "email" or "tel" type inputs for every kind of input.

3.5.2 Generic Matchers

We decided to try and use clues left by the phisher to determine what sort of data a form input is expecting. Trying to match labels from text on a page to inputs can be unreliable since they may be contained in separate DIVs or even be baked into images, meaning we would need optical character recognition (OCR) to read them, and since phishing is a global issue, may be written in many different languages. However, in the same way a global brand may rely on the same source code to serve sites in different regions but still display the local language to users, phishing kits may release several versions for different languages, including languages that don't use the Roman alphabet. In order to identify inputs when they are submitted, or to attach client-side validation, phishers will often place relevant identifiers in an input's attributes. These are often common across many phishing sites, especially for inputs with commonly used abbreviations. For example, an input requesting the Card Verification Value (the number on the back) of a credit card often has an ID or placeholder value including the abbreviation "CVV".

We combine a number of an input's attributes (id, type, name, value, class, placeholder) into a single string which we call an ID string. We then run the value through a number of "matchers", functions which target a specific type of input, such as credit card numbers or dates of birth. They rely on matching against the ID string with a number of regular expressions based on previously seen phish. If they match, they can return one or more potential values to try to solve the form. If several matchers return for a given ID string, we randomly choose values from the returned list until either the page accepts the values or until we hit a timeout.

3.5.3 Target-specific matchers

More advanced phishers may implement more advanced target-specific input validation. These cases are hard to catch using our general matchers, and may require the addition of target specific matchers. In the example below, the designer of this phishing site targeting the Monero cryptocurrency uses JavaScript to extract the word count of the input, and only validates if it equals 13. Adding a custom matcher involves adding a regular expression to match on any attribute labelled "login-key".



Figure 15: Example of a custom matcher solving a monero phish

While the matcher above will solve that particular phishing site, it may fail validation on another site targeting Monero. This can occur because a phisher automatically invalidates well known fake values (e.g. well published dummy credit card numbers), or due to additional target knowledge such as understanding the location of check digits within an account number. We encountered one example

of a site imitating a Spanish Santander login page. It asked victims to enter a "CPF" number, which is equivalent to a customer's online banking number. After failing to solve manually, we extracted the HTML from the page and were able to download the source files for the validation code. The phisher had implemented a checksum⁶ function for Santander customer IDs. A quick search on the internet revealed no obvious clues indicating the source of the logic of the method. Despite not knowing how valid this check is against real account numbers, we can assume that randomly generated numbers that pass this phisher's test are more likely to be validated than any random number of the same length. We therefore incorporated a cleaned-up version of the phisher's code into a matcher. We are able to use phishers' target knowledge against them to better match and take down their sites!

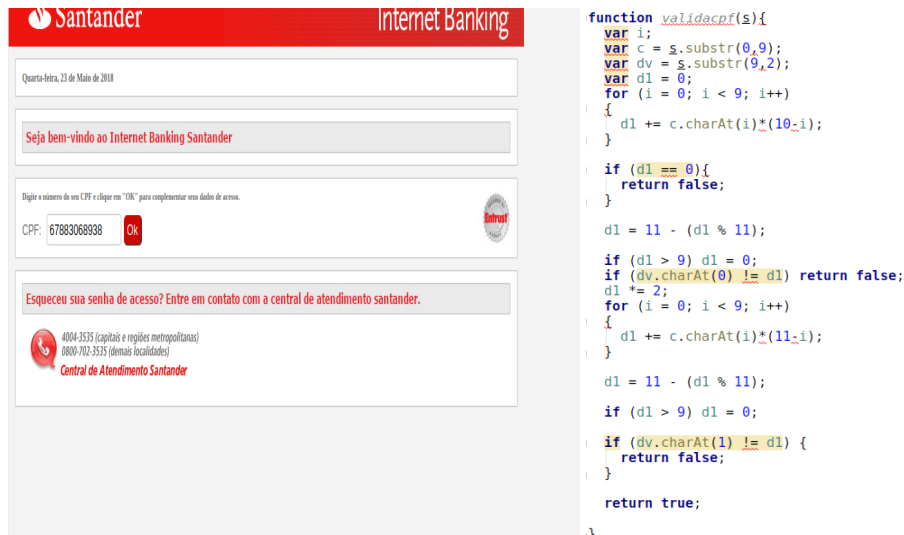


Figure 16: Example of Santander phishing site, and extracted phisher's code used to solve it

3.5.4 Speeding up matching

Using matchers can provide better "guesses" at what a form is requesting, but we can still end up with a large list of values to try. One way we can try to reduce this list is to try and use the client-side validation a phisher has placed into a page to validate our values without submitting the form.

The easiest check to perform is if the phisher has set the "pattern" attribute on an input. This allows them to set a regular expression for values it will accept which is enforced by the browser. We can extract this attribute and test our matches against it, dropping any which do not pass. However, we found only an extremely small number of inputs seen actually used this feature. Out of a sample of 3499 confirmed phishing URLs, only 4 contained inputs with the pattern attribute set. The maxLength attribute, defining the maximum number of characters that can be entered into a text input, was much more common, being found on 1774 of the visited sites. We use this in the same way, filtering out any of our matches which would be too long for a particular input.

Another common form of validation involves a phisher attaching a listener for "keydown" and "keypress" events when interacting with a form. The browser triggers these every time a user enters a character, allowing reactive displays of validation (e.g. colouring an input box red until what is inside passes validation). It can also prevent certain characters being entered, such as preventing non-numeric values in a field asking for a telephone number, or used to limit the maximum length similar to setting the maxLength attribute. We can compare the value of an input after simulating typing with the original string entered. If there are any differences, we know that some client-side code has modified it, with the likelihood being that it was an incorrect match. Since we don't have to submit the form to perform this check, avoiding potentially long waits for a server to respond, we

⁶A simple redundancy check for errors in data

can test hundreds of matches in a second or two.

Of course we still may have a number of matches that have passed all these tests. At this point we will try and submit them and see if the page accepts them. This process is slow since the page timeouts previously described mean that we take at least 20 seconds per match. However, we noticed that some phish return validation warnings using JavaScript alerts. Since alerts freeze all JavaScript activity, it would be strange for phishers to use them for any other purpose right after submitting a form since it could delay them receiving the values that a victim entered (since the request could be delayed until the user dismisses the dialog), so we use the detection of an alert to cancel any timeout period after submitting and automatically assume validation failure. This means instead of waiting 20 seconds for one match, we can check 20 matches in less than 5.

Another potential sign of validation failure is the lack of any outgoing connections after submitting the form. While legitimate sites implementing multiple pages of forms on a single paged application are likely to store answers locally in the browser and only submit them when the form is fully completed to reduce the number of requests to their servers, phishers will often submit data separately immediately after submission of each page.

3.5.5 Handling non-existent or suspended pages

Since phishing sites often have extremely short lifespans, when checking URLs we often come across URLs which may have hosted a suspicious page in the past, but have now been shutdown, either by the work of security companies, hosting companies, site owners discovering the phish or phishers taking the site down themselves.

While the majority of sites will serve the correct HTTP response containing a 404 status code (meaning Not Found), allowing us to detect a missing page and stop crawling further, other sites may serve a page indicating a 404 error, but return a status code of 200 (meaning OK). This is known as a 'soft 404' [49]. They are a well known problem for search engines, leading to bad entries in their indexes and wasting valuable crawling resources when they could be indexing useful data. For our classifier, it could also mean more false identifications, since some web hosting companies redirect not found or suspended URLs to their own home page, meaning we may inadvertently be classifying a legitimate site (we have on a few occasions found a login button on a account suspended page and spent a while trying to solve its login form).

We found that many soft 404 pages tend to have common keywords in the page title - phrases such as "account suspended", "contact hosting" or indeed "404 Not Found", allowing us to implement a simple filter using expression matching. This works fairly effectively since most international web hosts seem to use English error messages. One noticeable exception to this is the St Petersburg based Beget. It seems to host a disproportionately high number of the phishing URLs we have received from Netcraft (of around 12000 URLs received, Beget hosted 400, just over 3 % !). After adding the Russian translation of "The site is blocked by the hosting provider" to our title matcher, we no longer saw its distinctive "lost octopus" branding in our results (see figure 17).

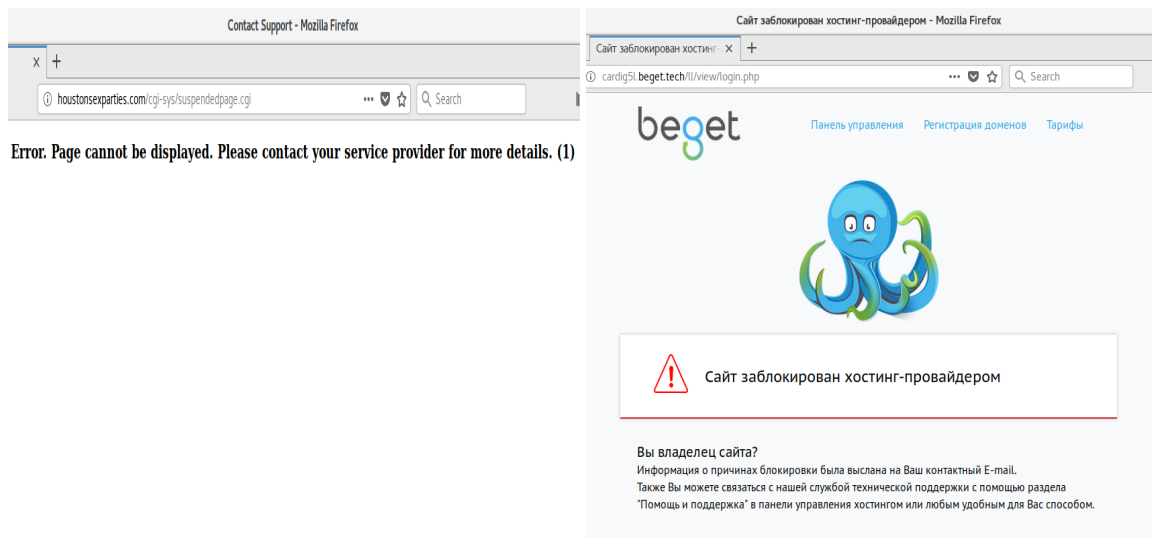


Figure 17: Examples of suspended page notices by hosting providers

There have been reports[50] that some phishing sites report a 404 status code intentionally, but still serve the phishing site content, exploiting the fact that browsers will render a page regardless of the status code sent with it. This is done intentionally to make automated crawlers think the site is offline, and not investigate further. We haven't seen an example of this so far ourselves, and for now we accept the server response's status code.

3.6 Target Identification

3.6.1 Overview

Our primary method of target identification is based on the assumption that phishing sites usually redirect to a legitimate page of the brand they are imitating. By providing a seamless transition between the phishing site and the legitimate content that a user was expecting to see, the chance that the user notices any irregularities is minimised. If the user spots their mistake and changes their passwords and/or replaces their credit card, the credentials become much less valuable to the phisher. A legitimate site is much less likely to redirect immediately to a new domain owned by another entity after asking for details, especially if filled with fake randomised data.

3.6.2 Domain to target mapping

We use a large lookup file containing a mapping between domain names and known brands that either own or are associated with that web address. It is collated together from several lists used internally at Netcraft for target identification. It links over 85000 known domains with just over 10000 brands from companies based all over the world. While we could have obtained most of this information by querying public registration databases (and there are online services which will try to discover all domains registered by a single entity, often for a small fee), having access to this data greatly reduced the amount of work needed to get started on this project.

Since our lookup file only stores domains, we need to reliably extract the domain part of a given URL before searching for a match. This is hard to solve with simple pattern matching due to the fact that domains are handled differently across different regions and organisations. For example, extracting the target domain for the fully qualified domain name (FQDN) "www.amazon.com" could be solved with a pattern matching the last two sections (i.e. anything after the second-to-last dot - returning "amazon.com"). However that approach would not work on "www.amazon.co.uk", as it would return the wrong domain ("co.uk" instead of the intended "amazon.co.uk"). ".co.uk" is

an example of a so-called "public suffix", where any internet user could register a domain name ending with it. Since a given domain owner may also maintain levels of sub-domains (e.g. "example.ahigherlevelexample.amazon.co.uk"), matching for the public suffix is crucial in detecting the intended target. Mozilla, the company behind the Firefox browser, maintains a public list of them called the Public Suffix Index[51]. Parse-domain [47], the library we use to extract the domains from URLs, queries a local copy of this list during extraction.

The list also includes common domains for shared and cloud hosting providers, since these providers usually allow their customers to register their own sub-domains on a domain provided by the cloud host to access managed services. This allows customers to take advantage of the companies SSL certificates and avoid having to set up their own. But it also benefits the phishers since they are able to host content on a legitimate looking URL. To avoid possible misidentification, we filtered our lookup file to remove the domains of the hosts found in this list.

3.6.3 Identifying pages based on network requests

For every page that we visit, we are able to monitor all network requests that the page makes. Puppeteer allows us to intercept a request and access or modify the response that the page's JavaScript ultimately receives. We extract the URLs of these requests and add them to a log, which we can later test against our domain list. Phishing sites may include content such as images from a target's site to avoid hosting it themselves. A potential downside to this approach is that it can be unreliable when based on single requests. For example, Google hosts a copy of the popular JavaScript library JQuery on a Google owned domain. Many sites including phishing sites include this copy instead of hosting their own. This alone is clearly not a good indicator that a site is targeting Google. Multiple requests to a targets domain over multiple forms however are a much more reliable indicator.

3.6.4 Identifying targets based on links in the page

All previous attempts to implement automated interaction with phishing sites that we have seen have avoided tackling validation, and often only visit the first page after submitting a form. Since many phishing pages consist of multiple pages, they have based their target identification on attributes extracted from that page, such as domain names in hyperlinks. On every page that we encounter while visiting a suspected site, we already extract the attributes of every input, button, hyperlink, form label, image, and DIV section for the purposes of filling in forms successfully, and we save this information to a log file. By parsing it for URLs and checking them against our list, we can also evaluate this approach and compare its reliability to page redirection.

3.6.5 Identifying previously classified pages

Since phishers often reuse the same phishing kits on multiple URLs, it is possible that our classifier has already encountered the exact same page before. In the cases where we have failed to identify a target, and passed it on to manual review, we want to be able to use the result of that classification to prevent the same site being sent to manual review again. To enable this, we generate a unique hash for every page we encounter.

