JACOBS
UNIVERSITY

Performing Service Dependency Discovery on Web Application Clients

by

Osama Zaid Saleem Al-Wardi

Bachelor Thesis in Computer Science

Submission: May 15, 2020 Supervisor: Prof. Jürgen Schönwälder

Jacobs University — Faculty of Computer Science and Electrical Engineering

English: Declaration of Authorship

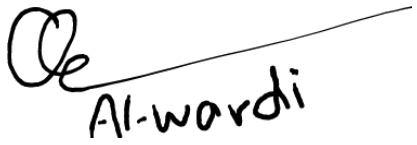
I hereby declare that the thesis submitted was created and written solely by myself without any external support. Any sources, direct or indirect, are marked as such. I am aware of the fact that the contents of the thesis in digital form may be revised with regard to usage of unauthorized aid as well as whether the whole or parts of it may be identified as plagiarism. I do agree my work to be entered into a database for it to be compared with existing sources, where it will remain in order to enable further comparisons with future theses. This does not grant any rights of reproduction and usage, however.

This document was neither presented to any other examination board nor has it been published.

German: Erklärung der Autorenschaft (Urheberschaft)

Ich erkläre hiermit, dass die vorliegende Arbeit ohne fremde Hilfe ausschließlich von mir erstellt und geschrieben worden ist. Jedwede verwendeten Quellen, direkter oder indirekter Art, sind als solche kenntlich gemacht worden. Mir ist die Tatsache bewusst, dass der Inhalt der Thesis in digitaler Form geprüft werden kann im Hinblick darauf, ob es sich ganz oder in Teilen um ein Plagiat handelt. Ich bin damit einverstanden, dass meine Arbeit in einer Datenbank eingegeben werden kann, um mit bereits bestehenden Quellen verglichen zu werden und dort auch verbleibt, um mit zukünftigen Arbeiten verglichen werden zu können. Dies berechtigt jedoch nicht zur Verwendung oder Vervielfältigung.

Diese Arbeit wurde noch keiner anderen Prüfungsbehörde vorgelegt noch wurde sie bisher veröffentlicht.

Signature: 

Date: 15.05.2020

Abstract

Software applications are becoming increasingly complex nowadays. A part of this complexity is attributed to software applications increasingly relying on a large number of external heterogeneous services. Many of these external services are essential to successfully deliver the end user with a correct service. These dependencies can get compromised and may result in serious security concerns. Therefore, understanding such dependencies is essential for the security of applications, especially in the presence of malicious attacks and service failures. Driven by these concerns, researchers have proposed techniques to discover service dependencies, giving birth to the field of service dependency discovery or SDD.

In this thesis, an overview of the state of the art in SDD will be presented. Many of the current techniques in SDD are focused on narrow scopes of applications and are sometimes ad hoc in nature. This thesis introduces a prototype of a new tool, SDDSpider, that performs SDD on the scope of web application clients without having access to any of the backend infrastructure of the web application. The tool is designed for Security Engineers, Penetration Testers, Red Teams and even System Designers to acquire an overview of the dependencies present in the application. The knowledge provided by SDDSpider is helpful in tasks such as risk assessment and attack surface evaluation.

Furthermore, to evaluate SDDSpider, a comparison between SDDSpider and two other commercial products was introduced. The comparison is designed to show the strengths and weaknesses of the techniques used by SDDSpider. The results of the comparison show that SDDSpider uses a more elaborate method of discovering dependencies in the client, which yields a higher number of dependencies. While the other tools display less data, they provide much more in terms of the analysis and categorisation of dependencies. This is because the tools are designed for slightly different purposes and target users making their intended output different. As for SDDSpider, the tool is designed to discover dependencies leaving the analysis and risk assessment tasks for experts. Therefore, success or failure is a perception that remains dependant on the specific use case of the tool.

Contents

1 Introduction

As a part of the evolution of software, the paradigm of code modularisation and reuse has emerged as a prominent theme. It's purpose is to make software development easier and faster. Nowadays, developers only focus on higher level concepts without needing to develop basic functionality from scratch. As a result of that, software rarely works in isolation [8] and even for the smallest of applications there is a web of service dependencies. The web of dependencies can evolve over time and become surprisingly complex. Acquiring, maintaining, and understanding dependency knowledge is critical for many network management and cyber defense activities [8]. These dependencies also need attention and maintenance from the developer the same as any other feature.

It is Leslie Lamport that said *"A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable"* [5]. Every software engineer nowadays is met with the challenge of maintaining the availability of dependencies and keeping them up to date. Studies show that approximately 70% of enterprise IT budgets are spent on maintenance [5]. Therefore, service dependency discovery is of major interest to enterprises and researchers alike. The research field of service dependency discovery (SDD) has emerged to find solutions to these problems.

Researchers have proposed service dependency discovery techniques by obtaining knowledge from network traffic, system logs etc. The end result of using such techniques is an overview of the dependencies discovered. This overview can be represented as a graph and it provides knowledge that is valuable to ensure the stability and security of software. Therefore, it goes without saying that creating tools to discover service dependencies could result in a large financial benefit. Unfortunately however, most solutions that are widely available are ad hoc to specific software, frameworks and environments. There is also little understanding about how well solutions work, what shortcomings they have, and whether their limitations can be overcome [9].

The contribution of this thesis is to summarise the state of the art of service dependency discovery and emphasize its effects on the security and dependability of modern systems. Furthermore, a prototype for a new tool will be presented. The tool aims discover service dependencies in web application clients without any access to backend infrastructure. The tool aims to achieve this by utilising red teaming techniques such as spidering. The strengths and weaknesses of tool are then evaluated using a comparison where the tool will be compared with two commercial products.

2 State of the Art

The discovery and analysis of dependencies between services is something that is often approached and needed in many situations in enterprises and development environments. However, there seems to be no consensus on the techniques and terminology used. The research in this topic is sparse and seems to be centered around specialised software and scenarios [9]. It is also very technique focused without laying out solid theoretical background of the topic. In this thesis, a comprehensive layout of the theory behind service dependency discovery and the state of the art will be presented.

While service dependency discovery is not widely explored, there are applications such as Sherlock [4], Dynatrace and NSDminer [14] which attempt to identify dependencies across different scopes of applications. This shows that there is a significant need for tools that help with investigating the dependencies in software with heterogeneous components. However, many of those applications assign their own different terminology and definitions to the topic. The fact that many developers in different development environments have created different solutions and gave it their own terminology signifies that we have stumbled across a topic that is highly fundamental. Therefore in the next section, the theoretical definitions of services, dependencies and how they relate to dependable systems and security will be presented.

2.1 Terminology

Service dependency discovery is the act of finding dependencies between different components of software called services. A service delivered by a system is its behaviour or function as it is perceived by its users. A user is another system that receives a service from the service provider [2].

There is a clear distinction between a service and a correct service. A correct service is delivered when the service implement the system function which is defined in the functional specification of the service [2]. The deviation of a service from the correct service is called a service failure which is an event that occurs when the correct service is not delivered. Service failures are one of the biggest threats to system dependability. Figure 1 below illustrates the tree of dependability.

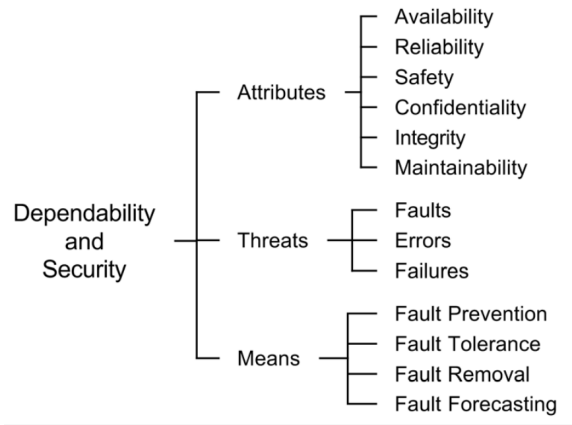


Figure 1: The Tree of Dependability and Security [2]

Dependability is the ability of a system to deliver a service that can justifiably be trusted. In other words the dependability of a system is the ability to avoid service failures that are more frequent and more severe than is acceptable [2]. Dependability has many different attributes but in our case we will be focusing on confidentiality, integrity and availability. The composite of these attributes gives the definition to security. Figure 2 shows the attributes of security and dependability accordingly. Most significantly, a dependency is a relation between services where one service depends on the other to deliver a correct service and a service failure in one can trigger a service failure in the other.

