WINPEN: A FAST CLUSTERING APPROACH FOR BLACK-BOX PENETRATION TESTING

by
ÖZGÜN ÖZERK

Submitted to the Graduate School of Engineering and Natural Sciences
in partial fulfilment of
the requirements for the degree of Master of Science

Sabancı University
Dec 2021

WINPEN: A FAST CLUSTERING APPROACH FOR BLACK-BOX PENETRATION TESTING

Approved by:

Date of Approval: Dec 16, 2021

ABSTRACT

WINPEN: A FAST CLUSTERING APPROACH FOR BLACK-BOX PENETRATION TESTING

ÖZGÜN ÖZERK

CYBER SECURITY M.Sc. THESIS, DEC 2021

Thesis Supervisor: Asst. Prof. Dr. Erdinç Öztürk

Keywords: pentesting, clustering, unsupervised, black-box, machine-learning, cyber-security

In black-box penetration testing, a payload is a piece of code that potentially enables unauthorized access to a computer system through an exploit. Grouping payloads based on the behavior they trigger in the target application is a labor-intensive process, where each payload and the corresponding behavior of the application to that payload should be analyzed and interpreted by humans. To assist human evaluation, we propose a new algorithm **WinPen**, which classifies the payloads based on the behavior they are triggering in the system. Each payload is represented as the length of the response strings generated after a payload is submitted in the system. WinPen performs mean-based comparisons for each point in the dataset with respect to the point's previous neighbors. We show on several datasets that WinPen performs with an average 99.85% accuracy score across several datasets. WinPen runs in $O(n \log n + n))$ and the time complexity is reduced to $O(n)$ for already sorted inputs. **WinPen** is programming-language and source-code independent, and can be utilized in Cyber Security applications, faster than the other clustering algorithms (e.g., up to 46× faster than **kmeans1d**), without the need for tedious hyper-parameter tuning procedures.

ÖZET

WINPEN: KARA-KUTU SIZMA TESTLERİ İÇİN HIZLI BIR KÜMELEME ÇÖZÜMÜ

ÖZGÜN ÖZERK

SİBER GÜVENLİK YÜKSEK LİSANS TEZİ, ARALIK 2021

Tez Danışmanı: Dr. Öğr. Üyesi Erdinç Öztürk

Anahtar Kelimeler: sızma testi, kümeleme, denetimsiz, kara-kutu, makine-öğrenmesi

Kara-kutu sızma testlerinde *yük*, bir bilgisayar sistemine, sistemde varolan bir açıktan faydalanarak potansiyel bir yetkisiz erişim sağlayan kod parçasıdır. Yükleri, hedef aplikasyonda tetikledikleri davranışlara göre guruplamak yoğun iş gücü gerektiren bir süreçtir, çünkü her bir yükün ve bu yüklerin hedef aplikasyondaki ilgili davranışlarının bir insan tarafından analiz edilmesi ve anlamlandırılması gerekmektedir. Bu tezde, insan değerlendirmesine katkı sağlayabilmek için **WinPen** olarak isimlendirilen yeni bir algoritma sunulmaktadır. Mevzu bahis algoritma, yükleri, hedef sistemde neden oldukları davranışlara göre kümelendirmektedir. Bunu yapmak için, her bir yük, hedef sistemden gelen cevabın karakter dizisi uzunluğu olarak nitelendirilir. WinPen, veri kümesindeki her bir eleman için, ilgili elemanın önceki komşularını kıstas alarak, ortalama bazlı karşılaştırmalar yapar. Bu tezde, farklı veri kümeleri için WinPen'in ortalama doğruluk puanı %99.85 olarak hesaplanmıştır. WinPen $O(n \log n + n))$ zaman kompleksliğinde çalışmakta, hatta sıralanmış girdiler için bu kompleksliklik $O(n)$'e düşmektedir. WinPen, programlama dillerinden ve kaynak-kodundan bağımsız olup, diğer kümeleme algoritmalarından 46 kata kadar daha hızlıdır. Üstelik zahmetli hiper-parametre ayarlama sürecine gerek yoktur. Bu özellikleri, WinPen'i Siber Güvenlik uygulamaları için ideal bir aday yapmaktadır.

ACKNOWLEDGEMENTS

Thanks to my fellow friend Buse Nur Karatepe, who is also pursuing her Master's Degree in Sabanci University at the time of this thesis, for her significant contributions to this work.Namely, helping me benchmarking Win-Pen by finding related works, creating confusion matrices, showing me how to use Tableau and various tools, double-checking everything, and her great emotional support.

Dedication page

My sincere thanks to Kazuki Takahashi for keeping me sane, letting me preserve my inner child, and enhancing my creativity with his majestic designs.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

CMS Hacker101 - CMS ..... xi, 21, 24, 27, 31

H2 HackThisSite - Realistic - 2nd Challenge ..... x, xi, 20, 21, 22, 23, 24, 26

H8 HackThisSite - Realistic - 8th Challenge ..... xi, 21, 24, 26, 30

N14 Natas - 14th Challenge ..... xi, 21, 24, 25

N15 Natas - 15th Challenge ..... x, xi, 21, 23, 24, 25

PHT Hacker101 - Photo ..... xi, 21, 24, 27

1. INTRODUCTION

With web applications becoming more widespread and prominent in our lives, day by day, the security testing of software plays a pivotal role in engineering secure applications (Stats, 2021). Most of the vulnerabilities are caused by misconfigured interactions between the application and the user. Improper or inadequate sanitization of user input may lead to various injection vulnerabilities (i.e. SQL injection, code injection), some of which can lead to severe problems. For example, a *SQL (Structured Query Language) injection* vulnerability may allow a malicious user to delete all tables, or retrieve all the sensitive data and/or credentials of registered users (Rua Mohamed Thiyab, 2017). Penetration testing aims at discovering and identifying such security vulnerabilities in case there are any in the system under test (Paráda, 2018).

To discover any potential vulnerabilities in the application, the software is penetrated with potential attack vectors. There are several approaches adopted in penetration testing such as white-box, grey-box, and black-box testing. In the white-box testing, the tester is assumed to have complete knowledge of the system under the test (i.e. information technology infrastructure, source code of the software, technologies used in the system, etc.) (Dahl & Wolthusen, 2006; Liu, Shi, Cai & Li, 2012).

Conversely, the black-box testing, which is among the most popular approach for penetration testing (Nidhra, 2012), assumes that the tester has no prior information about the system and no privilege is defined for the tester account in the system. Thus, a black-box penetration tester tries to discover the vulnerabilities in the system from outside such as a typical user/client, or an outsider thereof (Bau, Bursztein, Gupta & Mitchell, 2010; Seng, Ithnin & Shaid, 2018). As a concrete example, one can consider a website with a login page; and in a typical black-box penetrating testing scenario, the tester would inject various payloads to the login form to see if there are any vulnerabilities present in the webserver. This black-box penetration testing process can be divided into 3 sub-processes:

1.1 Finding the payloads / deciding on which payloads to use / generating payloads

1.2 Injecting the payloads to the system

1.3 Analysis of the results / identifying which payloads resulted in a successful injection

World-Wide-Web reserves numerous payloads, both for specific and general use-cases. Additionally, there are many payloads available offline in popular penetration testing tools for the convenience of penetration testers (i.e. *Metasploit*(Anonymous, 2021)).

Injecting every payload manually to the system under the test becomes tedious and repetitive work, since the list of attack payloads is usually too large for manual probing. Fortunately, there are tools available for automating this probing process (second step in the above list), which enables generating a request for each payload, and storing the corresponding response to each request. One such example can be *BurpSuite* (Mainka, Mladenov, Guenther & Schwenk, 2015). Burp Suite allows the tester to import/create his own payload list, or use a predefined one. After the payloads are determined, Burpsuite injects all the payloads in the given list to the selected input field of the website (i.e. *username, status, password*) by crafting a specific request for each payload. Lastly, it displays the content and the length (character count) of the responses' text as a list. Similar scripts/programs can be written easily to mimic the same behavior: sending a request for each payload, then retrieving the respective response length/content.

Currently, only the second step (crafting requests for the list of payloads, and injecting the system with these payloads) is automated while the third step (analysis of the results) is performed manually by human operators. To detect if there is any vulnerability present in the system, understand which payloads lead to different behaviors, and which behaviors are malicious/abnormal, the stored responses to payloads need a detailed analysis. This labor-intensive third step is still conducted mostly by humans. Thus, it would minimize the human effort and its associated costs to automate this analysis step.

This work aspires to offer a novel approach for further automating web penetration testing. To this end, we present a new method, *WinPen*, for grouping different behaviors based on the lengths of the responses to the payloads by the system. The data is represented as 1-dimensional data of the payloads. WinPen proceeds in windows of 1-dimension. To assess WinPen's accuracy in detecting different types of attacks, and its speed, we perform experiments on several datasets. WinPen achieves 99.1% accuracy on one dataset, while achieving 100% for the other five

datasets, outperforming k-means1d in both speed and accuracy. WinPen has only one hyper-parameter and our results indicate that in a very large regime of values, this hyper-parameter does not need to be tuned. The default hyper-parameter can be used in all data sets, and the user can still obtain optimal results. In other words, from a high-level abstraction, the end-user can treat WinPen as a clustering algorithm for 1-dimensional datasets, without a hyper-parameter.

2. BACKGROUND

A very basic of a SQL injection/payload example:

Say, the functionality of the web application under test, is to retrieve data from a table, with a search query. More specifically, let us imagine, when a user provides her password, the web application reveals her e-mail address and her username stored in the database, to the user.

