Designing and Evaluating Application Layer Web Threat Detection using Machine Learning Techniques

by

Tyler Wilding

supervised by

Dr. Salimur Choudhury

A thesis submitted in conformity with the requirements
for the degree of Bachelor of Computer Science (Honours)
Department of Mathematics and Computer Science
Algoma University

Abstract

Designing and Evaluating Application Layer Web Threat Detection using Machine Learning Techniques

Tyler Wilding

Bachelor of Computer Science (Honours)

Department of Mathematics and Computer Science

Algoma University

2017

This thesis examines the use of machine learning techniques, namely support-vector machines and genetic algorithms, for the purpose of detecting the following application layer web threats: SQL injections, cross-site scripting, and remote file inclusion attacks. Detecting these attacks becomes more important as the Internet grows and leveraging the strengths of machine learning is one of the many potential avenues in order to improve detection. The examination entails using the techniques to detect the aforementioned threats in a collection of unseen web request data and drawing critical conclusions about their strengths, weaknesses, and viability. Through this process several drawbacks to the genetic algorithm approach stood out, more specifically in its error prone detection and high variability of performance; and while the support-vector machine did solve several of these issues and produced great results it's complexity could be troublesome for real-world applications.

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

1.1 Problem Definition

As the Internet grows in usage both in the number and types of devices it serves the attack surface grows along with it. There are many attacks that are considered common and should always be a top priority when developing a new Internet connected application (Section 2). However, prevention is not the only task when dealing with these threats as detection is equally as important. Among many reasons, one scenario is if an attack known as a zero-day attack which has no established prevention method was just discovered and should be considered to already be causing damage. In this scenario, detection is the first line of defense to inform those who are in a position to take necessary actions to stop it.

Conventional means of detection typically involved manually creating detection signatures from known attacks in order to detect ones of a similar nature. However this creates multiple problems: the process of creating these signatures is slow and requires a reference attack, the created detection signatures will only detect attacks that are very similar to the reference, and finally it assumes the original attack will not change from its original state and can be redetected using the same signature (Section 3). Therefore

as of recently the focus has shifted to using other techniques to attempt to improve on the short comings of the conventional approaches by using machine learning techniques, such as genetic algorithms (Section 4). These approaches show great promise, but no system is perfect, and with the research being fairly new the results and techniques have not been as critically analyzed as much as the more established methods have been.

1.2 Objective

The objective of this research is first and foremost to more critically analyze the genetic algorithm approach presented in previous research, in addition to comparing it's performance to a new approach using a support-vector machine to classify the requests as threats or not threats instead. The critical analysis of both machine learning techniques will involve determining the success rate of detection and how often attacks are misidentified either as a false positive or the wrong attack type. Through this information a more complete picture of the performance of the algorithms can be developed and much more informed conclusions can be drawn from it. Another objective of the research is to ensure testing is done accurately and fairly between the two algorithms, this means testing and training with the same data across both techniques, and the test data is different from the training data. Lastly, in order to correct the requests which are represented with ordinary text into usable data, a simple parser must be made that is tailored to each attack type's nuances.

1.2.1 Expected Results

It is expected that the support-vector machine approach will outperform the genetic algorithm. This is due to the fact that the support-vector machine is a classification tool and appears to be much more suited for the problem as a result. In addition, the genetic algorithm relys on the mostly-random occurance of generating a new signature

in order to detect additional attacks which suggests it may have highly-variable levels of success. In contrast, the support-vector machine can get quite performance intensive when using more complicated kernel types and is more reliant on the training data. If using more complex kernels is required to achieve good detection than the applicability of the approach may not be there.

1.3 Thesis Overview

1.3.1 Scope and Limitations

This research is limited to only using genetic algorithms and support-vector machines for the application of detecting web threats and other machine learning techniques will not be examined. To that effect, it is also limited to examining the following three application layer web threats: SQL injections, cross-site scripting and remote file inclusion (Section 2.2, 2.3, 2.4).

The two techniques will be compared based on their detection results (success rate, false positives and incorrect detections), while time complexity and speed may be mentioned it will not be explicitly measured or recorded.

All tests will be carried out in a virtual environment using labelled data as to facilitate collection of results, this data will be collected using real automated exploit tools wherever possible.

Chapter 2

Web Threats

2.1 What is a Web Threat

At its most basic definition, a web threat is any malicious attack that uses the Internet as its main method of distribution, meaning the types of web threats is wide and varied. Web threats can be broken up into two main categories referred to as push or pull. Pull based threats are attacks that can affect any visitor to the website or service while push based attacks use luring techniques to get a user to fall victim to the attack such as phishing emails. The main motivation behind these attacks is the pursuit of confidential information and it is becoming increasingly commonplace to hear about large scale data breaches. While it is difficult to track all of the monetary gain from these activities due to their underground nature, there are some instances of millions of dollars being extorted from large businesses. As the number of users and the complexity of the devices attached to these networks increases so to it does the exploitability. Common web attacks can range from simple phishing emails, to malicious email attachments to malicious code injection directly into a vulnerable website and is a very serious problem.

Attackers will vary their methods and tools often, creating a situation where the attacker is always a step ahead of the prevention systems due to the unlimited number of

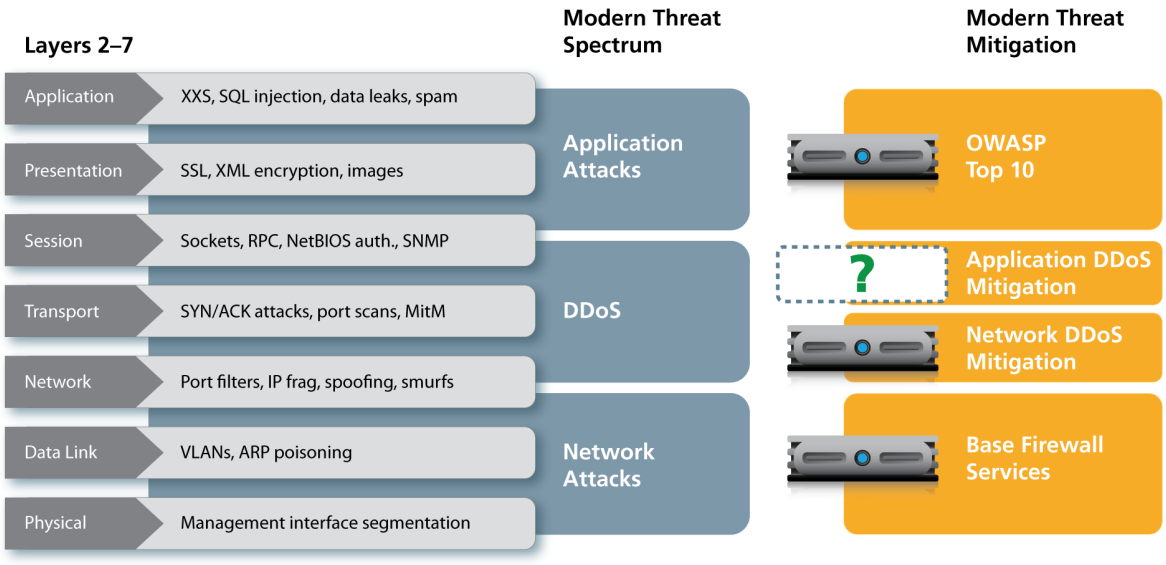

Figure 2.1: Possible threats at each OSI model layer and possible mitigation techniques. [2]

possibilities that have to be accounted for. This leads to the conclusion that conventional approaches grand-fathered in from other security sectors such as virus scanning are not adequate enough for web threat detection. There are two important reasons for this: first off the attacks are so varied in their approaches and transportation techniques that collecting samples to produce signatures for detection is not feasible; the second reason is unlike conventional viruses which are often designed to spread as fast as possible, web attacks are instead designed to go under the radar. In addition, web servers are required to be publically open to the world unlike traditional desktop computers which typically have all their ports closed, by accepting information from unknown parties by default many problems are created. Therefore, modern solutions employ a much more layered approach using conventional detection methods along with reputation systems, feedback loops, and other systems to increase the overall security (Section 3) [4].

Web threats can target any layer of the OSI model in order to exploit different weaknesses or perform different types of attacks for various reasons (Figure 2.1). Application attacks, which this research is focusing on, occur on the Application, Presentation, and

partially, the Session layers. While several of these attacks can be mitigated by preventing the OWASP Top10 security flaws the keyword is mitigation, there is always the possibility for a new attack variant to slip past existing safeguards.

Attackers will always prefer the most vulnerable target, the weakest link, and there is nothing more vulnerable than the public-facing application itself. Studies have found many of the existing techniques for handling these application layer attacks suffer from one or more of the following techniques [5]:

- Inherent limitations
- Incomplete implementations
- Complex frameworks
- Runtime overheads
- Intensive manual work requirements
- False positives and false negatives

This means that despite the fact that these attacks are commonly considered easy to mitigate, the existing solutions are not completely adequate in some aspect for defending against some of the most common web attacks. In order to solve this problem new methods of detection and prevention need to be explored and older conventional methods need improvement.

2.2 SQL Injection

Structured Query Language or SQL for short is the most dominant language for interacting with relational databases in recent years. A SQL injection is an attack where through various means arbitrary unauthorized SQL code is ran on the database. As the prime motivator for these attacks that was discussed was gathering confidential data, this is a very powerful way of acquiring said information [5]. There are four common methods of injecting SQL commands into an application:

- Injection through user input
- Injection through cookies
- Injection through server variables
- Second-order injections

Injection through user input is the simplest of the methods, the malicious user simply enters the arbitrary SQL code into any field that has some form of interaction with the database. Injection through cookies involves exploiting applications that read from the cookie's fields to restore the user's state, this information can instead be modified to contain SQL code which will be read on returning to the website by the application. Injection through server variables (such as PHP session variables, or other environment variables) are used in a similar manner to cookies by applications and can also be exploited in a similar fashion. Lastly, second-order injections are some of the hardest SQL injection to detect because they involve an attack that occurs at a later point after the initial exploit is entered [6].

The impact and purpose of SQL injections can also be broken down into four main categories. First, **confidentiality** is lost, databases that hold sensitive information which may be financial or identifiable is compromised and should now be considered public knowledge. Secondly, in addition to the data being potentially leaked publically, it now has also lost it's **integrity** as the malicious user can make any modification he/she wants. **Authentication** and **authorization** of the application is also broken, it is now possible to log in as any user with any access level, removing the need and purpose for passwords and safeguards.

While SQL injections are commonly utilized to simply retrieve the information contained within a database, there are much more dangerous things that can be done. For example, previous wide-scale botnet attacks have used SQL injections mimicking as Google queries in order to facilitate malicious drive-by-download malware attacks on websites [5].

2.2.1 SQL Injection Types

Often times many of these injection types are combined together in a single attack and are not necessarily absolute, but the techniques can be broken down and classified under the six following categories:

Tautology attacks are typically designed to bypass authentication, identify injectable parameters or extract data. It is often done by using conditional SQL statements to evaluate to true in the WHERE portion of the query. This will result in the query evaluating to true for every single row in a given table often returning all of the rows depending on how the application as well as the injection query is designed. For example, the '1=1' portion of the following query will always evaluate to true, and the remainder of the query is removed using the SQL comment character '–':

```
SELECT accounts FROM users WHERE login= or 1=1 -- AND pass= AND pin=
```

A **malformed** or invalid query are performed to gather information about the underlying database or applications structure in order to design targeted queries. This is due to the common mistake of having overly descriptive errors that reveal information that can be very useful to an attacker. This information can reveal information ranging from the DBMS used to the names of columns or tables. The following example forces a conversion error:

```
SELECT accounts FROM users WHERE login= AND pass= AND pin= convert
(int,(select top 1 name from sysobjects where xtype=u))
```

A third type of SQL injection exploits the **union** keyword which is used to combine rows of multiple tables together. This could be used for extracting data as you can combine the data of another table which you only know a limited amount of information but contains what you are interested in, with another table that is easily targeted. The following example combines the credit card information from another table with a null

table, returning only the credit information which we are interested in:

```
SELECT accounts FROM users WHERE login= UNION SELECT cardNo from
CreditCards where acctNo=10032 -- AND pass= AND pin=
```

Piggy-backed queries are where additional queries are added onto an existing query, this is the technique many people have heard about when learning about SQL injections because it involves ending the current query and beginning another. This is typically done in the following syntax: '; `< Piggy-Backed Query > --`'. The semi-colon signifies the end of the current query, a new query is added, and then the remainder of the query is commented out, here is an example:

```
SELECT accounts FROM users WHERE login=doe AND pass=; drop
table users --  AND pin=123
```

Stored procedures are typically designed to do something much greater than just extract data, and instead have the potential to do something much worse such as escalating the attacker in the database environment or a denial of service. Often times developers think using embedded procedures makes their code protected from injections as all of the queries are stored within the database environment, but this is not the case. Given the following stored procedure to check our login credentials, we can inject ' ; `SHUTDOWN; --`' and generate the following piggy-backed query.

```
CREATE PROCEDURE DBO.isAuthenticated @userName varchar2, @pass varchar2,
@pin int
     AS
          EXEC("SELECT accounts FROM users
          WHERE login=" +@userName+ " and pass=" +@password+ " and
          pin=" +@pin);
     GO
```

```
SELECT accounts FROM users WHERE login=doe AND pass= ;
SHUTDOWN; -- AND pin=
```

The last category is called **inference** attacks, these are used where malformed queries cannot provide the vital information needed to construct attacks. Instead, injections are performed and the website is monitored for changes or a response, this information is used to deduce vulnerable parameters and other information regarding the attributes in the database. There are two types of inference attacks, *blind injections* and *timing attacks*. Blind injections are posing simple true or false queries to the database, if the query evaluates to true then nothing will change, but a false evaluation will typically differ in some way. For example, the two following queries should both return an error if the input is handled properly, but if only the first one returns an error than the login parameter is vulnerable:

```
SELECT accounts FROM users WHERE login=legalUser and 1=0 --
AND pass= AND pin=0
```

```
SELECT accounts FROM users WHERE login=legalUser and 1=1
--  AND pass= AND pin=0
```

Timing attacks make use of the `WAITFOR` keyword to note the increase or decrease in the response time of the query instead of an error message. The following example is trying to extract table names by using a binary search method, if there is a delay in the response then the attacker knows how to adjust the search key.

```
SELECT accounts FROM users WHERE login=legalUser and
ASCII(SUBSTRING((select top 1 name from sysobjects),1,1)) > X
WAITFOR 5 --  AND pass= AND pin=0
```

It is also important to note that any of these attacks and the yet to be discussed attacks can also obscure their presence by using alternative encodings for the text that is entered, for example input can be converted to Unicode or hexadecimal [6].

2.3 Cross-Site Scripting

Cross-site scripting attacks or XSS for short are similar to SQL injections in that the arbitrary code is injected using user inputs but instead of influencing the back-end database directly they will leverage the public facing code of the application, more often the HTML or JavaScript. XSS attacks can be used for a variety of purposes, from redirecting users to other websites, to just simply changing the look of the website. In the worst case, XSS attacks can be used to hijack user's sessions to collect information from the user.

It is hard to pinpoint the level of impact of a XSS attack because it all depends on the intent of the attack. XSS attacks, unlike SQL injections can be just a minor nuisance or can be just as severe and collect sensitive information. Often times instead of a XSS attack being used alone, it is instead used as part of a larger scheme to send a user to another attack website where a phishing attack or something of similar nature lies [5][7].

2.3.1 Types of Cross-Site Scripting Attacks

The first type of XSS attack is called a **stored** or **persistent** attack. These are attacks that are permanently stored in a database, whenever a user access the page that retrieves the information from the database their browser runs the attack. A simple example is just inserting typical HTML code into a comment field, whenever someone views the comment they will see the output result of the injected HTML code.

The second type of XSS attack is referred to as a **reflected** attack. Instead of the result being stored in the server they instead originate from another source, sometimes an email link or another website. Upon clicking the link, the code runs as the browser

considers it to be from a safe source. These attacks are non-persistent because you would have to click the original link again for the attack to run as it is not tied to the page like in a stored attack would be, so other normal users cannot see it as well [8].

The final type of XSS attack is a newer distinction referred to as a **Document Object Model (DOM) based** attack. In this variant, the attack is ran by modifying the actual document model through JavaScript's document object. What separates this attack from a stored or reflected attack is the original page does not change, but instead the change occurs only on the user's client-side code, resulting in a different unintended result [9].

2.4 Remote File Inclusion

The final web-threat that will be examined in this research is remote file inclusion also known as RFI. It is defined as exploiting the vulnerabilities of the application to include remote files which may include arbitrary code of any particular language. The most common example is abusing PHP's include() command to get PHP code to run on the remote server. RFI attacks allow for code execution on the remote server and client-side, which can allow someone to remote shell into the server, leading to denial of service or data extraction to name a few potential problems [10].

While there are not any predefined variants for RFI attacks, for the purposes of this research RFI attacks will be divided into the following three categories:

- Only parameters with URLs
- Only parameters with PHP commands
- Parameters with URLs and PHP commands

Chapter 3

Current Detection and Prevention Methods

3.1 Prevention through Development

Web attacks can be significantly damaging to an organization in many ways and so detection and prevention is of very high importance. These attacks and subsequent problems are not only limited to only small time organizations as well, with many high profile websites are becoming victims to large but similar attacks as well. Some studies state over 90% of web applications are vulnerable to SQL injections alone [11]. Additionally, many web applications are vulnerable due to their usage of end-of-life software. PHP is the most dominant programming language used for websites accounting for over 80% of the market share, however of the websites that use PHP, 95% of them are still on version 5 [12]. As of the beginning of 2017 PHP 5 is no longer actively supported and it is recommended to switch to version 7 which is being actively supported, updating will bring with it all of the latest security fixes that could make a website easily exploitable otherwise [13][14].

The best way to stop these web attacks is to prevent the vulnerabilities from existing

in the first place, this is accomplished on the development side. Despite the fact that the majority of the attacks are well documented and understood, as safeguards are preventions are put in place to protect one aspect of an application, attackers shift their efforts to look for the next weakest area. From a security standpoint you need to assume your application is not bulletproof and you have only mitigated the risk but never removed it completely. As a result of this rationale, prevention alone is not enough as there is always the possibility that an attack finds a way past the safeguards and so detection must work hand in hand with prevention.

The simplest and most common method to prevent these attacks targeting the application layer of the web application is to never allow user input to be directly concatenated into any command that interacts with the server-side. This is done by making use of what is called prepared statements, you instead construct the entire query or command and then pass in your data from the user as parameters. This allows the server to distinguish between data and code no matter what kind of input is supplied by the user. Likewise, whenever data is accessed from storage to be displayed to the user, it should not be directly inserted into a command that could potentially treat it as arbitrary code. This will prevent issues such as stored XSS attacks or RFIs from occurring where stored code is inserted into the flow of the application code [15]. Many languages have different ways of accomplishing this but the simplest way is to strip out the portions of text that cause it to be interpreted as code, but a more comprehensive way is to filter the HTML against some form of filter or whitelist [16].

These are other measures that can be taken to mitigate the attacks but they mostly pertain to the environment on which the application is ran on and specifically for SQL injection prevention rather than all attack types. Separate database users should be made for each application and they should have the least amount of privilege possible. In the event something is compromised, that database is at least isolated and other applications are unaffected. A second measure that can be taken is to use views extensively instead

of using direct queries for all database interactions as it allows access to the tables to be denied and instead only to the specially tailored views. These strategies embrace the least privilege concept, where it is an unnecessary risk to be privy to more information or have more access than you truly need [15].

Research has been done on examining the code of potentially XSS vulnerable applications to determine their vulnerability and was able to accurately detect the vulnerable code with no false positives or negatives. So it is clear that modifying the code itself to be more secure should always be the top priority if it is possible to determine if a particular file is vulnerable that easily and accurately [7].

3.2 Signature Based Detection

A traditional and common way of detecting security threats is the use of signatures, however many of these signature based tools are more suited for the lower layers of the OSI model rather than the higher layers such as the application and presentation layers. These tools are referred to as Intrusion Detection Systems (IDS) and many rely on regular expressions and other pattern matching techniques with signatures produced using previous attacks methods, one example of such a tool is Snort [17]. Therefore, as long as there is an adequate number of signatures that cover the broad spectrum of possible attacks then the detection approach can be quite successful.

