# Google Dorks: Analysis, Creation, and new Defenses

Flavio Toffalini[1], Maurizio Abbà[2], Damiano Carra[1], and Davide Balzarotti[3]

[1] University of Verona, Italy
damiano.carra@univr.it, flavio.toffalini@gmail.com
[2] LastLine, UK
mabba@lastline.com
[3] Eurecom, France
davide.balzarotti@eurecom.fr

**Abstract.** With the advent of Web 2.0, many users started to maintain personal web pages to show information about themselves, their businesses, or to run simple e-commerce applications. This transition has been facilitated by a large number of frameworks and applications that can be easily installed and customized. Unfortunately, attackers have taken advantage of the widespread use of these technologies – for example by crafting special search engines queries to fingerprint an application framework and automatically locate possible targets. This approach, usually called *Google Dorking*, is at the core of many automated exploitation bots.

In this paper we tackle this problem in three steps. We first perform a large-scale study of existing dorks, to understand their typology and the information attackers use to identify their target applications. We then propose a defense technique to render URL-based dorks ineffective. Finally we study the effectiveness of building dorks by using only combinations of generic words, and we propose a simple but effective way to protect web applications against this type of fingerprinting.

## 1 Introduction

In just few years from its first introduction, the Web rapidly evolved from a client-server system to deliver hypertext documents into a complex platform to run stateful, asynchronous, distributed applications. One of the main characteristics that contributed to the success of the Web is the fact that it was designed to help users to create their own content and maintain their own web pages.

This has been possible thanks to a set of tools and standard technologies that facilitate the development of web applications. These tools, often called *Web Application Frameworks*, range from general purpose solutions like Ruby on Rails, to specific applications like Wikis or Content Management Systems (CMS). Despite their undisputed impact, the widespread adoption of such technologies also introduced a number of security concerns. For example, a severe vulnerability identified in a given framework could be used to perform large-scale attacks to compromise all the web applications developed with that technology.

Therefore, from the attacker viewpoint, the information about the technology used to create a web application is extremely relevant.

In order to easily locate all the applications developed with a certain framework, attackers use so-called *Google Dork Queries* [1] (or simply *dorks*). Informally, a dork is a particular query string submitted to a search engine, crafted in a way to fingerprint not a particular piece of information (the typical goal of a search engine) but the core structure that a web site inherits from its underlying application framework. In the literature, different types of dorks have been used for different purposes, e.g., to automatically detect mis-configured web sites or to list online shopping sites that are built using a particular CMS.

The widespread adoption of frameworks on one side, and the ability to abuse search engines to fingerprint them on the other, had a very negative impact on web security. In fact, this combination lead to complete *automation*, with attackers running autonomous scout and exploitation bots, which scan the web for possible targets to attack with the corresponding exploit [2]. Therefore, we believe that a first important step towards securing web applications consists of breaking this automation. Researcher proposed software diversification [3] as a way to randomize applications and diversify the targets against possible attacks. However, automated diversification approaches require complex transformations to the application code, are not portable between different languages and technologies, often target only a particular class of vulnerabilities, and, to the best of our knowledge, have never been applied to web-based applications.

In this paper we present a different solution, in which a form of diversification is applied not to prevent the exploitation phase, but to prevent the attackers from fingerprinting vulnerable applications. We start our study by performing a systematic analysis of Google Dorks, to understand how they are created and which information they use to identify their targets. While other researchers have looked at the use of dorks in the wild [4], in this paper we study their characteristics and their effectiveness from the defendant viewpoint. We focus in particular on two classes of dorks, those based on portions of a website URL, and those based on a specific sequence of terms inside a web page. For the first class, we propose a general solutions – implemented in an Apache Module – in which we obfuscate the structure of the application showing to the search engine only the information that is relevant for content indexing. Our approach does not require any modification to the application, and it is designed to work together with existing search engine optimization techniques.

If we exclude the use of simple application banners, dorks based on generic word sequences are instead rarely used in practice. Therefore, as a first step we created a tool to measure if this type of dorks is feasible, and how accurate it is in fingerprinting popular CMSes. Our tests show that our technique is able to generate signatures with over 90% accuracy. Therefore, we also discuss possible countermeasures to prevent attackers from building these dorks, and we propose a novel technique to remove the sensitive framework-related words from search engines results without removing them from the page and without affecting the usability of the application.

To conclude, this paper makes the following contributions:

– We present the first comprehensive study of the mechanisms used by dorks and we improve the literature classification in order to understand the main issues and develop the best defenses.
– We design and implement a tool to block dorks based on URL information without changing the Web application and without affecting the site ranking in the search engines.
– We study dorks based on combinations of common words, and we implement a tool to automatically create them and evaluate their effectiveness. Our experiments demonstrate that it is possible to build a dork using non-trivial information left by the Web application framework.
– We propose a simple but effective countermeasure to prevent dorks based on common words, without removing them from the page.

Thanks to our techniques, we show that there are no more information available for an attacker to identify a web application framework based on the queries and the results displayed by a search engine.

## 2 Background and classification

The creation, deployment and maintenance of a website are complex tasks. In particular, if web developers employ modern CMSes, the set of files that compose a website contain much more information than the site content itself and such unintentional traces may be used to identify possible vulnerabilities that can be exploited by malicious users.

We identify two types of traces: (i) traces left by mistake that expose sensitive information on the Internet (e.g., due to misconfiguration of the used tool), and (ii) traces left by the Web Application Framework (WAF) in the core structure of the website. While the former type of traces is simple to detect and remove, the latter can be seen as a fingerprint of the WAF, which may not be easy to remove since it is part of the WAF itself.

There are many examples of traces left by mistake. For instance, log files related to the framework installation may be left in public directories (indexed by the search engines). Such log files may show important information related to the machine where the WAF is installed. The most common examples related to the fingerprint of a WAF are the application *banners*, such as "Powered by Wordpress", which contain the name of the tool used to create the website.