Our web application will also need a back-end, that enables querying this database represented as Table 2.1, and fetching information from it. This back-end source code could be as simple as given in the Listing 2.1.

```
# Define POST variables
userInput =
request.POST['password']

# vulnerable SQL query
sql = "SELECT EMAIL AND USERNAME
FROM TABLE WHERE PASSWORD = '"
+ userInput + "'"

# Execute the SQL statement
database.execute(sql)
```

Listing 2.1 back-end source-code

The expected behavior here is, when the user enters her password, this query will run the code for every row in the table, and return `TRUE` for rows, whose column is the same as that password. And ultimately, will fetch all the rows, whose result is

Table 2.1 Database of the web application

Username	**Email**	**Password**
Sal	sal_vulcano@gmail.com	i_h4t3_c4ts
Brian	brian_quinn@hotmail.com	sp1ders_are_sc4ry
Joe	joe_frog_belly@gmail.com	i_fe4r_my_w!fe
Murr	murr_the_ferret@yahoo.com	1t's_t00_high

`TRUE` for this query. These fetched rows will be sent to the front-end, and displayed to the user.

```
SELECT EMAIL AND USERNAME
FROM TABLE WHERE PASSWORD =
'i_h4t3_c4ts'
```

Listing 2.2 Expected Query

For example: if the user is Sal Vulcano, and wants to see his `Username` and `Email` records, he can query so with his password, and the constructed query would be Listing 2.2. This query will run for every row, and will return `TRUE` only for Sal's row, fetching the results **Sal** and **sal_vulcano@gmail.com** from the database.

However, our web application's back-end is vulnerable to injection. Injecting the payload in Listing 2.3, would grant access to all the rows in the table.

```
abc' or '1' = '1
```

Listing 2.3 The Payload

The payload in this scenario, is the `userInput` (the only manipulable field in the query). When the payload in the Listing 2.3 is injected to the back-end, the final query would look like this:

```
SELECT EMAIL AND USERNAME
FROM TABLE WHERE PASSWORD =
'abc' or '1' = '1'
```

Listing 2.4 Injected Query

What's happening behind the scenes is, although `PASSWORD = 'abc'` will be `FALSE` for each row, `'1' = '1'` will be `TRUE` for every single row, independent of the **Password** field's content. Fundamentally, exploiting the query logic, this query will fetch the **Name** and **Email** fields for every row in the database (Table 2.1 for our case).

Gaining unauthorized access to the names and e-mail addresses of every user in the system is a critical vulnerability. One can defend the last standing brick of this demolished fortress by arguing, the attacker still does not have access to the passwords of other users, and it is not a *critical* vulnerability in that sense.

It is in fact possible to discover other users' passwords one by one, via iterating over our payload and improving it. Appendix A. elaborates on this, and includes the code piece that will retrieve all the user's passwords in the database.

2.1 Prior Work

Majority of the work in automating the penetration testing, focuses on automating the *testing* part rather than the *evaluation* (Appelt, Nguyen, Briand & Alshahwan, 2014; Deptula, 2013; Haubris & Pauli, 2013; Ke, Yang & Ahn, 2009; Kieyzun, Guo, Jayaraman & Ernst, 2009; Kwon, Lee, Lee, Kim, Kim, Nam & Park, 2005; Rak, Salzillo & Romeo, 2020).

For automating the evaluation step, Saleh et al. (Saleh, Rozali, Buja, Jalil, Ali & Rahman, 2015), proposed using Boyer-Moore string matching algorithm for detecting vulnerabilities. In a nutshell, Boyer-Moore string matching algorithm compares the pattern with target text, by inspecting their characters from right to left using 2 heuristics (bad-character shift and the good-suffix shift)(Boyer & Moore, 1977). However, we are questioning this approach, because of the following: it is not practical nor realistic to generate all the necessary keywords even for SQL-based attacks only. The reason for that is, there are no strict rules or standards for developing WEB applications, thus every developer/framework is free to craft their own behaviors/responses for each scenario. To give a more concrete example, a web application might prefer to display a "404 not found" error for every exception (even for SQL errors) that emerged in their application logic, as a cover. This approach is used to not reveal information about the error to the end-user, whilst storing the original detailed error description in the server logs. And some websites may prefer to create their own custom messages to obfuscate the errors.

Aliero, Ghani, Qureshi & Rohani (2020) addressed the evaluation step of the black-box penetration testing for SQL injections. Their work focuses on minimizing the incidence of false-positive and false-negative results of the evaluation phase, by utilizing an object-oriented approach specifically designed for SQL injections, which does not employ any machine-learning technique. Their proposed solution is static and specific to SQL injections, whereas our focus is to provide a general-purpose clustering algorithm for all types of attacks. To test their score, they have developed 3 different web applications, but have not tested their method on already existent real-world web applications. Their method SQLIVS reached a f score of 0.67.

Lastly, Ceccato et al., proposed a new security oracle, called *SOFIA*, made for SQL-injection vulnerabilities(Ceccato, Nguyen, Appelt & Briand, 2016). Security oracle is responsible for assessing whether test executions expose any vulnerabilities. SOFIA was able to detect all SQLi vulnerabilities with inputs generated by three attack

generation tools, and achieved a recall rate of 100%, with a false positive rate of 0.6%. SOFIA uses k-medoids algorithm, and the tuning of hyper-parameter k is crucial, as stated in their work. However, SOFIA's method is relatively complicated compared to other works (including ours). SOFIA's workflow consists of 2 phases, Training and Testing, and includes five steps: Parsing, Pruning, Computing Distance, Clustering, and Classification. They mention in their paper, the first 3 steps are shared by both phases, whereas *Clustering* step is being an exclusive part of the *training* phase, and *Classification* is a unique step to the *testing* phase. Sofia's process starts with a set of SQL statements gathered from safe executions of the system under test. Then, these statements are parsed and represented as parse trees, corresponding to the objects to be classified. Their oracle must use a legitimate statement for this purpose. In other words, the SQL statements need to be selected and verified as legitimate prior to feeding them to the SOFIA. After pruning the parse trees, the clustering step groups the similar SQL statements with respect to the distance between them. This concludes the training phase. The first three steps are also present in the testing phase. After these 3 steps, the classification step of the new SQL statements is being performed by assessing their distance to the center of the clusters (determined in the training phase). If a new statement is close enough to the center of the clusters, it is labeled as benign, otherwise as a potential attack vector.

The most significant difference that separates WinPen from the prior work mentioned here is, Winpen **clusters** the payloads based on their difference, without labeling them as **malicious** or **benign**. WinPen's sole purpose is to aid the penetration tester via grouping the payloads regarding their behaviors, hence reducing the workload of the penetration tester significantly. For example, assume that the penetration tester has 300 payloads to inspect, and these payloads will result in 3 different main behaviors in the system under test. Without WinPen, the penetration tester would have to analyze each one of the payloads one by one. Instead of analyzing each payload, or trying to group payloads, the penetration tester can use WinPen, and then analyze only 3 payloads (one from each group) to determine if the system has a vulnerability.

Because of this property of WinPen (not labeling the payloads as **benign** or **malicious**), there is no need for a *training phase*, unlike traditional machine-learning algorithms.

Web technologies (especially frameworks) change and evolve at a fast pace. What is considered not critical or malicious today, can be an attack vector tomorrow. And also, depending on the application, what can be a security flaw, may not be for

another. An example of that would be: an open-source company's application revealing some static source code when injected with a certain payload. The company may think this is behavior is for sure unexpected, yet harmless, since the source code is already available as this application is an open-source one. However, for another application that is not open-sourced, accessing the source code as an outsider is probably a quite impactful attack scenario. Hence, leaving the decision to the penetration tester on whether the behavior is benign or not, is a solid strategy when it comes to diverse and evolving web technologies.

This emphasizes another advantage of WinPen: letting the penetration tester decide whether the result is benign or malicious. Although this may sound like a disadvantage, it is quite the contrary. The definitions of benign and malicious do vary across different applications, and evolve constantly as explained above.

For example, to keep up with a penetration tester, SOFIA's database (SQL statements gathered from safe executions) would be in need of constant maintenance and updates throughout the time, and these changes should be tailored specifically to the application's purpose. On top of that, this cumbersome process needs to be done for every type of vulnerability (not just for SQLi), whereas WinPen would just work out of the box.

To the best of our knowledge, this work is the first one, that approaches interpreting the results of black-box penetration testing as a clustering problem (for all types of cyber attacks, not only SQL injections). Since WinPen is not deciding whether the payload is malicious or not, but only clustering the payloads into groups, it is more meaningful to compare WinPen with a clustering algorithm, that is suited for working on 1-dimensional data, that does not have a sequential relationship (like time-series), since all payloads are independent of each other.

`kmeans1d` is a Python library with the implementation of k-Means clustering on 1D data, based on the algorithm proposed by Xiaolin's work (Wu, 1991), as presented in section 2.2 of Grønlund's work (Grønlund, Larsen, Mathiasen & Nielsen, 2017). Finding optimal k-Means clustering is an NP-hard problem; however, there are polynomial-time algorithms for clustering 1-dimensional arrays. In the paper, Grønlund et al., propose dynamic programming and generalize it to data structures that can find the globally optimal k-Means clustering for any k. Their improvements make it possible to reduce the space to $O(n)$, and provide $O(nlogn + kn)$ running time ($O(kn)$ if the input is sorted).

Our clustering algorithm for 1-dimensional data, WinPen outperformed `kmeans1d` in both speed and accuracy. Also, all the other clustering algorithms that are currently

available, require hyper-parameter tuning specific to the data. The tuning process of hyper-parameter can be costly in terms of time, and can be detrimental in Cyber security applications, since the number and the type of payloads and tests to be done can vary greatly depending on the application to be tested.

3. PROBLEM DESCRIPTION

At the moment, no human intervention is needed for creating the requests, retrieving the responses, and calculating the lengths of the latter. The interpretation of the results, however, does necessitate human intervention.