However, signature based detection systems as well as other IDS systems are not completely fail proof. One of the biggest problems is the frequency of false positives, or in other words when the system believes something that is not harmful is harmful. Of course the opposite is also true and IDS systems can let attacks slip by unnoticed, this can be a result of the attacks using various tricks to evade detection such as using alternate encodings or fragmenting packets or just simply being a new form of attack [18]. In addition, if the IDS is signature based than what is a more likely explanation for these

accuracy problems is a lack of signatures that are either more accurate or cover these new undocumented attacks. It is becoming too impractical to produce these signatures fast enough due to the countless variants of the attacks and they are commonly designed to go unnoticed rather than spread as fast as possible like with conventional malware [4]. To give an idea on how difficult of a problem this is to solve with signatures alone, an average of 5,000 new software vulnerabilities have been identified per year; along with the number of unique malware programs allegedly in the tens of millions and doubling every year. With these rapidly rising trends, it is clear that the malicious user is easily always ahead of the detection tools. Static solutions including signature sets are becoming less and less practical every year for these reasons, and the current short comings of the detection systems to keep up themselves proves that point further [18].

However, this problem is not unique to the web threat world, although signature based detection is much more suited for the traditional desktop computer application virus scanning practices have had to adapt as well to a similar problem. Virus scanning is probably the best example of signature based detection in action, where malware is collected, a signature is developed to detect it, and then sent out to the masses as fast as possible. However, some types of viruses have begun to exploit this by transforming their own code when transferring which would require an entire new signature to detect. These so called metamorphic viruses are not impossible to defeat but they require approaching the idea of scanning for a virus completely differently than just collecting a single signature. Such techniques include but are not limited to hidden Markov models or reversing the morphing process of the malware itself [19][20]. If the area where signature based detection has proven to be the strongest has had to adapt its tactics to deal with the changing security climate, then by extension so to it does the web threat detection ecosystem.

3.3 Modern Methods of Detection

In order to overcome these challenges for detecting web threats, some people have suggested a multiple layered approach will provide the best defense. Such a system would not only have multiple layers of detection involving signatures and other techniques but also feedback loops to update the protection systems for improved future detection. A multi-layered approach would also be able to address all levels of the network rather than a single system that can only handle a subset of the layers such as the network or application layers. This approach would also enable portions of the processing to be centralized and in the cloud while other areas are closer to the endpoints. Traditional techniques like signature detection would still be used, but it would be able to be augmented with additional techniques such as behaviour analysis as often times web attacks are carried out in massive enumeration attempts and not just a single bad request. One final point is such a solution would allow for global collaboration to contribute to reputation lists, whitelists, and the like to further solve the problem of a growing threat instead of having the same tools deployed in multiple areas and poor solutions for consistent updates [4].

A multi-layered approach combines the best of the old traditional techniques with new potentially better solutions, and even more interestingly suggests a system that can inherently learn and improve as a core trait; very similar to the technique in question in this research.

Chapter 4

Describing Machine Learning Approaches

4.1 Machine Learning

Machine learning has become a popular topic in recent years as an answer to the sheer amount of data that has the potential to be processed. Not only does the large amount of data require more advanced and intelligent ways to process it, but there is also a big incentives driving the desire to do so. Through machine learning techniques and patterns additional meaning can be extracted from the data which can be used to draw conclusions or predictions [21]. Machine learning techniques are commonly applied to data mining tasks, examining at data and discerning additional information from it. In the case of this research's problem area the data being mined is the large amount of web requests.

This brings numerous benefits to a security application such as web threat detection, primarily it allows the system to have some sort of feedback mechanisms to improve its detection and perhaps even its prevention. A system that is unable to learn overtime will not be able to overcome new techniques designed by attackers to evade the current detection systems abilities unless it is manually updated. And as discussed before, manual

updating is not only a very inefficient time consuming task but it is also just plain infeasible as the number and complexity of these attacks grows. Identifying patterns from data is very useful when it's considered that many attacks follow some kind of a basic syntax or format and while there are many ways to evade detection, typically there is still a common method or intent that can be identified.

The way machine learning operates is by identifying a series of features in the data set in question, these features are then used by the algorithm learn and make better decisions. The key distinction of machine learning is that the algorithms are not told what to do explicitly but instead is allowed to make its decision based on measurements such as performance [3].

4.1.1 Supervised Learning

Machine learning algorithms can either make use of supervised learning or unsupervised learning. In supervised learning the system is provided with labelled data, data that states what the end result should be so the system can be accurately trained without additional intervention. There is also another type of learning called reinforcement learning where an external source informs the system how well it is working or not as it progresses.

Supervised learning has issues that must to be overcome however, the first of which being collecting the original data set. If there is prior research or informed sources that can suggest what features to use, then the process is much more trivial, otherwise the features are often identified using a brute force method. The problem with using a brute force method other than the computation time is the data has the potential to be noisy and miss important features which can lead to further problems. Learning from extremely large datasets is very inefficient as well. Therefore, it is often desired to minimize the size of the data set while still maintaining the final performance of the system through a process is called instance selection. Lastly, having a large amount of features in the data set can also increase the complexity of the system. To solve this irrelevant and redundant

features should be removed whenever possible, but if many of the features depend on each other then they shouldn't be removed as this can lead to inaccuracies in of the learning process. To deal with features that can't be removed, new features can be constructed or existing ones altered to be more concise and accurate, improving the entire system as a whole. One of the big steps in creating a machine learning system is selecting the right algorithm for the dataset, each which their own advantages, disadvantages and times where they are applicable (Section 7).

4.2 Genetic Algorithm

A genetic algorithm is a search-based algorithm that makes use of machine learning in order to locate optimal or near-optimal solutions for a particular problem. This is what is meant by the term 'search' that is often used to describe this and other algorithms such as local search or simulated annealing, it is not about locating something in a particular data set but rather searching for the best possible answer. Such algorithms, especially in the case of a genetic algorithm use fitness functions and possibly reward systems to distinguish between a better solution and one that should not be considered going forward [22].

As the name suggests, a genetic algorithm mimics how genetic development in the real world works and how species evolve over time. The algorithm begins with an initial population of individuals, an individual being a possible solution to the problem in question. Every individual has it's fitness evaluated using some form of calculation tailored to suit the problem at hand; for our purposes for web threat detection the fitness will be

evaluated with the following:

$$fitness = (\frac{correct\ detections}{\alpha}) - (\frac{false\ positives}{\gamma}) - (\frac{incorrect\ detections}{\beta \cdot 8})$$
$$\alpha \leftarrow The\ number\ of\ possible\ correct\ detections$$
$$\beta \leftarrow The\ number\ of\ possible\ incorrect\ detections$$
$$\gamma \leftarrow The\ number\ of\ possible\ false\ positives$$

Algorithm 1: Fitness algorithm for use in genetic algorithm

Correct detections with improve the fitness of a particular individual and the individual is more likely to be selected for genetic operators later on, whereas false positives and incorrect detections impact the fitness negatively, with incorrect detections having less of a negative impact than the former. As an example, if we are looking for SQL injections, every deleted request that is a SQL injection is correct, if instead it isn't an attack at all then it is a false positive and if it is actually a XSS or RFI attack then this is an incorrect detection.

In order for the genetic algorithm to produce a new population it makes use of what are called genetic operators which commonly include performing crossovers and mutations to the individuals. Two individuals at a time in the population are selected by a selection algorithm and crossed over, there are many ways to perform a crossover but a single-point crossover will be the method used for this research. A position is selected to perform the crossover, referred to as a locus, for our purposes this refers to selecting a segment, this segment is then swapped with the other selected individual's respective segment to produce two new individuals with unique chromosomes, or in other words different configurations. This process continues until the algorithm has produced enough new individuals to fill the next population, in addition, at the beginning of this process it is possible to mark some of the top individuals as elite and bring them into the new population unaltered. Before continuing to the next generation every single allele, or piece of information in each individual has a small chance to be mutated, this is what causes the population to have some sort of diversity. This entire process then repeats

many times, each time being referred to as a generation and by the end of the process there should be a set of individuals that are closer to solving the problem [23].

4.2.1 Current Genetic Algorithm Solutions

These genetic algorithm techniques have been applied to web threat detection already, one particular paper focused on using variants of an attack to detect network related attacks. While they may not have directly used a genetic algorithm in their solution, the core idea is very similar to how a genetic algorithm operates with generating different individuals and seeing if they fit a certain criterion. These exploit variants were used to test signature based detection methods to see if it was possible to evade them and results showed it was. This is proof that traditional models for detection cannot be made absolutely perfect and that using an approach similar to genetic algorithm techniques is worthwhile for at least evading detection [24].

As of recently research has taken this idea and done the opposite, using a genetic algorithm directly to detect web attacks rather than to evade detection. The genetic algorithm was used to make variants of attack detection signatures that can best detect SQL injections, XSS, and RFI attacks through the text-based web request logs. The results of which were very promising with around a 90% detection accuracy reported which exceeded the performance of a traditional regular expression signature based detection system [17].

These recent findings are the starting point for this research, improving the genetic algorithm approach and gathering more detailed results about it's function and performance, as well as being the comparison point for another machine learning technique, support vector machines.

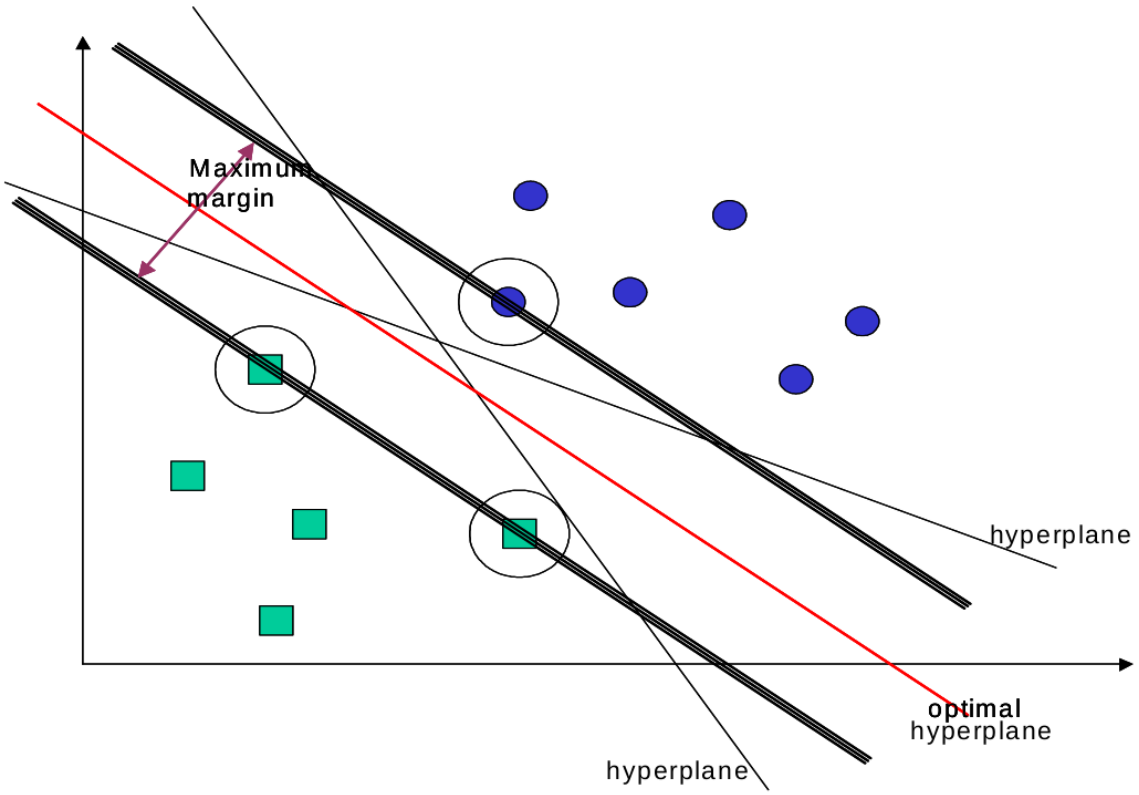

Figure 4.1: Example of a linear seperated SVM [3]

4.3 Support Vector Machine

A support vector machine's main technique for classifying data is to divide the data set into two or more categories with the largest margin possible between the separation(s), referred to as a hyperplane(s). The reason for maximizing this buffer between categories is to reduce the chances of classification error as much as possible. With the hyperplane(s) computed, points that lie within the margin are referred to as support vectors, hence the name, and it is these points which were used to calculate the hyperplane(s) in the first place with the other data points being ignored (Figure 4.1).

The fact that the SVM is determined by only the support vectors which are usually a very small subset of the training vectors, is great because it means the speed does not significantly slow down with a larger amount of features. However it is also realistic to

imagine data that cannot be easily divided, this can be solved using soft margins that allow for misclassifications or in more complex cases the data can be mapped to a higher dimensional space to open up other options of dividing the data. This higher dimensional space is referred to as the transformed feature space, however simple linear separations in the higher dimensional space transform into non-linear separations when you return back to the original space.

If this feature space data is mapped to a Hilbert space referred to with Φ, which allows for traditional Euclidean space vector calculations to be extended to an infinite amount of dimensions using dot products using equations in the form of: $\Phi(x_i) \cdot \Phi(x_j)$. This means we can use what is called a kernel function to avoid ever having to determine the mapping to Φ and calculate the necessary results directly in the feature space instead. A kernel function is in the form of: $K(x_i, x_j) = \Phi(x_i) \cdot \Phi(x_j)$ [3]. There are many commonly used kernels, three of which will be used in this research: linear, polynomial (with degree 3), and radial basis function (Table 4.1).

Kernel Function	**Mathematical Formula**
Linear	$K(x_i, x_j) = \langle x_i, x_j \rangle$
Polynomial	$K(x_i, x_j) = (\langle x_i, x_j \rangle + 1)^d, d : degree$
Radial Basis Function (RBF)	$K(x_i, x_j) = \exp\left(\frac{-\|x_i - x_j\|^2}{2\sigma^2}/\right), \sigma : width\ of\ RBF\ function$

Table 4.1: Kernels that will be used and their mathematical function [1]

Once the SVM is trained using a kernel method of choice the only step left is to pass in all of the testing data and see which side of the hyperplane(s) it falls in order to classify it. An SVM can at times get very computational intensive and can run very slow, this is often due to the choice of the kernel function as well as the parameters that influence the kernel. For example, a linear kernel is very simple where as an RBF kernel is much more complex. Two parameters that are worth mentioning for the SVM training process are γ (gamma) and C.

Gamma is used in the polynomial and RBF kernels to define how much influence each

training vector has on the separating hyperplane(s). A lower gamma value corresponds to a far influence, when gamma is too small the influence of any support vector may extend to the entire set and the end result would instead just be regions of high density being isolated from other high density areas. On the converse if gamma is too high then the influence would only extend to the support vector itself. The C parameter is the penalty cost associated with misclassification, if C is low then classification is more relaxed compared to a higher C which will encourage more support vectors to be chosen and achieve a more accurate division [25]. There is no way to know what are the best values to choose for gamma and C because it depends on the dataset in question and so this must be done by doing testing.

4.3.1 Current Support Vector Machine Solutions

Support vector machines have also been applied to web threat detection as well but only in the lower layers of the OSI model dealing with network related attacks such as denial of service. One such study used a cost based support vector machine to detect web attacks and was able to detect them with an overall accuracy of 99% [1]. Likewise, another study compared the usage of an SVM versus an artificial neural network and found the SVM was much faster in comparison along with achieving a 99% accuracy as well [26]. The results of these two studies show that the SVM approach is a viable one and can be used for practical applications with detection web threats, so it will be interesting to see how well the algorithm performs for application layer attacks as well as how it compares to the genetic algorithm approach.

Chapter 5

Methods & Procedures

5.1 System Overview

Despite the fact that two rather different algorithms will be used the system is designed to operate more or less the same by implementing the genetic algorithm and support vector machine components as loosely coupled modules to avoid having to redesign the system completely. Web requests will be processed through a parser that searches for various aspects related to each of the three possible web attacks. These results become output and can be used as input for either the genetic algorithm, support vector machine, or potentially another algorithm that could extend the research even further. Finally, the machine learning modules will output the results to a file that can be processed by graphing tools (Figure 5.1), in this case the R programming language will be used to create graphs of the results that can be used to drawn conclusions on the two approaches.

5.2 Gathering Test Data

All data will be gathered from as close to a real-world scenario as possible and in order to do so, automated enumeration exploiting tools will be used to gather a large sample size of varied attacks (Table 5.1). In order to gather SQL injection attacks the popular

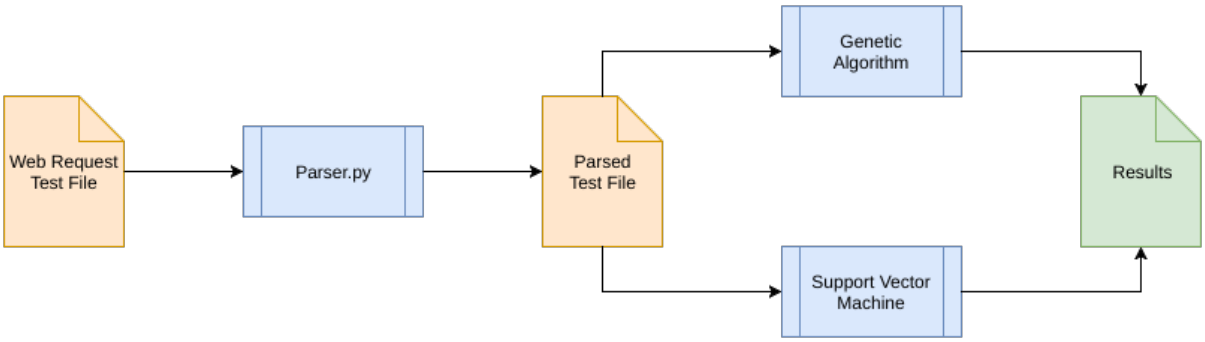

Figure 5.1: Overview of the system

Request Type	Generation Method
SQL Injection	sqlmap [27]
XSS Attack	grabber [28], xsser [29]
RFI Attack	randomized generation
Normal Requests	httpfox [30]

Table 5.1: Breakdown of test data generation

tool *sqlmap* will be used, for XSS attacks *grabber* and *xsser* will be used.

These tools will be let loose on a private apache web-server that is hosting a very simple database connected application. However, for gathering a large amount of RFI attacks it will require generation of a large sample size as the tooling for these types of attacks is rather limited and the tools that do exist do not perform large scale enumeration attacks and instead attempt to compromise the server as fast as possible [31]. Because RFI attacks are the simplest in terms of their design and variations, a simple automated Python script can be used to generate a large amount of attacks using a heavy amount of randomization. Finally, it is necessary to also include some normal web requests which are not attacks to test for false positives. This can easily be done by using the application as well as other websites normally, submitting form inputs, etc., and collecting all of the resulting HTTP requests. This will be done with a simple browser extension for Mozilla Firefox, HTTPFox.

Like most web-servers, Apache has the ability to log all of the requests it serves to a file. The data that we need to parse for the genetic algorithm and support vector machine

Request Type	Features
SQL Injection	# SQL keywords, is encoded, # fields containing SQL keyword, attack variant
XSS Attack	# of HTML or JavaScript keywords, is encoded, # fields containing a HTML or javascript keyword, attack variant
RFI Attack	# of URLs, is encoded, # of commands, attack variant

Table 5.2: Parseable Features

is the GET or POST HTTP request line. Therefore, the final step of gathering the testing and training data is to compile the log files and strip out all unneeded information so it can be passed to the parser. Another small Python script will be used to generate a test file of the correct size and proportions using these large banks of requests. The script will generate a file where each line contains the request line content and what type the request is, either SQLi, XSS, RFI, or not an attack this labels the data for supervised learning and performance evaluations.

5.3 Parsing the Requests

With the completed test file(s) containing the proper amount of each attack and normal requests, the next step is to parse each request into numeric values so the algorithms can work with them. Each request has their own respective features that are worth identifying, with the genetic algorithm requiring more features to be identified than for the SVM. These features have been identified for each request type by previous research (Table 5.2) but is important to distinguish what the meaning of each is, as well as what can and cannot be parsed [17].

The number of SQL keywords or reserved words is obtained by using comprehensive lists obtained from the Oracle and MySQL DBMS documentation [32][33]. This works well with our test environment as well, as the DBMS our application is using is MySQL and many of the automated tools will use MySQL specific vulnerabilities. Similarly, the HTML and JavaScript keywords were obtained from the official W3C

```
https://duckduckgo.com/?q=HTTP+Request&t=vivaldi&ia=web
```

Figure 5.2: A sample HTTP request with fields highlighted

documentation[34][35][36], as well as the PHP related commands were sourced from the PHP documentation [37].

