Chapter 9

COLLECTING NETWORK EVIDENCE USING CONSTRAINED APPROXIMATE SEARCH ALGORITHMS

Ambika Shrestha Chitrakar and Slobodan Petrovic

Abstract Intrusion detection systems are defensive tools that identify malicious activities in networks and hosts. In network forensics, investigators often study logs that store alerts generated by intrusion detection systems. This research focuses on Snort, a widely-used, open-source, misuse-based intrusion detection system that detects network intrusions based on a pre-defined set of attack signatures. When a security breach occurs, a forensic investigator typically starts by examining network log files. However, Snort cannot detect unknown attacks (i.e., zero-day attacks) even when they are similar to known attacks; as a result, an investigator may lose evidence in a criminal case.

This chapter demonstrates the ease with which it is possible to defeat the detection of malicious activity by Snort and the possibility of using constrained approximate search algorithms instead of the default Snort search algorithm to collect evidence. Experimental results of the performance of constrained approximate search algorithms demonstrate that they are capable of detecting previously unknown attack attempts that are similar to known attacks. While the algorithms generate additional false positives, the number of false positives can be reduced by the careful choice of constraint values in the algorithms.

Keywords: Network forensics, constrained approximate search, Snort

1. Introduction

Alerts generated by network intrusion detection systems are one of the important sources of evidence in network forensic investigations. The alerts indicate illegal network connection attempts and provide information about the source IP address, time of event and type of illegal attempt, among other information. Indeed, it is very likely that most

© IFIP International Federation for Information Processing 2018
Published by Springer Nature Switzerland AG 2018. All Rights Reserved

investigations begin by looking into this information. While alerts do not provide detailed information about malicious events, they can guide an investigator in the process of gathering evidence. Additionally, the log files that store intrusion detection system alerts can be presented as evidence in a court of law. Therefore, it is important to store as many alerts as possible and to reduce the number of false negatives (i.e., failures to raise alerts when suspicious activities take place).

Snort [7] is a widely-used, open-source, misuse-based system that detects intrusions based on a pre-defined set of attack signatures. The attack signatures are stored in its misuse database in the form of rule files, and exact searches are used to identify the signatures in real network traffic. Snort and many other intrusion detection systems engage Aho-Corasick search as their default algorithm [1]. This choice is motivated by the fact that the Aho-Corasick algorithm is fast enough for intrusion detection in networks with up to moderate bandwidth. In addition, it is easy to implement and is resistant to algorithmic attacks, where an attacker produces traffic that is difficult for search algorithms to detect efficiently.

Snort generates an alert when one of the known attack signatures is matched in incoming network traffic. Due to its use of exact search, Snort cannot detect unknown (i.e., zero-day) attacks even when they are similar to known attacks. Indeed, it could be enough for an attacker to change just a single bit in known attack traffic to evade Snort. In such an instance, a network forensic investigator could lose valuable evidence.

In order to detect attacks that are mutually similar and preserve the corresponding alerts in a log file, an attempt could be made to list all the variations of the known attack patterns and produce the corresponding signatures. However, this is impractical; in fact, it is impossible in the case of a zero-day attack. Another solution is to use regular expressions that enable input strings to be varied efficiently in an exact search algorithm. However, complex regular expressions are difficult to understand by human signature creators, which often results in erroneous signatures that produce large numbers of false positives and false negatives. Additionally, the interpretation of regular expressions by a machine is often very resource intensive. Moreover, it is impossible to create a regular expression that could be used to detect a zero-day attack.

A solution to these problems is to apply constrained approximate string matching instead of exact search. Approximate search [2, 13] allows some level of error tolerance in string comparisons. The errors are presented in the form of elementary edit operations such as insertions, deletions and substitutions. Constrained approximate search is based on *a priori* knowledge about possible edit operations that can be used in approximate matching. This knowledge can be obtained by reconnaissance and/or attacker profiling. A recent study [11] has demonstrated that constrained and unconstrained approximate search algorithms can detect zero-day suspicious activities. However, constrained approximate search algorithms produce fewer false positives and false negatives compared with unconstrained search algorithms.

This chapter demonstrates Snort evasion by an attacker, which results in the loss of evidence. Next, constrained approximate search algorithms, such as CRBP-OpType [12], CRBP-OpCount [11], modified CRBP-OpCount and CRBP-Indels (insertions and deletions taken together) [10], are employed to detect SQL injection attack signatures. These algorithms are implemented in a bit-parallel manner. The results obtained with the constrained approximate search algorithms are compared with those obtained with an unconstrained approximate search algorithm implemented using row-based bit-parallelism (RBP), which underlies the constrained RBP (CRBP) algorithms mentioned above. The experimental results demonstrate that the constrained and unconstrained approximate search algorithms can detect new attacks that are similar to known attacks up to a level of tolerance specified in advance. This facilitates the gathering of evidence related to network intrusions. However, trade-offs must be struck between the desired (small) numbers of false positives and the speed of the constrained approximate search algorithms to obtain quality results in reasonable time.

2. Evidence Detection Using Snort

Joshi and Pilli [4] have presented a generic process model for network forensics, which covers the preparation, detection, incident response, collection, preservation, examination, analysis, investigation and presentation processes. All these processes are connected and tools such as network intrusion detection systems play an important role in detecting attacks and collecting evidence. This section discusses the use of Snort as a tool for generating alerts associated with detected attacks.

