Designing and Evaluating Application Layer Web Threat Detection with Evolutionary Algorithms

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 with Evolutionary Algorithms

Tyler Wilding

Bachelor of Computer Science (Honours)

Department of Mathematics and Computer Science

Algoma University

2017

This thesis examines the use of evolutionary algorithms, named support-vector machines and genetic algorithms for the purpose of detecting application layer web threats, namely SQL injections, cross-site scripting, and remote file inclusion. Detecting these attacks becomes more of importance as the Internet grows and leveraging machine learning is one of the many potential avenues to improve this area in secuity. The examination entails running the algorithms on 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 while the support-vector machine did solve several of these issues it's complexity could be troublesome for real-world application.

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 amount of 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 consideration when developing a new Internet connected application.

However, prevention is not the only task for dealing with these attacks as detection is equally as important. Among many reasons, one example is an attack is a zero-day attack and had no prevention method planned for and is already causing damage. In this scenario detection is the first line to defense to inform those in the know to take nessecary actions.

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

Therefore as of recently the focus has shifted to using other techniques to attempt

to improve on the downsides of the conventional approaches using machine learning techniques, such as genetic algorithms. These approaches show great promise, but no system is perfect and the research is fairly new and they have no been as picked apart as more established methods have.

1.2 Objective

The objective of this research is to more critically analyze the genetic algorithm approach presented in previous research, in addition to comparing it to a new approach of using a support-vector machine to classify the requests as threats or not threats.

The critical analysis of both evolutionary algorithms will involve determining the success rate of detection and how often attacks are mis-identified either as a false positive or the wrong attack type. Through this information a more complete picture of the algorithms can be gathered and much more informed conclusions can be drawn.

Another objective of the thesis is to ensure that testing is done accurately and fairly between the two algorithms, this means testing with the same data that is different from the data used in the training process.

Lastly, in order to turn the requests which are ordinary text-strings into usable data a simple parser must be made that is tailored to each attack types nuances.

1.2.1 Hypothesis

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 much more suited for the problem. In addition, the genetic algorithm relys on the semi-random occurance of generating a new signature in order to detect further attacks which suggests it may have highly-variable levels of success.

However, the support-vector machine can get quite performance intensive when using

more complicated kernel types and is reliant on the training data. If using more complex kernels is required to achieve good performance, 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 this application and other evolutionary or machine learning techniques will not be examined. To that effect, it is also limited to examining the three application layer web threats: SQL injections, cross-site scripting and remote file inclusion.

The two approaches 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 that will have labelled data that would not be typical of the real world but is nessecary in order to determine results.

Chapter 2

Web Threats

2.1 What is a Web Threat

Put simply, 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. The main motivator behind these attacks is for the pursuit of confidential information and it is becoming more and more commonplace to hear about large scale data breaches. While it is difficult to track all of the monetary gain from these activities due to the 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.

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

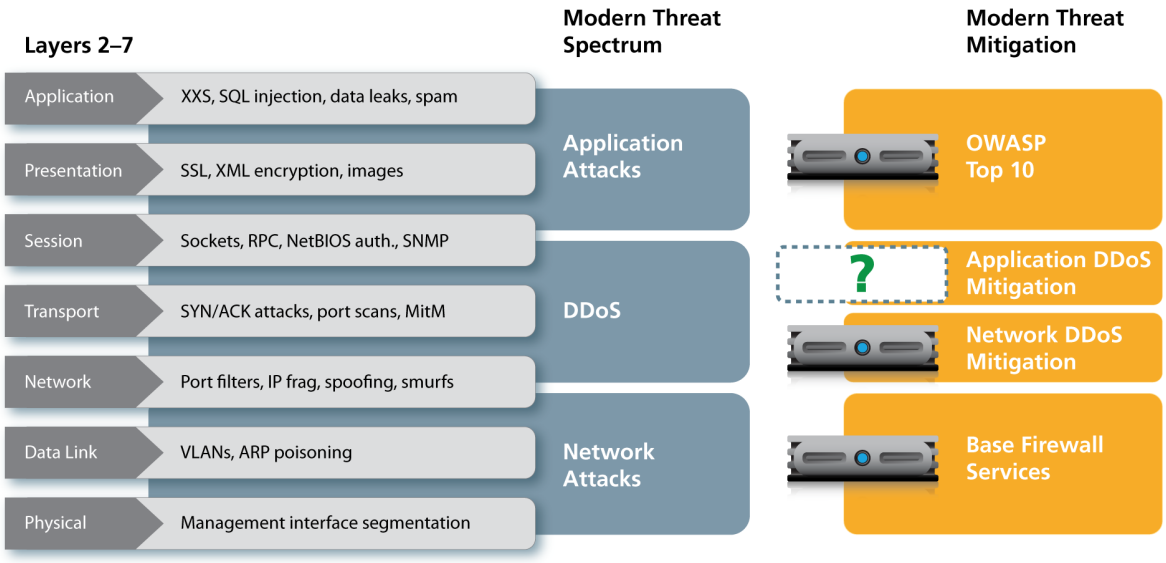

Figure 2.1: Possible threats at each OSI model layer and possible mitigation techniques.

possibilities. This leads to the conclusion that conventional approaches grand-fathered in from other security sectors like virus scanning are not adequate for web threat detection. There are two main 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 enough, the second reason is that unlike conventional viruses web attacks are designed to go under the radar instead of spreading as fast as possible. Web servers are required to be publically open to the world unlike traditional desktop computers which typically have all ports closed, this opens up the problem of accepting information from unknown parties. Therefore, modern solutions employ a much more layed approach using conventional detection methods along with reputation systems.