Payload #	Response Length
1	120
2	112
3	180
4	187
5	150
6	30
7	31
8	30
9	31
10	120
11	120
12	179
13	120
14	122
15	122
16	125
17	118
...	...
300	125

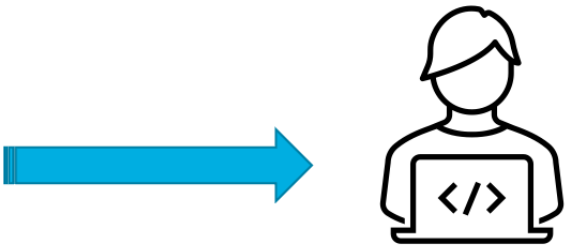

Figure 3.1 manual testing approach

For every payload (as represented in Fig. 3.1), a human decides if there is a vulnerability. Considering there are numerous payloads present on the world wide web, deciding upon which payloads create malicious behavior in the system can be time-consuming and therefore, costly. The advantage of this manual testing is, application specific edge cases are covered, since a human interpreted the results.

The prior works mentioned in this are mainly trying to label the payloads as benign or malicious. The disadvantages of this approach are the following:

- A specific tool needs to be designed for each attack type (SQL, XSS, IDOR, etc.).
- All these tools need to be updated (or re-trained) as the world wide web evolves.

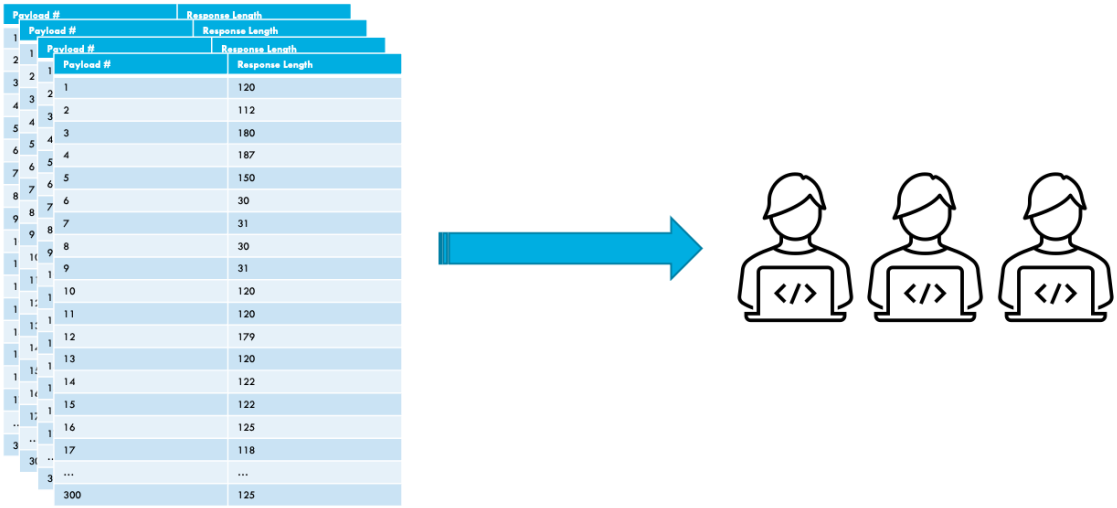

Figure 3.2 labeling the payloads as malicious/benign approach

- Application specific edge cases may not be covered. A malicious behaviour for an application may be benign for another application.

Fig. 3.2 tries to illustrate how much effort will be required for this approach.

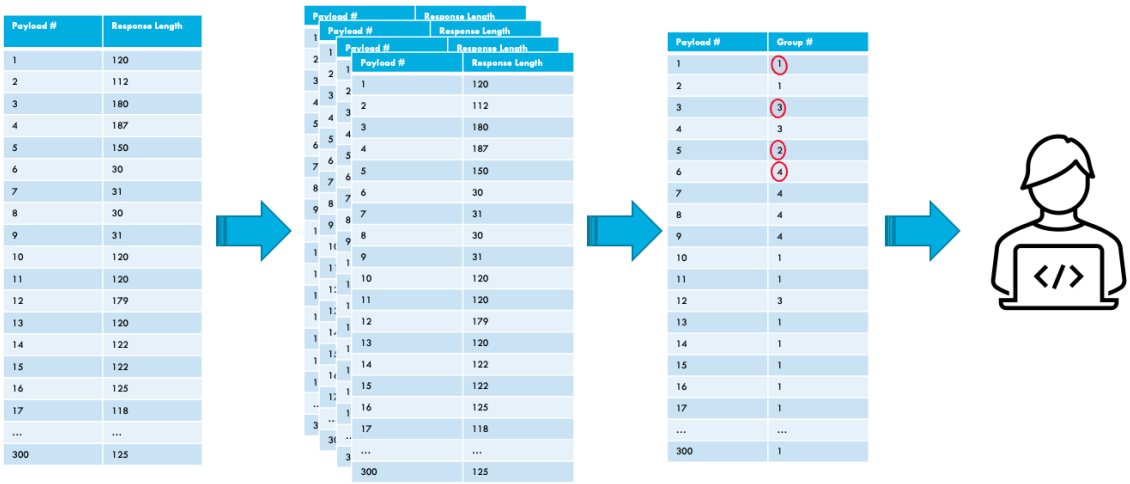

Figure 3.3 using a clustering algorithm with hyper-parameter tuning

An alternative approach would be to use a regular clustering algorithm to diminish the amount of payloads that the penetration tester needs to inspect.

The outputs (lengths of the responses) can be fed to an already existing clustering algorithm to automate the clustering process (Grønlund et al., 2017; Wang & Song, 2011). This approach, however, creates another problem to solve for black-box penetration testing: hyper-parameter tuning. To eliminate the need for human engagement, we again needed human participation as an intermediate step that aims to tune the hyper-parameters for the model. Also, the suggestion of feeding the results to a clustering algorithm, assumes that the white-hat hackers have a

background in Machine Learning, which may not always prove to be true for all.

Fig. 3.3 tries to illustrate this workflow. In this approach, the penetration tester applies the clustering algorithm to the dataset, with each possible hyper-parameter. Thus, generating multiple results. After that, he needs to manually inspect the original dataset and create a basis for comparing the various results he got from different hyper-parameters, so select the best one. Now that, this ultimate result has grouped the payloads accordingly, the penetration tester can inspect only 1 sample from each group, and this should suffice.

However, in this approach, the penetration tester has already spent more effort than manual testing. Hence, this approach does not help. On the contrary, it is making the whole process more difficult, the exact opposite of what we aim.

Although our proposed algorithm needs a hyper-parameter, using the default value for hyper-parameter turns out to generate the nearly same accuracy scores, with a negligible difference in the performance. Thus the hyper-parameter tuning is not necessary in most cases. This allows clustering to be done without any human intervention, and also with better accuracy compared to other clustering algorithms.

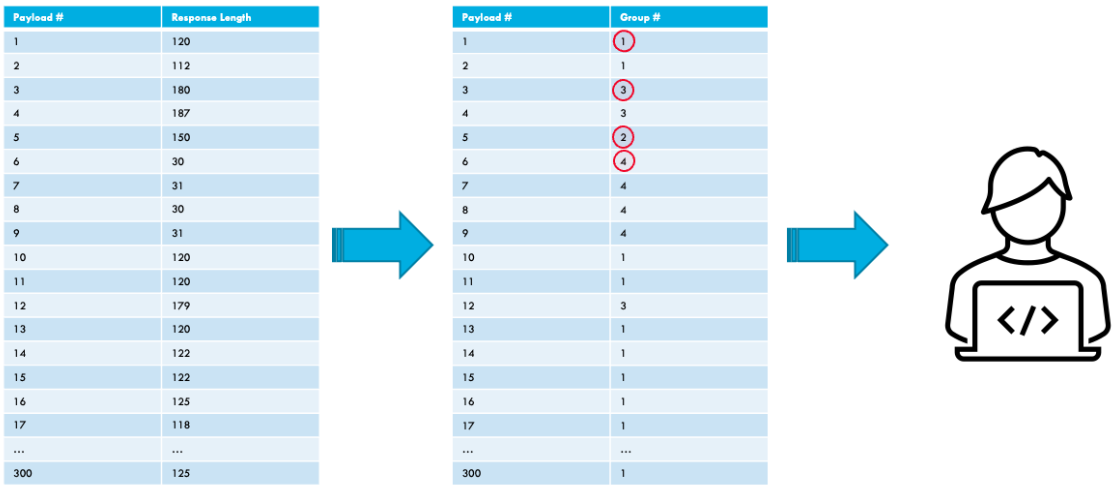

Figure 3.4 clustering algorithm without hyper-parameter tuning

In Fig. 3.4 the workflow of the proposed algorithm from the point of view of the penetration tester is illustrated. The penetration tester feeds the inputs to **WinPen** without tuning any hyper-parameter. **WinPen** groups the payloads with 99.85% accuracy. Allowing the penetration tester to inspect only 1 sample from each group, and finalize the evaluation phase, instead of inspecting nearly every payload.

In short, here are the advantages of **WinPen**'s approach:

- The penetration tester dramatically less time/effort on the evaluation step (it

is almost automated)

- Since **WinPen** groups the payloads based on the behaviour (it is not using some keywords, or does not need training phase), there is no need to tweak/update/train **WinPen** as world wide web evolves. It is expected for **WinPen** to work out of the box, always.
- Ultimately, a human is interpreting the results. So, application specific edge cases are covered.

4. METHODS

WinPen consists of two phases: *sorting* and *clustering*. Sorting helps to increase our results and enables the algorithm to be simpler. Clustering is the labeling phase, where the magic of WinPen actually happens. In this section, the details of these steps will be explained, along with the dataset generation, input format, and the time/space complexities of WinPen.