Requests are also capable of being encoded as well to further evade detection (Section 2) which is something that can be easily detected and recorded. HTTP requests, usually GET requests specifically, will contain fields and information that the request carries to the application code. This information can be user supplied information (ex. a username or password) or application supplied information (ex. the current page number), either way it is able to be directly modified by a user and is where injections and malicious code is likely to lie (Figure 5.2). For this reason, it is good to make a distinction between just a total number of keywords found and the number of fields that actually contain keywords to get a more complete picture of the request. Lastly, all of the discussed attack variants (Section 2) can be detected with the exception of *Stored Procedure SQL injections*, which brings up the limitations of this type of parsing (Section 7.1.1).

The parser makes heavy use of regular expressions (Appendix C) to determine much of this information such as the keywords or contents of the fields and is designed to overcome common evasion tactics. One such tactic is padding the alternate encodings of the request, instead of using the common two-byte hexadecimal conversion of the ASCII values, several zeros can be appended to the two bytes to confuse simple parsers using built-in decoding libraries. Another issue that had to be overcame was not double-counting keywords that were a prefix to another keyword, which is done simply by associating each word with its prefix combinations. This results in only a minor amount of extra

computation as there is not many keywords with prefixes.

```
Data: File with HTTP Requests and their true type
Result: Resulting test file with every request stored along with the parsed
        features for the three types of web threats in the following order:
        Original → SQLi → XSS → RFI
read in input file;
for line in input file do
    if SVM testing then
        disregard encoding and attack variant features;
    end
    pass request to each parsing module (sql, xss, and rfi);
    store original request and type in resulting file;
    for each parsed result do
        if for genetic algorithm then
            if lengths of segment 1 and 3 should be permuted for length testing then
                for each segment length combination up to specified maximum do
                    convert result to binary based on the maximum lengths of each
                     segment;
                    if decimal value exceeds maximum value in segment then
                        use maximum allowed value in segment;
                    end
                    store result in a list;
                end
                store complete list on a new line;
            else
                convert result to binary based on the maximum lengths of each
                 segment;
                if decimal value exceeds maximum value in segment then
                    use maximum allowed value in segment;
                end
                store complete bitstring in file on new line;
            end
        else
            store decimal values into file on new line;
        end
    end
end
```

Algorithm 2: General overview of parsing procedure

Full documentation on the usage of the parser (Appendix B.2).

5.4 Genetic Algorithm Based Signature Detection

The method of the using a genetic algorithm for signature based detection is largely the same as the proposed system from previous research with a few modifications [17]. One major difference is instead of allowing the signatures to change to different attack type signatures (ex. SQLi to XSS) the type of attack we want to search for is specified and the algorithm uses the respective parsed result from every request. This of course requires every original request to be parsed three times instead of just once, but it makes the most sense as if it is possible to transition between attack types so easily then there is no reason to differentiate between them in the first place. This also allows later results and subsequent conclusions to not be influenced by other factors that cannot be measured. If signatures were allowed to switch to different attacks types, then it would be unknown if the results were due to the random nature of a genetic algorithm causing signatures of the wrong type to be chosen or the other variables in question being changed.

The second major difference is the bitstring length for each signature is increased to avoid problems of exceeding the quantity in a segment. In the previous research only 3 bits were used for the segments that count the number of segments or fields in the requests which only allows for a count up to 7. In a real world setting the amounts for the number of fields and keywords can be much larger and exceeding these values often creates a situation where there are many signatures that match which normally would not. For example, if two signatures have 10 and 11 keywords respectively, with the old segment lengths they would be capped at 7 (111 in binary), resulting in a match which should not have occurred. Therefore these segment lengths with a size problem have been extended to 6 bits allowing for counts up to 63 (Table 5.3). Lastly, the fitness algorithm has been altered so false positives and incorrect detections affect the fitness negatively as these are aspects of a detection system you do not want Algorithm 1).

The genetic algorithm implementation is very standard, allowing for the following parameters to be changed (Appendix B.3):

Request Type	**Segment Information**			
SQL Injection	# of SQL Keywords	is encoded	# of fields containing a SQL keyword	attack variant
	6	1	6	3
XSS Attack	# of HTML or javascript keywords	is encoded	# of fields containing a HTML or javascript keyword	attack variant
	6	1	6	3
RFI Attack	# of URLs	is encoded	# of PHP commands	attack variant
	6	1	6	3

Table 5.3: Genetic algorithm default segment breakdown

- Maximum population per generation
- Maximum number of generations
- Mutation rate
- Elitist selection amount

Most of the genetic operators are fairly simplistic to implement with the most complex being the selection algorithm. The higher the fitness of the individual the more likely it will be selected by the selection algorithm to be crossed over. There are several ways for the selection algorithm to be implemented but for this research Roulette Wheel Selection, also referred to as Fitness Proportionate Selection, will be used (Algorithm 3). This algorithm was chosen because the proposed genetic algorithm technique from existing research was fitness based [17], not reward based like some other selection algorithms. In addition to it being very simple and practical to implement and understand considering

the scope of the research.

Data: Fitness values of all individuals in population
Result: The selected individual
$totalWeight = 0$;
for *all individuals weights* **do**
  $totalWeight = totalWeight + weight$;
end
$rand = arandomnumberbetween0andthetotalWeight$;
for *all individuals weights* **do**
  $rand = rand - weight$;
  if $rand < 0$ **then**
    return that individual's index;
  end
end
if *Unable to find an individual* **then**
  error occured, fall back condition is to return the last item;
end

Algorithm 3: Basic pseudocode algorithm for Roulette Wheel Selection in $\mathrm{O}(n)$

The genetic algorithm was written from scratch using the Python programming language and is designed to handle input files produced by the parser containing lines of single bitstrings or with several variations of the segment lengths (Algorithm 4).

Data: Bitstrings for training and testing and all parameters for genetic algorithm
Result: The optimized bitstrings that can be used for detection
for *each bitstring of varying segment length* **do**
  for *the number of generations* **do**
    remove duplicate bitstrings in the current population;
    evaluate fitness for all individuals;
    preserve the top elitist percentage into the new population called the offspring;
    while *offspring amount < maximum population allowed* **do**
      select two individuals and perform a single point crossover;
      add these two newly individuals to the next population
    end
    trim the offspring to the maximum population;
    loop through every allele in every offspring with the chance to mutate;
    set the current population to the offspring
  end
  store the bitstring signature results for that length
end

Algorithm 4: Pseudocode algorithm for genetic algorithm

Request Type	Number in Sample
SQL Injection	300
XSS Attack	300
RFI Attack	300
Non Attacks	100
Total	1000

Table 5.4: Breakdown of the training file for genetic algorithms

Request Type	Number in Sample
SQL Injection	1500
XSS Attack	1500
RFI Attack	1500
Non Attacks	500
Total	5000

Table 5.5: Breakdown of the testing file for both genetic algorithm and support vector machine

5.4.1 Testing Procedure

In order to the test the genetic algorithm fairly, both training data and testing data will have equal proportions of all three attacks, 30% for each attack and then 10% of non-attacks for false positive measurements (Table 5.4 & 5.5). In addition, the testing data contains different requests than that used in training as this would be the situation that these approaches would be used for in the real world. They would be trained using supervised learning and then used to identify unlabeled data entering the system, therefore it does not make sense to test with the same data it was trained on. These testing files can be automatically constructed using a small Python script (Appendix B.1).

Each test of the genetic algorithm will use the same training and testing data, instead the parameters of the genetic algorithm will be altered to observe any difference in the performance. The genetic algorithm will first be trained which will output bitstrings that will act as signatures that have been optimized for detecting the particular attack type. Those signatures will then finally be used to match with the unseen testing data's

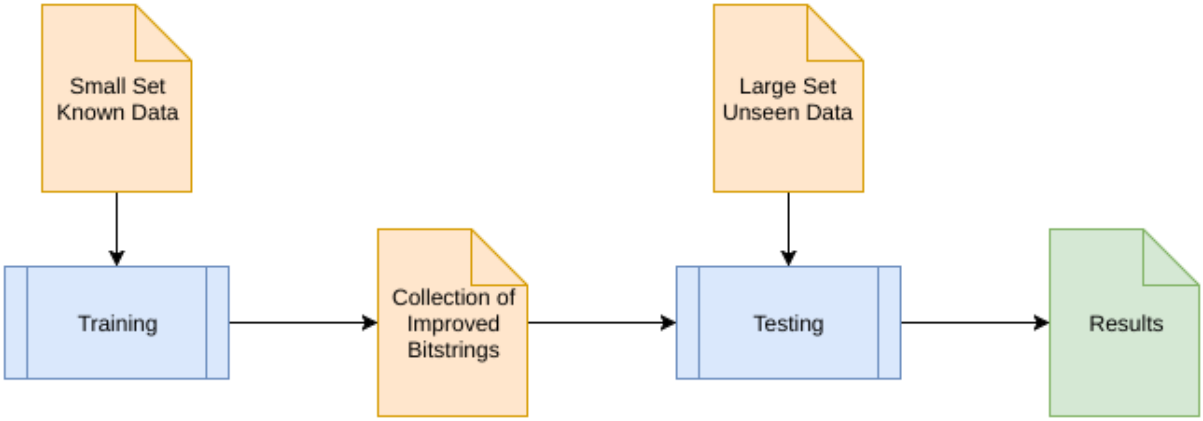

Figure 5.3: Overview of the genetic algorithm system

bitstring representations it believes are the attacks (Figure 5.3).

In addition to adjusting the various parameters of the genetic algorithm, using multiple iterations of the algorithm to generate a combined signature set will also be tested. The belief is that with the additional signatures in the detection set the more likely it is to have a signature in order to detect more of the attacks, so this test will see if that is the case as well as if it has any negative side effects. This is also one of the purported benefits of this technique, that the genetic algorithm can generate additional signatures for detection automatically instead of relying on manually created signatures. Lastly, testing the performance using different segment lengths of the bitstrings will also be conducted, the reasoning for this goes back to the issue of overflowing a segment (Section 5.4). Smaller segments should be able to detect more attacks as there are fewer possible combinations, making it easier to generate bitstrings that cover a wider range of attack signatures. In the extreme case, a segment that can only hold a maximum count of 1 would detect anything if it contained a keyword even if the request contained much more than 1, essentially becoming a flag.

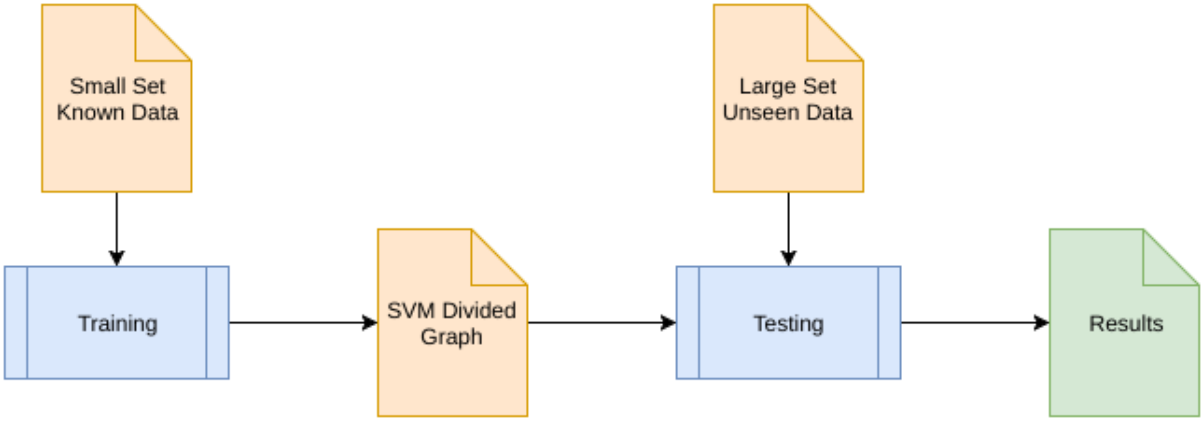

Figure 5.4: Overview of the support vector machine system

5.5 Support Vector Machine Detection

The SVM detection approach will follow a similar process to the genetic algorithm but rather than changing the parameters of the algorithm, the training data will be manipulated instead (Figure 5.4). This is due to the fact that the parameters for the SVM that matter for performance are automatically optimized by using a grid search provided by the same Python library used to implement the SVM, scikit-learn [38]. The implemented support-vector machine's usage has been documented like all of the other tools for this research (Appendix B.4).

The parameters that are optimized by the grid search are gamma (γ) and the penalty cost (C), however gamma only makes a difference in the polynomial and RBF kernels. Support-vector machines use various different kernel methods to determine the separating hyperplane in the given data, the kernels that will be used from the provided library are: linear, polynomial with degree 3, and RBF. Every test will be executed using all three kernels to see which kernel is best-suited for the data sets.

The one major difference between the SVM and genetic algorithm approaches is the SVM only requires two of the four parsed segments and they can be in normal decimal representation as mutations need not be (Table 5.6). The reason that the other two segments are not used by the SVM is because they do not vary nearly as much as the

Request Type	Segments	
SQL Injection	# of SQL Keywords	# of fields containing a SQL keyword
XSS Attack	# of HTML or JavaScript keywords	# of fields containing a HTML or JavaScript keyword
RFI Attack	# of URLs	# of PHP commands

Table 5.6: Support Vector Machine segment breakdown

other two. Many attacks and non-attacks could be encoded or not, as well as being the same attack type, so that information is not very descriptive on its own. These two values are plotted on a simple X, Y plane which is then passed into the SVM to be trained, so having more than 2 values would make this step more complicated and is unnecessary. Each of these X, Y pairs are labelled with either as an attack or not an attack in order to facilitate supervised learning (Algorithm 5).

Data: Segment information for each request
Result: A trained SVM classifier that test data can be passed into and results gathered from
gather all segment information from training set;
pack into a numpy array;
for *each kernel type ('linear', 'polynomial-3', 'rbf')* **do**
  if *that kernel type has not already had its parameters optimized* **then**
    optimize using a GridSearch;
    store the resulting parameters to save time on the next repeat use of the kernel;
  end
  build the svm using scikit-learn and the optimized parameters;
  train the classifier using the training vectors and targets;
  pass all testing data through the classifier and record results;
  plot the resulting graph for visual purposes;
  store results of testing;
end

Algorithm 5: Pseudocode algorithm for support vector machine

5.5.1 Testing Procedure

For the SVM there are several ways the training data will be adjusted to measure how the performance changes (Figure 5.4). The exact same testing set that was used for the

genetic algorithm will be used and for smaller tests the same training data will also be used but once tests require beyond the 1000 request amount, new training data must be added. The first test case will use the same proportion of 30% for all three attack types and 10% for non-threats, this test will essentially be a fair comparison between the SVM and the genetic algorithm approaches. The second test is designed to see if false positives can be reduced by increasing only the amount of non-attack requests in the training set between the various tests. And lastly, seeing if the number of incorrect attack detection can be reduced, incorrect being identifying a request that is an attack but is not the type we are trying to detect, by increasing only the amount of requests containing the type of attacks we are **not** looking for.

Chapter 6

Results

6.1 Genetic Algorithm

In the following results each test was repeated three times and averaged to give a better idea of the typical performance of the genetic algorithm approach. Each test is categorized by the parameter that was changed as well as the reason and expected results of the changes to the parameters. Also, with the exception of the bitstring segment length test, all other tests are conducted using the normal segment lengths (Table 5.3). Lastly, for brevity only the SQL injection results will be compared across all test cases as it is the most complex request out of the three, a full listing of the graphical results for the other two attack types as well as complete text based results for all attack types is included (Appendix D).

6.1.1 Finding Best Parameters

The performance and effectiveness of the genetic algorithm can be attributed to the parameters that are used and there is no universal choice for the parameters as it will always depend on the data set in question [39]. Each parameter of the genetic algorithm will affect the results in different ways, therefore it was important to first determine

what settings would be most suitable to use for later tests so that those subsequent results would not be heavily influenced by the parameters instead of the independent variable being manipulated (Table 6.1).

Test	Population	Generations	Iterations	Mutation Rate	Elitist Pool
Population Size	x	100	1	0.5%	5%
# of Generations	1250	x	1	0.5%	5%
Mutation Rate	1250	100	1	x	5%
Eltitist Pool	1250	100	1	0.5%	x
Multiple Iterations	1250	100	x	0.5%	5%
Bitstring Length	1250	100	1	0.5%	5%

Table 6.1: Parameters used in each Genetic Algorithm Test

Population Size

Population size is one of the most important parameters because a genetic algorithm with a low population size performs very poorly due to there not being a large enough sample size to grow and advance from. Larger populations are more likely to generate new individuals that perform well however this also comes at a performance cost as every individual that is added is another individual that must have its fitness evaluated [40]. In addition, for our purposes having a larger population increases the

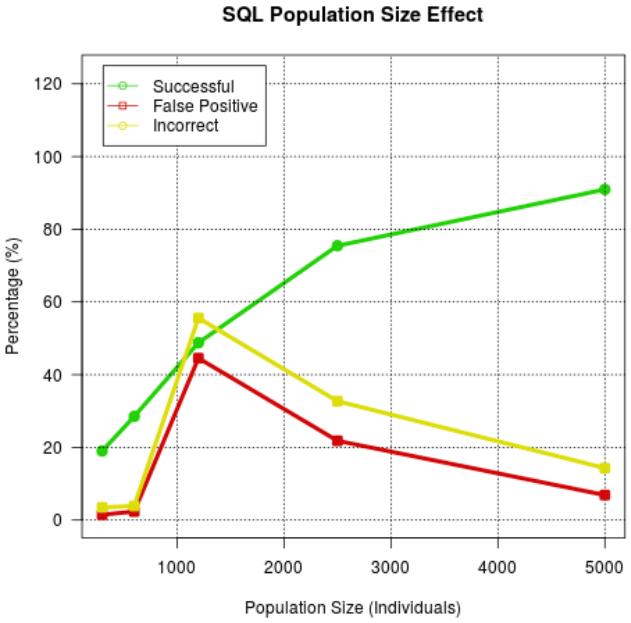

Figure 6.1: Effects of Population Size on Detecting SQLi

chances of having signatures that perform badly as well, causing increased false positives and incorrect detections. In the worst case every attack in the testing set will be unique and require a new signature so a population size any less than that amount may miss attacks (Figure 6.1).

Generations

The amount of generations also significantly matters because it is the amount of times the genetic algorithm will run. The more generations, the more likely the algorithm can produce better results and improve upon the existing results (Figure 6.2).

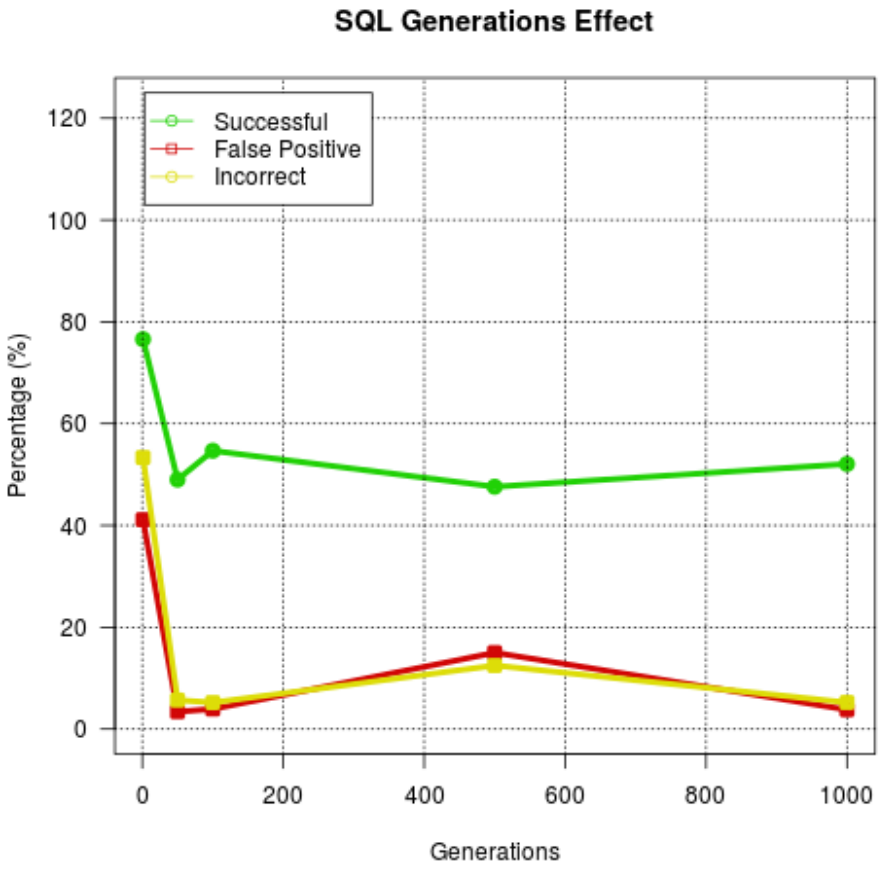

Figure 6.2: Effects of Generations on Detecting SQLi

Mutation Rate

If the mutation rate is too high then the genetic algorithm basically becomes a random search, while having no mutation rate at all will result in a lack of genetic diversity, with new bitstrings only being produced due to crossovers. It is very difficult to determine a single mutation rate to get the best results and more often than not this is done by trial and error like the following tests will perform [41]. Mutation rates are typically set to a very low amount since it is on a per allele basis and you would not want to randomize every single bit in a signature and so a value between 0.0 and 1.0% is often used with 1.0% being quite high (Figure 6.3) [40].

