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

by

Tyler Wilding

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

1 Introduction **1**

1.1 Problem Definition 1

1.2 Objective 2

1.2.1 Expected Results 2

1.3 Thesis Overview 3

1.3.1 Scope and Limitations 3

2 Web Threats **4**

2.1 What is a Web Threat 4

2.2 SQL Injection 6

2.2.1 SQL Injection Types 8

2.3 Cross-Site Scripting 11

2.3.1 Types of Cross-Site Scripting Attacks 11

2.4 Remote File Inclusion 12

3 Current Detection and Prevention Methods **13**

3.1 Prevention through Development 13

3.2 Signature Based Detection 15

3.3 Modern Methods of Detection 16

4 Describing Machine Learning Approaches **18**

4.1 Machine Learning 18

4.1.1 Supervised Learning 19
4.2 Genetic Algorithm 20
4.2.1 Current Genetic Algorithm Solutions 21
4.3 Support Vector Machine 22
4.3.1 Current Support Vector Machine Solutions 25

5 Methods & Procedures **26**
5.1 System Overview 26
5.2 Gathering Test Data 26
5.3 Parsing the Requests 28
5.4 Genetic Algorithm Based Signature Detection 31
5.4.1 Testing Procedure 34
5.5 Support Vector Machine Detection 35
5.5.1 Testing Procedure 37

6 Results **39**
6.1 Genetic Algorithm 39
6.1.1 Finding Best Parameters 39
Population Size 40
Generations 40
Mutation Rate 43
Elitist Pool 43
6.1.2 Combining Multiple Signature Sets 43
6.1.3 Bitstring Segment Length Effects 43
6.2 Compared With Random Permutations with Fitness 47
6.3 Support Vector Machine 47
6.3.1 Comparison with Genetic Algorithm 49
6.3.2 Increasing Non-Threats 50

6.3.3 Increasing Incorrect-Threats 50

7 Discussion **52**

7.1 Parser 52

7.1.1 Weaknesses 52

7.1.2 Strengths 53

7.2 Genetic Algorithm 54

7.2.1 Parameter Testing Results 54

7.2.2 Expanded Signature Set 55

7.2.3 Influence of Segment Length 55

7.2.4 Strengths and Weaknesses 55

7.2.5 Comparison with Random Permutations 57

7.3 Support Vector Machine 58

7.3.1 Comparison with Genetic Algorithm 58

7.3.2 Reducing False Positives 58

7.3.3 Reducing Incorrect Detections 58

7.3.4 Strengths and Weaknesses 58

8 Conclusion **60**

8.1 Conclusions 60

8.2 Future Work 60

Appendices **61**

A Regular Expression Documentation **61**

B Full Genetic Algorithm Testing Results **62**

B.1 Full Text-Based Results 62

B.2 Remaining Graphical Results 62

C Full Support Vector Machine Testing Results **63**

C.1 Full Text-Based Results 63

C.2 Remaining Graphical Results 63

Bibliography **64**

List of Tables

4.1 Kernels that will be used and their mathematical function 24

5.1 Breakdown of test data generation 27
5.2 Parseable Features 28
5.3 Genetic algorithm default segment breakdown 32
5.4 Breakdown of the training file for genetic algorithms 34
5.5 Breakdown of the testing file for both genetic algorithm and support vector machine 34
5.6 Genetic algorithm default segment breakdown 36

6.1 Parameters used in each Genetic Algorithm Test 40
6.2 Parameters used in each Support Vector Machine Test 49

List of Figures

2.1 Possible threats at each OSI model layer and possible mitigation techniques. [1] 5

4.1 Example of a linear seperated SVM 23

5.1 Overview of the system 27
5.2 A sample HTTP request with fields highlighted 29

5.3 Overview of the genetic algorithm system 35
5.4 Overview of the support vector machine system 38

6.1 Effects of Population Size on Detecting SQLi 41
6.2 Effects of Generations on Detecting SQLi 42
6.3 Effects of Mutation Rate on Detecting SQLi 44
6.4 Effects of Elitist Pool on Detecting SQLi 45
6.5 Effects of Multiple Iterations on Detecting SQLi 46
6.6 Effects of Different Segment Lengths on Detecting SQLi 47
6.7 Permutation of Bitstrings to Detect SQLi 48
6.8 Genetic algorithm and SVM comparison for SQLi 49
6.9 Effects of increasing non-threat training data in SVM for SQLi detection 50
6.10 Effects of increasing incorrect attack training data in SVM for SQLi detection 51

List of Algorithms

1 Fitness algorithm for use in genetic algorithm 20

2 General overview of parsing procedure 30

3 Basic pseudocode algorithm for Roulette Wheel Selection in $\mathrm{O}(n)$ 33

4 Pseudocode algorithm for genetic algorithm 33

5 Pseudocode algorithm for support vector machine 37

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 2.1).

However, prevention is not the only task when dealing with these threats as detection is equally as important. Among many reasons, one scenario is an attack knwon as a zero-day attack that has no established prevention method, was just discovered and can be considered to already be causing damage. In this scenario, detection is the first line of defense to inform those in the know to take nessecary 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 that the original attack will not change from its original state and can be redetected using the same signature (Section **??**).

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 **??**). These approaches show great promise, but no system is perfect, and with the research being fairly new the results and technqiues 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 that 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 repsented 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, ??, ??).

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

Simply put, a web threat is any malicious attack that uses the Internet as its main method of distribution, meaning that the types of web threats is wide and varied. Web threats can be broken up into two main cateogires 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 preventation systems due to the unlimited number of

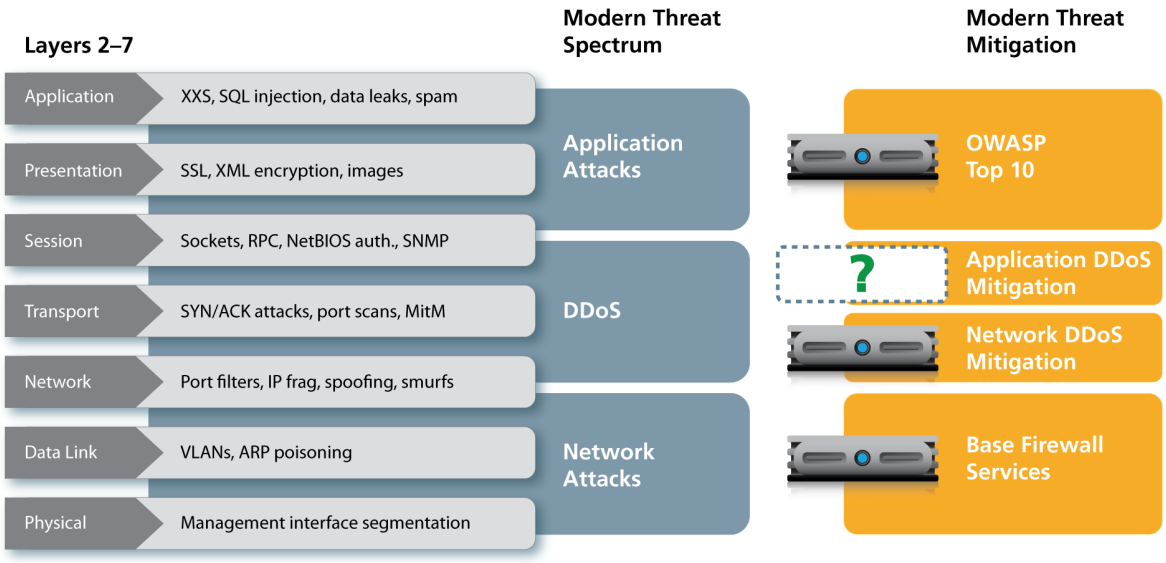

Figure 2.1: Possible threats at each OSI model layer and possible mitigation techniques. [1]

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 that 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 **??**). [2]

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 that many of the existing techniques for handling these application layer attacks suffer from one or more of the following techniques [**?**]:

- 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. [**?**] 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 simpliest 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. [**?**]

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 mimicing as Google queries in order to facilitate malicious drive-by-download malware attacks on websites. [**?**]