Google Dorks still lack a formal definition, but they are typically associated to queries that take advantage of advanced operators offered by search engines to retrieve a list of vulnerable systems or sensitive information. Unfortunately this common definition is vague (what type of sensitive information?) and inaccurate (e.g., not all dorks use advanced operators). Therefore, in this paper we adopt a more general definition of dorks: any query whose goal is to locate web sites using characteristics that are not based on the sites content but on their structure or type of resources. For example, a search query to locate all the e-commerce

applications with a particular login form is a dork, while a query to locate e-commerce applications that sell Nike shoes is not.

Dorks often use advance operators (such as `inurl` to search in a URL) to look for specific content in the different parts of the target web sites. Below, we show two examples of dorks, where the attacker looks for an installation log (left by mistake) or for a banner string (used to fingerprint a certain framework):

```
inurl:"installer-log.txt" AND intext:"DUPLICATOR INSTALL-LOG"
intext:"Powered by Wordpress"
```

Note that all search engine operators can only be used to search keywords that are *visible* to the end users. Any information buried in the HTML code, but not visible, cannot be searched. This is important, since it is often possible to recognize the tool that produced a web page by looking at the HTML code, an operation that however cannot be done with a traditional search engine.

Since there are many different types of information that can be retrieved from a search engine, there are many types of dorks that can be created. In the following, we revise the classification used so far in the literature.

## 2.1 Existing Dorks Classification

Previous works (for a complete review, please refer to Section 6) divide dorks into different categories, typically following the classification proposed in the Google Hacking Database (GHDB) [5, 6], which contains 14 categories. The criteria used to define these categories is the purpose of the dork, i.e., which type of information an attacker is trying to find. For instance, some of the categories are:

**Advisories and vulnerabilities:** it contains dorks that are able to locate various vulnerable servers, which are product or version-specific.

**Sensitive directories:** these dorks try to understand if some directories (with sensitive information) that should remain hidden, are made public.

**Files containing passwords:** these dorks try to locate files containing passwords.

**Pages containing login portals:** it contains dorks to locate login pages for various services; if such pages are vulnerable, they can be the starting point to obtain other information about the system.

**Error messages:** these dorks retrieve the pages or the files with errors messages that may contain some details about the system.

Different categories often rely on different techniques – such as the use of some advance operators or keywords – and target different parts of a website – such as its title, main body, files, or directories.

While this classification may provide some hints on the sensitive information a user should hide, the point of view is biased towards the attacker. From the defendant point of view, it would be useful to have a classification based on the techniques used to retrieve the information, so that it would be possible to check

if a website is robust against such techniques (independently from the aim for which the technique is used). For this reason, in this paper we adopt a different classification based on the characteristics of the dorks.

## 2.2 Alternative classification

We implemented a crawler to download all the entries in the GHDB [5, 6] and a set of tools to normalize each dork and automatically classify it based on the information it uses[4].

We have identified three main categories, which are not necessarily disjoint and may be combined together in a single query:

**URL Patterns:** This category contains the dorks that use information present in the structure of the URL.

**Extensions:** It contains the dorks used to search files with a specific extension, typically to locate misconfigured pages.

**Content-Based:** These dorks use combination of words in the content of the page – both in the body, and in the title.

Since the content-based category is wide, we subsequently split such category into four sub-categories:

**Application Banners:** This category contains strings or sentences that identify the underlying WAF (e.g., "*Powered by Wordpress*"). These banners can be found in the body of the page (often in the foothold) or in the title.

**Misconfiguration Strings:** This category contains strings which correspond to sensitive information left accessible by mistake by human faults (such as database logs, string present in configuration files, or part of the default installation pages)

**Errors Strings:** Dorks in this category use special strings to locate unhandled errors, such as the ones returned when a server-side script is not able to read a file or it processes wrong parameters. Usually, besides the error, it is also possible to find on the page extra data about the server-side program, or other general information about the system.

**Common Words:** This class contains the dorks that do not fit in the other categories. They are based on combinations of common words that are not related to a particular application. For instance, these dorks may search for ("*insert*", "*username*", and "*help*") to locate a particular login page.

Table 2.2 shows the number of dorks for each category. Since some of the dorks belongs to different categories, the sum of all categories is greater than the total number of entries. The classification shows that most of the dorks are based on banners and URL patterns. In particular, 89.5% of the existing dorks use either a URL or a banner in their query.

---

[4] Not all dorks have been correctly classified automatically, so we manually inspected the results to ensure a correct classification.

| Category | | Number | Perc. (%) |
|---|---|---|---|
| URL Pattern | | 2267 | 44 |
| Extensions | | 318 | 6 |
| Content-based | Banners | 2760 | 54 |
| | Misconfigurations | 414 | 8 |
| | Errors | 71 | 1 |
| | Common words | 587 | 11 |
| Total entries in GHDB [6] | | 5143 | |

**Table 1.** Number of dorks and relative percentage for the different categories. Since a dork may belong to different categories, the sum of the entries of all categories is greater than the total number of entries extracted from GHDB.
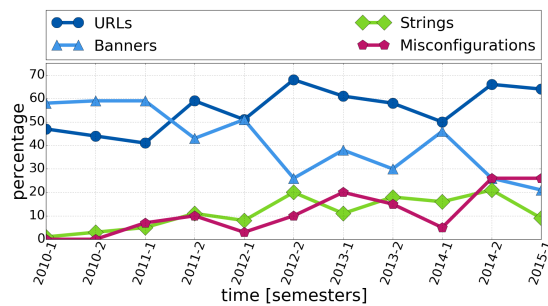


**Fig. 1.** Dorks evolution by category.

Besides the absolute number of dorks, it is interesting to study the evolution of the dork categories over time. This is possible since the data from GHDB [6] contains the date in which the dork was added to the database. Figure 1 shows the percentage over time of the proposed dorks, grouped by category. It is interesting to note that banner-based dorks are less and less used in the wild, probably as a consequence of users removing those strings from their application. In fact, their popularity decreased from almost 60% in 2010 to around 20% in 2015 – leaving URL-based dorks to completely dominate the field.