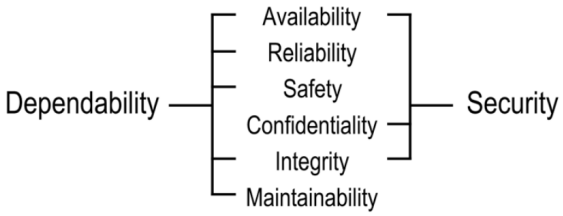


Figure 2: Dependability and Security Attributes [2]

The scope of this thesis is focused on web applications, therefore it is necessary to briefly define the term 'web application'. Web applications are usually defined by their architecture. There are many architectures to describe web applications such as the 2-tier, 3-tier, N-tier architectures and the MVC model. Despite the different types of architectures, in this thesis the 3-tier architecture will be presented. The reason of focusing on the 3-tier architecture is because it is widely accepted and simple to use while in the case of N-tier architectures many can get confused when arguing about the different tiers.

According to the 3-tier web application architecture, on the front line of a typical web application is the presentation tier [12]. This tier has three functionalities, firstly a web server that receives requests from the clients and serves web requests. Also, it forwards complex dynamic content requests to the business tier (2nd tier). The presentation tier receives responses from the business layer and presents them back to the clients [12]. All the business logic resides in the 2nd tier also called the business tier (the application server) [12]. This layer receives requests from the presentation tier, looks up information from the 3rd tier also called the data access tier (database management system) and processes the information [12]. The data access tier queries the requested data from a database file. The processed information will be passed back to the web server where it will be formatted and displayed to the client. Figure 3 is a diagram of the flow of requests and responses in the 3-tier architecture.

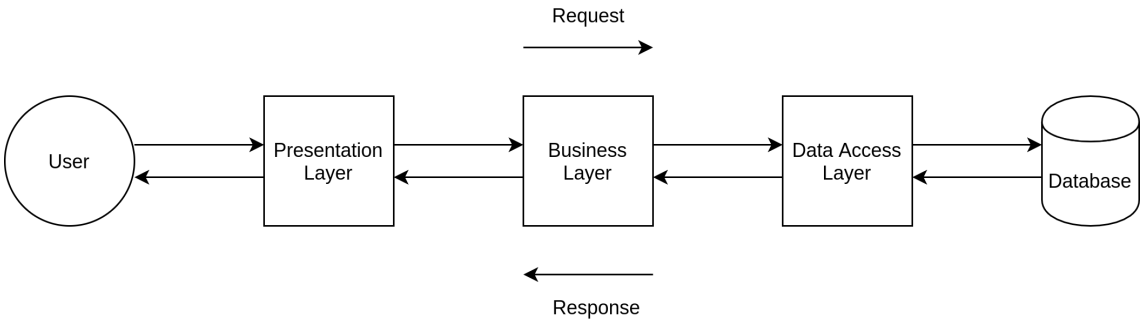


Figure 3: Requests and Responses in the 3-Tier Architecture of Web Applications

2.2 Theoretical Background

Before discussing the technical details on implementation and current solutions there are some theoretical considerations to be taken. Below are two important observations on the nature of dependencies and their properties.

2.2.1 The Dependency Relation

A dependency can be viewed as a relation between services. As this relation is examined many interesting properties can be observed. First, lets denote the dependency relation with the $\rightarrow$ operator. An example would be $A \rightarrow B$, this is pronounced as "service A depends on service B". This formalisation can be improved by modeling the dependency relation. There can be variety of different models but, it will be simply a discrete math model in this report.

Reflexivity is the first property. Lets consider a service called A. $A \rightarrow A$ is trivial, as service A depends on service A to deliver a correct service. The second property is *Transitivity*. Lets consider these three services A, B and C. If $A \rightarrow B$ and $B \rightarrow C$ then, $A \rightarrow C$. The transitivity property is important because, it has large implications on the design of tools. This is because, this property states that if a service dependency $A \rightarrow B$ is discovered, then B need to be scanned for services because there might be a transitive dependency (indirect dependency) between $A \rightarrow C$. Finally in modern systems today we face scenarios such that $A \rightarrow B|C$ for which $|$ represents the logical "or" operator. This can be present in a system as a backup in case service B is not available then A can fallback on service C. This is a common fault tolerance mechanism.

There is a similar relation in the field of distributed systems. It is the causal relation denoted by '$\prec$'. This is a relation that is fundamental to many problems occurring in distributed computing. For example, determining a consistent global snapshot of a distributed computation [15]. This relation is well defined in the model of distributed systems alongside many other concept and exploring this causal relation $\prec$ is the motivation to introduce such a model for the dependency relation $\rightarrow$. While not identical such model is invaluable when it comes to arguing about the correctness of techniques in SDD.

2.2.2 Cyclic Dependencies

Dependency cycles occur for services A and B, when $A \rightarrow B$ and $B \rightarrow A$. This creates a cycle that can be dangerous for applications and SDD tools alike. For tools, such dependencies are dangerous as the tool can get stuck in an endless cycle, continually traversing from A to B and vice versa. This causes the tool to waste system resources and not discover all dependencies present.

However, cyclic dependencies have the most dangerous consequences when it comes to repairing broken dependencies. For instance, consider the cyclic dependency between A and B, if service B is compromised then service A is compromised. This makes fixing B impossible because A is compromised and fixing A is impossible because B is compromised [10]. In fact these dependencies can be especially destructive when they involve the mechanisms used to access and modify a service. This can cause the classic dead lock scenario as the operator knows what steps to take to repair the broken service, but it is impossible to take those steps without the service [10].

2.3 Technical Background

There has been many SDD techniques introduced by researchers over the years. There are many different types of techniques but in general the techniques can be grouped into two logical categories.

The first category uses readily available information in the system to draw conclusions regarding service dependencies. [3] defines *static dependency analysis* which uses information present in various configuration files describing a distributed system, to derive dependency information. [8] defines *passive discovery* methods to observe without causing effects. While definitions might vary, in essence this category finds information available in the system that describe the configurations of the distributed system and the state of the local systems. It tends to process this data in order to derive dependency information. Below are most notable techniques and their descriptions.

Network Packet Inference solutions collect and statistically analyze transmitted packets [19]. This can be done by passively observing application traffic. There is readily available information contained in IP, TCP, and UDP headers that is enough to make a reasonable judgement on the dependencies in the system [9]. Many techniques are based on traffic delay distributions, co-occurrence analysis other metrics. There are some drawbacks to this method because, this technique only discovers dependencies between services that are communicating over IP, TCP and UDP which makes it suitable for networks but, it does not have the ability to discover services communicating via inter-process communication like signals, pipes etc.

System Log Mining approaches parse available log files in the system and calculate the correlation of service components and group the dependent ones [19]. Some approaches parse log messages into log keys and parameters, and use co-occurrence analysis of corresponding parameters to find dependent key pairs [13]. Other approaches adopted the correlation of CPU and packet flow to calculate the distance between service components. They adopted hierarchical clustering algorithms to determine which category the service components should belong to [1]. One drawback of this method, is the need for a comprehensive logging infrastructure already present in the system in order to achieve good results. There are many other approaches in this category such as source code analysis and process tracing on the OS layer by monitoring inter-process communication.

