Model Query, Tokenization and Character Matching: A Combined Approach to Prevent SQLIA

Sudhakar Choudhary
Student
SISTech-E
Bhopal, MP, India

Arvind Kumar Jain
Assistant Professor
SISTech-E
Bhopal, MP, India

Anil Kumar
M.Tech
IIIT, Allahabad
UP, India

ABSTRACT
With the rise of internet, web application, such as online banking and web-based email, the web services as an instant means of information dissemination and various other transactions has essentially made them a key component of today's Internet infrastructure. Web-based systems consist of both infrastructure components and of application specific code. But there are many reports on intrusion from external hacker which compromised the back end database system. SQL-Injection Attacks are a class of attacks that many of these systems are highly vulnerable to.

Keywords
SQL Injection Attack, SQLIA Prevention, Tokenization, Character List.

1. INTRODUCTION
Information is the most important business asset today and achieving an appropriate level of information security can be viewed as essential requirement. SQL Injection Attacks (SQLIAs) are one of the topmost threats for web application security and SQL injections are one of the most serious vulnerability types. However, these are easy to detect and exploit; that is why SQLIAs are frequently employed by malicious users for different reasons, e.g. financial fraud, theft confidential data, deface website, sabotage, espionage, cyber terrorism, or simply for fun. Furthermore, SQL Injection attack techniques have become more common, more ambitious, and increasingly sophisticated, so there is a deep need to find an effective and feasible solution for this problem. To achieve those purposes, automatic tools and security systems have been implemented, but none of these are complete or accurate enough to guarantee an absolute level of security on web applications. One of the important reasons of this shortcoming is that there is a lack of common and complete methodology for the evaluation either in terms of performance or needed source code modification which in terms is an over head for an existing system. So The authors feel that there should be such type of mechanism which will be easily deployable, does not need source code modification as well as provide a good performance and to achieve this, proposed research work is driven to the way of developing a new modified SQL injection detection technique.

Proposed research work focused on the analyses and resolution of the problem of SQL Injection attacks, in order to protect and make reliable web applications. The authors try to provide a technique to prevent SQLIAs without any source code modification and without huge performance degradation. To achieve desired goal the authors propose a general and complete evaluation methodology which can be easily deployable to an existing system to preserve the security of the system against SQLIAs. In the proposed research work, The authors combine the techniques describe by AMNESIA [6] and ADAPTIVE METHOD [1], with some modification, and try to remove the necessity of the source code modification as well as to minimize the runtime response time.

2. PRPPOSED METHODOLOGY
In this combined method, the authors maintain a database for storing valid query structure called as *model query*. In runtime validation it checks the dynamically generated query with the previously stored *model queries* and *characters list* to determine the possible SQLIA. Database of the valid query structure and character list is made in static phase. The authors are storing all the valid query structure by linked list representation where each individual *singly link list* represents a valid query structure and to store the starting address of all these singly link list the authors use a doubly link list called as *main link list* whose each node store the starting address of a singly link list. In second stage of static phase the authors create a list of characters such as: *single quote, double quote, semi colon, double dash, slash and SQL Keywords*. The authors store character list in an array. So when the authors found a new query is arrived to the database server, after the tokenization process, at first the authors start searching the structure of the query in the linked representation and then match the characters of inputted value with the characters stored in the character list. If it is a successful search, according to first stage and no match found according to second stage then that query will be a valid query otherwise it's interpreted as an SQLIA.

In this scheme, the authors stored the structure of the query by preserving the order of the sequence of tokens generated by the query; means the authors are checking the sequence of tokens generated by the arrived query is in the same order as the authors stored in the model query. If the sequence of the arrived query is in same order as the query stored in the database then that arrived query interpreted as valid query, otherwise if the authors do not found any ordered sequence like that arrived query in the entire database then it's a possible SQLIA as the authors stored all the possible structure of the valid SQL query in the database in static phase. During the searching phase of the query in the singly linked list if the authors found a valid query, means: (1) Number of tokens generated by the dynamic query is same as that of model query and (2) Except from *user inputted token value* all other tokens are same in both the queries, then extract that token value which is inputted by user and parse the queries by character wise. If any of these characters match with the characters stored in character list then reject that query as it may cause SQLIA.