The steps to produce a hash are as follows :

1. **Generate a HTML string describing the entire page using `document.body.outerHTML`.**
This causes the browser to serialise all the objects that are currently present in the DOM.
2. **Remove any empty tags using a regular expression.**
Empty tags do not affect the layout of a page.

3. Remove all attributes from each tag.

Attributes can be changed by JavaScript on a page, and identifiers may be dynamically generated in an attempt to avoid detection. Since we are already using a web browser, we can use it as a HTML parser by creating a new element in the DOM of the page we are currently on, and settings its innerHTML property to the HTML string from the previous step. Since this new object is not referenced to by any element on the page, it does not affect the current page at all. Additionally, inbuilt security measures in the browser prevent any script tags being added using the innerHTML setter from running. We can therefore loop through all the new elements produced and use the browsers removeAttribute method on each element.

4. Flatten script tags.

Since JavaScript may be obfuscated to avoid detection, meaning it appears unique every time the same page is loaded, we remove it from the HTML. We reason that this is safe to do since any JavaScript that affects the initial content of the page will have already run and any elements it created will already have been serialised into the HTML.

5. Generate HTML from object created in step 3 and generate MD5 hash.

We based this method on existing methods used at Netcraft for fingerprinting phishing kits by statically analysing the HTML served by a suspicious URL. Phishers have tried to avoid this sort of detection by serving HTML files containing nothing but a large obfuscated SCRIPT tag containing JavaScript that generates all the visual elements of a page. Since we actually render the page and execute this JavaScript, this does not prevent us from reliably fingerprinting a site.

3.7 Observations

The results seen earlier in table 10 show that redirecting post form submission can be an effective indicator of phishing, and a seemingly reliable way to determine the target brand. In order to confirm this for a number of targets we need to test against many more URLs. However, since URLs taken from a "takedown" system are actively in the process of being shut down and removed from the web, the quicker we can process them, the better chance of useful results. Here are some of our observations about the crawling process.

3.7.1 Time per URL

In the introduction we decided that we wanted to beat the classification rate of a human (around one every 10 seconds). However, that rate is based on humans visually inspecting screenshots and metadata. This data has already been extracted pre-classification, since if humans were directly interacting with suspicious URLs, they would probably be a lot slower. For example, it could take more than 10 seconds for the initial link to load ! In a similar way, once we have extracted enough features from a site, we can apply a number of heuristics to this data to quickly produce a classification result. But gathering these features can involve up to several minutes of interacting with a page, especially if it contains multiple pages of forms. In order to process thousands of URLs within an hour, we need to be interacting with many sites concurrently.

3.7.2 Browser Resources

While running a browser in headless mode does result in significant savings in resources when run on a server (since we don't have the overheads of a graphical environment), the actual memory saved on the application side is only around 30 % less (see figure 18), which is lower than we were expecting.

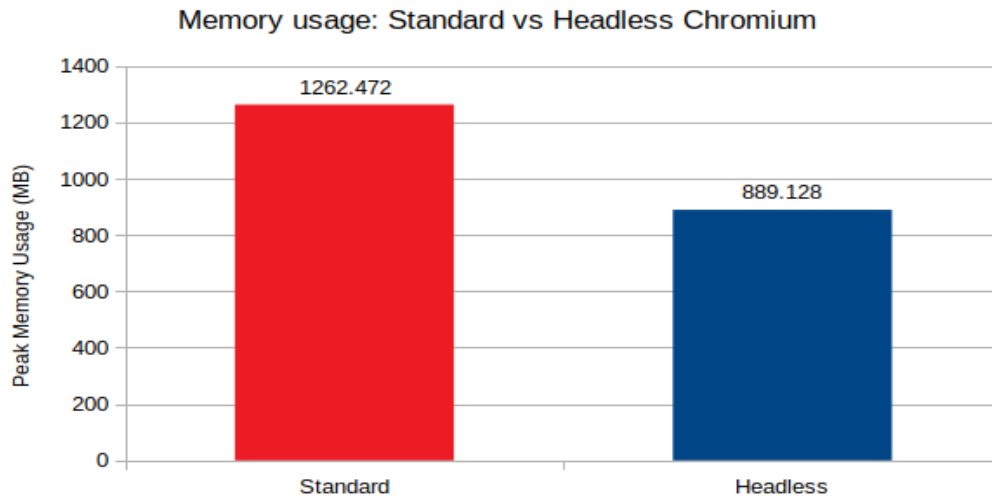


Figure 18: Comparison of maximum memory usage of our crawler code using graphical and headless Chromium based on crawling a phish containing three input forms

We did find that the memory use did fluctuate depending on the complexity of the page opened. In the top graph of figure 19, we can see how that memory usage by Chromium varied by up to 250 MB during crawling, with the initial page load of a simple login form requiring noticeably less memory than the full Amazon home page (visited at about 110 seconds). CPU usage, on the other hand stayed surprisingly stable despite the fact that there were a number of periods where our crawler was idle and waiting for timeout periods to pass before proceeding. (The initial high load spike is down to processing our configuration files and domain lists. This only has to happen once when the application starts up, so we can ignore this).



Figure 19: Memory and CPU% statistics while crawling an Amazon phishing page containing three input forms

Based on figure 19, if we assume that we use 700MB of memory and around 45 % of CPU utilisation (on a single core @ 2.3 GHz) on average per extraction task, we should be able to comfortably run around 10 active instances concurrently on a typical quad-core machine with 8 GB of RAM.

3.7.3 Concurrency with Puppeteer

We tried to see if we could parallelise page crawling by interacting with multiple suspect sites with a single instance of Puppeteer and Chromium, with each site starting in a new tab, but we found that this makes detecting pop-up windows harder, since we frequently check the list of open pages to get the top window. When visiting multiple URLs at once, it becomes difficult to determine which site spawned which additional page. Since we would be using a single browser, we cannot guarantee a clean session for each page visit. Any cookies set by one task on a particular domain may affect page behaviour when visited again by another task. Puppeteer and Chromium developers have recently tried to mitigate this by introducing the concept of Browser Contexts [52], essentially creating multiple isolated "incognito mode" sessions, but as of June 2018 this isolation currently leaks a browser's local storage between contexts [53]. Even if isolated in separate contexts, if a page causes the browser to crash, it will affect every crawling task that is being run in that browser. We found the safest way of running multiple instances concurrently was to spawn separate Node processes controlling a single

browser each.

3.7.4 Geoblocking

The continuing battle between phishers and those working to thwart their efforts has meant that criminals will often blacklist entire IP ranges to try and avoid detection. Analysis of some of the phishing kits installed earlier revealed a number of common blocks, including Amazon's IP ranges as well as those of many security companies. Additionally, if a phishing site is targeting a brand from a particular country, they may only whitelist IP ranges from that country, serving false 404 or 500 error codes to requests from elsewhere. Finally, some corporate networks may block suspicious domains and/or IP ranges (as is the case at Imperial College). During our testing, we had to proxy all our traffic through a commercial VPN provider in the UK, since all the URLs we receive are already included in block lists which Imperial use to safeguard their network. We also noticed that many sites (especially targeting languages other than English) that refused to load or served 404 pages, actually served content when switching to a South American or Asian VPN.

4 Large scale crawling and classification

4.1 Overview

In order to be able to study a large number of different phishing sites, we needed a distributed platform in which to run large number of crawlers concurrently. We designed the system to be ready to integrate into a larger classification pipeline, accepting URLs and returning a classification result after crawling.

We split the system into three components: Task Manager, Crawler and Classifier. The Task Manager component handles the submission and scheduling of tasks. The Crawler module receives tasks from the Task Manager, and runs our crawling code from section 3. When it completes, the Classifier module analyses the returned logs and produces a classification decision.

The separation of crawling and classification allows easy modification of classification rules, without shutting down the crawler nodes and allows us to re-run classification on previously collected extraction data to compare classification outcomes.

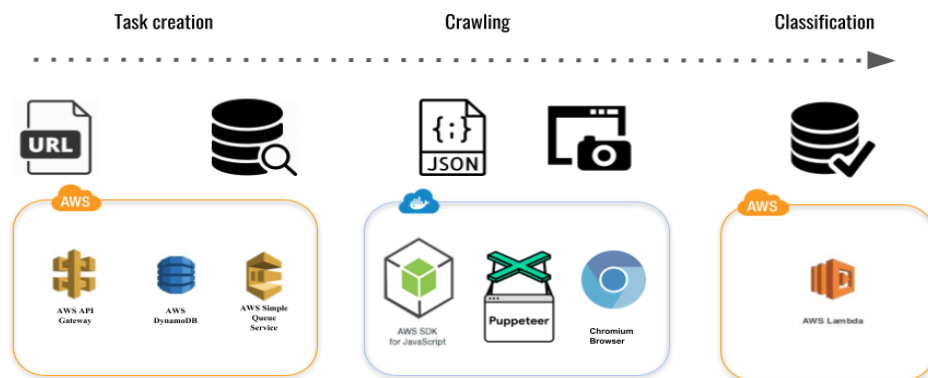


Figure 20: Outline of pipeline

4.2 Architecture

The system follows a distributed micro-services architecture that ensures future scalability while simultaneously keeping deployment costs low. We run all our task creation and classification code on managed services to achieve easy transparent scaling, while running our crawler workers on a number of Infrastructure as a Service (IaaS) providers to take advantage of lower cost compute power. The code powering all stages of the pipeline is written in JavaScript, reducing the project's complexity and allowing us to use the same coding style across the code base.

4.2.1 Amazon Web Services

We chose to use Amazon Web Services (AWS)[54] since it provides a number of managed services that link together easily as well as a generous free tier which covered almost all the resources our system needed to handle loads the size of Netcraft's manual classification feed.

4.2.2 AWS Lambda

The end-points for the Application Programming Interface [API] are powered using AWS Lambda[55], an event-driven, server-less computing platform that can automatically scale to handle thousands of concurrent requests. AWS API Gateway provides an external HTTPS endpoint with SSL preconfigured, and passes all requests to a Lambda function handling task creation, which uses AWS' NodeJS library to interact with the database (AWS DynamoDB) and the queueing service (AWS Simple Queue Service).

We also use Lambda to produce the final classification decision. This is based on data generated by the crawlers that is uploaded to AWS S3 [56]. The classifier needs to react to a completed extraction job, download the log files from S3, produce a result and post this back to the database. A benefit of Lambda is that functions can be directly "triggered" by database events, meaning no additional API calls are needed when an extraction job completes.

4.2.3 AWS DynamoDB

AWS DynamoDB [57] is a fully managed NoSQL database service from Amazon, with billing based on throughput rather than storage. It integrates well with other AWS services such as Lambda, and its consistent key-value lookup times at any scale suit our system where an external process (a security company's classification system) is likely to poll for specific request IDs. Its almost schema-less design (requiring only the primary key to be defined) allows for quick prototyping and addition of extra fields when needed.

We primarily use it to store meta-data about a submitted job, but it also acts as a distributed locking service. When a crawler node receives a job, first it will request all the meta-data for that job from DynamoDB, including the current value of an atomic job counter. It will only attempt to "lock" that job if it's still pending, or if it contains a process timestamp older than 5 minutes (this could occur if a previous extraction job failed). To acquire a lock, it will post a current process timestamp back as well as incrementing the job counter, but with a condition attached on the write that the job counter before incrementing must be equal to the value it received earlier. If the write fails (as seen in figure 21), it means another extractor node has already claimed the lock by incrementing the job counter, and the slower node polls SQS for another job.

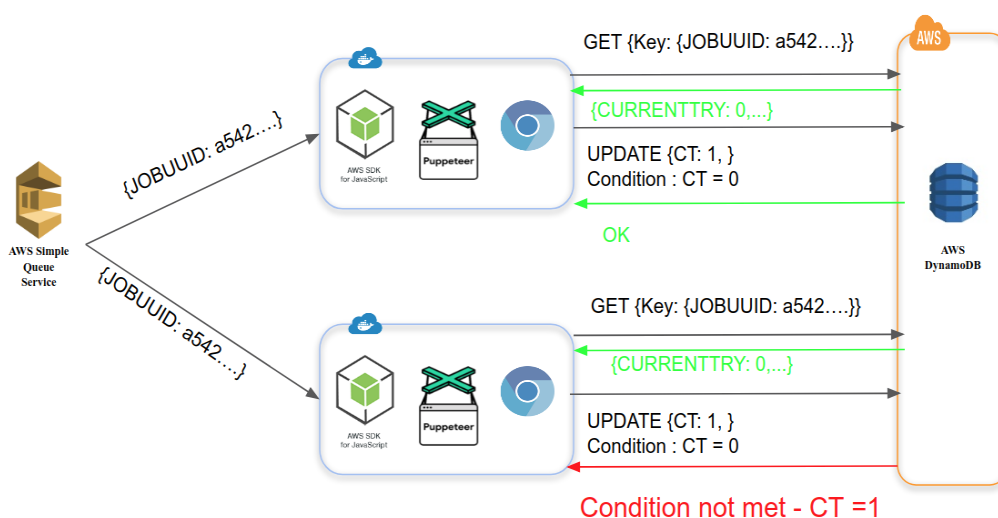


Figure 21: Example of counter preventing race conditions

One pitfall in using DynamoDB is that while fetching a record by primary key (a get operation) is

efficient, retrieving an entire table of records (a scan operation) is slow and expensive. This operation is important during testing since it allows us to produce statistics based on all submitted jobs at once. Due to the distributed nature of the database architecture, where multiple nodes have to work together to produce a unified view of the table, Amazon have hard-coded a restriction of 1MB per request to prevent saturating each individual node. This means obtaining an entire table means having to make multiple requests. Originally we were using DynamoDB to store the feature logs of an extracted page, but since these could be a few MB in size, our provisioned database instances were being overwhelmed with very low traffic. Instead of spending significantly more money on increasing the number of DynamoDB instances, we found it was much more efficient to store these large files on AWS S3, and store the URL pointers in the database. With records now sized in the order of a few KB, scanning an entire table became much easier and scalable.

4.2.4 AWS Simple Queue Service

AWS Simple Queue Service (SQS)[58] is a managed queueing service. It provides a simple API for sending messages between distributed applications. The sending application pushes a message to a particular queue URL, while receiving clients poll the same URL to receive messages if there are any. We use this to send jobs to feature extraction nodes, which are currently hosted in multiple locations on different hosting platforms. SQS allows them to receive jobs without any extra configuration as long as they can access AWS' servers.