4.1 Data Set

There was no data set available for the experiments. So, the first task was generating the data. For the selection of payloads, we chose SQL vulnerabilities. SQL vulnerabilities are one of the most popular vulnerabilities, increasing at a rate of 250% a year (Moyle, 2007). One of the reasons for this popularity is, SQL injections are granting unauthorized access to databases, enabling the malicious user to query them, modify and delete their contents. Thus, SQL injection is usually becoming one of the ultimate goals in web penetration testing (other types of vulnerabilities are becoming the intermediate steps to exploit the SQL vulnerabilities). We obtained the necessary payloads for SQL injections from this GitHub repository: *https://github.com/sh377c0d3/Payloads*.

For request forgery, each payload is embedded into a different request, that is specifically crafted for the target URL. As the next step, these crafted requests are sent to the server, which generates a response for each request. Each response has a behavior, which can manifest itself in different ways (e.g., displaying an error message on syntax rules, returning an empty page, returning 403 unauthorized error, etc.). The returned response length is stored in a list (response is a text in *HTTP protocol*, length of the response means, character count in the response text). This length information is used to represent the payload in WinPen.

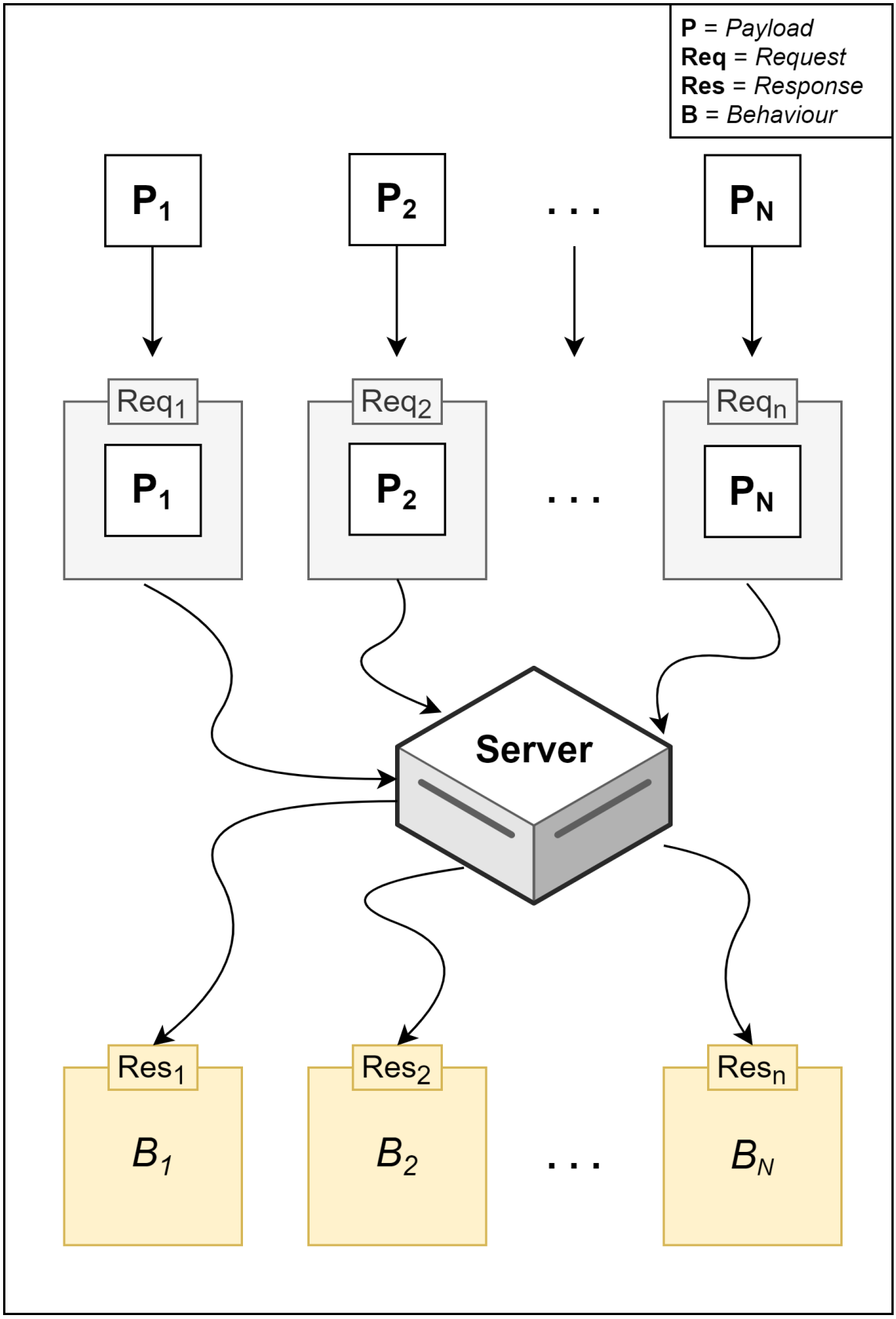

Figure 4.1 Dataset Generation Scheme: unique payloads are embedded in the crafted requests, sent to the server, resulting in various behaviors in the responses created by the server

Data sets were generated from various CTF (Capture the Flag) websites (overthewire's Natas Challenges, hacker101's challenges, and hackthissite's realistic missions) using the methodology explained in Fig. 4.1. For automated request generation and response retrieval, we use Python 3 *Requests* library to interact with web servers via HTTP protocol.

4.2 Proposed Algorithm

4.2.1 Inputs

Given a series of response lengths, $x_1, x_2, ..., x_N$ corresponding to the requests created by payloads $\rho_1, \rho_2, ..., \rho_N$, define a neighborhood of points $\eta_t^{(\kappa)} = \{x_{t-(\kappa-1)}, x_{t-(\kappa-2)}, ..., x_{t-1}, x_t\}$ where κ is the size of the neighborhood window, starting at $t-(\kappa-1)$, and ending at t.

For example, ρ_i might be consisting of `<' or 1=1 '>`, which is one of the most common SQL injection payloads. This payload tries to close the input string with an apostrophe ('), effectively ending the string, and opening a window to interfere with the logic. Then, manipulates the condition logic, in such a way that it always returns true for every entry in the table, ultimately resulting in retrieval of all the data in the table that the server querying on. ρ_i being one of our payloads, the length of the corresponding response we have received from the server, x_i may be *927*. Although the algorithm is supplied with 2 inputs ($\{\rho_1, \rho_2, ..., \rho_N\}$ and $\{x_1, x_2, ..., x_N\}$), the clustering process uses only $\{x_1, x_2, ..., x_N\}$. After the abnormalities are found and classifications are made, the generated labels $\{y_1, y_2, ..., y_N\}$ for each result are passed on to their respective payloads, in other words, ρ_i is labeled with y_i.

4.2.2 Sorting

Two lists were created for payloads and response lengths. The payloads do not carry any sequential relationship, they are all unique and independent from each other.

In other words, the payloads can be shuffled. Our goal is to find the abnormalities and cluster the resulting behaviors, and WinPen is carrying out these computations for response lengths. By sorting the response lengths, we are easing the clustering process and increasing the accuracy. Although, when the list consisting of response lengths $\{x_1, x_2, ..., x_N\}$ is sorted, in order to preserve the index relation between payloads and response lengths, the payload list $\{\rho_1, \rho_2, ..., \rho_N\}$ is also need to be sorted with respect to the response length list.

4.2.3 Clustering

Our algorithm iterates in windows on the input sequence. A window consist of w items (an item is an integer: response length of a payload), from the beginning of the list till the end. To find y_i, in each iteration, mean of the neighborhood of points η_t^w is computed, represented by m_t^w. If $(x_{t-1} - m_t^w) < (x_{t-1} - x_t)$, then x_t is perceived as a spike by the algorithm.

The reason for not using a simple equality check instead of *mean* calculation is: an equality check amongst x_{t-1} and x_t could not suffice for spike detection when the application to be tested is a search form (displaying the results of relevant elements from the database). In such cases, every payload might be resulting in different response lengths, but show the same behavior (encountered in our datasets). Also considering that, world wide web is a vast place to make generalized assumptions, the *mean* approach provides more robustness against possible different scenarios. Not belonging to the window can mean two different things: being an abnormality, or the beginning of a new class. The importance of the hyper-parameter w here is, it determines how many items should play a role in a window for getting a *mean*. We have tried different values for the parameter w, which indicates how many elements are there in the window. The results for different w values will be given and further discussed in the *Results* section.

Algorithm 1 WinPen

Input: $list$

Input: $w = 12$

Output: $labels$

```
1: length = len(list)
2: labels = []
3: newLabel = 0
4: wind = list[0 : w]
5: mean = mean(wind)
6: for i = 1; i < w; i++ do
7:     diff = abs(wind[i − 1] − mean)
8:     spike = wind[i] − wind[i − 1]
9:     if diff < spike then
10:        newLabel++
11:    end if
12:    labels[i :] = newLabel
13: end for
14: oldVal = list[0]
15: for (i = 1; i < length − (w − 1); i++) do
16:    wind = list[i : i + w]
17:    mean+ = (wind[w − 1] − oldVal)/w
18:    diff = wind[w − 2] − mean
19:    spike = wind[w − 1] − wind[w − 2]
20:    if diff < spike then
21:        newLabel++
22:    end if
23:    labels[i + w − 1 :] = newLabel
24: end for
25: return labels
```

4.2.4 Time Complexity

The algorithm compares the last element of the windows to the second last element. This strategy is chosen in order to evade the redundant comparisons that emerged from the intersection of the windows. A handicap of this strategy is, the first *w-1* elements are not inspected. Another *for loop* has been constructed (the first for

loop) solely for the purpose of including the first *w-1* elements in our classifications.

First *for loop* consists of $w-2$ comparisons, which does not depend on n (assume the size of the inputs is denoted by n). The second *for loop* performs $n-(w-1)$ comparisons. When these two *for loops* complexities merged together, we get $w-2+n-(w-1)=n-1$. So our algorithm's complexity is $O(n-1)$, which is equivalent to $O(n)$, allowing us to achieve linear complexity.