In the proposed technique when a query executed from the application program for validation, it knows that which action point of the application program generates this query, so the

authors have to match this incoming query only to those model queries which belongs to that action point.

```
// DBCS is a connection string defined in web.config file
string CS = ConfigurationManager.ConnectionStrings["DBCS"]
                                .ConnectionString;

public int getEmployeeId(string empName, string empPwd)
{
    SqlConnection conn = new SqlConnection(CS);
    if (conn.State == ConnectionState.Closed)
    {
      conn.Open();
    }
    string query = "select ID from Employee where" +
                   "EmpName = @name and EmpPwd = @pwd";
    SqlCommand cmd = new SqlCommand(query, conn);
    cmd.Parameters.AddWithValue("@name", empName);
    cmd.Parameters.AddWithValue("@pwd", empPwd);
    int result = cmd.ExecuteNonQuery();
    conn.Close();
    return result;
}
```

Figure 2.1: finding of Action Point

In the figure 2.1, red part of the code indicates that action point. An *action* point is defined as a point in the application code that issues SQL queries to the underlying database. There is one query model for each action point. For each ***Action Point*** the authors generate a query model that represent all the possible queries generated by that *Action Point* and store the length of all possible model queries in an array. There is one array for each *Action Point*.

Figure 2.2: Array to store the length of queries

To store the starting address of the doubly link list or the starting address of the group the authors use this array where each cell of the array stores the starting address of a group. The index of this array cell will represent the group number, the number of tokens each singly link list possesses in that group. For example, in figure 2.2, If a group having all the singly link list with 'n' number of tokens then the starting address of the group be assigned to n^{th} cell in the array. In addition, in this representation there are many cells in the array may not be used but by using some extra storage the authors have a great advantage that The authors don't need to search the starting address of a specific group because after calculating the number of tokens in the incoming query the number itself represents the cell number of the array holding the starting address of the group that the incoming query may belong. Suppose the authors get this query from that *action point* which the authors have discussed in the previous section.

select ID from Employee where EmpName = "vats" and EmpPwd = "vatsChy"

To find an ordered sequence of token in that singly link list for an incoming query to the database the authors first separate the tokens from the query by a SQL parser of the specific DBMS. After parsing those into tokens, on the basis of *space* by using code, convert these string tokens into sum of its ASCII Code value of each character. For example consider the keyword 'select', the corresponding ASCII decimal values for the literals is s = 115, e = 101, l = 108, e = 101, c = 99, t = 116, so adding the ASCII value of each literals the authors get the corresponding integer value of 'select' is 640.

select = 115 + 101 + 108 + 101 + 99 + 116 = 640

After getting the ASCII value of all strings of a query the authors store into in the form of linked list. If the authors closely analyze any web application most of the cases similar type of query is used with different user input. To store a valid individual query structure, the authors preserve the sequence of tokens generated by the query using a singly link list where each node store a single token of a query. After token separation and integer conversion the authors get the ordered sequence of the tokens of the query then the authors start searching the singly link list.

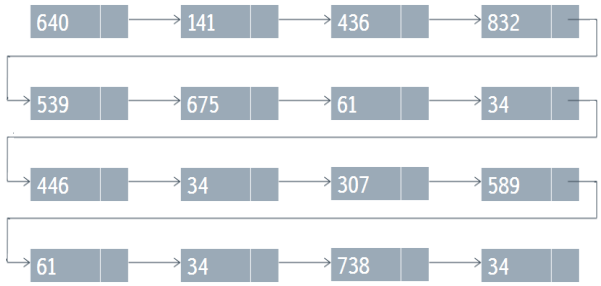


Figure 2.3: Singly linked list to store integer tokens

The figure 2.3 shows the linked list representation of the query after token translation and integer conversion. For a valid incoming query the number of tokens is same as the number of tokens in its corresponding query structure in the Database. However, to reduce the search space and time the authors group together all the query structure having same number of tokens. It can be interpreted as a query having 4 tokens belongs in a separate group than a query having 5 tokens. Therefore, before searching the similar structure for an incoming query the authors first calculate the number of tokens it have, then the authors start searching in the group having all the structure of valid query having the same number of tokens. Moreover, to group all the singly link list having same number of tokens, The authors use a doubly link list usually referred as *main link list* whose each node holds the starting address of a singly link list among all the singly link list having same number of tokens. That means if The authors have 'n' groups of singly link list then The authors have 'n' number of doubly link list and for an incoming query The authors only search a single doubly link list among the 'n' number of list. Figure 2.4 shows a group of singly linked list having 4 tokens connected to main linked list.