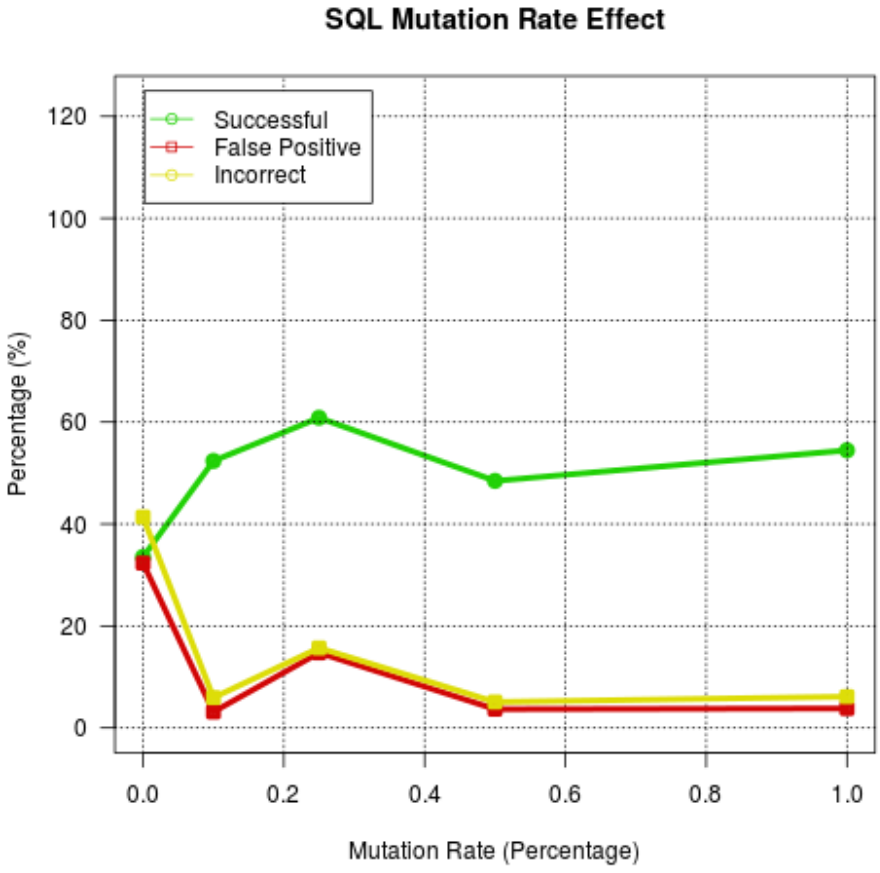

Figure 6.3: Effects of Mutation Rate on Detecting SQLi

Elitist Pool

The more of the better performing population that survives to the next generation the more likely strong individuals will be selected to produce new individuals of similar strength. However, if too much of the population is preserved than it may not be able to improve very quickly and the search may stagnate. Sometimes this is also referred to as a generation gap (Figure 6.4) [40].

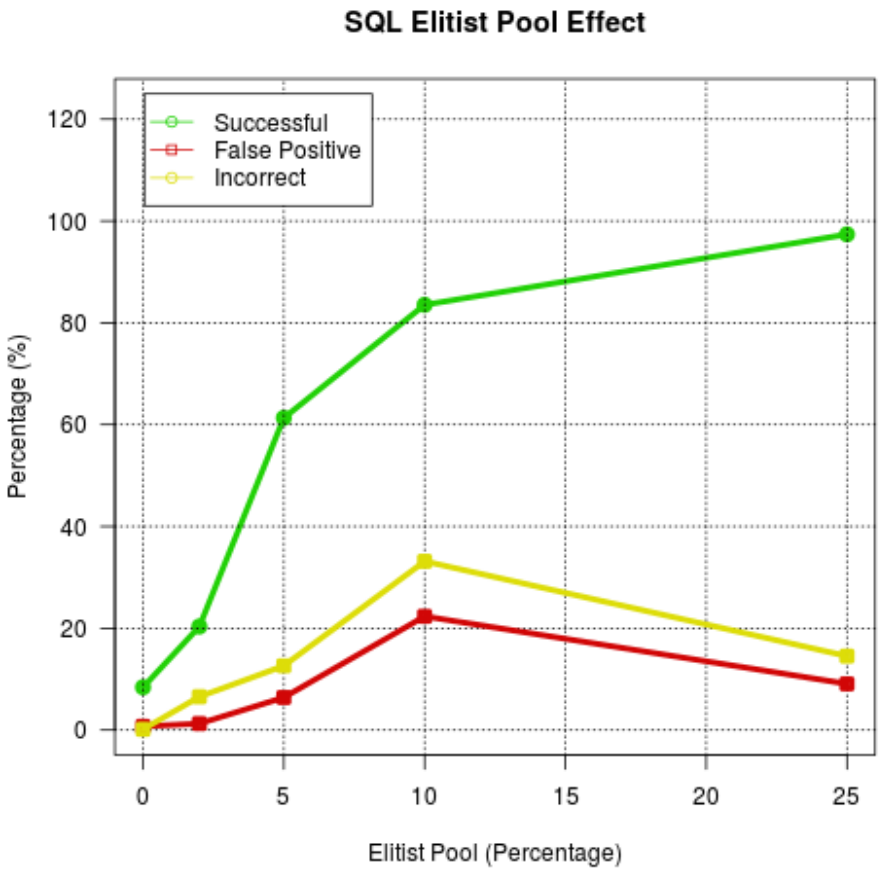

Figure 6.4: Effects of Elitist Pool on Detecting SQLi

6.1.2 Combining Multiple Signature Sets

One of the claimed advantages of the genetic algorithm approach, and a big reason why it can work well is because the genetic algorithm can be used to produce new signatures in order to keep adding to an existing set to increase the breadth of detection. For this reason, running the algorithm multiple times and combining the signatures should result in more detections, but may also result in increased false positives and incorrect detections due to many poor signatures being stored along with one another (Figure 6.5).

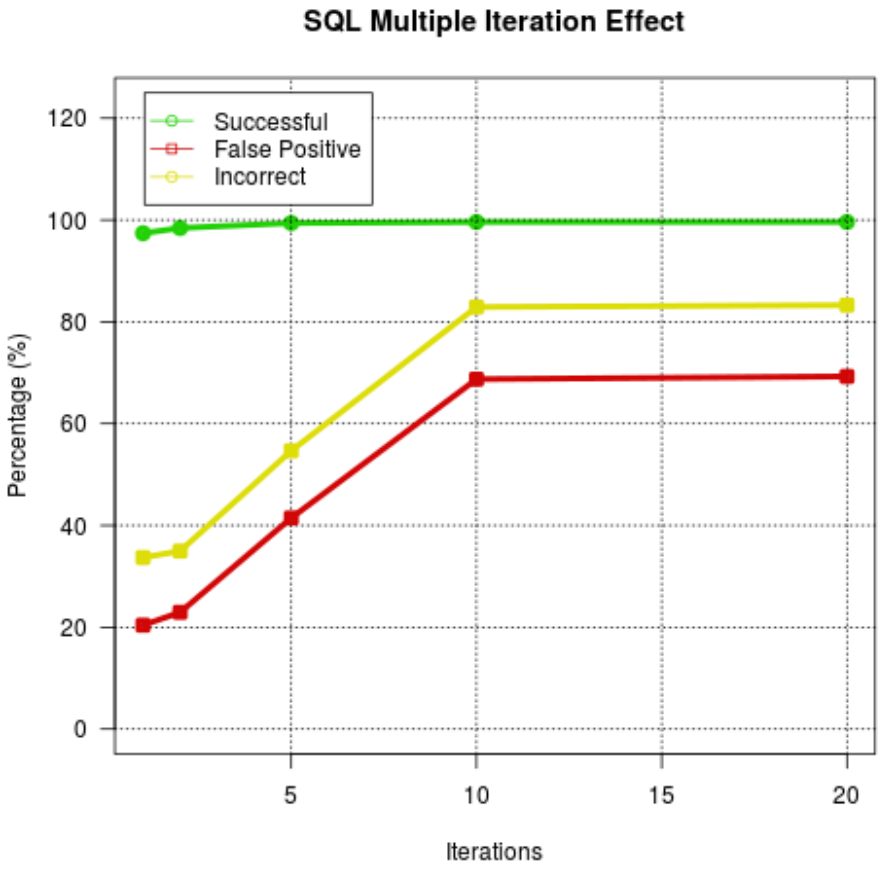

Figure 6.5: Effects of Multiple Iterations on Detecting SQLi

6.1.3 Bitstring Segment Length Effects

Because the genetic algorithm is able to detect additional attacks by generating new signatures, if the number of possible signature combinations is less thanks to the segment lengths, then it would be more likely to generate these signatures that match with the training or testing attacks. However it also opens up the possibility of making it easier to generate poor signatures due to a smaller search space, as well as potentially artificially increase results due to segment overflows (Figure 6.6).

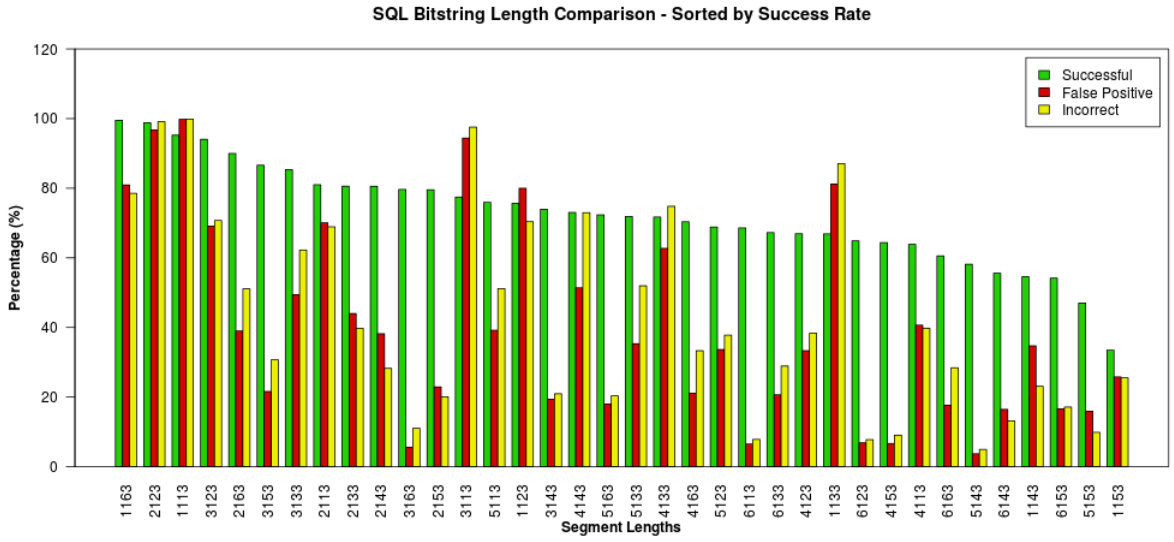

Figure 6.6: Effects of Different Segment Lengths on Detecting SQLi

6.2 Compared With Random Permutations with Fitness

The genetic algorithm approach works because of the ability to randomly generate new signatures using fitness to weed out the bad signatures so it would be very interesting to compare the approach with just simply generating all possible combinations and only using the bitstrings that performed well using the same fitness algorithm used in the genetic algorithm and avoiding the complexity and computation time that comes with a genetic algorithm (Algorithm 1), (Figure 6.7).

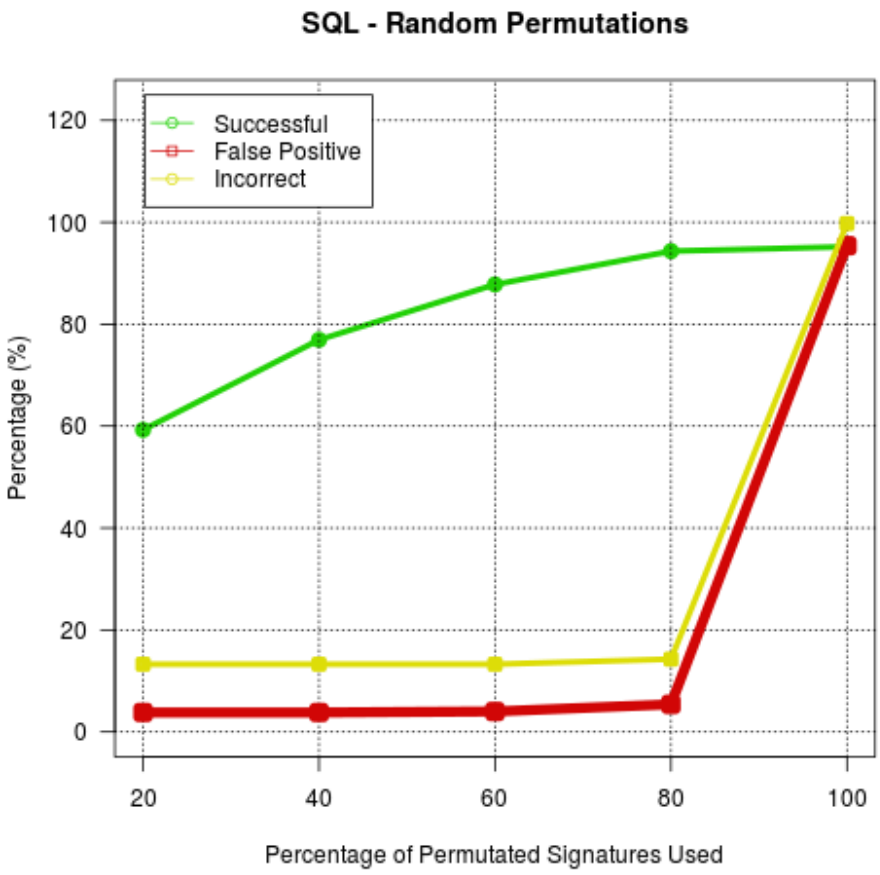

Figure 6.7: Permutation of Bitstrings to Detect SQLi

Test	**# of requests of intended detection type**	**# of requests of incorrect detection type**	**# of requests of non-attacks**
GA Compairson (30%/30%/30%/10%)	x	$2x$	y
Increasing Non-Threats	300	600	x
Increasing Incorrect-Threats	300	$2x$	350

Table 6.2: Parameters used in each Support Vector Machine Test

6.3 Support Vector Machine

For the testing of the support vector machine it was not necessary to average together multiple results as there are no random elements in the algorithm and so the same results are produced every time. All tests used the same testing data to verify the training process as well as the same training data whenever the required amount of requests did not exceed the amount of used in the genetic algorithms training set. When doing the genetic algorithm testing it was possible to measure changes by adjusting the algorithm's parameters but in the case of the SVM this is not the case and so instead the training data will be manipulated to observe the changes in performance.

6.3.1 Comparison with Genetic Algorithm

The first results are obtained by using the same proportions of attack types used in the genetic algorithm training set of 30% for each attack type and 10% remaining is non-threats. The purpose of this test is to create as fair a comparison as possible with the genetic algorithm (Figure 6.8). The output classifier for the best performing instance of each kernel is also included (Figure 6.9).

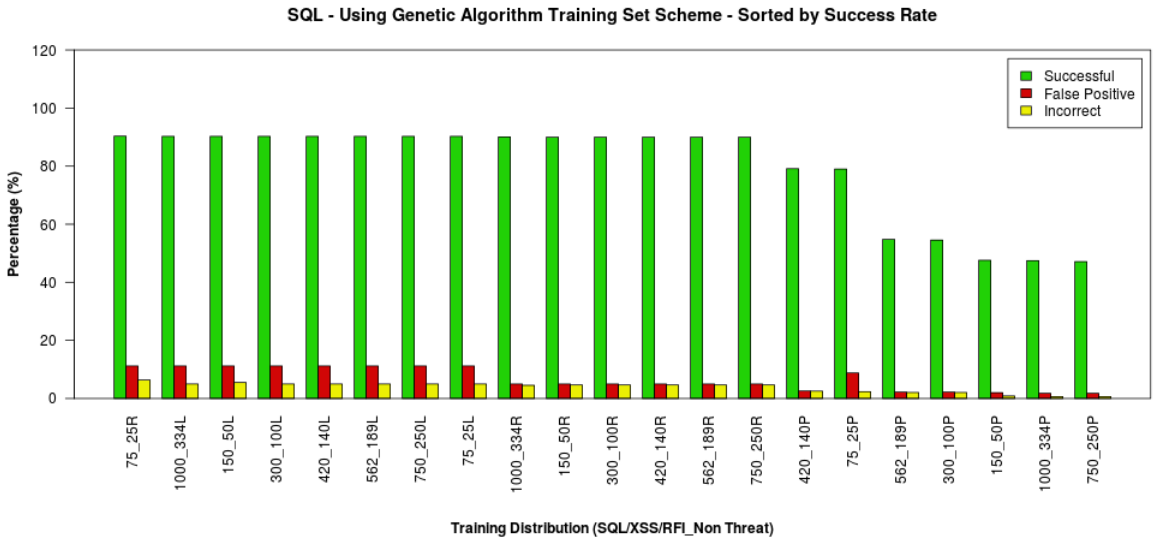

Figure 6.8: Genetic algorithm and SVM comparison for SQLi

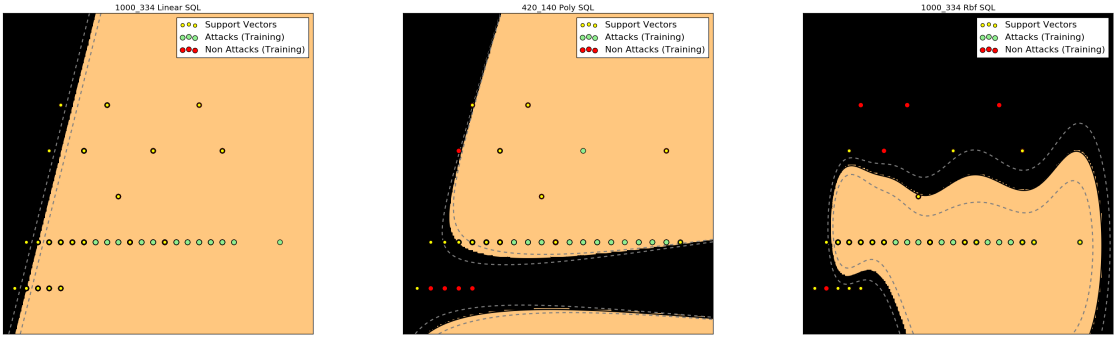

Figure 6.9: Classifier Output, Linear: 1000_334, Poly: 420_140, RBF: 1000_334

6.3.2 Increasing Non-Threats

Increasing the amount of non-threats in the training data should create a classifier that is more resilient to detecting false positives, the more training data the less likely false positives should occur (Figure 6.10).