Snort [7] is an open-source, network intrusion detection system that is supported by commercially-funded research and development efforts. Because of this, Snort is actively updated and its rules are freely available. Moreover, the rules may be customized or augmented as desired.

Snort compares known attack patterns against network traffic using the Aho-Corasick algorithm [1], a well-known, exact multi-pattern search algorithm. The known attack patterns are stored in the Snort misuse database in the form of rules. The rules define the type of traffic that is considered to be illegal in a monitored network. Snort generates alerts when it finds traffic that triggers one of its rules. The following example demonstrates how rules are triggered when Snort examines network packets or sessions.

Consider the following Snort rule:

```
alert tcp $EXTERNAL_NET any -> $HOME_NET $HTTP_PORTS (
   msg:"SQL 1 = 1 - possible sql injection attempt";
   flow:to_server,established; content:"1%3D1";
   fast_pattern:only; http_client_body;
   pcre:"/or\++1%3D1/Pi"; metadata:policy balanced-ips drop,
   policy security-ips drop, service http; reference:url,
   ferruh.mavituna.com/sql-injection-cheatsheet-oku/;
   classtype:web-application-attack; sid:30040; rev:2;)
```

This rule triggers on an SQL injection attempt. In the rule, all the fields outside the parentheses belong to the rule header and the parameters within the parentheses belong to the rule option (see the Snort manual [8] for more details).

In this particular case, the rule header tells Snort to trigger an alert on any TCP traffic coming from an external network $EXTERNAL_NET through any port to the destination/home network $HOME_NET and port number $HTTP_PORTS. $EXTERNAL_NET, $HOME_NET and $HTTP_PORTS are variables that can be defined in the Snort configuration file (`snort.conf`).

The rule option provides the exact requirements for traffic to generate the alert. The `flow:to_server` rule option guides Snort to perform a match on a proper TCP segment sent from the client to the server as a part of connection establishment. If this condition is met, the Snort examines the string `1%3D1` (or `1=1` because `%3D` is "=" in Unicode) in the payload of incoming traffic. The `fast_pattern` keyword tells Snort to further evaluate the rule only if the content is found in the payload. The `http_client_body` keyword restricts the search to the body of an HTTP client request. After meeting this condition, Snort examines the regular expression inside the `pcre` keyword (`or` followed by one or more whitespaces (\+ is an HTTP-encoded whitespace) that terminates with `1%3D1`). The text inside the regular expression is not case sensitive.

This rule is assigned a class type of priority 1 and it is a second revision with a unique Snort identification (SID) number of 30040. More information about the attack is available in the URL provided by the `reference` keyword.

The Snort rule triggers the following alert for an `OR 1=1` string in an HTTP request body:

```
[**][1:30040:2] SQL 1 = 1 - possible sql injection attempt[**]
[Classification:  Web Application Attack] [Priority:  1]
{TCP} 10.0.2.15:40020 -> 10.0.2.25:80
```

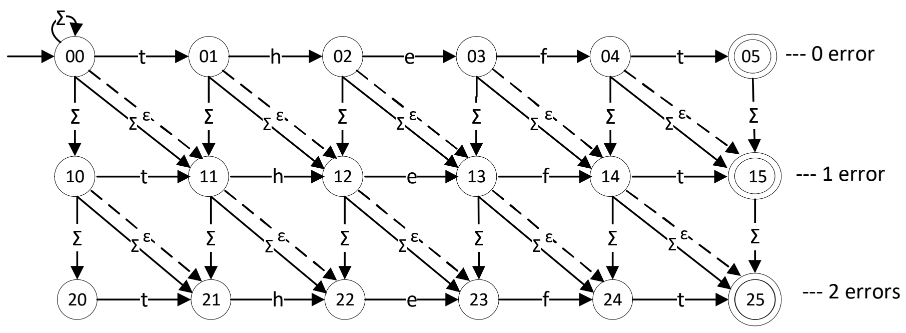


Figure 1. NFA for the search pattern “threat” permitting up to two errors [12].

The string `[1:30040:2]` in the first line of the alert has the structure `[GID:SID:Revision]` where `GID` is the generator ID that created the alert and `SID` and `Revision` are the same as in the rule above; this portion is followed by an alert message, which is provided by the message option of the rule. The second line provides information about the classification type and its priority. The third line shows `TCP` as the embedded protocol, which is followed by the source IP address:port number (`10.0.2.15:40020`) and the destination IP address:port number (`10.0.2.25:80`).

The alert text provides human readable information about the source, time of the event, type of attack attempt, etc., which could be used to gather additional evidence. Snort alerts can be presented as evidence in a court of law.

3. Row-Based Bit-Parallelism

Row-based bit-parallelism (RBP) [13] has been exploited by many constrained approximate search algorithms such as CRBP-Indels [10], CRBP-OpType [12] and CRBP-OpCount [11]. In this technique, a non-deterministic finite automaton (NFA) for the search pattern is simulated in a row-wise fashion for each search symbol in order to find a match. Bit-parallelism is an important technique that speeds up a search algorithm. It reduces the number of multiple operations by a factor of the number of bits in a computer word [3].