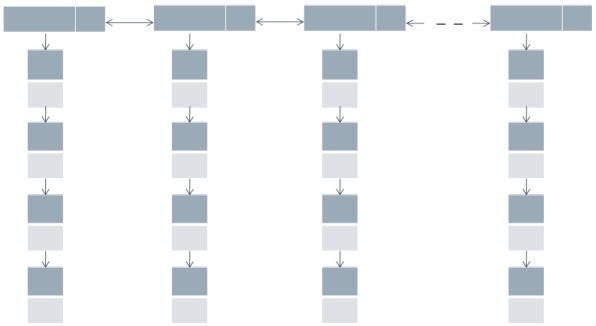


Figure 2.4: Singly and Doubly linked list connection

While searching the single link list if position of the token from the incoming query matches the token of the same position in the singly link list then The authors move to the next token as well as to the next node of the list until any mismatch found or the end of the list. In this way if the authors reach at the end of the single link list and there is no more token left in the incoming query then it's a successful search and the incoming query is a valid query. But during the matching process if a mismatch occurs and the node in the linked list indicate that it is a position of user input then the authors extract the value of the token which are inputted by user and parse the queries by character wise by using some code. Now match these characters with the characters stored in the ***character list***. Character List is a list which contains some prohibited characters such as:

' , ' , " , " , space , -- , / and SQL Keyword

If any character of token variable match with the characters stored in the character list then that query will be an invalid query otherwise go for matching process of the next token. If the authors found a match at this position then continue the matching process or if a mismatch found at that position then it is clear that the link list the authors are searching for the similar query structure for the incoming query is not correct list so the thread searching this singly list should stop its execution. The search for the similar structure for the incoming query is done in this fashion in a multithreaded way. If no similar type of structure found in the model query then it's a possible SQLIA. But if a thread found the correct path, it intimates the other threads to terminate as it already performs a successful search. From the above description of matching technique it is clear that for a successful search, number of token in the incoming query is same as the length of the link list stored its structure. In the figure 2.5 the authors combine all the procedures in a single unit.

For runtime token matching, if the authors used literal wise matching then it will be a huge computational over head. In worst case the authors have to check 'n' number of literals for each 'q' number of query available in the database of the same length of the incoming query, therefore the complexity will be O (n × q). Instead of using literal wise string matching algorithms the authors simply mapped each token into an integer value. The authors also store these integer values in the database instead of storing the tokens in string format. It also takes very less space, for example if there are 'n' no of literals and 'm' no of tokens in the query instead of storing 'n' no of values The authors are storing 'm' no of values where m<<n. So in run time validation the length of a singly link list storing the query having 'm' no of tokens is m. In cases of incoming query after token separation the authors transform each token into its corresponding integer values then the authors start searching. In worst case the authors have to check 'm' no of integer values for each 'q' no of query, so the complexity become O (m × q) instead of O (n × q) where m<<n. This is a performance gain as both space complexity and time complexity improves.

