Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

BACHELOR THESIS

Automated Inference of Web Software Packages and Their Versions

vorgelegt von

Pascal Wichmann

MIN-Fakultät

Fachbereich Informatik

Sicherheit in Verteilten Systemen

Studiengang: Informatik

Matrikelnummer: 6792948

Betreuer: Matthias Marx, M. Sc.

Erstgutachter: Prof. Dr.-Ing. Hannes Federrath

Zweitgutachter: Prof. Dr. Dominik Herrmann

Task Description

Today, many websites are dynamically generated by server-side software, such as Wordpress or Joomla. This approach can reduce the workload of administrators. However, there is also the risk of exposing personal data of users, if administrators run outdated versions with vulnerabilities. While website owners are able to look up which version they use, users often have no possibility to check websites for outdated software. To make informed decisions whether or not to trust a website with their personal data, they would need to infer the type of software and its version on the server.

Therefore, the task of this thesis is to find a technique which can be used to remotely detect software versions used on the server. The technique shall be implemented in the form of a non-interactive command-line tool. Consequently, this tool can be used to evaluate whether publicly available pieces of information, e. g., headers, the HTML body, and static files served, are sufficient to differentiate between different versions of software packages with high accuracy. The method is given access to the software for which the version should be detected beforehand. The tool should be easily extendable without having to implement custom detection algorithms for each software package. Finally, potential limitations of the approach should be discussed.

Abstract

Many websites use popular software packages, e. g., content management systems such as Joomla, Typo3, or Wordpress. A site using outdated versions of these software packages might be vulnerable to security issues and thereby endanger the privacy and security of its users.

Common content management systems consist of a large number of individual files that are available on a web server. For example, such systems provide static assets like JavaScript or stylesheet files enhancing the user experience. At least some of those assets are unique to a specific software version and can therefore be used to identify it. For some of these files, the content may be characteristic for a particular version.

This thesis implements VersionInferrer, a tool to index static files of many popular software packages and efficiently apply that generated index to infer the software version used by websites. VersionInferrer implements a generic approach which does not require manual adaptions for individual software packages.

VersionInferrer is evaluated against a sample set of 500 000 popular websites. For 19.5% of those, it is able to detect the usage of one of the 16 tested software packages. For 67.9% of those sites, VersionInferrer yields an *unambiguous* estimation for a specific version. Of the sites for which at least one version was detected, 25.5% may use a version for which known vulnerabilities exist. A manual verification of 50 sites showed that the results of VersionInferrer are reliable with only few exceptions.

The work confirms the results of previous research that revealed outdated software on a considerable number of websites. With VersionInferrer, there is a tool that simplifies such examinations and makes them available to ordinary users. The provided transparency can create an additional incentive for website operators to update their sites.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

Many websites use popular software packages, e. g., content management systems (CMSs) such as Joomla [24], Typo3 [37], or Wordpress [41]. A site using outdated software versions may be vulnerable to security issues and thereby endanger the privacy and security of its users. This issue is rated as a severe problem by the open web application security project (OWASP) ten most critical web application security risks [28].

Another related issue is the use of third-party libraries by many software systems, e. g., JavaScript libraries like jQuery [35], or server-side libraries like Django [7]. An outdated version of those libraries may pose a security vulnerability. When the developers are not aware of vulnerabilities in libraries they use, the resulting product is likely to contain such exploitable vulnerabilities.

Software systems have unique characteristics, some of them varying between versions. For example, websites usually provide static assets like JavaScript or stylesheet assets enhancing the user experience. At least some of those assets are unique to a specific software version and can therefore be used by web security analysts to identify the version of the software system used by a website. If outdated versions are found, this may be an indicator of poor operations and security processes, especially if versions with known security vulnerabilities are detected.

For open source software, static assets can be automatically collected and mapped to specific versions from their public source code repositories. This thesis introduces VersionInferrer, a tool that implements methods to index static assets of many popular web software packages. The generated index allows to infer the software version used by websites efficiently, i. e., causing a minimum number of requests to the analyzed web server. Reducing the number of requests keeps the server load low, which is important when applying the method without informing the site operator in advance (for ethical considerations cf. Chap. 7).

There are multiple use cases for VersionInferrer. On the one hand, it can be used by professionals during a black-box penetration test. On the other hand, it may also empower regular web users, because it enables them to determine whether a particular site neglects security best practices.

On their own, low-level results, i. e., the currently used version number, are only useful for advanced users. However, the version information could also be presented in a more user-friendly manner (e. g., by stating whether the version is subject to known vulnerabilities). It could also be integrated into security assessment and benchmark platforms such as PrivacyScore [22].

1.2 Related Work

This section summarizes the related research on the prevalence and impact of outdated software versions on the web. Afterwards, relevant related work on automated software inference is presented.

Van Goethem et al. provide an analysis of several security features in use on the web [8]. They focus on general security measures, e. g., insecure transport layer security (TLS) implementations, sensitive information leakages, and mixed-content inclusion. Additionally, V. G. et al. analyze outdated server software and CMSs using a small set of software systems, which consists of WordPress, Joomla, vBulletin, and MediaWiki. According to their finding, 8.39% of the analyzed websites use an outdated CMS version, which is a significant number given the small set of CMS software packages.

Vasek et al. investigate risk factors for web server compromise [38]. They show that popular CMSs can create a large attack surface, which is correlated to their popularity. Their work clarifies that outdated software versions are a risk that should be mitigated.

Nikiforakis et al. evaluate the usage of third-party providers for JavaScript library inclusion, focusing on potential vulnerabilities caused by the incorporation of external scripts [26]. They give an overview of the impact content delivery networks serving libraries have on web security.

Lauinger et al. analyze the usage of outdated JavaScript libraries on popular websites [16]. Their work focuses on the client-side security of websites deteriorated by outdated and vulnerable JavaScript libraries. They analyze which websites use outdated libraries and find common causes for it.

Wappalyzer [1] aims to detect all technologies used by a website, i. e., used CMSs, frameworks, and programming languages. To achieve that, they use various indicators for all supported software packages, for example, regular expressions, header checks, or icons. The accuracy of their results differs based on the strength of the indicators. Some indicators are rather unreliable, e. g., frequently yielding false positives. Wappalyzer could still be used to get first estimates on which to base further guesses on. Furthermore, no specific versions are detected. The Wappalyzer project provides comprehensive datasets that can be used to infer software packages on websites. Therefore, VersionInferrer uses it to get first estimates for the inference process.

The "web app tool sniffer" Chrome extension [25] checks the list of source files referenced on a visited webpage against a mapping of known files to frameworks and software packages. However, it only considers the information about static files from a single request made by the browser. In addition, it only detects software packages and frameworks and not their versions, e. g., it states that a site is using Drupal by displaying its logo.

Pyfiscan [31] allows server administrators to scan their local filesystem for outdated web software packages. This enables hosting providers to check whether their customers are using outdated and vulnerable software versions. As pyfiscan inspects local filesystems and not the publicly accessibly webroots, it is not feasible for scanning arbitrary websites for outdated versions in order to assess their security and privacy features. While pyfiscan addresses multiple software packages at once, others focus a specific package to provide in-depth information. For example, the `wp_check_plugin_dir` project provides a script [12] that cheks for plugin vulnerabilities in a local WordPress plugin directory.

Guess [40] is a website providing users with the option to submit URLs which are scanned for used frameworks. Unfortunately, it does not provide any insights on how they gain those results. The author of Guess has been contacted during the research for this thesis. He did not disclose any internal details of the tool but only named aspects similar to those regarded by Wappalyzer. “built with” [4] is a similar tool which provides more information with less accuracy.

The W3Techs project [29] provides extensive surveys on the usage of web technologies, including the use of CMSs and specific versions. However, they neither disclose their analysis methods nor their source code. Instead, they only provide a short summary giving a very rough overview of their methods.

Several other projects can be used during the detection process. The OWASP maintains a favicon database [27] containing checksums of popular software packages’ icons. These can be used to get a first estimation of the software package in use. In addition, it may reduces the number of possible versions.

Fingerprinter [6] generates fingerprints for websites and software packages. A fingerprint in this context consists of identifiers of the static files of a website, or a software package. Those fingerprints can be applied to infer a software version for a URL in three modes. The first one checks every single fingerprint from the database of software package fingerprints against the webroot. The second one checks only those which uniquely identify a specific version. The last mode only compares the fingerprints of those files which are referenced from the provided URL. The user has to specify the software package for Fingerprinter to detect its version. Only those features required for the fingerprinting itself are provided by this tool. Furthermore, software packages which disclose their versions in common ways and therefore do not need fingerprinting for detection are explicitly not supported. In contrast to VersionInferrer, Fingerprinter does not optimize the number of requests (cf. Chap. 7).

BlindElephant [36] is a similar fingerprinting tool. In addition to the functionality provided by Fingerprinter, they use a more extensive inference process resulting in multiple returned versions ordered by their probabilities. The name of the software package of which the version should be detected is required as well. Table 1.1 compares Fingerprinter and BlindElephant to VersionInferrer.

Table 1.1: Comparison of Fingerprinter, BlindElephant, and VersionInferrer

	Fingerprinter	BlindElephant	**VersionInferrer**
version inference without any knowledge about a website except for its URL	✗	✗	✓
detection of multiple technologies on a single website	✓	(✗)[a]	✗[b]
generic indexing of software packages without package-specific information	(✗)[c]	✗	✓
minimization of the load generated on the web server (i. e., the number of requests)	✗	✗	✓
intuitive usage	✗	✗	✓

a. The tool can be used multiple times for separate technologies, thus being able to detect multiple technologies if used multiple times.

b. VersionInferrer can be extended to support multiple technologies as well.

c. An object-oriented approach is used that supersedes the redundancy of code, but several custom functions are required for each software package.

The plugin-based whatweb tool [13] utilizes several methods to detect software packages and versions. Its plugins combine simple checks like HTML headers as well as fingerprinting approaches like the related works presented before. This focus on plugins allows arbitrary methods to be integrated into whatweb. Therefore, it is primarily a collection of several tools that give helpful insights on the internals of websites.

Similar fingerprinting approaches are used in other areas than the web as well. For example, nmap [20] uses a fingerprinting approach of the TCP/IP stack to detect operating systems on the network [19].

Concluding, several researches have been made regarding vulnerable web software and the inference of software versions. However, only a few projects focus on developing a reliable and easy-to-use method to automatically detect software versions in use by arbitrary websites. Furthermore, it is not sufficient to keep the client-side scripts up to date. A vulnerable backend software can have an even bigger impact on the security of website users. Therefore, both the used JavaScript libraries and their versions as well as the server-side software need to be assessed. This thesis focuses on assessing the server-side software packages.

1.3 Reference Implementation

In this thesis, *VersionInferrer* is presented, an automated tool that infers software packages and versions used by websites. It is handed in together with this thesis as a proof-of-concept. The implementation is also released as free software [39]. A short explanation of the usage of VersionInferrer is provided in Appendix A.

1.4 Organization of the Thesis

The thesis starts out in Chap. 2 by analyzing the problems that need to be solved for the automated inference of website software packages. Several fundamentals for the index construction and the version inference are explained in Chap. 3, followed by a presentation of the approach in Chap. 4. An evaluation of the chosen approach including automated and manual verification of the results is presented in Chap. 5. Limitations of that approach as well as challenges encountered during this thesis are discussed in Chap. 6. Chapter 7 discusses ethical considerations of providing a tool that detects versions. Chapter 8 presents protective strategies for site owners which make it more difficult to infer the version. Finally, Chap. 9 proposes future work and concludes this thesis.