Taking the mean of the window in a trivial way is $O(w)$, which would make the total complexity $O(nw)$. If we keep in mind that the window is iterating through the list, the iterations of the window would include w-1 common elements in between. By subtracting the first element of the previous window from the last element of the current window, dividing this result to w, and updating the *mean* via adding this value, we can reduce the calculation of the *mean* to $O(1)$.

4.2.5 Space Complexity

We need an extra list to store the assigned labels, which should have the same size as the inputs. Along with that, we also need to store another list consisting of w elements. Our total space requirement is $O(2n+w)$.

5. RESULTS

In this section, WinPen's accuracy results and speed performance will be explained in detail and compared to kmeans1d.

5.1 Accuracy

5.1.1 Effect of Hyper-Parameter `w` on Accuracy

The driving motivation for the creation of this algorithm, was to replicate, or even better the humans' decision-making process for 1-dimensional lists. When an average human tries to classify elements in a list, he/she can keep 5 to 9 items in his memory. This is the short-term memory limit for the average human being (Miller, 1956). Replicating a human, thus using the values 5, 7, and 9 for `w` was a good starting point. Then we have incremented the value of `w` up to 300 (since we have 345 payloads per dataset). A small value for `w` resulting in higher sensitivity for spike detection; a bigger value for `w` meaning higher tolerance for spike detection. As can be seen in the Table 5.1, being too tolerant (for example, `w` = 300) can lead to an increase in erroneous labels, similarly, too much sensitivity (where `w` is small) will also result in many erroneous labels. We have found the sweet spot in between the range 9-20, and chose 12 as our hyper-parameter for generic purposes.

With the chosen hyper-parameter value, we have tested the algorithm with 6 different CTF(Capture The Flag) challenges from 3 different domains *overthewire, hacker101, hackthissite*. In 5 of these test datasets, our algorithm's accuracy score came out as a perfect 1; in only one of them (**H2**) there were 3 errors amongst 345.

Table 5.1 WinPen error amounts for different `w` values on 6 different datasets (out of total 345 payloads)

w	**N14**	**N15**	**H2**	**H8**	**PHT**	**CMS**
5	0	0	22	0	0	1
7	0	0	3	0	0	1
9	0	0	3	0	0	0
12	0	0	3	0	0	0
15	0	0	3	0	0	0
20	0	0	0	0	0	81
25	0	0	0	0	0	81
30	0	0	0	3	0	81
50	0	0	0	4	0	81
100	0	11	0	4	0	81
200	0	11	0	51	0	81
300	0	11	2	51	0	100

In total, our accuracy turned out to be 99.85%. This accuracy score might seem too good to be true. But it should be kept in mind that, the datasets we are dealing with in penetration testing, are not comparable to the usual datasets that are used in the Machine Learning area for general purpose (i.e. iris dataset, real-estate prices dataset). In our dataset, there were clear distinctions between the results of the payloads that are creating different behavior (probable successful injection) and those that are not. Thus, it was easy for our algorithm to distinguish between these different response lengths. Although we do not possess a vast amount of datasets at the moment, the accuracy score of perfect 1 is unlikely to change for most of the datasets, since different behaviors are usually tractable from the response lengths.

5.1.2 Effect of The Dataset on Accuracy

From Table 5.1, it can be seen that Datasets can affect the accuracy. For example, our algorithm made no errors for the dataset **N14**, but made various errors for **H2** dataset. **N14** generated 3 very distinctive responses for the payloads we sent, each of them representing a different behavior. And for all the responses belonging to the same behavior group, their character amount was equal to each other. Thus, our algorithm could easily label the responses. However, for **H2** the server was including some random text (a quote related to Cyber-Security) in the response. On top of that, for some payloads that were creating errors in the database, the error text was including the payload itself in the response as plain text (recall the length of

Table 5.2 WinPen's Confusion Matrix on H2

Predicted Labels	***0***	2	0	0	0	0
	1	0	237	0	0	0
	2	0	0	80	0	0
	3	0	0	0	23	0
	4	0	0	0	3	0
		0	***1***	***2***	***3***	***4***
		True Labels				

payloads vary greatly). Hence, **H2** server complicated the process of determining the label for each response depending on the character amount.

There is no guarantee that servers will handle the requests identically. On the contrary, it is most probable that each server will treat requests dramatically different than the others. To inspect this, we can further examine the 3 faulty labels in **H2**, and dive into its details. The Table 5.2, presents the confusion matrix of WinPen (with w=12) on the dataset **H2**.

Label 0 corresponds to the server not responding to our request. This could be due to a timeout, or a fake error message to mask some different behavior. Label 1 corresponds to an error message about *invalid credentials*. Label 2 corresponds to another error message about *invalid credentials*, but this time, with an additional message indicating an SQL error has occurred. Label 3 corresponds to a successful injection.

Our algorithm decided to label the last three responses with `4`, although they did not show any different behavior than the 23 other responses previous to them. Our algorithm should have labeled these last three responses as `3` instead of `4`. The reason our algorithm made the erroneous decision is simple: on the successful injection, the server also displays a random famous quote related to cyber-security. For example, a quote from one of the responses labeled as `3` was as follows: *"Being able to break security doesn't make you a hacker any more than being able to hotwire cars makes you an automotive engineer." -Eric Raymond*; a quote from a response labeled as `4` was on the other hand: *"I think the very concept of an elite commission deciding for the American people who deserve to be heard is profoundly wrong." –former Congressman Newt Gingrich on the "Commission on Presidential Debates"*. Since the first quote is significantly shorter than the latter one (according to our algorithm), they were labeled differently.

However, when our hyper-parameter `w` is set to 20, our algorithm makes no mistake. Since our primary goal was to be able to skip hyper-parameter tuning, achieving a perfect score for **H2** where `w = 20` did not hold importance. The value 20 for our

Table 5.3 kmeans1d's Confusion Matrix on H2

Predicted Labels	***0***	2	0	0	0	0
	1	0	237	80	0	0
	2	0	0	0	4	0
	3	0	0	0	19	0
	4	0	0	0	3	0
		0	***1***	***2***	***3***	***4***
		True Labels				

hyper-parameter `w` did not provide optimal results in other datasets, and 3 mistakes for the value 12 for hyper-parameter `w` are tolerable for the sake of greater good.

5.1.3 Comparison with Kmeans1d

The predictions of **kmeans1d** with $k = 5$ have been displayed in the Table 5.3. It can be seen that, 237 payloads (true label `1`) and the 80 payloads after them (true label `2`) are grouped into the same cluster by **kmeans1d**, although they should have been put into different ones (since the behavior they are creating is different). In this dataset, the 80 payloads should have been discriminated from the 237 payloads, so, 80 payloads have been labeled erroneously. We should not say 237 of them should be separated from the 80 ones, and label them as wrong. As next, **kmeans1d** placed the 26 payloads into 3 different groups (4 ones into label `2`, 19 ones into `3`, and the last 3 into `4`). Our evaluation criteria are the same: we are not comparing the predicted labels to the correct labels, we are comparing the distinctions to the correct ones. In this specific case, 26 of them should have been put into the same group, but 2 extra labels were created. Because of these very reasons, confusion matrices do not trivially allow us to calculate the accuracy for clustering purposes. Hence, we have selected AMI (Adjusted Mutual Information) as our performance evaluation metric. AMI is a variation of mutual information, and can be used in clustering, fitting our needs (Nguyen, Epps & Bailey, 2009). Other evaluation metrics such as Adjusted Rand Index(ARI) and Normalized Mutual Information (NMI) have also been tried, however, AMI gave us the most reasonable and fair results among all.

Table 5.4, demonstrates the effect of misinterpreting the confusion matrices. If we are to simply check the correct and predicted labels, the accuracy score of **kmeans1d** with $k = 5$ would be 0 on the dataset **N15**. Yet, it's crystal clear that the algorithm managed to differentiate the necessary 3 different behaviors, and relate them to their corresponding payloads with 100% accuracy. AMI, on the other hand,

Table 5.4 kmeans1d's Confusion Matrix on N15

Pred labels					
	0	0	0	0	0
	1	41	0	0	0
	2	0	11	0	0
	3	0	0	293	0
		0	***1***	***2***	***3***
		True Labels			

Table 5.5 AMI Scores of WinPen's and kmeans1d's

Algorithm	***Hyper Parameter***	***AMI Score***					
		N14	*N15*	*H2*	*H8*	*PHT*	*CMS*
WinPen	*w=12*	1.00	1.00	0.98	1.00	1.00	1.00
kmeans1d	*k=2*	0.84	0.91	0.48	0.23	0.40	0.79
	k=3	1.00	1.00	0.53	0.75	1.00	0.87
	k=4	1.00	1.00	0.51	0.85	1.00	0.78
	k=5	1.00	1.00	0.50	0.86	1.00	0.77
	k=6	1.00	1.00	0.96	0.99	1.00	0.74
	k=7	1.00	1.00	0.94	1.00	1.00	0.73

can interpret these results correctly, and find the perfect score 1 for **kmeans1d** on **N15** as it should be. This is the reason, we are using AMI as our performance metric.

In Table 5.5, WinPen's AMI scores for all datasets are presented (using the optimal hyper-parameter $w = 12$). Average AMI score for **WinPen** is 0.99. Similarly, **kmeans1d**'s AMI scores are shown for every hyper-parameter value `k`. Since our datasets included 7 different behaviors at a maximum per dataset, the upper limit of k is set to 7 (this is in favor of kmeans1d). And since it would be the most basic approach to assume there will be 2 different behaviors (i.e., malicious, non-malicious), lower-bound is set to 2. Average AMI score for **kmeans1d** is 0.84, whereas **WinPen**'s score is 0.99.