Figure 1 shows a non-deterministic finite automaton for the search pattern “threat” that permits up to two errors [12]. Each row in the non-deterministic finite automaton represents a match with number of errors equal to the row number (row number starts with zero). The node with the self-loop is the initial state and the double-circled nodes in each row are the final states. The numbers inside a node indicate the position of the node in the non-deterministic finite automaton: the first number indicates the row number and the second indicates the column number. Arrows represent transitions between states: a horizontal transition represents a character match, a vertical transition represents a character insertion, a dashed-diagonal transition represents a character deletion and a solid-diagonal transition represents a character substitution. The Σ symbols in the vertical and solid-diagonal transitions denote that the insertion and substitution characters are from the alphabet Σ. The ϵ symbol in a dashed-diagonal transition denotes the ϵ-transition, which does not consume an input character. In general, a match occurs at a certain search symbol of the search string when a final state in the non-deterministic finite automaton is reached.

In order to simulate the non-deterministic finite automaton using row-based bit-parallelism, a search algorithm engages three processes: (i) bit-mask generation; (ii) non-deterministic finite automaton initialization; and (iii) search.

A bit-mask is created for all the unique characters of a search pattern. In this process, the length of the bit sequence for each character is equal to the length of the search pattern and the bit positions where the character exists in the search pattern are set to one; the other bits are set to zero.

In the non-deterministic finite automaton initialization process, each state of the non-deterministic finite automaton is assigned an activity bit, where one indicates an active bit that can be reached with a successful transition and zero indicates an inactive bit. The process of assigning bits in the non-deterministic finite automaton differs for various search algorithms [11–13].

The third process, performing the search, computes an update formula to search for a given search symbol. The following update formula [13] determines a sequence of bits for each row of the non-deterministic finite automaton for the given search symbol:

$$R_0^{'} \leftarrow ((R_0 << 1)|0^{m-1}1)\&B[t_j] \tag{1}$$

$$R_i^{'} \leftarrow ((R_i << 1)\&B[t_j])|R_{i-1}|(R_{i-1} << 1)|(R_{i-1}^{'} << 1)|1 \tag{2}$$

The first part of the update formula (Equation (1)) computes the horizontal transition for row zero while the second part of the update formula (Equation (2)) computes the horizontal, vertical, solid-diagonal, and dashed-diagonal transitions, respectively, for all the rows greater than zero. In the update formula, m is the length of the search pattern, $B[t_j]$ is the bit-mask of the character at position j of search string t, i is the row number, R is a sequence of bits of the row for a search symbol at position $j - 1$ and R' is a sequence of bits of the row for the search string at position j. A match is said to occur at position j of the search string when the last bit of any row is active.

4. Constrained Approximate Search

Constrained approximate search applies *a priori* knowledge about edit operations and sets constraints on them while performing approximate matching. Example constraints include the maximum number of allowed indels, allowed types of edit operations and maximum number of allowed individual edit operations [12].

Sankoff and Kruskal [9] have specified a constrained edit distance calculation algorithm with constraints on the number of indels. In the algorithm, the indel distance value indicates the number of insertions, deletions or their combinations. The Sankoff-Indels algorithm [10] is a modified version of the constrained edit distance algorithm, which converts it to a constrained approximate search algorithm. The CRBP-Indels algorithm [10] is also a constrained approximate search algorithm that allows constraints on indels. This algorithm, which employs row-based bit-parallelism [13], uses counters in each state to control the use of indel operations. For example, if the number of indels is two, then the algorithm attempts to find matches by allowing maximum two insertions ($I = 2$), maximum two deletions ($E = 2$) or one insertion and one deletion ($I = 1$ and $E = 2$). The Sankoff-Indels and CRBP-Indels algorithms have been applied in spam filtering to detect spam words that were modified but are still intelligible to humans [10].

The constraints on the types of edit operations enable users to specify the edit operations to be used in an approximate search algorithm. Examples include allowing only substitutions (if possible) or only deletions and substitutions. The CRBP-OpType algorithm [12] is a constrained approximate search algorithm that allows a subset of insertions, deletions and substitutions to be applied as constraints during approximate search. The algorithm exploits bit-parallelism and is, in fact, based on the RBP unconstrained search algorithm [13].

An experiment was performed to detect similar attack patterns. The results demonstrate that the CRBP-OpType algorithm exhibits better performance in terms of speed compared with the RBP unconstrained search algorithm. The simulated NR-grep tool [5] also enables a subset of edit operations (including transpositions) to be used as constraints in approximate matching. NR-grep is based on the BNDM algorithm [6], which exploits bit-parallelism to simulate a suffix automaton.

The constraint on the number of individual edit operations is another type of constraint used in approximate search. The constraint enables a user to set the maximum number of allowed individual edit operations. An example is allowing two insertions, one substitution and one deletion to find the occurrences of a search pattern in a search string.

The CRBP-OpCount algorithm [11] is a constrained approximate search algorithm that allows such constraints. The algorithm, which is also based on the RBP unconstrained search algorithm [13], uses counters for all the active bits of the non-deterministic finite automaton to control the allowed number of individual edit operations. Before applying the algorithm, the attack signatures in the content field of the alerts in the Snort `backdoor.rules` file are extracted and converted to hexadecimal values. The hexadecimal values are considered to be search patterns and search strings are created by introducing some errors.

An experiment was performed to find approximate matches using the CRBP-OpCount and RBP unconstrained search algorithms. The experimental results indicate that the CRBP-OpCount algorithm can reduce the number of false positives compared with the RBP unconstrained search algorithm. However, due to the space complexity introduced by the counters and computations, the CRBP-OpCount algorithm is slower than the RBP unconstrained search algorithm [11].

5. Modified CRBP-OpCount Algorithm

This section provides details about the modified CRBP-OpCount algorithm, which improves on the CRBP-OpCount constrained approximate search algorithm [11].

