Predictive Analytics to SQL Injection Attack Detection and Prevention

E. Saraswathi, Krishna Kant, Harshit Ranjan, Ankur Kumar, Abhinav Anand

Abstract*: *In Internet age, the most basic security danger of vulnerable web applications is SQLi attack. SQLi attack is very serious attack in the web application that allow attackers to use unrestricted use of data access which contain precise information of data in database. SQL is basically a correspondence medium between Web application and back end database. In this way, for the most part attackers use SQL for getting to a database. We show here a machine learning driven technique to find holes in firewall where SQLiA bypass. We will also show more and more learning approach in the tests that are passed or blocked by the firewall that show features with bypassing the firewall & change them efficient to do new bypassing attacks. The result which we get here is an excellent performance and efficiency in detection and prevention in SQL injection attack.

Keywords : SQLi, RAN, WAF, SQLiA, SOAP, CFG

I. INTRODUCTION

SQL injection is the most serious issue in the web application. In web applications, an attacker can easily access the of hidden database with the help of SQLiA. These types of databases regularly contain precise consumer data or client data, the subsequent security infringement can incorporate wholesale fraud in market, loss of private client data and extortion. Once in a while, aggressors also use SQL injection to obtain accountability for and degenerate the structure of the web application. Web applications which is accessible against SQLiA are boundless—an investigation by Group of Gartner more than 300 Internet web destinations that has appeared more majority of them could be vulnerable against SQL Injection Attacks.

The reason for SQLi accessible is straightforward and knew surely: incomplete approval of client's input.

Revised Manuscript Received on April 08, 2019.

E. Saraswathi, Professor, Department of Computer Science and Engineering, SRM Institute of Science and Technology, Ramapuram, Chennai, India

Krishna Kant, Student, Department of Computer Science and Engineering, SRM Institute of Science and Technology, Ramapuram, Chennai, India

Harshit Ranjan, Student, Department of Computer Science and Engineering, SRM Institute of Science and Technology, Ramapuram, Chennai, India

Ankur Kumar, Student, Department of Computer Science and Engineering, SRM Institute of Science and Technology, Ramapuram, Chennai, India

Abhinav Anand, Student, Department of Computer Science and Engineering, SRM Institute of Science and Technology, Ramapuram, Chennai, India

In order to address this type of issue, developers have planned a new set of coding rules that maximize guarded coding guidelines. For example, encoding the client information & approval.

A thoroughly & deliberate uses of these method is a successful answer for avoiding SQLi accessible. In any case, the utilization of such methods is mostly human-based. Moreover, fixing these legacy type of code bases that may be contain SQLi accessible can be work to escalated assignment.

In the past, there has been so many attentions of the problem of SQLi accessible, so many proposed arrangements neglect to address the full extent of the issue. Researchers and professionals are consistently ignorant heap of other methods which used in SQL injection attacks. In this way, a large portion of the arrangement of proposed recognize or avoid just a subset of the possible SQL Injection Attacks. In order to show this issue, we present a review of SQLIAs known to date. So, in order to complete the review, we take the data from different sources like web sites, mail records & specialists in region. For each and every type of attack that we considered a property of attack, that show its issue and how it could be performed. These types of attack are then used to check with the help of the detection and prevention methods.

II. BACKGROUND

SQL Injection Vulnerabilities. In framework the SQL statement are used to back end database which is basically used by native application coding as string. So, these strings are basically shaped by linking different section of string which is based on client decisions or the application control stream. After this, all done a SQL explanation is framed to utilized then SQL statement send to the database for execution. Here in below, we can see a SQL statement as follow:

```
$sql = "select * from hotelList where country ='";
$sql = $sql . $country; $sql = $sql . "'";
$result = mysql_query($sql) or die(mysql_error());
```

The *"$country"* variable is taken by user as input in the statement of SQL and taken to the string "*$sql"* variable as shown in above. After then it is moved to the "*mysql_query"* function which basically send the statement of SQL to server database for execution.