2 Problem Analysis

This chapter discusses the problems that need to be solved in order to construct an automated method inferring software packages and versions used by websites.

The following assumptions were determined to be reasonable for the development of a version inference tool. It has the possibility to retrieve arbitrary paths from the internet. The tool is provided with a URL of a website that should be analyzed for its used software packages and their versions. Furthermore, it has access to the source code of software packages beforehand.

2.1 Characteristics of Software Packages and Versions

A software package usually consists of source code that executes on the server as well as source code that is run on the client. The server-side code is not available to website users, but only the results of the server-side execution. In contrast, the client-side code is sent to the user as it has to run in the user's browser.

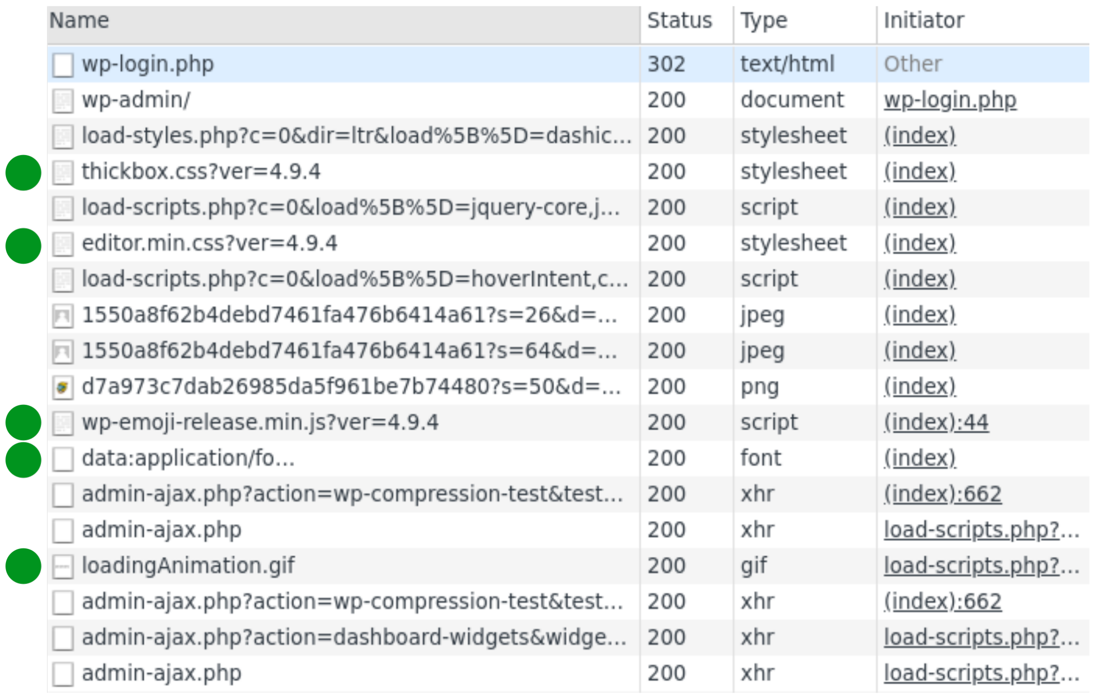
The server either serves the client-side source code unprocessed, i. e., as static files, or generates it dynamically, e. g., for localization and internationalization purposes. However, dynamically generated source code may contain static parts.

Figure 2.1 shows the site assets that the admin dashboard of the WordPress software references. The resources marked with a green dot are static, the others are dynamically generated during runtime.

Distinct software releases can differ in the server-side source code as well as in the client-side source code. Those differences in the client-side source code can be detected by retrieving the source code over HTTP, like a web browser would do. Changes in the server-side source code result in different behavior of the website. This can not generally be recognized from the data transmitted by the server. Instead, it can only be observed after having analyzed the differences in the server-side logic.

Most software packages are maintained using a version control system (VCS) such as Git. Those contain metadata about software releases and allow the comparison of different versions. For open source software, the VCS repositories are usually publicly available. Therefore, those can be used to check out the source code for specific versions of that software.

Releases of software packages usually contain information about the software, changes of the version at hand, and documentation on its installation. A common way to deliver such information is using *README* or *CHANGELOG* files within the release. Some packages place such files in the same directory as the program files that are deployed to the webroot. Therefore, it is likely that a site operator deploying such a software package does not remove those files, making them available to the public. For example, the WordPress



	Name	Status	Type	Initiator
	wp-login.php	302	text/html	Other
	wp-admin/	200	document	wp-login.php
	load-styles.php?c=0&dir=ltr&load%5B%5D=dashic...	200	stylesheet	(index)
●	thickbox.css?ver=4.9.4	200	stylesheet	(index)
	load-scripts.php?c=0&load%5B%5D=jquery-core,j...	200	script	(index)
●	editor.min.css?ver=4.9.4	200	stylesheet	(index)
	load-scripts.php?c=0&load%5B%5D=hoverIntent,c...	200	script	(index)
	1550a8f62b4debd7461fa476b6414a61?s=26&d=...	200	jpeg	(index)
	1550a8f62b4debd7461fa476b6414a61?s=64&d=...	200	jpeg	(index)
	d7a973c7dab26985da5f961be7b74480?s=50&d=...	200	png	(index)
●	wp-emoji-release.min.js?ver=4.9.4	200	script	(index):44
●	data:application/fo...	200	font	(index)
	admin-ajax.php?action=wp-compression-test&test...	200	xhr	(index):662
	admin-ajax.php	200	xhr	load-scripts.php?...
●	loadingAnimation.gif	200	gif	load-scripts.php?...
	admin-ajax.php?action=wp-compression-test&test...	200	xhr	(index):662
	admin-ajax.php?action=dashboard-widgets&widge...	200	xhr	load-scripts.php?...
	admin-ajax.php	200	xhr	load-scripts.php?...

Figure 2.1: Site Assets Loaded by the WordPress Admin Panel

software package contains a *README* file that explains what WordPress is, lists its requirements, and describes the steps necessary for its installation. While it does not contain a version identifier, the combination of the versions of the dependencies as well as the copyright information yield several *README* files that are unique to one specific WordPress version.

Furthermore, software packages may provide information about themselves, e. g., by using a *generator* meta attribute or a footer containing the name of the software and possibly its version. Some software packages allow this behavior to be changed in the configuration. For example, WordPress serves a *generator* meta attribute that contains the package and version identifier by default. Moreover, the initial site theme contains a footer containing the string “Proudly powered by WordPress”.

2.2 Characteristics of Websites

A website can be retrieved using its uniform resource locator (URL). This contains a domain name or IP address identifying the host from which the webpage should be requested. The actual request is sent using the hypertext transfer protocol (HTTP). The server replies with a response that consists of a response body containing the actual content as well as headers specifying information about the response, and the status.

Both the headers and the response body can contain information about software packages in use by the website. Some software packages set an HTTP header called *X-Powered-By* that contains the name and the version of the package. The response body contains the code that should be interpreted by the browser of the client, or additional assets required to render the webpage, e. g., style definitions, image files, data files, or similar. Depending on the application, it can also contain arbitrary files for download.

The contents of the HTTP response can either be dynamically generated or static. Dynamic content has the property that it can depend on the request sent to the server as well as the environment. For example, a request may contain information about the current user – a session identifying him across multiple requests, or a preferred language to display information in.

Static assets can be optimized during deployment to reduce the transmission time. This can be achieved by minifying client-side source code files (e. g., removing redundant spaces and newlines, or replacing identifiers by shorter names) or compressing data files (e. g., images). Additionally, static assets can be embedded within the body of dynamically generated responses. In this case, parts of dynamic pages are actually static. To compare those parts against known artifacts of software packages, they must be extracted from the page. However, extracting them in a generic way is not possible, because the extraction method depends on the specific implementation of the underlying software package. Therefore, reliable extraction requires additional information for each software package.

2.3 Combining the Characteristics of Software Packages and Websites

Given the knowledge of the characteristics of software packages and their versions, and of websites, it is possible to infer the software versions that are in use by websites. The characteristics of the website can be determined by sending specific requests and analyzing the returned responses. An efficient approach for that will be described in Sect. 4.4. In order to compare the characteristics of websites with those of software packages, the characteristics of the software packages need to be known as well. To prevent the necessity of analyzing all supported software packages for each website, an index of the characteristics of the software packages can be built in advance. The construction of such an index is explained in Sect. 4.3.

The primary characteristics dealt with in this thesis result from the assets for the client. That includes static files served by the web server, optimized static site assets, and also dynamically generated information like the markup of the main page.

To infer a software package and version of a given website, the generated index is queried for the software packages' characteristics. The performance of this approach is evaluated in Chap. 5.

3 Fundamentals

This chapter reviews techniques that have informed the design of VersionInferrer.

3.1 Inverse Document Frequency

The inverse document frequency (IDF) is used to calculate the importance of specific words within a set of documents. Frequently used words differentiate less between multiple documents while rarely used words bring a higher information gain. Therefore, rare words are weighted higher than frequent words. The IDF is calculated by the following formula where N is the total number of documents and n_i the number of documents containing the term t_i [30]:

$$\operatorname{idf}(t_i) = \log \frac{N}{n_i}$$

This IDF can also be applied for static files. The words correspond to the static files from the index, and the documents to the software versions. Consequently, static files that are in use by many different software packages and versions, i. e., common JavaScript libraries like jQuery, do not generate much entropy. At the same time, a static file which is in use by a single software version only is rather valuable for the recognition. This property is modeled using the IDF.

3.2 Decision Trees

Decision trees are used to model decisions and their possible outcomes. A decision tree is a hierarchical data structure that defines several *splits* based on attributes of data samples using test functions. It consists of nodes that correspond to the data attributes being tested. Each node has outgoing branches according to the possible outcomes of its tests [14]. Figure 3.1 shows an example of a decision tree.

A common algorithm for decision tree construction is the ID3 algorithm. It starts with a tree with one node (the root node) containing all items. The attribute with the highest entropy is selected and an outgoing branch from the node created for each value, moving the items to the new nodes based on their value of the chosen attribute. This step is called *split*. It is repeated recursively on all nodes until all attributes have been split [14].