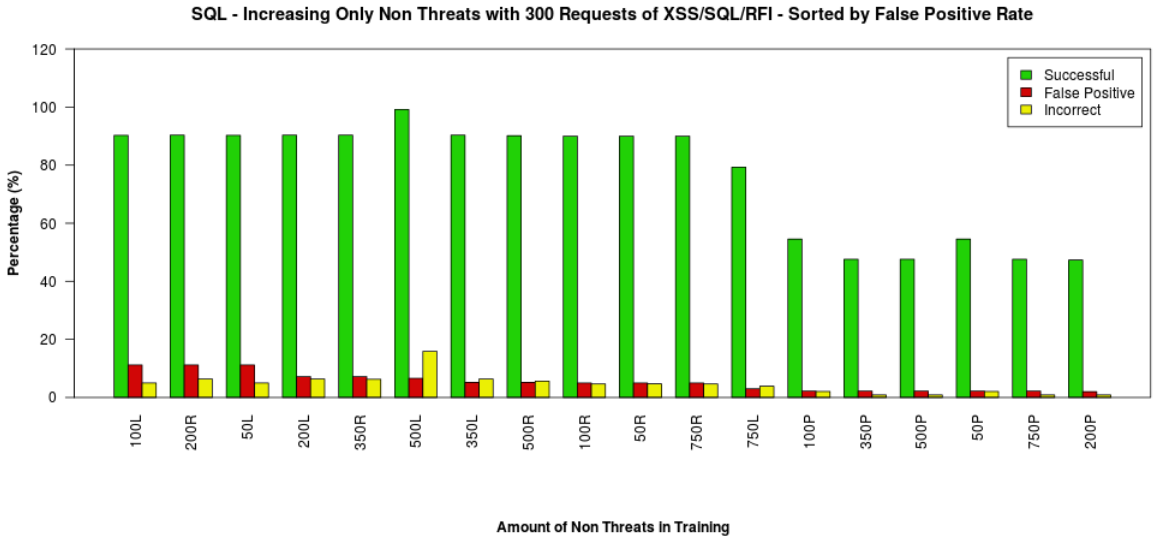

Figure 6.10: Effects of increasing non-threat training data in SVM for SQLi detection

6.3.3 Increasing Incorrect-Threats

Similar to the last test, the more threats in the training data that are not the one we are looking for, hopefully the less likely it is for the classifier to incorrectly identify an attack (Figure 6.11).

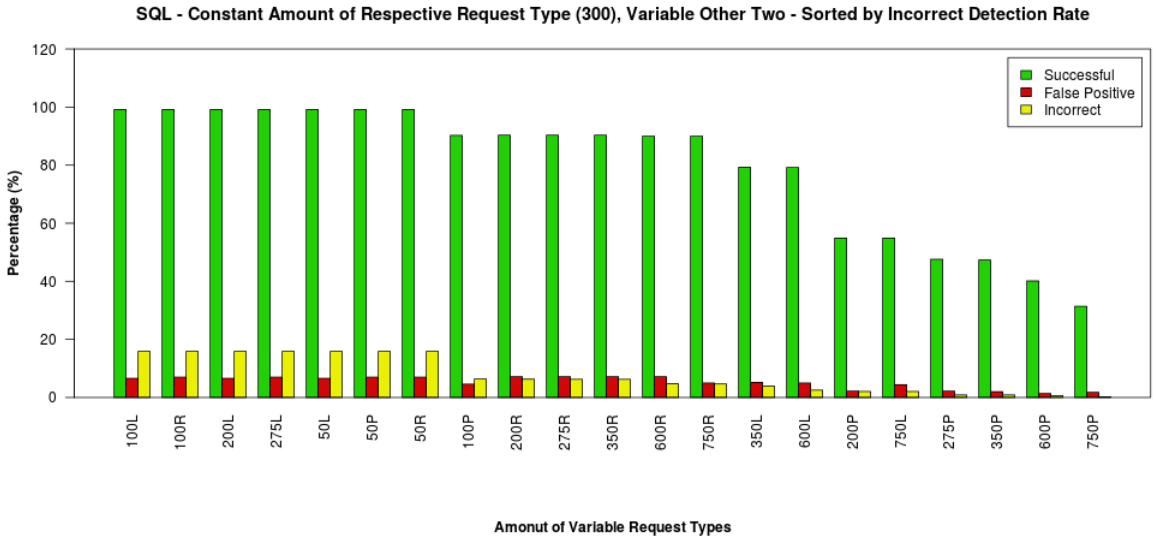

Figure 6.11: Effects of increasing incorrect attack training data in SVM for SQLi detection

Chapter 7

Discussion

7.1 Parser

7.1.1 Weaknesses

The parser is the main point of failure for both of the approaches and therefore many of the parser's weaknesses become apparent when measuring the performance of the system. If the output of the parser is not accurate than it is impossible to achieve consistently good performance either from the genetic algorithm or support vector machine. Although, both algorithms have been given the same parsed results so the question becomes which algorithm can do a better job with the same input.

In the end, despite the fact that both approaches at least to avoid the use of regular expressions and pattern matching, they both depend on a parser to get the input needed for the respective algorithms and as a result there is always a chance the parser has been fooled or in accurate and influences the performance. The question then becomes how good is the training process of these machine learning techniques in order to weed out these mistakes from the parser. Part of the reason why the parser may be vulnerable to making these mistakes it only employs a simple token-based approach. The parser cannot understand the logic or intent behind the requests by only picking up on keywords and

fields within the requests. If the parser was instead made using other data structures such as abstract syntax trees or program dependency graphs, then it may be possible to get a more complete picture of the request lead to greater accuracy. This limitation is quite obvious as some attack types as well as even variants of the attack types covered such as *stored SQL injections* cannot be detected as the information required is outside of the scope of the text request (Section 5.3).

A final point of weakness is some attacks were much easier to parse than others. SQL injections were definitely the most complicated of the three in addition to the keywords essentially being English words so there is always the potential of mistaking a normal word for an SQL word. In contrast, the XSS and RFI keywords must follow a particular syntax so these attacks do not suffer as much from this problem. The RFI attacks on the other hand had to be randomly generated in order to get a significant sample size for training and testing and so there are some inherent biases there. All of these issues can skew find the results either a positive or negative manner.

7.1.2 Strengths

Despite some of the problems that could potentially be solved by moving away from a token-based approach, it is the fastest of the parsing techniques as building trees or dependency graphs adds a lot of extra computational time. Additionally, for the purposes of detecting the selected features for each attack type a token-based approach is really all that is required to count up keywords and fields. Using a more complicated technique than this might be simply overkill and reduce the overall speed of parsing when attempted to be used in a real world environment.

Despite the weaknesses previous mentioned, parsing the results for these features is realistically the only way to determine the intent of the incoming application layer attack. There is no other useful information to pull out of the request text and no other deductions can be made on the intent of the request because it has not yet been executed.

7.2 Genetic Algorithm

7.2.1 Parameter Testing Results

In regards to changing the population size the expected results were observed. As the algorithm was allowed to generate additional signatures per generation the detection rates increased and false positives and incorrect detections declined but did not always disappear completely. This is the most obvious result as a greater population is needed in order to detect additional unique attacks, however the computational cost goes up along with it. In addition, increasing population size alone will not result in perfect detection, as none of the attack types were able to achieve near 100% successful detection and in the case of the RFI and XSS results, the detection rate actually went slightly down in the later tests (Section 6.1.1, Appendix D.2.1).

Increasing the number of generations did not seem to have any discernable effect on the performance of the algorithm and in many cases it made the results worse and more variable. Therefore the number of generations alone is not enough to optimize the results of the algorithm seen in an extreme example, the XSS detection, where it goes from a very high successful detection rate to 0% with more generations (Section 6.1.1, Appendix D.2.2). Every generation the entire population is thrown out depending on the elitist pool, meaning that while the previous generation might have great signatures, the generated offspring could be terribly performing signatures as there is no correlation between the old and new signatures (Section 7.2.4).

Mutation rate changes also had almost no discernable effect on the performance as well, in some tests the results improved while in the RFI detection for example the results flat line. Mutation rate only slightly effects the genetic diversity and the of the performance is instead mostly determined by the generation of signature through crossovers, mutation rate alone is not significant enough to save the algorithms performance (Section 6.1.1, D.2.3).

Finally, the elitist pool changes had the best results out of all the tests in terms of its impact on the performance in a positive way. For all three attack types, the more of the best performing individuals that were preserved the better, as most likely it allowed the transitions from generation to generation to be much more consistent and not be left up to pure chance (Section 6.1.1, Appendix D.2.4).

7.2.2 Expanded Signature Set

Doing multiple iterations of generating signature sets allowed for all three attack types to be able to hit near 100% successful detection, which was not a result seen using any other tests so far. With the exception of the RFI attack results most likely due to its inherent biases (Section 7.1.1), XSS and SQL injection attacks increased their false positive and incorrect detections drastically unfortunately. Having multiple iterations clearly causes this problem as the larger the signature set becomes the more likely it is to contain a signature that performs badly due to the random nature (Section 6.1.2, Appendix D.2.5).

7.2.3 Influence of Segment Length

Out of all of the test cases however, the segment length variations test is the most telling of the flaws of the genetic algorithm approach. It is quite obvious from the results that smaller segment sizes result in higher success rates, false positives, and incorrect detections on average when compared to the larger segment sizes (Section 6.1.3, Appendix D.2.6). This matches the expected result as not only does having a smaller segment size lead to inaccurate bitstrings that become artificial matches, but it also makes it much more likely for a poor performing bitstring to be generated and get through the fitness process (Section 7.2.4).

7.2.4 Strengths and Weaknesses

The genetic algorithm approach does indeed seem to be able to achieve the claimed success rates and successfully generate new signatures can at times perform very well. However, proof of the high variability and very poor of false positives and incorrect detections rates, can be found everywhere in the results. It seems the approach works best when given existing well performing signatures, this can be seen through the results of the elitist pool testing. Albeit in these tests it did not achieve the best success rates seen in other tests. This is likely explained due to the use of fitness proportionate selection, which can cause the search to either stagnate or converge too quickly; by maintaining high performing bitstrings the chances of those bitstrings ever leaving the population decrease [42]. Lastly, the genetic algorithm has a lower dependence on the accuracy of the parsed results compared to the SVM because the results are more of a result of the random generation of signatures rather than making improvements in the training process.

These results overall show the approach in quite a weak light and it seems ill-suited for this application. The first of these flaws is how highly variable and random the approach is. In order for an attack to be detected, a signature matching its parsed result must be generated, however the fitness algorithm does not rate how close the bitstrings are to this matching signature, rather it rates how well the signature detected multiple results and avoided mistakes. It would be impractical to use a genetic algorithm to do this however, as it would be much easier to just construct signatures from parsed results of known attacks instead. Therefore, the fitness algorithm's intent seems to instead be focused on keeping the better performing bitstrings in the population until the end of the algorithm and hopefully kicking out the bad performing bitstrings; however, this does not always work out as was explained. This is ultimately the root cause of the variability between generations and what causes odd anomalies in some of the results. For these reason, the genetic algorithm seems to be a bloated random permutation generator that

uses a fitness algorithm to remove the poorly performing bitstrings, and so this claim was tested (Section 7.2.5).

In addition, segment lengths of the bitstrings are very important. If the bitstrings are too small there is a higher chance to generate a poorly performing signature as the search space is much too small. To give an example, the proposed bitstring length was 10 bits which means 1024 unique combinations, meaning that since having duplicate bitstrings is redundant and they are removed, if the population size is around 1000 it is very likely that a bitstring representing a non-attack or incorrect attack type is generated and the fitness algorithm will not be remove it until the next generation. Likewise, having such small segment lengths are not applicable to the real world, enumeration attacks can use dozens of keywords per segment and having such a small count in each segment results in artificial matches that would not occur otherwise. For example, if every single SQL injection attack had a unique number of keywords but at least 7 per request, and the segment only had 3 bits to hold the number of keywords then the algorithm is only required to generate a single bitstring to detect **all** of them. The signature is not truly matching with the attack, instead it is artificially padding the find results of the techniques.

7.2.5 Comparison with Random Permutations

The results of using simple random permutation generator is quite telling of the poor performance of the genetic algorithm. Not only are the results much more predictable and non-volatile, but by using 80% of the well performing randomly permutated bitstrings the success rate was over 80% on all three attack types with minimal false positives and incorrect detection, a result that could not be achieved with the genetic algorithm. In addition to these excellent results, this approach also completed in a fraction of the time as the bitstrings only had to be randomly generated once, and had fitness evaluated once rather than multiple times over again in the genetic algorithm along with additional

selection, crossover, and mutation procedures.

This demonstrates how it really is just the random permutation process that is giving the genetic algorithm the results it has achieved, and therefore the genetic algorithm component is completely unnecessary. While the amount of permutations that must be generated exponentially grows with the size of the bitstring signature, the size used for the important segments that maintain the count of keywords and fields is double the size of the proposed length from previous research.

7.3 Support Vector Machine

7.3.1 Comparison with Genetic Algorithm

With the results from these tests it is quite clear which is the better approach out of the two, and that would be the SVM. All three attack types achieved a successful detection rate between 90% and 100%, but also with very minimal false positive and incorrect detection rates, only showing up in the SQL injection results infact (Section 6.3.1, Appendix E.2.1).

7.3.2 Reducing False Positives

Increasing the number of false positives in the training set did indeed decrease the rate of false positive detection overall in the tests for SQL injections, as the other two attack types did not have substantial amounts of false positive detection to begin with. This shows that training does indeed directly affect the performance of the SVM (Section 6.3.2, Appendix E.2.2).

7.3.3 Reducing Incorrect Detections

Similar to the results of the false positive training, the more incorrect results input into the training data, the better the SVM was able to avoid detecting them. However as overall these results were not as much of an improvement, this may be due to the larger amount of training data having parsing mistakes that ended up effecting the results (Section 6.3.3, Appendix E.2.3).

7.3.4 Strengths and Weaknesses

Overall, the SVM approach appears to be a much more suitable choice when compared to the genetic algorithm and even the random permutation trial approach. One of the advantages is that the SVM can be trained once and then new attacks can be passed into it without the need for re-training or checking with thousands of signatures to see if any are a match. It also is able to run much faster if the proper kernel is choosen, as the RBF kernel produced good results but is quite complex and slow, the polynomial kernel as well is slow but produced very poor results in comparison showing that this is not the ideal kernel. Therefore, the linear kernel which is also the fastest of the three and performed very well overall, seems to be the ideal and an overall improvement, otherwise the kernel complexity would be a major weakness of the SVM. In addition, the SVM requires fewer features to be parsed from the request, only needing the keywords and the field counts, the attack variant and encoded flag are not required speeding up the parsing process as well.

The main weakness of the SVM approach is its strong dependence on the accuracy of the parsed results. If the results from the parser are not correct then the algorithm can not hope to be trained properly and produce accurate output. But considering that the same training data was used in both approaches and the SVM came out majorly on top and these same results can be reproduced, it does not seem to be too much of an issue for the performance. Lastly, despite the mentioned biases with the RFI parsing, the fact

that the SVM achieved no false positives or incorrect results for RFI detection is not a result of bias, as the XSS results went the same way and were all generated through real automated attack scripts.

Chapter 8

Conclusion

8.1 Conclusions

To reiterate the purpose of this research, current methods of detecting web threats by and large make use of static solutions especially for application layer web threats. These application layer web threats include such attacks as: SQL injections, cross site scripting, and remote file inclusions which can all be incredibly dangerous to an organization if gone unchecked and account for some of the most notable data breaches in recent years. Therefore, it would be ideal to have a dynamic solution that can improve with the rapidly changing security climate of these web threats, using techniques proposed that employed genetic algorithms served as a starting point for the research.

The genetic algorithm approach claimed to greatly surpass the traditional signature based systems and similar results were achieved to verify these claims, however issues with the approach began to surface as well. With the high amount of misclassifications and the high variability of the results of the genetic algorithm was beginning to seem less and less attractive. The advantage of generating new signatures through the use of the random selections, crossovers, and mutations is ultimately what the genetic algorithm brings to the table. However, when tested to see if the performance could be replicated

by singling out these advantages from and simply generating combinations of bitstrings, the results were surprisingly very similar and improved. If very similar results could be achieved using a fraction of the computational cost of the genetic algorithm, and by some metrics even out performing then the future for using genetic algorithms for this purpose and in this way doesn't seem to be bright.

On the contrary, the support-vector machine approach seemed to be a very good potential candidate as it is designed for this particular type of problem, a classification problem. It was expected from the beginning that the SVM would be able to outperform the genetic algorithm, and through the results and analysis it was demonstrated that the SVM was able to better optimize for avoiding misclassifications while still maintain a high success rate. Although, the stellar performance of the SVM brought into question the reliance on the results of the parser for both algorithms, not just the SVM, which in some ways defeats the original purpose of avoiding the use of signatures and pattern matching in the first place. However, the features that the parser must identify are much more general and simple than a complex attack itself so this reliance may not be that much of an issue.

In conclusion, through the use of machine learning techniques it is indeed possible to detect application layer web threats including SQL injections, XSS attacks and RFI attacks. By utilizing the classification power of a support-vector machine it was shown to be possible to achieve upwards of 90-100% successful detection with a very minimal amount of misclassification, outperforming the genetic algorithm approach from previous research. This is due to how a SVM truly learns with each training input to refine its classification of the parsed results, rather than using machine learning as a mechanism only to produce random permutations that allows the SVM to come out on top.

8.2 Future Work

In regards to future work, it would be interesting to explore how these approaches could be improved with the use of more advanced parsing techniques such as abstract syntax trees or program dependency graphs. This may allow for a deeper understanding of the code within the requests, perhaps even allowing additional attack types to be detected beyond SQLi, XSS, or RFI. Additionally, being as the SVM approach seems to be the best of the options examined, additional research looking into how to further optimize its speed and perhaps utilize it as a feedback mechanism in a more fully-fledged detection and prevention system to see how it would be implemented in a real world setting, would be an interesting topic for future research.

Appendix A

Source Code

All source code and testing data sets are available online open-source under the GPLv2 license. External Link: https://github.com/xTVaser/web-threat-thesis

Appendix B

System Documentation and Usage

B.1 Test Generator

This small script takes in all of the files with the individual web attack types, and combines them to create a valid test file.

For training it will divide the file into two segments, 30/70% of the file. The first 30% is the initial set for the genetic algorithm but it will use the entire file. For the support vector machine it doesnt matter as it will only pass through the file once.

Training size = 1000 requests

Testing size = 5000 requests

Usage:

`TestGenerator.py -s SQL_Training -x XSS_Training -r RFI_Training -n NT_Training -sp 30 -xp 30 -rp 30 -np 10 -t -o TEST_Training`

`-s` file containing sql injection attacks, line seperated

`-x` file containing xss attacks, line seperated

`-r` file containing rfi attacks, line seperated

`-n` file containing non-attacks, line seperated

`-sp` percentage makeup of the file for sql attacks

`-xp` percentage makeup of the file for xss attacks

`-rp` percentage makeup of the file for remote file inclusion attacks

`-np` percentage makeup of the file for non-attacks (false positive purposes)

`-t` specify the test is for training

`-f` specify the test is for testing

`-o` output file path and name

B.2 Parser

This small script takes in a file that holds all of the requests for the test and will extract the various metrics from each. For example: number of keywords, number of urls, attack type, etc.

Usage:

```
RequestParser.py -f pathto/input.txt -ga -b 4444 -p -d -o output
```

`-f` file input with requests, line seperated with each line being in the syntax

`-ga` parse the bitstrings for the use in a genetic algorithm.

`-svm` parse the bitstrings for the use in a supper vector machine algorithm.

`-b` specify the max length of each of the 4 segments in the bitstring (genetic algorithm only) Each segment can be a max length of 8.

`-d` the first bitstring will be printed out as a decimal number instead of binary.

`-p` permute the bitstrings to get all possible combinations up to the max length.

`-o` output file name, the algorithm will be appended to the end automatically.

B.3 Genetic Algorithm Documentation

Driving script used to run the genetic algorithm on the given input and produce output. In real usage, GATest.py is used to perform automated testing. The GA will run through

all the combinations and output a file for each combination, for the number of iterations, etc.

Sample Usage:

`GADriver.py -p 100 -g 10 -m 0.5 -e 5 -i 1 -b 36 -f Length_Training_GA -o Test_GA`

`-p` Define the maximum population size per generation

`-g` Define the number of generations to compute

`-m` Percentage chance to mutate an allele

`-e` Top percentage of current population to carry over unchanged to the next generation

`-i` Repeat the genetic algorithm (i) times to generate more than the maximum population size at the end

`-b` Specify the number of multiple combinations of the same bitstring length excluding the decimal representation

`-f` File name containing requests and parsed bitstrings

`-o` output file name, the algorithm will be appended to the end automatically.

B.4 SVM Documentation

Script that builds an SVM for every detection each of the three attack types, and each of the three kernels given training input and determines results from testing input. In real usage, TestSVM.py would be used to call this program to allow for automated testing of a large amount of results.