The original CRBP-OpCount algorithm incorporates the same three processes as the RBP unconstrained search algorithm [13]: (i) bit-mask generation; (ii) non-deterministic finite automaton initialization; and (iii) search.

The bit-mask generation process is the same as in the RBP unconstrained search algorithm, but the initialization and search processes incorporate counters for all the active bits to control the allowed numbers of edit operations.

In the non-deterministic finite automaton initialization process, i consecutive bits from right to left are set to active for each row of the automaton, where $i = 0, 1, \ldots, k$. The active bits in each row are then assigned for all the possible counters by reducing each counter for row $i - 1$ by one on each edit operation.

The search process uses the same update formula as the RBP unconstrained search algorithm, but the transitions (for insertions, deletions

Algorithm 1: NFA initialization (modified CRBP-OpCount).

```
$R_0 \leftarrow 0^m$;  $R_0.D_0 \leftarrow con$
for $i$ = 1 to $k$ do
    if $i \leq con[E]$ then
        $R_i \leftarrow 0^{m-1}1^i$
    end
    else
        $R_i \leftarrow 0^m$
    end
    for all $j$ such that $R_i.D_j = 1$ do
        $R_i.C_j \leftarrow R_{i-1}.C_{j-1}[I, E-1, S]$
    end
    for all $R_i.D_0$ do
        $R_i.C_0 \leftarrow R_{i-1}.C_0[I-1, E, S]$, $R_{i-1}.C_0[I, E, S-1]$
        /* counter values should not be negative */
    end
end
```

and substitutions) are controlled by checking the corresponding counter values for the transitions from where the transitions must be initiated. A transition is successful and the bit to which the transition has to be directed is set to active when the counter value is greater than zero; otherwise, the bit is set to inactive. A match is found if the last bit of any row in the non-deterministic finite automaton can be reached by following the search process.

The modified CRBP-OpCount algorithm only differs from the original CRBP-OpCount algorithm in the initialization process of the non-deterministic finite automaton. The modified algorithm reduces the number of counters used during initialization compared with the original CRBP-OpCount algorithm. The original CRBP-OpCount algorithm uses all the possible counters for the active bits, although all of them are not needed in the search. By removing the unnecessary counters in the non-deterministic finite automaton initialization process, the modified CRBP-OpCount algorithm further reduces the unnecessary computations involving the counters during the search. As a consequence, the modified CRBP-OpCount algorithm saves storage while increasing the speed compared with the original CRBP-OpCount algorithm. In particular, the theoretical time complexity of the modified CRBP-OpCount algorithm is $O(knb)$ where k is the maximum number of allowed errors in the approximate search, n is the length of the search string and b is the counter computation for active bits in a row.

Algorithm 1 presents the pseudocode for non-deterministic finite automaton initialization in the modified CRBP-OpCount algorithm. Note

Table 1. Non-deterministic finite automaton initialization.

Row	CRBP-OpCount	Modified CRBP-OpCount
0	$R_0 = 0000000$ $R_0.C_0 = [0, 2, 1]$	$R_0 = 0000000$ $R_0.C_0 = [0, 2, 1]$
1	$R_1 = 0000001$ $R_1.C_0 = [0, 1, 1], [0, 2, 0]$	$R_1 = 0000001$ $R_1.C_0 = [0, 1, 1]$
2	$R_2 = 0000011$ $R_2.C_0 = [0, 0, 1], [0, 1, 0]$ $R_2.C_1 = [0, 0, 1], [0, 1, 0]$	$R_2 = 0000011$ $R_2.C_0 = [0, 0, 1]$ $R_2.C_1 = [0, 0, 1]$
3	$R_3 = 0000111$ $R_3.C_0 = [0, 0, 0]$ $R_3.C_1 = [0, 0, 0]$ $R_3.C_2 = [0, 0, 0]$	$R_3 = 0000000$ $R_3.C_0 = [0, 0, 0]$

that *con* denotes the constraints on edit operations, k is the maximum number of allowed errors (i.e. tolerance), m is the length of the search pattern, R is a row, D is a status bit (0 or 1), j is a bit position and C is a counter for the active bits.

During the non-deterministic finite automaton initialization process of the modified CRBP-OpCount algorithm, i consecutive bits are set to one only for the rows starting from zero to the number of allowed deletions; otherwise, all the bits in all the rows are set to zero. The process of assigning counters for the active bits is also different compared with the original CRBP-OpCount algorithm. In the case of the modified CRBP-OpCount algorithm, *con* is assigned in position zero of row zero ($i = 0$). For rows greater than zero and for every active bit j, the deletion counter is decremented by one from the counter value of bit $j - 1$ of row $i - 1$. Counters for the insertions and substitutions are computed only for the zero positions of the rows. A check is made whether or not a counter value is greater than zero in bit zero of row $i - 1$. If it is greater than zero, then the value is decremented by one and stored in the counter of position zero of row i. Moreover, substitution is not applied when a deletion has already been applied for an active bit.

Table 1 shows all the bits and their counters for each row of the non-deterministic finite automaton. Note that the goal is to find distorted occurrences of the search pattern "threats" in the search string "treet" by allowing the constraint $con[I, E, S] = [0, 2, 1]$. The results in Table 1 reveal that the number of counters used by the modified CRBP-OpCount algorithm is less than the number required by the original CRBP-OpCount algorithm to solve the same problem. Other processes in the modified CRBP-OpCount algorithm (i.e., bit-mask and search) are not presented because they are the same as in the original CRBP-OpCount algorithm [11].