SQS offers two types of queue, one offering high-throughput with at least once delivery, and a low-throughput, exactly once First-In-First-Out (FIFO) queue. We chose to use the cheaper high-throughput variety, since our code is already idempotent by using DynamoDB to acquire locks on a job.

Messages sent using SQS are not automatically deleted after delivery. Each message has a adjustable "visibility timeout" set, which starts after delivery. During this time if a receiver decides to delete a message, it will be removed from the queue, otherwise if the time expires it will become available for redelivery. If a job completes successfully, we delete the job message from SQS. Should extraction fail (for example it received a 404 error) or if the extractor node fails (e.g. we shut it down mid job), the job will become available again timeout of a few minutes another extractor node to handle. After a certain number of failures, messages can also be sent to a "dead letter queue". We use a lambda function to monitor this queue and update job records in DynamoDB as failed.

4.2.5 AWS Simple Storage Service

AWS Simple Storage Service (S3)[56] is a managed object storing service. It provides a simple RESTful API allowing highly available and low latency access to files

We use S3 to store data which does not need to be indexed since it is always obtained in full. This includes the log files generated by the feature extraction process, as well as all the screenshot images. While the screenshots are not used during automated classification, they are useful for passing on further to manual review without having to revisit the site again, and for debugging purposes.

4.2.6 Docker and Docker Swarm

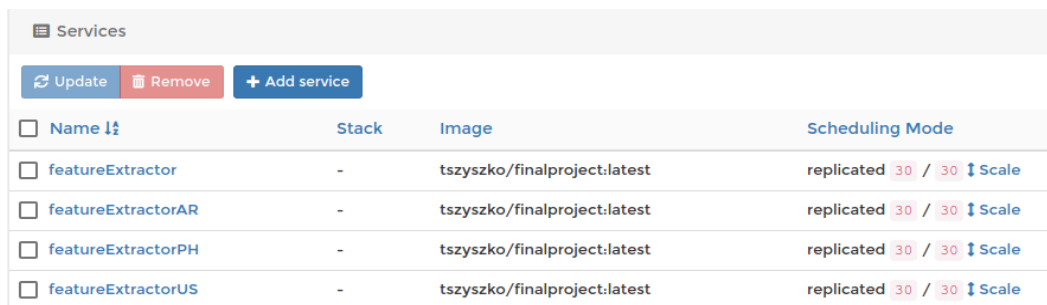
The core of our feature extraction process relies on being able to interact with a full version of the Chromium browser.

Originally, we planned to compile our own version of Chromium to run on AWS Lambda. The binaries supplied with Puppeteer would not run in the Lambda environment. Using Lambda we could easily spawn multiple crawlers (up to 1000 concurrently). While this solution would scale easily, we

would have to manually maintain the correct version of Chromium for each puppeteer update (instead of letting Puppeteer automatically download the correct binary). The hard limit of 5 minutes per Lambda also could cause issues, since when tackling complicated sites or solving difficult crawling, the crawling process may often exceed this limit, and while we currently kill jobs running longer than this, it could have restricted us in the future. Additionally, since we are frequently waiting for a target site to respond or for something to change on a page, Lambda can be an expensive option since it charges by the second of execution time.

Instead, we decided to exploit cheaper computing resources outside of AWS. This includes Cloudstack VMs provided by the Computing Department at Imperial, VM droplets at DigitalOcean[59], as well as on personal computers. This is made possible by producing a self contained docker[60] image bundled with Chromium’s huge list of dependencies (mainly fonts and graphics libraries) and a pre-configured NodeJS library to handle receiving jobs from SQS. In theory, any machine with an internet connection and docker installed can run a feature extraction node by running two commands on the command line (one to download the image and another to run it!).

Docker is a containerisation library that allows multiple ”containers” to run on a single Linux instance. It also isolates those instances from each other, and the operating system, meaning that if one browser instance crashes or is compromised, it does not take down the host it is running on. We can also limit the resources each container receives so one job does not prevent any others from completing. To equally distribute feature extraction containers between multiple underlying VMs, we used Docker Swarm[61]. This is a clustering solution provided by Docker that allows us to treat multiple hosts as a single virtual host. With the addition of a web interface provided by the Portainer plugin[62], this makes horizontal scaling as easy as adding more VMs and increasing the number of instances we want to use as seen in figure 22.



<input type="checkbox"/>	Name ↓↑	Stack	Image	Scheduling Mode
<input type="checkbox"/>	featureExtractor	-	tszyszko/finalproject:latest	replicated 30 / 30 ↓ Scale
<input type="checkbox"/>	featureExtractorAR	-	tszyszko/finalproject:latest	replicated 30 / 30 ↓ Scale
<input type="checkbox"/>	featureExtractorPH	-	tszyszko/finalproject:latest	replicated 30 / 30 ↓ Scale
<input type="checkbox"/>	featureExtractorUS	-	tszyszko/finalproject:latest	replicated 30 / 30 ↓ Scale

Figure 22: 120 Crawler instances organised by region in the Portainer web interface (we previously named this module featureExtractor)

Currently, the number of VMs is manually scaled. In the future, we could use a managed container service such as AWS Beanstalk to host the docker containers and auto scale the instances based on the amount of jobs in the queue, giving the potential for significant monetary savings.

4.2.7 Netcraft GeoProxy

To avoid the geoblocking issues discussed in section 3.7.4, we need to be able to try accessing a site from multiple geographical regions using fairly inconspicuous IP addresses. Netcraft kindly provided access to their internal proxy system, named Geoproxy. We are able to access proxies in large number of countries using the globally available proxy service. Since Chrome does not provide a method of programmatically setting proxies with authentication, we run a local node proxy server [63] that injects the credentials in while providing an open proxy on localhost for Chromium to use.

For our VMs supplied by Netcraft, we are able to access some of the global connections directly, avoiding the overhead of multiple proxy layers.

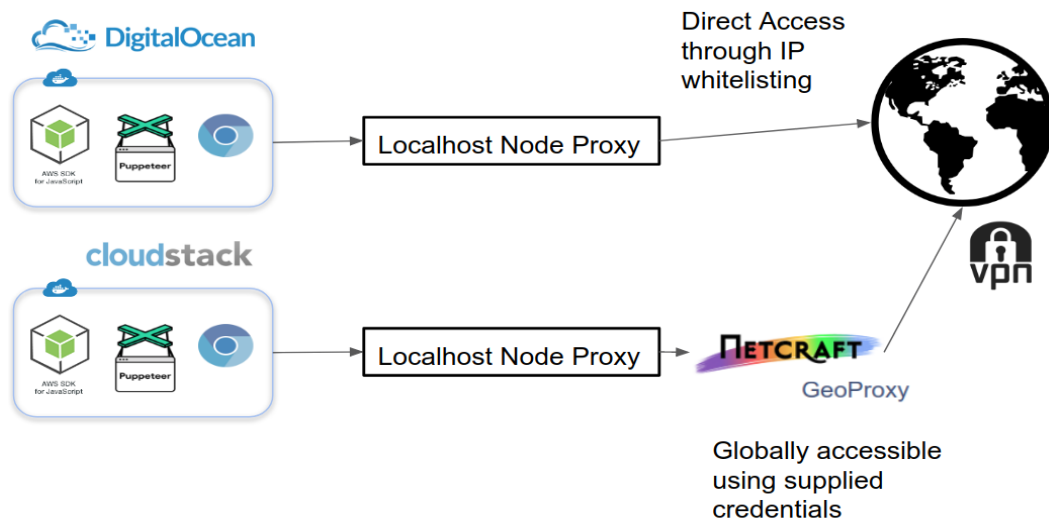


Figure 23: Proxy connections

4.3 Task Submission

The system can be accessed using a simple REST-ful API, allowing easy integration with existing classification systems. Submitting a job is as easy as sending a PUT request containing a URL to investigate along with any meta-data to identify the job as a JSON-encoded payload. When the system receives a job, it generates a job identifier, verifies it by querying the database for it, and then creates multiple records in the database using that id, one for each proxy region that we are currently testing in.

```
{
  "attackURL": "http://transcaspianforum.org/m/?platform=hootsuite"
  "meta" : {
    "target": "xfinity",
    "status": "blocked",
    "url_id": "616447904"
  }
}
```

Listing 1: An example submission from Netcraft's manual classification system. The meta-data is ignored by the classifier, but useful for generating statistics

Upon confirmation that the records were created successfully, it adds jobs to the respective queues of each proxy region, containing all the location of all the data a worker node needs to process a job. This includes the job identifier, the country-code of the proxy for that job, and the database table name where updates to that jobs status should be written.

Checking the status of a job involves sending a GET request to the same end-point supplying a job identifier in the query string. The system will return a JSON encoded response containing an array of the database record for each proxy region, allowing the calling application to decide whether to instantly block based on a single positive result, or wait for all jobs to complete to check for inconsistencies in target identification.

```

{
  {"Items":
  [
    {
      "ATTACKURL": "http://mail.supportinstagrams.com/",
      "ERROR": true,
      "JOBUUID": "38136ded-ac59-437d-9c88-00ee33f205b7",
      "META": "{...}",
      "CURRENTTRY": 5,
      "STATUS": "Error - Maximum Error Count",
      "SUBMITTIME": 1527520645186,
      "MAXRETRY": 5,
      "COUNTRYCODE": "ar",
      "PROCESSLOG": "[...]",
      "PROCESSTIME": 1527533567547
    },
    {
      "ATTACKURL": "http://mail.supportinstagrams.com/",
      "ERROR": false,
      ...
    },
    ...
  ],
  "Count": 6,
  "ScannedCount": 6
}

```

Listing 2: An example query for a job containing at least one failed task. The process log contains error codes explaining the failure.

4.4 Crawling

A crawler node will poll the queue for a pending job. When it receives one, it will pull the job record from the database. When a worker starts processing a job, it will write a timestamp to the database, acting as a lock. Should any additional workers receive the job within 5 minutes of this timestamp, upon checking the timestamp on the job, they will return the message back to the queue to be tried again later. Failed jobs are retried 5 times, with each attempt incrementing an atomic counter on the job record. Additionally, the queue system will attempt to redeliver a job if a job is completed by a node, it will mark its status as successful.

Crawler worker nodes consist of swarms of self-contained docker containers with the crawling code developed in section 3 bundled with a current version of Chromium and NodeJS. They receive their queue and proxy urls through environment variables, allowing easy deployment of identically configured clusters.

4.5 Classification

When a feature extraction job completes, it uploads its logs to S3 and updates the DynamoDB record for that job with the S3 object's URL. In order to return a result we need to parse this file and apply our classification rules. We do this using a Lambda function by DynamoDB updates. If a job has a status of "PENDING CLASSIFICATION", it will download its log file from S3, pass it to the classifier module, and update the database with the result.

The classification rules we use are simple, and rely on the assumption that a legitimate page will not

redirect to a new domain after completing a form. We decided that URLs being hosted on a domain already in our target list is likely too high risk to automatically classify. If a phishing site is found here, the most likely reason is that the target’s own site has been compromised. This would need manual review to determine if this is definitely the case, since if a page on a well known target is made inaccessible through a block placed on a phishing feed, the target could suffer a severe reputational and financial loss.

The next rule is to see if the page instantly redirects from an unknown domain to a known target’s domain. This is based on an observation that many URLs we encountered in phishing feeds instantly redirected to targets. We suspect that some may be performing a cross-site scripting attack (XSS) where they are attempting to use authenticated cookies in a user’s browser to perform some malicious action on a legitimate site, or that the URLs reported are for the final page of a phishing kit that redirects to the target. This is a high-risk strategy since there are legitimate reasons why sites will redirect to different domains. However, we assume that any URLs that end up in a manual review queue would have already been tested for simple redirects, and the reason the page has redirected is due to some JavaScript that has been executed as we load the page.

If a page loads, and we encounter a form (or series of forms) which after completion causes the page to navigate to a known target’s domain, we mark it as a phish.

Finally, if the site didn’t redirect, or we didn’t find any forms, we check the page hashes of every page seen since visiting the initial URL to see if any hashes match those of any known phishing sites. Known sites are added using a separate API endpoint which adds a page hash to a DynamoDB table

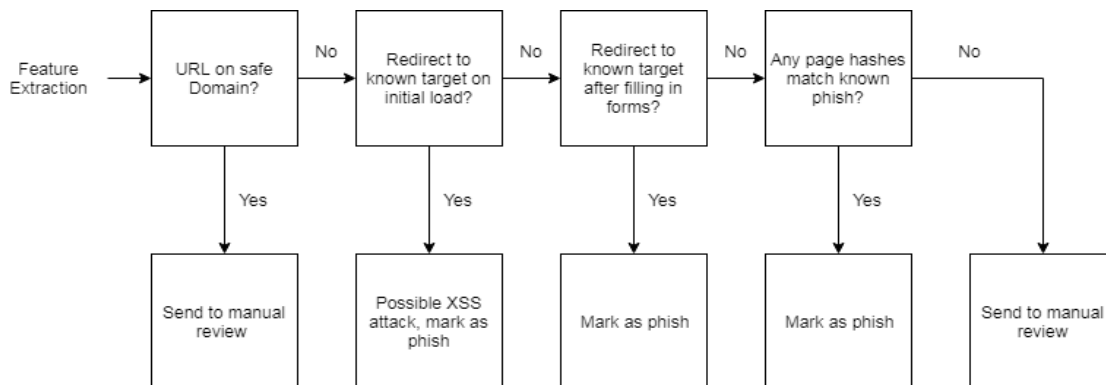


Figure 24: Classification decision tree

4.6 Web Interface

In order to evaluate the classifier and to test the APIs, we produced a simple web interface. It is a statically powered web-page using the Bootstrap framework[64] and is hosted on AWS Cloudfront[65], Amazon’s content delivery network (CDN) service. Cloudfront allows us to use Lambda functions to provide authentication.