Sample Usage:

`SVM.py -s 1000 -x 1000 -r 1000 -n 350 -f Training/New/ -o 75_25`

`-s` The number of SQL attacks to include in training

`-x` The number of XSS attacks to include in training

`-r` The number of RFI attacks to include in training

`-n` The number of nonthreat attacks to include in training

`-f` Directory containing the 4 threat files for training

`-o` Output file name

Appendix C

Regular Expression Documentation

Attack Type	**Purpose**	**Regular Expression**
SQLi	Find all instances of a keyword	`A-Za-z{?&</]" + keyword + "[^A-Za-z}.>]`
SQLi	Find all fields	`(\?|&)[\w\d]+=`
XSS	Find all fields	`(\?|&)[\w\d]+=`
XSS	Find all instances of attributes	`<.*"+ attribute +".*>`

Table C.1: Descriptions of Regular Expressions Used

Appendix D

Full Genetic Algorithm Testing Results

D.1 Full Text-Based Results

D.1.1 Population Size

Population Size	Successful Detection Rate (%)	False Positive Rate (%)	Incorrect Detection Rate (%)
300	19.00	1.46	3.48
600	28.51	2.33	3.86
1200	48.77	44.53	55.57
2500	75.46	21.80	32.68
5000	90.95	6.86	14.30

Table D.1: Text based effects of population size on SQLi detection

Population Size	Successful Detection Rate (%)	False Positive Rate (%)	Incorrect Detection Rate (%)
300	2.28	0.06	0.02
600	23.68	0.06	0.02
1200	39.04	0.13	0.03
2500	55.71	18.53	25.1
5000	47.51	0.26	0.08

Table D.2: Text based effects of population size on XSS detection

Population Size	Successful Detection Rate (%)	False Positive Rate (%)	Incorrect Detection Rate (%)
300	9.69	0.00	0.00
600	21.13	0.00	0.00
1200	35.11	0	0.01
2500	49.20	0	0.00
5000	65.15	0	0.35

Table D.3: Text based effects of population size on RFI detection

D.1.2 Generations

Number of Generations	Successful Detection Rate (%)	False Positive Rate (%)	Incorrect Detection Rate (%)
1	76.57	41.13	53.33
50	49.00	3.40	5.67
100	54.66	3.93	5.20
500	47.60	15.00	12.48
1000	52.06	3.80	5.21

Table D.4: Text based effects of the number of generations on SQLi detection

Number of Generations	Successful Detection Rate (%)	False Positive Rate (%)	Incorrect Detection Rate (%)
1	72.24	70.06	83.46
50	21.26	0.13	0.03
100	37.08	0.13	0.01
500	0.11	14.86	8.25
1000	17.80	18.26	25.03

Table D.5: Text based effects of the number of generations on RFI detection

Number of Generations	Successful Detection Rate (%)	False Positive Rate (%)	Incorrect Detection Rate (%)
1	42.24	0.00	0.01
50	31.13	0.00	0.01
100	39.04	18.40	32.38
500	42.95	18.40	32.54
1000	33.44	0.00	0.00

Table D.6: Text based effects of the number of generations on RFI detection

D.1.3 Mutation Rate

Mutation Rate (%)	Successful Detection Rate (%)	False Positive Rate (%)	Incorrect Detection Rate (%)
0.00	32.53	32.33	41.36
0.10	52.37	3.13	5.93
0.25	60.84	14.73	15.67
0.50	48.46	3.66	5.07
1.00	54.46	3.80	6.12

Table D.7: Text based effects of mutation rate on SQLi detection

Mutation Rate (%)	Successful Detection Rate (%)	False Positive Rate (%)	Incorrect Detection Rate (%)
0.00	25.64	15.06	8.31
0.10	64.31	0.33	0.06
0.25	51.02	15.06	8.32
0.50	24.17	0.00	0.01
1.00	5.75	0.13	0.03

Table D.8: Text based effects of mutation rate on XSS detection

Mutation Rate (%)	Successful Detection Rate (%)	False Positive Rate (%)	Incorrect Detection Rate (%)
0.00	23.66	0.00	0.04
0.10	35.75	0.00	0.02
0.25	32.37	0.00	0.02
0.50	32.28	0.00	0.15
1.00	32.71	0.00	0.45

Table D.9: Text based effects of mutation rate on RFI detection

D.1.4 Elitist Pool

Elitist Pool Size (%)	Successful Detection Rate (%)	False Positive Rate (%)	Incorrect Detection Rate (%)
0	8.40	0.73	0.14
2	20.33	1.26	6.54
5	61.31	6.40	12.62
10	83.53	22.33	33.14
25	91.37	9.06	14.52

Table D.10: Text based effects of the size of elitist pool on SQLi detection

Elitist Pool Size (%)	Successful Detection Rate (%)	False Positive Rate (%)	Incorrect Detection Rate (%)
0	2.68	14.93	8.27
2	0.57	0.06	0.01
5	28.48	14.933	8.25
10	70.22	0.26	0.03
25	87.08	18.53	25.07

Table D.11: Text based effects of the size of elitist pool on XSS detection

Elitist Pool Size (%)	Successful Detection Rate (%)	False Positive Rate (%)	Incorrect Detection Rate (%)
0	1.28	0.00	0.00
2	11.53	0.00	0.01
5	35.46	18.40	0.03
10	56.91	18.40	0.00
25	72.35	0.00	0.00

Table D.12: Text based effects of the size of elitist pool on RFI detection

D.1.5 Combining Multiple Signature Sets

Number of Iterations	Successful Detection Rate (%)	False Positive Rate (%)	Incorrect Detection Rate (%)
1	97.40	20.40	33.68
2	98.42	22.93	34.96
5	99.40	41.46	54.66
10	99.60	68.73	82.91
20	99.60	69.26	83.27

Table D.13: Text based effects of multiple iterations on SQLi detection

Number of Iterations	Successful Detection Rate (%)	False Positive Rate (%)	Incorrect Detection Rate (%)
1	97.04	70.13	83.46
2	98.06	85.06	91.73
5	98.64	70.20	83.47
10	98.55	70.26	83.48
20	99.75	100.00	100.00

Table D.14: Text based effects of multiple iterations on XSS detection

Number of Iterations	Successful Detection Rate (%)	False Positive Rate (%)	Incorrect Detection Rate (%)
1	74.97	0.00	0.00
2	87.53	0.00	0.15
5	98.24	0.00	0.51
10	99.24	0.00	0.51
20	99.95	0.00	1.25

Table D.15: Text based effects of multiple iterations on RFI detection

D.1.6 Bitstring Segment Length Effects

Segment Lengths	Successful Detection Rate (%)	False Positive Rate (%)	Incorrect Detection Rate (%)
1113	95.24	99.80	99.86
1123	75.64	80.00	70.40
1133	66.88	81.20	87.06
1143	54.57	34.73	23.10
1153	33.51	25.80	25.53
1163	99.51	80.93	78.54
2113	81.00	70.06	68.92
2123	98.82	96.73	99.12
2133	80.57	44.00	39.75

2143	80.55	38.20	28.25
2153	79.55	22.93	20.10
2163	89.97	39.00	51.08
3113	77.46	94.40	97.54
3123	94.04	69.13	70.73
3133	85.35	49.40	62.24
3143	73.95	19.40	20.95
3153	86.62	21.60	30.74
3163	79.64	5.60	11.03
4113	63.86	40.66	39.78
4123	66.93	33.33	38.35
4133	71.73	62.73	74.80
4143	73.04	51.40	72.97
4153	64.37	6.66	9.05
4163	70.37	21.13	33.32
5113	75.93	39.20	51.17
5123	68.84	33.66	37.80
5133	71.88	35.33	51.96
5143	58.15	3.73	4.96
5153	47.00	15.93	9.87
5163	72.35	18.00	20.37
6113	68.60	6.53	7.85
6123	64.88	6.86	7.77
6133	67.26	20.66	28.92
6143	55.60	16.46	13.16
6153	54.17	16.60	17.13

6163	60.55	17.66	28.42

Table D.16: Text based effects of bitstring segment lengths on SQLi detection

Segment Lengths	**Successful Detection Rate (%)**	**False Positive Rate (%)**	**Incorrect Detection Rate (%)**
1113	100.00	99.93	99.98
1123	86.80	85.06	91.74
1133	82.20	81.60	74.93
1143	71.08	55.33	75.22
1153	69.73	99.93	99.96
1163	83.33	33.60	33.38
2113	87.44	99.86	99.97
2123	73.42	48.60	41.67
2133	60.00	33.60	33.37
2143	87.33	37.06	50.17
2153	71.62	18.53	25.10
2163	72.71	33.53	33.37
3113	92.64	70.20	83.46
3123	68.26	51.73	58.36
3133	67.31	48.46	41.64
3143	76.42	18.66	25.14
3153	70.31	15.13	8.32
3163	59.55	36.66	50.08
4113	82.46	63.26	49.88
4123	67.62	30.00	16.57

4133	71.15	48.13	41.60
4143	40.24	33.33	33.37
4153	61.62	0.20	0.03
4163	46.77	15.20	8.31
5113	83.20	15.13	8.31
5123	65.46	51.66	58.34
5133	50.11	18.40	25.07
5143	61.48	48.26	41.60
5153	50.62	0.20	0.04
5163	45.00	15.06	8.30
6113	73.24	51.66	58.37
6123	54.20	36.66	50.11
6133	52.57	15.13	8.31
6143	27.66	0.26	0.04
6153	44.42	0.33	0.07
6163	39.64	33.33	33.33

Table D.17: Text based effects of bitstring segment lengths on XSS detection

Segment Lengths	**Successful Detection Rate (%)**	**False Positive Rate (%)**	**Incorrect Detection Rate (%)**
1113	100.00	100.00	99.64
1123	100.00	81.60	67.58
1133	76.06	70.13	99.66
1143	78.97	14.93	1.94
1153	71.51	33.33	34.16

1163	70.80	0.00	0.68
2113	85.33	66.66	67.06
2123	95.04	66.66	67.12
2133	92.28	33.33	33.65
2143	81.24	14.93	0.93
2153	73.24	0.00	0.47
2163	68.17	0.00	0.45
3113	75.55	85.06	99.21
3123	89.13	81.60	66.87
3133	67.73	0.00	0.16
3143	60.15	0.00	0.11
3153	56.75	0.00	0.51
3163	52.68	0.00	0.01
4113	82.80	66.66	66.44
4123	78.22	0.00	1.41
4133	58.77	18.40	32.58
4143	46.53	0.00	0.62
4153	47.08	0.00	0.02
4163	46.44	0.00	0.92
5113	73.37	33.33	33.20
5123	72.80	0.00	0.70
5133	59.60	18.40	33.56
5143	46.84	14.93	0.24
5153	42.46	0.00	0.15
5163	42.04	0.00	0.00
6113	74.20	81.60	65.88

6123	64.73	14.93	0.62
6133	53.33	0.00	0.16
6143	42.55	14.93	0.08
6153	35.88	0.00	0.15
6163	38.42	0.00	0.04

Table D.18: Text based effects of bitstring segment lengths on RFI detection

D.1.7 Compared With Random Permutations with Fitness

Amount of Permutated Signatures Used (%))	**Successful Detection Rate (%)**	**False Positive Rate (%)**	**Incorrect Detection Rate (%)**
20	59.26	3.80	13.26
40	76.93	3.80	13.26
60	87.80	4.00	13.26
80	94.33	5.40	14.26
100	95.19	95.39	99.76

Table D.19: Text based effects of permutated bitstrings for SQLi detection

Amount of Permutated Signatures Used (%))	**Successful Detection Rate (%)**	**False Positive Rate (%)**	**Incorrect Detection Rate (%)**
20	70.13	0.20	0.03
40	84.46	0.40	0.10
60	91.20	0.40	0.10
80	94.53	0.40	0.10
100	98.73	99.8	99.96

Table D.20: Text based effects of permutated bitstrings for XSS detection

Amount of Permutated Signatures Used (%))	**Successful Detection Rate (%)**	**False Positive Rate (%)**	**Incorrect Detection Rate (%)**
20	17.66	0.00	0.00
40	35.33	0.00	0.00
60	53.33	0.00	0.00
80	71.00	0.00	0.00
100	83.13	100.00	99.86