6. Experimental Results

The experimental setup incorporated two virtual machines: (i) attacker machine; and (ii) victim machine. The host computer had a 2.7 GHz processor and 8 GB RAM. The attacker machine ran the Kali Linux operating system while the victim machine ran the Windows 7 operating system.

6.1 Evading Snort with a Buffer Overflow

In order to perform a buffer overflow attack, a vulnerable echo server was created on the victim machine. When executed, the echo server listened for incoming client connections and transmitted the same text received from the clients back to the clients. The echo server was made vulnerable by using function `vulnerable`, which was invoked before sending back the message transmitted by a client.

The `vulnerable` function is defined as follows:

```
int vulnerable_func(char *input){
    char buffer[16];
    strcpy(buffer, input);
    return 1;
}
```

The function copies the content of the `input` parameter into the `buffer` variable. The size of the `buffer` variable is fixed, but the `strcpy` function allows an input message of any size to be copied into the `buffer`. If the echo server receives input text longer than the defined size of the `buffer`, the program crashes.

The experiment sought to perform a buffer overflow attack and execute the `calc.exe` application on the victim machine. In order to accomplish this, it was necessary to find the `EBP` and `EIP` register contents so that the `EIP` return address could be replaced with a new address that starts the malicious code that invokes `calc.exe`. After some trials and debugging, it was discovered that the `EBP` register contained the `GGGG` value when the text `AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHH...` was sent. A new `ESP` address was then added manually and the text `AAAABBBBCCCCDDDDEEEEFFFFGGGG` was combined with the address in reverse order. When the client message `AAAABBBBCCCCDDDDEEEEFFFFGGGG [new ESP address][code to execute calc.exe]` was sent to the victim machine, the application crashed and `calc.exe` was executed.

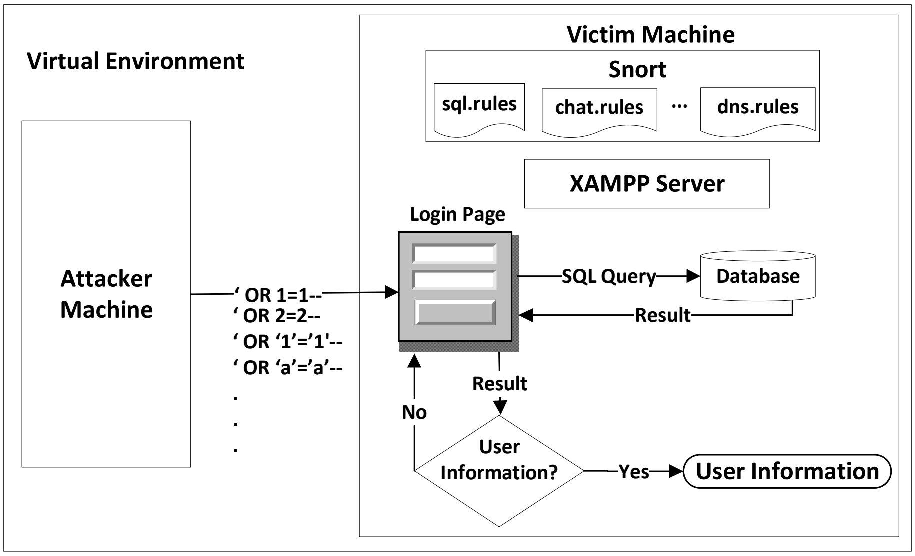


Figure 2. Using Snort to detect selected SQL injection patterns.

The following is the hexadecimal representation of the shellcode that executed `calc.exe`:

```
6681e4fcff31d2526863616c6389e65256648b72308b760c8b76
0cad8b308b7e188b5f3c8b5c1f788b741f2001fe8b4c1f2401f9
0fb72c5142ad813c0757696e4575f18b741f1c01fe033caeffd7cc
```

Snort did not have a rule corresponding to this buffer overflow attack. Therefore, it was unable to detect the attack and log it for subsequent analysis.

6.2 Evading Snort with SQL Injection

Figure 2 shows the experimental setup for performing SQL injection attacks from the attacker machine and using Snort on the victim machine to detect the injection attacks.

A vulnerable website was hosted on the victim machine using the XAMPP web server. The vulnerable website contained a login form for a user to enter login credentials. In the case of a valid login, the user was redirected to access his/her user information page; otherwise, the user was redirected to the login page with an error message. The login form on the vulnerable website allowed any characters in the username and password fields. However, the username field accepted a maximum of 16 characters. The PHP programming language was used to create the vulnerable website and MySQL to create the database.

The validation of user credentials on the login page of the vulnerable website was performed using the following SQL statement:

```
SELECT * FROM users WHERE uname='$uname' and pass='$pass'
```

This SQL statement fetches all the details from the user table in the database when the username (uname) and password (pass) field values match the username ($uname) and password ($pass) values provided in the user login form.

In the experiment, Snort had the default set of rules, which were downloaded from the website `snort.org`. Snort was always started when the victim machine started. The SQL statement used in the experiment to authenticate users could be modified by the attacker to inject a malicious SQL statement, this is referred to as a SQL injection attack.