To receive an outcome from a decision tree, it is evaluated against complete datasets, i. e., all attributes of all the items are already known and the full tree is constructed in advance. For the inference of software versions based on static site assets, determining information for a decision is expensive (e. g., requires at least one request to the web server). In addition, a lot of information from the index is usually not required because the main webpage already allows the exclusion of the majority of known static files. Therefore,

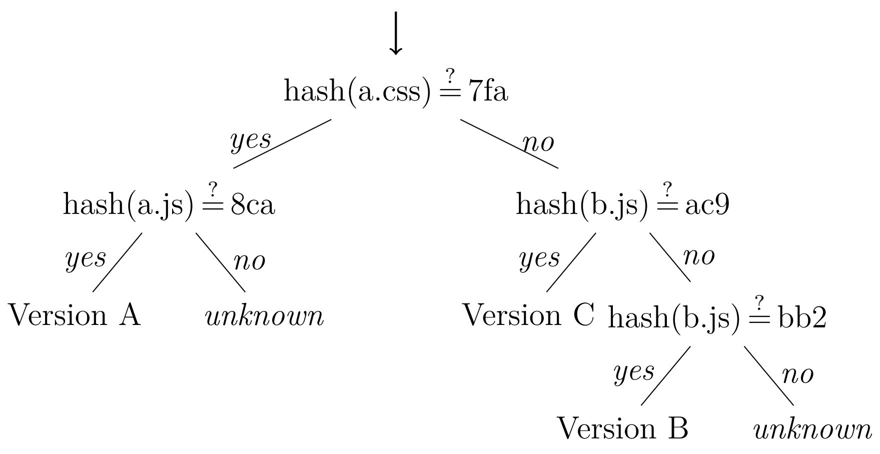

hash(a.css) $\overset{?}{=}$ 7fa
yes
no
hash(a.js) $\overset{?}{=}$ 8ca
hash(b.js) $\overset{?}{=}$ ac9
yes
no
yes
no
Version A
unknown
Version C
hash(b.js) $\overset{?}{=}$ bb2
yes
no
Version B
unknown

Figure 3.1: Example of a Decision Tree

constructing a complete decision tree would result in a huge overhead due to the creation of many unused branches. Instead, VersionInferrer dynamically decides which site asset could yield a high entropy given the previously gained information. This is similar to the decision tree approach, but does not require constructing a full decision tree.

3.3 Association Rule Mining

Association rule mining deals with the discovery of relations between itemsets. It allows the construction of statements of how likely items influence each other, e. g., how likely it is that an item occurs together with another item, or whether the probability is low that an item occurs if another occurred as well. Association rule mining usually deals with several *transactions* consisting of itemsets, e. g., purchases of customers in a supermarket [42].

Association rules are of the form $A \to B$. This rule states that an itemset containing the item (or the set of items, depending on the specific definition) A also contains the item(set) B.

Definition 1. *The* ***support*** *of an item within a dataset is the number of transactions that contain it [42].*

Definition 2. *The* ***confidence*** *of an association rule $A \to B$ is the probability that a transaction contains the item B under the condition that it contains the item A [42].*

The task dealt with in this thesis is similar to the tasks handled in association rule mining. In this thesis, static assets from websites are considered. Given a guess for a specific software version, the assets expected under the assumption that the guess is true can be extracted from the index. By retrieving those paths (or a subset of them) from the webroot, the ratio of expected assets that actually exist and whose contents match the expectations can be determined. Thus, we do not consider transactions in this thesis, but collections of static files.

The methods of association rule mining can be transferred to VersionInferrer, thus giving a formal background on how to define support and confidence. However, no explicit association rule construction is used in this thesis. Furthermore, not the complete collection of static files is known, but only a subset of the assets of a website is retrieved that is based on the expected entropy of the files.

3.4 Wappalyzer Project

The Wappalyzer project [1], which has been introduced in Sect. 1.2, develops a tool which uses generic patterns on websites to detect technologies and software packages they use. They maintain a comprehensive JSON file (the “apps” file) containing the information that Wappalyzer uses to detect software packages. This file is used by VersionInferrer in order to find first estimates for the inference process. Using these first estimates, the fingerprinting approach can be applied more targetedly, i. e., without the need to retrieve a lot of files which probably do not exist on the webroot.

Wappalyzer is not directly used as a subroutine within VersionInferrer, but the results of requests are compared against the apps file. This contains checks for HTTP headers, for HTML meta tags, for specific patterns contained in the body of a request, and for specific script tags indicating a specific software package. These indicators are used to get a first estimate of a software *package*, not yet a specific *version* of a package. The main fingerprinting approach described in Sect. 4.4 is used to isolate potential versions of those packages yielded in the first estimation phase.

While Wappalyzer executes the JavaScript code embedded on websites during their analysis, VersionInferrer only parses the HTML and the HTTP response headers. Therefore, the context of the JavaScript environment is not regarded for the first estimation using the Wappalyzer apps file. This has positive security implications as no program code retrieved from the web is executed during the analysis. Furthermore, this makes the analysis more efficient as no additional code (possibly requiring arbitrary time to complete) needs to be run. Nevertheless, this comes at the cost of potentially missing information that is only visible when executing the program code of the website.

3.5 File Normalization

3.5.1 Fuzzy Hashing

Usual cryptographic hashes like SHA-1 [5] can be used to recognize identical files. However, already a single differing bit produces a significantly different hash. While this is an important property for cryptographic hash functions, in the context of (static) files from a webroot, small modifications are rather common. For example trailing newlines or the concatenation of multiple static assets into a single file are commonly found in site deployments [15]. Using cryptographic hashes, such files would not yield a match when compared to the index as they are not exactly identical.

Fuzzy hash functions [17] provide a possible solution by generating similar hashes for similar input data. Therefore, a similarity to known files within the index suffices to gain further information where a conventional hash would not provide any further insights. This allows the detection of files which are almost but not completely identical.

A primary application of fuzzy hashing algorithms is in malware and spam protection. Those hashes are used to compare executable files or emails against hashes of known bad artifacts. The literature highly focuses on the attack surface of those algorithms as vulnerabilities in those fuzzy hash functions could be exploited in order to bypass malware protection software. Therefore, using a single fuzzy hash function for such purposes is not secure, but a combination of multiple such algorithms can be sufficiently secure [17]. For the purpose of comparing files from source code repositories to deployed files, the dangers of intentional collisions are negligible as they are not used to protect against malware. Software package maintainers and website operators could exploit vulnerabilities in fuzzy hashing algorithms in order to make the version detection of their packages more difficult, but this is rather unlikely and the fuzzy hashes are not used as the only comparison mechanisms. Furthermore, the static file versions of software packages are available and can be compared to files found on the web using different methods.

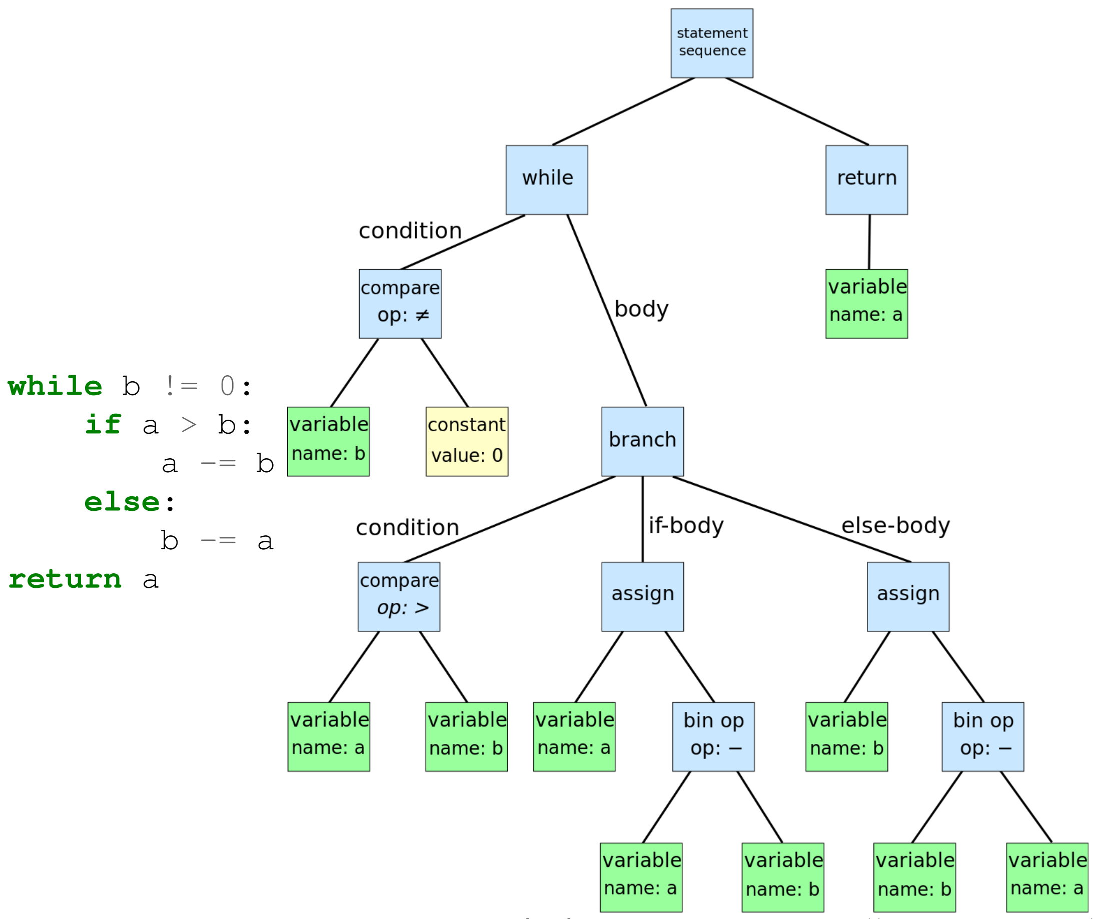
Most of the fuzzy hashing schemes proposed in the literature have the property that fuzzy hashes of similar files differ only slightly from each other, i. e., have a low edit distance. To find all the files within a database of fuzzy hashes which are similar to a given file, the edit distance of the fuzzy hash of the given file to every file in the database needs to be calculated. All the files in the database whose edit distance to the compared file is below a specific threshold yield a match. This aggregation is rather expensive if done naively as the calculation of the edit distance can not easily be optimized using database indexes or similar technologies. Especially when the database contains a very high number of files – as is the case for the static file index proposed in this thesis –, it is too slow to be used in practice to search for every file retrieved from a webroot in the database. This problem does not exist with hashes where an exact match is required as any database management system is able to efficiently search for exact matches in huge amounts of data. Therefore, more efficient alternatives than the edit distance of fuzzy hashes are required in practice. Data structures like k-d trees [3] or r-trees [10] support similarity searches. However, it needs to be evaluated if these data structures can be used with fuzzy hashes.

A comparison of the performance of different fuzzy hashing techniques and related approaches is left for future work. The VersionInferrer prototype matches files without fuzzy hashing and was found to perform already reasonably well (cf. Sect. 5).

3.5.2 Abstract Syntax Trees

An abstract syntax tree (AST) is a semantic representation of a source code grammar that is independent from a specific programming language a program is written in. ASTs can contain annotations about the source code [9]. Due to the properties of ASTs, differences in the source code that do not have a semantic impact do not change the generated AST, e. g., different indentations in the simplest case. Furthermore, when only the structure of the tree is compared, different names of expressions can be considered as equal. Especially files with minification applied are rather common on websites, thus yielding to several different variable names used in actually identical files. Figure 3.2 gives an example of an AST for Python code implementing the Euclidean algorithm.

ASTs can be used as an abstraction for source code files. They can be used for the normalization of files before the indexing or analysis process to get a deterministic canonical form resilient against source code minification. However, static files of different



```
while b != 0:
    if a > b:
        a -= b
    else:
        b -= a
return a
```

Image Source: By Dcoetzee [CC0], via Wikimedia Commons, https://commons.wikimedia.org/w/index.php?title=File:Abstract_syntax_tree_for_Euclidean_algorithm.svg&oldid=126128927

Figure 3.2: Abstract Syntax Tree for a Python Script (Euclidean Algorithm)

software versions may only differ in the naming of specific variables or results of a similar small refactoring. Those static files of different versions would be regarded as identical when applying the AST.

To ensure the reliability of using ASTs for file matching, it has to be ensured that they actually yield deterministic results, i. e., provide the same output for the same input independent from the environment or software version. A way to achieve that is to use the normalization explained in Sect. 3.5.3 which has the purpose of converting data structures into deterministic canonical forms.