### 2.3 Existing Defenses

Since the classification of the dorks has traditionally taken the attacker viewpoint, there are few works that provide practical information about possible defenses. Most of the them only suggests some best practices (e.g., remove all sensitive information), without describing any specific action. Unfortunately, some of these best practice are not compatible with Search Engine Optimizations (SEOs). SEOs are a set of techniques used to improve the webpage rank – e.g., by including relevant keywords in the URL, in the title, or in the page headers. When removing a content, one should avoid to affect such SEOs.

As previously noted, most of the dorks are based on banners and URL patterns, with mis-configuration strings at the third place. While this last category

is a consequence of human faults, which are somehow easier to detect, the other dorks are all based on the fingerprint of the WAFs.

Banners are actually simple to remove, but the URL patterns are considerably more complex to handle. In fact, the URL structure is inherited from the underlying framework, and therefore one should modify the core structure of the WAF itself – a task too complex and error prone for the majority of the users. Finally, word-based dorks are even harder to handle because it is not obvious which innocuous words can be used to precisely identify a web application.

In both cases we need effective countermeasures that are able to neutralize such dorks. In the next sections, we show our solutions to these issues.

## 3   Defeating URL-based Dorks

The URLs of a web application can contain two types of information. The first is part of the structure of the web application framework, such as the name of sub-directories, and the presence of default administration or login pages. The second is part of the website content, such as the title of an article or the name of a product (that can also be automatically generated by specific SEO optimization plugins). While the second part is what a search engine should capture and index, we argue that there is no reason for search engines to also maintain information about the first one.

The optimal solution to avoid this problem would be to apply a set of random transformations to the structure of the web application framework. However, the diversity and complexity of these frameworks would require to develop an ad-hoc solution for each of them. To avoid this problem, we implement the transformation as a filter in the web server. To be usable in practice, this approach needs to satisfy some constraints. In particular, we need a technique that:

1. It is independent from the programming language and the WAF used to develop the web site.
2. It is easily deployable on an existing web application, without the need to modify the source code.
3. It supports dynamically generated URLs, both on the server side and on the client side (e.g., through Javascript).
4. It can co-exist with SEO plugins or other URL-rewriting components.

The basic idea of our solution is to obfuscate (part of) the URLs using a random string generated at installation time. Note that the string needs to be random but it does not need to be secret, as its only role is to prevent an attacker for computing a single URL that matches all the applications of a give type accessible on the Web.

Our solution relies on two components: first, it uses standard SEO techniques to force search engines to only index obfuscated URLs, and then applies a filter installed in the web server to de-obfuscate the URLs in the incoming requests.

### 3.1 URL Obfuscation

The obfuscation works simply by XOR-ing part of the original URL with the random seed. Our technique can be used in two different ways: for selective-protection or for global protection. In the first mode, it obfuscates only particular pieces of URLs that are specified as regular expressions in a configuration file. This can be used to selectively protect against known dorks, for instance based on particular parameters or directory names.

When our solution is configured for global protection, it instead obfuscate all the URLs, except for possible substrings specified by regular expressions. This mode provides a better protection and simplifies the deployment. It can also co-exist with other SEO plugins, by simply white-listing the portions of URLs used by them (for example, all the URLs under `/blog/posts/*`). The advantage of this solution is that it can be used out-of-the-box to protect the vast majority of small websites based on popular CMSs. But it can also be used, by properly configuring the set of regular expressions, to protect more complex websites that have specific needs and non-standard URL schemes.

Finally, the user can choose to apply the obfuscation filter only to particular `UserAgent` strings. Since the goal is to prevent popular search engines from indexing the original URLs, the entire solution only needs to be applied to the requests coming from their crawlers. As we discuss in the next session, our technique works also if applied to all incoming requests, but this would incur a performance penalty for large websites. Therefore, by default our deployment only obfuscates the URLs provided to a configurable list of search engines[5].

### 3.2 Delivering Obfuscated URLs

In this section, we explain our strategy to show obfuscated URLs, and hide the original ones, in the results of search engines. The idea is to influence the behavior of the crawlers by using common SEO techniques.

**Redirect 301** The Redirect 301 is a status code of the HTTP protocol used for permanent redirection. As the name suggests, it is used when a page changes its URL, in combination with a "`Location`" header to specify the new URL to follow. When the user-agent of a search engine sends a request for a cleartext URL, our filter returns a 301 error with a pointer to the obfuscated URL.

The advantage of this technique is that it relies on a standard error code which is supported by the all the search engines we tested. Another advantage of this approach is that the search engines move the current page rank over to the target of the redirection. Unfortunately, using the 301 technique alone is not sufficient to protect a page, as some search engines (Google for instance) would store in their database both the cleartext and the obfuscated URL.

**Canonical URL Tag** The Canonical URL Tag is a meta-tag mainly used in the header of the HTML documents. It is also possible to use this tag as HTTP

---

[5] Here we assume that search engines do not try to disguise their requests, as it is the case for all the popular ones we encountered in our study

header to manage non-HTML documents, such as PDF files and images. Its main purpose is to tell search engines what is the real URL to show in their results.

For instance, consider two pages that show the same data, but generated with a different sorting parameter, as follow:

```
http://www.abc.com/order-list.php?orderby=data&direct=asc
http://www.abc.com/order-list.php?orderby=cat&direct=desc
```

In the example above, the information is the same but the two pages risk to be indexed as two different entries. The Canonical tag allows the site owner to show them as a single entry, improving the page rank. It is also important that there is only a single tag in the page, as if more tags are presents search engines would ignore them all.

Our filter parses the document, and it injects a Canonical URL Tag with the obfuscated URL. To avoid conflict with other Canonical URL Tags, we detect their presence and replace their value with the corresponding obfuscated version.

A drawback of this solution is that the Canonical URL Tag needs to contain a URL already present in the index of the search engine. If the URL is not indexed, the search engine ignores the tag. This is the reason why we use this technique in conjunction with the 301 redirection.

**Site Map** The site map is an XML document that contains all the public links of a web site. The crawler uses this document to get the entire list of the URLs to visit. For instance, this document is used in blogs to inform the search engine about the existence of new entries, as for the search engine it is more efficient to poll a single document rather than crawling the entire site each time.