Web threats can target any level 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, it is only mitigation and there is always the possibility for a new attack variant to slip past.

Attacks will always prefer the most vulnerable target, the weakest link, and there is nothing more vulnerable than the public-facing application. 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 considered easy to mitigate, the existing solutions are not completely adequate for defending against some of the most common web attacks. In order to solve this problem new methods of detection and prevention need to explored and older conventional methods need to be improved.

2.2 SQL Injection

Structured Query Language or SQL or 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. When one of the primary reasons behind performing these attacks is to gather confidential data this is an obvious means of acquiring the information.There are four common methods of injecting the SQL commands into the application:

- Injection through user input

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

Injection through user input is the simpliest method, where the malicious user simply enters the arbitrary SQL code into any field that has some interaction with the database. Injection through cookies involves applications that read from the cookie's fields to restore the users state, this information can be modified to contain SQL code to be read on returning to the website. 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 be exploited in a similar fashion. Lastly, second-order injections are some of the hardest to detect because they involve an attack that occurs later on after the data is entered.

The impact and purpose of SQL injections can be broken down into four main categories. First, **confidentiality** is lost, database that hold sensitive information which may be financial or identifiable is compromised and can now be considered public knowledge. Secondly, in addition to the data being potentially leaked it now also has its **integrity** compromised as the malicious user can make any modification he wants to the data. **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 for passwords and bypassing any safe guards.

While SQL injections are commonly used to retrieve the information from a database, there are much more nefarious things that can be done with it. For example, previous wide-scale botnet attacks have used SQL injections mimicing as Google queries to facilitate malicious drive-by-download attacks on websites to spread malware.

2.2.1 SQL Injection Types

Often times many of these injection types are combined together and are not absolute, but the techniques can be classified under the six following categories.

Tautology attacks are designed to typically bypass authentication, identify injectable parameters or extract data. It is typically 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 the table and will often return all of them depending on how the application and the injection query is designed. For example the 1=1 portion of the following query will always evaluate to true:

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

A **malformed** or invalid query are often performed to gather information about the underlying database or applications structure to design more targetted queries. This is due to the common mistake of having overly descriptive errors that reveal information that is very useful to an attacker. This 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. This would often be used for extracting data as you can combine the data of another table which you only know a limited amount of information but is what you are interested in, with another table that is easily exploitable. 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 who have heard about when learning about SQL

injections because it involves ending the current query and starting 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 do something much worse such as escalating themselves in the database environment or denying service. Often times developers think that using embedded procedures makes their code protected from injections as all of the queries are 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

Recent Developments and Potential Solutions

4.1 Improving Signature Based Detection with Genetic Algorithms

As of recently, research in evolutionary algorithms for web threat detection has been on the rise, but the vast majority of this research is limited to the lower levels of the OSI model as well. A very recent study used a genetic algorithm in particular in order to detect SQL, XSS, and RFI attacks and showed that over 90% of attacks could be detected so these techniques definitely have some sort of application to the higher-level layers as well.

4.2 Alternative Solutions

4.2.1 Classification Tools (SVM)

4.3 Conclusions and Analysis on Potential Solutions

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 5.2). In order to gather SQL injection attacks the popular