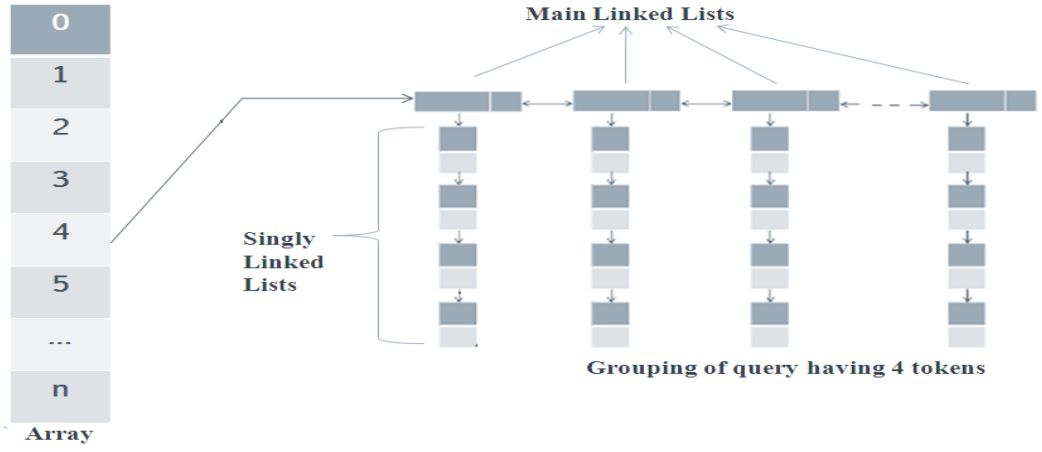


Figure 2.5: The complete structure of proposed methodology

3. RESULT ANALYSIS

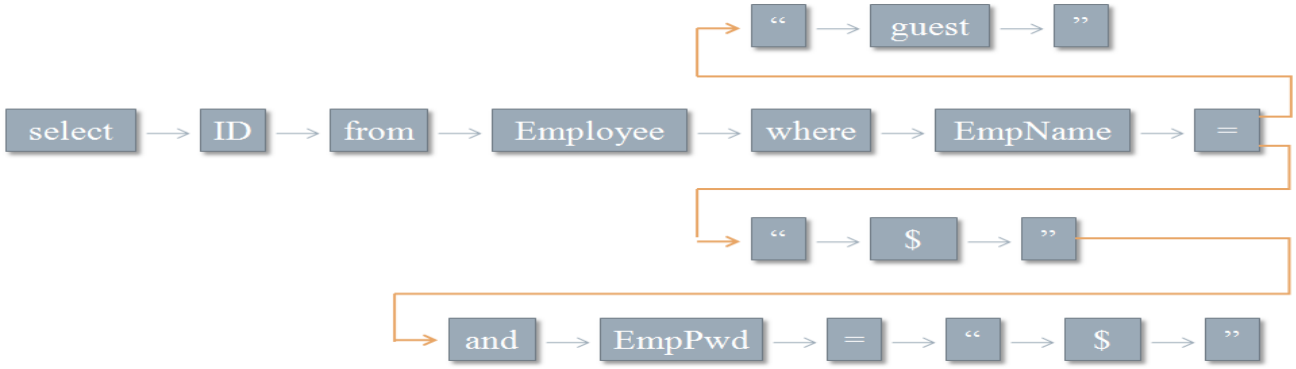
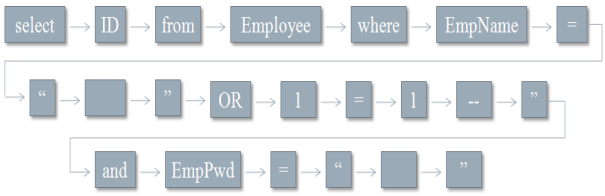


Figure 3.1: Design of query model for LogIn Action Method

Suppose there is a query model for ***LogIn*** action point on the home page by which user can visit website by two mode of authentication, either user is a registered user or any user visit website to just take a look. Figure 3.1 shows the two modes of authenticated user. If user is not a registered user then he/she will be treated as ‘guest’ but if user already registered him/her then he/she will have to input ‘EmpName’ and ‘EmpPwd’ field with correct information. These input variables will take the places of both the ‘$’ placeholders. This *LogIn Action Query Model* have two *model queries* having query length (number of tokens) 10 and 16. The separation point for both the structures is ‘=’, after EmpName. The structure above this ‘=’ is for ‘guest’ user and the structure below this ‘=’ is for registered user. User inputs a *blank space* for token number 15 in the lower part of figure 3.1 in case 1 and a *$* sign in case 2. For the sake of better readability and understandability the authors represent tokens in the form of string in place of integer in the figure 3.1 and figures present in all the three cases in this section.

3.1 Case 1

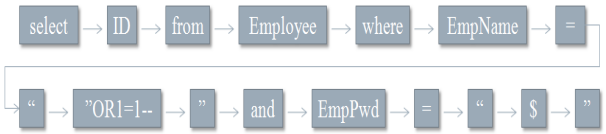
User inputs ‘ **” OR 1 = 1 --**’ for *EmpName* variable and ‘ ’ for *EmpPwd* variable. So the complete query formed after tokenization is given in the picture below:



In this case user inputs value for the *EmpName* field, token number 9 in the lower part of figure 3.1, separated with blank space and the authors know, according to proposed method, tokens are separated whenever a *blank space* found so the number of tokens generated by this query is 22. Now control goes to array and search for the cell which contains the value 22 but in array there is not a single cell having value 22. So further steps will not take place and process stops as the query structure doesn’t match with any of the structure described (guest or registered) in the model queries. So this is an invalid query.