If a search engine tries to get a site map, our filter replaces all the URLs with their obfuscated versions. This is another technique to inform the crawler about the site structure and populate its cache with the obfuscated URLs.

**Obfuscation Protocol** In this section, we show how the previous techniques are combined together to obtain our goal. Figure 2 shows the behavior of our tool when a crawler visits a protected web site. When the crawler requests a resource 'a' our tool intercepts the request and redirect it to the obfuscated URL `O(a)`. The crawler then follows the redirect and requests the obfuscated resource. In this case, the system de-obfuscates the request, and then serves it according to the logic of the web site. When the application returns the result page, our filter adds the Canonical URL Tag following the rules described previously.

In Fig.2, we also show how the tool behaves when normal users visit the web site. Typically, users would first request an obfuscated URL (as returned by a query to a search engine, for example). In this case, the request is de-obfuscated and forwarded to the web application as explained before. This action incurs a small penalty in the time required to serve the requests. However, once the user gets the page back, he can interact with the websites following links and/or forms that contain un-obfuscated URLs. In this case, the requests are served by the web server without any additional computation or delay.

Even if this approach might appear as a form of *cloaking*, the clocking definition requires an application to return different resources for a crawler and for
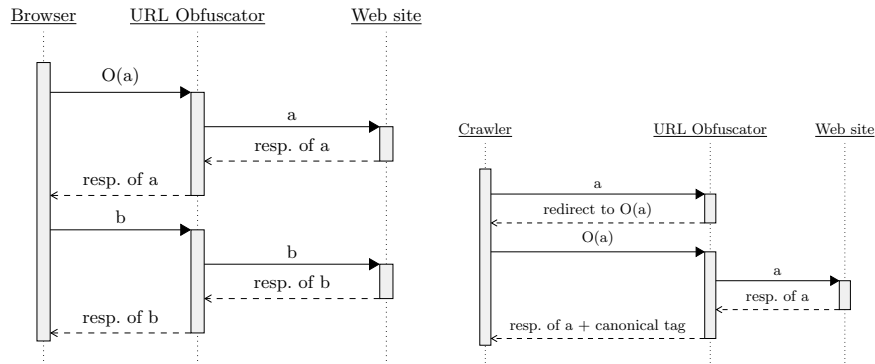
**Fig. 2.** On the left side: messages exchanged between a protected application ad a normal user. On the right side: messages exchanged between a protected application and a search engine crawler.

other clients, as described in the guidelines of the major search engines [7–9]. Our technique only adds a meta-tag to the page, and does not modify the rest of the content and its keywords.

### 3.3 Implementation

Our approach is implemented as a module for the Apache web server. When a web site returns some content, Apache handles the data using the so-called *buckets* and *brigades*. Basically, a bucket is a generic container for any kind of data, such as a HTML page, a PDF document, or an image. In a bucket, data is simply organized in an array of bytes. A brigade is a linked list of buckets. The Apache APIs allow to split a bucket and re-link them to the corresponding brigade. Using this approach, it is possible to replace, remove, or append bytes to a resource, without re-allocating space. We use this technique to insert the Canonical URL Tag in the response, and to modify the site map. In addition, the APIs also permit to manage the header of the HTTP response, and our tool use this feature in order to add the Canonical URL Tag in the headers.

Since the Apache server typically hosts several modules simultaneously, we have configured our plugin to be the last, to ensure that the obfuscation is applied after any other URL transformation or rewriting step. Our obfuscation module is also designed to work in combination with the `deflate` module. In particular, it preserves the compression for normal users but it temporarily deactivate the module for requests performed by search engine bots. The reason is that a client can request a compressed resource, but in this case our module is not able to parse the compressed output to insert the Canonical Tag or to obfuscate the URLs in the site-map. Therefore, our solution removes the output compression from the search engine requests – but still allows compressed responses in all other cases.

Finally, to simplify the deployment of our module, we developed an installation tool that takes as input a web site to protect, generate the random seed,

analyzes the site URL schema to create the list of exception URLs, and generate the corresponding snippet to insert into the Apache configuration file. This is sufficient to handle all simple CMS installations, but the user can customize the automatically generated configuration to accommodate more complex scenarios.

### 3.4 Experiments and results

We tested our solution on Apache 2.4.10 running two popular CSMs: Joomla! 3.4, and Wordpress 4.2.5. We checked that our websites could be easily identified using dorks based on "`inurl:component/user`" and "`inurl:wp-content`".

We then protected the websites with our module and verified that a number of popular search engines (Google, Bing, AOL, Yandex, and Rambler) were only able to index the obfuscated URLs and therefore our web sites were no longer discoverable using URL-based dorks.

Finally, during our experiments we also traced the number of requests we received from search engines. Since the average number was 100 access per day, we believe that our solution does not have any measurable impact on the performance of the server or on the network traffic.

## 4 Word-based Dorks

As we already observed in the Section 2, dorks based on application banners are rapidly decreasing in popularity, probably because users started removing these banners from their web applications. Therefore, it is reasonable to wonder if is also possible to create a precise fingerprint of an application by using only a set of generic and seemingly unrelated words.

This section is devoted to this topic. In the first part we show that it is indeed possible to automatically build word-based dorks for different content management systems. Such dorks may be extremely dangerous because the queries submitted to the search engines are difficult to detect as dorks (since they do not use any advanced operator or any string clearly related to the target CMS). In the second part, we discuss possible countermeasures for this type of dorks.

### 4.1 Dork Creation

Given a set of words used by a CMS, the search for the optimal combination that can be used as fingerprint has clearly an exponential complexity. Therefore, we need to adopt a set of heuristics to speed up the generation process. Before introducing our technique we need to identify the set of words to analyze, and the criteria used to evaluate such words.