In SQL injection the attackers attack by inserting some malicious code of SQL as input values so that it failed to validate proper. Also, attacker may develop some new values as input that resulting SQL statement performs abnormally behave action in the database like loss of important data or data leaks etc.

In the above example suppose the attackers uses in the input "$*country"* is *or 1=1* --, then answer of such SQL statement is:

select * from hotelList where country=" or 1=1 --'

The symbol *or 1=1–*' is a condition which makes the tautology always true and with the help of this an attacker will easily bypass the initial situation by the *where* clause and finally it will return all *rows* data of the table in SQL query.

Web Application Firewalls. Coming to WAF, as we all know that WAF actually filters, monitors or block the data packets from a web application. It may be network based, host based or cloud based depending on the requirement. So here basically in the WAF firewall's set of rules is defined which is called a black-list where the regular expression is present in the string pattern. An attacker is generally targeting those regular expression and make some change set for malicious attacks i.e., SQL injection so

it blocked.

e.g., In the below regular expression we use SQL comments i.e., *# or /**/* which is generally mostly used by the SQLi attackers:

```
/\*!?|\*/|[';]--|--[\s\r\n\v\f]|(?:--[^-]*?-) |
([^\-&])#.*?[\s\r\n\v\f]|;?\\x00
```

III. APPROACH

In this paper, we are going to introduced three types of approach for the prevention and detection of SQLi namely as context free-grammar *(CFG)*, random-attack generation & then machine-learning driven approach. Here in ML-Driven approach we added a new approach in the existing one *i.e., ML Driven E.* Now let's talk each approach in detail one by one.

A. CFG for SQL injection attacks:

In this attack we focus three things for the SQLi attacks i.e., UNION, BOOLEAN & PIGGY that basically manipulate the logic of statements in the original queries of SQL. We actually show a grammar that handle three different things for the SQLi attacks and it can be inserted into "*dQuoteContext", "numericContext"* and *"sQuoteContext"*. Here the first and third target on the SQL statement which is basically the string data fields and the second one targets on the number data field like weight, money etc. An expert of grammar is showing as follow in which <ATTACK> is first *starting* symbol, "::=" is the production symbol, "," is the concatenation symbol, and "|" represents alternatives symbol.

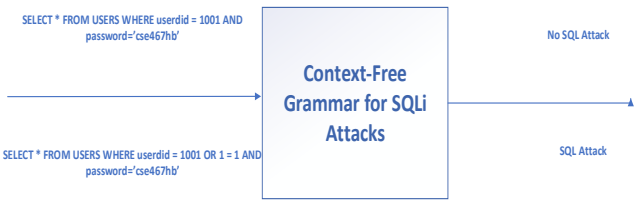

Fig. 1. Context free-grammar (CFG) for SQL injection attacks

⟨*ATTACK*⟩ ::= ⟨*numericContext*⟩ | ⟨*sQuoteContext*⟩
| ⟨*dQuoteContext*⟩ ;

⟨*numericContext*⟩ ::= ⟨*digitZero*⟩, ⟨*wsp*⟩, ⟨*booleanAttack*⟩, ⟨*wsp*⟩
| ⟨*digitZero*⟩, ⟨*parC*⟩, ⟨*wsp*⟩, ⟨*booleanAttack*⟩, ⟨*wsp*⟩, ⟨*opOr*⟩, ⟨*parO*⟩, ⟨*digitZero*⟩
| ⟨*digitZero*⟩, [⟨*parC*⟩], ⟨*wsp*⟩, ⟨*sqliAttack*⟩, ⟨*cmt*⟩ ;