Table D.21: Text based effects of permutated bitstrings for RFI detection

D.2 Remaining Graphical Results

D.2.1 Population Size

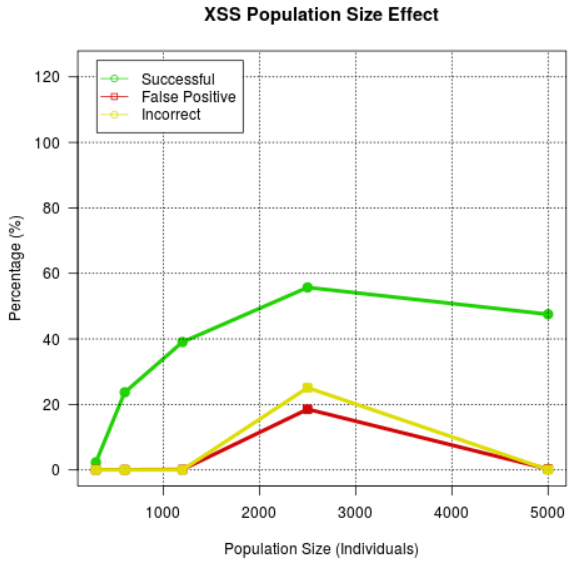

Figure D.1: Effects of Population Size on Detecting XSS

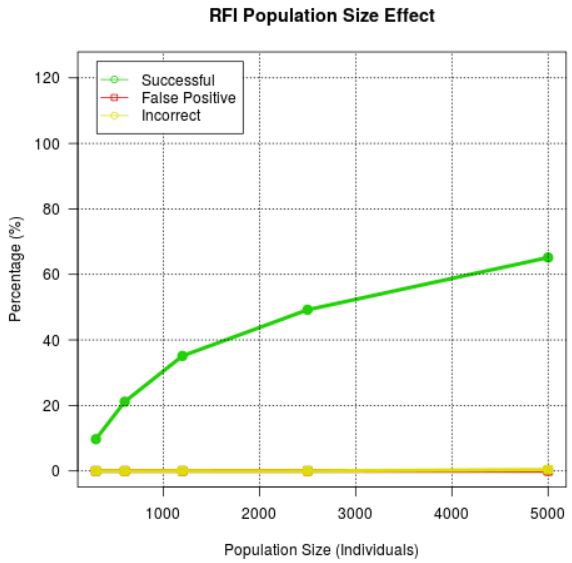

Figure D.2: Effects of Population Size on Detecting RFI

D.2.2 Generations

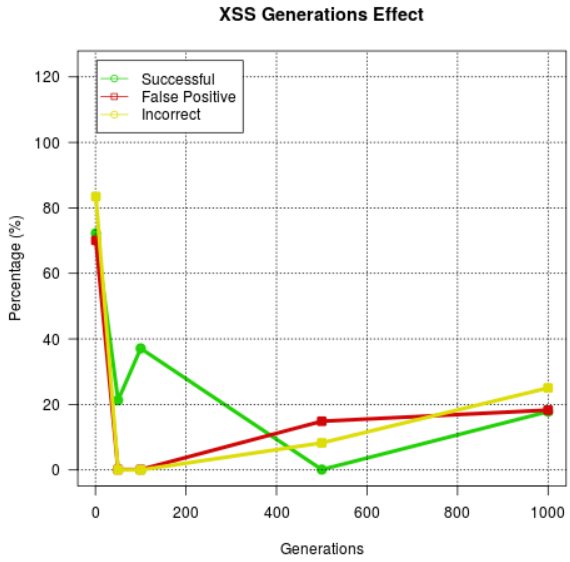

Figure D.3: Effects of Number of Generations on Detecting XSS

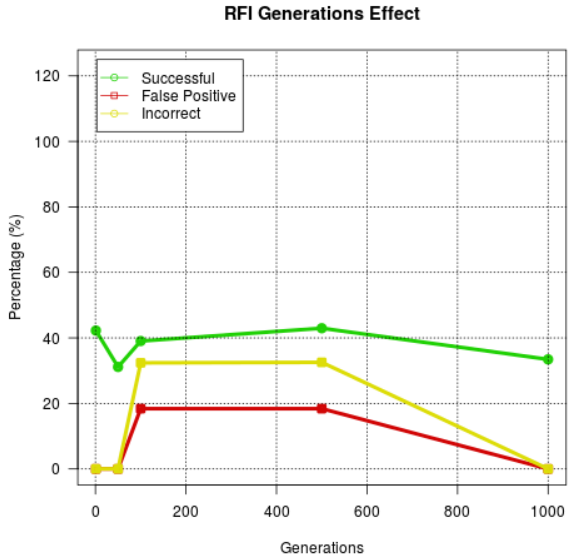

Figure D.4: Effects of Number of Generations on Detecting RFI

D.2.3 Mutation Rate

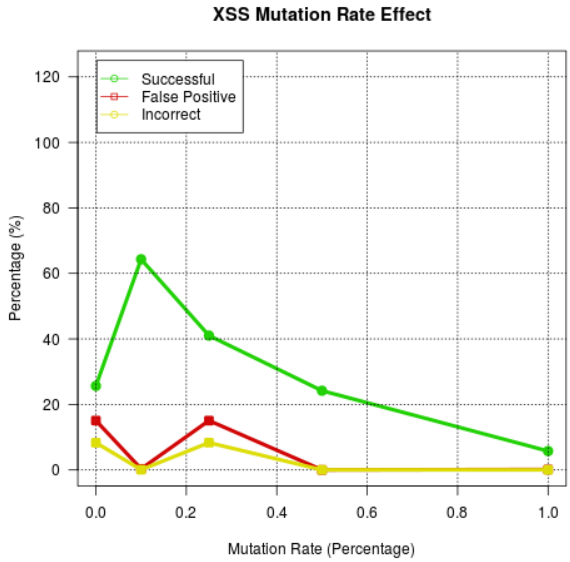

Figure D.5: Effects of Mutation Rate on Detecting XSS

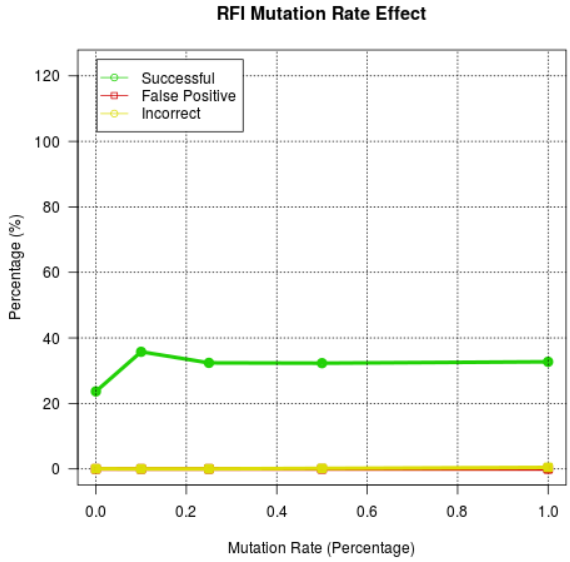

Figure D.6: Effects of Mutation Rate on Detecting RFI

D.2.4 Elitist Pool

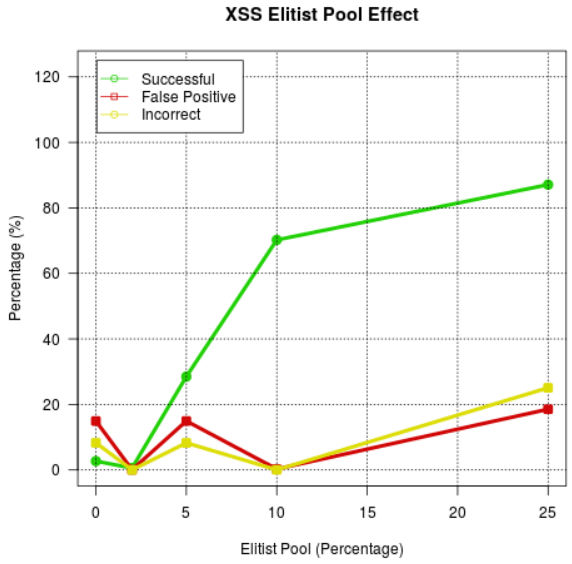

Figure D.7: Effects of Elitist Pool on Detecting XSS

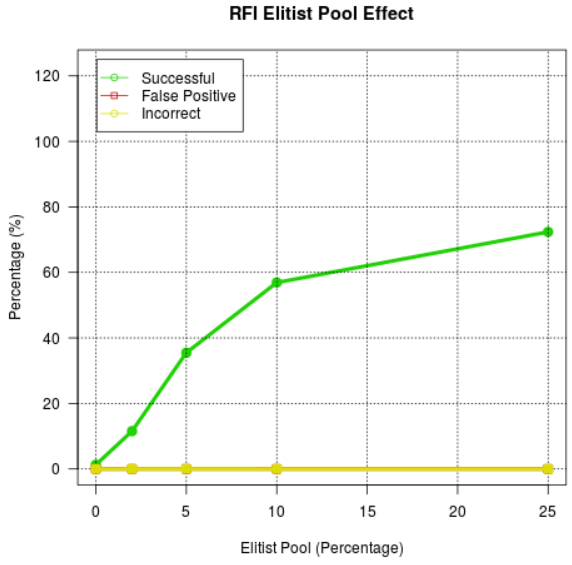

Figure D.8: Effects of Elitist Pool on Detecting RFI

D.2.5 Combining Multiple Signature Sets

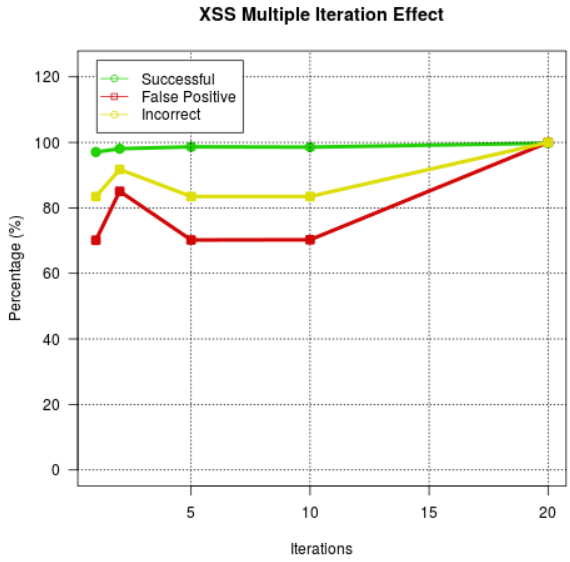

Figure D.9: Effects of Multiple Iterations on Detecting XSS

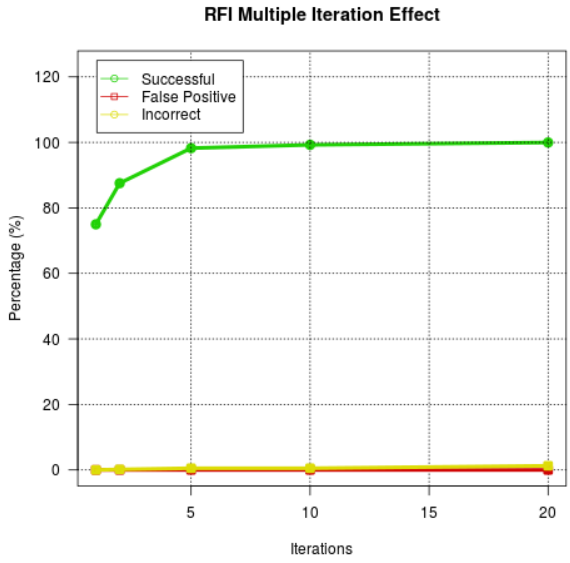

Figure D.10: Effects of Multiple Iterations on Detecting RFI

D.2.6 Bitstring Segment Length Effects

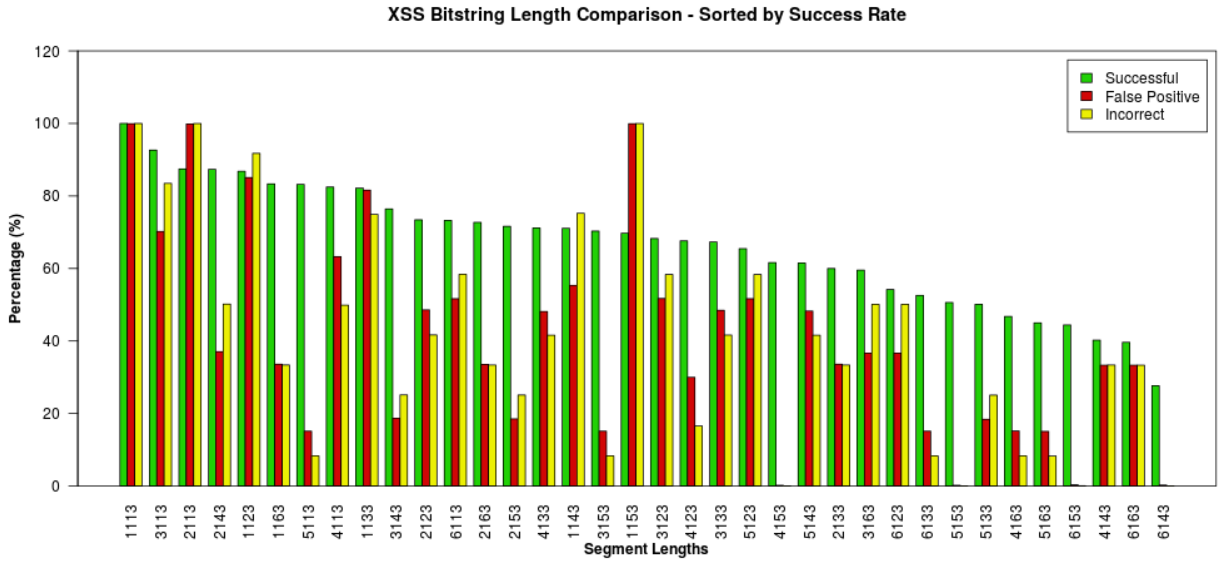

Figure D.11: Effects of Segment Lengths on Detecting XSS

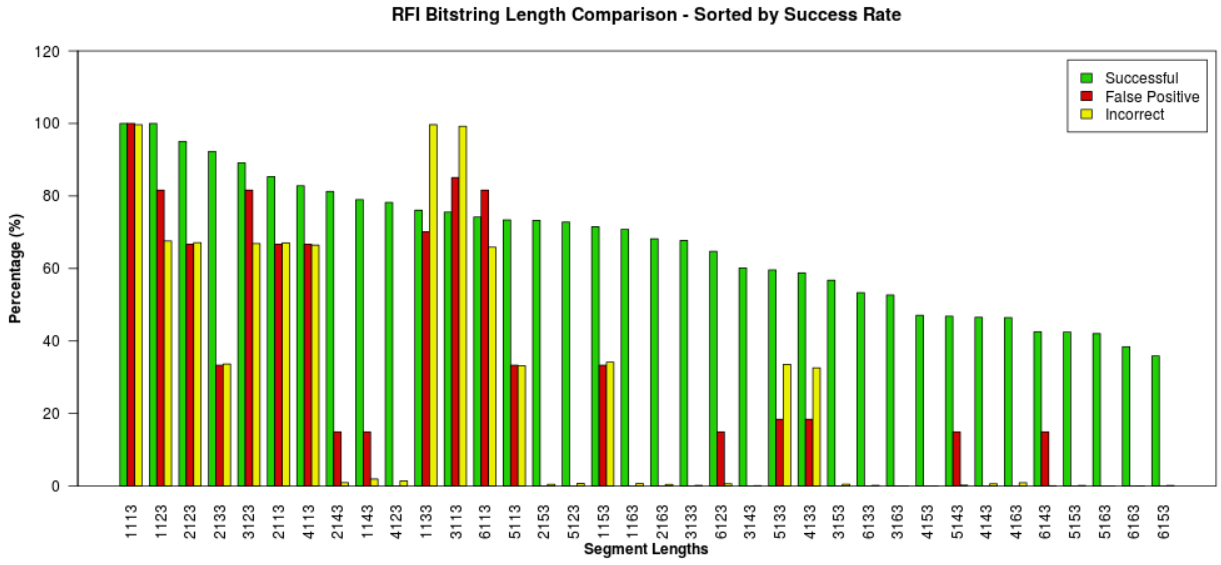

Figure D.12: Effects of Segment Lengths on Detecting RFI

D.2.7 Compared With Random Permutations with Fitness

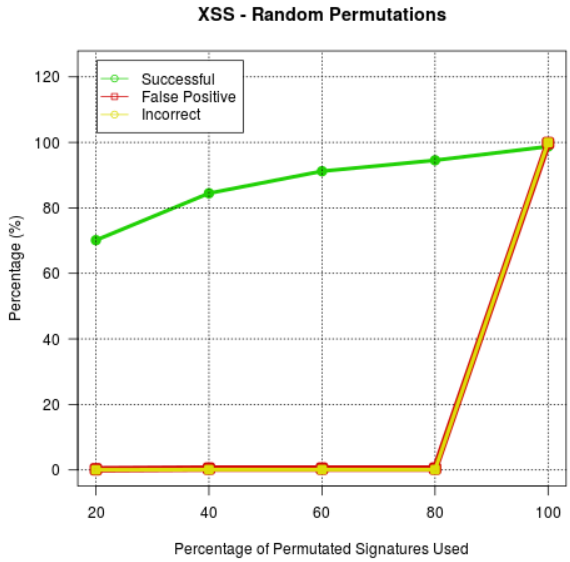

Figure D.13: Random Permutations to Detecting XSS

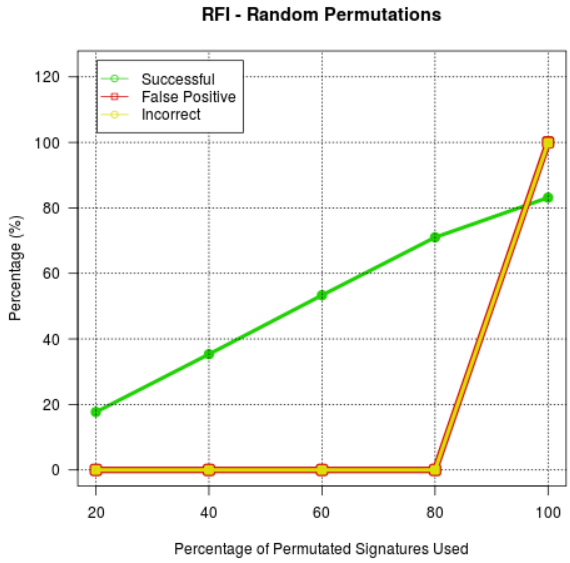

Figure D.14: Random Permutations to Detecting RFI

Appendix E

Full Support Vector Machine Testing Results

E.1 Full Text-Based Results

E.1.1 Comparison with Genetic Algorithm

Training Info & Kernel Info	**Successful Detection Rate (%)**	**False Positive Rate (%)**	**Incorrect Detection Rate (%)**
1000_334L	90.26	11.20	5.00
1000_334P	47.46	1.80	0.66
1000_334R	90.06	5.00	4.53
150_50L	90.26	11.20	5.50
150_50P	47.60	2.00	0.86
150_50R	90.00	5.00	4.66
300_100L	90.26	11.20	5.00
300_100P	54.53	2.20	2.06

300_100R	90.00	5.00	4.66
420_140L	90.26	11.20	5.00
420_140P	79.13	2.60	2.53
420_140R	90.00	5.00	4.66
562_189L	90.26	11.20	5.00
562_189P	54.80	2.20	2.06
562_189R	90.00	5.00	4.66
750_250L	90.26	11.20	5.00
750_250P	47.13	1.80	0.66
750_250R	90.00	5.00	4.66
75_25L	90.26	11.20	5.00
75_25P	79.00	8.80	2.33
75_25R	90.33	11.20	6.36

Table E.1: Text based effects of genetic algorithm and SVM comparison on SQLi detection

Training Info & Kernel Info	**Successful Detection Rate (%)**	**False Positive Rate (%)**	**Incorrect Detection Rate (%)**
1000_334L	90.86	0.20	0.03
1000_334P	90.86	0.20	0.03
1000_334R	90.86	0.20	0.03
150_50L	90.86	0.20	0.03
150_50P	90.86	0.20	0.03
150_50R	90.86	0.20	0.03
300_100L	90.86	0.20	0.03

300_100P	90.86	0.20	0.03
300_100R	90.86	0.20	0.03
420_140L	90.86	0.20	0.03
420_140P	90.86	0.20	0.03
420_140R	90.86	0.20	0.03
562_189L	90.86	0.20	0.03
562_189P	90.86	0.20	0.03
562_189R	90.86	0.20	0.03
750_250L	90.86	0.20	0.03
750_250P	90.86	0.20	0.03
750_250R	90.86	0.20	0.03
75_25L	90.86	0.20	0.03
75_25P	90.86	0.20	0.03
75_25R	90.86	0.20	0.03

Table E.2: Text based effects of genetic algorithm and SVM comparison on XSS detection

Training Info & Kernel Info	**Successful Detection Rate (%)**	**False Positive Rate (%)**	**Incorrect Detection Rate (%)**
1000_334L	99.20	0.00	0.16
1000_334P	99.20	0.00	0.00
1000_334R	100.00	0.00	0.03
150_50L	100.00	0.00	0.16
150_50P	99.20	0.00	0.06
150_50R	100.00	0.00	0.06

300_100L	99.20	0.00	0.16
300_100P	99.20	0.00	0.00
300_100R	100.00	0.00	0.00
420_140L	99.20	0.00	0.16
420_140P	99.20	0.00	0.00
420_140R	100.00	0.00	0.06
562_189L	100.00	0.00	0.53
562_189P	99.20	0.00	0.00
562_189R	100.00	0.00	0.06
750_250L	100.00	0.00	0.56
750_250P	99.20	0.00	0.00
750_250R	100.00	0.00	0.03
75_25L	99.20	0.00	0.16
75_25P	97.66	0.00	0.03
75_25R	99.20	0.00	0.26

Table E.3: Text based effects of genetic algorithm and SVM comparison on RFI detection

E.1.2 Increasing Non-Threats

Training Info & Kernel Info	**Successful Detection Rate (%)**	**False Positive Rate (%)**	**Incorrect Detection Rate (%)**
100L	90.26	11.20	5.00
100P	54.53	2.20	2.06
100R	90.00	5.00	4.66
200L	90.33	7.20	6.36
200P	47.33	2.00	0.86
200R	90.33	11.20	6.36
350L	90.33	5.20	6.36
350P	47.60	2.20	0.86
350R	90.33	7.20	6.26
500L	99.13	6.60	15.90
500P	47.60	2.20	0.86
500R	90.13	5.20	5.60
50L	90.26	11.20	5.00
50P	54.53	2.20	2.06
50R	90.00	5.00	4.70
750L	79.26	3.00	3.90
750P	47.60	2.20	0.86
750R	90.00	5.00	4.66

Table E.4: Text based effects of increasing non-threats in training data on SQLi detection

Training Info & Kernel Info	**Successful Detection Rate (%)**	**False Positive Rate (%)**	**Incorrect Detection Rate (%)**
100L	90.86	0.20	0.03
100P	90.86	0.20	0.03
100R	90.86	0.20	0.03
200L	90.86	0.20	0.03
200P	90.86	0.20	0.03
200R	90.86	0.20	0.03
350L	90.86	0.20	0.03
350P	90.86	0.20	0.03
350R	90.86	0.20	0.03
500L	90.86	0.20	0.03
500P	90.86	0.20	0.03
500R	90.86	0.20	0.03
50L	90.86	0.20	0.03
50P	90.86	0.20	0.03
50R	90.86	0.20	0.03
750L	90.86	0.20	0.03
750P	90.86	0.20	0.03
750R	90.86	0.20	0.03

Table E.5: Text based effects of increasing non-threats in training data on XSS detection

Training Info & Kernel Info	**Successful Detection Rate (%)**	**False Positive Rate (%)**	**Incorrect Detection Rate (%)**
100L	99.20	0.00	0.16
100P	99.20	0.00	0.00
100R	100.00	0.00	0.00
200L	99.20	0.00	0.16
200P	97.66	0.00	0.00
200R	99.20	0.00	0.26
350L	99.20	0.00	0.16
350P	99.20	0.00	0.00
350R	99.20	0.00	0.26
500L	99.20	0.00	0.16
500P	99.20	0.00	0.00
500R	99.20	0.00	0.16
50L	99.20	0.00	0.16
50P	99.20	0.00	0.00
50R	100.00	0.00	0.00
750L	99.20	0.00	0.16
750P	99.20	0.00	0.00
750R	99.20	0.00	0.10

Table E.6: Text based effects of increasing non-threats in training data on RFI detection

E.1.3 Increasing Incorrect-Threats

Training Info & Kernel Info	**Successful Detection Rate (%)**	**False Positive Rate (%)**	**Incorrect Detection Rate (%)**
100L	99.13	6.60	15.90
100P	90.26	4.60	6.36
100R	99.13	7.00	15.90
200L	99.13	6.60	15.90
200P	54.86	2.20	2.06
200R	90.33	7.20	6.30
275L	99.13	7.00	15.90
275P	47.60	2.20	0.86
275R	90.33	7.20	6.26
350L	79.26	5.20	3.90
350P	47.33	2.00	0.86
350R	90.33	7.20	6.26
50L	99.13	6.60	15.90
50P	99.13	7.00	15.90
50R	99.13	7.00	15.90
600L	79.20	5.00	2.53
600P	40.13	1.40	0.66
600R	90.00	7.20	4.70
750L	54.86	4.40	2.06
750P	31.40	1.80	0.23
750R	90.00	5.00	4.66

Table E.7: Text based effects of increasing incorrect-threats in training data on SQLi detection

Training Info & Kernel Info	**Successful Detection Rate (%)**	**False Positive Rate (%)**	**Incorrect Detection Rate (%)**
100L	90.86	0.20	0.03
100P	90.86	0.20	0.03
100R	90.86	0.20	0.03
200L	90.86	0.20	0.03
200P	90.86	0.20	0.03
200R	90.86	0.20	0.03
275L	90.86	0.20	0.03
275P	56.93	0.20	0.03
275R	90.86	0.20	0.03
350L	90.86	0.20	0.03
350P	90.86	0.20	0.03
350R	90.86	0.20	0.03
50L	90.86	0.20	0.03
50P	56.93	0.20	0.03
50R	90.86	0.20	0.03
600L	90.86	0.20	0.03
600P	90.86	0.20	0.03
600R	90.86	0.20	0.03
750L	90.86	0.20	0.03
750P	90.86	0.20	0.03

750R	90.86	0.20	0.03

Table E.8: Text based effects of increasing incorrect-threats in training data on XSS detection

Training Info & Kernel Info	**Successful Detection Rate (%)**	**False Positive Rate (%)**	**Incorrect Detection Rate (%)**
100L	100.00	0.00	0.73
100P	99.20	0.00	0.06
100R	100.00	0.00	0.26
200L	99.20	0.00	0.16
200P	99.20	0.00	0.06
200R	100.00	0.00	0.56
275L	99.20	0.00	0.16
275P	97.66	0.00	0.03
275R	99.20	0.00	0.16
350L	99.20	0.00	0.16
350P	99.20	0.00	0.00
350R	99.20	0.00	0.16
50L	100.00	0.00	0.10
50P	99.20	0.00	0.06
50R	100.00	0.00	0.10
600L	99.20	0.00	0.10
600P	99.20	0.00	0.00
600R	99.20	0.00	0.16
750L	99.20	0.00	0.10

750P	99.20	0.00	0.00
750R	99.20	0.00	0.16

Table E.9: Text based effects of increasing incorrect-threats in training data on RFI detection