The construction of an AST is rather expensive. While building the index, the AST for hundreds of files has to be constructed. When indexing hundreds of software versions of several software packages, this results in a far longer execution time than without using ASTs. A prototypical implementation of AST-based matching was found to have a prohibitive impact on the performance of VersionInferrer. Therefore, as with fuzzy hashing, the integration and evaluation of ASTs is left for future work.

3.5.3 Simple File Normalization Techniques

Identical program files can be represented in different ways, thus yielding to different checksums when using cryptographic hash functions. While fuzzy hashing algorithms as presented in Sect. 3.5.1 can be used to detect similar files, querying fuzzy hashes in the database is rather expensive, thus not being a feasible option in practice.

File normalization provides another method to achieve a similar goal. During normalization, files are cleaned into a canonical form that is the same for files which are semantically identical. The concrete normalization depends on the file type, e. g., for text files, trailing newlines and spaces can be stripped in the simplest case. Applying such a normalization leads to identical (cryptographic) checksums for files that were only similar before being processed. Unlike the checksums generated by the fuzzy hashing schemes which require a fuzzy matching against the index, the checksums of the normalized files can be compared against the index efficiently using exact matches, thus not comprising the disadvantages of using fuzzy hashes.

3.6 Outdated Software Versions and Risk Analysis

Different software packages employ different versioning and release strategies, checking whether a version is up-to-date is not straight-forward. For example, many software packages provide (several) long-term-support release branches that receive security patches while the main development is done on a different release branch that receives regular updates. Thus, it is not sufficiently reliable to compare the release date of an inferred software version to the release date of the most recent stable version released. Instead, release strategies specific to each software package need to be modeled. As these strategies can be arbitrary, it is not possible to elaborate a reliable generic method to check the up-to-dateness of a software version that does not use software-specific information. Consequently, only the *most recent stable version* (identified by its release date instead of it version number) is regarded as up-to-date in the evaluation (cf. Chap. 5).

It is important to have the most recent software version of a release branch installed because it often contains security patches. Accordingly, the existence of known security vulnerabilities for a specific software version implies risks accompanied to its usage and indicates that a newer version has probably been released. Common vulnerabilities and exposures (CVE) records can be used to automatically check for security vulnerabilities. Those usually contain information about the software package as well as the versions affected by the vulnerability. However, the information regarding the specific versions affected is not always accurate. The results gained from using the CVE data are therefore not completely reliable. Nevertheless, they can be used for an automated assessment of the risks arising from the use of a website using outdated software. VersionInferrer constructs CVE statistics from the publicly available datasets[1]. VersionInferrer compares the results of the inference against those CVE statistics to provide the user with information about security vulnerabilities.

3.7 Software Stability and Robustness

To ascertain the stability of the VersionInferrer implementation as well as its robustness in unexpected situations, testing is required. The reference implementation contains several test cases that test core features of the software. These test cases can be used to verify that code changes or updates of the environment do not break the software.

1. CVE statistics for *YEAR* can be retrieved from https://static.nvd.nist.gov/feeds/json/cve/1.0/nvdcve-1.0-YEAR.json.gz

4 Detection Technique

This chapter presents the techniques that are used within the VersionInferrer tool.

4.1 Abstractions

The focus of this thesis is to provide a generic approach that does not need to be adjusted to specific software packages. To achieve that, abstractions to the software packages, their versions, and their static files are required.

Figure 4.1 gives an overview of the used abstractions. First of all, the *software definitions* contain general metadata of software packages, e. g., their name(s), vendor, and license. In addition, *code providers* are used to access the source code, providing a well-defined interface. They can implement arbitrary methods to retrieve the source code and versions, i. e., using version control system repositories, downloadable archive files, or local filesystem paths. The *path mapping* defines a mapping from the source code tree path to the webroot path. This is required to construct the expected webroot path of static files based on their location within the source. Lastly, the *ignored paths* can be used to exclude files or directories within the paths that are mapped to the webroot using the path mapping from indexing.

Using these abstractions, all software packages with available source code can be indexed using a common approach. Figure 4.2 shows the usage of those abstractions for the WordPress software package. A similar generic definition can be created for any other software package in order to allow the automated indexing.

Additional abstractions are used for the data store backing the index. All access to the index is handled through a *backend* abstraction.

VersionInferrer uses a relational SQL database as data store. Due to the construction of relational databases, aggregations can be calculated rather efficiently on database level and indices can be used to further improve the performance of frequent queries. Nonetheless, the abstractions allow the usage of any other data store as a backend as well.

Further abstractions are used for static *file types*. These abstractions are used for the decision whether a file is of a specific type, whether it is relevant for the indexing or the inference process, and for the preprocessing of its contents, i. e., the canonicalization described in Sect. 3.5.3. Moreover, the file abstractions provide a modular way of adding (or removing) supported file types.

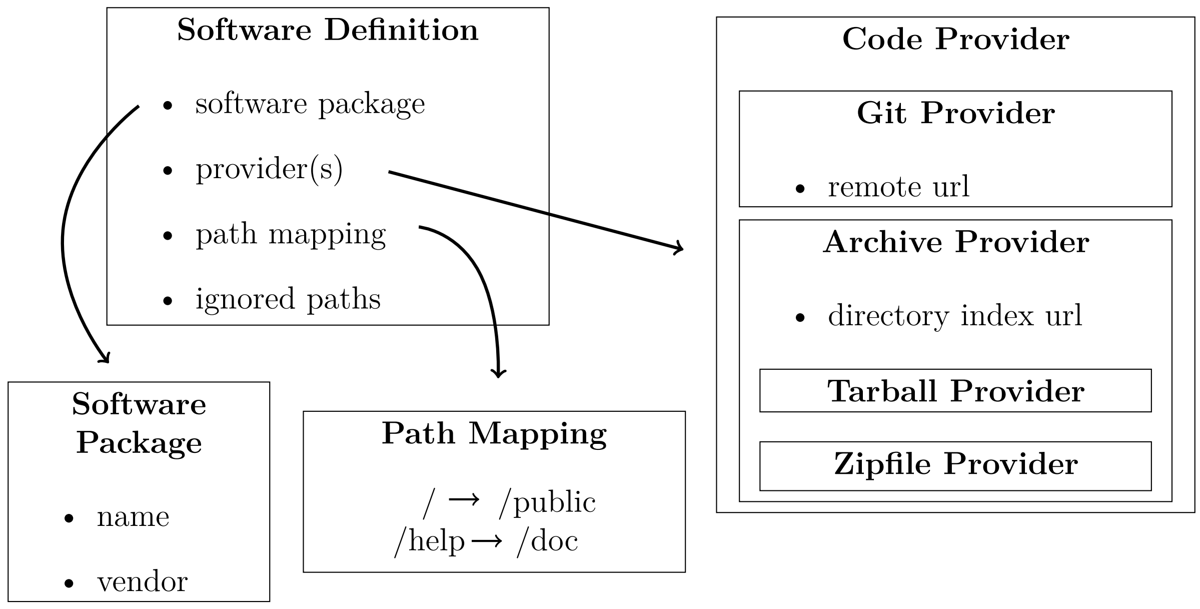


Figure 4.1: Software Package and Source Code Abstractions

- **software package**
  - **name**: WordPress
  - **vendor**: WordPress
- **code provider**: Git provider
  - **url**: https://github.com/WordPress/WordPress.git
- **path mapping**: / → /
- **ignore paths**: None

Figure 4.2: Software Definition of Wordpress

4.2 Static File Processing

The files loaded from the source code directories and retrieved from the analyzed websites are processed before they are used for indexing or inferring, respectively.

The file type abstractions described in Sect. 4.1 are used to determine the file type. Furthermore, they define whether a specific file (based on its determined type) should be considered during index construction or during the inference.

The processing of static files includes converting them into a canonical form as described in Sect. 3.5.3. Due to the drawbacks of abstract syntax trees and fuzzy hashing algorithms described in Sect. 3.5, they are not used in the final prototype of VersionInferrer. Instead, only JSON and YAML files are brought into a canonical form by enforcing a deterministic order of items within key-value structures and lists, and stripping leading and trailing spaces.

4.3 Index Construction

To identify candidate versions from static files, an index of known static files is required for comparison. As many popular software packages exist and most of them have frequent releases, an automated way of creating such an index is needed. The number of required manual adaptions for each software package should be minimal.

To construct the index, the known software packages are loaded from the available definitions. For each of them, all released stable versions are fetched using the abstractions described in Sect. 4.1.

To index a software version, its source code is obtained using the source code provider. Subsequently, all relevant static files are determined using the file type abstractions. These abstractions define a processing procedure. The actual processing is described in Sect. 4.2.

To reduce the required computation time, the indexing is applied incrementally, i. e., ignoring all versions which have been indexed previously.

4.4 Software Package and Version Inference

The user only provides a URL of a website for which VersionInferrer should infer the software package and its version. VersionInferrer's inference procedure is divided into two steps. Firstly, initial guesses are identified using the response from the provided URL. These guesses consist of a software package and its version. Secondly, the list of guesses is used to get webroot paths from the index providing entropy to differentiate between the guesses. This second step is repeated several times until a best guess is inferred or too many webroot paths do not provide new information gain. The process is graphically visualized in Fig. 4.3. The following sections describe the two steps in detail.

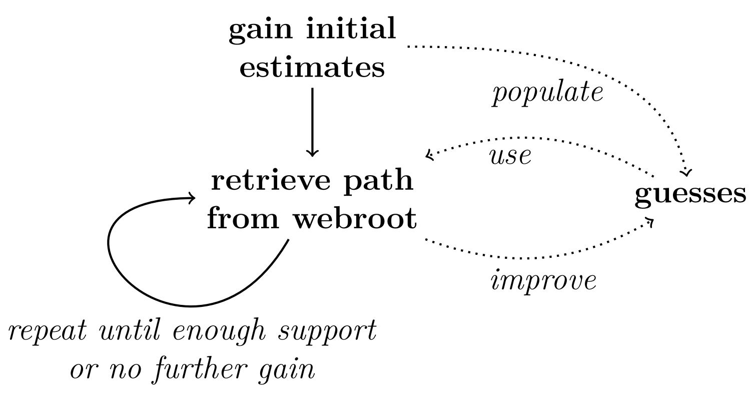

gain initial estimates
populate
use
retrieve path from webroot
guesses
improve
repeat until enough support or no further gain

Figure 4.3: The Two Steps of the Inference Process

4.4.1 Gaining Initial Estimates

First of all, VersionInferrer tries to detect a set of software versions that might be used on that site. In order to achieve that, the page served at the provided URL is retrieved. The response is analyzed for some general hints, e. g., a *generator* meta tag. That tag often contains the name of the used software package and – in some default configurations – includes a version string as well.

In addition, the data of the Wappalyzer project presented in Sect. 3.4 is used to get names of software packages that might be in use. For that, both the response headers and the response body are compared against the Wappalyzer data for those software packages that are stored in the index. Since the Wappalyzer data does not include version information for most of the software packages, all the versions of a software package are added to the set of initial estimates.

Furthermore, all the assets that are referenced from the main page are retrieved and compared against the index. All the indexed versions that use any of those assets are added to the set of first estimations.

4.4.2 Finding Strong Guesses Using Iterations

After having collected first estimates for the used software versions, these are improved in multiple iterations to yield *guesses*.