The second category addresses some of the shortcomings of the first category. *Active Dependency Discovery* (ADD) is defined to actively perturb the system components while monitoring the system's response [6]. An advantage of this category of techniques is its ability to differentiate causal relationships indicating actual resource dependencies from simple correlations in monitoring data since there is knowledge of which component is being perturbed [3]. However, the coverage of the approach is dependent on the ability to insert controlled perturbation to the different system components [3]. While this technique is considerably more successful in discovering dependencies, it is reliant on the use of appropriate faults that can effectively uncover dependencies [3].

Fault Injections are considered as an important tool in ADD for evaluating the dependability of computer systems [3]. Faults are injected into the system to study the dependability bottlenecks, the behavior of the system when a fault is introduced, and evaluate the performance impact of fault tolerance mechanisms, such as, error detection mechanisms and recovery mechanisms [3]. Fault injectors can be based on both the hardware and on

the software layer. The software fault injectors are characterised according to fault types, locations and triggers that they can support. Faults can also be permanent or transient ..etc [3].

2.4 Notable Solutions

As a part of summarising the state of the art, the most notable solutions for SDD and their descriptions must be presented.

1. Sherlock is a system that encapsulates the services and network components that applications depend on in an inference graph. This graph is combined with end-user observations of application performance to localize faults in an enterprise network [4].
2. Service Map[1] is a tool that discovers TCP-connected process in Windows and Linux virtual machines running in Azure, in third-party clouds, or on-premises in data centers.
3. Orion is a tool that discovers dependencies by passively observing application traffic. It uses readily available information contained in IP, TCP, and UDP headers without parsing higher-level application-specific protocols. It's technique is based on traffic delay distributions [9].
4. Dynatrace[2] is a software that provides application performance management, cloud infrastructure monitoring, and user experience management. As a part of the infrastructure monitoring module there is SDD functionality.
5. NSDMiner is a tool to discover network service dependencies in distributed systems. NSDMiner utilises non-intrusive techniques such as traffic co-occurrence analysis to discover dependencies in passively observed network traffic [14].
6. Wappalyzer is a utility that reveals the technologies used on websites. It detects management systems, server software, advertisement and analytics services. It works mainly by using regular expressions to inspect HTML, scripts, cookies, and headers of visited pages, looking for technology fingerprints [18].
7. Builtwith is a web application profiling tool that uses technology tracking techniques to identify analytics services, management systems, advertisement services and content delivery networks. It also uses technology fingerprinting methods [7].
8. BurpSuite[3] is a web application penetration testing framework. It utilises various tools such as an interception proxy and a web spider. It allows for mapping and analysis of the attack surface of web applications. While it's purpose is not to perform SDD, it has valuable techniques and tools that can aid the process of SDD.
9. CloudScout is a research project that introduces an approach that analyzes the correlation among service components based on the time-series information from monitoring system logs in a non-intrusive manner [19].

[1]Service Map https://docs.microsoft.com/en-us/azure/azure-monitor/insights/service-map
[2]Dynatrace Homepage https://www.dynatrace.com/
[3]BurpSuite Homepage https://portswigger.net/

Solution	Application Domain	Technique	Type
Sherlock [4]	Enterprise Networks	NPI	Research Project
Service Map	Cloud Environments	NPI	Commercial Product
Orion [9]	Enterprise Networks	NPI	Research Project
Dynatrace	Application Backends	NPI and SLM	Commercial Product
NSDminer [14]	Distributed Systems	NPI	Research Project
Cloudscout [19]	Application Backends	SLM	Research Project
Wappalyzer	Web Application Clients	Fingerprinting	Commercial Product
Builtwith	Web Application Clients	Fingerprinting	Commercial Product
BurpSuite	Web Applications	Interception and Spidering	Commercial Product

NPI = Network Packet Inference

SLM = System Log Mining

Table 1: A Summary Table of Current Solutions in Service Dependency Discovery

Most of these tools are built to perform SDD on particular scopes. Orion is mostly focused on enterprise networks [9], while service map and dynatrace are focused on application backends. The summary table 1 shows the most notable solutions and their techniques.

3 SDDSpider

The main goal of this thesis is to summerise the state of the art and prototype a tool that performs SDD on web application clients. Before discussing the potential design and implementation for the tool many assumptions need to be cleared.

Assumptions

In the previous chapter the terminology of dependability and web applications were presented. However, these definitions can be broad and they have to be further specified to be applied in a practical context. For instance, it is assumed that a web application is different from a web API. This means that web APIs are outside of the scope of the tool. This is because web APIs might require completely different SDD techniques than traditional web applications. This limitation of scope is necessary to increase focus and chances of success.

In the previous chapter a clear distinction was made between a service and a correct service. The definition of a correct service is very useful as it helps us to clearly define the dependencies to be discovered. The functional specification of a service is the basis that a correct service is judged. The functional specification is something that differs from one application to the other because, they are designed according to the intended purpose of the application. For instance, some application might consider DNS as a service dependency because, in the case of a service failure the web application will be unavailable via its uniform resource locator or URL. However, others might argue that it is not a dependency because, the web application is still available via IPv4 or IPv6. In order for a generic tool to be created, many more assumptions need to be made. Below are the most important ones:

- DNS Availability: Any external service used in the application that can affect the availability of the application from it's domain, is considered as a dependency.

- Presentation and Content: Any external service used in the application that can affect the appearance, presentation or content of the application, is considered as a dependency. This includes elements like pictures, themes, fonts and even external references.
- Caching: Any external service used to improve the performance of the application such as a content delivery network or CDN is considered as a dependency.

The meaning of the word ”external” in the assumptions above refers to any service belonging to a domain name that is different from the original domain name of the web application. Even different subdomains of the same domain name are considered external.

3.1 Design

In this chapter the design and implementation of a new tool will be presented. The tool is named SDDSpider (Sevice Dependency Discovery Spider) which is a python based tool that performs service dependency discovery on web application clients. The tool is modular and extensible. It is built to reveal dependencies regardless of the web application type. It is a tool designed for Security Engineers, Penetration Testers, Red Teams and even System Designers to acquire an overview of the dependencies present in the application. This makes it easier to evaluate and analyse those dependencies and thus make good design and security choices for their product. This subsection is to explain some of the design choices and the technique selection process behind SDDSpider.

3.1.1 Traversal Mechanism

In web application clients, the results of service dependency discovery are highly influenced by the amount of pages discovered in the client. Therefore, it is desirable to discover as many pages from the web application as possible. *Spidering* is a technique that allows to traverse (enumerate the pages and capture contents) the web application and it comes in two forms *Automated Spidering* and *User Directed Spidering*. Any type of spidering is typically a mapping exercise used by penetration testers that includes enumerating the content and functionality of the application in order to understand the application's structure and behavior.