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 that 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 are **inference** attacks, these are used where malformed queries cannot be used to provide vital information to construct attacks. Instead, injections are preformed and the website is monitored for changes or a response, this information is used to deduce vulnerable parameters and information on the values 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 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 his search.

```
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 future 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.

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 from user inputs but instead of influencing the database it will leverage the facing code of the web application, more often the HTML or Javascript. XSS attacks can be used for a variety of purposes, to redirecting users to other websites or just simply changing the look of the website. In the worst case, XSS attacks can be used to hijack users 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 what it is being used for. XSS attacks, unlike SQL injections can be just a minor nuisence 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.

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 they're browser runs the attack. A simple example is just inserting typical HTML code into a comment field.

The second type of XSS attack is referred to as a **reflected** attack. Instead of the result being stored in the remote-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 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.

The final type of XSS attack is a newer distinction for XSS attacks called a **Document Object Model (DOM) based** attack. In this variant, the attack is ran through

modifying the actual document model through javascript's document object. What seperates this attack from a stored or reflected attack is the original page does not change, but instead of users actions will result in a different result because their environment has been modified.

2.4 Remote File Inclusion

The final web-threat that will be examined in this research is remote file inclusion also known as RFI. Put simply, it is using the vulnerabilities of the application to include remote files which may include arbitrary code of any language. The most common example is abusing PHP's include() command to get some PHP code to run on the remote server. RFI attacks allow for code execution on the remote server and client-side, denial of service attacks due to remote shells and data extraction.

Whle 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 paramters with PHP commands
- Parameters with URLs and PHP commands

Chapter 3

Current Detection and Prevention Methods

3.1 Prevention through Development

These web attacks can be significantly damaging to an organization in many ways and so detecting and preventing them is of a very high importance. These problems are also not limited to only small time websites as well, with many high profile websites already being victim to attacks, and some studies stating that over 90% of web applications being vulnerable to SQL injections alone.

The best way to stop these attacks is to prevent the vulnerabilities from existing in the first place on the development side. Despite the fact that majority of the attacks are well documented and understood, as safeguards are preventions are put in place to protect one aspect of an application, attacks shift their efforts to look for the next weakest link. From a security standpoint you need to assume that your application is not bulletproof and that you have only mitigated the risk, not removed it completely. As a result, prevention alone is not enough as there is always the possibility that an attack finds a way past the safe guards and so prevention and detection must work hand in

hand.

The simpliest and most common way to prevent these attacks that are at the application layer level of the web application is to never allow user input to be directly concatenated into any command that interacts on 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 like stored XSS attacks or RFIs from occuring where stored code is inserted into the flow of the application code. Many languages have different ways of accomplishing this but the most simpliest way is to simply strip the portions that cause it to be interpreted as code, but a more comprehensive way is to filter the HTML against a whitelist.

There are other measures that can be taken but these mostly pertain to the environment on which the application is ran on and specifically for SQL injection prevention. Seperate database users should be made for each application and they should have the least amount of privledge possible. In the event that something is compromised, that database is at least isolated and other applications are uneffected. 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 only to the specially tailored views. These strategies embrace the least privledge idea of security, where it is an unnessecary risk to be privy to more information or have more access than you need

Research has been done on examing 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 safe should always be the first and most important step if it is possible to

determine if a particular file is vulnerable that easily, detection is a nessecary backup plan.

3.2 Signature Based Detection

A very traditional way of detecting for security threats is the use of signatures, however many of these signature based tools are more suitable for the lower levels of the OSI model rather than the application and presentation levels. These tools are referred to as Intrusion Detection Systems (IDS) and rely on regular expressions and other pattern matching tools produced using existing or previous attacks, one example of such a tool is Snort. Therefore, as long as there is an adequate number of signatures that cover the broad spectrum of possible attacks then the technique can be quite accurate.

However signature based detection systems as well as other IDS systems can have many problems associated with them. One of the biggest problems is the frequency of false positives, when the system believes that something that is not harmful is. Of course the opposite is also true and IDS systems can let attacks slip by, this can be caused by the attacks using various tricks to evade detection such as using alternate encodings or fragmenting packets. In addition, if the IDS is signature based than what is most likely the problem for these accuracy problems is a lack of signatures that are either more accurate or cover new undocumented attacks. It is becoming much too impractical to produce these signatures fast enough due to the countless variants of the attacks and that the attacks are commonly designed to be targetted and go unnoticed rather than spread as fast as possible like with conventional computer attacks. To give an idea on how difficult of a problem this is to solve with signatures, an average of 5,000 new software vulnerabilities have been identified per year; and with the number of unique malware programs alledgely in the tens of millions and doubling every year. With these rising trends, it is clear that the malicious user is easily always ahead of the detection tools,

static solutions such as signature sets are becoming less and less practical every year, and the current short comings of the detection sysytems themselves proves that point.

However, this problem is not unique to just 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 and 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 signature, such techniques include but are not limited to hidden Markov models or reversing the morphing process of the malware. If the area where signature based detection is the most strong has to adapt tactics to deal with the changing environment, then by extension so to it does the web threat detection ecosystem.

3.3 Modern Methods of Detection

In order to combat these challenges for web threats, some people have suggested that a multiple layered approach will provide the best defense. Such a system would not only have multiple layers of detection but also feedback loops to process the information and update the protection systems for future detection. A multi-layered approach would also be able to address all levels of the network rather than a system for only the network layers, and another for the application layers. Such an approach would also enable for portions of the processing to be centralized and on the cloud while other areas be closer to the endpoint. Traditional techniques like signature detection would still be used, but it would be able to be augmented with behaviour analysis for example as often times web

attacks are carried out in massive enumeration attempts and not a single bad request. One final point is that 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 not constantly and consistently updated.

This kind of a multi-layered approach combines the best of the old techniques with new potential solutions and most interestingly suggests a system that is inherently evolutionary, growing and improving as a core trait.

Chapter 4

Describing Machine Learning Approaches

4.1 Machine Learning

Machine learning has become a very popular topic in recent years due to the sheer amount of data that has to be processed. Not only does this large amount of data need more advanced and intelligent ways to dealing with it, but there is also a huge incentive to do so. By using machine learning techniques additional meaning and patterns can be extracted from the data and conclusions can be drawn from it. Machine learning is often applied to data mining tasks, looking at data and decerning information from it, in this case the data being mined is the large amount of web requests.

This has numerous benefits when applied to a security application such as web threat detection. Primarly it allows the system to have some sort of feedback mechanicism to improve its detection and perhaps even prevention. A system that is unable to learn overtime will not be able to overcome new techniques designed to evade the current detection systems unless it is manually updated. And as discussed before, manual updating is not only a very inefficient time consuming task, but it is becoming infeasible with the

growth of these attacks. Identifying patterns from data itself is very useful when the fact that many of the attacks follow some kind of a basic syntax or format, while there are many ways to evade detection typically a common method and intent for the attack to be done.

The way machine learning operates is by identifying a series of features in the data set in question, the features can be classified as continuous, categorical or binary. These features are used by the algorithm to learn and to make decisions, the key distinction about machine learning is it is not told what to do but instead is allowed to make its decision based on measurements such as performance.

4.1.1 Supervised Learning

Machine learning algorithms can either use 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 from the beginning. There is also another type of learning called reinforcement learning where an external source informs the system how well it is working or not.

Supervised learning has its issues that have to be overcome however, the first of which being collecting the original data set. If there is prior research or people in the know that can suggest what features to use than the process is more trivial, however if not then this is often identified using a brute force method. The problem with using a brute force method other than the complexity is that the data has the potential to be noisy and missing important features which can lead to further problems. Learning from extremely large datasets is very inefficient and slow as well, therefore it is often desired to minimize the data set while still maintaining the performance of the system; this process is referred to as instance selection. Lastly, having a large amount of features in your data set can also increase the complexity of the system. To solve this, irrelevant and redundant features should be removed but if many of the features depend on each

other and cant be removed this can lead to inaccuracy from the results of the learning. In order to remedy this, new features can be constructed or altered that are more concise or accurate which can improve the entire system as a whole.