Two SQL injection attack patterns were employed: (i) `OR 1=1`; and (ii) `'1'='1`. The Snort rule corresponding to the second pattern is:

```
alert tcp $EXTERNAL_NET any -> $HOME_NET $HTTP_PORTS (
   msg:"SQL 1 = 1 - possible sql injection attempt";
   flow:to_server,established; content:"%271%27%3D%271";
   fast_pattern:only; http_client_body; metadata:policy
   balanced-ips drop, policy security-ips drop, service http;
   reference:url, ferruh.mavituna.com/sql-injection-cheatsheet
   -oku/; classtype:web-application-attack; sid:30041; rev:2;)
```

The rule triggers an alert for a possible SQL injection attempt when the `'1'='1` attack pattern is found in the payload content of the network traffic.

In the experiment, the attacker entered the inputs `' OR 1=1--` and `' OR '1'='1'--` in the username field. These inputs modified the SQL statements as follows:

```
SELECT * FROM users WHERE uname='' OR 1=1-- and pass=''
SELECT * FROM users WHERE uname='' OR '1'='1'-- and pass=''
```

Since the username and password fields in the login page were not validated, the malicious SQL statements enabled the attacker to successfully log into the system. Specifically, the attacker was logged in with the user credentials that were located at the top of the user list in the database. In the two malicious SQL statements, the blank username was ORed with the valid arithmetic operations (`1=1` and `'1'='1'`, respectively) and everything that followed was commented with `--`. This demonstrates that the SQL statements were valid regardless of what the attacker entered in the password field.

Snort detected both the SQL injection attacks because the rules for detecting the `' OR 1=1--` and `' OR '1'='1'--` attack patterns were included in the Snort rule set. However, there are unlimited variations of these attack patterns. For example, `‘ OR 2=2--` and `‘ OR 9=9--` corresponding to the `‘ OR 1=1--` attack pattern; and `‘ OR ‘2’=‘2’--`, `‘ OR ‘a’=‘a’--` and `‘ OR ‘a1’=‘a1’--` corresponding to the `‘ OR ‘1’=‘1’--` attack pattern. Since Snort does not incorporate rules for these attack pattern variations, by default it was unable to detect SQL injection attempts corresponding to these attack patterns.

6.3 Detecting Similar Attack Patterns

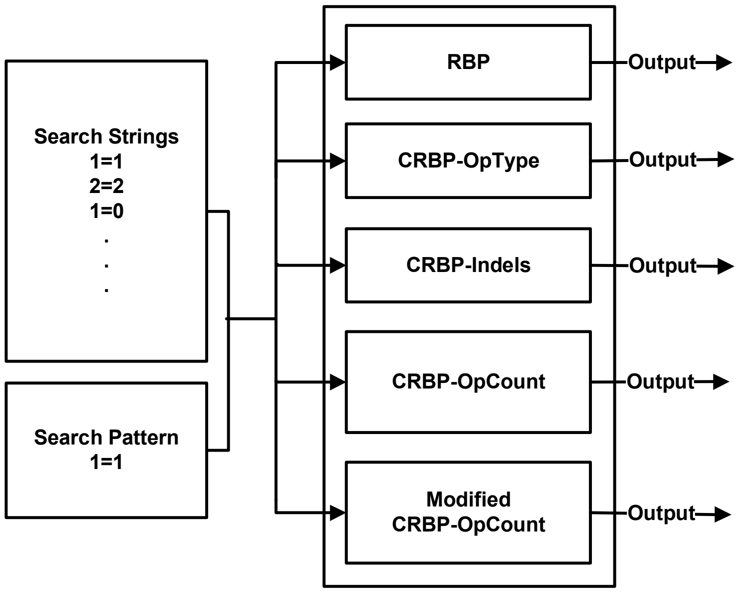
The next set of experiments applied four constrained approximate search algorithms (CRBP-OpType, CRBP-Indels, CRBP-OpCount and modified CRBP-OpCount) and one unconstrained approximate search algorithm based on row-based bit-parallelism to find randomly-created similar attack patterns such as `1=1` and `‘1’=‘1’`.

In order to perform the experiments, 200 similar variations of the attack pattern `1=1` and 11,280 variations of the attack pattern `‘1’=‘1’` were generated. Only ten strings among the 200 similar variations of the `1=1` search pattern could cause harm and only 80 strings of the 11,280 variations of the `‘1’=‘1’` search pattern could cause harm. In both cases, more strings were created that did not cause any harm but could be indications of attack attempts; the detection of such a string corresponds to a false positive. There could be fewer false positives in a real-world scenario, but the numbers were deliberately increased to evaluate the performance of the constrained approximate search algorithms.

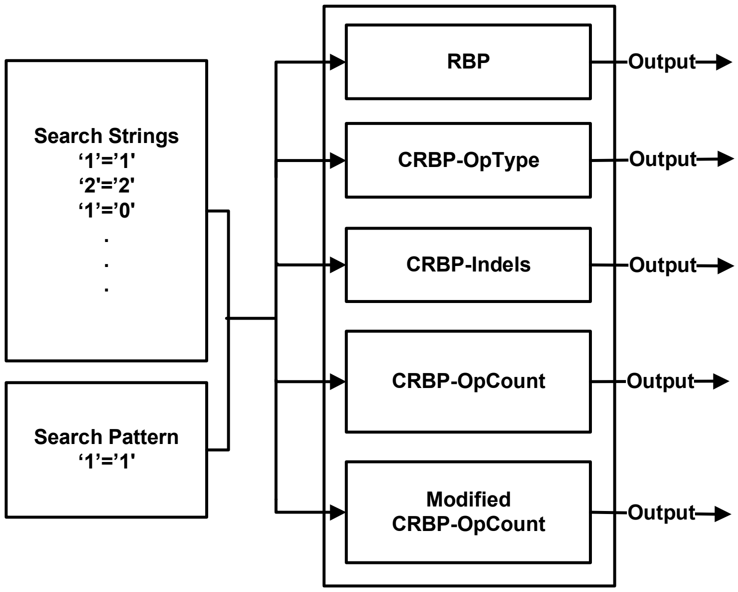
Figure 3 shows two simulations in which approximate search algorithms were applied to find the occurrences of the attack patterns `1=1` and `‘1’=‘1’` in the list of their randomly-generated similar variations (search strings). In both cases, all the approximate search algorithms were executed 30 times and the average time required by each algorithm was taken as its final execution time. The output also includes the performance of the algorithms based on the false positive results they generated.