**Building Blocks** The first step to build a dork is to extract the set of words that may characterize the CMS. To this aim, we start from a vanilla installation of the target website framework, without any modification or personalization. From this clean instance, we remove the default *lorem ipsum* content, such as "Hello world" or "My first post". Then, using a custom crawler, our tool extracts all the visible words from all the pages of the web site, i.e., the words that are

actually displayed by a browser and that are therefore indexed by search engines. After removing common stop words, usually discarded also by the search engines (e.g., *and, as, at, . . .*), our crawler groups the remaining words by page and also maintains a list with all the words encountered so far.

In order to build an automatic tool that creates word-based dorks for the different CMSes, we need two additional building blocks: (i) a set of APIs to interrogate a search engine, and (ii) an oracle that is able to understand if a website has been created with a specific CMS.

As for the APIs, we make use of the Bing APIs in order to submit a query to the Bing search engine. Clearly, any other search engine would be equivalent: we have chosen Bing since it has less restrictions in terms of the number of queries per day that a user can make. Given a query, Bing provides the total number of entries found for that query: this value represents the **coverage** of a query. Among these entries, our tool retrieve the first 1000 results. For each of these pages, we use the Wappalyzer-python library [10] to confirm whether the page is built using the target CMS. Wappalyzer looks at the HTML code of the page and tries to understand if there are traces left by a given CMS: this is fundamentally different from looking at the visible words, because to take a decision the tool needs to process the HTML content of the web page that is not indexed by the traditional search engines. Using this technique, we compute the **hit rank**, i.e., the number of results that are actually built with a given CMS divided by the number of results obtained by the query[6].

To build a dork, we need to evaluate its *precision* during the building process: the precision is a combination of the coverage and the hit rank, i.e., a good dork is the one that obtains the largest number of results with the highest accuracy.

**Dork Generation** The basic idea used in our prototype is to build the dork step by step, adding one word at a time in a sort of gradient ascent algorithm. The first observation is that when a new word is added to an existing dork, its coverage can only decrease, but the hit rank may increase or decrease. As an example, in Figure 3 we show the impact of adding a new word $w_i$ while building a dork (in this case, the initial dork contained three words, with a hit rank equal to 30%). For each word $w_i$ we measure the new coverage and the new hit rank of the whole dork (three words with the addition of $w_i$), and we order the results according to the hit rank. As we can see, half of the new words decreases the hit rank, and therefore can be discarded from our process. Words that result in a higher hit rank usually considerably decrease the coverage – i.e., they return very few results. The goal is to find the best compromise, where we still retain a sufficiently high coverage while we increase the hit rank.

Our solution is to compute at each step the median coverage of all the candidate words for which the hit rank increases – shown in the figure as an horizontal dashed line at 16.6 M; we then choose the word that provides the highest hit rank *and* a coverage above the median – in this case, the word "posts".

---

[6] For efficiency reasons, we compute the hit rank by visiting a random sample that covers 30% of the first 1000 results.
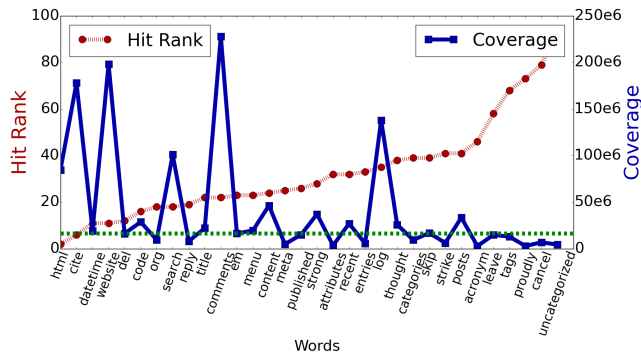
**Fig. 3.** Evolution of hit rank and coverage for a dork while adding different words.

The complete algorithm is shown in Algorithm 1. One of the inputs to the procedure is an initial set of words $\mathcal{D}$ for a given page, i.e., a basic dork to which new words should be added. Ideally we should start from an empty dork, and then we should evaluate the first word to add. However, since the coverage and the hit rank of a single word may be extremely variable, we decided to start from an initial dork of at least three words, so that to obtain meaningful values for the coverage and the hit rank.

As initial dork, we have chosen the top three words with the highest coverage (singularly) and that would provide a hit rank higher than 30% (together). While the choice of the initial point may seem critical, we have tested different combinations of words obtaining similar results (in terms of final coverage and hit rank). However, it is important to stress the fact that our goal is not to find the best dork, but to find at least one that can be used to find websites created with a specific CMS. In other words, our algorithm can return a local optimum solution that changes depending on the initial point. However, any dork that provides results in the same order of magnitude of other classes of dorks (such as URL or banner-based) is satisfactory for our study.

The other input of the procedure is the set of words $\mathcal{V}$ that has been extracted from the vanilla instance of the CMS, without the words used in the starting dork, i.e., $\mathcal{V}' = \mathcal{V} \setminus \mathcal{D}$. Finally, we need to specify which CMS should be used to compute the hit rank.

The algorithm keeps adding words to the dork until the final hit rank is greater than or equal to 90% or there are no more words to add. For each word in $\mathcal{V}'$, it computes the new hit rank and the new coverage, and it stores the entry in a table only if the word improves the hit rank (line 10).

If none of the words are able to improve the hit rank (line 14), the algorithm stops and returns the current dork. Otherwise, the algorithm computes the median coverage which is used as a reference to obtain the best word. The best word is the word with the highest hit rank among the ones with a coverage above the median. Finally, the best word is added to the dork and removed from $\mathcal{V}'$.