In the Figures 5.1,5.2,5.3,5.4,5.5,5.6, the accuracy scores of **WinPen** and **kmeans1d** are given for every dataset.

5.2 Runtime Complexity

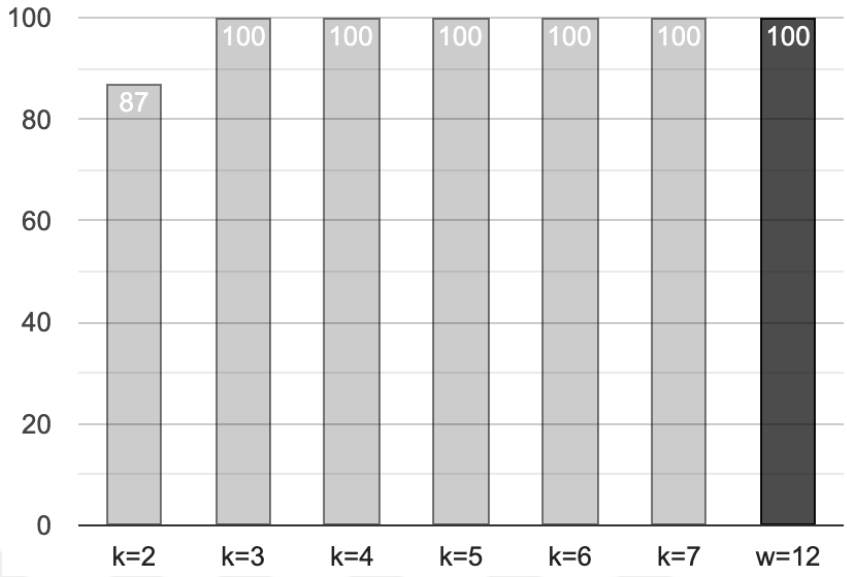

Figure 5.1 Kmeans1d vs WinPen's Accuracy Scores on N14

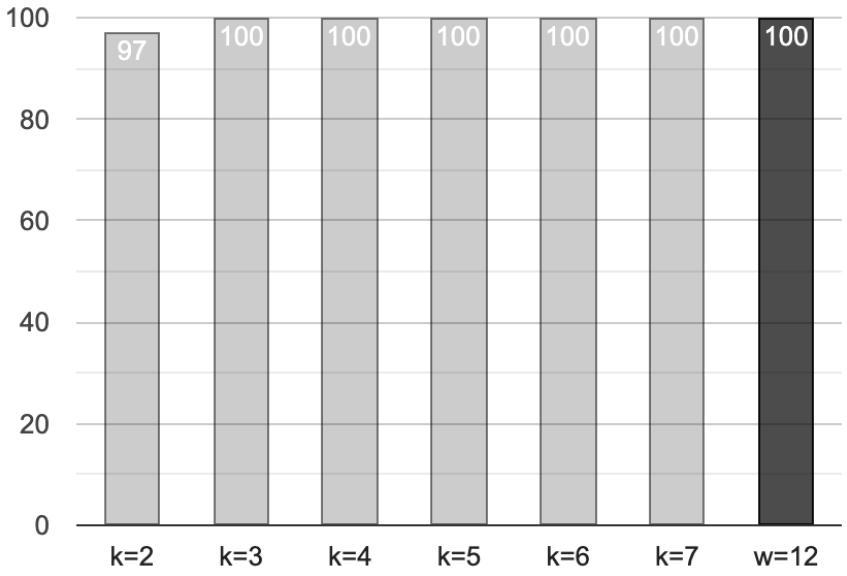

Figure 5.2 Kmeans1d vs WinPen's Accuracy Scores on N15

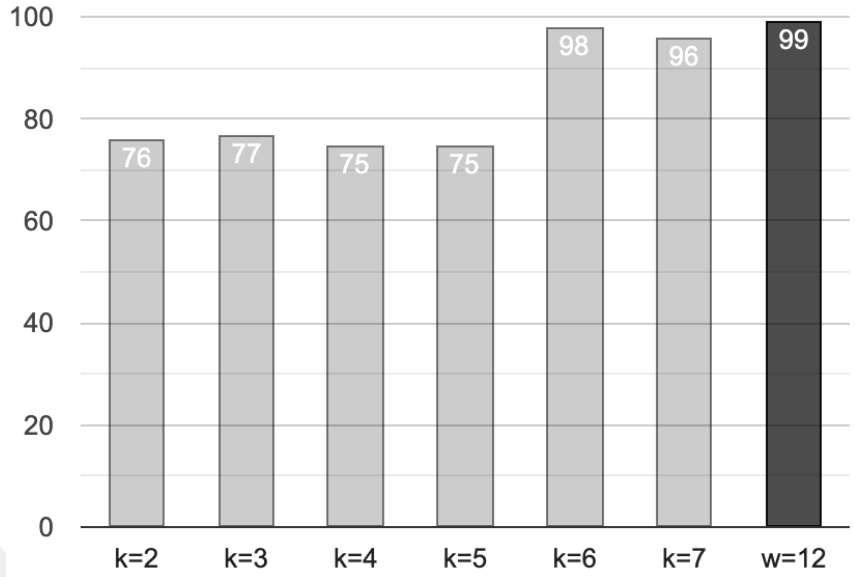

Figure 5.3 Kmeans1d vs WinPen's Accuracy Scores on H2

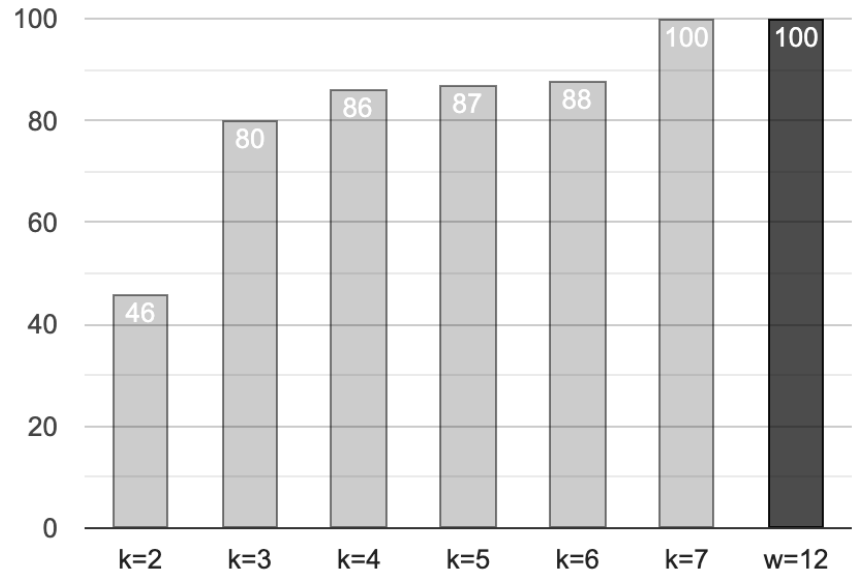

Figure 5.4 Kmeans1d vs WinPen's Accuracy Scores on H8

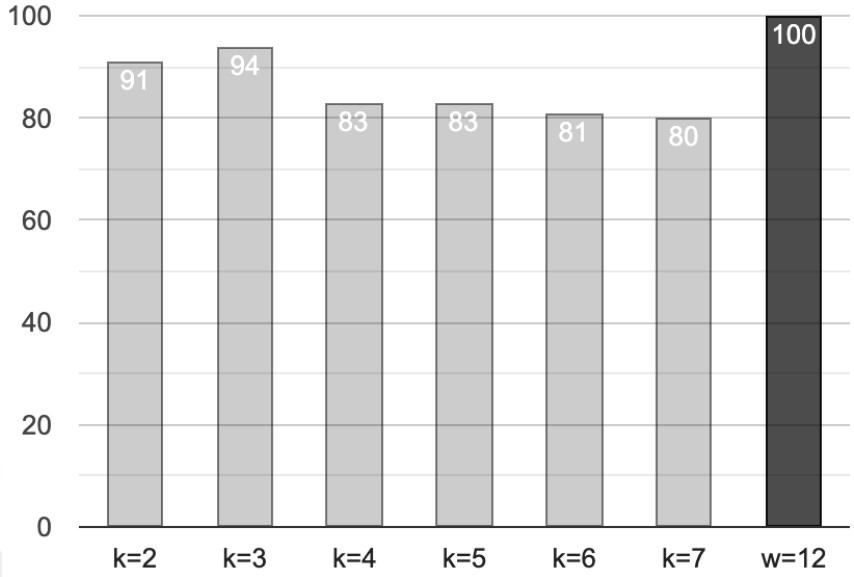

Figure 5.5 Kmeans1d vs WinPen's Accuracy Scores on CMS

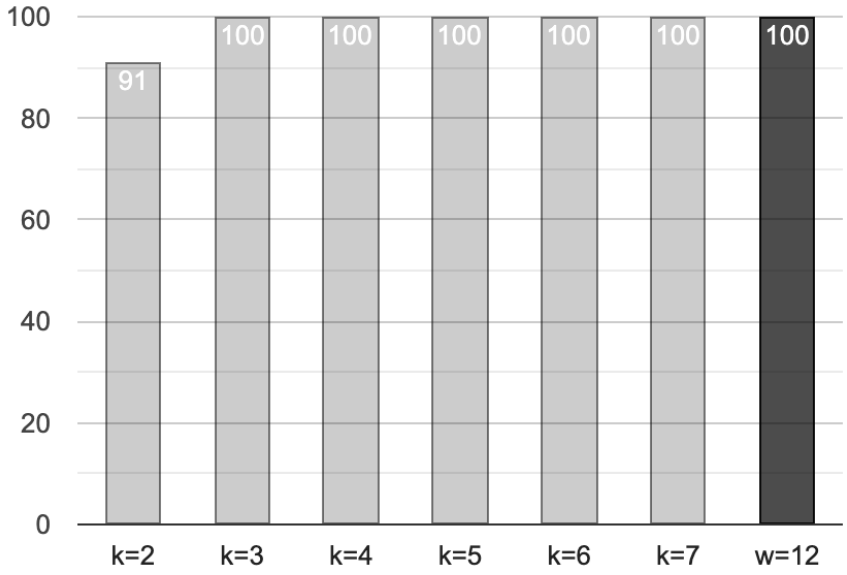

Figure 5.6 Kmeans1d vs WinPen's Accuracy Scores on PHT

The complexity of our algorithm is $O(n+nlog(n))$, whereas the complexity of original k-means algorithm is $O(n^2)$. **kmeans1d**, the most appropriate candidate to compare with, has the time complexity $O(kn+nlog(n))$. Pakhira(Pakhira, 2014) has proposed an alternative k-means version with linear time complexity. Unfortunately, it requires an additional hyper-parameter α to tune other than k, making it unsuitable for automation purposes (requires even more human intervention).