Table 2 shows the false positive results generated by the algorithms for the two attack patterns. The true positive error (TP) corresponds to the number of attack patterns detected as harmful that were actually harmful. The false positive error (FP) is the number of attack patterns detected as harmful that were not actually harmful. The true negative error (TN) is the number of attack patterns that were not detected and that were not harmful.

The results in Table 2 show that, for the search pattern `1=1`, all the constrained approximate search algorithms produced the same num-



(a) Attack pattern 1=1.



(b) Attack pattern '1'='1'.

Figure 3. Applying approximate search algorithms with two SQL injection patterns.

Table 2. False positive results generated by the algorithms

Pattern	Strings	Algorithm	Tolerance	TP	FP	TN
1=1	200	RBP	k=2	10	181	9
		CRBP-OpType	k=2, S	10	159	31
		CRBP-Indels	k=2, indels=0, S	10	159	31
		CRBP-OpCount	k=2, S	10	159	31
'1'='1'	11,280	RBP	k=4	800	7,802	2,678
		CRBP-OpType	k=4, IS	800	7,636	2,844
		CRBP-Indels	k=4, indels=2, S	800	7,636	2,844
		CRBP-OpCount	k=4, I=2, S=2	800	5,928	4,552
		Modified CRBP-OpCount	k=4, I=2, S=2	800	5,928	4,552

ber of false positives when their tolerance values were set carefully. Note that S denotes substitutions and I denotes insertions. The RBP unconstrained search algorithm (RBP), on the other hand, generated more false positives than the constrained approximate search algorithms with same tolerance k. For the search pattern `'1'='1'`, the CRBP-OpCount and modified CRBP-OpCount constrained approximate search algorithms generated the smallest number of false positives whereas the RBP unconstrained search algorithm (RBP) generated the largest number of false positives.

Figure 4 shows the CPU times (in milliseconds) required by the approximate search algorithms. The left-hand side shows the execution speed for the pattern `1=1` whereas the right-hand side shows the execution speed for the pattern `'1'='1'`. The results reveal that the CRBP-OpType algorithm is faster than all the other algorithms evaluated in the experiments. In fact, the CRBP-OpType algorithm is also faster than the RBP unconstrained approximate search algorithm. The CRBP-OpCount algorithm appears to be the slowest of the approximate search algorithms. The modified CRBP-OpCount algorithm also outperformed the CRBP-OpCount algorithm.

7. Discussion

Experimentation has shown that it is very easy to perform a zero-day attack and evade Snort. Since Snort alerts constitute important evidence in network forensic investigations, it is desirable to collect as many relevant Snort alerts as possible. The experimental results reveal that approximate search algorithms (constrained and unconstrained) can be applied by Snort to collect more network forensic evidence by detecting attack signatures that are similar to those of known attacks. However,

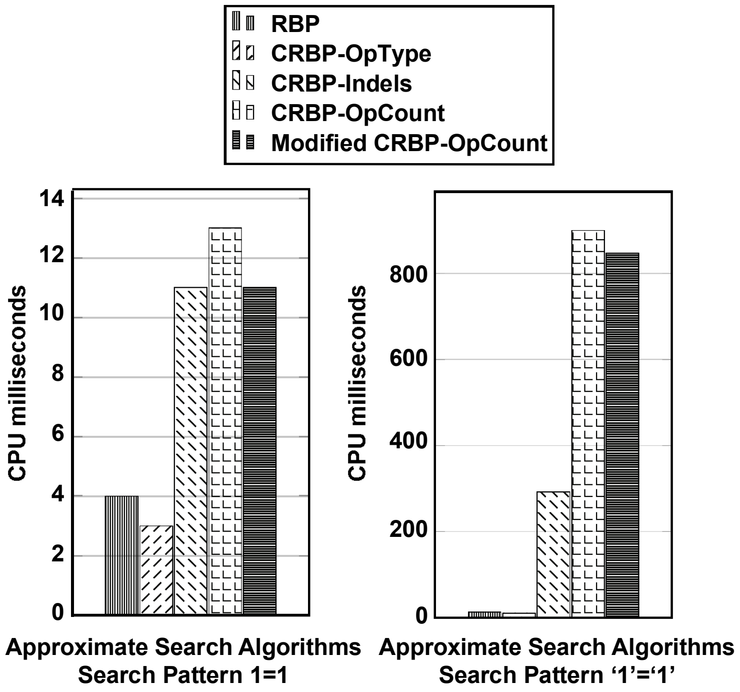


Figure 4. Speed comparison of the approximate search algorithms.

the results indicate that many false positives can be generated and the search speed can suffer if the wrong type of algorithm is selected with improper constraint parameters.