---

**Algorithm 1.** Our algorithm to create a word-based dork

---
```
 1:  procedure GETDORK(D,V′,CMS)
 2:      url_list ← apiBing.search(D)                              ▷ retrive a list of URL given D
 3:      max_hr ← calcHitRank(url_list, CMS)                       ▷ calculate hit rank from URL List
 4:      while max_hr < 90% ∧ V′ ≠ ∅ do
 5:          table ← empty()
 6:          for all w ∈ V′ do
 7:              cov ← calcCoverage(D ∪ w)
 8:              url_list ← api_bing.search(D ∪ w)
 9:              hr ← calcHitRank(url_list, CMS)
10:              if hr > max_hr then
11:                  table ←_row (w, hr, cov)
12:              end if
13:          end for
14:          if table == ∅ then
15:              return D                                          ▷ final dork
16:          end if
17:          median ← calcMedian(table)
18:          (best_word, hr) ← getBestWord(table, median)
19:          D ← D ∪ best_word
20:          max_hr ← hr
21:          V′ ← V′ \ {best_word}
22:      end while
23:      return D                                                  ▷ final dork
24:  end procedure
```
---

**Experiments and Results** In order to test our solution, we consider five well known Web Application Frameworks: three general purpose CMSes (Wordpress, Joomla!, and Drupal) and two E-Commerce CMSes (Magento and OpenCart). We run the tests on a machine with Ubuntu 15.10, Python 3.4, BeautifulSoup, and Wappalyzer-python.

For each CMS, we have created dorks starting from two different installations:

– **Vanilla**: we consider the basic out-of-the-box installation, with no changes to the default website obtained from the CMS;
– **Theme**: we add some personalization to the website, such as changing the basic graphical theme.

For two CMSes, Drupal and Opencart, the lists of words extracted with our crawler from the two installations (Vanilla and Theme) are the same, therefore the dorks obtained from the Vanilla and Theme installations are the same too.

We compare the results of the dorks created with our tool with the banner-based dorks. Table 4.1 shows, for each CMS, the hit rank and the coverage for the two dorks (derived from the Vanilla and Theme installations), as well as for the dork taken as a reference.

The results show that our dorks obtain a coverage with the same order of magnitude of the reference dork, with similar hit rank, i.e., they are as effective as banner-based dorks in finding targeted CMSes. It is interesting to note also that the differences between the Vanilla and the Theme dorks are small, suggesting that minor customizations of the website have little impact on our methodology.

**Customized Websites** While a little customization have a small impact on the effectiveness of the dorks we created, it is interesting to understand if, instead,

|  | Vanilla | Theme | Reference |  |
|---|---|---|---|---|
| **Wordpress** | 93.8% | 74.1% | 96.7% | hits |
|  | 47.1 M | 22 M | 83.6 M | cover. |
| **Joomla** | 87.8% | 75.6% | 88.7% | hits |
|  | 7.24 M | 1.44 M | 3.73 M | cover. |
| **Drupal** | 82.7% | 82.7% | 99.7% | hits |
|  | 7.87 M | 7.87 M | 3.27 M | cover. |
| **Magento** | 87.1% | 93.2% | 85.2% | hits |
|  | 0.39 M | 0.22 M | 0.68 M | cover. |
| **OpenCart** | 89.1% | 89.1% | 99.8% | hits |
|  | 0.59 M | 0.59 M | 1.42 M | cover. |

**Table 2.** Hit rank and coverage of the dorks created with our *GetDork* tool, compared with a reference banner-based dork. For each CMS, we consider the dorks derived from two installations, Vanilla and Theme.
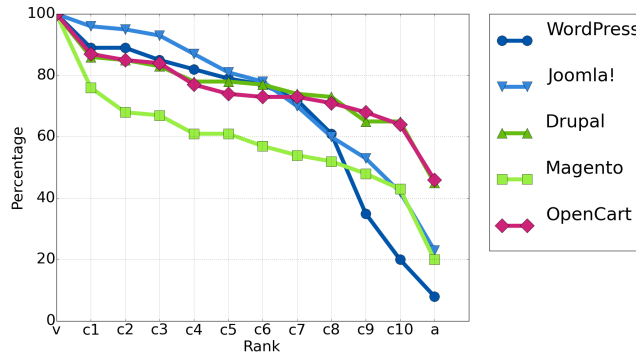


**Fig. 4.** Graph of common words for CMSs

major modifications may make our methodology ineffective. In other words, we investigate if customization can compromise the fingerprint left by the CMS, and implicitly be a countermeasure to word-based dorks.

To this aim, we selected and analyzed a set of popular websites built with known CMSes but largely customized, such as www.toyota.com, www.linux.com, and www.peugeot.com. For each CMS, we collected ten websites and extracted the list of words with our crawler. We then compared these lists with the corresponding lists extracted from the vanilla instances. Figure 4 shows the percentage of common words that each customized website has with the vanilla instance – for each CMS we have ordered the websites according to this percentage, therefore the x-axis shows the ranking, and not a specific website.

The point where the x-axis is labeled with "v" represents the vanilla instances, while the point "a" represents the intersection of all custom websites for that CMS with the vanilla instance. These points indicate the percentage of words that are common to all the websites, including the vanilla, and therefore

represent the starting point for the creation of a dork. Except for Wordpress, most of the customized websites have high percentage of common words with the vanilla instance. Nevertheless, finding a common subset of words is not easy. We have actually tried to build a dork starting from the intersection of the sets of words of all the customized website and the vanilla instance, and we were successful only for Drupal and Opencart. This means that large customizations may be indeed a countermeasure for word-based dorks, but not for all CMSes. It is also important to note that such high customization is typical of a limited number of websites that are managed by large organizations, therefore the probability that such websites are vulnerable due to limited maintenance is not high.

## 4.2 Defense against Word-based Dorks

In the previous sections, we discuss an alternative method to create dorks using a combination of common words. While slightly less effective than banner-based dorks, we were able to achieve a relevant combination of hit rank and coverage, showing that this technique can be used by criminals to locate their victims.

To protect against this threat, we propose a simple solution in which we insert invisible characters into the application framework keywords. Luckily, in the Unicode standard there is a set of empty special characters that are not rendered in web browser. Thus, the appearance of the web sites does not change but a search engine would index the keywords including these invisible characters, preventing an attacker from finding these keywords in her queries. This technique also allows the obfuscation of the application banners without removing them.