Each iteration uses the guesses that have been yielded in the previous iteration. The first iteration uses the initial estimates instead of guesses from the previous iteration.

To improve these guesses, paths with a high entropy are required to further differentiate between the current guesses. Thus, the index yields new paths, which correspond to decisions in a decision tree. In detail, the number of software versions from the set of guesses that use a static file from the index is determined for each known static file. Then, these are ordered by their IDF as explained in Sect. 3.1. Those paths that have already been retrieved are excluded to prevent the redundant retrieval of site assets.

The paths with the highest estimated entropy are then retrieved from the webroot. The exact number of retrievals per iteration depends on the parameters (cf. Sect. 4.5). Furthermore, more assets are retrieved if requests fail, configured by a parameter as well.

Those retrieved paths are not necessarily referenced from the website. This can provide additional information that is not directly available from the regular website, i. e., some assets might only be referenced for authenticated users, or when a specific feature is enabled by the site administrator.

After having retrieved the determined paths, the information they provide is used to rate all guesses and possibly add new guesses, which is explained in the next subsection. It results in a new set of best guesses that are a subset of all guesses.

If the best guesses are not improved in an iteration, the counter of *useless iterations* is increased. When this counter reaches a configurable value, the analysis is stopped regardless of the quality of the result.

Based on the new set of best guesses, another iteration is started or the analysis stopped. If there is already only one single best guess left and its support is high enough (cf. Sect. 4.4.3), it is returned as a result and no further iterations are executed. In addition, the maximum number of iterations is configurable. When that number is reached, the analysis stops and no further iterations are used to improve the result.

4.4.3 Rating Guesses

Guesses are rated based on the information that the retrieved site assets and their webroot paths provide. This rating is used to determine a subset of *best guesses*. Furthermore, these best guesses are rated on whether they have enough support to accept them as a final result.

Each guess is rated according to the number of assets retrieved from the webroot of the site that match the index, that did not match the index, or that did not exist on the expected path on the webroot. These ratings are weighted using the IDF described in Sect. 3.1, giving rare assets a higher weight than static files that are in use by many software packages. Furthermore, the positive and negative matches are weighted according to the parameterization as explained in Sect. 4.5.

The rating of the guesses is similar to the support definition used in association rule mining explained in Sect. 3.3. Therefore, the weighted rating of the guesses divided by the number of guesses is called *support*.

Having rated all guesses, comparing them is possible. Often, some guesses have a very low support that does not significantly increase. Such guesses are therefore not regarded in iterations to improve the efficiency. In addition, this optimizes the entropy and therefore reduces the length of the path towards the final guess in the corresponding decision tree. It is still possible for new insights of future iterations to improve the support of such guesses, thus making them relevant for new path decisions.

The guess with the best rating is called the *best guess*. If multiple guesses have the same rating, the best guess is a set of multiple guesses. It is possible that the final result of

VersionInferrer is a set of multiple best guesses. In that case, no further distinction between those versions (and possibly packages) is possible with the given parameterization.

To assess the support of a set of guesses, the *guess decisiveness* is determined. It is calculated by the mean of the differences from the support of the best guess to the support of every other guess. This is used to measure the information gain of an iteration. The change of this decisiveness from an iteration to the next is called the iterations *gain*. If the gain is below a configurable threshold, the iteration is marked as useless (cf. Sect. 4.4.2).

4.4.4 Exceptional Situations

Firstly, if no first software version estimate can be guessed in the first step, it is not clear which assets should be retrieved in the second step. Potentially, it would be necessary to retrieve a lot of site assets in order to isolate the search space, and the probability of actual results is rather low. Therefore, the second step is skipped and VersionInferrer terminates without a result if no single estimate is yielded by the first step.

Secondly, many failing HTTP requests are considered as a possible attack indication by several website hosters. Therefore, if many retrievals fail within an iteration, the inference process is aborted.

4.5 Parameterization

VersionInferrer has to make several decisions, including about what paths to retrieve from a webroot, how to assess gained information, and whether more iterations are required to get enough confidence for the guess. While constructing the index, decisions about what files to include and how they are processed are required. These decisions depend on several factors, some of them influencing each other. Furthermore, the optimal decisions highly depend on the site that is being analyzed. As VersionInferrer focuses on providing a generic approach towards the web application inference, such optimal decisions can not always be taken. Balancing between false positives, false negatives, and resource consumption requires adaptable *settings* which are presented in Table 4.1.

Table 4.1: Parameters Used by the Reference Implementation

Parameter	**Description**
guess ignore distance	The minimum distance of the support of distinct guesses required to discard the weaker guess (default value: 3)
guess ignore min positive support	The minimum positive support the best guess needs to have in order to discard any guess at all (default value: 2)
guess relative ignore distance	The relative distance of the best guess's support to the support of inferior guesses that is required to ignore the inferior guesses (default value: 0.3)

guess limit	The number of guesses that should be considered to select paths with high entropy (default value: 7)
iteration min improvement	The minimum improvement in the support of the best guess compared to other guesses in order to mark an iteration as not worthless (default value: 0.5)
max iterations	The maximum number of iterations allowed in total (default value: 15)
max iterations without improvement	The maximum number of iterations that are allowed to not yield an improvement until the analysis is stopped (default value: 3)
min absolute support	The minimum absolute support that the best guess needs to have in order to be accepted (default value: 10)
min support	The minimum support that the best guess needs to have relative to the number of site assets that have been regarded in order to accept the guess (default value: 0.2)
positive match weight	The weight to apply on positive matches (i. e., site assets that are part of the guessed software version and actually match the asset deployed on the analyzed site) (default value: 1)
negative match weight	The weight to apply on negative matches (i. e., site assets that are part of the guessed software and do *not* exist on the analyzed site at the expected path or do not match the asset from the index (default value: 0.1)
failed asset weight	The weight to apply on assets that can not be retrieved due to a request failure (default value: 1)
min assets per iteration	The minimum number of successful assets that should be retrieved to stop an iteration early (default value: 2)
max assets per iteration	The maximum number of assets that should be tried to be retrieved within an iteration (default value: 8)
html parser	The backend that is used to parse hypertext markup language (HTML) source code (default value: *html.parser*)
html relevant elements	The HTML tags that are relevant for the detection of referenced site assets (default value: *a*, *link*, *script*, *style*)
cache directory	The directory in which to store cache data, i. e., VCS repositories from software packages for indexing (default value: *cache*)
supported schemes	The schemes/protocols that are supported for file retrieval (default value: *http*, *https*)

http timeout	The maximum number of seconds to wait for an answer from the web server before terminating the request (default value: 3)

There are different categories of parameters. Firstly, technical parameters define implementation details such as which backend should be used in order to parse responses and source code files, or which parts of source code (e. g., which HTML tags) are relevant. Secondly, decisional parameters define the actual beviour during the analysis of websites. This includes the required confidence for guesses in order to accept them, or the distance of distinct guesses that is required to discard the weaker guesses.

To determine the default parameters, different values and their impact have been tested. The default values listed in Table 4.1 yielded good results for a set of websites with known versions. Future research might include the optimization of the parameter values (cf. Sect. 9.1.4).

Some parameters influence others. For example, the relative and absolute distance required to ignore guesses depend on each other. If the distance is sufficiently high to be above either of them, the guess is ignored independently from the other parameter.

5 Evaluation

In order to assess the reliability of VersionInferrer, its accuracy is evaluated with real-world software packages. Firstly, *white-box* evaluation is carried out to test the tool against websites of which the used software package and version is known. Secondly, a *black-box* evaluation analyzes the behavior of VersionInferrer when scanning arbitrary websites of which the exact software package and version is not known.

5.1 White-Box Evaluation

A white-box evaluation analyzes synthetic sites that have been set up for this purpose. Thus, ground truth (software package and exact version) is known. Such a white-box approach allows an evaluation of the theoretical performance of VersionInferrer. This section presents the results of this white-box evaluation using the Drupal and the WordPress software packages.

During the white-box evaluation, VersionInferrer's index contained all the versions that were released. Hence, more recent versions were already known to the index while scanning old versions. If the versions had been scanned while they were the most recent version, the newer versions would not have been known to the index.

Table 5.1 shows the inference results for clean Drupal installations, Table 5.2 for clean Joomla installations, and Table 5.3 for clean WordPress installations. All the tested versions are detected correctly by VersionInferrer. Except for the 4.9.4 and 4.9.3 versions of WordPress and the 3.7.4 and 3.7.5 versions of Joomla, VersionInferrer uniquely identifies the correct version, i. e., its result consists of a single guess. Version 4.9.4 of WordPress is a maintenance release that only fixes a server-side issue. Versions 3.7.5 of Joomla fixed a server-side bug that was introduced in version 3.7.4. Therefore, VersionInferrer cannot differentiate between those versions as they do not change the static assets.

Table 5.1: Results of Scanning a Clean Drupal Installation

Version	Correct version guessed	Additional versions guessed
8.0.0	✓	–
8.0.1	✓	–
8.0.2	✓	–
8.4.0	✓	–
8.4.1	✓	–
8.4.2	✓	–
8.4.3	✓	–
8.4.4	✓	–

Table 5.2: Results of Scanning a Clean Joomla Installation

Version	**Correct version guessed**	**Additional versions guessed**
3.7.0	✓	–
3.7.1	✓	–
3.7.2	✓	–
3.7.3	✓	–
3.7.4	✓	3.7.5
3.7.5	✓	3.7.4
3.8.0	✓	–
3.8.1	✓	–
3.8.2	✓	–
3.8.3	✓	–
3.8.4	✓	–
3.8.5	✓	–

Table 5.3: Results of Scanning a Clean WordPress Installation

Version	**Correct version guessed**	**Additional versions guessed**
3.9	✓	–
3.9.1	✓	–
3.9.2	✓	–
3.9.3	✓	–
3.9.4	✓	–
4.9	✓	–
4.9.1	✓	–
4.9.2	✓	–
4.9.3	✓	4.9.4
4.9.4	✓	4.9.3

Upgrading software packages instead of a clean installation can lead to inferior results when using VersionInferrer (cf. Sect. 6.2). Therefore, *updated* Drupal, Joomla, and WordPress sites have been analyzed in the white-box evaluation. These sites were not set up with a clean installation of the target version, but instead updated from a previous version. For Drupal, version 8.4.0 was installed initially, upgraded to version 8.4.2, and then upgraded to version 8.4.4. For Joomla, version 3.8.0 was installed initially, upgraded to version 3.8.3, and then upgraded to version 3.8.5. For WordPress, version 3.9 was installed initially, upgraded to version 4.9, and then upgraded to version 4.9.4. After each upgrade step, VersionInferrer was used to analyze the site. VersionInferrer yields the same results for the chosen upgraded versions as for the cleanly installed.

5.2 Black-Box Evaluation

While the white-box approach allows an evaluation of the theoretical performance of VersionInferrer, websites may not use cleanly installed or updated software packages. For example, website operators can apply custom patches to the software packages they use, which can modify aspects of their site. Therefore, public websites need to be analyzed with VersionInferrer to assess the quality of the approach on real websites.

5.2.1 Sample Set

There are multiple sources that distribute lists of popular websites. A well-known provider of such lists is Alexa (a service run by Amazon). The Alexa Top 1 Million list [2] is used by many researchers (e. g., by Helme in his regular scans, cf. [11]). However, the Alexa Top 1 Million has some inherent drawbacks. Firstly, the ranking may be biased because it is based on usage statistics collected from users of the Alexa toolbar. Secondly, the list is provided without any metadata such as the time of the last update. Thirdly, it is not clear under what license the list is provided. Therefore, it is not used in this thesis.