⟨*sQuoteContext*⟩ ::= ⟨*squote*⟩, ⟨*wsp*⟩, ⟨*booleanAttack*⟩, ⟨*wsp*⟩, ⟨*opOr*⟩, ⟨*squote*⟩
| ⟨*squote*⟩, ⟨*parC*⟩, ⟨*wsp*⟩, ⟨*booleanAttack*⟩, ⟨*wsp*⟩, ⟨*opOr*⟩, ⟨*parO*⟩, ⟨*squote*⟩
| ⟨*squote*⟩, [⟨*parC*⟩], ⟨*wsp*⟩, ⟨*sqliAttack*⟩, ⟨*cmt*⟩ ;

⟨*dQuoteContext*⟩ ::= ⟨*dquote*⟩, ⟨*wsp*⟩, ⟨*booleanAttack*⟩, ⟨*wsp*⟩, ⟨*opOr*⟩, ⟨*dquote*⟩
| ⟨*dquote*⟩, ⟨*parC*⟩, ⟨*wsp*⟩, ⟨*booleanAttack*⟩, ⟨*wsp*⟩, ⟨*opOr*⟩, ⟨*parO*⟩, ⟨*dquote*⟩
| ⟨*dquote*⟩, [⟨*parC*⟩], ⟨*wsp*⟩, ⟨*sqliAttack*⟩, ⟨*cmt*⟩ ;

⟨*sqliAttack*⟩ ::= ⟨*unionAttack*⟩ | ⟨*piggyAttack*⟩ | ⟨*booleanAttack*⟩ ;

⟨*unionAttack*⟩ ::= ... ; ⟨*piggyAttack*⟩ ::= ... ; ⟨*booleanAttack*⟩ ::= ⟨*orAttack*⟩ | ⟨*andAttack*⟩ ;

⟨*opNot*⟩ ::= ! | not ; ⟨*opBinInvert*⟩ ::= ~ ; ⟨*opEqual*⟩ ::= = ; ⟨*opLt*⟩ ::= < ; ⟨*opGt*⟩ ::= > ; ⟨*opLike*⟩ ::= like ; ⟨*opIs*⟩ ::= is ;

⟨*opMinus*⟩ ::= - ; ⟨*opOr*⟩ ::= or | || ; ⟨*opAnd*⟩ ::= and | && ; ⟨*opSel*⟩ ::= select ⟨*opUni*⟩ ::= union ; ⟨*opSem*⟩ ::= ; ;

⟨*cmt*⟩ ::= # | ⟨*ddash*⟩, ⟨*blank*⟩; ⟨*ddash*⟩ ::= --

⟨*inlineCmt*⟩ ::= /**/ ; ⟨*blank*⟩ ::= ␣ ; ⟨*wsp*⟩ ::= ⟨*blank*⟩ | ⟨*inlineCmt*⟩

B. RAN as Random Attack Generation

Here RAN itself suggest its meaning by random attack in the proposed grammar and its generation is quite simple and easy one. Initially, it's starts from the start symbol and select the randomly production rule until terminals are left out. In this there is no any loop in grammar so due to this the procedure will always terminate & due to this the output of the SQL injection attack produce by adding every terminal symbol which is shown in the generated grammar.

So, the method which we are implemented in paper is basically the simple plan for the input to the grammar attack. Also keep in mind that by using again and again RAN multiple times up to maximus number of attacks or the test is reached.

C. ML-Driven Generation

As we see that the attackers always bypassing the attacks and it becoming hard to find the all the SQL attacks. If we are talking about *Random Attack Generation* methodology will progressively wind up wasteful.

In such circumstances, a further developed advanced age system that spends progressively computational exertion to distinguish experiments with a highest bypassing likelihood is relied upon increasingly productive. So, we present a ML *(Machine-Learning based)* methodology, called as *ML-Driven*, to test information in best and more effective way than *Random Attack Generation* does.