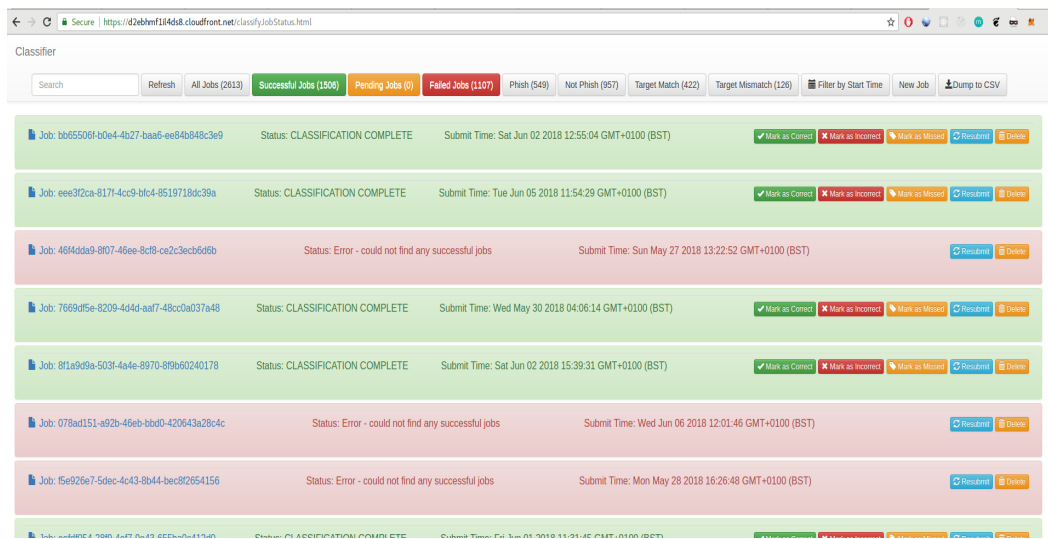


Figure 25: Default view of all jobs

Once signed in, it produces a view with the status of all jobs in the database, colour coded by their status. Clicking on the tabs on the navigation bar allows us to sort by status code and classification result. We are able to search by a job's id, or filter by submission time. When the page loads, it requests and receives a JSON encoded array containing all job ID's, along with their submission times and a computed status. This is done server-side by the Lambda function powering our API. For each function it checks if the job is still pending in any region. If it isn't, it will return a status of complete if any are successfully completed (seen as a green bar in figure 25), or an error if the job did not succeed in any region (seen as a red bar in figure 25).

Clicking on a job will query the API for the results from each region (as seen in figure 26).

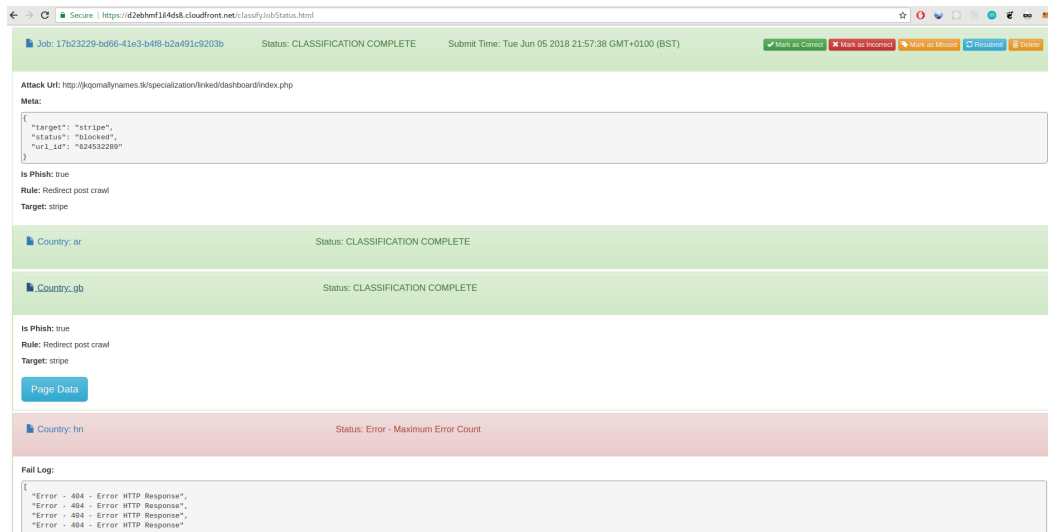


Figure 26: Clicking on a job reveals a breakdown of result by region

The web interface also features a simple manual review feature. Any successful job can be viewed, with the screen shots of each page visited during feature extraction loaded from S3 (as seen in figure 27).

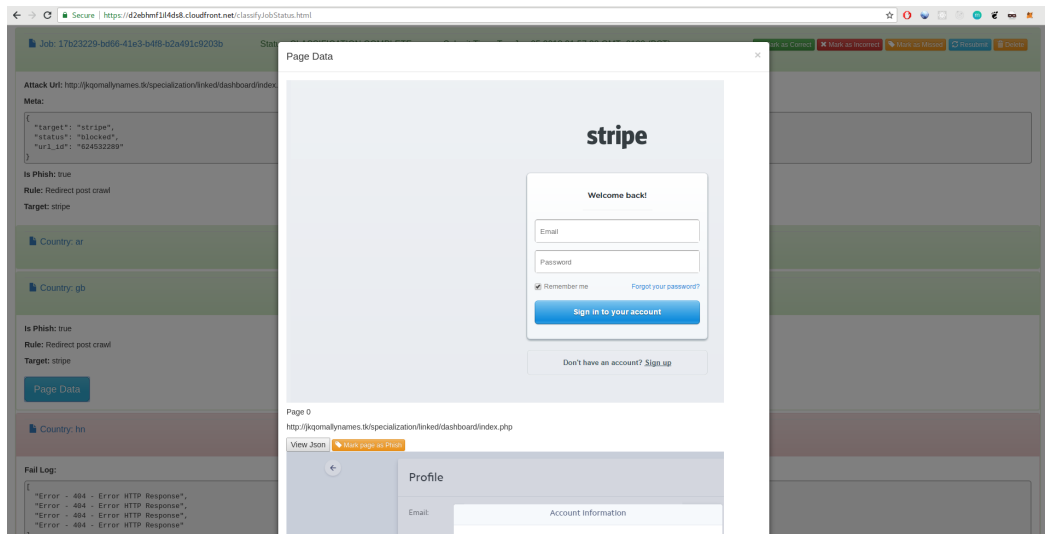


Figure 27: The manual review tool allows easy manual analysis of the pages the crawler encountered

Pages can be manually marked as being a phish of a particular brand (see figure 28). This submits the page hash to a separate DynamoDB table which means if the same phishing kit is being hosted on other URLs, we will be able to correctly mark it. This is particularly useful for sites which do not redirect after filling in forms (or fail). Access to the feature extraction logs is also provided, which can be useful in developing better regular expression matchers to solve similar phishing forms in the future.

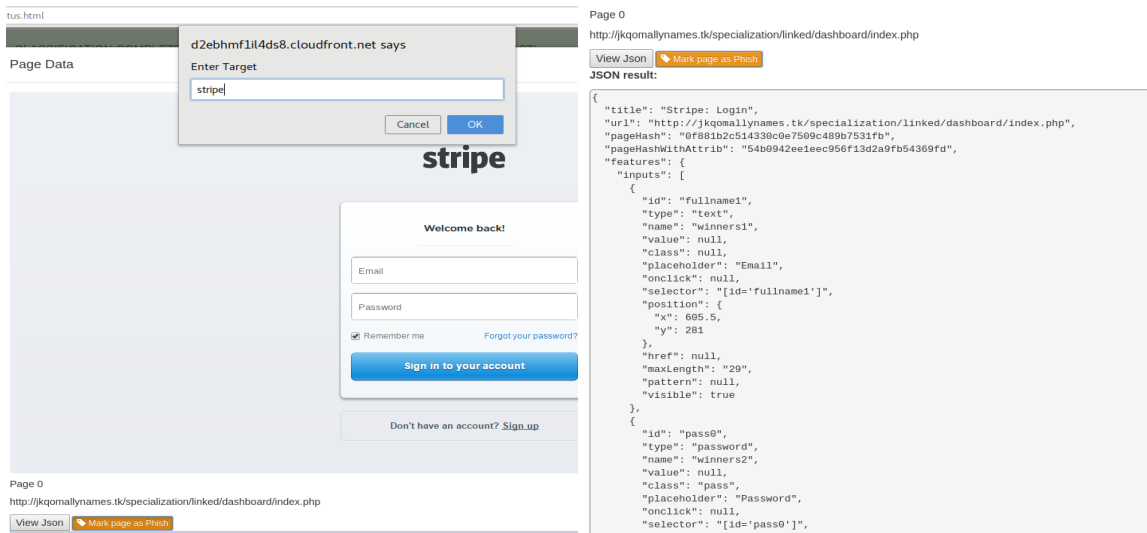


Figure 28: The tool allows manual classification to be submitted as well as viewing of feature extraction logs

5 Evaluation

In this section we evaluate the data produced by applying our distributed classification system against live URLs, first on URLs which have been identified using existing classifiers, and then on URLs that have failed multiple types of existing automated classification and have been sent for manual review. We attempt to identify optimal classification rules based on this data. We evaluate the performance of the distributed system and produce a cost analysis.

5.1 Per target testing for general URLs

We tested our distributed classifier against URLs confirmed to be targeting 6 distinct brands from Netcraft’s phishing feed system: Amazon, Netflix, Blockchain, Visa, HSBC and HM Revenue & Customs. These were chosen since they each had the largest number of reported URLs for their business category. The vast majority of these URLs (> 95 %) were detected using existing classification systems, including URL analysis, content hash matching and image hashing. All together we tested over 10000 unique URLs that were marked as live.

Target Brand	Category	URLs
Amazon	Telecommerce	3191
Blockchain	Cryptocurrency	573
HM Revenue & Customs	Government	214
HSBC Europe	Banking	103
Netflix	Entertainment	6103
Visa	Payment processor	338
Total		10522

Table 8: URL breakdown by target

5.1.1 Crawler success rates

Before we could produce a classification decision on these URLs, we first needed to successfully crawl each url. Table 9 shows the success rate broken down by target. We define a successful crawl as

$$\text{Successful Crawl} = \begin{cases} 1 & \text{if a crawler node completes in at least one geographic region} \\ 0 & \text{otherwise.} \end{cases}$$

Target Brand	Total URLs	Success	Failed	% Success
Amazon	3191	2683	508	84 %
Blockchain	573	478	95	83 %
HM Revenue & Customs	214	177	37	83 %
HSBC Europe	103	91	12	88 %
Netflix	6103	5173	930	85 %
Visa	338	231	107	68 %
Total	10522	8833	1689	84 %

Table 9: Crawler success rates

Around 17 % jobs failed due to the crawler spending more than 4 minutes trying to solve a form, and subsequently timing out.

Of the 8833 jobs that did complete, 579 instantly redirected to their target. 907 jobs were unable to find any forms. We suspect that many of these were probably suspended pages that we didn’t catch with expression matching or faulty pages that served blank pages. We filled out 1541 forms that didn’t redirect anywhere after completion, and 5676 forms that did. If

we assume the worse-case scenario, where all of the 293 validation failures and the 1541 forms that didn't redirect occurred on phishing sites that would have redirected after completing a form, instead of always rejecting input, we achieve a successful form completion rate of 76 % !

However, the completion rate is likely to be higher due to the likelihood of phishing sites failing, and the likelihood of some sites not redirecting by design.

Target Brand	4XX Code	5XX Code	Soft 404	Suspended	Timeout	Browser Error	Validation failure	Total
Amazon	130	4	100	24	210	0	40	508
Blockchain	3	14	0	9	34	5	30	95
HM Revenue & Customs	7	0	0	4	5	0	21	37
HSBC Europe	0	0	0	5	3	0	4	12
Netflix	327	24	1	200	204	1	173	930
Visa	43	5	0	21	13	0	25	107
Total %	30 %	3 %	6 %	16 %	28 %	c. 0 %	17 %	

Table 10: Failure rates

5.1.2 Target Identification using redirects

Using the classification plan described in Section 5, we achieve the results presented in table 11, with a breakdown by target in table 14 . Overall we achieve a false positive rate of 2.75 %.

TP	FP	FN	Precision	Recall
6084	172	171	0.9725	0.9723

Table 11: Identification using redirects' performance on any input

If we choose to discard results from sites containing forms where our crawler could not identify the type of any text input, and only entered random values, the false positive rate drops to 1.7 %.

TP	FP	FN	Precision	Recall
6038	103	103	0.9832	0.9832

Table 12: Identification using redirects' performance on sites with at least 1 matched input

Finally, if we discard any site where we didn't match on every single text input on the page, the false positive rate drops to 0.8 %. However, this does reduce the number of matches by 36%.

TP	FP	FN	Precision	Recall
4213	33	33	0.9922	0.9922

Table 13: Identification using redirects' performance on sites with only matched inputs

These are quite promising results, since we can see that the recall rates for every target except Visa are very high. This seems to show that the vast majority of phishers do redirect to their intended target, with only a relative handful redirecting to completely unrelated sites.

We suspect the reason for the low recall rate of Visa phish is that very few phishers target Visa directly. They tend include Visa branding on fake payment pages targeting other brands.

If we take the example of a URL that was labeled as a Visa phish but ultimately redirected to Netflix (see figure 29), when looking at the screenshots the crawler generated, we can see that this is unlikely to be a Visa phish. For example, next to the Visa logo, there is an equally sized MasterCard logo, suggesting that this is in fact a more generic payment page. If we look at the form, it is requesting a password to verify a payment. Card-processors often ask customers to verify large online payments by requesting a password known only by the user. They usually ask for a random selection of letters or numbers from it. Here the phisher is asking for the whole password. This is clearly part of a longer form consisting of at least one page asking for a credit card number. In the text, it has labeled the payment as being "From Netflix". What we suspect has happened is that the person who reported the phish submitted this URL instead of the one from the first form. In fact, all 21 reported false negatives turned out to be payment forms for the targets they ended up redirecting to.

We also noticed that the false positive rate may be improved if we count navigating to the home page of a domain as a redirect. We noticed that a common reason for a misclassification is that a phishing page located on a compromised site redirects to the home page of the compromised site. Since we have not changed domain, if there is a form on that page (usually a search box, or a subscription form), we treat it as part of the same form. A number of email mailing lists are managed by the email marketing provider Mailchimp, helping explain why 33 % of the mistaken targets left when filtering for only matched inputs redirected there.