Table 5.6 WinPen's vs. kmeans1d's execution times

Algorithm	**Hyper Parameter**	***Time (ms)***	
		n=345	*n=78*
WinPen	*w=12*	0.023	0.005
kmeans1d	*k=2*	0.249	0.067
	k=3	0.378	0.102
	k=4	0.496	0.132
	k=5	0.604	0.163
	k=6	0.712	0.202
	k=7	0.862	0.230

n denotes the element amount in the dataset
timings are taken using Intel i7 4770k - 16GB Memory(average of 10k runs)

The speed comparison between **kmeans1d** and **WinPen** is given in the Table 5.6.

It must be highlighted that, **WinPen**'s hyper-parameter `w` and **Kmeans1d**'s hyper-parameter `k` are representing different things, and it's not meaningful to relate them (as it would made no sense to relate **KNN**'s `k` with **Kmeans**'s `k`).

6. DISCUSSION

6 different test datasets have been tried. Our algorithm accurately clustered which payloads were responsible for malicious behaviors in all of them. Since the same principles are applicable for other types of injections as well (*i.e. code injection, OS command injection*), the same results (99.85% accuracy) can be expected in every aspect of black-box pentesting, and maybe even in different areas (other than cyber-security) which can utilize 1-dimensional array clustering. As mentioned in the introduction, this algorithm may further automate the black-box penetration testing methodologies.

One weakness of our algorithm is, it relies solely on the response length to distinguish different behaviors. If the different behaviors do not express themselves on the length of the response text, our algorithm will surely not be able to differentiate this behavior. An example can be *blind SQL injections*. In these scenarios, it is not possible to see the output of the payload, but a *sleep* command can be put in the injection, and the load time of the web page can be checked in order to see if *sleep* command has worked or not. This time difference can also be added to the algorithm to inspect. A caveat would be, if the payloads do not cover the injection, our algorithm's result will be nearly useless, since there will be only one behavior, and nothing to cluster. Thus, the importance of the payload list should not be underestimated. These lists can be found online, or can be generated manually for specific purposes.

The biggest advantage compared to the similar clustering algorithms is that **WinPen** provides us an option to skip the process of the hyper-parameter tuning. Hyper-parameter tuning for black-box testing can be more ambiguous than other ML applications, since the tester does not know how many different behaviors will the system provide. For example, if the black-box pen testing is performed on a login form of a web page, the tester might expect two different behaviors only: an error message (indicating a successful injection), and a normal output / no output (indicating an unsuccessful injection). In the examples of **natas14** and **natas15** however, we have encountered three different behaviors: an error message (indicat-

ing a semi-successful injection), no output (indicating an unsuccessful injection), and an output (indicating a fully successful injection). These results also surprised us, since we were expecting only two different behavior. If we had used **kmeans1d** as the first algorithm, we would have tuned *k=2* for two different clusters, which will yield incorrect results, and significantly reduce the accuracy. We also want to emphasize that, *H8* yielded 7 different behaviors, which was not predictable.

On the next page, one can see a detailed comparison between WinPen's and **kmeans1d**'s accuracy scores as it is represented in a graph for 6 different cases for different k values. It can be clearly seen that WinPen (with the parameter w=12) provides the best scores in terms of accuracy.

Aside from payload variety, data generation is also a critical step for this algorithm. If some errors occur during the dataset generation process (i.e. busy traffic, unreliable network connection, server maintenance), these errors will be there to stay. Our algorithm's sole aim is to cluster, not to correct errors. For the best results, the tester should check if the platform to be tested and his own connection to the platform are both reliable during the test.

Having the luxury of not worrying about the hyper-parameter tuning also benefits the accuracy score. Our algorithm distinguishes between different behaviors by detecting the spikes, so it can deduce the number of different behaviors, in other words, how many clusters should be there by itself, without the need for a hyper-parameter. Recall that, WinPen's hyper-parameter has no correspondance with the cluster amount. Other alternatives like **kmeans1d** on the other hand, require the correct hyper-parameter (how many clusters are there) to be able to get close to our accuracy score. This is problematic, since the tester himself does not know in advance, how many different behaviors will the system produce. Leaving the tester with the only option of brute-forcing the possible values for hyper-parameter, and then interpreting all the results by himself to decide on the correct hyper-parameter. This is even more time-consuming than interpreting the results manually without kmeans1d. Because in order to find the best hyper-parameter, one does have to know in advance what are the correct results are (so that results can be compared to each other and the best hyper-parameter can be selected). And for clustering algorithms that require hyper-parameter tuning, selecting the best hyper-parameter for each case corresponds to:

- Identify the correct results (the tester needs to do the clustering process manually)
- Run the algorithm that requires hyper-parameter tuning for each possible

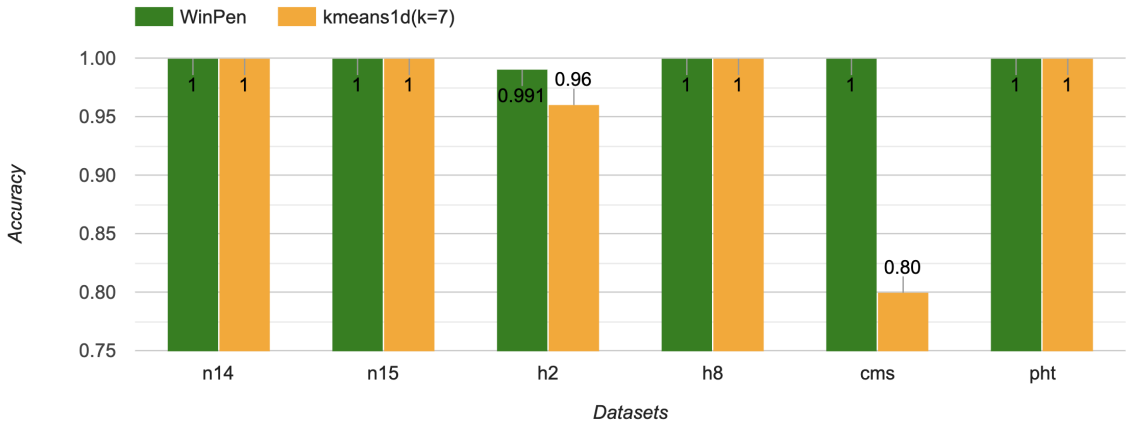

Figure 6.1 WinPen's default vs kmeans1d's best accuracy scores

hyper-parameter

- Compare the results with the manual clustering, and select the best hyper-parameter

This is contradicting to our goal of automation of penetration testing, since it now requires even more work for humans: human interpretation of the results, brute-forcing the values for hyper-parameter, then selecting the correct hyper-parameter. This counter-example shows why clustering algorithms are not heavily used in black-box penetration testing, and signifies why having the option to not tune the hyper-parameter (and still getting the optimal results) can hold crucial importance.

Additionally, our algorithm outperforms its competitors in both speed and accuracy. In the Figure 6.1, the accuracy scores of **WinPen** with the optimal hyper-parameter $w = 12$, and **kmeans1d** with it's best performing hyper-parameter $k = 7$ (w.r.t. accuracy) are given. This comparison is actually in favor of kmeans1d, since every possible value for its hyper-parameter has been tried, and the best-performing one is selected. If new datasets were to behave like **CMS**, **kmeans1d** with $k = 7$ may get worse accuracy scores. In other words, tuning the hyper-parameter $k = 7$ might be specific to these 6 datasets, and might perform poorly for other datasets. As it is for original k-means, the best strategy would be tuning the hyper-parameter for each dataset specifically. It is unfortunately not possible to claim that $k = 7$ is a general-purpose tuning, and would produce nearly optimal results for each dataset.

Contrarily, **WinPen** does not require its hyper-parameter to be tuned. The default value ($w = 12$) would produce nearly optimal results for every dataset.

Thereby, we are actually comparing a general purpose algorithm **WinPen** ($w = 12$) with a specifically tuned **kmeans1d** ($k = 7$). Yet, **kmeans1d** never bests **WinPen** in any dataset.