One of the biggest 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 7.2.4, 7.3.4).

4.2 Genetic Algorithm

A genetic algorithm is a search-based algorithm that makes use of the mechanism of machine learning in order to locate optimal or near-optimal solutions. This is what is meant by the term 'search' that is used in other algorithms such as local search, or simulated annealing, it is not about locating something from 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 reward systems to distinguish between a better solution and one that should not be considered anymore.

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

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

Algorithm 1: Fitness algorithm for use in genetic algorithm

Correct detections improve the fitness of a particular individual and it is more likely to be selected for genetic operators later on, where as false positives and incorrect detections impact the fitness negatively with incorrect detections having less of an impact than the

former. For example, if we are looking for SQL injections, every request that is a SQL injection that is detected is correct, if it identifies a request that isnt an attack at all as a SQL injection then it is a false positive and if it identifies an XSS or RFI attack as an SQL injection, then this is incorrect.

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. Two individuals at a time in the population are selected by a selection algorithm and first crossed over, there are many ways to perform a crossover but a single-point crossover will be used for this research. A position is selected to perform the crossover, this is referred to as a locus, for our purposes this refers to selecting a segment, this segment is then swapped with the other selected individual to produce two individuals with unique chromosomes, or in other words a different configuration. This continues until the algorithm has produced enough new individuals to fill the population, in addition, at the beginning of this process it is possible to mark some of the top individuals as elite and preserve them over to the new population. Before continuing to the next iteration every single allele, or piece of information, in each individual has the small potential to be mutated this is what causes the population to have diversity. This 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.

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 various 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 in generating different individuals and seeing if they perform better. These exploit variants were used to test signatures based detection methods to see if it was possible to evade them and

results showed that it was. This is proof that traditional models for detection cannot be made absolutely perfect and that the technique of using genetic algorithm technniques is atleast worthwhile for evading detection.

As of recently however, research has taken this idea and done the opposite, using a genetic algorithm directly to detect web attacks rather than evade detection. This was done by using the genetic algorithm to make variants of attack signatures that 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 a traditional regular expression signature based detection system.

For this research, this recent research is the starting point for improving the genetic algorithm approach and gathering more detailed results about its function, as well as the comparison point to another machine learning technique, support vector machines.

4.3 Support Vector Machine

A support vector machine's main technique for classifying data is by dividing the data set in two or more categories with the largest margin between the seperation(s), referred to as a hyperplane(s). The point in maximizing this buffer between the two data sets is to reduce the chances of error as much as possible. Once this hyperplane is computed, points that lie within the margin are referred to as support vectors, hence the name, and it is these points which were what calculated the hyperplane in the first place, the other data points were ignored (Figure 4.3).

The fact that the SVM is determined by the support vectors which is usually a very small subset is great because it means that the speed does not significantly slow down with a larger amount of features. However it is very realistic to imagine data that cannot be easily divided and so soft margins that allow for misclassifications and in worse cases the data can be mapped in a higher dimensional space to open up other options.

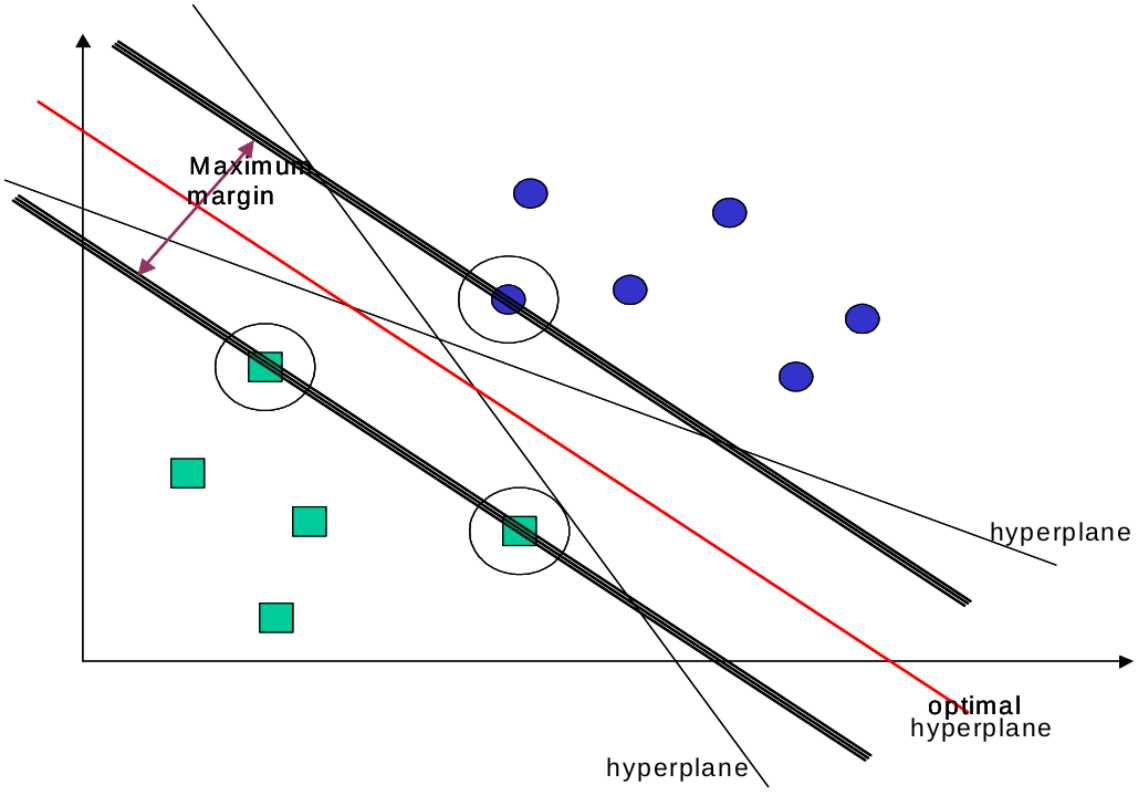

Figure 4.1: Example of a linear seperated SVM

This higher dimensional space is referred to as the transformed feature space, however it makes simple linear seperations in the higher dimensional space transform into non-linear seperations when you return back from the higher dimension.

If this feature space data is mapped to Hilbert space, referred to with Φ, which allows for traditional vector calculations to be extended to many dimensions using dot products using equations in the form of: $\Phi(x_i) \cdot \Phi(x_j)$. This means that we can use what is referred to as a kernel function to avoid ever having to determine the mapping to Φ and can calculate directly in feature space. A kernel function is in the form of: $K(x_i, x_j) = \Phi(x_i) \cdot \Phi(x_j)$. 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 4.3).

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

Once the SVM is trained using the kernel method the only step is to pass in all of the testing data and see where on the graph 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 choosen as well as the parameters that go into 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 seperation. 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 others. On the converse if the gamma is too high then the influence would only be the support vector itself. The C parameter is the penalty cost assocated with misclassification, if C is low then classification is more relaxed where as a higher C will encourage more support vectors to be choosen to achieve a more accurate result. There is no way to know what are the best values to choose because it depends on the dataset in question 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 attacks. 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%.Likewise, another study compared the usage of an SVM versus an artificial neural network and found that the SVM was much faster in comparison along with achieving a 99% accuracy as well.These two studies shows that the SVM approach is a viable one and can be used for practical applications with web threats, so it will be interesting to see how well the algorithm preforms compared to the genetic algorithm approach for application layer attacks as well.

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 with the genetic algorithm or support vector machine components as loosely coupled modules to avoid having to redesign the system for both approaches. Web requests will be processed through a parser that looks for various aspects related to each of the three possible web attacks. The results are then output and can be used as input for either the genetic algorithm, support vector machine, or potentially another algorithm that could extend the testing. Finally, the testing 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 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. 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

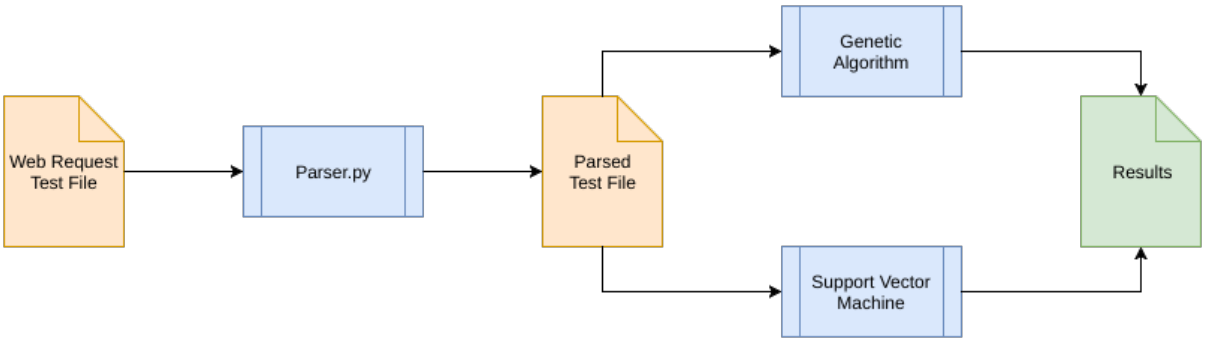

Figure 5.1: Overview of the system

Request Type	Generation Method
SQL Injection	sqlmap
XSS Attack	grabber, xsser
RFI Attack	randomized generation
Normal Requests	httpfox

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. In regards to 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. Because RFI maps are the simpliest 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 nessecary 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, submit form inputs, etc, and collecting all of the resulting HTTP requests. This can be done with a browser extension for Mozilla Firefox, HTTPFox.

Like most web-servers, Apache has the ability to log all of the requests that it serves to a file. The data that we need to parse for the genetic algorithm or support vector machine is the GET or POST HTTP request line. The final step of gathering the testing

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

and training data is to compile the log files and strip out the unneeded information so it can be passed to the parser. Another small Python script can be used to generate a test file using these large banks of the correct size and proportions, this 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.

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 that the algorithms can work on them. Each request has their own respective features that are worth identifying, with more features needing to be identified for the genetic algorithm than for the SVM. These features have been identified for each request type by previous research (Table 5.3) but is important to distinguish the meaning of each as well as what can and cannot be detected in this way.

The number of SQL keywords or reserved words is obtained by using a comprehensive list provided by the Oracle and MySQL DBMS documentation. This works well with our test environment as well, as the DBMS our application is using is MySQL so many of the automated attacks will use MySQL specific vulnerabilities. Similarly, the HTML and javascript keywords were provided by the official W3C documentation and the official PHP related commands were sourced from PHP's main documentation.

Requests are capable of being encoded as well to further evade detection (Section and

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

Figure 5.2: A sample HTTP request with fields highlighted

so this is something that can be easily detected and recorded. HTTP requests, usually GET requests specifically, will contain along with them fields and the information that they carry 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 5.3). 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. Lastly, all of the discussed attack variants (Section can be detected with the exception of *Stored Procedure SQL injections*, which brings up the limitations of this type of parsing (Section

The parser makes heavy use of regular expressions (Appendix A 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 follwing order Original ¿
        SQLi ¿ XSS¿ RFI
read in input file;
for line in input file do
    if for 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).

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 and tested system in previous research with a few modifications. One major difference is that instead of allowing the signatures to change to different attack type signatures (ex. SQLi to XSS) we specify what type of attack we want to search for and the algorithm uses that parsed result for 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 causes later conclusions to not be influenced by factors that can not 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 or the other variables being changed.

The second major change is that 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 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, or 111, resulting in a match which should not have occured. Therefore these segment lengths with a size problem have been extended to 6 bits allowing for counts up to 63 (Table 5.4).

The genetic algorithm implemented is very standard, allowing for the following parameters to be changed:

- Maximum population per iteration
- Maximum number of generations
- Number of iterations

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

- Mutation rate

- Elitist selection amount

Most of the genetic operators are fairly simple to implement with the most complex being selection. The higher the bitstrings fitness is the more likely it should be selected. There are several ways for this to be accomplished but for this implementation Roulette Wheel Selection is used, also referred to as Fitness Proportionate Selection (Algorithm 3). This selection algorithm was choosen as the originally proposed genetic algorithm technique was fitness based and not reward based like other selection algorithms, as well as it is simple to implement and understand for these basic testing purposes. For

crossovering two individuals a single point crossover scheme is used.