Target Brand	TP	TP rate	FP	FP rate	FN	FN rate	Precision	Recall	Mistaken for
Amazon	1555	0.9586	0	0	67	0.0413	1	0.9586	bild: 57, mobilepagol: 4, linkedin: 2, monster: 1, ebay: 1, capitalbank -botswana: 1, google: 1
Netflix	4084	0.9812	1	0.0004	78	0.0187	0.9997	0.9812	mailchimp: 10, example.com: 62, microsoft: 5, wordpresscom: 1
Blockchain	337	1	0	0	0	0	1	1	
Visa	6	0.2222	0	0	21	0.7777	1	0.2222	antivir: 3, spotify: 2, cardcomplete: 10, bncca: 2, netflix: 1, nordea: 3
HSBC Europe	31	0.9393	0	0	2	0.0606	1	0.9393	hewlettpackard: 1, pinterest: 1
HM Revenue & Customs	71	0.9594	0	0	3	0.0405	1	0.9594	microsoft: 3

Table 14: Target identification statistics

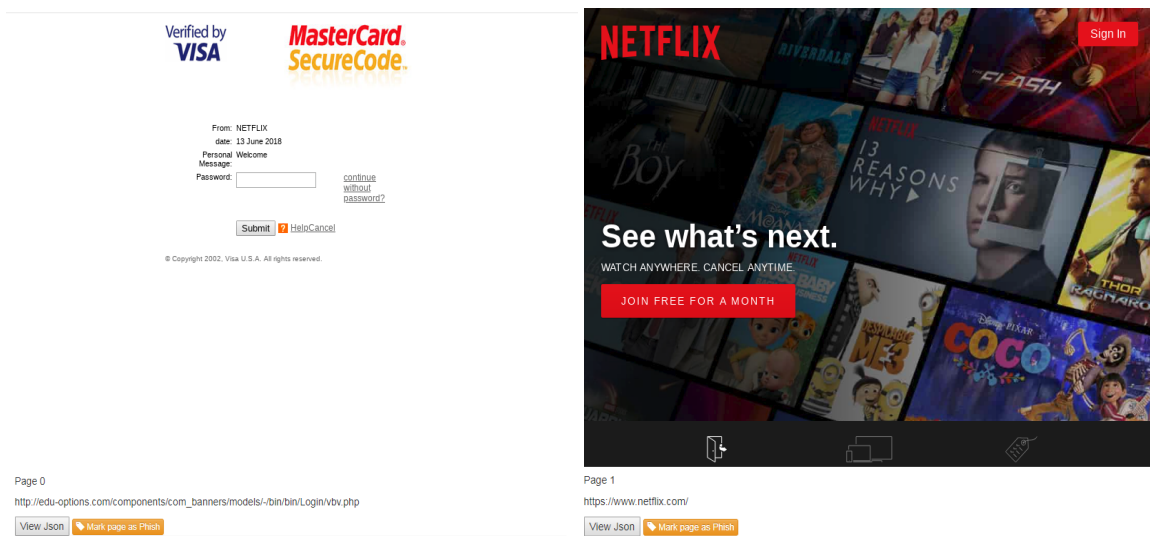


Figure 29: Example of confusion caused by some reported links. Is this targeting Visa or Netflix?

5.1.3 Target Identification using extracted features

We decided to compare the results of redirect identification with the identification method described by Rao et al[33]. They proposed using hyperlinks on the page located directly after filling in a password field. Since we are focusing on all form filling based phishing, not just login scams, we extract all URLs from hyperlinks from all pages, check them against our target list, and find the most common matched target for each page. We then find the most common target of all those targets to determine the brand a phishing site is targeting.

The results can be seen in table 15 and a breakdown by target can be seen in Appendix A.

TP	FP	FN	Precision	Recall
2901	253	253	0.9198	0.9198

Table 15: Identification using hyperlink destinations' performance

We find that it only matches half as many correctly as our classifier, and with a false positive rate of over 8 %, it is much less suitable to use as a reliable identifier of targeting.

Finally, we analysed identifying the target by looking at the most common identified domain from all connection requests made by each site. It achieved a better false positive rate than hyperlink matching, mainly due to the greatly reduced false negative rate.

The results can be seen in table 16 and a breakdown by target can be seen in Appendix B.

TP	FP	FN	Precision	Recall
2784	79	79	0.9724	0.9724

Table 16: Identification using requested domains' performance

5.2 Testing using manual classification feed

5.2.1 Overview

One of this project's aims was to try and perform reliable classification on sites which have failed to be classified by other techniques, and that are currently only solved by human reviewers.

Over a period of 2 weeks we received URLs from Netcraft's manual classification system, along with the classification assigned to it by a human. Whenever a manual decision was submitted into the system, we received a job on our classifier using the REST API, with the reviewer's decision attached as metadata.

Jobs were assigned one of 4 classifications, with a "blocked" status for a positive identification of a phishing site, and three degrees of negative identification, depending on the confidence of the decision, ranging from "dismiss", where the reviewer is not fully satisfied that it is legitimate, but has erred on the side of caution, through to "delete", where there is strong confidence that a site is legitimate, and finally "safe", where a site is certainly legitimate.

We received 5692 unique reports during this period, of which 1162 were marked as benign during human classification, and 4530 reports of sites targeting over 480 different brands. A breakdown of the URLs by their status code can be seen in table 17.

Our aim is to achieve as low false positive rate as possible, while also minimising the amount of URLs which we fail to make an identification on. If our classifier was used in a commercial system, these would be the task we would pass on to Manual Review.

5.2.2 Crawler success rates

Table 18 shows a significantly larger proportion of sites were inaccessible than in our previous experiment (30%). This time, failure to complete forms made up only 27 % of the failures, with the vast majority being 404 status codes. The poor reliability of some of these sites was the most likely reason why these links had failed previous automated classification and were now manually being reviewed. Of the 647 failures to complete a form (either because we tried all our matches and we timed out while solving one), 120 were on legitimate sites while 527 were on suspected phishing pages.

Error	URLs	%
Blocked	4530	80
Dismiss	75	1
Delete	1069	19
Safe	18	<1
Total	5692	

Table 17: Breakdown of jobs by status code

Error	URLs	%
40X Error Code	786	33
50X Error Code	336	14
Soft 404	7	<1
Suspended	63	3
Timeout	449	18
Browser Error	120	5
Validation Failure	647	27
Total	2408	

Table 18: Breakdown of crawler failures

5.2.3 Redirect to target results

A significant number of sites did not redirect at all. However, only 52 % of those actually contained forms. We suspect that many of those that didn't may actually be phish that have already been taken down, and we have failed to match on the account suspended page. A breakdown of redirect results is shown in table 19.

Reason	Safe	Phish	%
Redirect to known domain	40	1001	32 %
Redirect to unknown domain	105	240	11 %
Did not redirect or contained no forms	496	1252	52 %
Ignored as URL on a safe domain	88	62	5 %

Table 19: Breakdown of redirect results

When applying our classification rules presented in Section 4 without checking the inputs we achieve a high false positive rate of 23.3 %. This achieved a reduction of URLs sent to manual review of 18 %.

TP	FP	FN	Precision	Recall	% Classified	% Sent to Manual Review
760	211	207	0.7827	0.7859	18 %	82 %

Table 20: Identification using target redirection's performance

When applying our strict rule set ensuring we only classify pages where we matched on every input, we found that every single URL redirected after completing a form. However, the false positive rate was extremely high at 23 %, while only filtering 7 % of the manual classification jobs (see table 21).

TP	FP	FN	Precision	Recall	% Classified	% Sent to Manual Review
239	144	988	0.6240	0.1948	7 %	93 %

Table 21: Identification using strict rule set's performance

5.2.4 Case studies

We decided to investigate possible reasons why the target mapping had been so unreliable. When looking at the target mappings, we noticed that the vast majority of false positives were caused by slightly different variations between the names of targets in our blocklist and the named targets we received from the manual classification queue. For example, we matched a phish targeting the mobile network Everything Everywhere (EE) as "everythingeverywhere", while the metadata in the task had the label "eenetwork". Another example was simply different spacing in the name: "hmrevenuecustoms" failed to match with "hm revenue & customs". We identified 45 such cases.

We also noticed a number of pages that were marked as not phish during manual classification, but after manually viewing the screenshots, appeared to be phishing sites. For example, 4 URLs marked as safe pointed to a login page written in Chinese (see figure 30). After entering a password, it redirected to a well known Chinese email provider. On each visit, we have entered randomly generated email addresses and passwords and it has accepted every one.

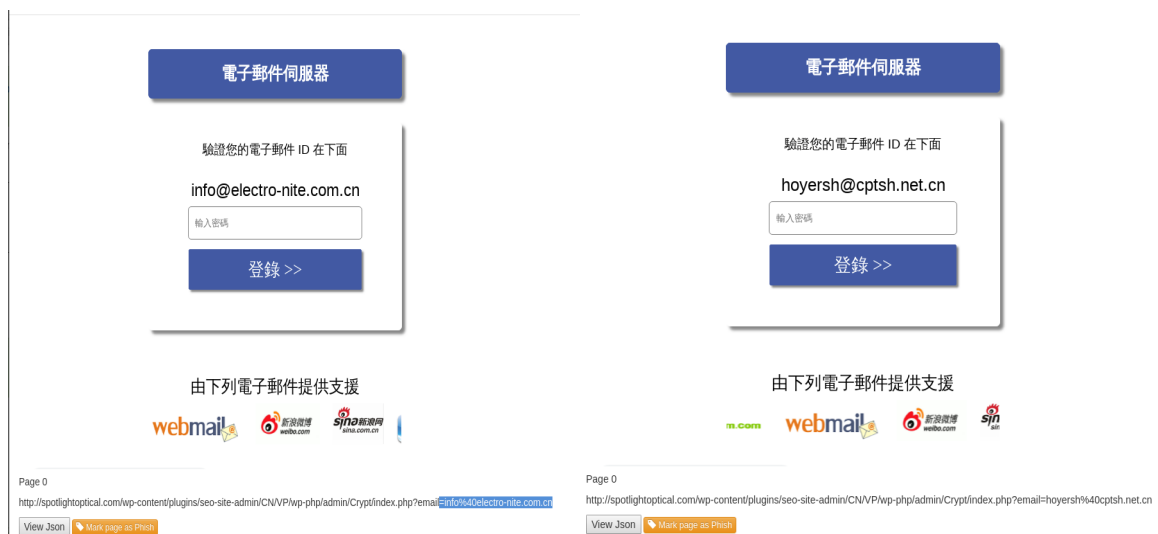


Figure 30: Example targeting the popular Chinese email provider 163.com

Another example of a phish that was not spotted by manual review was one targeting the online game Runescape. It presented a fake login form, before asking for a user's 6 digit verification code before redirecting to the legitimate Runescape site. We identified 14 misclassifications amongst the false positives of the strict classification results. After adjusting for these our false positive rate improves to just under 15 %.

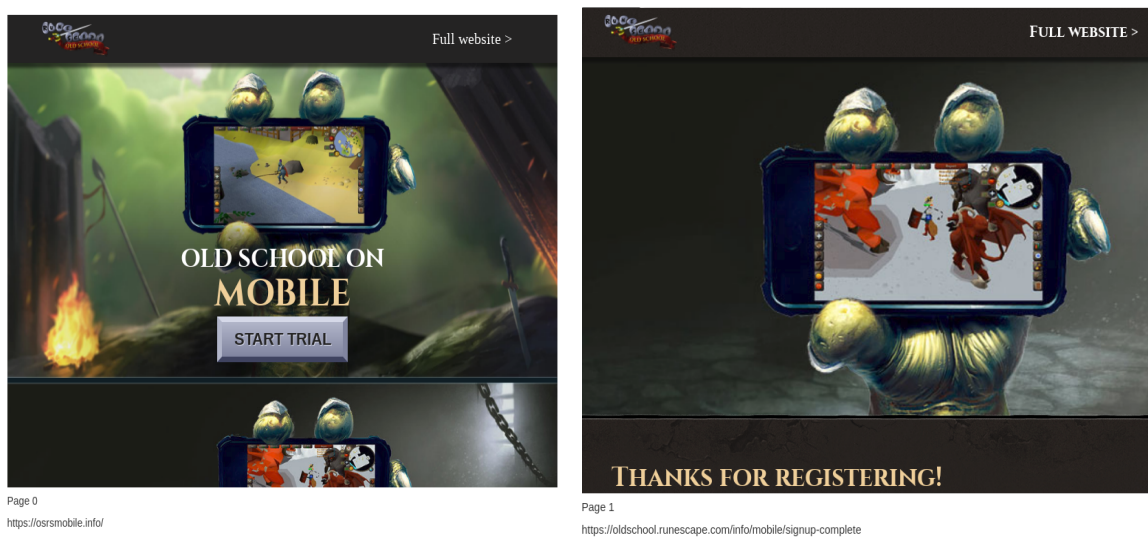


Figure 31: Example targeting the popular online game Runescape

One of the largest causes of false identification was generic web-mail phishing pages, which behaved similarly to those we saw at the beginning of this project. Often these pages make very little effort to imitate a particular site's branding, so there is less value in redirecting to a relevant site, since the user will notice the large difference in page style anyway. In these cases, it may even be beneficial to redirect to an unrelated domain, since a less-technically aware victim may assume that they themselves had accidentally navigated to a page, instead of being deliberately redirected. A possible solution to recognise these pages would be to look for multiple button inputs on the page before the form.