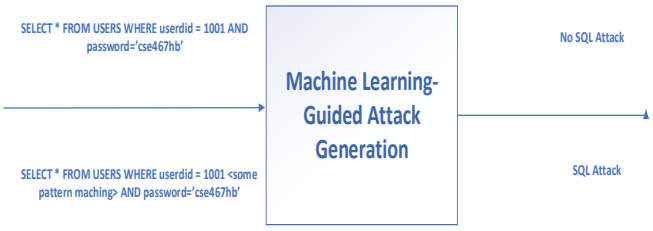

Fig. 2. ML-Driven Generation

Published By:
Blue Eyes Intelligence Engineering
& Sciences Publication

This methodology, called *Machine-Learning Driven*, is motivated by hereditary search-based test & programming-based test generation. So here in the *Machine-Learning Driven* model we see that how the components of test are related with the high probabilities bypass the web application firewall.

Actually, this method employs first in the RAN to provide some starting tests which sent to web application, depending upon either blocked or bypass the web application firewall, they named as "B" or "P". Now we will extract and use all tests as first training set of data to learn a model of predicting the similarity or likelihood (f) so the test can be bypassing the web application firewall.

With this, we select & rank the test with high-values of *"f"* to produce the new tests, such tests are run after completed and the result of "B" or "P" are actually help to improve the model of prediction which eventually help to bypass the web application firewall.

Now, we are going to show in detail of about the test which is encoded and integrated for machine learning and make new procedure and use to produce new *SQLi attacks.*

1) Test of Decomposition:

We are able to derive assessments from grammar through applying repeatedly by its producing policies. Any such process is represented as parse tree. A parse tree takes a look at a graphical image of the derivation steps which can be concerned in generating the test. In a parse tree, an intermediate node gives a leave node represents a terminal one, a non-terminal symbol and edges. Figure 2 shows a parse-tree of the BOOLEAN test : OR "a"="a"#. To start the test, we initially apply <*ATTACK*> rule.

$$(ATTACK) ::= (numericContext) \ / \ (sQuoteContext) \ / \ (dQuoteContext) ;$$

From the below fig.3, we can see that after the test started we will move to <sQuoteContext> then we will use rule of grammar for <*wsp*>, <*sqliAttack*>, <*squote*>, and <*comment*>. This will use up to when all the node become with terminal symbols.

Now we determine trees for distinguish of SQLi attack in charge of attack block or pass. For every test we decompose inference tress into slices as follow:

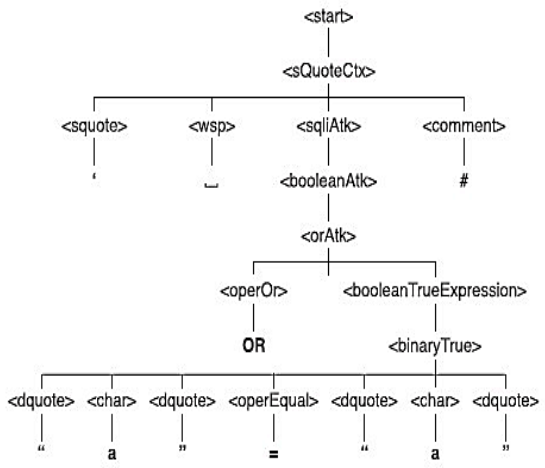

Fig. 3. The parse tree of SQLiA: ' OR"a"="a"#.

Explanation 1 (Slice). In this Slice s that is., s1, s2, s3,s4 of a parse tree T is a subtree where root is basically non-terminal node 'T', except that show "*<ATTACK>", "dQuoteContext", "numericContext"* & *"sQuoteContext"*.