Moreover, this technique does not influence the search engine optimization and ranking, because the obfuscated keywords are only part of the template, and not of the website content. In our prototype we use the *Invisible Separator* character with code "U+2063" in the Unicode standard. As the name suggests, it is a separator between two characters that does not take any physical space and it is used in mathematical applications to formally separate two indexes (e.g., $ij$). It is possible to insert this character in the HTML page using the code "&#x2063;". For instance, an HTML code for a banner like "Powered by Wordpress" can be automatically modified to:

```
<div class="site-info">
    <a href="https://wordpress.org/">
        Power&#x2063;ed by Wor&#x2063;dpress
    </a>
</div>
```

The characters are ignored by search engines, effectively breaking each keyword in a random combination of sub-words. Obviously, this technique only works if a random number of invisible characters are placed at random locations inside each template keyword, so that each web application would have a different footprint. The highest combination that can be obtained using this technique is on the order of magnitude of $O(2^n)$, where $n$ is the sum of entire characters of all the words that can be used to create a signature.

To test the effectiveness of this solution, we created a test web site containing two identical pages – one of which uses our system to obfuscate its content using the invisible separator character. We then performed queries on Google, Bing, AOL, and Rambler and we were able to confirm that all of them properly indexed the content of web site. However, while it was possible to find the website by searching for its cleartext content, we were not able to locate the obfuscated page by querying for its keywords.

## 5    Discussion

When a new vulnerability is disclosed, attackers rely on the ability to quickly locate possible targets. Nowadays, this is usually done by exploiting search engines with specially crafted queries called dorks. In Section 2.2 we showed that there are three main ways to fingerprint an application: using banner strings, using URL patterns, or using combination of words. The first type is easy to prevent, and in fact many popular websites are not identifiable in this way. This is also confirmed by our longitudinal study, which shows that the percentage of dorks belonging to this category is steadily decreasing over time.

In this paper we show that it is also possible to prevent the other two classes of dorks, without affecting the code or the rank of a web application. We believe this is a very important result, as it undermines one of the main pillar of automated web attacks. If criminals were unable to use dorks to locate their target, they would need to find workarounds that, however, either increase the cost or reduce the coverage of their operations.

The only way left to fingerprint an application is by looking at its HTML code. To do that, attacker needs to switch to special search engines that also index the raw content of each web pages. However, these tools (such as *Meanpath* [11]) have a much smaller coverage compared to traditional search engines, and typically require the user to pay a registration to get the complete list of results. Either way, this would slow down and reduce the number of attacks. Another possible venue for criminals would be to implement their own crawlers, using special tools that can automatically identify popular CMS and web application frameworks (such as Blind Elephant [12] and WhatWeb [13]). However, this requires a non negligible infrastructure and amount of time, so again it would considerably raise the bar – successfully preventing most of the attacks that are now compromising million of web sites.

## 6    Related Work

Google hacking has been the subject of several recent studies. Most of them discuss tricks and techniques to manually build dorks, and only few propose limited defenses, statistics, or classification schemes.

Moore and Clayton [14] studied the logs of compromised websites, and identified three classes of "evil queries": the ones looking for vulnerabilities, the ones looking for already compromised websites, and the ones looking for web shells.

SearchAudit [15] is a technique used to recognize malicious queries from the logs of a search engine. The study provides an interesting perspective on dorks

that are used in the wild. In particular, the authors identify three classes of queries: to detect vulnerable web sites, to identify forums for spamming, and a special category used in Windows Live Messenger phishing attacks. While the first category is predominantly dominated by URL-based and banner-based dorks, the forum identification also included common strings such as "Be the first to comment this article". John et al, [16] then use SearchAudit to implement an adaptive Honeypot that changes its content in order to attract malicious users and to study their behaviors. The purpose of the study is to gain information about the attackers, and not to propose countermeasures.

Two books discuss Google Hacking: "*Google Hacking for Penetration Tester*" [1] and "*Hacking: The Next Generation*" [17]. Both of them show the techniques required to build dorks [18,19] using banner strings or URL patterns. These books adopt the same classification proposed by Johnny Long [5] and `exploit-db.com` [6]. As we already discussed in Section 2, this classification is focused on the goal of the dork (e.g., detect a vulnerable web sites, the presence of sensitive directories, or a mis-configuration error), while, in our work, we propose a classification based on the information that are used to build the fingerprint. Moreover, the defenses proposed in these books, as well as the ones proposed by L. Lancor [20], only discuss simple best practices – such as removing the name of the web application framework from the HTML footer.

Zhang et al., [4] study the type of vulnerabilities searched by the dorks (such as SQL-injection, XSS, or CSRF), and they compared them with the corresponding CVE. The authors also study the relation between dorks and advanced operators, but without considering the countermeasures as we do in this paper. Pelizzi at al. [21] propose a technique to create URL-based dorks to automatically look-up web sites affected by XSS vulnerabilities. Other works, such as Invernizzi et al. [22], and Zhang et al. [23], propose different techniques to create word dorks to identify malicious web sites. Their aim is to enlarge a database of compromised pages and not to find a fingerprint for a target web application framework. They create word dorks only with the common content of the infected pages without discussing how to improve the quality of the results.

Billing et al. [24] propose a tool to tests a list of provided dorks to find the ones that match a given web site. Similarly, several tools exist to audit a target site using public dorks databases, such as GooScan [25], Tracking Dog [26], or Diggity [27]. Sahito et al. [28] show how dorks can be used to retrieve private information from the Web (e.g., credit card numbers, private addresses, and telephone numbers). Similarly, Tath et al. [29,30] show how to use Google hacking in order to find hashed password, private keys or private information.

The literature also includes works not strictly related to dorks, but that deal with the problem of similarity of Web pages. For example, Soska et al. [31] show a correlation between attacked Web sites and future victims. The authors use comparison techniques (which include DOM features, common words, and URL patterns) in order to demonstrate that Web pages with similar features of compromised ones have high probability to be attacked too. Vasel et al. [32] use a database of compromised Web sites to calculate a risk-factor of future victims.

They also seek common features of Web pages as Soska. Although their aims is different from ours, it could be intriguing to use their approach to improve our algorithm to create word-based dorks.

Finally, some studies discuss dorks as a propagation vector for malware. For example, Cho et al., [33] show a technique to retrieve C&C botnet servers using Google Hacking. Provos et al., [28] and Yu et al, [34] analyze a set of worms able to find other vulnerable machines using dorks. In these papers, the authors propose a system to block the malicious queries in order to stop the worm propagation.