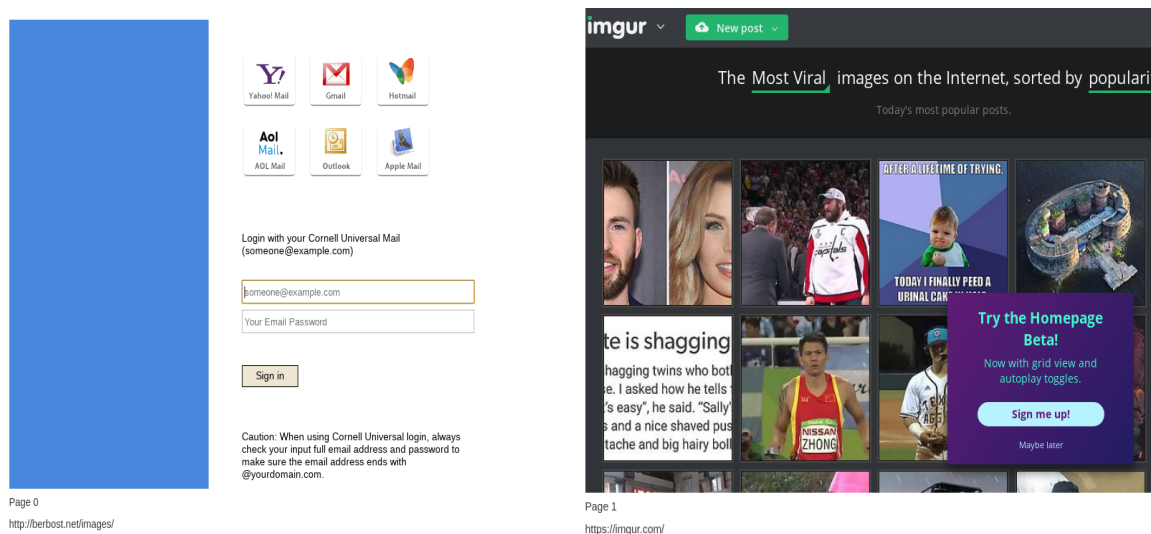


Figure 32: Example of a generic email phishing page redirecting to the unrelated Imgur image site.

There were also some cases that at first glance seemed to behave exactly as a phishing site, but were legitimate pages. One notable example was a suspicious URL that seemed to behave like a Wells Fargo phish. After completing the login form on the first page, our crawler was redirected to the legitimate Wells Fargo page, and consequently it was reported as a phish. However, it turns out that the suspicious looking URL was actually a Chinese translation service accessing the Wells Fargo page with the language set to English. We note the similarity of the behaviour to a real Wells Fargo phish, shown at the top of figure 34. Without knowing that the Chinese domain was safe, it would be difficult to catch such a case. Using selective blacklisting is likely to improve the success rates of our technique significantly, but it would likely take a while to find a good balance between the false positive rate and the recall. For our survey, since so many of our false positives redirected to Mailchimp or Google, blacklisting these domains drops the false positive rate to under 5 %.

We noticed that we could sometimes time-out on complicated phish that didn't redirect after filling a form. This is because we always attempt to search one-level deep for another form, as sometimes a site asks for a confirmation button to be pressed before displaying the next form. If there are a lot of links to explore, we sometimes reach our job timeout limit and fail to return a result. While we could increase the limit and allow crawling jobs to run for longer, this is likely to clog up the crawler queues with slow responding sites. We could apply some filtering to the links extracted from a page to try, and instead of sequentially clicking them, try clicking on links on different sections on the page each time, to prevent wasting time on inactive regions of a page.

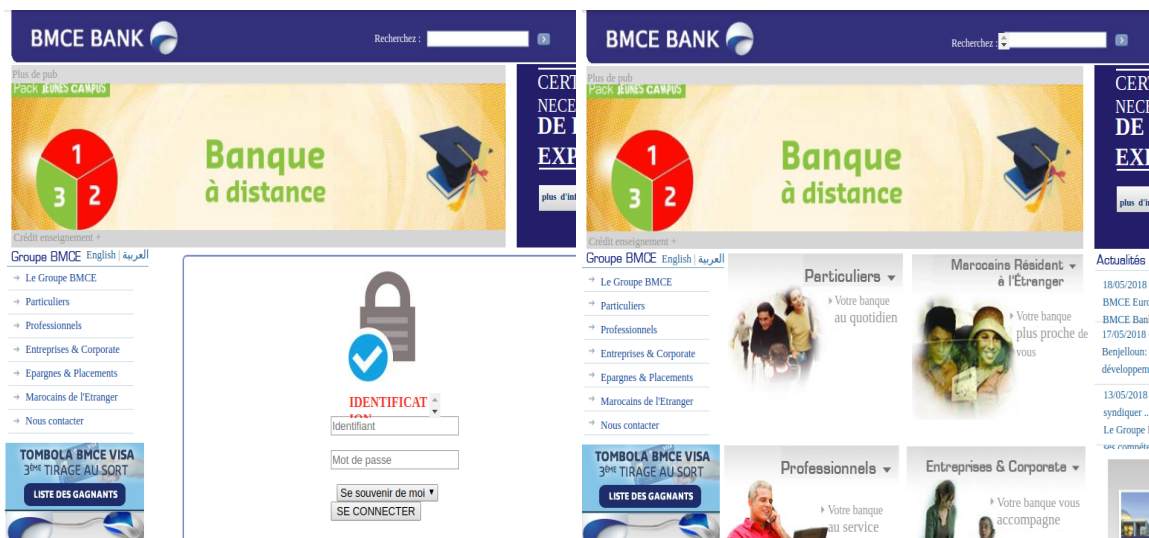
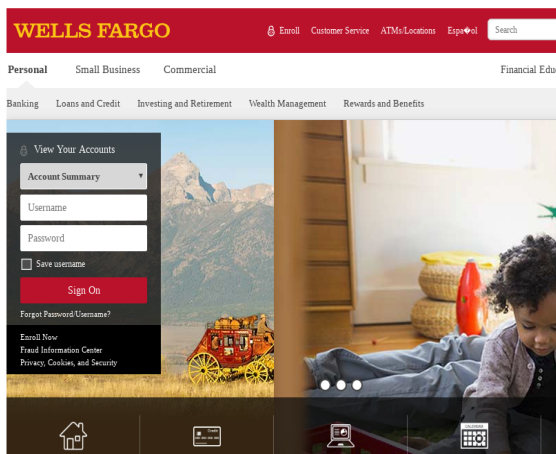
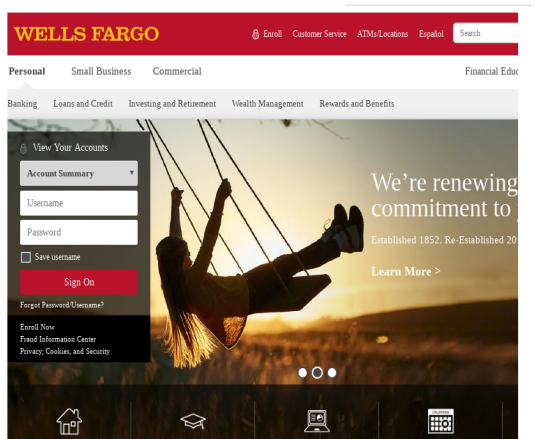


Figure 33: Almost full clone of target site causes classifier to Time out before clicking on all the links

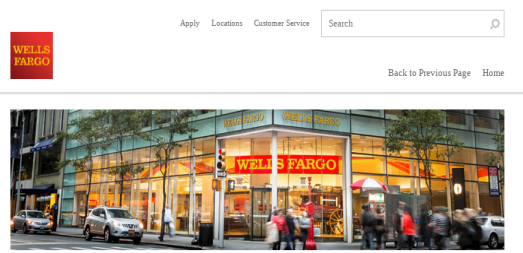
Finally, using the strict approach to matching caused our recall rate to drop to a tiny fraction of the URLs presented to manual classification. However, currently we have only developed a limited set of matchers based on a few target sites (Amazon, Netflix, Blockchain, HMRC, HSBC and Visa). This helps explain some of the discrepancy between the success rates when we tested URLs from those targets to more general targets found in the manual classification queue. We believe that if the system was limited to only producing positive results for targets which matchers have been tested on, with extra targets being added by simply adding more matchers to the system, it could still potentially be a useful filter for existing manual classification queues.



Page 0
http://robert-blair.com/wp-content/wellsfargo/
View Json Mark page as Phish

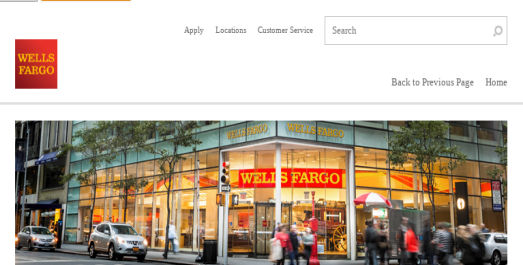


Page 0
https://translate.sogoucdn.com/pcvtsnapshtorlogin?url=https%3A%2F%2Fwww.wellsfargo.com%2F&query=&tabMode=1&noTrans=0&tr=default&from=auto&to=zh-CHS&_t=1527367595907
View Json Mark page as Phish



Sign On to View Your Accounts
Enter your username and password to securely view and manage your Wells Fargo accounts online.
Account Summary
Username Password
Save Username
Related Information
Enrollment FAQs
Online Security Guarantee
Privacy, Security and Legal
Online Access Agreement
Other Services
Applications In Progress
Credit Card Rewards

Page 1
https://connect.secure.wellsfargo.com/auth/login/present?origin=cob&LOB=CONS
View Json Mark page as Phish



Sign On to View Your Accounts
To protect your account, please enter the moving letters and/or numbers in the CAPTCHA, and re-enter your username and password to continue.
Enter your username and password to securely view and manage your Wells Fargo accounts online.
Account Summary
Username Password
Save Username
Related Information
Enrollment FAQs
Online Security Guarantee
Privacy, Security and Legal
Online Access Agreement
Other Services
Applications In Progress
Credit Card Rewards

Page 1
https://connect.secure.wellsfargo.com/auth/login/present?origin=cob&error=yes&destination=MessageAlerts

Figure 34: The top two images are examples of a phish targeting the Wells Fargo bank. The bottom left is the legitimate site viewed through a Chinese online translation service, and the subsequent redirect to the real Wells Fargo page

5.3 Scalable Distributed Classification

5.3.1 Resources and throughput

Based on our memory usage observations in Section 3, where a single version of the crawler code used around 900 MB of peak memory, and an initial plan to run 204 classifier nodes, we provisioned a cluster of 21 Virtual Machines (VMs) with around 160 GB of memory, expecting the memory usage to be almost fully saturated. Since we test every URL in 6 regions, this allows 34 jobs to be operated on concurrently.

However, we found that our memory usage was much lower than expected, with the statistics gained from Sematex [66], a resource monitoring tool, showing only around 80GB of memory in use, and an average CPU load of around 6%.

In fact our main bottleneck at this scale is that we were overwhelming the web accessible proxy service at Netcraft with the amount of web requests we were making to a single endpoint.

To solve this issue, we added four new VMs to the new Swarm cluster on the Digital Ocean cloud platform. These 4 VMs were white-listed to access the internal commercial Virtual Private Networks (VPNs) that power the proxy service directly.

By offloading some of the pressure from the proxy service, we were able to expand the number of nodes to 250 per region, 1500 nodes in total.

Figure 35 shows the effect on the resource utilisation of the cluster as the number of workers is increased from 500 to 1500. The memory usage seems to have barely been affected (in fact it even seems to decrease slightly) but the CPU usage increases by a factor of 4. The significant increase in disk writes reveals the increased usage of the swap file. The fact that the disk write rate is significantly higher than the reads suggests that a large chunk of the memory footprint of a browser is not used when in headless mode, so by having large swap files we are able to run significantly more instances than initially expected.

However, trying to run more crawler nodes than 1500 did start to cause instability in our cluster. We noticed some swarm nodes were starting to terminate processes due to lack of memory while others in the cluster were fine. It turns out that the swarm task scheduler had not distributed the share of docker containers equally between each node. We had set the scheduling strategy to "spread", which means that when starting a container, it first looks for the swarm node with the most free system resources, and then for the one with the fewest containers. Since a crawler process that has not yet started an instance of Chromium has much smaller resource usage than a crawler instance actively processing a URL, the scheduler may have scheduled too many on a single node at once. When they received a job from the queue and started a browser, they then proceed to overwhelm that node.

The only other part of our distributed system that requires manual scaling is DynamoDB. DynamoDB is billed using "throughput" units, meaning we specify how much traffic we want DynamoDB to be able to handle, and Amazon handle the scaling. A read unit allows 20KB/s of data to be accessed using consistent reads, and double that for eventually consistent. A write unit is more restricted, only allowing 5KB/s. Our job locking mechanism relies on syncing on writes, meaning we can use eventually constant reads to double our throughput. As seen in figure 36, Amazon does allow us to operate above the provisioned level for a few minutes before they start to throttle access to a table.

We found 15 write units (allowing 75KB/s) to be the minimum level of throughput needed to run 1500 crawler nodes.