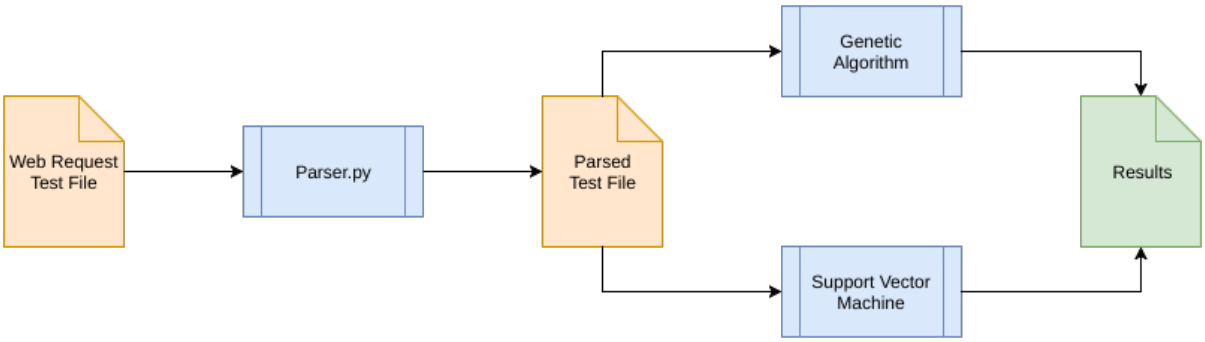

Figure 5.1: Overview of the system

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 1: 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 2). 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 2: 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 3).

```
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 3: 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).

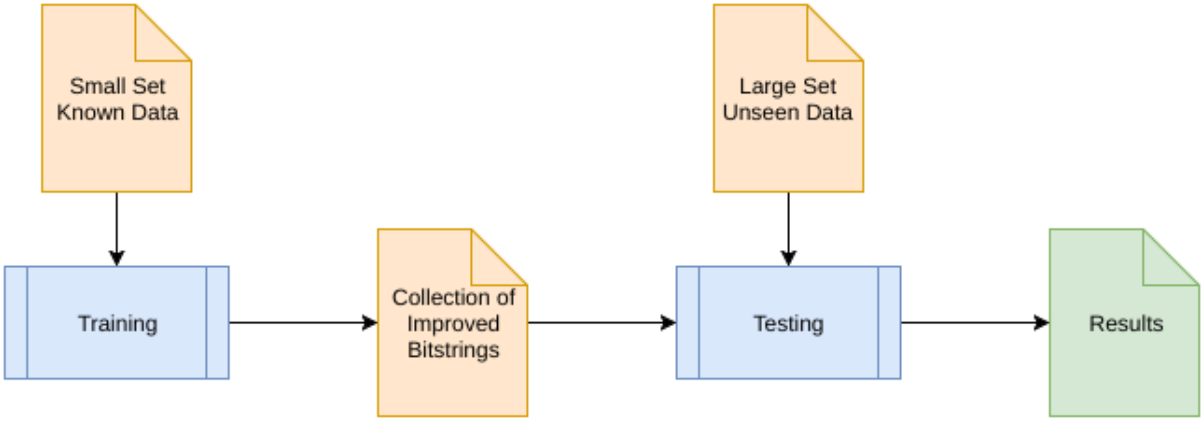

Figure 5.3: Overview of the genetic algorithm system

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). 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 4).

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 4: 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.

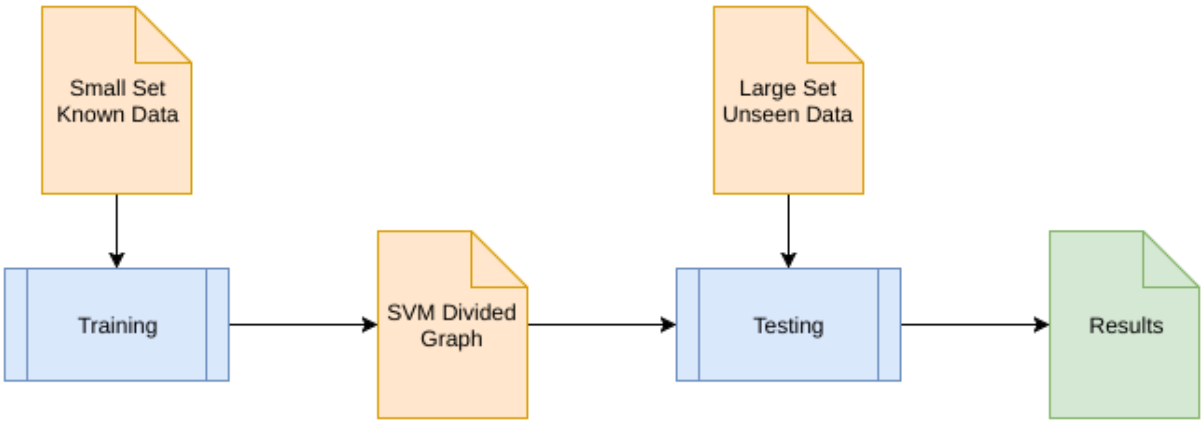

Figure 5.4: Overview of the support vector machine system

Chapter 6

Results

6.1 Genetic Algorithm

6.1.1 Best Parameters

6.1.2 Impact of Expanding Signature Set

6.1.3 Genetic Algorithm Bitstring Length Comparison

6.1.4 Genetic Algorithm vs Random Permutations

6.1.5 Genetic Algorithm vs Random Permutations with Fitness

6.2 Support Vector Machine

Chapter 7

Discussion

7.1 Disadvantages in Parsing

7.2 Genetic Algorithm

7.2.1 Advantages and Disadvantages

7.3 Support Vector Machine

7.3.1 Advantages and Disadvantages

Chapter 8

Conclusion

8.1 Conclusions

8.2 Future Work

Appendix A

Regular Expression Documentation

lorem

Appendix B

Remaining Genetic Algorithm Testing Results

B.1 1

lorem

B.2 2

lorem

Appendix C

Remaining SVM Testing Results

C.1 1

lorem

C.2 1

lorem

Bibliography

[1] I. Freely, "A small paper," *The journal of small papers*, vol. -1, 1997. to appear.