## 7    Conclusion

In this paper we presented the first study about the creation, classification, and accuracy of different categories of Google dorks. We started by improving previous classifications by performing an analysis of a large database of dorks used in the wild. Our measurements showed that most of the dorks are based on URL patterns or banner strings, with the last category in constant declining. Therefore, we proposed a novel technique to randomize parts of the website URLs, in order to hide information that can be used as fingerprint of the underlying WAF. Our tool, implemented as a module for the Apache web server, does not require any modification to the sources of the WAF, and it does not decrease the rank of the web site.

We then showed how it is possible to combine common words in a CMS template to build a signature of a web application framework, with an accuracy and a coverage comparable to URL-based dorks. We implemented a tool to build these signatures and tested it on five popular CMS applications. Finally, we proposed a new technique to prevent this kind of dorks. The idea is inject invisible Unicode characters in the template keywords, which does alter the web site appearance or its usability.

## References

1. J. Long and E. Skoudis, *Google Hacking for Penetration Testers*. Syngress, 2005.
2. N. Provos, J. McClain, and K. Wang, "Search worms," in *Proc. of the 4th ACM Workshop on Recurring Malcode*, pp. 1–8, 2006.
3. M. Christodorescu, M. Fredrikson, S. Jha, and J. Giffin, "End-to-end software diversification of internet services," in *Moving Target Defense*, vol. 54, pp. 117–130, 2011.
4. J. Zhang, J. Notani, and G. Gu, "Characterizing google hacking: A first large-scale quantitative study," in *Proc. of SecureComm*, September 2014.
5. "Johnny google hacking database." `http://johnny.ihackstuff.com/ghdb/`.
6. "Exploit database." `https://www.exploit-db.com/`.
7. "Yandex cloacking condition." `https://yandex.com/support/webmaster/yandex-indexing/webmaster-advice.xml`.
8. "Baidu cloacking condition." `http://baike.baidu.com/item/Cloaking`.
9. "Google cloacking condition." `https://support.google.com/webmasters/answer/66355?hl=en`.
10. "Wappalyzer-python." `https://github.com/scrapinghub/wappalyzer-python`.

11. "meanpath." `https://meanpath.com/`.
12. "Blind elephant." `https://community.qualys.com/community/blindelephant`.
13. "Whatweb." `http://www.morningstarsecurity.com/research/whatweb`.
14. T. Moore and R. Clayton, "Evil searching: Compromise and recompromise of internet hosts for phishing," in *Financial Cryptography and Data Security*, vol. 5628, pp. 256–272, 2009.
15. J. P. John, F. Yu, Y. Xie, M. Abadi, and A. Krishnamurthy, "Searching the searchers with searchaudit," in *Proceedings of the 19th USENIX Conference on Security*, (Berkeley, CA, USA), pp. 9–9, 2010.
16. J. P. John, F. Yu, Y. Xie, A. Krishnamurthy, and M. Abadi, "Heat-seeking honeypots: design and experience.," in *Proc. of WWW*, pp. 207–216, 2011.
17. K. Michael, *Hacking: The Next Generation*. Elsevier Advanced Technology, 2012.
18. "Google advanced operators." `https://support.google.com/websearch/answer/2466433?hl=en`.
19. "Bing advanced operators." `https://msdn.microsoft.com/en-us/library/ff795667.aspx`.
20. L. Lancor and R. Workman, "Using google hacking to enhance defense strategies," in *Proc. of the 38th SIGCSE Technical Symposium on Computer Science Education*, pp. 491–495, 2007.
21. R. Pelizzi, T. Tran, and A. Saberi, "Large-scale, automatic xss detection using google dorks," 2011.
22. L. Invernizzi, P. M. Comparetti, S. Benvenuti, C. Kruegel, M. Cova, and G. Vigna, "Evilseed: A guided approach to finding malicious web pages," in *IEEE Symposium on Security and Privacy*, pp. 428–442, 2012.
23. J. Zhang, C. Yang, Z. Xu, and G. Gu, "Poisonamplifier: a guided approach of discovering compromised websites through reversing search poisoning attacks," in *Research in Attacks, Intrusions, and Defenses*, pp. 230–253, 2012.
24. J. Billig, Y. Danilchenko, and C. E. Frank, "Evaluation of google hacking," in *Proc. of the 5th annual conference on Information security curriculum development*, pp. 27–32, ACM, 2008.
25. "Gooscan." `http://www.aldeid.com/wiki/Gooscan`.
26. M. Keßler, S. Lucks, and E. I. Tatlı, "Tracking dog-a privacy tool against google hacking," *CoseC b-it*, p. 8, 2007.
27. "Pulp google hacking: The next generation search engine hacking arsenal."
28. F. Sahito, W. Slany, and S. Shahzad, "Search engines: The invader to our privacy - a survey," in *Int. Conf, on Computer Sciences and Convergence Information Technology*, pp. 640–646, Nov 2011.
29. E. I. Tatlı, "Google hacking against privacy," 2007.
30. E. I. Tatlı, "Google reveals cryptographic secrets," *Kryptowochenende 2006-Workshop über Kryptographie Universität Mannheim*, p. 33, 2006.
31. K. Soska and N. Christin, "Automatically detecting vulnerable websites before they turn malicious," in *Proc. of USENIX Security*, (San Diego, CA), pp. 625–640, 2014.
32. M. Vasek and T. Moore, "Identifying risk factors for webserver compromise," in *Financial Cryptography and Data Security*, pp. 326–345, 2014.
33. C. Y. Cho, J. Caballero, C. Grier, V. Paxson, and D. Song, "Insights from the Inside: A View of Botnet Management from Infiltration," in *Proc. of the USENIX Workshop on Large-Scale Exploits and Emergent Threats*, (San Jose, CA), April 2010.
34. F. Yu, Y. Xie, and Q. Ke, "Sbotminer: Large scale search bot detection," in *ACM International Conference on Web Search and Data Mining*, February 2010.