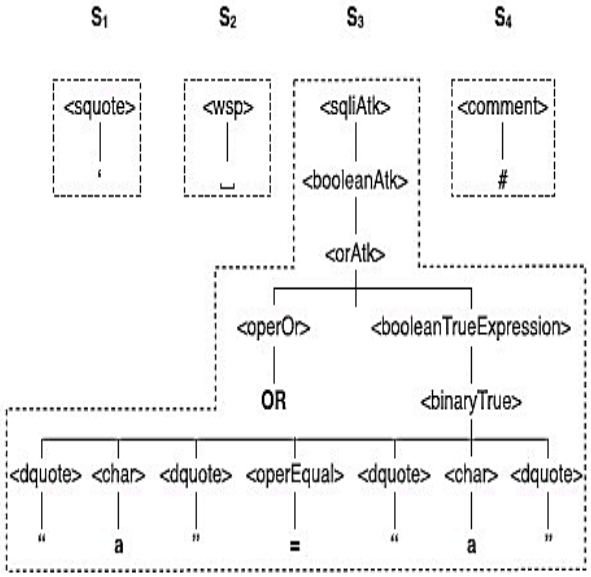

Fig. 4. Test of Decomposition slices from the tree in Fig. 3.

Explanation 2 (Minimum Slice). In the case of minimum slice s, it contains two nodes only i.e., a root, and a child in terminal symbol.

We conclude here display of either one or more slices in test that result in bypassing or getting blocked in the test. After that, we will expand a new idea with the help of machine learning and also show that how the test affects the bypassing a web application firewall (WAF) or block.

Algorithm 1 Test decomposition into slices.

```
1: procedure DECOMPOSE(inputTree)
2:     S ← ∅
3:     context ← inputTree.child                ▷ the only child
4:     for all child ∈ context do
5:         VISIT(child, S)
6:     end for
7:     return S
8: end procedure
9: procedure VISIT(node, S)
10:    s ← sliceFrom(node)                      ▷ get a sub-tree from node
11:    if s ∉ S then
12:        S ← s
13:    end if
14:    if s is minimal then
15:        return
16:    else
17:        for all child ∈ s do
18:            VISIT(child, S)
19:        end for
20:    end if
21: end procedure
```


Published By:
Blue Eyes Intelligence Engineering
& Sciences Publication

***2)* Training Set of ML for encoding**:

In the training set of ML for encoding, initially we take a set of tests that execute against web application firewall i.e., *"B" or "P"* label. We change every test to run machine-learning algorithm.

a) In every test initially, It's break down into slices ti = s1, s2,s3,s4,s5…… sNi using decomposition test method.

b) Every slice is allowed of unique-identifier. If the slice matches with multiple tests of slice then it's called as same identifier. We design every unique slice for machine-learning to the output data.

c) Each test is changed and checked that the slices is present or not as attributes.

From below table we have taken three id of test as t1, t2, t3; where the first and second one is blocked and the last one is actually bypassing a web application firewall. Decomposition which is showing below on left-side of table & encoded is showing in the right-side of table. If a slice in the test appears then the value in attribute training set become *"1"* if not then become *"0"*.

TABLE I
EXAMPLE OF TEST DECOMPOSITIONS AND THEIR ENCODING

t.id	vector	label
1	$\langle s_1, s_2, s_3 \rangle$	B
2	$\langle s_1, s_2, s_4 \rangle$	B
3	$\langle s_4, s_5 \rangle$	P

t.id	s_1	s_2	s_3	s_4	s_5	clz
1	1	1	1	0	0	B
2	1	1	0	1	0	B
3	0	0	0	1	1	P

***3)* Driven Test of Machine Learning:**

By test of decomposition in slices and changing the conclusion data, now we apply this machine-learning method that help to conclude which slice and what content are join among test bypassing a web application firewall.

We actually want that the number of tests which is blocked much greater than those a web application firewall in practice. This means, number of conclusions among *"B"* label larger than *"P"* label. So, our purpose is to deal the unbalance data with the help of algorithm of machine learning.

Here we use Random Tree Classifier that is achieve in Weka suite. In the Weka suite we found that Random Tree Classifier handle better in terms of unbalance data and its more define and precise. From the below algorithm we can easily exploited our method.