The most known type of spidering is called *Automated Spidering*. Automated spidering works by requesting a web page, parsing it for links to other content, requesting these links, and continuing recursively until no new content is discovered [17]. Figure 4 shows a basic workflow of an automated spider[4]. Furthermore, automated spiders attempt to achieve a higher coverage by also parsing HTML forms and submitting these back to the application (either by set values or randomly). This approach generally has some drawbacks [17]:

- Unusual mechanisms for navigation such as, JavaScript navigation bars that dynamically created, are often not handled properly by this technique.
- Automated spiders typically use URLs as identifiers to unique content [17]. To avoid endlessly spidering, they recognize when content has already been requested using it's URL and therefore do not request it again. However, many applications use

[4]Diagram obtained from https://en.wikipedia.org/wiki/Web_crawler

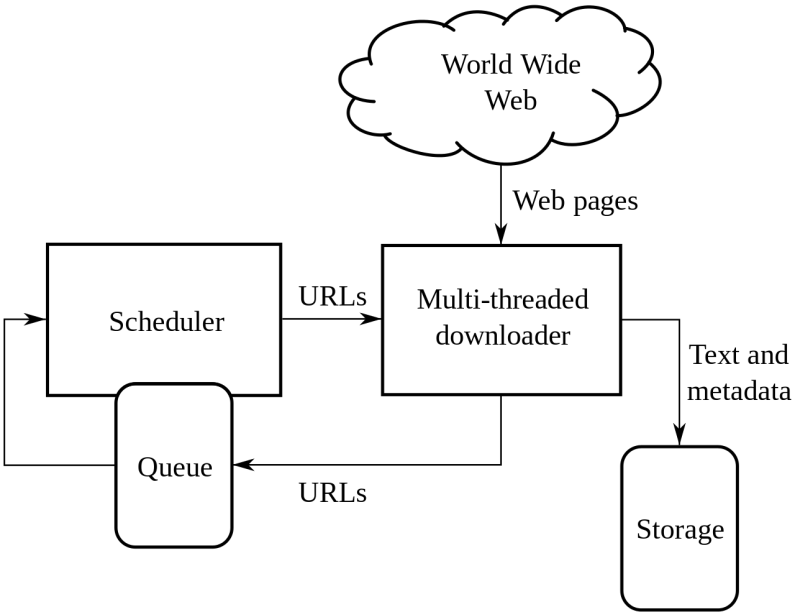


Figure 4: Basic Architecture of Automated Spider

forms-based navigation in which the same URL may return very different content and functions [17].

- Conversely to the previous point, some applications use volatile data within URLs that are not used to identify different resources such as, parameters containing timers or random number seeds [17].
- When an application uses authentication such as a login feature, an effective spider must be able to handle this to cover the functionality beyond the authentication process. Some spiders achieve this by configuring the spider manually to either have credentials or a token for an authenticated session. However, this introduces more problems such as the spider requesting the logout function, causing the session to break, and many other issues.

SDDSpider is required to perform SDD on web applications regardless of their type. The reasons stated above make automated spidering unfit for this task. A more controlled technique that is sometimes preferred to automated spidering is called *user directed spidering*. This is a better alternative as the user is allowed to browse the application in the normal way using a standard browser, attempting to navigate through all the application's functionality [17]. This can be implemented in different ways. While the user is in the process of browsing, the resulting traffic is passed to a tool (combining an intercepting proxy and a spider in the case of BurpSuite) which monitors, records visited pages and captures content. The tool builds a sitemap of the application, incorporating all the URLs visited during the spidering process. This approach offers numerous advantages such as [17]:

- Where the application uses unusual or complex mechanisms for navigation, the user can follow them using a browser in the normal way.
- The user controls all submitted data and can ensure that validation requirements are correct.

- The user can log into the application normally and ensures that the authenticated session remains active throughout the spidering process.

Therefore user directed spidering was selected instead of automated spidering.

3.1.2 Extraction Mechanism

Now that a clear traversal mechanism has been introduced, it is time to think about dependency extraction mechanisms. In order to extract dependencies from a web client the tool must perform analysis that is able to find all external hypertext references, images, scripts etc. In the early days of the internet web browsers used to simply render static HTML code which was enough to handle the functionality web applications at that time. Later as web applications became more dynamic, there was a need to introduce a programming language that is more capable of handling the dynamic nature of web applications. This language was called *JavaScript*. JavaScript is a programming language that supports event-driven, functional, and imperative programming paradigms. It has capabilities for working with text, standard data structures and most importantly the *Document Object Model* or DOM [11]. Nowadays JavaScript is used in many web applications to dynamically generate content and update the DOM without needing to request another static page or refresh the current page.

The presence of JavaScript in a web application makes the problem of dependency extraction not trivial. It is easy to extract external hypertext references, images and scripts from HTML however, trying to extract dependencies from JavaScript that has not yet been executed is hard. One idea of SDDSpider is to focus on the DOM instead of focusing on language components like HTML and JavaScript. The DOM represents a document with a logical tree wherein each node is an object representing a part of the document [11]. In web browsers the browser downloads the client side code executes it and creates the DOM tree, with the topmost node named as ”Document object” [11]. Figure 5 shows an example of a basic DOM tree of a web page. The page content is stored in the DOM and may be accessed and manipulated via JavaScript. This means that by interacting only with the DOM, it is possible to extract dependencies without needing to execute JavaScript.

After a thorough examination of the different possibilities, two technologies standout that allow for the analysis and manipulation of the DOM, *browser extensions* and *browser drivers*. Browser extensions are small software modules that allow for customizing functionality on a web browser[5]. The extension code (usually written in JavaScript) runs within the browser and has access to various browser functionality including the DOM. While it is possible to use browser extensions to implement user directed spidering and dependency extraction mechanisms they were not selected to be used in SDDSpider. It was observed that browser extensions have the following disadvantages:

- Browser extensions are usually tasked with extending features to a website or removing unwanted features such as, pop-up ads. Therefore, creating a browser extension that monitors and captures an arbitrary number of pages and dependencies can affect the performance of the browser, especially when there are also other extensions running in the browser.
- Browser extensions work within the browser sandbox. While possible, it can make

[5]Definition obtained from https://developer.chrome.com/extensions

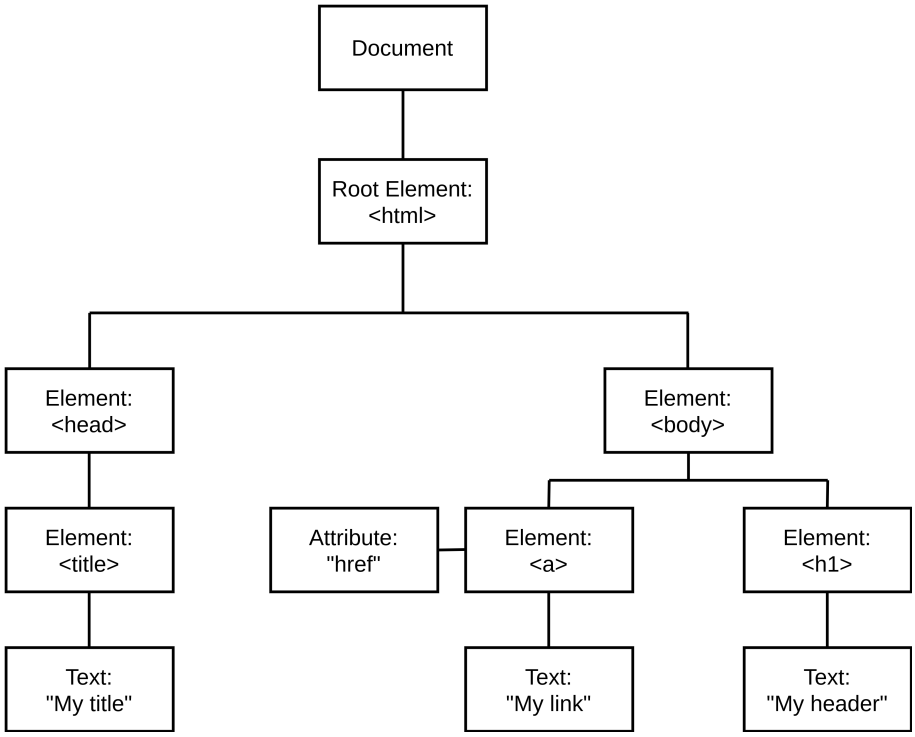


Figure 5: An Example of the Document Object Model [11]

it more difficult to extend the tool with other tools that do not run in the sandbox.

- Other penetration testing tools do not have a browser extension interface. They usually have a command line interface or desktop app interface, therefore it is beneficial to select a similar interface to already existing penetration testing tools.