Comparisons of the speeds of the approximate search algorithms considered in this research (RBP unconstrained search, CRBP-OpType, CRBP-Indels, CRBP-OpCount and modified CRBP-OpCount) reveal that the CRBP-OpType algorithm exhibits the best performance while the CRBP-OpCount algorithm yields the worst performance. However, the overall performance of each algorithm also depends on the number of false positives it generates.

It is easy to use the RBP unconstrained search algorithm because all that a user needs to consider is the maximum number of allowed errors k. In the case of the constrained approximate search algorithms, it is recommended to acquire and engage *a priori* knowledge about the search problem. Constrained approximate search algorithms vary according to the types of constraints they can handle and their speeds also depend on the constraints. For example, the CRBP-OpType algorithm can consider the subset of edit operations to be used, but is unable to take into account the maximum number of allowed errors for individual edit operations as in the case of the CRBP-OpCount algorithm.

Sometimes, the same results can be obtained using different constrained approximate search algorithms. In such a situation, a user should chose the fastest algorithm. For example, if $C[I, E, S] = [0, 0, 2]$ in CRBP-OpCount and $k = 2$, S in CRBP-OpType, then both the algorithms will attempt to find search patterns that allow a maximum of two substitutions. Since CRBP-OpType is faster than CRBP-OpCount, it is wise to select CRBP-OpType as the search algorithm.

When using Snort, the selection of the constrained approximate search algorithm can be made based on the Snort rules. For example, the CRBP-OpType constrained approximate search algorithm was shown to be good enough to detect attack patterns similar to `'1'='1'`. Other constrained approximate search algorithms applied to the same problem yielded the same number of false positives as CRBP-OpType, but with lower performance.

It is wise to select a constrained approximate search algorithm that is slightly slower than the other algorithms, but will reduce the number of false positives. For example, in the case of the `'1'='1'` search pattern, the CRBP-OpType algorithm is the best in terms of its speed. However, it generates many false positives compared with the CRBP-OpCount and modified CRBP-OpCount algorithms. Since the CRBP-OpCount and modified CRBP-OpCount constrained approximate search algorithms differ only in their speed, it is better to select the modified CRBP-OpCount algorithm over CRBP-OpCount for search problems for which both the algorithms are suitable.

8. Conclusions

Evidence from a network intrusion detection system such as Snort is very valuable in a network forensic investigation. However, Snort cannot detect zero-day attacks even when they are similar to known attacks; as a result, Snort does not record evidence pertaining to these attacks. This research has demonstrated that the problem can be addressed by incorporating a constrained approximate search algorithm in Snort. Constrained approximate search algorithms, which are more powerful than exact search, enable users to set constraints on edit operations and control searches based on their requirements. The experimental results reveal that all the approximate search algorithms (constrained and unconstrained) can detect zero-day attacks that are similar to known attacks. However, they differ in their speed and in the number of false positives they generate.

References

[1] A. Aho and M. Corasick, Efficient string matching: An aid to bibliographic search, *Communications of the ACM*, vol. 18(6), pp. 333–340, 1975.

[2] R. Baeza-Yates and G. Navarro, Faster approximate string matching, *Algorithmica*, vol. 23(2), pp. 127–158, 1999.

[3] S. Faro and T. Lecroq, Twenty years of bit-parallelism in string matching, in *Festschrift for Borivoj Melichar*, J. Holub, B. Watson and J. Zdarek (Eds.), Prague Stringology Club, Prague, Czech Republic, pp. 72–101, 2012.

[4] R. Joshi and E. Pilli, *Fundamentals of Network Forensics: A Research Perspective*, Springer-Verlag, London, United Kingdom, 2016.

[5] G. Navarro, NR-grep: A fast and flexible pattern-matching tool, *Software – Practice and Experience*, vol. 31(13), pp. 1265–1312, 2001.

[6] G. Navarro and M. Raffinot, A bit-parallel approach to suffix automata: Fast extended string matching, *Proceedings of the Annual Symposium on Combinatorial Pattern Matching*, pp. 14–33, 1998.

[7] M. Roesch, Snort – Lightweight intrusion detection for networks, *Proceedings of the Thirteenth USENIX Conference on System Administration*, pp. 229–238, 1999.

[8] M. Roesch and C. Green, Snort Users Manual 2.9.9, The Snort Project (`manual-snort-org.s3-website-us-east-1.amazonaws.com`), 2017.

[9] D. Sankoff and J. Kruskal, *Time Warps, String Edits and Macromolecules: The Theory and Practice of Sequence Comparison*, Addison Wesley, Reading, Massachusetts, 1983.

[10] A. Shrestha Chitrakar and S. Petrovic, Approximate search with constraints on indels with application in spam filtering, *Proceedings of the Norwegian Information Security Conference*, pp. 22–33, 2015.

[11] A. Shrestha Chitrakar and S. Petrovic, Constrained row-based bit-parallel search in intrusion detection, *Proceedings of the Norwegian Information Security Conference*, pp. 68–79, 2016.

[12] A. Shrestha Chitrakar and S. Petrovic, CRBP-OpType: A constrained approximate search algorithm for detecting similar attack patterns, in *Computer Security*, S. Katsikas, F. Cuppens, N. Cuppens, C. Lambrinoudakis, C. Kalloniatis, J. Mylopoulos, A. Anton and S. Gritzalis (Eds.), Springer, Cham, Switzerland, pp. 163–176, 2018.

[13] S. Wu and U. Manber, Fast text searching: Allowing errors, *Communications of the ACM*, vol. 35(10), pp. 83–91, 1992.