```
Data: Fitness values of all individuals in population
Result: The selected individual
totalWeight ← 0;
for all individuals weights do
    totalWeight ←totalWeight + weight;
end
generate a random number between 0 and the totalWeight;
for all individuals weights do
    subtract the weight from the random number;
    if random number is less than 0 then
        return that individual
    end
end
if Unable to find an individual, error occured then
    fall back condition is to return the last item;
end
```

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

The genetic algorithm was written from scratch using the Python programming language and is designed to handle input files with single bitstrings from the parser or lines with several variations on the lengths of the bitstrings (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 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 is less than maxmimum population allowed do
            locate two individuals and perform a single point crossover ;
            add these two new individuals to the next population
        end
        trim the offspring to the maximum population just incase;
        loop through every bit in every offspring with the chance to mutate it;
        set the current population to the offspring
    end
    store the bitstring 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 metrics (Table 5.4.1 & 5.4.1). In addition, the testing data will be different requests than that used in training as this is the situation that these approaches would be exposed in the real world. They would be trained using supervised learning and then used to identify unlabeled data entering the system so it does not make sense to test with the same data it is trained on.

Every test will make use of the same training data and testing data and instead the parameters for the genetic algorithm will be altered to see if they make a difference on the results. The genetic algorithm will be trained using the training data, which will output bitstrings that act as signatures that are optimized for detecting the particular attacks. Those signatures will be used to hopefully match correctly with the unseen testing data (Figure 5.4.1).

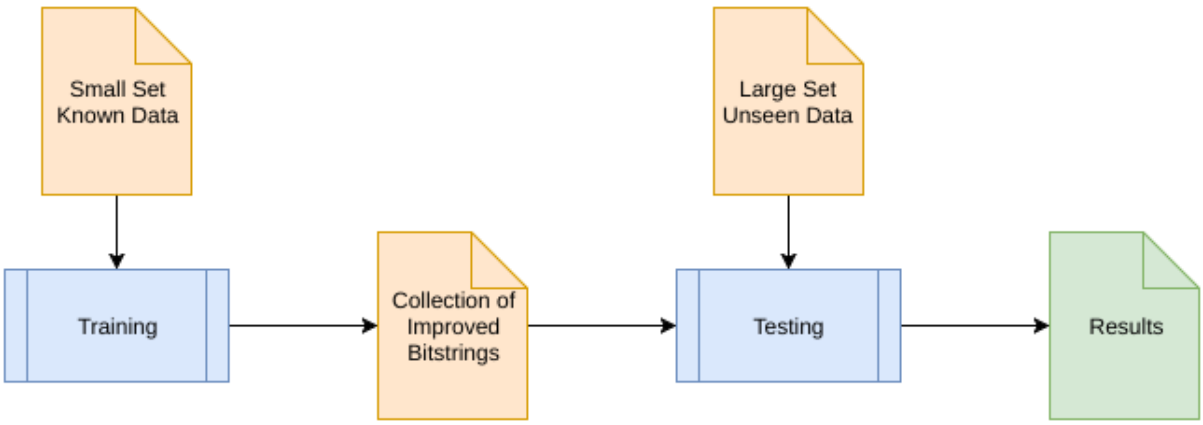

Figure 5.3: Overview of the genetic algorithm system

In addition to the various parameters to the genetic algorithm having multiple iterations of the algorithm ran to generate a combined amount of signatures will be tested. The thought is that the more signatures in your detection set the more likely you are to have one to detect the attacks so more iterations combined together should be able to detect more attacks. This is also one of the main reasons why this technique was proposed, so that the genetic algorithm could generate additional signatures automatically for testing instead of relying on manually made patterns. Also, testing the results with different lengths of bitstrings will also be attempted, the thought here goes back to the problem of overflowing a segment (Section 5.4). Small segments should be able to detect more attacks as they should be able to more easily generate bitstrings that cover a wider range of attacks. For example, a segment that can only hold a count of 1 would detect anything if it contained a keyword even if that request had a much greater amount, it would essentially become a flag.

5.5 Support Vector Machine Detection

The SVM detection will follow a similar process to the genetic algorithm but instead of changing the parameters of the algorithm, the training data will be changed instead. This is because the parameters for the SVM that will make a difference are automatically

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: Genetic algorithm default segment breakdown

optimized by a grid search approach provided by the same library used to implement the SVM in Python.

SVM use various different kernel methods to determine a pattern in the given data, the kernels that will are being used from the provided library is a linear, polynomial degree 3, and RBF kernels. This will be the only parameter that will be changed for the SVM but every test is ran on each kernel so it is consistent throughout the entire process. The parameters for the SVM that are optimized by the gridsearch are gamma and the penalty cost, gamma only effects the polynomial and RBF kernels however.

The one main difference between the SVM and genetic algorithm approaches is that the SVM only requires two of the four segments and it does not need to be in binary form to support mutations (Table 5.5). The reason that the other two segments are not used in the SVM is because they do not vary as much as the other two, many attacks and non-attacks could be encoded or not as well as the same attack type so that information is not very descriptive on it's own. These values are plotted on a simple X,Y plane which is then passed into the SVM to be trained, each of these values is labelled data cooresponding to either it is an attack or it is not (Algorithm 5).