An alternative to the Alexa Top 1 Million Sites is the Majestic Million [23] which does not suffer from these limitations. The ranking of the domain names within the Majestic Million is based on the number number of subnets that refer to them, which is supposed to be more robust than just counting the number of websites that contain a link to a specific site. It is provided under a Creative Commons license allowing it to be used freely.

The domains contained in the Majestic Million list are used as a sample for an evaluation of VersionInferrer. The 500 000 most popular sites according to the Majestic Million have been scanned in the course of this thesis.

The index consists of 16 software packages (cf. Table 5.6). All versions that were published in the VCSs of the software packages at the time of the evaluation are indexed.

5.2.2 Results

Table 5.4 summarizes the results of the sample set. Failures occurred for 38 sites, which means no connection to the web server could be established or the domain name could not be resolved. Inference results could be obtained for approximately 19.5% of the sites.

Table 5.4: Number of Scanned Sites

Sites with inference result	97 273
Sites without inference result	402 689
Sites with one guessed package	97 273
Sites with more than one guessed package	0
Failed sites	38
Total sites	500 000

Table 5.5: Frequency of Counts of Different Guessed Versions

Versions guessed	**Frequency**	
0	402 689	(80.5%)
1	66 043	(13.2%)
2	6 510	(1.3%)
3	21 107	(4.2%)
4	1 260	(0.3%)
5	1 246	(0.2%)
6	456	(0.1%)
7	651	(0.1%)

The decision for a specific software package is unambiguous in the evaluated sample set, i. e., no result contains guesses for multiple different packages.

Table 5.5 shows the frequency with which different guesses occurred, i. e., between how many versions (of potentially different software packages) VersionInferrer could not differentiate further. For example, five different guesses for a site mean that VersionInferrer could only distinguish that the site is probably using one of those five software versions. For the majority of sites for which versions could be inferred (approximately 13.5%), the inferred version is unique, i. e., exactly one version was guessed. Hence, many sites have distinguishable characteristics to infer their exact version.

Table 5.6 shows on how many websites a particular software package was detected. Except for Etherpad Lite, all packages that were contained in the index during the evaluation were detected on at least one website. WordPress is the package that was detected most often with a high lead over the other software packages. This matches the market share analysis results provided by related works like the W3Techs project [29].

Table 5.7 contains information on how many sites of the sample set are likely to use vulnerable software versions. Site Results that yielded a set of multiple version guesses may contain only some vulnerable versions. Therefore, four different categories of vulnerable sites are regarded.

Table 5.6: Detected Software Packages

Package	**Using sites**	**Proportion**
WordPress	72 255	74.3%
Drupal	12 826	13.2%
Joomla! CMS™	8 578	8.8%
TYPO3	2 101	2.2%
OpenCart	632	0.6%
PrestaShop	326	0.3%
MediaWiki	136	0.1%
Moodle	135	0.1%
Contao Open Source CMS	102	0.1%
phpBB	85	0.1%
SOGo	45	$< 0.1\%$
DokuWiki	31	$< 0.1\%$
Magento Open Source	17	$< 0.1\%$
ownCloud	3	$< 0.1\%$
Nextcloud	1	$< 0.1\%$
Etherpad Lite	0	0.0%
Total	97 273	100.0%

Table 5.7: Number of Sites With Vulnerable Software Versions Detected

Most recent guess vulnerable	24 021	25.5%
Any guess vulnerable	43 347	46.0%
All guesses vulnerable	23 973	25.4%
No guess vulnerable	53 926	57.2%

1. A site is considered vulnerable if the *most recent guess* of the result is vulnerable. The most recent guess is chosen by its release date: the newest version guessed is regarded as most recent, not respecting long-term-support release branches and similar constructs that might distort the selection.
2. A site result is classified as *any guess vulnerable* if at least one of the guesses the method has inferred is vulnerable according to the CVE statistics.
3. A site is classified as *all guesses vulnerable* when all determined software version guesses contain known vulnerabilities
4. A site result is rated as *no guess vulnerable* if none of the guesses contains known vulnerabilities according to the CVE data.

For results that yield a single distinct guess, the first three categories are equivalent. For the results that did not narrow down to a guess of a single version, the *most recent guess* vulnerability category is equivalent to a best-case assumption, e. g., it is assumed that the site operator runs the most recent version of the qualifying versions.

The number of sites on which vulnerable software versions were detected is shown in Table 5.8. The vulnerability detection relies on CVE statistics collected from published data as described in Sect. 3.6. The matching of CVE records to software versions is not always accurate and the quality of the references within the CVE data differs among different software packages and different vulnerability records. Different aspects influence the amount of vulnerable versions detected on the web. For example, the Magento Open Source CMS has a very low number of CVE identifiers. All currently known CVEs of that software package affect only versions that are at least one year old.

A detailed empirical evaluation is out of scope for this thesis. Therefore, results will only be discussed for two software packages in the following. Similar results have been found in previous experiments, e. g., by Lauinger et al [16], and by Vasek et al. [38].

The highest proportion of vulnerable sites were found for the Joomla package. The majority of sites that use Joomla is using version 2.5.28, which is the final release of the 2.X branch. That release branch is no longer maintained and contains known vulnerabilities. Many site operators have not yet upgraded their websites to the 3.X release branch, thus using a vulnerable version. For DokuWiki, the number of sites where *any* guess is vulnerable is much higher than the number where *all* guesses are vulnerable. This implies that for DokuWiki, security vulnerabilities are fixed in dedicated releases that do not result in publicly visible changes of static assets, i. e., only the vulnerable part of the server-side code is changed in updates. As a consequence, VersionInferrer cannot unambiguously determine whether a site is running the vulnerable version of DokuWiki or the patched version.

5.2.3 Result Verification

To estimate the quality of the results yielded by VersionInferrer, the accuracy of the results was verified via manual inspection. However, given the sample size of 500 000 websites, it is infeasible to verify the results for all websites reliably. Instead, a subsample of 50 randomly selected sites is drawn and verified.

Table 5.8: Number of Sites With Vulnerable Software Versions Detected by Package

Package	**Vulnerable sites (guesses)**			**Using sites**
	Most recent	**Any**	**All**	
Joomla! CMS™	98.8%	98.9%	98.8%	8 578
Moodle	88.1%	92.6%	84.4%	135
MediaWiki	66.9%	86.0%	65.4%	136
TYPO3	39.9%	47.1%	39.9%	2 101
phpBB	35.3%	47.1%	0.0%	85
Drupal	28.8%	29.7%	28.7%	12 826
WordPress	14.9%	40.9%	14.9%	72 255
PrestaShop	7.7%	8.3%	7.4%	326
DokuWiki	3.2%	71.0%	3.2%	31
Contao Open Source CMS	0.0%	0.0%	0.0%	102
Magento Open Source	0.0%	0.0%	0.0%	17
Nextcloud	0.0%	0.0%	0.0%	1
OpenCart	0.0%	30.4%	0.0%	632
SOGo	0.0%	0.0%	0.0%	45
ownCloud	0.0%	0.0%	0.0%	3

Table 5.9: Results of the Manual Verification

Correct result	43
- Exact manual verification possible	35
- No exact manual verification possible	8
Wrong inference result	7
- Wrong version	6
- Wrong package	1

For manual verification, each site is retrieved with a web browser. The source code and static objects are analyzed for conclusive hints that disclose the used software package and the used version. In contrast to the automated process of VersionInferrer, manual inference is more accurate due to human intuition and experience such as knowledge about typical administrative endpoints that may disclose a package due to "typical" layout elements. The result of manual verification is then compared with the guess of VersionInferrer. Where no accurate information about the specific version could be determined, but the software package could be verified, the result is judged as correct. This was the case for eight results.

Table 5.9 shows the results of this manual verification. Only one of the 50 results that were manually verified did not match the software package that was detected by VersionInferrer. Instead, that website served a domain parking page stating that the domain has expired. As the automated scanning of the 500 000 websites took several days, it may very well be the case that the domain was still active at the time of scanning and expired only later.

Six software packages were correctly detected but with the wrong version. All of those randomly selected websites with a wrong result served a WordPress in a version that was more recent in the minor version identifier than the version detected by the automated tool. The more recent version that was actually deployed on those websites was not known to the index at the time of scanning. The deployed release was a security release of WordPress released at the time of scanning. After updating the index to include those newer versions, a repeated analysis yielded the correct version for some of these sites. Others, however, were still classified with the wrong version, or the version could not unambiguously be detected. Therefore, these versions are likely to contain only minimal changes that are not enough to differentiate between them reliably based on the publicly available site assets.

Altogether, only one of the fifty randomly selected inference results yielded a wrong software package which is likely to be no wrong result of the inference method as explained before. As shown in the table, 43 results contained the correct version that was actually in use by the analyzed websites (including the eight results where no accurate manual classification of the version was possible).

In the sample used for the verification, VersionInferrer performs well, even without tuning of the default parameters (cf. Sect. 4.5). This indicates a very small probability for false positives. Limitations of the evaluation are considered in Sect. 6.2.2.

6 Challenges and Limitations

Multiple challenges were encountered. While some of them could be solved, others are a limitation of the chosen approach. This chapter gives an overview of the most important challenges and limitations.

6.1 Challenges Solved in the Course of this Thesis

6.1.1 Bias by Generic Site Assets

Several generic libraries exist that are widely used on the web as static site assets, i. e., JavaScript libraries like jQuery [35]. Many software packages use such libraries as well. Thus, comparing such a site asset to the index database yields a lot of matching versions of different software packages. The combination of multiple such generic assets can cause false positives in VersionInferrer. However, combinations of such generic libraries might as well create a unique fingerprint of a software package.

To lower the impact of the problem of generic assets, the IDF described in Sect. 3.1 is used to weight the relevance of site assets. A site asset that is used by a lot of software packages gets a lower weight in the index than those assets that occur only rarely.

6.1.2 Index Size

A high number of software versions are indexed by VersionInferrer. This leads to a huge amount of data that needs to be stored in the database of the index, resulting in an increasing demand on storage space. Furthermore, users of VersionInferrer would rather not create the complete index themselves but instead download a precomputed index from the internet. Thus, measures need to be taken to reduce the size of the index database.

VersionInferrer uses a relational database backend by default. Each static file is stored in it only once. To store file usages, references from software versions to these static files are used. This minimizes the redundancy within the index. The resulting SQLite database yields good compression ratios that allow fast transmissions over the internet. The database used for the evaluation in Chap. 5 has a size of 230 MB. Using gzip compression, that size can be reduced to 80 MB.

6.2 Limitations

6.2.1 Limitations of VersionInferrer

Firstly, many software packages enforce authentication for a subset of their features. This often causes several static site assets to be unreferenced for anonymous users. As a result, they are hidden from VersionInferrer. However, assets known to the index are retrieved even if no reference to them is found on the website. Therefore, such assets can be used by VersionInferrer as long as the assets are served without enforcing authentication.

Secondly, dynamic site assets are generated during the runtime by the software package. Thus, they are not contained within the VCS repository. Consequently, such assets are invisible to VersionInferrer during the index construction. During the analysis, they do not provide any insights as they are unknown to the database. At the same time, a manual analysis would allow the inspection of dynamically generated content. For example, the Nextcloud software package contains a dynamic *version.php* file that contains detailed information about the used version.