E.2 Remaining Graphical Results

E.2.1 Comparison with Genetic Algorithm

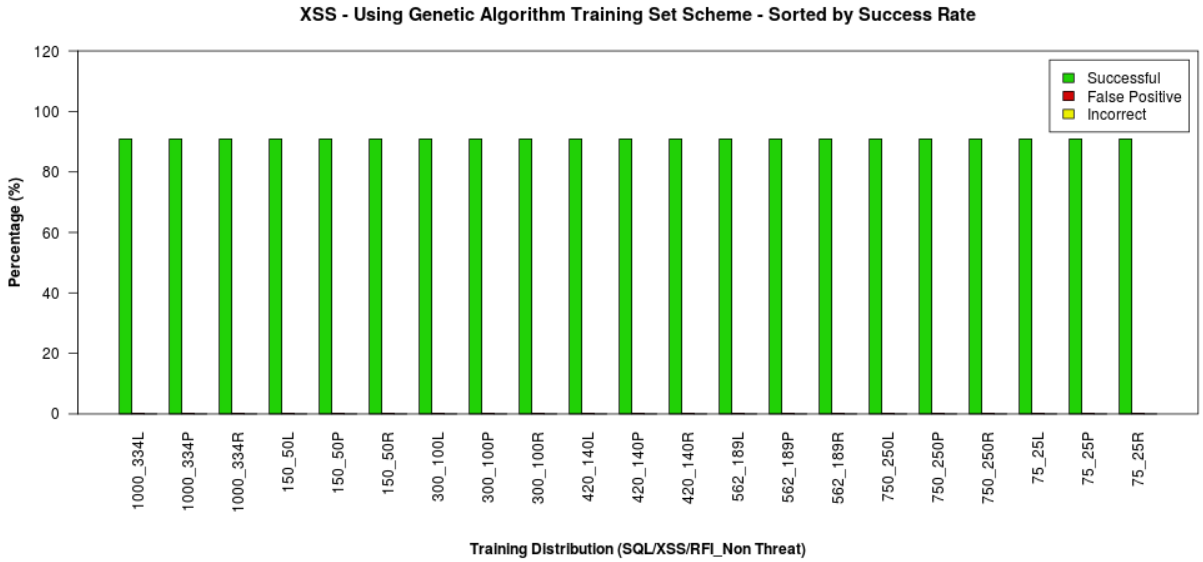

Figure E.1: Genetic Algorithm and SVM comparison for Detecting XSS

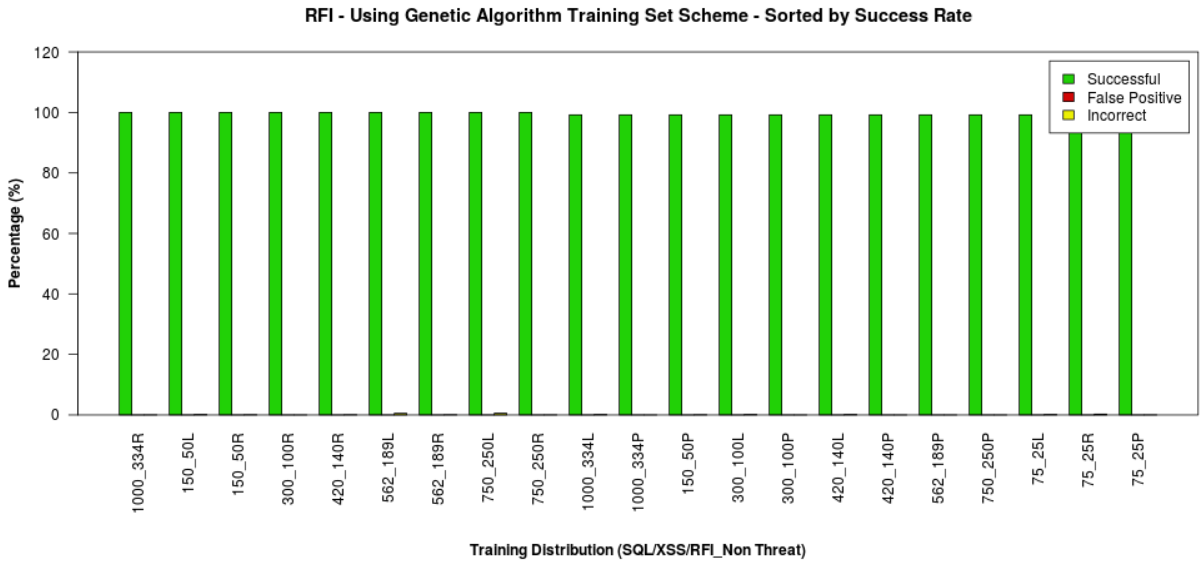

Figure E.2: Genetic Algorithm and SVM comparison for Detecting RFI

E.2.2 Increasing Non-Threats

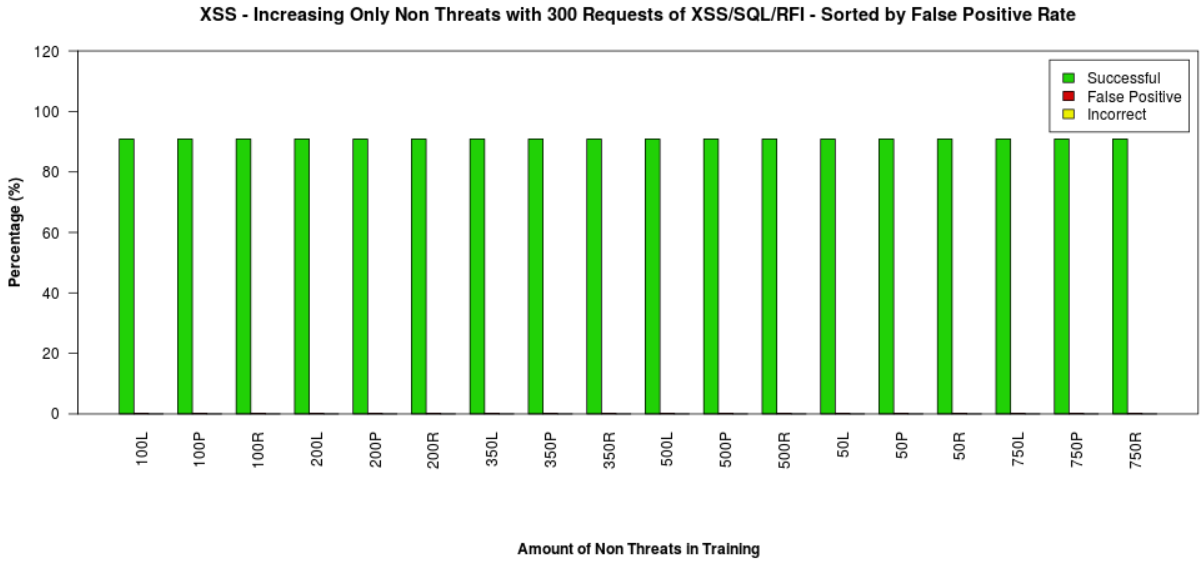

Figure E.3: Effects of Increasing Non-Threat training data in Detecting XSS

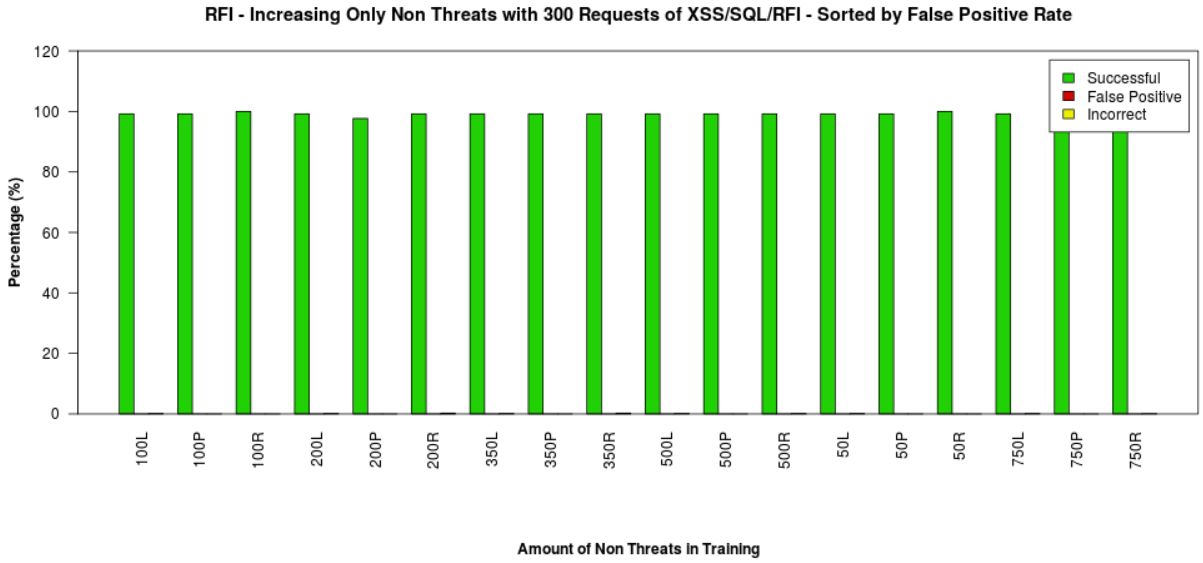

Figure E.4: Effects of Increasing Non-Threat training data in Detecting RFI

E.2.3 Increasing Incorrect-Threats

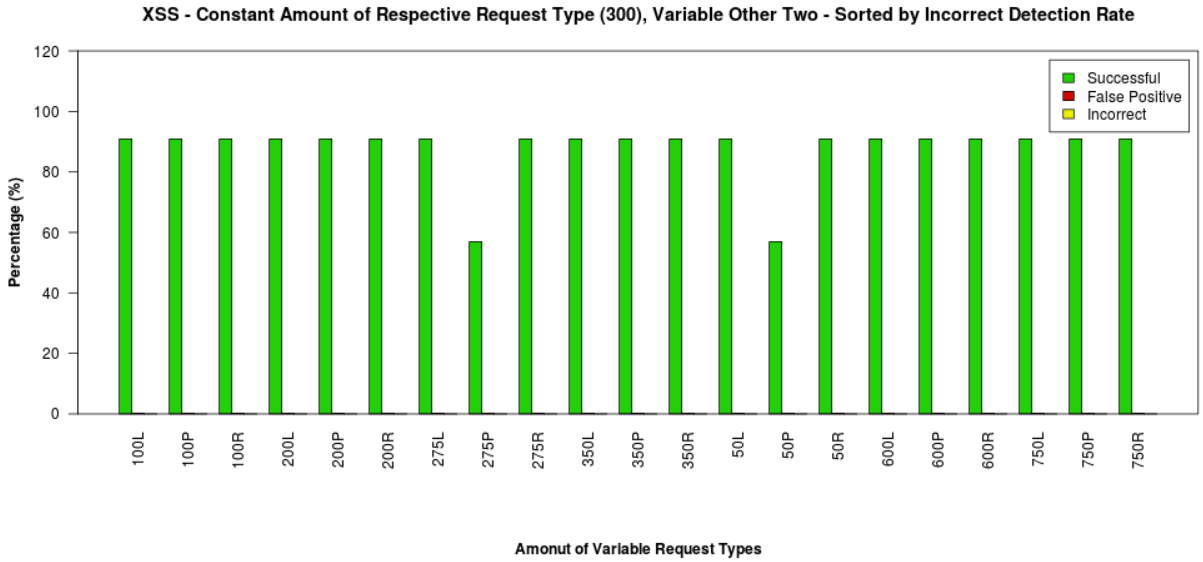

Figure E.5: Effects of Increasing incorrect attack type training data in Detecting XSS

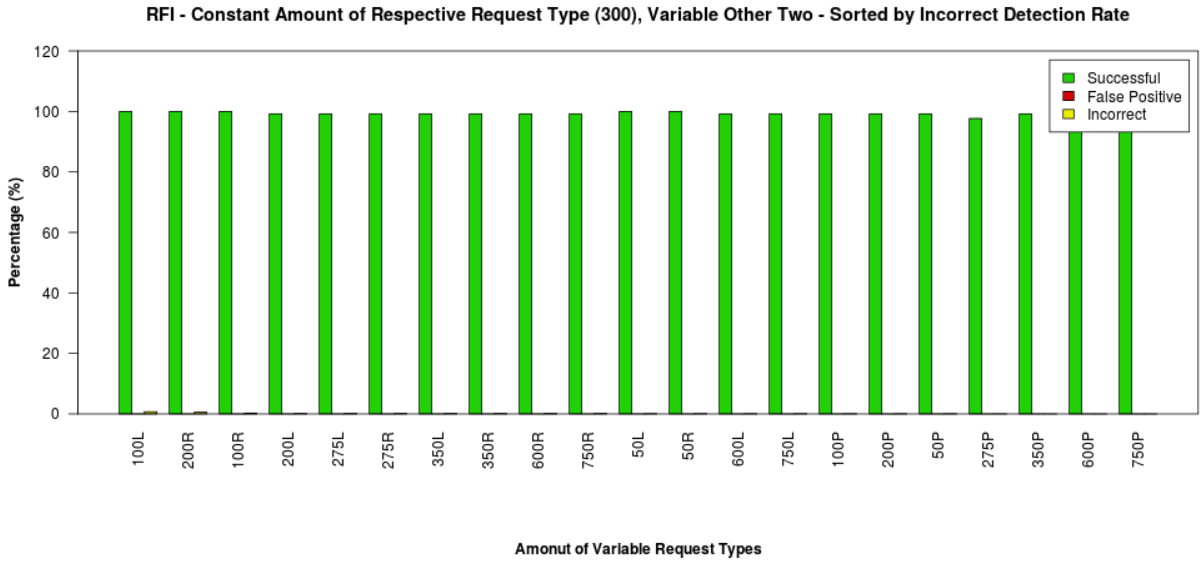

Figure E.6: Effects of Increasing incorrect attack type training data in Detecting RFI

Bibliography

[1] M. R. Hassan, "Intrusion Detection System Based on Cost Based Support Vector Machine," in *Recent Advances in Information and Communication Technology 2016*, P. Meesad, S. Boonkrong, and H. Unger, Eds. Cham: Springer International Publishing, 2016, vol. 463, pp. 105–115. [Online]. Available: http://link.springer.com/10.1007/978-3-319-40415-8_11

[2] L. MacVittie and D. Holmes, "The New Data Center Firewall Paradigm," *F5 Networks, Inc., Seattle*, 2012.

[3] S. B. Kotsiantis, "Supervised machine learning: A review of classification techniques," in *Proceedings of the 2007 Conference on Emerging Artificial Intelligence Applications in Computer Engineering: Real Word AI Systems with Applications in eHealth, HCI, Information Retrieval and Pervasive Technologies*. Amsterdam, The Netherlands, The Netherlands: IOS Press, 2007, pp. 3–24. [Online]. Available: http://dl.acm.org/citation.cfm?id=1566770.1566773

[4] M. Sharma, G. Pragati, and O. Nagamani, "Challenges and Countermeasures for Web Applications." *International Journal of Advanced Research in Computer Science*, vol. 2, no. 3, 2011. [Online]. Available: http://search.ebscohost.com/login.aspx?direct=true&profile=ehost&scope=site&authtype=crawler&jrnl=09765697&AN=66863080&h=VZkqiPeGIMQw4pWWsRLai4mHhcy2a3wr4%

2F%2Bu9YWPaebVhg4XRoDApRzOW8KdH%2FdFJHdcNVU%
2BWEcxWm0jdfzsUg%3D%3D&crl=c

[5] R. Johari and P. Sharma, "A Survey on Web Application Vulnerabilities (SQLIA, XSS) Exploitation and Security Engine for SQL Injection." IEEE, May 2012, pp. 453–458. [Online]. Available: http://ieeexplore.ieee.org/document/6200667/

[6] W. G. Halfond, J. Viegas, and A. Orso, "A classification of SQL-injection attacks and countermeasures," in *Proceedings of the IEEE International Symposium on Secure Software Engineering*, vol. 1. IEEE, 2006, pp. 13–15. [Online]. Available: http://www.cc.gatech.edu/fac/Alex.Orso/papers/halfond.viegas.orso.ISSSE06.pdf

[7] M. K. Gupta, M. C. Govil, G. Singh, and P. Sharma, "XSSDM: Towards detection and mitigation of cross-site scripting vulnerabilities in web applications." IEEE, Aug. 2015, pp. 2010–2015. [Online]. Available: http://ieeexplore.ieee.org/document/7275912/

[8] "Cross-site Scripting (XSS) - OWASP," Feb. 2017. [Online]. Available: https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)

[9] "DOM Based XSS - OWASP," Feb. 2017. [Online]. Available: https://www.owasp.org/index.php/DOM_Based_XSS

[10] "Testing for Remote File Inclusion - OWASP," Feb. 2017. [Online]. Available: https://www.owasp.org/index.php/Testing_for_Remote_File_Inclusion

[11] W. G. Halfond and A. Orso, "Detection and prevention of sql injection attacks," in *Malware Detection*. Springer, 2007, pp. 85–109. [Online]. Available: http://link.springer.com/chapter/10.1007/978-0-387-44599-1_5

[12] "Usage Statistics and Market Share of PHP for Websites, March 2017." [Online]. Available: https://w3techs.com/technologies/details/pl-php/all/all

[13] "PHP: Supported Versions." [Online]. Available: http://php.net/supported-versions.php

[14] "About Requirements WordPress." [Online]. Available: https://wordpress.org/about/requirements/

[15] "SQL Injection Prevention Cheat Sheet - OWASP." [Online]. Available: https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet#Least_Privilege

[16] "HTML Purifier - Filter your HTML the standards-compliant way!" [Online]. Available: http://htmlpurifier.org/

[17] R. Bronte, H. Shahriar, and H. M. Haddad, "A Signature-Based Intrusion Detection System for Web Applications based on Genetic Algorithm." ACM Press, 2016, pp. 32–39. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2947626.2951964

[18] F. Massicotte and Y. Labiche, "On the Verification and Validation of Signature-Based, Network Intrusion Detection Systems." IEEE, Nov. 2012, pp. 61–70. [Online]. Available: http://ieeexplore.ieee.org/document/6405405/

[19] A. Lakhotia, A. Kapoor, and E. U. Kumar, "Are metamorphic viruses really invincible," *Virus Bulletin*, vol. 12, p. 57, 2004. [Online]. Available: http://www.iscas2007.org/~arun/papers/invincible-complete.pdf

[20] D. Lin and M. Stamp, "Hunting for undetectable metamorphic viruses," *Journal in Computer Virology*, vol. 7, no. 3, pp. 201–214, Aug. 2011. [Online]. Available: http://link.springer.com/10.1007/s11416-010-0148-y

[21] I. Witten, *Data mining: practical machine learning tools and techniques*. Boston, MA: Elsevier, 2016.

[22] M. Harman, S. A. Mansouri, and Y. Zhang, "Search-based software engineering: Trends, techniques and applications," *ACM Computing Surveys*, vol. 45, no. 1,

pp. 1–61, Nov. 2012. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2379776.2379787

[23] "How the Genetic Algorithm Works - MATLAB & Simulink." [Online]. Available: http://www.mathworks.com/help/gads/how-the-genetic-algorithm-works.html

[24] G. Vigna, W. Robertson, and D. Balzarotti, "Testing network-based intrusion detection signatures using mutant exploits." ACM Press, 2004, p. 21. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1030083.1030088

[25] "RBF SVM parameters scikit-learn 0.18.1 documentation." [Online]. Available: http://scikit-learn.org/stable/auto_examples/svm/plot_rbf_parameters.html

[26] S. Mukkamala, G. Janoski, and A. Sung, "Intrusion detection using neural networks and support vector machines," in *Neural Networks, 2002. IJCNN'02. Proceedings of the 2002 International Joint Conference on*, vol. 2. IEEE, 2002, pp. 1702–1707. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1007774

[27] "sqlmap: automatic SQL injection and database takeover tool," Nov. 2016. [Online]. Available: http://sqlmap.org/

[28] "Grabber | Penetration Testing Tools," Nov. 2016. [Online]. Available: http://tools.kali.org/web-applications/grabber

[29] "epsylon/xsser-public: XSSer - Cross Site "Scripter"," Nov. 2016. [Online]. Available: https://github.com/epsylon/xsser-public

[30] "HttpFox." [Online]. Available: https://addons.mozilla.org/en-US/firefox/addon/httpfox/

[31] "fimap | Penetration Testing Tools," Nov. 2016. [Online]. Available: http://tools.kali.org/web-applications/fimap

[32] "PL/SQL Reserved Words and Keywords," Nov. 2016. [Online]. Available: https://docs.oracle.com/cd/B19306_01/appdev.102/b14261/reservewords.htm

[33] "MySQL :: MySQL 5.7 Reference Manual :: 10.3 Keywords and Reserved Words." [Online]. Available: https://dev.mysql.com/doc/refman/5.7/en/keywords.html

[34] "Window Object." [Online]. Available: https://www.w3schools.com/jsref/obj_window.asp

[35] "HTML DOM Document Objects." [Online]. Available: https://www.w3schools.com/jsref/dom_obj_document.asp

[36] "HTML Attributes." [Online]. Available: https://www.w3schools.com/tags/ref_attributes.asp

[37] "PHP: List of Keywords - Manual." [Online]. Available: http://php.net/manual/en/reserved.keywords.php

[38] "scikit-learn: machine learning in Python scikit-learn 0.18.1 documentation," Nov. 2016. [Online]. Available: http://scikit-learn.org/stable/index.html

[39] S. G. B. Rylander, "Optimal population size and the genetic algorithm," *Population*, vol. 100, no. 400, p. 900, 2002. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.417.1552&rep=rep1&type=pdf

[40] J. J. Grefenstette, "Optimization of control parameters for genetic algorithms," *IEEE Transactions on systems, man, and cybernetics*, vol. 16, no. 1, pp. 122–128, 1986. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4075583

[41] N. Stark, G. F. Minetti, and C. Salto, "A new strategy for adapting the mutation probability in genetic algorithms," in *XVIII Congreso Argentino de Ciencias de*

la Computacin, 2012. [Online]. Available: http://sedici.unlp.edu.ar/handle/10915/23593

[42] L. D. Whitley, "The genitor algorithm and selection pressure: Why rank-based allocation of reproductive trials is best," in *Proceedings of the 3rd International Conference on Genetic Algorithms*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1989, pp. 116–123. [Online]. Available: http://dl.acm.org/citation.cfm?id=645512.657257