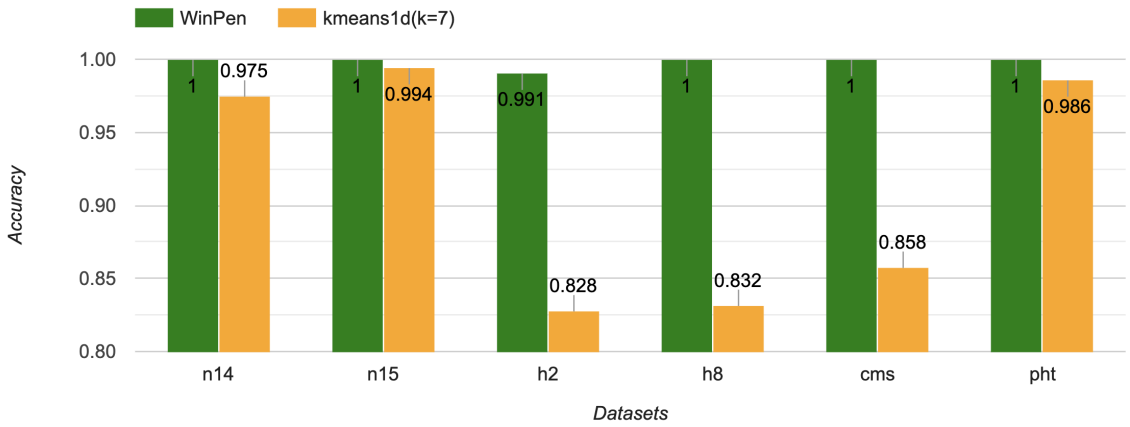

Figure 6.2 WinPen's default vs kmeans1d's average accuracy scores

A more fair comparison between **WinPen** and **kmeans1d** would be taking the mean of all the results **kmeans1d** produced, since we would not know which hyper-parameter would perform the best. It can be seen in Fig. 6.2 that, the gap between **WinPen** and **kmeans1d** is becoming wider when the scenario is realistic.

7. CONCLUSION

For 6 different datasets generated from popular CTF challenges, our proposed algorithm **WinPen** was able to classify 99.85% of the payloads correctly, and identify different behaviors. It's execution time was 10x to 46x faster than the **kmeans1d** (with k=2 and k=7 respectively). Its accuracy was 8.63% better than **kmeans1d**'s mean scores on average, and 1.1% better than **kmeans1d**'s maximum scores in total.

Considering **WinPen** also allows the user to skip the hyper-parameter tuning process, its convenience and speed can be utilized in many different areas (especially Cyber-Security), helping to the transition into automation by eliminating the need for human intervention in the clustering step.

7.1 Future Work

Coverage for other types of behavior (other than response length), for example, time (blind-SQL) might be implemented into the algorithm. This can be done by utilizing an extra array, that stores the response time. The script that generates the inputs needs to be changed accordingly as well in this case. The steps should be:

- craft requests from the payloads (no change needed here)
- send the requests to the system under test (no change needed here)
- start a timer for each request sent to the system under test
- when a response arrives, the script should also store the time it took for the server to reply back
- store the response's string length (no change needed here)

Then, the decision can be done with respect to two criteria instead of one. Or simply, the penetration tester can be prompted on whether s/he wants to categorize the payloads based on time or string length. Or, both of the results can be independently computed, and then can be displayed to the penetration tester. These options are more of a design choice of the developer.

BIBLIOGRAPHY

Aliero, M. S., Ghani, I., Qureshi, K. N., & Rohani, M. F. (2020). An algorithm for detecting SQL injection vulnerability using black-box testing. *J. Ambient Intell. Hum. Comput.*, *11*(1), 249–266.

Anonymous (2021). List of Metasploit Payloads (Detailed Spreadsheet) - InfosecMatter. [Online; accessed 20. Oct. 2021].

Appelt, D., Nguyen, C., Briand, L., & Alshahwan, N. (2014). Automated testing for sql injection vulnerabilities: An input mutation approach. In *2014 International Symposium on Software Testing and Analysis, ISSTA 2014 - Proceedings*.

Bau, J., Bursztein, E., Gupta, D., & Mitchell, J. (2010). State of the art: Automated black-box web application vulnerability testing. In *2010 IEEE Symposium on Security and Privacy*, (pp. 332–345).

Boyer, R. S. & Moore, J. S. (1977). A fast string searching algorithm. *Commun. ACM*, *20*(10), 762–772.

Ceccato, M., Nguyen, C. D., Appelt, D., & Briand, L. C. (2016). Sofia: An automated security oracle for black-box testing of sql-injection vulnerabilities. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, (pp. 167–177).

Dahl, O. M. & Wolthusen, S. D. (2006). Modeling and execution of complex attack scenarios using interval timed colored petri nets. In *Fourth IEEE International Workshop on Information Assurance (IWIA'06)*, (pp. 12 pp.–168).

Deptula, K. (2013). Automation of cyber penetration testing using the detect, identify, predict, react intelligence automation model.

Grønlund, A., Larsen, K. G., Mathiasen, A., & Nielsen, J. S. (2017). Fast exact k-means, k-medians and bregman divergence clustering in 1d. *CoRR*, *abs/1701.07204*.

Haubris, K. P. & Pauli, J. J. (2013). Improving the efficiency and effectiveness of penetration test automation. In *2013 10th International Conference on Information Technology: New Generations*, (pp. 387–391).

Ke, J.-K., Yang, C.-H., & Ahn, T.-N. (2009). Using w3af to achieve automated penetration testing by live dvd/live usb. In *Proceedings of the 2009 International Conference on Hybrid Information Technology*, ICHIT '09, (pp. 460–464)., New York, NY, USA. Association for Computing Machinery.

Kieyzun, A., Guo, P. J., Jayaraman, K., & Ernst, M. D. (2009). Automatic creation of sql injection and cross-site scripting attacks. In *2009 IEEE 31st International Conference on Software Engineering*, (pp. 199–209).

Kwon, O.-H., Lee, S. M., Lee, H., Kim, J., Kim, S. C., Nam, G. W., & Park, J. G. (2005). HackSim: An Automation of Penetration Testing for Remote Buffer Overflow Vulnerabilities. In *Information Networking. Convergence in Broadband and Mobile Networking* (pp. 652–661). Berlin, Germany: Springer.

Liu, B., Shi, L., Cai, Z., & Li, M. (2012). Software vulnerability discovery techniques: A survey. In *2012 Fourth International Conference on Multimedia Information Networking and Security*, (pp. 152–156).

Mainka, C., Mladenov, V., Guenther, T., & Schwenk, J. (2015). Automatic recogni-

tion, processing and attacking of single sign-on protocols with burp suite. In Hühnlein, D., Roßnagel, H., Kuhlisch, R., & Ziesing, J. (Eds.), *Open Identity Summit 2015*, (pp. 117–131)., Bonn. Gesellschaft für Informatik e.V.

Miller, G. A. (1956). The magical number seven, plus or minus two: Some limits on our capacity for processing information. *The Psychological Review*, *63*(2), 81–97.

Moyle, S. (2007). The blackhat's toolbox: Sql injections. *Network Security*, *2007*(11), 12–14.

Nguyen, V., Epps, J., & Bailey, J. (2009). Information theoretic measures for clusterings comparison: is a correction for chance necessary? In Bottou, L. & Littman, M. (Eds.), *Proceedings, Twenty-Sixth International Conference on Machine Learning (ICML 2009)*, (pp. 1073 – 1080)., United States of America. Association for Computing Machinery (ACM). International Conference on Machine Learning 2009, ICML 2009 ; Conference date: 14-06-2009 Through 18-06-2009.

Nidhra, S. (2012). Black box and white box testing techniques - a literature review. *International Journal of Embedded Systems and Applications*, *2*, 29–50.

Pakhira, M. K. (2014). A linear time-complexity k-means algorithm using cluster shifting. In *2014 International Conference on Computational Intelligence and Communication Networks*, (pp. 1047–1051).

Paráda, I. (2018). Basic of cybersecurity penetration test. *Hadmernok*, *13*(3), 435–442.

Rak, M., Salzillo, G., & Romeo, C. (2020). Systematic iot penetration testing: Alexa case study. In *ITASEC*.

Rua Mohamed Thiyab, Musab A. M. Ali, F. B. A. (2017). The impact of sql injection attacks on the security of databases. In *Proceedings of the 6th International Conference of Computing & Informatics*, (pp. 323–331).

Saleh, A. Z. M., Rozali, N. A., Buja, A. G., Jalil, K. A., Ali, F. H. M., & Rahman, T. F. A. (2015). A method for web application vulnerabilities detection by using boyer-moore string matching algorithm. *Procedia Computer Science*, *72*, 112–121. The Third Information Systems International Conference 2015.

Seng, L. K., Ithnin, N., & Shaid, S. Z. M. (2018). Automating penetration testing within an ambiguous testing environment. *International Journal of Innovative Computing*, *8*(3).

Stats, I. W. (2021). Internet Growth Statistics 1995 to 2021 - the Global Village Online. [Online; accessed 25. Dec. 2021].

Wang, H. & Song, M. (2011). Ckmeans.1d.dp: Optimal k-means clustering in one dimension by dynamic programming. *The R Journal*, *3*, 29–33.

Wu, X. (1991). Optimal quantization by matrix searching. *Journal of Algorithms*, *12*(4), 663–673.

APPENDIX A

A realistic scenario for crafting and injecting SQL payloads to retrieve passwords in the table using Python

```
import string
import requests

url = ""  # this string to be initialized with the web applications handle
users = []  # this list to be initialized with all known user names from Table 1
chars = string.printable
passwords = {}
for userName in users:
    cracked_password = []
    no_new_letter_found = True
    password_found = False

    while (no_new_letter_found and not password_found):  # we don't know the length of the password

        for char in chars:  # trying for every possible char from our printable characters list

            response = requests.post(url, data = { "password" : " abc' or name='" + userName + "' AND BINARY password LIKE '" + ''.join(cracked_password) + char + "%' # " })
            # "BINARY password LIKE 'x%'" will match every password starts with "x"
            # "#" will comment out the rest of the statement

            webPage = response.text
            if ( 'user exists' in webPage ):  # user exist means, our query has evaluated to true
                cracked_password.append(char)  # so we know that our current trial was successful
                no_new_letter_found = False
                break  # break the loop for this digit, we already figured it out
        passwords[userName] = cracked_password

        password_found = True  # means we tried every possibility, and there were no further match

```

```
30 for k,v in passwords:
31     print(f"Username: {k}, and the corresponding password: {v}")
```