Algorithm 2 ML-Driven SQLi attack generation.

```
procedure MLDRIVENGEN(initTests, outputTests)
    execute(initTests)
    trainData ← transform(initTests)
    DT ← learnClassifier(trainData)        ▷ learn the initial classifier
    currTests ← initTests
    while not-done do
        rankTests(currTests, DT)
        repeat
            t ← selectATest(currTests)
            V ← getSliceVector(t)
            pathCondition ← getPath(V, DT)
            s ← pickASliceFrom(V)
            while s ≠ null do
                if satisfy(s, pathCondition) then
                    newTests ← mutate(t, s, MAX_M)
                    currTests ← currTests ∪ newTests
                end if
                s ← pickASliceFrom(V)
            end while
        until shouldUpdClassifier(currTests)
        execute(currTests)                 ▷ new tests only
        trainData ← transform(currTests)
        DT ← learnClassifier(trainData)
    end while
    outputTests ← filterByPassTests(currentTests)
    return outputTests
end procedure
```

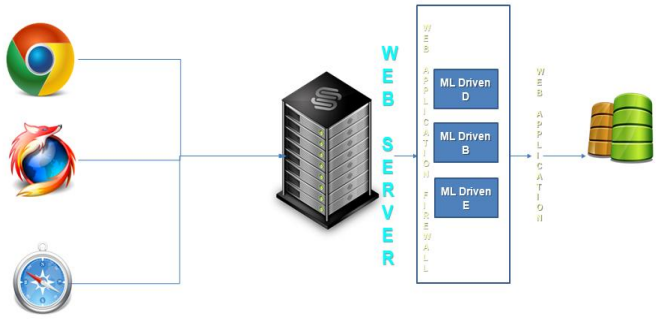

Fig. 5. Proposed Application

IV. EXPERIMENTS AND RESULT

Here, we execute two sets of method to find (i) growth of ML classifier that is we do some development in classifier by different training, (ii) structure of ML algorithm that give outstanding results, & (iii) the global achievement of the expected machine learning driven test in capability & performance.

A. Subject Applications

In this observation we choose the *ModSecurity* in current version of OWASP as target web application firewall. *ModSecurity* is basically an open source net software firewall that open with HTTP Apache server for the protection of web application. Relying at the programs under-protection, distinct firewall set of rule described for distinct purposes is used. It is basically used to defence from different attacks like SQLiA, Trojan & Denial of Service.

As we all know that the web application is protected from HotelRS, SugarCRM & Cyclos where HotelRS is a service oriented architecture that provide the room reservation. SugarCRM is actually customer relationship management system and coming to the Cyclos is open source of java web application which mostly used by online site like ecommerce. In our analysis three of the applications are set up on HTTP Apache server. *ModSecurity* in the web server that protect from SQLiA in the web application. In this journal, our target is to prevention & detection of SQLi, not the application so here SugarCRM, HotelRS, & Cyclos play an important role for a destination for SQL injection test that bypass the web application firewall.

B. Procedure

We introduce the machine learning driven technique along with RAN and combine them into SQL injection tools named as Xavier. It is automatic testing of web application service for *SQLi* accessible and define it. ML-Driven or RAN can bring out tests in the form of SQL injection strings attack i.e., OR"1"="1"#. Xavier holds such types of tests and add in form of "*SOAP"* message, turn input as malicious & finally send them to an application under test.

SOAP first requests to server in web application firewall then forward to web application, if not then it's blocked. When the tool of testing obtained by, Xavier as feedback, it points test with blocked label "B", or else "P".