Thirdly, a common way to update web software packages on websites is by uploading the contents of a new release archive to the webspace above the previous version. Files that have been removed in the new version are then kept on the webspace. If such a dangling asset is regarded by VersionInferrer, it indicates an older version than the website is actually using due to this old file.

Fourthly, VersionInferrer's parameters can be used to improve the false positive and false negative rates. However, these are tendentially contrary demands. While VersionInferrer can be configured to accept guesses earlier – thus reducing the number of false negatives –, this increases the risk that false positives are yielded, e. g., if the static files do not actually belong to the inferred software package.

Fifthly, the risk assessment of software versions is based on CVE data. However, this data is not always accurate, e. g., for some vulnerabilities, no CVE identifier might be requested. In addition, the mapping of software versions in the CVE data to software versions in the index of VersionInferrer is not always reliable. For example, the exact version string of CVE software versions can have a different format than the version identifier extracted from the VCS repository. This has an impact on the reliability of the risk assessment.

Sixthly, and last, it is easy to pretend running a specific web application without actually using it. Several methods to produce such false results are discussed in Chap. 8.

6.2.2 Limitations of the Evaluation

The evaluation in Chap. 5 uses VersionInferrer to analyze the 500 000 top sites according to the Majestic Million list. While this evaluation yields usable results, some aspects that might influence them need to be considered.

Firstly, only the homepages of the main domains provided in the Majestic Million list were analyzed. Some websites might serve a small static site on their primary domain and use a subdomain for actual contents, e. g., a webshop or a blog. Such subdomains are not analyzed during the evaluation, resulting in fewer positive results than scanning such referenced subdomains would have yielded.

Secondly, the evaluation uses a sample set of sites with high popularity. Thus, it contains a lot of sites that are operated by big companies that do not use ready-to-use software packages but develop their own software. Only a small number of middle-class sites operated by regular people is included in the sample set. Although this prevents a representative analysis of the market shares of software packages on the web, the sample set is sufficient for the evaluation of VersionInferrer whose goal is to infer information about the software used by any given website.

Thirdly, and last, the verification method of the evaluation does not allow conclusions about false negatives, because only sites for which VersionInferrer found a result were considered for the manual verification. To gain insights about false negatives, the verification could be repeated with a sample set that consists of sites for which VersionInferrer could not find a result. That verification would try to infer the used software package and version manually.

7 Ethical Considerations

The work from this thesis provides users a tool that can be used to detect technologies used by websites. Firstly, this information can easily be related to security issues they might be vulnerable to. While this is important knowledge for ordinary website users as well as people affiliated with the operation of the website, adversaries can also use this knowledge for more targeted attacks. Therefore, the insights provided by this thesis are of dual use.

To make it hard to exploit vulnerabilities of websites, no details regarding the potential vulnerabilities (e. g., CVE identifiers) are presented to the user. Instead, only the fact that there might be vulnerabilities is communicated. This gives ordinary users enough information to be warned of potential risks without providing an attacker with helpful resources for exploitation. While an adversary could patch VersionInferrer to display CVE identifiers, a manual lookup of vulnerabilities for a specific software version is possible anyway.

Secondly, the tool can potentially be used without asking the website operators for their permission to analyze their sites. Therefore it is important to ensure that the normal operation of the analyzed websites is not disturbed.

VersionInferrer generates traffic for the websites it analyzes by sending HTTP requests. However, the amount of those requests is minimized as much as possible. The load produced on the server is not greater than the load an ordinary user following a few links on the site would produce.

Running VersionInferrer against a site without having obtained the owner's permission is quite similar to scanning a site with a TLS scanner such as Qualys SSL Scan or executing a scan with PrivacyScore. Such acts are legal (in Germany) because only publicly available files are accessed and no access control mechanisms are circumvented during the process [21].

8 Prevention of Version Fingerprinting

VersionInferrer could be used by attackers to detect new targets (cf. Chap. 7). Consequently, site operators might have an incentive to harden their sites against such an automated detection. Even operators of up-to-date sites might have such a motivation since attackers may remember the software packages in use to run newly available exploits right after their availability, maybe even before official patches are released. Thus, site operators might not only want to hide the specific version, but that they are using a ready-to-use software package. While this can be criticized as being a security-by-obscurity strategy, it is an effective approach to reducing the risk of successful attacks. This chapter focuses on protective strategies for site owners to complicate the automated detection of their software packages and versions by VersionInferrer.

8.1 Removing Explicit Hints

The inference heavily relies on the first estimates. Preventing VersionInferrer from getting any first estimates reduces the probability of yielding usable results significantly. Therefore, a site reducing common package detection hints like default meta tags, software-specific headers, or typical page contents already has a rather good protection against VersionInferrer. However, this only reduces the success probability of VersionInferrer. Other methods that to not try to reduce the number of requests as much as possible could, e. g., skip the step of the first estimations and retrieve all URLs contained in the index.

8.2 Moving Static Assets to Non-Default Locations

After building first estimates from the main page and its referenced assets, known paths are retrieved for the inference to get actual guesses. If the static files are moved to non-default paths, the retrieval of the paths known to the index will not be successful. Therefore, no further information can be gained by simply retrieving the indexed paths from the webroot. Nonetheless, all static files used by the software package need to be referenced from it somewhere. Accordingly, this can only trick automated tools like VersionInferrer and does not prevent the software package and version from being inferred in a manual analysis of the website.

8.3 Merging Static Assets

Static files need to be referenced somewhere on the website in order to fulfill their function. These referenced assets can easily be investigated for inference without the need to know their paths in advance. Therefore, it is necessary to modify the static assets shipped by the software package. One approach is to concatenate all included static files of the same type into a single file (if that is not already done as a standard feature by the software package maintainers). Since the deployed files differ from the files shipped by the package maintainers, this reduces the probability of a match with indexed files, as all files are indexed separately.

8.4 Adding “Dummy” Assets

VersionInferrer determines the best guesses – which are returned as the result – based on the assets that it has retrieved. Serving several static assets that lead to a different software package than that in use can cause the tool to determine this wrong software package as best guess. This method is targeted at the specific approach of VersionInferrer and does not help against manual inference attempts.

Reflecting, there are several measures that site operators can take to harden their websites against automated inference of their used software packages and versions. While an inference using VersionInferrer can be prevented, complete protection against (manual) inference can not be achieved with sustainable effort, especially when details of a specific software package are used to infer its version. Needed assets can not be hidden reliably because they need to be referenced from the website somewhere. It is only possible to increase the number of candidates in some degree by adding further not actually used assets.

9 Future Work and Conclusion

This chapter presents avenues for future research and the concluding remarks.

9.1 Future Work

A website can consist of more than a single software package, of modified versions (“forks”) of software packages, or of custom software that is developed in-house or on behalf of the website operator. Such websites have common characteristics as well, several of them detectable using a generic approach similar to VersionInferrer. This section discusses some aspects that could be studied in future work.

9.1.1 Server-Side Frameworks

Several frameworks such as Django or Ruby on Rails provide web developers with tools and abstractions that are commonly required for the web use case. Many ready-to-use web software packages as well as customly developed websites use such frameworks.

Security vulnerabilities in such frameworks can have severe consequences as the framework code is executed on the web server that usually has access to the data of all users [18]. While the framework maintainers usually provide security updates in a timely manner, the maintainers of software packages that use such frameworks might not notice the availability of such security updates and thus fail to patch their systems. Therefore, gaining information about the frameworks that are used by websites is an important aim for improving the security and privacy on the web.

9.1.2 Client-Side Libraries

There are several open source JavaScript libraries assisting software developers with the creation of client-side code, such as jQuery [35]. They are not only used by several software packages, but also in many customly built software projects.

Vulnerabilities in client-side libraries can endanger the website user as well. New vulnerabilities in popular JavaScript libraries are found on a regular basis. While patched releases of the libraries are usually available before the end of the embargo period of such vulnerabilities, the outdated, vulnerable versions are still widely used on the web (Lauinger et al. provide a survey on this topic [16]). Hence, it is valuable to have an automated approach allowing the detection of libraries used by websites as well as their exact versions. Such an approach could take advantage of the measures for the construction of a static file index that are proposed in this thesis.

9.1.3 Dynamic Site Assets

Some software packages generate their site assets during runtime. This way, they can not be indexed in a straightforward manner as they are not included within the VCS repository. Furthermore, they depend on the environment, e. g., the tools installed on the target system, or the locale of the specific site. Thus, they can neither be added nor compared to the index using the proposed method.

Future research could elaborate a method to convert such assets into a canonical form trying to eliminate or normalize all environment-dependent contents.

9.1.4 Parameterization

The default parameters used in this thesis provide convenient results in the evaluation. However, it may be possible to increase the reliability of the results and reduce the load generated on web servers by optimizing the parameters. Future research could therefore consider a comprehensive evaluation of the impact of different parameters.

9.1.5 Content Delivery Networks

The proposed method heavily relies on static assets that are retrieved from a sites webroot. Many sites use content delivery networks (CDNs) that serve static files. These CDNs distribute the traffic among several servers at different locations, dynamically choosing a server that is near to the user. This increases the performance of websites and reduces the costs for the website operator.

When a CDN is used by a website, the static files usually can not be found on the webroot but are served from a dedicated domain of the CDN. This prevents VersionInferrer from finding any assets that provide information about the website.

Furthermore, CDNs often contain static files for several different software packages and/or versions. This makes it difficult to gain reliable information about the website even if the fact that it uses a CDN is known.

These issues could be mitigated in a future version of VersionInferrer by retrieving a target website with an instrumented browser (such as Selenium [32]) in order to observe the URLs from which assets are retrieved. This data could then be used to dynamically adapt the paths from the index.

9.1.6 Notifying Website Operators

Many websites of the scanned sample set are likely to run vulnerable software. The operators of those websites may not be aware of the resulting threats. Notifying site operators would benefit their security and ultimately the security of their customers.

Future research could perform a study on the reaction of website operators when they are notified of problems with their websites detected by VersionInferrer as well as the improvements this notification actually has, i. e., how many websites are updated upon the notification. Similar studies were already performed, e. g., by Stock et al. [34, 33].

9.2 Conclusion

Popular software packages, e. g., content management systems such as Joomla, Typo3, or Wordpress, are used on many websites. A site using outdated versions of those packages may have security vulnerabilities that endanger the privacy and security of its users. Therefore, users should be able to determine on their own what software version a website uses.

This thesis presented VersionInferrer, a tool that implements techniques for the automated inference of software versions on the web. VersionInferrer does not require specific information about software packages. It can be utilized by several users, including professional penetration testers, advanced users, and regular web users.

To gain information, VersionInferrer focuses on static files that software packages ship. An index of those static files is constructed based on VCS repositories of the software packages.

This index allows the selection of paths whose retrieval provides high entropy. The path selection uses an approach similar to decision trees, and techniques from association rule mining. Furthermore, the data of the Wappalyzer project is used to infer initial information about the used software package.

In a white-box evaluation, VersionInferrer was able to infer the correct version for 30 analyzed versions of three software packages. A black-box evaluation showed that it yields good results when used on a sample set of real-world websites. It inferred a software package and version for 19.5% of the 500 000 scanned websites. For 67.9% of these websites, the software version was distinctively inferred. A manual verification on a subsample of 50 sites revealed that the inferred version was correct for 86% of the sites, the package for 98% of them.