Browser drivers are frameworks that allow for the control of browser functionality programmatically, usually for the purpose of testing web applications[6]. Browser drivers allow for the control of almost all browser functionality including the DOM. It is also possible to use them to implement a user directed spider. While being equivalent to browser extensions in terms of the DOM functionality, browser drivers offer multiple advantages:

- Browser drivers run in a process separate from the browser instance. They both interact with each other but the tasks performed by the browser driver will not slowdown the browser as much as a browser extension would.
- Browser drivers run on the host operating system. They are not limited by the browser sandbox and can easily work together with other tools installed.
- Browser drivers allow to develop the tool with various interfaces that follow the trends of other penetration testing tools.

Therefore, browser drivers were selected to be the main technology used in SDDSpider.

Other than spidering the application, there are other services that web applications depend on that are not found in the DOM. Dependency knowledge can also be obtained from public information in protocols and internet records. In web applications such dependency knowledge can reside in a variety of protocols. SDDSpider will examine infor-

[6]Definition obtained from https://chromedriver.chromium.org/home

mation that resides in HTTP, IP and DNS protocols. In HTTP communication headers, application servers sometimes display which server software that is currently used in the web application. This is a service that the web application client depend on to function correctly. Inspecting registration and routing data can reveal what service is used to host the web application which is a highly important dependency. Authoritative name servers that resolve DNS queries for the domain name of the web application is also a dependency. Therefore, SDDSpider contains one module to inspect HTTP responses with hopes of finding more information about the server hosting the application. SDDSpider also has another module that utilises the RIPEstat API [7] to obtain more information regarding the IP and DNS of the web application. In the next section all implementation and programming details of the previously mentioned modules will be discussed.

3.2 Implementation

In this section the implementation details of SDDSpider will be discussed. SDDSpider has three main modules ”spider.py”, ”requestor.py” and ”ripe.py”. Figure 6 shows the general architecture and workflow of SDDSpider's modules. The tool takes the URL of the web application as the input. The tool (the spider module) then displays a browser instance where the user can browse the web application. After the browsing is completed the results will be saved in JSON [8] format. The specific information stored in each file will be discussed in the section of their respective module. All the programming aspects of creating these modules will be discussed in this section.

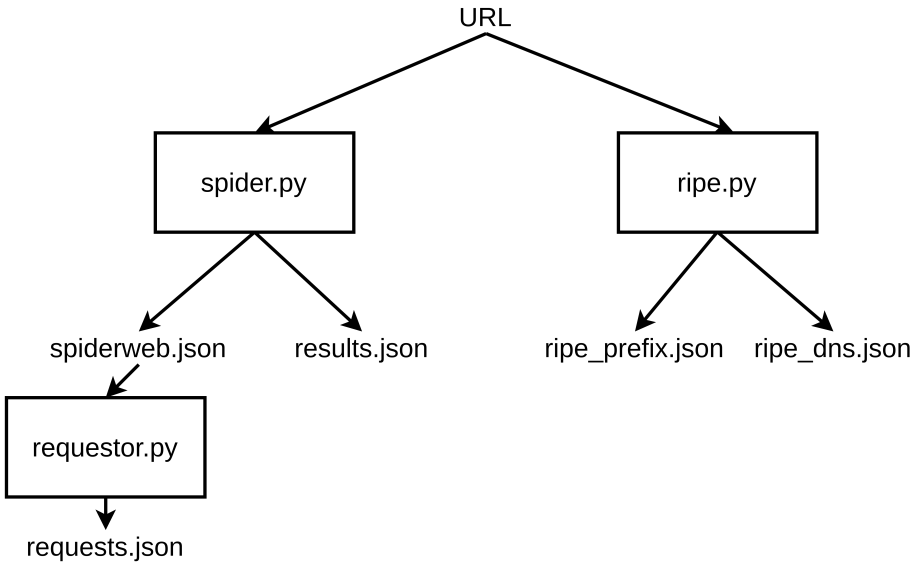


Figure 6: The Architecture of SDDSpider

3.2.1 Selenium

The spider module is responsible for user directed spidering and extracting the dependencies from the DOM. It utilizes a browser driver called *Selenium WebDriver* [16] which is implemented through a browser-specific driver, which sends commands to the browser and retrieves results. In the case of SDDSpider the browser-specific driver is ChromeDriver

[7] RIPEstat API Homepage https://stat.ripe.net/docs/data_api
[8] JavaScript Object Notation https://www.json.org/json-en.html

(for Google Chrome) which is a separate executable that the Selenium WebDriver uses to control Chrome. The Selenium client API has many useful functionalities that allow for implementing a user directed spider and for analysing the DOM tree. For instance, listing 1 shows how hypertext references, images and scripts can be extracted from the DOM tree. Listing 2 shows another example of how the user directed spider can record how many pages have been visited and how to create a sitemap of the spidered application.

Listing 1: Code Snippet from the Spider Module

```
# Finding all Hypertext References, Images and Scripts without duplicates
element = browser.find_elements_by_xpath('//*[@href]')
img = browser.find_elements_by_tag_name('img')
script = browser.find_elements_by_tag_name('script')

for i in element:
    if i.get_attribute('href') not in links:
        links.append(i.get_attribute('href'))

for i in img:
    if i.get_attribute('src') not in links:
        links.append(i.get_attribute('src'))

for i in script:
    if i.get_attribute('src') not in links:
        links.append(i.get_attribute('src'))
```

Selenium has different types of selectors that allow for the extraction of links from the DOM. Selectors are able to find elements by the Xpath[9], tag name etc. These selectors return a list of elements found in the DOM with all their attributes which is very convent to process and save.

Selenium also has features that allow for user directed spidering. Listing 2 shows another example of how the user directed spider can record how many pages have been visited and how to create a sitemap of the spidered application. This code is always working in the background as the browser is launched.

Listing 2: Code Snippet from the Spider Module

```
spiderweb = []
browser.get(base_url) # launch the browser window with base_url
current_url = base_url
# Starting Spider
while True:
    # Checking if new page was opened
    if browser.current_url != current_url:
        current_url = browser.current_url
        spiderweb.append(current_url) # add page to the sitemap

    # INSERT DEPENDENCY EXTRACTION CODE HERE

browser.quit()
```

When the user closes the browser window SDDspider will save all the dependencies

[9]XML Path Language https://en.wikipedia.org/wiki/XPath

extracted in ”resources.json”. This file saves the URL of each page visited in the web application as a key and a list of the URLs extracted of that particular page as value. The URLs in ”resources.json” are parsed and separated into internal and external URLs using pythons urllib[10]. External URLs are ones that do not belong to the same domain of the original input URL. The sitemap is saved in ”spiderweb.json”. This file contains the exact browsing steps that the user took. For example, the first page visited will have the key ”1” and the value is the URL of the web page.

Listing 3: Code Snippet from the Spider Module

```
external = [] # List for external URLs
internal = [] # List for internal URLs
base_url = urlparse(base_url)[1] # Parsing the input URL and saving it's domain
for page in resources: # Enumerating through each page discovered
    for url in resources[page]: # Enumerating through the URLs found in the page
        url_tokens = urlparse(url) # Tokenising the URL
        if url_tokens[1] != base_url:
            external.append(str(url_tokens[1]+url_tokens[2]))
        else:
            internal.append(str(url_tokens[2]))
```

3.2.2 Requestor

After spidering the application, HTTP communication needs to be investigated. In web applications HTTP response headers such as the Server and X-Powered-By headers can reveal service dependencies regarding the document and the server. For instance, consider listing 3, it is possible to see that the web application is using an Apache server version 2.2.14. The requestor module performs a GET request to each URL of pages visited and checks if the response headers contain the ”Server” or ”X-Powered-By” headers. If such headers are found they are saved in the ”requests.json” file.