Published By:
Blue Eyes Intelligence Engineering
& Sciences Publication

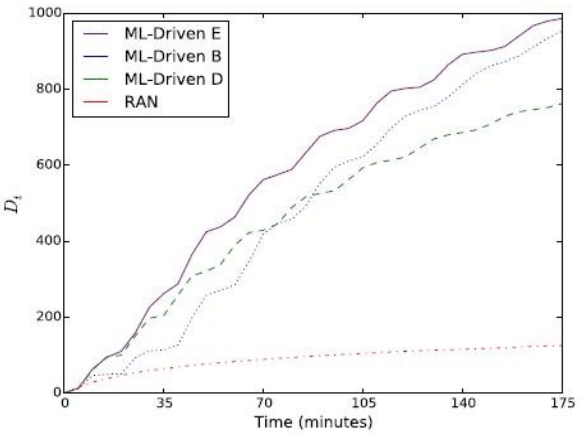

Fig. 6. Average number of bypassing tests for ModSecurity.

C. ModSecurity Results:

For the open source web application firewall, we arbitrary choose nine guidelines from web application SugarCRM, Cyclos and HotelRS. From the above fig.6 we can represent the average number of different bypass tests in the ModSecurity that count of 5 minutes of gap. The result of each individual guideline is in above report.

The first measurement in all the techniques can achieve test to bypass the web application firewall, recommending the web application firewall does not give full cover from SQL injection attacks, bringing online system is on risk. Moreover, through examine the SQL statement execution in database, we can bypass tests that can do SQLi vulnerabilities in SugarCRM, Cyclos & HotelRS. Secondly, Machine learning technique i.e., ML Driven B/D/E, show more powerful than RAN. Overall, the outcome shows that the Machine Learning Driven methods surpass RAN by an order of consequence & bypassing tests in the ML-Driven methods. ML- Driven E consistently finds the most bypassing tests as compared to Machine Learning Driven D/E which helps allocation works well. The changes in ML Driven E and its protype are always statistically significant in all the five-time windows. One issue with M-Driven B/D is start the test from starting. ML Driven B and ML Driven D develop a fixed mutant's number per test, efficiently E adjusts the mutant's number in proportion to the bypassing test. So, Finally Machine Learning Driven E test is efficiently, effective more and finds bypassing attacks sooner.

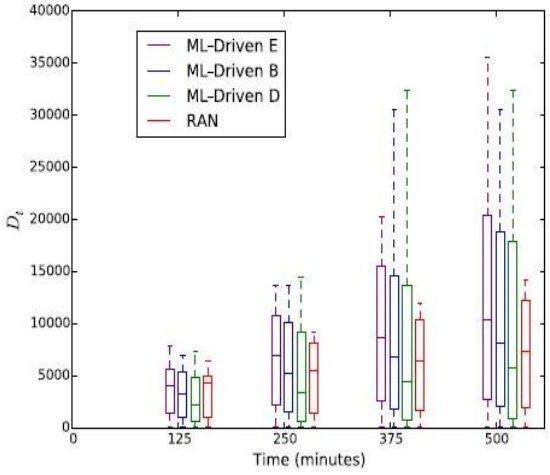

Fig. 7. Bypassing tests in the number of Boxplots

COMPARISON OF THE ATTACK PATTERNS FOUND BY ML-Driven E, ML-Driven B, AND ML-Driven D

Id	Pattern	ML-E	ML-B	ML-D
pc_1	*or␣1, /**/*	√	√	√
pc_2	*or, ␣1*	√	√	✗
pc_3	*or,), #*	√	√	√
pc_4	*or, ␣0*	√	√	√
pc_5	*or, true, #*	√	✗	✗
pc_6	*or, 0*	√	√	√
pc_7	*or, "*	√	√	√
pc_8	*or, /**/, #, ␣, 0, !*	√	✗	✗
pc_9	*or, /**/, 1, ~␣1*	√	√	✗

From the above attack pattern comparison, Machine Learning Driven E generates the most attack pattern. If we talk about the most bypassing test out of all the techniques then of course Machine Learning Driven E generates the most among ML Dirven D/B and RAN produces the minimal bypassing test in all. From the Fig. 7 of boxplots of number of bypassing tests. Here parameter of same group shares similar input constraints to produce similar bypassing test. It's not necessary that the same group produce same results.