Using publicly available CVE data, it is possible to assess the risks that can emerge from the usage of websites that run outdated software versions. Concretely, the evaluation shows that 25.5% of the successfully analyzed sites may be subject to known vulnerabilities.

VersionInferrer is not flawless but subject to some limitations, e. g., it neither analyzes dynamic site assets, nor regards assets whose retrieval enforces authentication. Limitations of VersionInferrer and possible solutions to some of them were discussed in this thesis.

Moreover, defensive techniques and ethical issues were considered. Several methods that allow site operators to make it more difficult to infer the software packages they use were presented.

The work confirms the results of previous research that revealed outdated software on a considerable number of websites. With VersionInferrer, there is now a tool that simplifies such examinations and makes them available to ordinary users. Empowering users with tools like VersionInferrer helps to improve transparency, which in turn may create additional incentives for website operators to update their sites in a more timely manner.

Acronyms

AST abstract syntax tree

CDN content delivery network

CMS content management system

CVE common vulnerabilities and exposures

HTML hypertext markup language

HTTP hypertext transfer protocol

IDF inverse document frequency

OWASP open web application security project

TLS transport layer security

URL uniform resource locator

VCS version control system

Bibliography

[1] Elbert Alias. *Wappalyzer: Identify technology on websites.* 2017. URL: https://wappalyzer.com/ (visited on 09/23/2017).

[2] Amazon. *Alexa Top 1 Million.* 2018. URL: http://s3.amazonaws.com/alexa-static/top-1m.csv.zip (visited on 02/16/2018).

[3] Jon Louis Bentley. *Multidimensional Binary Search Trees Used for Associative Searching.* In: *Commun. ACM* 18.9 (1975), pp. 509–517.

[4] BuiltWith. *built with.* 2017. URL: https://builtwith.com/ (visited on 09/23/2017).

[5] D. Eastlake and P. Jones. *US Secure Hash Algorithm 1 (SHA1).* RFC 3174. http://www.rfc-editor.org/rfc/rfc3174.txt. RFC Editor, Sept. 2001.

[6] erwanlr. *Fingerprinter.* 2017. URL: https://github.com/erwanlr/Fingerprinter.git (visited on 09/23/2017).

[7] Django Software Foundation. *Django web framework.* 2017. URL: https://www.djangoproject.com/ (visited on 02/16/2018).

[8] Tom van Goethem et al. *Large-Scale Security Analysis of the Web: Challenges and Findings.* In: *Trust and Trustworthy Computing: 7th International Conference, TRUST 2014, Heraklion, Crete, June 30 – July 2, 2014. Proceedings.* Ed. by Thorsten Holz and Sotiris Ioannidis. Cham: Springer International Publishing, 2014, pp. 110–126.

[9] Dick Grune et al. *Modern Compiler Design.* 2nd. Springer Publishing Company, Incorporated, 2012.

[10] Antonin Guttman. *R-Trees: A Dynamic Index Structure for Spatial Searching.* In: *SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts, June 18-21, 1984.* 1984, pp. 47–57.

[11] Scott Helme. *Alexa Top 1 Million Analysis - August 2017.* 2018. URL: https://scotthelme.co.uk/alexa-top-1-million-analysis-aug-2017/ (visited on 02/15/2018).

[12] Michael Helwig. *wp_check_plugin_dir.* URL: https://github.com/mhelwig/wp_check_plugin_dir.

[13] Andrew Horton. *whatweb.* 2017. URL: https://www.morningstarsecurity.com/research/whatweb (visited on 09/23/2017).

[14] Mehmed Kantardzic. *Data Mining: Concepts, Models, Methods and Algorithms.* New York, NY, USA: John Wiley & Sons, Inc., 2002.

[15] KeyCDN. *18 Tips for Website Performance Optimization.* 2017. URL: https://www.keycdn.com/blog/website-performance-optimization/ (visited on 12/29/2017).

[16] Tobias Lauinger et al. *Thou Shalt Not Depend on Me: Analysing the Use of Outdated JavaScript Libraries on the Web.* In: *Proceedings of the Network and Distributed System Security Symposium (NDSS).* San Diego, CA, USA, 2017.

[17] Amanda Lee and Travis Atkison. *A Comparison of Fuzzy Hashes: Evaluation, Guidelines, and Future Suggestions.* In: *Proceedings of the SouthEast Conference.* ACM SE '17. Kennesaw, GA, USA: ACM, 2017, pp. 18–25.

[18] John Leyden. *Ruby off the Rails: Enormo security hole puts 240k sites at risk.* 2013. URL: https://www.theregister.co.uk/2013/01/10/ruby_on_rails_security_vuln/ (visited on 02/18/2018).

[19] Gordon Lyon. *Nmap OS Detection.* 2018. URL: https://nmap.org/book/man-os-detection.html (visited on 02/18/2018).

[20] Gordon Lyon. *Nmap Security Scanner.* 2018. URL: https://nmap.org (visited on 02/18/2018).

[21] Max Maass, Anne Laubach, and Dominik Herrman. *PrivacyScore: Analyse von Webseiten auf Sicherheits- und Privatheitsprobleme.* In: *INFORMATIK 2017.* Ed. by Maximilian Eibl and Martin Gaedke. Gesellschaft für Informatik, Bonn, 2017, pp. 1049–1060.

[22] Max Maaß et al. *PrivacyScore: Improving Privacy and Security via Crowd-Sourced Benchmarks of Websites.* In: *Privacy Technologies and Policy - 5th Annual Privacy Forum, APF 2017, Vienna, Austria, June 7-8, 2017, Revised Selected Papers.* 2017, pp. 178–191.

[23] Majestic. *The Majestic Million.* 2018. URL: https://majestic.com/reports/majestic-million (visited on 01/16/2018).

[24] Open Source Matters. *Joomla! CMS.* 2017. URL: https://www.joomla.org/ (visited on 07/16/2017).

[25] James Nadeau. *Chrome Sniffer Extension.* 2017. URL: https://github.com/jamesjnadeau/web-app-tool-sniffer (visited on 07/19/2017).

[26] Nick Nikiforakis et al. *You Are What You Include: Large-scale Evaluation of Remote Javascript Inclusions.* In: *Proceedings of the 2012 ACM Conference on Computer and Communications Security.* CCS '12. Raleigh, North Carolina, USA: ACM, 2012, pp. 736–747.

[27] OWASP. *Favicon Database.* 2017. URL: https://www.owasp.org/index.php/OWASP_favicon_database (visited on 09/23/2017).

[28] OWASP. *OWASP Top 10 2017.* URL: https://www.owasp.org/images/7/72/OWASP_Top_10-2017_%28en%29.pdf.pdf.

[29] Q-Success. *W3Techs: Web Technology Surveys.* 2017. URL: https://w3techs.com/ (visited on 09/23/2017).

[30] Stephen Robertson. *Understanding inverse document frequency: on theoretical arguments for IDF.* In: *Journal of Documentation* 60.5 (2004), pp. 503–520.

[31] Henri Salo. *Pyfiscan free web-application vulnerability and version scanner.* 2017. URL: https://github.com/fgeek/pyfiscan (visited on 09/23/2017).

[32] SeleniumHQ. *Browser Automation.* 2013. URL: http://www.seleniumhq.org/ (visited on 02/18/2018).

[33] Ben Stock et al. *Didn't You Hear Me? – Towards More Successful Web Vulnerability Notifications.* In: *Proceedings of the 25th Annual Symposium on Network and Distributed System Security (NDSS '18).* Feb. 2018.

[34] Ben Stock et al. *Hey, You Have a Problem: On the Feasibility of Large-Scale Web Vulnerability Notification.* In: *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.* 2016, pp. 1015–1032.

[35] The jQuery Foundation. *jQuery JavaScript framework.* 2017. URL: https://jquery.com/ (visited on 07/16/2017).

[36] Patrick Thomas. *BlindElephant: Web Application Fingerprinter.* 2017. URL: http://blindelephant.sourceforge.net/ (visited on 09/23/2017).

[37] TYPO3 GmbH. *TYPO3 CMS.* 2017. URL: https://typo3.org/ (visited on 07/16/2017).

[38] Marie Vasek and Tyler Moore. *Identifying Risk Factors for Webserver Compromise.* In: *Financial Cryptography and Data Security - 18th International Conference, FC 2014, Christ Church, Barbados, March 3-7, 2014, Revised Selected Papers.* 2014, pp. 326–345.

[39] Pascal Wichmann. *VersionInferrer.* 2018. URL: https://github.com/wichmannpas/webapp-version-inferrer (visited on 02/18/2018).

[40] Ivo van der Wijk. *Guess: Detect CMS, Framework, Webserver and more.* 2017. URL: http://guess.scritch.org/ (visited on 09/23/2017).

[41] WordPress. *WordPress CMS.* 2017. URL: https://wordpress.org/ (visited on 07/16/2017).

[42] Mohammed J. Zaki and Wagner Meira Jr. *Data Mining and Analysis: Fundamental Concepts and Algorithms.* Cambridge University Press, 2014.

Appendices

A Usage of the Reference Implementation

A.1 Obtaining the Software

The reference implementation can be obtained from its public VCS repository available at [39]. Furthermore, it is contained on the storage media provided with this thesis.

The software is written in Python 3. Its dependencies can be installed from the Python Package Index, which is a central repository for Python packages. Running the command `pip install -r requirements.txt` in the root of the source tree installs all required dependencies.

A.2 Creating the Index

To be able to infer the software versions that websites use, it is required to have an index of the available software packages. This can be constructed using the tools provided by the reference implementation.

The command `./update_index.py` fetches the VCSs of the upstream software packages and indexes their versions. If the command is used when the index already contains software versions, only versions that are not completely indexed are added. This allows a regular fast incremental update of the index.

A.3 Inferring the Version of a Website

The `analyze_site.py` script is a wrapper around the inference components of the reference implementation. It requires a URL of the website that should be analyzed. Furthermore, an optional `json-file` parameter specifying a JSON output file can be provided. Detailed information about the analysis results are saved to that file in the machine-readable JSON serialization. This can be used to utilize the tool in other software packages.

Figure A.1 shows the output of VersionInferrer for a WordPress installation. The first output line contains statistics of how many resources were retrieved from the web server. The second line contains the best guesses. The numbers in the brackets (*+A[a]-B[b]*) state how many assets matching that specific version were found (A), how many expected assets were not found or did not match the expected version (B), and the weighted positive and negative strengths (a, b). These weighted strengths are determined as explained in Sect. 4.4.3. When a more recent version is available, the user is warned about this fact. However, this information only regards the release date of the version and not the additional indicators described in Sect. 3.6 that are used for the vulnerability classification in the evaluation.

```
$ ./analyze_site.py http://192.168.122.42 --json-file
 ↪  wordpress_result
{'retrieved_assets_total': 9, 'retrieved_resources_total': 10,
 ↪  'retrieved_resources_successful': 9}
[<Guess 'WordPress 4.9.1 (+8[21.344485266933134]-0[0])'>]
More recent version WordPress 4.9.4 released, possibly
 ↪  outdated!
```

Figure A.1: Example Output of VersionInferrer

Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit im Bachelorstudiengang Informatik selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen – benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

Ich stimme der Einstellung der Arbeit in die Bibliothek des Fachbereichs Informatik zu.

Hamburg, den 19. Februar 2018

Pascal Wichmann