Listing 4: Typical HTTP Response

```
HTTP/1.1 200 OK
Date: Sun, 18 Oct 2009 08:56:53 GMT
Server: Apache/2.2.14 (Win32)
Last-Modified: Sat, 20 Nov 2004 07:16:26 GMT
ETag: "10000000565a5-2c-3e94b66c2e680"
Accept-Ranges: bytes
Content-Length: 44
Connection: close
Content-Type: text/html
X-Pad: avoid browser bug

<html><body><h1>Hello World!</h1></body></html>
```

3.2.3 RIPEstat API

RIPEstat API is utilised by the ”ripe.py” module. RIPEstat provides open source information about specific IP addresses and prefixes, ASNs, and related information for hostnames. This includes registration, routing and DNS data including data from external

[10]Urllib https://docs.python.org/3/library/urllib.parse.html#module-urllib.parse

sources, such as Regional Internet Registries and IANA[11]. There are two data calls made by SDDspider to the RIPEstat API, one is for a prefix overview of the web application and one is for a DNS overview. Prefix overview is a data call that gives a summary of the given prefix. SDDSpider is interested in the ”holder” information as it reveals the hosting service of the web application. Below is an example of a prefix overview query and response from the RIPEstat API[12].

Listing 5: Simple Query and Response from the RIPEstat API

```
https://stat.ripe.net/data/prefix-overview/data.json?resource=151.101.128.67
```


```
{
  "data": {
    "query_time": "2020-05-01T16:00:00",
    "is_less_specific": true,
    "resource": "151.101.128.0/22",
    "actual_num_related": 1,
    "num_filtered_out": 0,
    "asns": [
        {
            "holder": "FASTLY",
            "asn": 54113
        }
    ],
    "block": {
        "resource": "151.0.0.0/8",
        "name": "IANA IPv4 Address Space Registry",
        "desc": "Administered by RIPE NCC"
    },
    "related_prefixes": [
        "151.101.0.0/16"
    ],
    "type": "prefix",
    "announced": true
}
```


There is one required parameter in the query which is called ”resource”. This parameter states the prefix or IP of the resource to be queried. The DNS chain data call follows the same format and returns the recursive chain of forward records starting from the domain name of the application. SDDSpider is interested in the list of authoritative name servers returned by this data call.

3.2.4 Usage and Interface

SDDspider is a tool developed on Ubuntu 18.04 and has a command line interface (CLI). Since a URL of the web application is the main input to SDDSpider's modules, it is a required argument for SDDSpider. Other arguments allow for enabling different modes such as verbose mode with the ”-v” argument or allow the user to set an output directory with the ”-o” argument. The requirements to run the tool correctly are all listed in the documentation provided with source code.

[11] RIPEstat Homepage https://www.ripe.net/analyse/statistics
[12] RIPEstat API Documentation https://stat.ripe.net/docs/data_api

4 Experiment

After creating a new solution, it is standard procedure in computer science to create an experimental setup for the purpose of evaluation. The experimental setup must be designed to test for metrics that are display the strengths and weaknesses of SDDSpider. An experiment can also be created to test the correctness and effectiveness of the techniques used against different types of web applications. Therefore in this chapter an experiment is going to be conducted as comparison between SDDSpider and two other commercial products.

4.1 Setup

The experiment is setup to compare SDDSpider, Wappalyzer [18] and Builtwith [7]. Wappalyzer and Builtwith (both discussed in section 2.4) are two commercial products that utilise a variety of techniques that can be compared to SDDSpider. All of the tools perform their tasks on web application clients without having access to the backend infrastructure of the web application. Wappalyzer and Builtwith have different frontend interfaces (browser extension and web application), the purpose of their frontend is to pass the URL of the targeted web application to their backend infrastructure where most of the tasks are done.

Wappalyzer is a browser extension that reveals technologies used in web applications such as content management systems, advertisement services, marketing tools and analytics services. It inspects HTML code, as well as JavaScript variables and response headers, looking for technology fingerprints. More specifcally, Wappalyser uses a long list of regular expressions which is used to identify technologies on web pages[13]. Wappalyzer has fingerprints of 1,264 technologies and places them across 61 categories[14]. The output of the extension provides a brief general overview which is displayed in the extension window inside the browser.

Builtwith is a web application profiling tool that uses technology tracking techniques to identify widgets, analytics services, content management systems, advertisers, content delivery networks etc. Builtwith states that ”every website gives off signals that they are using a particular technology. We can track these signals and determine if a site is using a particular web technology from it”[15]. Builtwith performs full index data look ups which is described by them as ”render the full website, follow redirects, execute JavaScript, load Iframes and download all the resources that make up a page”[16]. According to Builtwith, this technique allows to uncover more technologies and provide more accurate results such as uncovering advertising technologies, tag managers and custom installs. Builtwith is interested in usage analytics for the internet and they tracking over 32,000 web technologies[17] but their methods for analysing and categorising their data is not explicitly described. The output of Builtwith provides a more detailed overview than Wappalyzer and is delivered in a webpage.

Both these commercial products have free versions with limited functionality that allow for conducting a small experiment with them. The Wappalyzer free browser extension and

[13] Wappalyzer Specification https://www.wappalyzer.com/docs/dev/specification#specification
[14] Technologies tracked by Wappalyzer https://www.wappalyzer.com/technologies
[15] Builtwith FAQs https://builtwith.com/faq
[16] Builtwith Full Index https://blog.builtwith.com/2013/01/09/introducing-builtwith-full-index/
[17] Builtwith Technologies https://blog.builtwith.com/2019/05/21/highlight-technologies/

the Builtwith free technology profile lookup were selected. In order to compare the tools, they must be tested on a variety of web applications. The web applications must be of different nature and different contents. On this basis, five web applications were selected for testing. Table 2 shows the selected web applications and their different features.

Web Application	Features
Facebook	Varied Content (Text, Videos, Images), Log in, Profiles
Youtube	Video Based Content, Advertisements, Recommendations
Amazon	E-commerce, Payment System, Basket, Products, Log in
Wikipedia	Text Based Content, References, Languages
Imgur	Image Based Content

Table 2: Web Applications Selected for the Experiment

Since the tools have different different interfaces and workflows, a clear testing procedure must be defined. Builtwith and Wappalyzer use a URL as an input and they return the dependencies of the given URL. This means that they do not cover multiple pages of the web application and are capable of only analysing one page at a time. SDDSpider does utilise user directed spidering which allows for the capture and analysis of multiple pages of the web application. Since the purpose of this experiment is compare the number of dependencies discovers, it is more fair to test the tools on the same number of pages in each web application. Therefore, all the pages visited by SDDSpider will also be tested by the other tools.

In addition, the tools output the results in different formats such as JSON files in the case of SDDSpider, HTML pages for Builtwith or simply render the results in the browser extension window in the case of Wappalyzer. First, the output of these tools will be saved in their native format and then manually transformed and placed in an xlsx (Excel) format which is more convenient to process the raw results.

After collecting the results and saving them into the appropriate format, the next step will be the comparison. The purpose of the comparison is to see the contrasts between the techniques and interfaces of tools to determine which dependency overview is more suitable for penetration testing. The comparison is going to be conducted according to criterion. A numerical comparison will be made where the number of dependencies between the three tools will be directly compared. The results of this comparison will show how many dependencies are found per web application which provides many insights. It can show if a certain tool is more successful with a specific type of web application. Another result of the comparison can show how the features of different web applications influence the total number of dependencies observed. In the next chapter the raw and processed results of the comparison will be discussed.

4.2 Results

The raw results of this experiment are in the attachments of this thesis where each dependency is listed with a description. Table 3 shows the numerical comparison across all the different tools. Table 3 shows that SDDSpider has the highest number of dependencies discovered, second comes Builtwith and last comes Wappalyzer. It can be observed from table 3 that the numbers of total dependencies between the three tools are very different from each other.