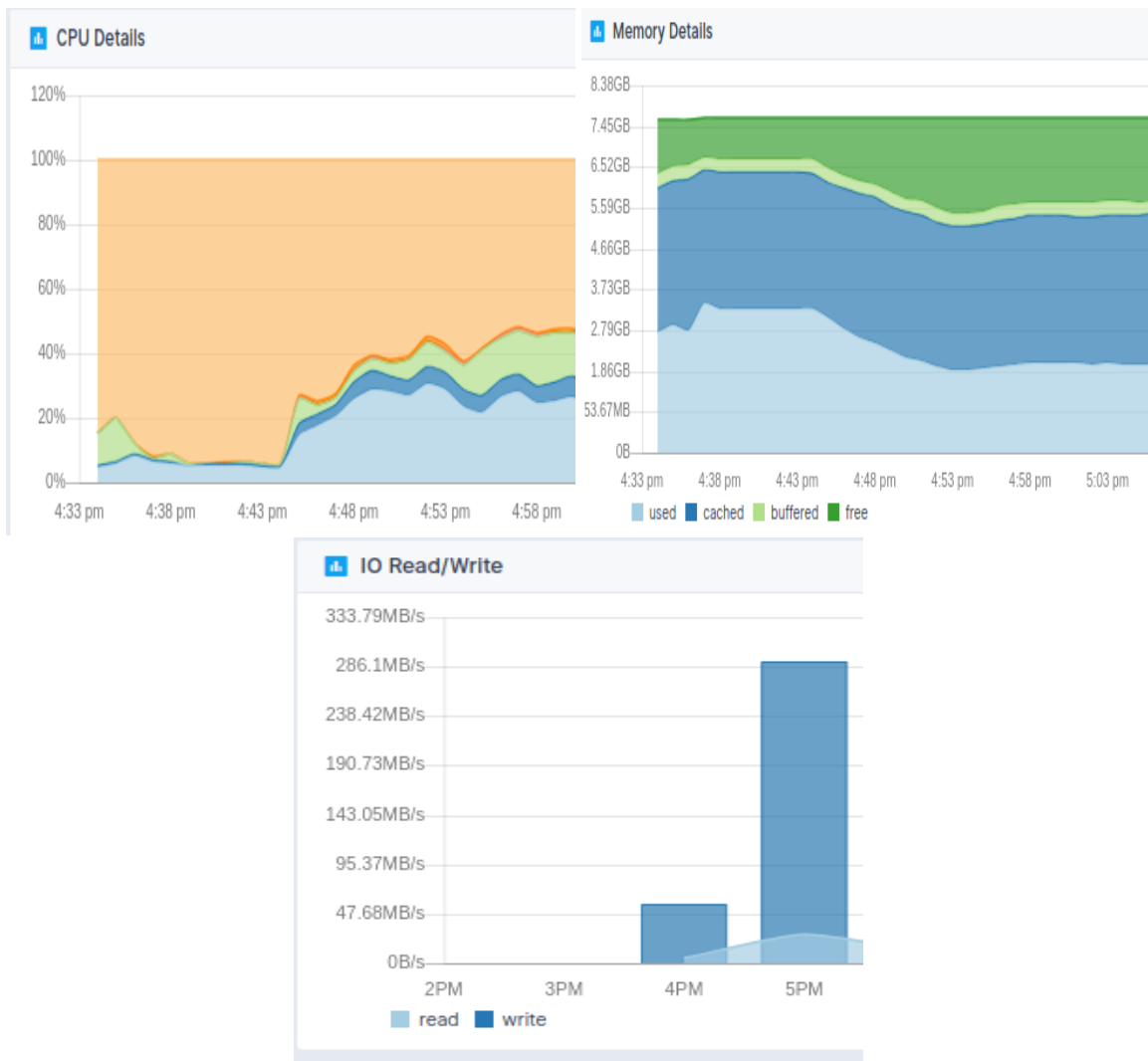


Figure 35: Aggregated and averaged CPU and Memory usage data

Provider	Processor type	Cores	Clock -speed (GHz)	Memory (GB)
Imperial CloudStack	Intel(R) Xeon(R) CPU E5-2690 v2	56	3.0	163.23
DigitalOcean Droplet	Intel(R) Xeon(R) CPU E5-2650 v4	16	2.2	33.48

Table 22: Resources assigned to our crawler cluster

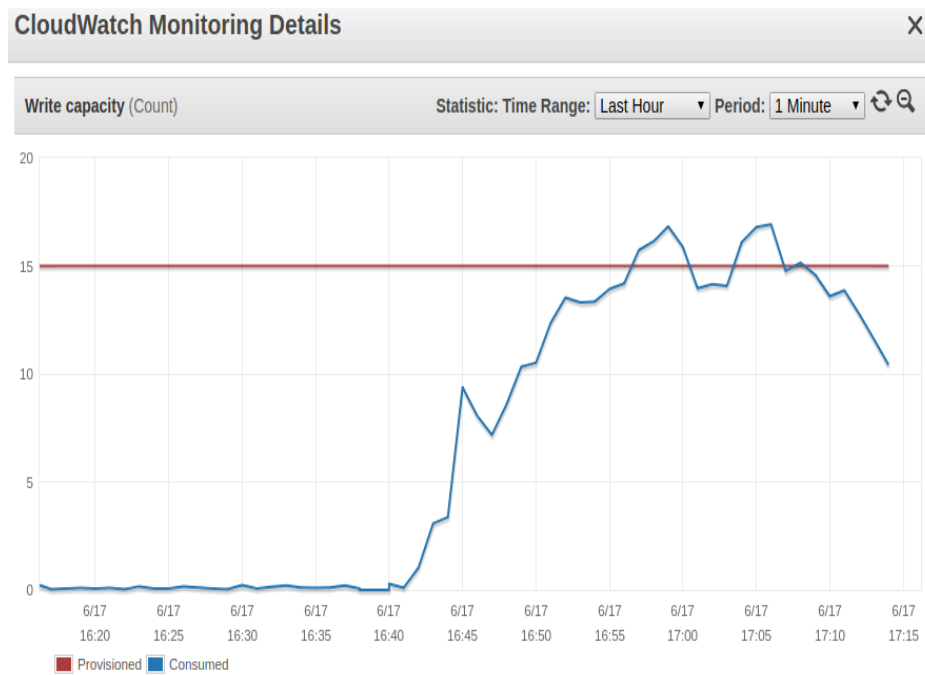
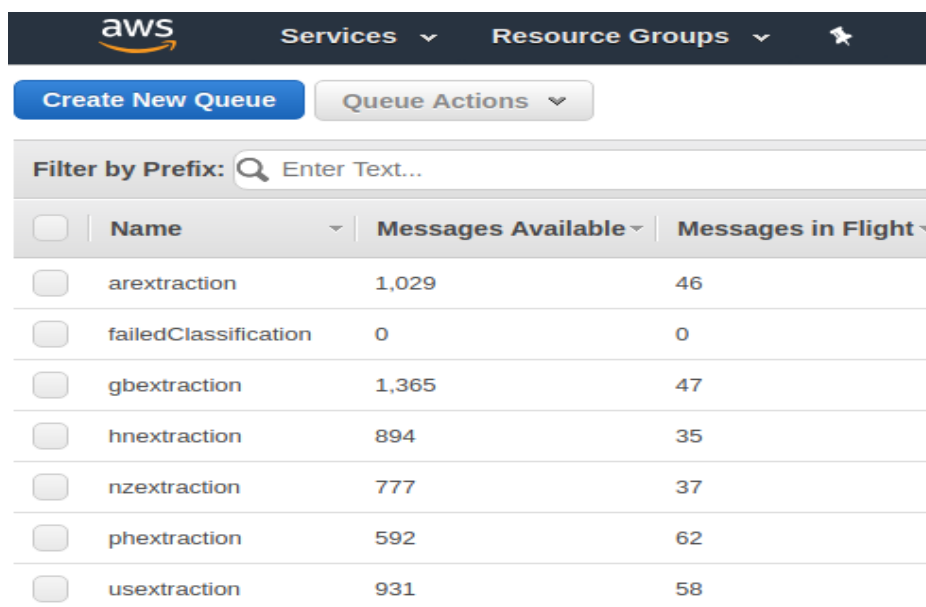


Figure 36: DynamoDB write capacity graph

To benchmark our system, we saturated our job queue with URLs from Netcraft’s phishing feed and measured how many jobs were completed. In a 10 minute period we were able to complete jobs for 426 URLs. This produces a rough URL rate of just under one URL a second. However, the time a job takes to complete can vary significantly. Each URL received by the system is assigned to 6 separate jobs for each of the regions we test in. If any of those jobs fails, it can be retried up to another 5 times. We did notice that occasionally we would end up with a “straggler”, a region that has built up such a back log of jobs that it causes the classification wait time to take anywhere from several minutes to several hours. In figure 37, we can see the GB region starting to fall behind the other 5. Reassigning nodes from one queue to another can help rebalance the queues.



<input type="checkbox"/>	Name	Messages Available	Messages in Flight
<input type="checkbox"/>	arextraction	1,029	46
<input type="checkbox"/>	failedClassification	0	0
<input type="checkbox"/>	gbextraction	1,365	47
<input type="checkbox"/>	hnextraction	894	35
<input type="checkbox"/>	nzextraction	777	37
<input type="checkbox"/>	phextraction	592	62
<input type="checkbox"/>	usextraction	931	58

Figure 37: A screenshot from Amazon SQS, showing some regions with hundreds more pending jobs than others.

5.3.2 Cost analysis

The majority of the costs for running a distributed classifier is the cost of compute time for performing the classification. We ran the majority of our VMs on a university platform incurring no costs to ourselves, but we can estimate the costs of our cluster by using the cost of similar VMs at Digital Ocean.

Since we did not fully utilise the CPU resources of our VMs, we will estimate pricing using the lower clock-speed VMs (2.2GHz) for all 72 cores.

As of June 2018, an 8GB, 4 CPU VM costs \$40 a month, approximately \$0.056 an hour. 18 VMs would cost around \$1 an hour. A DynamoDB table with 100 read throughput units and 15 write units costs \$20 a month, about \$0.028 an hour. The lambda functions are harder to estimate a cost for, since they are billed by the millisecond of run time. Each job will require at least 2 lambda functions to be invoked, one to create the jobs in the DynamoDB table and add the jobs to the queue, and another to apply the classification rules. Lambdas cost \$0.000000834 per 100-ms when set with a memory limit of 500 MB. If they run for, at most, 5 seconds, it costs around \$0.0000417 per job. Assuming a rate of 1 URL a second, we will amass around \$0.90 an hour in Lambda charges. S3 storage cost is dependent on how long you want to store site data for. Currently we store both log files from crawling as well as screenshots. We will assume that we will not keep the files for more than a few hours after a job, meaning the free 5 GB/hours a month on the free tier should cover storage costs for images. In terms of usage fees, S3 costs \$0.0053 per 1000 PUT requests and \$0.0042 per GET. Each job will need at least one PUT request to store the log and one GET to retrieve it for classification. This will cost \$0.0342 an hour at 1 job a second. The final component in our system is the queueing system SQS. Amazon charges \$0.40 per 1,000,000 requests. Crawler nodes will poll the queue if there are no jobs waiting for it. Assuming the worst case scenario that every node polls the queue every 20 s, a cluster of 1500 nodes will cost \$0.108 an hour.

In summary, we have an hourly cost of \$1.1262 an hour. Assuming 1 URL processed a second, we achieve a classification cost of 4054 classifications per dollar.

5.3.3 Success by region

We decided to investigate whether there were particular regions in which URLs failed more commonly, to see if our multi-region approach helps defeat geo-blocking of regions by phishing kits.

First we summed all the successful jobs for each region to see if there were significant differences in success rates. Southern-hemisphere regions seem to have a slightly increased success rate, but the difference between the most successful region - the Philippines, and the least successful - the United States, was less than 700 jobs.

Region	Ranking	Success Count	Average retry count
Philippines	1	11443	2.9064
Argentina	2	11348	2.9311
New Zealand	3	11323	2.9062
Great Britain	4	10875	3.0147
Hungary	5	10771	3.0282
United States	6	10754	3.0597

Table 23: Region ranking by number of successful jobs

However, if we count the successes on a per-URL basis, it reveals that there were a significant number of jobs that only solved in a small sub-set of regions. While the reduced success rate of a URL could be simply down to an unreliable web host or connection, the fact that the two most common scenarios are that a URL is accessible in all regions (45% of the time) or in only one region (15%), and not somewhere in the middle, suggests that there is active geo-blocking occurring and that running in multiple regions is worth having a slightly slower classification rate by waiting for multiple responses.

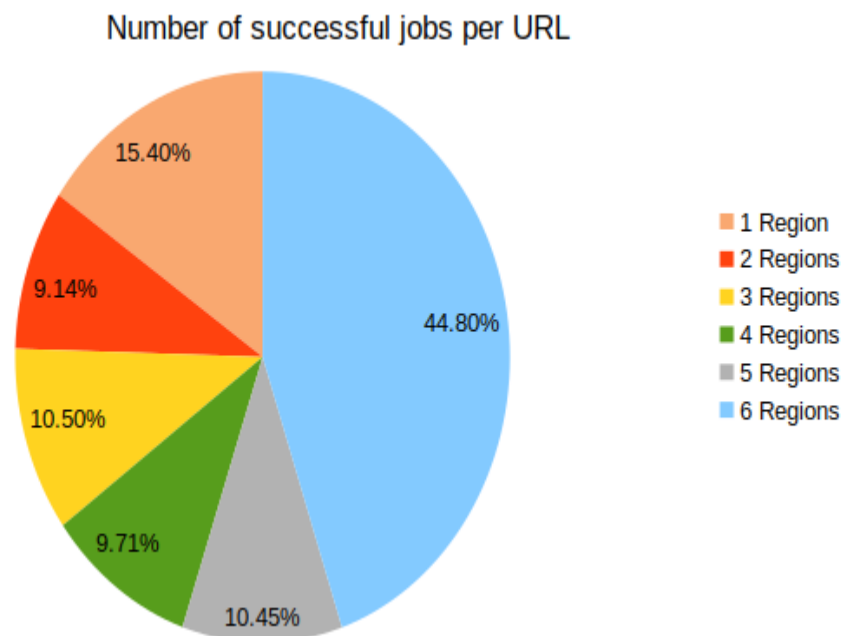


Figure 38: A breakdown of how geographical distribution affects the successful crawling of a page

6 Conclusion

In this section we summarise our project's achievements, and propose some possible future extensions.

6.1 Objectives

6.1.1 Behaviour based classifier

We have been successful in producing a behaviour-based brand-aware classifier that is able to identify the vast majority of phishing URLs found in phishing feeds with high precision and a false positive rate of under 1 %, making it suitable for use as a classifier in a phishing pipeline.

We have been less successful at developing an effective general classifier to solve URLs which currently can only be reliably solved by a human as part of a manual classification system. While we are able to produce results where other classification technologies have failed, our false positive rate of under 15 % is too high to be used directly in its current form, however we propose some methods that can be explored to try and improve its success rate.

The robust crawler we have developed has shown it is able to traverse the vast majority of phishing sites it encounters. It could be used as a platform to run existing pattern-matching and content analysis methods on pages that were previously inaccessible since they were behind several validated forms.

6.1.2 Scalable distributed classification system

We have produced a scalable distributed classification system that can scale to achieve rates of existing commercial systems. We have demonstrated that its simple API makes it easy to integrate into other classification systems, and that it takes advantage of multiple geographical region testing to be able to work around geo-blocking.

The modular structure of the system allows alternative classifiers to be easily slotted into our pipeline, and the manual review web tool shows that we are able to export the data we gather on pages we visit for external use.

6.2 Future Work

We have a few ideas for improvements to the project that we would have liked to have investigated given more time.

6.2.1 Tool for easily creating matchers

We envisage an extension to the manual review tool that makes it easy to add new patterns and suggested values to our crawler nodes. Since we already store the X, Y coordinates for every input we extract, we could highlight areas on the screenshot for a page, and display drop down options for common categories. With a tool like this, it could allow a large enough database of identifiers to be built up for us to be able to identify a target by particular combinations of inputs.

6.2.2 Deal with IFrames

Currently the crawler process is not able to deal with IFrames in the page. The reason for this is that handling IFrames would add complexity to the way we interact with the page using Puppeteer, since we would have to pass information about which frame we are dealing with to every function and module that interacts or monitors a page. Given the time constraints of this project, and the low incident rates of phishing sites using IFrames, we chose to omit IFrame support.