Data: Segment information for each request
Result: A trained SVM classifier that test data can be passed into and results gathered with
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 that the training data will be adjusted to produce different results (Figure 5.5.1). The exact same testing set will be as what was used in the genetic algorithm, and for early tests the same training data will also be used but beyond the initial 1000 sample size, new training data must be used. The first test 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 will see if false positives can be reduced by increasing only the amount of false positives between the various tests. And lastly, seeing if the number of incorrect attacks can be reduced, incorrect being identifying a request that is an attack but as the wrong type of attack, by increasing only the amount of the attacks we are **not** looking for.

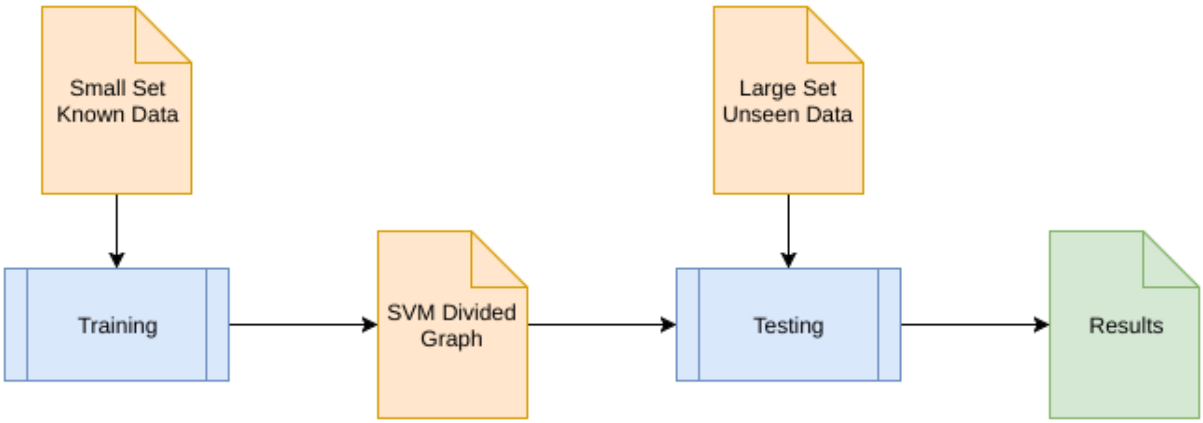

Figure 5.4: Overview of the support vector machine system

Chapter 6

Results

6.1 Genetic Algorithm

In the following results, each test was ran three times and averaged together to give a better idea of the typical performance of the genetic algorithm approach. Each test is categorized by the parameter that was changed and the reason and expected results of these parameters is explained. Also, with the exception of the bitstring segment length test, all other tests are conducted using the normal segment lengths (Table 5.4). Lastly, for brevity only the SQL injection results will be compared across all test cases as it is the most complex result 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 **??**).

6.1.1 Finding Best Parameters

The performance and effectiveness can often be attributed to the parameters that are used for the genetic algorithm, and there is no universal choice for the parameters as it will depend on your context. Each parameter of the genetic algorithm will affect the results in different ways, therefore it was important to first determine what settings would

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

be most suitable to use for later tests so that the results would not be largely caused by the parameters instead of independent variable in question.

Population Size

Population size is one of the most important parameters as a genetic algorithm with a low population size performs very poorly as there is not a large enough sample size to grow and advance. Larger populations are more likely to generate new individuals that perform better however this also comes at a performance cost. In addition for our purposes, having a larger population increases the chances of having bitstrings that perform badly as signatures, causing false positives and the like to stick around. Additionally, 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.

Generations

The amount of generations also matters because it is the amount of times the genetic algorithm will run, the more generations the more likely that the algorithm can produce better results and improve upon the old ones.

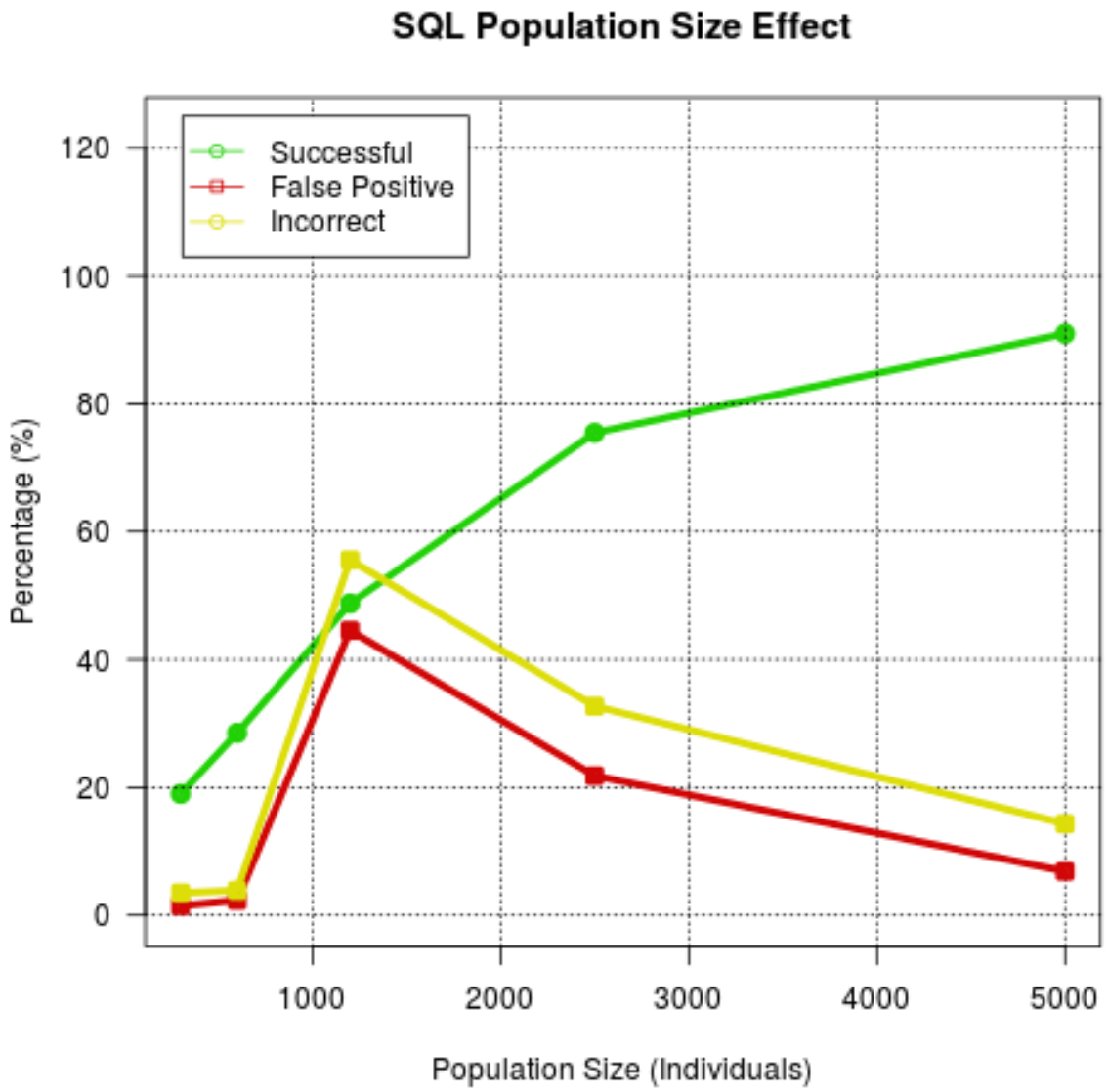

Figure 6.1: Effects of Population Size on Detecting SQLi

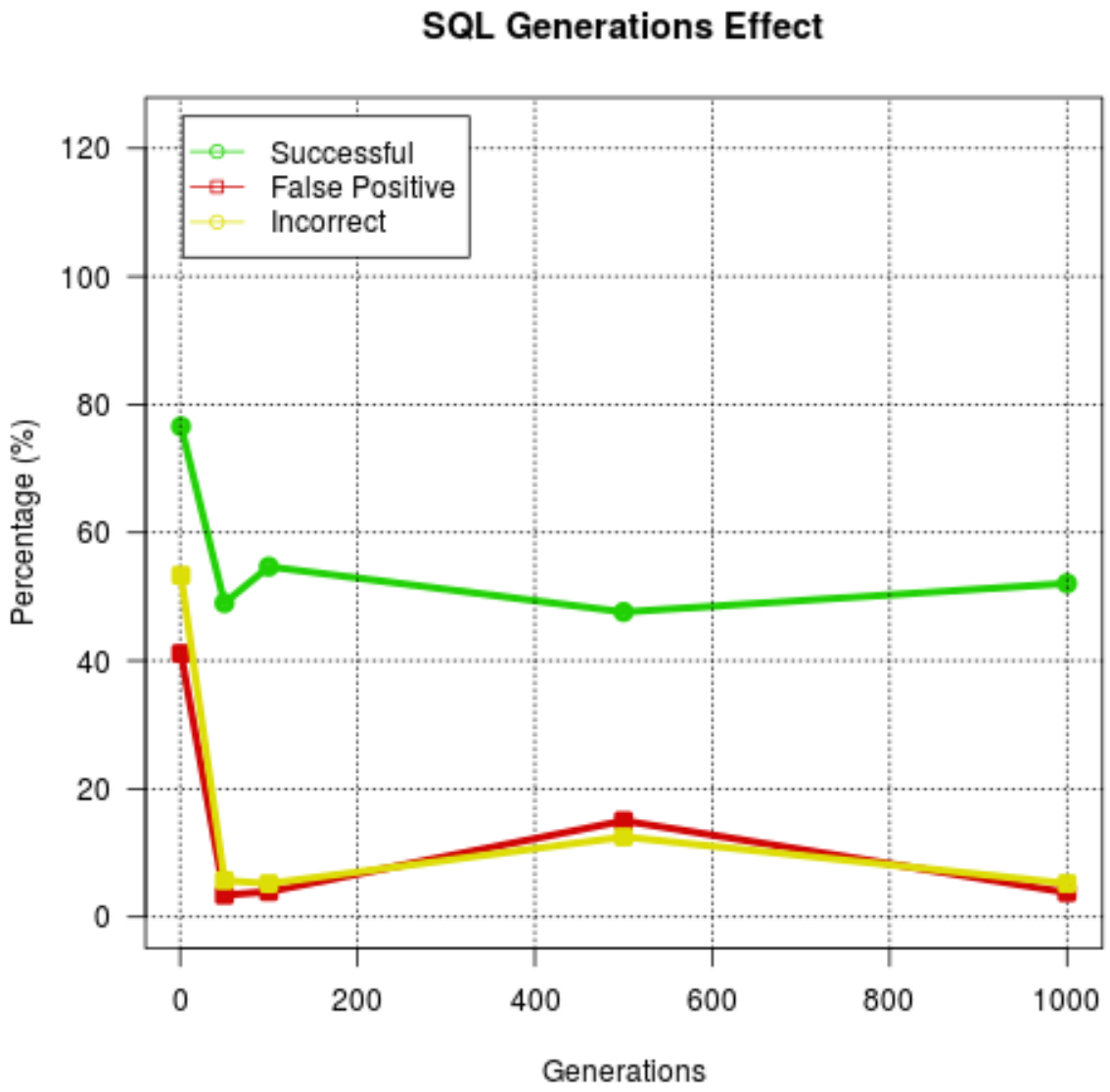

Figure 6.2: Effects of Generations on Detecting SQLi

Mutation Rate

If the mutation rate is too high than the genetic algorithm basically becomes a random search, while no mutation will result in no diversity other than 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. Mutation rates are typically set to a very low amount since it is on a per allele basis, a value between 0.0 and 1.0 is often used.

Elitist Pool

The more of the better performing population that survives to the generation the more likely we will have strong individuals to produce newer strong individuals, but if too much of the population is preserved than it may not be able to improve very quickly. Sometimes this is also referred to as a generation gap.

6.1.2 Combining Multiple Signature Sets

One of the claimed advantages of this approach, and a main reason why it can work is because the genetic algorithm can be used to produce new signatures in order to keep adding to an existing set to increase the detection possibilities. For this reason, running the algorithm multiple times and combining the signatures should result in more detections, buy may also result in more false positives and incorrect detections.

6.1.3 Bitstring Segment Length Effects

Because the genetic algorithm can detection new and more attacks by generating new signatures, if the number of possible signatures is less due to the segment length, then it would be more likely to generate these signatures. However it also opens up the possibility of making it easier to generate bad signatures often.

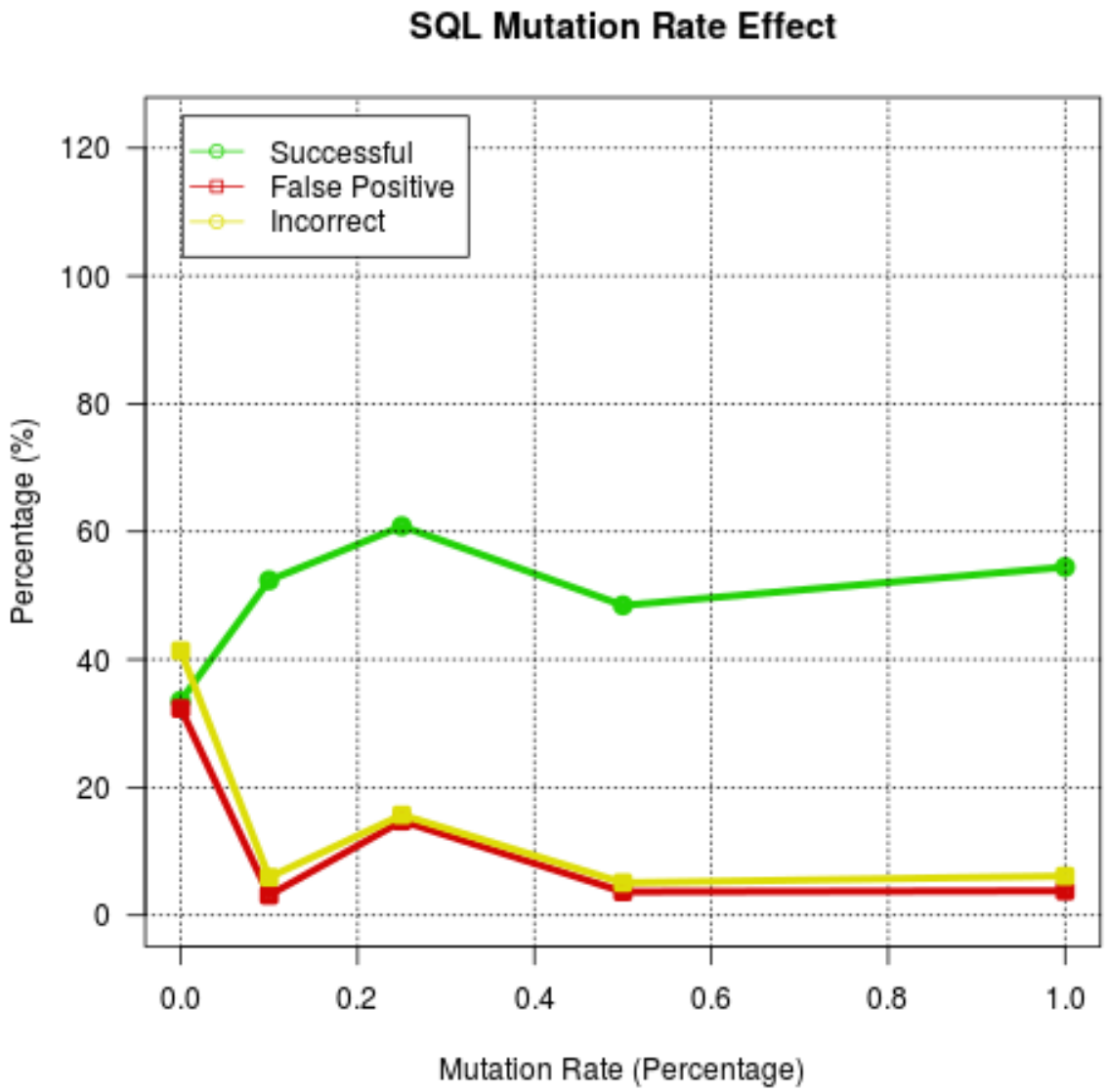

Figure 6.3: Effects of Mutation Rate on Detecting SQLi

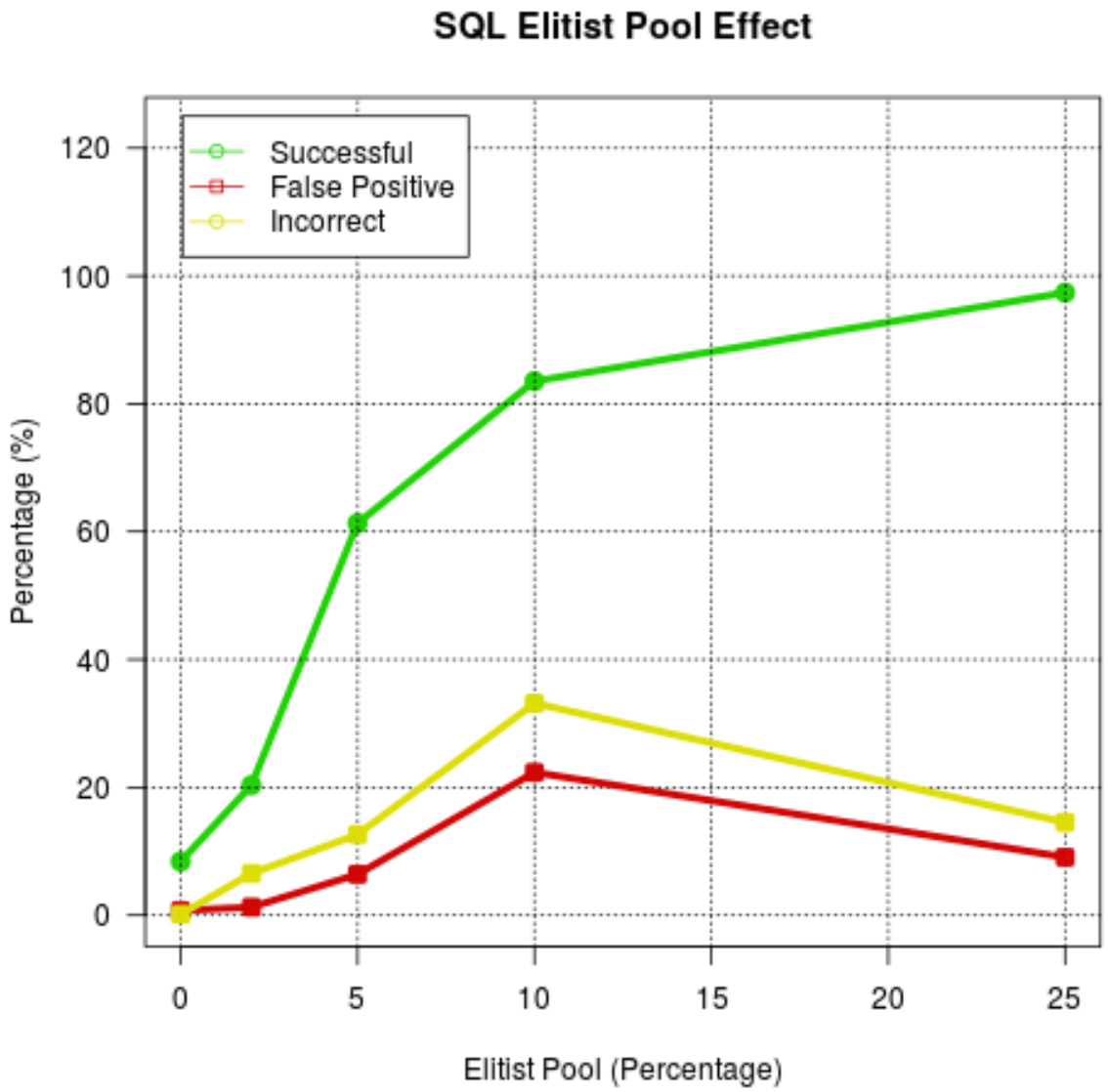

Figure 6.4: Effects of Elitist Pool on Detecting SQLi

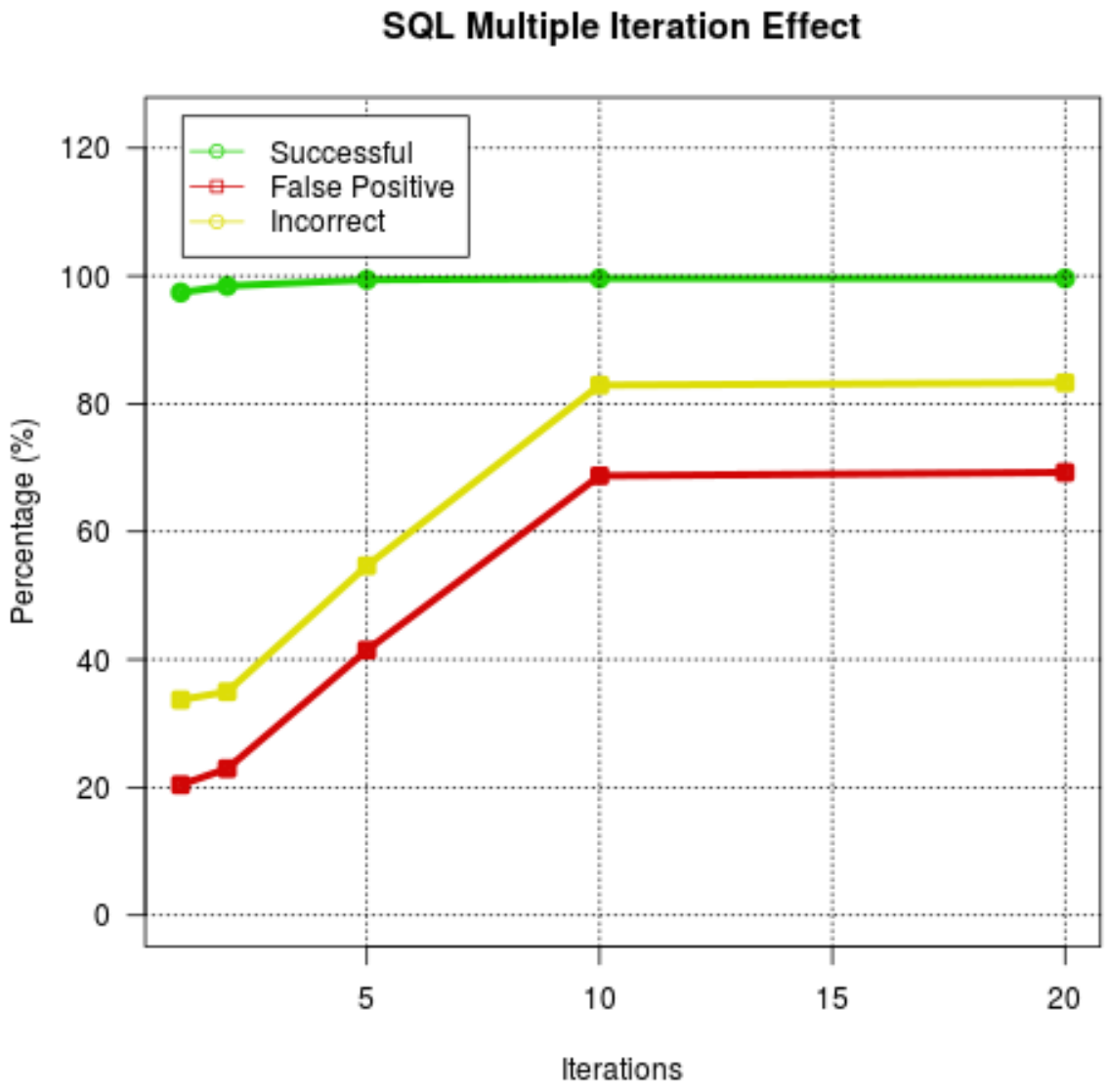