Tool	Web Applications	Number of Dependencies
SDDSpider	Facebook	46
	Imgur	39
	Youtube	16
	Amazon	66
	Wikipedia	399
Builtwith	Facebook	16
	Imgur	6
	Youtube	11
	Amazon	50
	Wikipedia	14
Wappalyzer	Facebook	1
	Imgur	5
	Youtube	2
	Amazon	3
	Wikipedia	2

Table 3: Raw Results Table

Figure 7 illustrates the results from table 3 in a bar graph format. Looking at figure 7 it can be clearly seen that the dependencies discovered by SDDSpider surpass the other tools in all web applications. The variance in numbers can be especially seen in the case of the Wikipedia web application. Figure 7 shows that SDDSpider was able to discover 399 dependencies while Builtwith discovered 14 and Wappalyzer discovered only 2.

There are many reasons that can justify this high variance in numbers. First, the tools have different criterion to what they report as a dependency. In the case of Wikipedia, there were many external references and subdomains such as ”de.wikipedia.org” which are considered by SDDSpider as a dependency. This goes back to the parsing process employed by SDDSpider as those subdomains are compared with the original input domain ”en.wikipedia.org” and consequently get reported as a dependency. Builtwith parsed those subdomains as well and also reported that there were dependencies but not as much as SDDSpider. It seems that Wappalyzer does not report such subdomains and does have not such fingerprints in it’s database. It is an assumption (section 3) of SDDSpider that embedded URLs and external references are considered dependencies as they can be of importance depending of the functional specification of the web application.

SDDSpider analyses the DOM tree and allows the user to interact freely with the web application. This can yield more dependencies as the user activates parts of the web application that do not get activated by the automated approach of the other tools. For instance, Builtwith reported ”Problem with Lookup: Service Unavailable” for certain parts of Facebook and Amazon because they required authentication. The user also has the ability to scroll and load more content contributing to the number of dependencies found by SDDSpider.

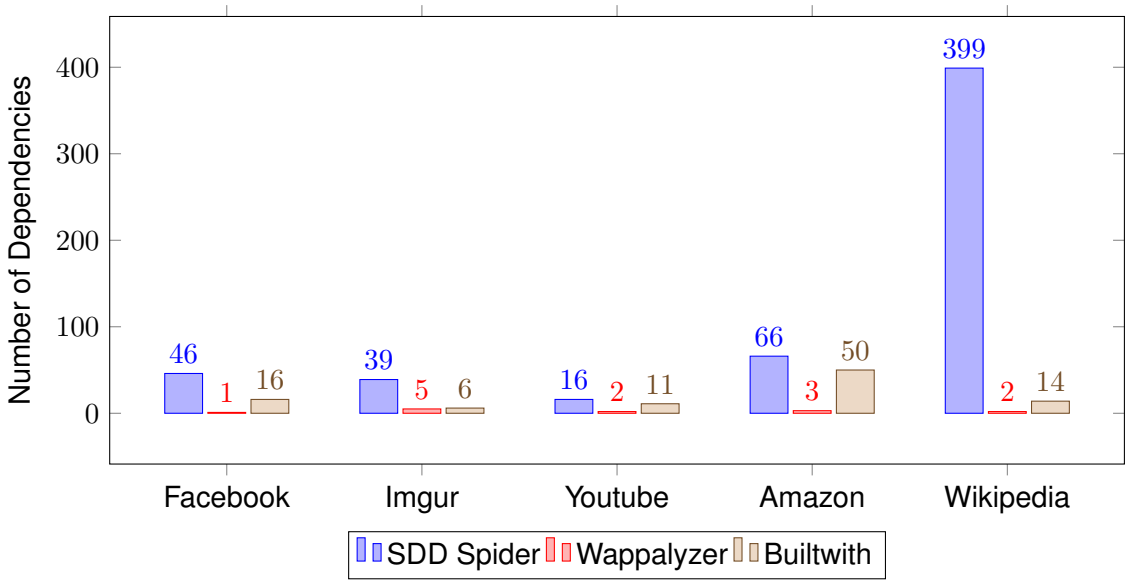


Figure 7: Numerical Comparison with Web Applications

Figure 7 and table 3 showed the raw numerical comparison. In order to understand the nature of dependencies discovered, it would be better to categorise them. In the course of this experiment it can be observed that many dependencies were of a similar type. Table 4 explains the dependency categories and their description. The categories group dependencies with a similar type. Categories were reported in the output of Builtwith and Wappalyzer but not by SDDSpider. The dependencies of SDDSpider were categorised manually.

Category	Description
Content Servers	Any external servers that load content such as images ..etc.
External APIs	Any external scripts that are run by the web application e.g fonts.
External References	Any external reference to an external resource such as an image.
Hosting Service	Any dependency related to the hosting service or server.
DNS	Any DNS dependency such as Nameservers or DNS service providers.
CNDs	Any Content Delivery Networks or caching services.
Advertisment Services	Any external service for Advertisements.
Analyitcs Services	Any external service that provides Analytics.
Other	Any that does not belong to any category such as redirect links ..etc.

Table 4: Dependency Categories

Table 5 shows a more detailed picture as it shows the categories with the raw results.

4.3 Additional Observations, Shortcomings and Future Improvements

While conducting the experiment and looking at the results there were some interesting observations:

- Wappalyzer discovered most of it's dependencies from the Imgur web application. This can explained as Imgur is a web application that utilises many external APIs, advertisement and analytics services that Wappalyser had existing fingerprints for.

Tool	Website	CS	E APIs	E REFs	HS	DNS	CDNs	ADS	AS	Other	Total
SDDSpider	Facebook	0	0	4	1	8	3	0	0	30	46
	Imgur	2	5	9	1	4	3	9	6	0	39
	Youtube	3	4	0	1	4	0	4	0	0	16
	Amazon	3	3	14	2	36	0	8	0	0	66
	Wikipedia	3	0	106	2	3	0	2	0	283	399
											566
Builtwith	Facebook	0	2	0	0	0	2	2	3	7	16
	Imgur	0	1	0	0	0	2	0	2	1	6
	Youtube	0	4	0	2	0	1	1	1	2	11
	Amazon	0	12	0	4	2	3	22	5	2	50
	Wikipedia	0	0	0	1	0	0	1	0	12	14
											97
Wappalizer	Facebook	0	0	0	0	0	0	0	0	1	1
	Imgur	0	2	0	0	0	0	1	2	0	5
	Youtube	0	1	0	0	0	0	1	0	0	2
	Amazon	0	0	0	1	0	1	0	0	1	3
	Wikipedia	0	0	0	1	0	0	0	0	1	2
											13

CS = Content Server | E APIs = External APIs | E REFs = External References

HS = Hosting Service | CNDs = Content Delivery Networks | ADS = Advertisement Services | AS = Analytics Services

Table 5: Raw Results Table with Categories

- SDDSpider discovered more DNS dependencies while Builtwith discovered only two and Wappalyzer discovered none. Wappalyzer did not discover any DNS dependencies because, it uses regular expressions to find technology fingerprints in the source or the headers of the website and does not go beyond those. Builtwith however, did display some DNS service providers but only for Amazon. SDDSpider uses the RIPEstat lookups to fetch the authoritative name servers of the domain. They are dependencies as they are the servers that perform the resolution of domain names into IP addresses.
- SDDSpider was the only tool to discover external domains that are used to fetch contents to be rendered inline (content servers). Youtube utilises ”s.ytimg.com” and ”i.ytimg.com” to render the Youtube logo and video thumbnails inline. The reason the other tools did not report those domains is because they did not have fingerprints that match it.