6.2.3 Detect which field has actually failed validation

Currently we only detect validation failure on a whole form. However, phishers will often highlight the field that has failed their checks in a similar manner to legitimate sites. We already extract all elements from a page after submitting forms, but don't currently take advantage of this information. We expect that simply looking for the nearest form to any new element that is created would be a good indicator of which form has failed. Pattern matching on any text in the alert may also give clues about the type of input a form is expecting, increasing the likelihood of correct matching and subsequent classification based on these features.

6.2.4 Protect victims by flooding phish with fake data

A potential use of our distributed platform would be to use multiple crawler nodes to flood a known phishing URL with fake data. Since we are matching on the types of inputs a form is expecting, we can submit real-looking but generated data. This can help protect victims who have already fallen for the phish by burying their credentials amongst these fake ones. Since phishers may not actually test the credentials themselves but simply sell them on to other criminals, the fake data may devalue the phisher's datasets enough to no longer make phishing profitable. At the very least, it could place enough of a strain on the phishing site to take it offline. The relatively low costs of running our distributed system (of the order of a few dollars an hour) could make it an attractive solution for companies such as banks who are often obligated to pay compensation if money is stolen from their customers through phishing attacks.

6.2.5 Generate more insights into phishing campaigns

Our distributed system has already visited in the order of tens of thousands of URLs. In doing so, we have produced several GB of log files, containing unique hashes of all the pages we have seen. After adding just two Netflix page hashes to our manual review list, we saw hundreds of matches, indicating the same phishing kit was being used on many different hosts. Since we log all requests a page makes, we can see if there are trends in where they connect to, the hosting companies that the phishers use, and possible other insights from the large amounts of data we have produced during the crawling process.

7 Bibliography

References

- [1] Anti-Phishing Working Group. *Phishing Activity Trends Report, 3rd Quarter*. 2014. pages 5
- [2] Info-Security Magazine. *Social Media Phishing Attacks Soar 500%*, 2017. <https://www.infosecurity-magazine.com/news/social-media-phishing-attacks-soar/> (Accessed 21-01-2018). pages 5
- [3] IT Governance Blog. *The psychology behind phishing attacks*, 2016. <https://www.itgovernance.co.uk/blog/the-psychology-behind-phishing-attacks/> (Accessed 21-01-2018). pages 5
- [4] Anti-Phishing Working Group. *Phishing Activity Trends Report, 4th Quarter*. 2016. pages 5
- [5] Google Chrome. <https://www.google.com/chrome/> (Accessed 21-01-2018). pages 5, 10
- [6] Mozilla Firefox. <https://www.mozilla.org/en-GB/firefox/> (Accessed 21-01-2018). pages 5, 10
- [7] Anti-Phishing Working Group. *Phishing Activity Trends Report, 1st Half*. 2017. pages 5
- [8] Marria Nazif Colin Whittaker, Brian Ryner. *Large-Scale Automatic Classification of Phishing Pages*. 2010. pages 6, 11
- [9] Oxford Dictionary. <https://en.oxforddictionaries.com/definition/phishing> (Accessed 21-01-2018). pages 8
- [10] Tech Target Brandan Blevins. *Despite skeptics, security awareness training for employees is booming*, 2014. <http://searchsecurity.techtarget.com/news/2240234092/Despite-skeptics-security-awareness-training-for-employees-is-booming> (Accessed 21-01-2018). pages 9
- [11] CYBERSECURITY VENTURES. *Security Awareness Training Report, 2017*. <https://cybersecurityventures.com/security-awareness-training-report-2017/> (Accessed 21-01-2018). pages 9
- [12] Stefan Gorling. *The Myth of User Education*. 2006. Proceedings of the 16th Virus Bulletin International Conference. pages 9
- [13] A. Acquisti L. Cranor P. Kumaraguru, S. Sheng and J. Hong. *Teaching Johnny Not to Fall for Phish*. 2010. ACM Transactions on Internet Technology (TOIT), Volume 10, Issue 2. pages 9
- [14] Anti-Phishing Phill. <http://www.ucl.ac.uk/cert/antiphishing/> (Accessed 21-01-2018). pages 10
- [15] Jason Hong Lorrie Cranor, Serge Egelman and Yue Zhang. *Phinding Phish: An Evaluation of Anti-Phishing Toolbars*. 2006. pages 10
- [16] Too Google's Safe Web Surfing Tool Is Quietly Protecting Apps Like Snapchat. <https://gizmodo.com/googles-safe-web-surfing-tool-is-quietly-protecting-app-1803135437> (Accessed 21-01-2018). pages 10
- [17] Desktop vs Mobile vs Tablet Market Share Worldwide. <http://gs.statcounter.com/platform-market-share/desktop-mobile-tablet> (Accessed 21-01-2018). pages 10
- [18] Pradeep Atrey Gaurav Varshney, Mano Misra. *A phish detector using lightweight search features*. 2016. pages 11

- [19] San Nah Sze Ee Hung Chang, Kang Leng Chiew. *Phishing Detection via Identification of Website Identity*. 2013. pages 11
- [20] Ali Yazdian Varjani Mahmood Moghimi. *New rule-based phishing detection method*. 2016. pages 11
- [21] Lee McCluskey Rami M. Mohammad, Fadi Thabtah. *Predicting phishing websites based on self-structuring neural network*. 2014. pages 11
- [22] Twitter. <https://www.twitter.com/> (Accessed 21-01-2018). pages 12
- [23] About Twitter's link service (<http://t.co>). <https://help.twitter.com/en/using-twitter/url-shortener> (Accessed 21-01-2018). pages 12
- [24] URL shortening – are these services now too big a security risk to use? Computer World UK. <https://www.computerworlduk.com/security/apples-encryption-dilemma-explained-does-giving-fbi-access-matter-3635303/> (Accessed 21-01-2018). pages 12
- [25] Stefan Savage Geoffrey M. Voelker Justin Ma, Lawrence K. Saul. *Beyond Blacklists: Learning to Detect Malicious Web Sites from Suspicious URLs*. 2019. pages 12
- [26] PhishTank. <https://www.phishtank.com/> (Accessed 21-01-2018). pages 12, 25
- [27] Spamscatter. <https://www.phishtank.com/> (Accessed 21-01-2018). pages 12
- [28] Kun Li Tao Wei Zhenkai Liang Jian Mao, Pei Li. *BaitAlarm: Detecting Phishing Sites Using Similarity in Fundamental Visual Features*. 2013. pages 12
- [29] Scott Dick James Miller Teh-Chung Chen, Torin Stepan. *An Anti-Phishing System Employing Diffused Information*. 2014. pages 13
- [30] Samir Saklikar and Subir Saha. *Public Key-embedded Graphic CAPTCHAs*. 2008. pages 13
- [31] Debabrata Das Subir Saha Yogesh Joshi, Samir Saklikar. *PhishGuard: A browser plug-in for protection from phishing*. 2008. pages 13, 15
- [32] Haining Wang Chuan Yue. *Anti-Phishing in Offense and Defense*. 2008. pages 13
- [33] Alwyn R Pais Routhu Srinivasa Rao. *Detecting Phishing Websites using Automation of Human Behavior*. 2017. pages 14, 18, 22, 25, 54
- [34] XVFB Man page. <https://www.x.org/archive/current/doc/man/man1/Xvfb.1.xhtml> (Accessed 06-06-2018). pages 15
- [35] Getting Started with Headless Chrome. <https://developers.google.com/web/updates/2017/04/headless-chrome> (Accessed 21-01-2018). pages 16
- [36] MDN Docs Headless mode. https://developer.mozilla.org/en-US/Firefox/Headless_mode (Accessed 21-01-2018). pages 16
- [37] Browser Market Share Worldwide Dec 2017 Statcounter. <http://gs.statcounter.com/browser-market-share#monthly-201712-201712-bar> (Accessed 21-01-2018). pages 16
- [38] Vitaly Slobodin. *Stepping down as maintainer*. <https://groups.google.com/forum/#!topic/phantomjs/9aI5d-LDuNE> (Accessed 21-01-2018). pages 16
- [39] Selenium. <http://www.seleniumhq.org/> (Accessed 21-01-2018). pages 16
- [40] Puppeteer. <https://github.com/GoogleChrome/puppeteer/> (Accessed 21-01-2018). pages 16

- [41] Dursun Delen David Olson. *Advanced Data Mining Techniques*, Springer, 1st edition. 2008. pages 16
- [42] Jason Hong Serge Egelman, Lorrie Faith Cranor. *You've Been Warned: An Empirical Study of the Effectiveness of Web Browser Phishing Warnings*. 2008. pages 17
- [43] Browser Rankings Stat Counter. <http://gs.statcounter.com/> (Accessed 06-06-2018). pages 18
- [44] jsoup Java HTML parser. <https://jsoup.org/> (Accessed 06-06-2018). pages 22
- [45] Puppeteer sourcecode ExecutionContext.js Github. <https://github.com/GoogleChrome/puppeteer/blob/master/lib/ExecutionContext.js#L66> (Accessed 06-06-2018). pages 27
- [46] Simmer JS Github. <https://github.com/gmmorris/simmerjs> (Accessed 06-06-2018). pages 27
- [47] parse-domain Github. <https://github.com/peerigon/parse-domain> (Accessed 06-06-2018). pages 29, 37
- [48] Faker.js Github. <https://github.com/marak/Faker.js/> (Accessed 06-06-2018). pages 32
- [49] Google Console Help Soft 404s. <https://support.google.com/webmasters/answer/181708?hl=en> (Accessed 06-06-2018). pages 35
- [50] Allison Rhodes Cisco Umbrella Blog. *The Phish That Almost Duped Phishtank Almost*. <https://umbrella.cisco.com/blog/2011/09/28/the-phish-that-almost-duped-phishtank-almost/> (Accessed 06-06-2018). pages 36
- [51] Public Suffix List. <https://publicsuffix.org/list/> (Accessed 06-06-2018). pages 37
- [52] feat(BrowserContext): introduce Browser Contexts. #2523 Pull Request Merged May 2018 Github. <https://github.com/GoogleChrome/puppeteer/pull/2523> (Accessed 06-06-2018). pages 40
- [53] Headless: BrowserContext local storage isolation broken-Chromium bug tracker. <https://bugs.chromium.org/p/chromium/issues/detail?id=754576#c2> (Accessed 06-06-2018). pages 40
- [54] Amazon Web Services. <https://aws.amazon.com/> (Accessed 06-06-2018). pages 42
- [55] AWS Lambda. <https://aws.amazon.com/lambda/> (Accessed 06-06-2018). pages 43
- [56] AWS Simple Storage Service. <https://aws.amazon.com/s3/> (Accessed 06-06-2018). pages 43, 44
- [57] AWS DynamoDB. <https://aws.amazon.com/dynamodb/> (Accessed 06-06-2018). pages 43
- [58] AWS Simple Queue Service. <https://aws.amazon.com/sqs/> (Accessed 06-06-2018). pages 44
- [59] Digital Ocean Droplets. <https://www.digitalocean.com/products/droplets/> (Accessed 06-06-2018). pages 45
- [60] Docker. <https://www.docker.com/> (Accessed 06-06-2018). pages 45
- [61] Docker Swarm mode overview. <https://docs.docker.com/engine/swarm/> (Accessed 06-06-2018). pages 45
- [62] Portainer. <https://github.com/portainer/portainer> (Accessed 06-06-2018). pages 45
- [63] Proxy-Chain Programmable HTTP proxy server for Node.js. <https://github.com/apifytech/proxy-chain> (Accessed 06-06-2018). pages 45
- [64] Bootstrap. <https://getbootstrap.com/> (Accessed 06-06-2018). pages 48

[65] AWS Cloudfront. <https://aws.amazon.com/cloudfront/> (Accessed 06-06-2018). pages 48

[66] Sematex Docker Agent. <https://github.com/sematex/sematex-agent-docker> (Accessed 06-06-2018). pages 61

Icons in diagrams sourced from FlatIcon.com

Appendix A Target identification based on hyperlinks in the page

Target Brand	TP	TP rate	FP	FP Rate	FN	FN rate	Precision	Recall	Mistaken for
Amazon	516	0.8643	0	0	81	0.1357	1	0.8643	google: 58, skype: 1, googleurlshortener: 1, instagram: 2, github: 8, snipli: 3, facebook: 2, hostgator: 3, bild: 3 ,
Blockchain	346	0.9858	0	0	5	0.0142	1	0.9858	github: 4, instagram: 1
HM Revenue & Customs	31	0.6079	0	0	20	0.3922	1	0.6078	google: 16, github: 1, microsoft: 3
HSBC Europe	29	0.5370	0	0	25	0.4630	1	0.5370	google: 25
Netflix	1976	0.9564	0	0	90	0.0436	1	0.9564	github: 25, mailchimp: 7, instagram: 12, microsoft: 5, linkedin: 1, facebook: 9, wordpresscom: 6, uol: 3, twitter: 5, drexeluniversity: 2, verisign: 9, bloomberg: 1, betterbusinessbureau: 3, skype: 1, cpanel: 1
Visa	3	0.0857	0	0	32	0.9143	1	0.0857	skype: 4, spotify: 1, cardcomplete: 10, github: 11, land1: 1, twitter: 2, bitly: 1, facebook: 2

Appendix B Target identification based on requests from a page

Target Brand	TP	TP rate	FP	FP Rate	FN	FN rate	Precision	Recall	Mistaken for
Amazon	507	0.8492	0	0	9	0.1508	1	0.9826	snipli: 3, facebook: 2, google: 1, bild: 3
Blockchain	322	0.9174	0	0	0	0	1	1	
HM Revenue & Customs	17	0.3333	0	0	3	0.6667	1	0.8500	microsoft: 3
HSBC Europe	29	0.5370	0	0	0	0.4630	1	1	
Netflix	1908	0.9235	0	0	55	0.0765	1	0.9720	google: 15, facebook: 16, microsoft: 3, paypal: 3, wordpresscom: 1, uol: 3, verisign: 9, twitter: 1, cpanel: 1, archiveorg: 3
Visa	1	0.0286	0	0	12	0.9714	1	0.0769	spotify: 1, cardcomplete: 6, google: 3, facebook: 2