Figure 6.5: Effects of Multiple Iterations on Detecting SQLi

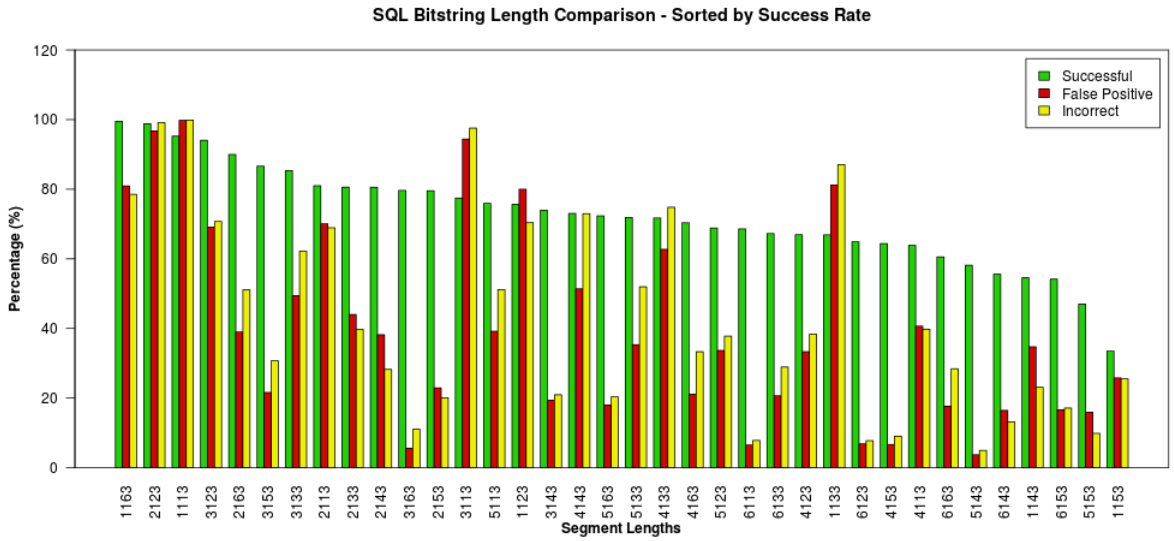

Figure 6.6: Effects of Different Segment Lengths on Detecting SQLi

6.2 Compared With Random Permutations with Fitness

Because the genetic algorithm is able to increase its detection by generating new signatures automatically, it would be interesting to compare the approach with generating all possible combinations and only using the bitstrings that performed well using the same fitness algorithm used in the genetic algorithm (Algorithm 1).

6.3 Support Vector Machine

For the support vector machine tests it was not nessecary to average together multiple results as there are no random elements in the approach and the same results are produced everytime. All results used the same testing data to verify the training process as well as when possible the same training data, so long as the required amount did not exceed the amonut of training data used in the genetic algorithm. In the genetic algorithm changes can be observed by adjusting parameters, however in the SVM this is not the case and

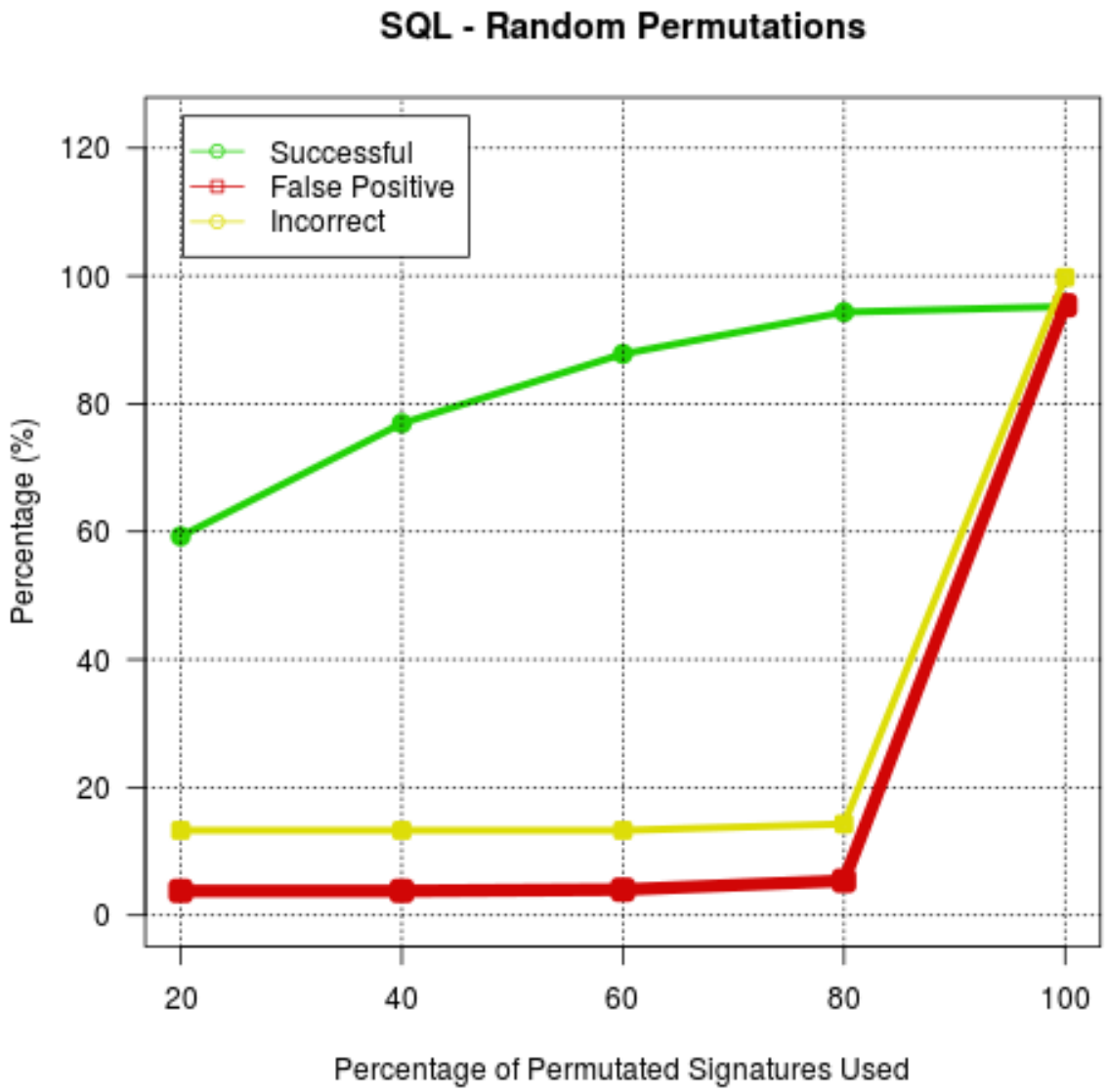

Figure 6.7: Permutation of Bitstrings to Detect SQLi

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

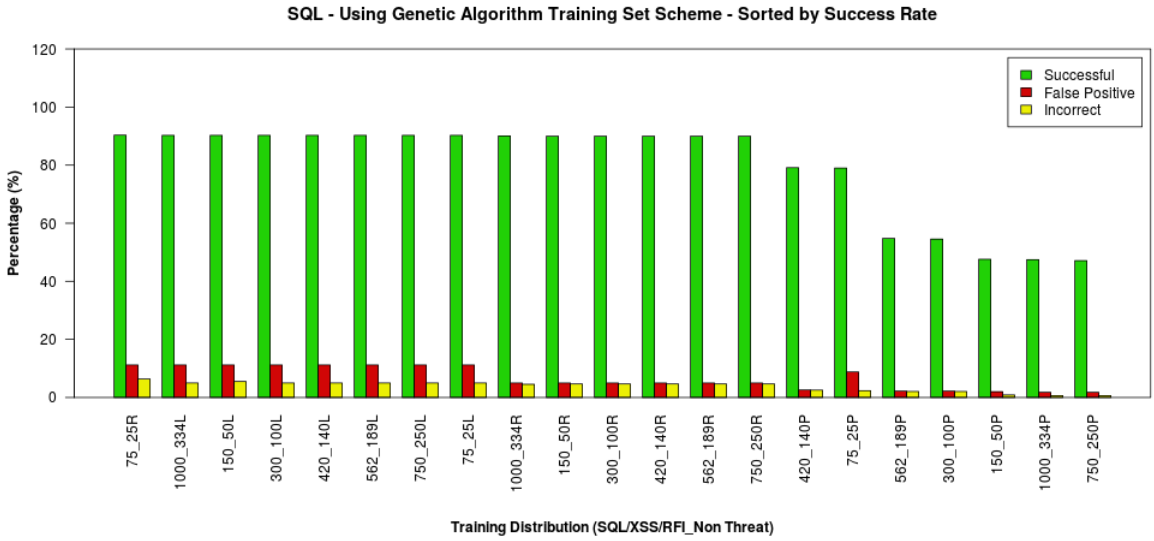

Figure 6.8: Genetic algorithm and SVM comparison for SQLi

so the amount of training data and what kinds of training data is changed to observe results.

6.3.1 Comparison with Genetic Algorithm

The first results are using the same training data, as well as the same training proportions used in the genetic algorithm tests of 30% for each attack type and 10% of non-threats for the remaining.

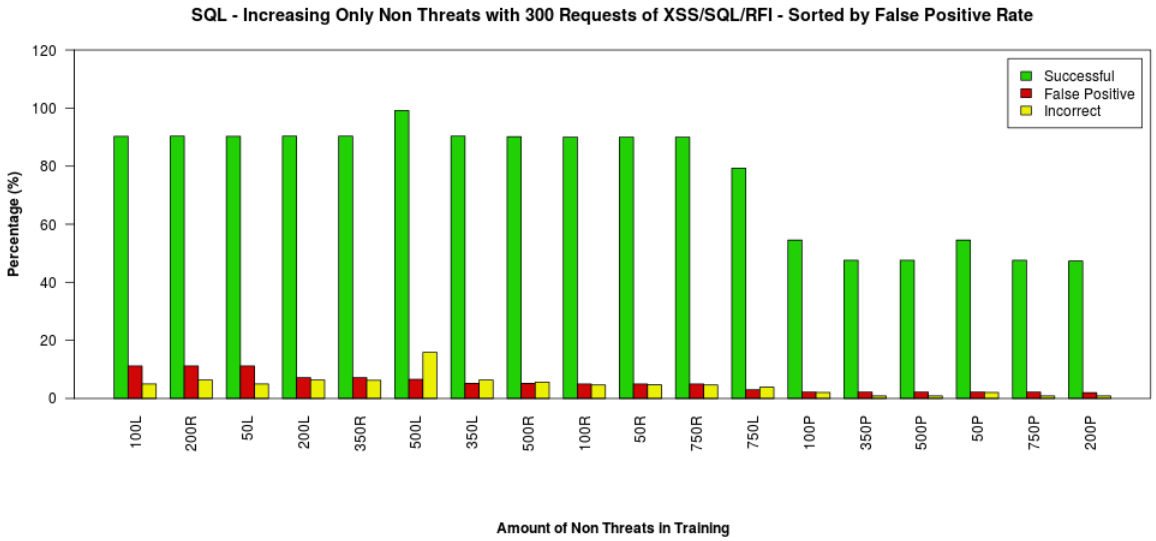

Figure 6.9: Effects of increasing non-threat training data in SVM for SQLi detection

6.3.2 Increasing Non-Threats

Increasing the amount of nonthreats in the training data should creating a classifier that is more resilient to detecting false positives, the more training data the less likely false positives should occur.

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, the less likely it is for the classifier to incorrectly identify an attack.

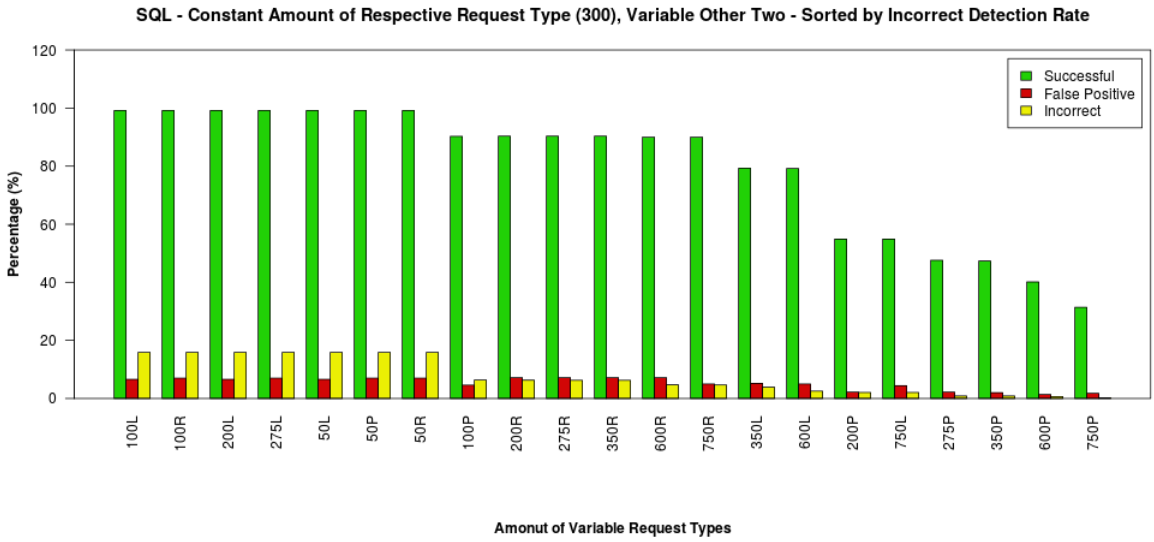

Figure 6.10: Effects of increasing incorrect attack training data in SVM for SQLi detection

Chapter 7

Discussion

7.1 Parser

7.1.1 Weaknesses

The parser is the main point of failure for the algorithms in question, and therefore many of its weaknesses become obvious when working with this system. If the results of the parser are not accurate than it is impossible to have decent or consistent performance from the genetic algorithm or the support vector machine, however both algorithms have been given the same parsed results so the question becomes which algorithm can do a better job.

In the end, despite the fact that these approaches try to avoid the use of regular expressions and pattern matching, they depend on a parser to get results that employ these techniques and as a result there is always a chance that the parser is fooled and not accurate. The question then becomes good is the training process of these machine learning techniques in order to discount these mistakes. Part of the reason why the parser may be vulnerable to making mistakes as well as it is only a simple token-based parser. This means that the parser cannot understand the logic behind the requests or their intent only picking up on keywords and patterns within the requests, if a parser was

made that used techniques such as abstract syntax trees or program dependency graphs then it may be able to get a more complete picture and generate more accurate results. This limitation becomes obvious as some attack types not covered in this research, as well as even stored SQL injections cannot be detected as the information required to make that determination is outside of the scope of the text request.

A last point of weakness for the parser is that some attacks were much easier to parse than others. The SQL injections are the most complicated of the three and the keywords are essentially english words so there is always a chance that it gets confused with a normal word as an SQL word. In contrast, the XSS keywords must follow a syntax similar to the RFI keywords so these attacks do not suffer from such problems. The RFI attacks however had to be randomly generated in order to get a significant sample size and so there are some inherent biases there. All of these issues can skew the results in someway or another.

7.1.2 Strengths

Despite the problems that could potentially be solved by moving away from a token-based parser, it is the fastest of the parsing techniques as building the trees and dependency graphs can take a long time. In addition, for the purposes of detecting the features in question for the attack types being examined a token-based approach is really all that is required as it is just counting up keywords and testing for whether or not it is encoded. Using a more complicated technique than this might be overkill and reduce the speed of parsing when attempted to be used in a potential real world environment.