We have seen here three different kind of technique in machine learning for generation of SQLiA named as Machine Learning Driven B, Machine Learning Driven D & Machine Learning Driven E. Machine Learning Driven D and Machine Learning Driven B are basically generating test in different strategies whereas Machine Learning Driven E rectify these differences and produce best performance.

Our analysis prove that ML Driven is actually perform well with other techniques. We also used the bypassing-attacks in string pattern to fix web application firewall. Also, we extract more and more path condition that helps to identify the pattern.

V. CONCLUSION AND FUTURE WORK

Web application firewall shows a critical job in online frameworks. The growing event of newly kind of attacks & expanding advancement necessitate that firewalls to be updated and tried normally, as generally attacks may stay undetected and achieve the frameworks under protection.

In this we added a context free grammar i.e., CFG for the SQL injection attack that uses in various different context like testing web services or application directly. With the help of grammar, we come up with an approach based on ML efficiently and more effectively detect the SQLiA in WAF's. Also, we see that the achievement which we get from Machine Learning Driven techniques is higher than RAN technique. We clearly show that the ML Driven achieve a large number of bypassing tests to pinpoint the issues in the firewall. With these we can able to extract successfully pattern and fix the firewall and prevent the SQL injection attack.

For the future work of our project then we will more focus on the automation like approaches that generate effectively patches for the web application firewall and learn new kind of attack pattern so that the system become more and more secure.

REFERENCES

1. A. D. Brucker, L. Brugger, P. Kearney, and B. Wolff, "Verified firewall ¨ policy transformations for test case generation," in Proc. Third Int. Conf. Softw. Testing, Verification Validation, 2010, pp. 345–354.
2. S. Varrette, P. Bouvry, H. Cartiaux, and F. Georgatos, "Management of an academic HPC cluster: The UL experience," in Proc. Int. Conf. High Performance Comput. Simul., Bologna, Italy, Jul. 2014, pp. 959–967.
3. E. Al-Shaer, A. El-Atawy, and T. Samak, "Automated pseudo-live testing of firewall configuration enforcement," IEEE J. Sel. Areas Commun., vol. 27, no. 3, pp. 302–314, Apr. 2009.
4. N. Antunes, N. Laranjeiro, M. Vieira, and H. Madeira, "Command injection vulnerability scanner for web services," 2009. [Online]. Available: http://eden.dei.uc.pt/ mvieira/
5. N. Antunes, N. Laranjeiro, M. Vieira, and H. Madeira, "Effective detection of SQL/XPath injection vulnerabilities in web services," in Proc. 6th IEEE Int. Conf. Services Comput., 2009, pp. 260–267.
6. D. Appelt, N. Alshahwan, and L. Briand, "Assessing the impact of firewalls and database proxies on SQL injection testing," in Proc. 1st Int. Workshop Future Internet Testing, 2013, pp. 32–47.
7. D. Appelt, A. Nguyen, Cu D. Panichella, and L. Briand, "Automated testing of web application firewalls," Univ. Luxembourg, Luxembourg City, Luxembourg, Tech. Rep. TR-SnT-2016-1, 2016.
8. D. Appelt, C. D. Nguyen, and L. Briand, "Behind an application firewall, are we safe from sql injection attacks?" in Proc. IEEE 8th Int. Conf. Softw. Testing, Verification Validation, 2015, pp. 1–10.
9. D. Appelt, C. D. Nguyen, L. C. Briand, and N. Alshahwan, "Automated testing for sql injection vulnerabilities: An input mutation approach," in Proc. 2014 Int. Symp. Softw. Testing Anal., 2014, pp. 259–269.
10. D. Appelt, A. Panichella, and L. C. Briand, "Automatically repairing web application firewalls based on successful SQL injection attacks," in Proc. 28th IEEE Int. Symp. Softw. Rel. Eng., Toulouse, France, Oct. 23–26, 2017, pp. 339–350.