There are also certain drawbacks to the experimental setup that can be improved in future experiments:

- This experiment had a sample size of five web applications. Testing five web applications can be limiting especially when trying to measure the effects of the different techniques on different types of web applications. Therefore in the future a bigger sample size would be a good improvement.
- The experiment was conducted using Builtwith’s and Wappalyzer’s free functionality. There exists premium versions that have extended functionalities including advanced lookups with more detailed information. When an experiment is conducted in the future, utilising the premium versions might yield different results.

- SDDSpider does not offer any categorisation or analysis of the dependencies but the other tools do so. This caused the dependencies found by SDDSpider to be manually categorised. Therefore, a future extension that adds an automated data aggregation module is necessary.
- In this experiment the tools compared were designed for different purposes and had different interfaces. SDDSpider is a command line interface tool. Builtwith and Wappalyzer however, have very different interfaces (a browser extension for Wappalyzer and a web application for Builtwith) which made collecting and processing data more difficult. SDDSpider provides more raw data and the other tools provide more aggregated data. It is more interesting to compare SDDSpider with BurpSuite (mentioned in section 2.4) in the future. They are both penetration testing tools that analyse web application clients with no access to backend infrastructure. Also they both perform user directed spidering with slight differences in the implementation. These two tools would create a better future comparison.

5 Conclusion

In conclusion, this thesis furthers the state of the art by introducing a new tool, SDDSpider, that performs service dependency discovery. The tool discovers the dependencies on web application clients without access to any of the backend infrastructure. The discovery process is done through the use of user directed spidering. The tool also utilises browser drivers to read the contents of the DOM tree of the browser and extract dependencies. RIPEstat API was used to provide more information regarding the web application such as IP and prefix information. The tool was designed, implemented and later compared to other commercial tools. An experimental setup was created to evaluate the strength and weaknesses of the techniques used. A comparison between different was created and the tools were tested on five web applications of different contents and features.

The results of the experiment conducted in this thesis give an overview on the strengths and weaknesses of the approaches used by SDDSpider, Wappalyzer and Builtwith. The main weakness of SDDSpider is it's manual approach. SDDSpider collects each page of the web application as the user performs user directed spidering. This is a compromise to collect as many pages of the web application as possible in order to maximize the amount of dependencies to be discovered. The other tools analysed only one page at a time but they did so completely autonomously. Also, SDDSpider does little to no dependency analysis or processing of raw data and leaves such tasks to be done manually by experts. The other tools however, had data aggregation functionalities that are necessary to process the raw data into information about the software industry and internet trends etc. Therefore the numbers displayed in the results of the experiment does not indicate that SDDSpider is the clear winner of the comparison. The strength of SDDSpider is it's approach of activating most of the web application functionality to expose as many dependencies as possible while sacrificing some autonomy. The numerical comparison plays well in the side of an approach that produces higher numbers.

Generally, the reason for the difference in the results stems from the different purposes and goals the tool were designed to achieve. Builtwith is a tool interested in web technologies and is used as a business intelligence tool providing technology adoption and usage analytics for the internet. Wappalyzer is interested in technographic datasets that

provide insights into the software industry, market research and competitor analysis. This means that both Wappalyzer and Builtwith are more interested in data gathered over a very large sample of web applications and it is necessary for them to aggregate and process the raw data into information about the software industry and internet trends etc. However, SDDSpider is a tool to be used by Security Engineers and System Designers to acquire an overview regarding the service dependencies present in the web application. This overview can be used for better risk assessment and more secure design. Therefore, the success and failure of the tools compared is dependant on the scenario they are used for. For security and penetration testing verbose results take priority to a general overview which makes SDDSpider more suitable. However, for data analytics and internet trends, Builtwith and Wappalyzer provide a better data anaysis and a more suitable interface.

References

[1] R. Apte, L. Hu, K. Schwan, and A. Ghosh. "Look Who's Talking: Discovering Dependencies between Virtual Machines Using CPU Utilization". In: *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*. HotCloud'10. USENIX Association, 2010, p. 17.

[2] A. Avizienis, J. -. Laprie, B. Randell, and C. Landwehr. "Basic concepts and taxonomy of dependable and secure computing". In: *IEEE Transactions on Dependable and Secure Computing* 1.1 (2004), pp. 11–33.

[3] S. Bagchi, G. Kar, and J. Hellerstein. "Dependency Analysis in Distributed Systems using Fault Injection: Application to Problem Determination in an e-commerce Environment". In: *12th International Workshop on Distributed Systems: Operations & Management*. 2001, pp. 151–164.

[4] P. Bahl, R. Chandra, A. Greenberg, S. Kandula, D.A. Maltz, and M. Zhang. "Towards Highly Reliable Enterprise Network Services via Inference of Multi-Level Dependencies". In: *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. SIGCOMM '07. Association for Computing Machinery, 2007, pp. 13–24.

[5] V. Bahl, P. Barham, R. Black, R. Chandra, M. Goldszmidt, R. Isaacs, S. Kandula, L. Li, J. MacCormick, D. Maltz, R. Mortier, M. Wawrzoniak, and M. Zhang. "Discovering Dependencies for Network Management". In: *Workshop on Hot Topics in Networks (HotNets-V)*. Association for Computing Machinery, Inc., 2006.

[6] A. Brown, G. Kar, and A. Keller. "An active approach to characterizing dynamic dependencies for problem determination in a distributed environment". In: *2001 IEEE/IFIP International Symposium on Integrated Network Management Proceedings. Integrated Network Management VII. Integrated Management Strategies for the New Millennium (Cat. No.01EX470)*. 2001, pp. 377–390.

[7] *Builtwith Homepage*. https://builtwith.com/.

[8] T. E. Carroll, S. Chikkagoudar, and K. Arthur-Durett. "Impact of network activity levels on the performance of passive network service dependency discovery". In: *MILCOM 2015 - 2015 IEEE Military Communications Conference*. 2015, pp. 1341–1347.

[9] X. Chen, M. Zhang, Z.M. Mao, and P. Bahl. “Automating Network Application Dependency Discovery: Experiences, Limitations, and New Solutions”. In: *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*. USENIX Association, 2008, pp. 117–130.

[10] S. Esparrachiari, T. Reilly, and A. Rentz. “Tracking and Controlling Microservice Dependencies”. In: *Queue* 16.4 (Aug. 2018).

[11] Matt Frisbie. *Professional JavaScript for Web Developers*. 4th. Wrox Press, 2019. Chap. 14, pp. 491–514.

[12] X. Liu, J. Heo, and L. Sha. “Modeling 3-tiered Web applications”. In: *13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*. 2005, pp. 307–310.

[13] J.G. Lou, Q. Fu, Y. Wang, and J. Li. “Mining Dependency in Distributed Systems through unstructured log analysis”. In: *the 2nd Workshop on Analysis of System Logs (selected as BEST PAPER and published in SIGOPS OS Review)*. Association for Computing Machinery, Inc., 2009.

[14] A. Natarajan, Peng Ning, Yao Liu, S. Jajodia, and S. E. Hutchinson. “NSDMiner: Automated discovery of Network Service Dependencies”. In: *2012 Proceedings IEEE INFOCOM*. 2012, pp. 2507–2515.

[15] Reinhard Schwarz and Friedemann Mattern. “Detecting causal relationships in distributed computations: In search of the holy grail”. In: *Distributed Computing* 7.3 (1994), pp. 149–174.

[16] *Selenium Project Homepage*. https://www.selenium.dev/projects/.

[17] Dafydd Stuttard and Marcus Pinto. *The web application hackers handbook: discovering and exploiting security flaws*. Books on Demand, 2018.

[18] *Wappalyzer Homepage*. https://www.wappalyzer.com/.

[19] J. Yin, X. Zhao, Y. Tang, C. Zhi, Z. Chen, and Z. Wu. “CloudScout: A Non-Intrusive Approach to Service Dependency Discovery”. In: *IEEE Transactions on Parallel and Distributed Systems* 28.5 (2017), pp. 1271–1284.