Also, despite the weaknesses previous mentioned, parsing the results like this is realistically the only way to determine the intent of the incoming attack before it happens. There is no other useful information to pull from the individual requests and no determinations can be made on the intent of the request because it has not yet executed.

7.2 Genetic Algorithm

7.2.1 Parameter Testing Results

In regards to changing in population size, the expected results were observed. As the algorithm was allowed to generate more and more signatures the detection rates increased and false positives and incorrect detections went down but did not always dissappear completely. This is the most obvious result as greater population is needed for situations like this, however it comes at a very large amount of computational cost. In addition, increasing this result alone cannot 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 went slightly down as the tests went on.

Increasing the generations did not seem to have any discernable effect on the performance of the algorithms and in many cases it made the results worse and more varied. The number of generations alone is not enough to optimize the results of the algorithm clearly, as the most extreme example is in the XSS test where it goes from a very high detection rate to 0% later on. Every generation the entire population is thrown out depending on the elitist pool, this means that while the previous generation might have good signatures, the next generation's offspring could have terribly performing signaures (Section 7.2.4).

Mutation rate changes had almost no discernable effect as well, in some tests the results improved, while in others like the RFI attacks, the results flatlined. Mutation rate only slightly effects the genetic diversity and the vast majority of the performance is determined by the bitstrings themselves generated by crossovers for mutation rate alone cannot save the algorithms performance.

Finally, out of all tests the elitist pool changes had the best results interms of positivity and impact. For all three attack types, the more of the best performing population that was perserved the better, most likely because it allowed the changes from generation to

generation to be more consistent and not be left up to chance.

7.2.2 Expanded Signature Set

Doing multiple iterations of generating signature sets resulted in all three attack types being able to hit near 100% successful detection, which was not seen using the other manipulations so far. With the exception of the RFI attack results most likely due to it's inherent biases (Section **??**), XSS and SQL injection attacks also increased in their false positive and incorrect detections drastically. This is a clear result of having multiple iterations as the larger and larger your signature set is the more likely it is to contain a generated signature that performs badly.

7.2.3 Influence of Segment Length

Out of all of the variations however, the segment length changes is the most telling of this approach. It is quite discernable from the results that the smaller segment sizes result in higher success rates, false positives, and incorrect detections on average when compared to the larger segment sizes. This matches the expected result as not only does having a smaller segment size lead to inaccurate bitstrings that become artifical matches, but it also makes it much more likely for a poor performing bitstring to get through the 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 that can at times perform very well. However, due to the high variability very poor results interms of false positives and incorrect detections can be found everywhere in the results. It seems that the approach is best when given existing well performing bitstrings, this result can be seen through the

elitist pool testing, however in these tests it did not achieve the best success rates possible. This is likely 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 we decrease the chances of those bitstrings ever leaving the population in question. Lastly, the genetic algorithm has a lower dependence on the parsed results as the SVM will because the results are more aligned with the random generation of signatures rather than making conclusions off the results.

However from these results overall the approach seems quite weak and 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 it's parsed result must be generated, however the fitness algorithm does not rate how close the bitstrings are to this ideal signature, rather it rates how well it detected multiple results and avoided mistakes. It would be impractical to use a genetic algorithm to do this, as it would be much easier to just construct signatures from parsed results of known attacks. Therefore, instead the fitness algorithm's intent instead seems to be about keeping the better performing bitstrings around until the end of the algorithm and hoping to kick out the bad performing bitstrings; however this does not always work out as explained above. This is the root cause of the variability between generations and what causes the odd anomalies in some of the results. For this reason, the genetic algorithm seems to be a bloated random permutation generator with a fitness algorithm to remove the poorly performing bitstrings, and so this claim was tested (Section).

In addition, segment lengths of the bitstrings are very important. If the bitstrings are too small there is a higher chance to generate a poor performing signature as the search space is much smaller. 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 your population is around 1000 it is very likely that a bitstring representing a non-attack or incorrect is generated and the fitness algorithm

will not remove it. Likewise, having such small segment lengths are not applicable to the real world, enumeration attacks can use dozens of keywords per segment having such a small count in each segment results in artifical matches that would not occur otherwise. For example, if every single SQL injection attack had a unique number of keywords but above 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 detecting the attack, it is artifically padding the results of the approach.

7.2.5 Comparison with Random Permutations

The results of the simple random permutation approach 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. In addition to these excellent results, the approach also completed in a fraction of the time as the bitstrings only had to be randomly generated once, rather than multiple times over again in the genetic algorithm with additional selection procedures.

This shows how it really is just the random permutation process that is giving the genetic algorithm the results that it is, and therefore the genetic algorithm component is unnessecary and ill-suited. While the amount of permutations exponentially grows with the size of the bitstring signature, the size used for the important segments that maintains the counts 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 these results it is obvious 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 very minimal false positive and incorrect detections, only showing up in the SQL injection results infact.

7.3.2 Reducing False Positives

Increasing the number of false positives did indeed decrease the amount of false positives overall in the results for SQL injections, as the other two attacks did not have substantial amounts of false positives to begin with. This shows that training does indeed directly affect the performance of the SVM.

7.3.3 Reducing Incorrect Detections

Similar to the results of the false positive training, the more incorrect results input into the system the better the SVM was able to avoid them. However as a whole these results were not as much of an improvement, and this may be due to parsing mistakes with the larger amount of training data effecting the results.

7.3.4 Strengths and Weaknesses

Overall, the SVM approach appears to be a much stronger choice when compared to the genetic algorithm and even the permutation approach. One of the advantages if 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 runs much faster if the proper kernel is choosen, as the RBF kernel has good results

but is quite slow and complex, and the Polynomial kernel is slower and produced very poor results in comparison. Therefore the linear kernel which is the fastest of the three, and also performed very well overall is the clear choice and not a problem, otherwise this would be a weakness. In addition, the SVM requires fewer features to be parsed from the request, it only needs the keywords and the field counts and the attack variant and encoded check is not required, this speeds up the parsing process.

The main weakness of the SVM approach is its strong dependence on the parsed results, if the results are not correct from the parser than the algorithm can not hope to train properly and produce valid output. But considering that the same training data was used in both approaches and the SVM came out hugely on top, and the same results can be reproduced over and over, it does not seem to be too much of an issue. In addition, despite the mentioned biases with RFI parsing, the fact that the SVM achieved no false positives or incorrect results for RFI detection is not a problem due to the biases, as the XSS results were the same way and they are all generated through attack scripts.

Chapter 8

Conclusion

8.1 Conclusions

8.2 Future Work

Appendix A

Regular Expression Documentation

lorem

Appendix B

Full Genetic Algorithm Testing Results

B.1 Full Text-Based Results

lorem

B.2 Remaining Graphical Results

lorem

Appendix C

Full Support Vector Machine Testing Results

C.1 Full Text-Based Results

lorem

C.2 Remaining Graphical Results

lorem

Bibliography

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

[2] 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.