3.2 Case 2

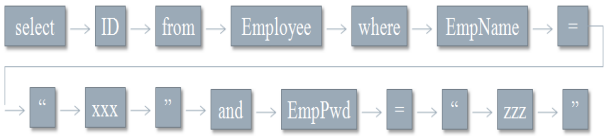
User inputs ‘ **”OR1=1--**’ for *EmpName* variable and ‘ ’ for *EmpPwd* variable. Since no blank space is inputted by user for token number 9 in the lower part of figure 3.1 so the complete query formed after tokenization is given in the picture below:



The number of tokens generated by this query is 16, so the control goes to array and search for the cell which contains the value 16 and found it. This cell redirects query to the main linked list which contains the address of the singly linked list which further contains the valid query model. As the authors reach at the very first node of the singly linked list matching will start. After successful matching up to 8th token, when The authors reach at the ‘9th’ token of the query (since it is an input field for ‘EmpName’) The authors get the value of that token and parse the query on character basis by using code. Now match each character with the character stored in character list. Here match occurs as input variable contains *single quote* and *double dash*. So this is an invalid query.

3.3 Case 3

User inputs ‘**xxx**’ for *EmpName* variable and ‘**zzz**’ for *EmpPwd* variable so the complete query formed after tokenization is given in the picture below:



The number of tokens generated by this query is 16, so the control goes to array and search for the cell which contains the value 16 and found it. This case acts same as case 2 until control reach at the 8th token in the above figure. After successful matching up to 8th token, when The authors reach at the ‘9th’ token of the query The authors get the value of that token and parse the selected queries into on character basis by using code. Now match each character with the character stored in character list but no match are found, so process goes for next token matching and successfully matched up to 14th token. Again when the authors reach at the 15th token same matching process executes but this time also no match found, so process goes for the next and last token matching. After the matching of 16th token the authors can say that the entire tokens are matched with the stored query structure and character list so this is a valid query.

4. ALGORITHM

Step 1: Separates each token from the *sqlStatement*, store in tokenSequence and returns total number of tokens as *numberOfToken*.

```
numberOfToken = tokensOfSql (tokenSequence,
sqlStatement);
```

Step 2: Convert each token into its corresponding integer value and store it in a linked list named as *integerSequence*.

```
tokenToInt (numberOfToken, tokenSequence,
integerSequence)
```

Step 3: Initialize starting node of the doubly link list to search.

```
while (searchNode -> rlink! = NULL) do
{
    startThreadSearch (tokenSequence,
    integerSequence, searchNode);
    if (token is an input field)
    {
        if (searchValidCharacter ( ))
        {
            return false;
            // it's a SQL injection
        }
        else
        {
            return true;
            // it's a valid query
        }
    }
    searchNode = searchNode -> rlink;
}
```

Step 4: After a successful search; the thread, which found the right sequence, informs all other thread to stop execution as it found a valid query structure.

Step 5: Exit.

Declaration of variables:

sqlStatement is a query string generated by the Action Point.

tokenSequence is 2D arrays where each row represents a token and each cell holds a literal of that token and total number of rows represent total number of tokens it stored.

numberOfToken is the total number of tokens present in *tokenSequence.*

integerSequence is an integer linked list where each cell stores an integer value of a token.

searchNode = sqlDbArray [*numberOfToken*]

sqlDbArray is an array holding the starting address of all groups storing individual array.

Declarations of methods

tokensOfSql ( ) separates each token from the *sqlStatement* on the basis of *space*, collection of characters between two *spaces* will be treated as a token, and store into the in *tokenSequence* and returns total number of token in a *sqlStatement.*

tokenToInt ( ) convert token into its corresponding integer value. For example, *select* will be converted to 640.

startThreadSearch ( ) search for an ordered sequence of integer provided by *integerSequence* in the singly link list whose starting address is stored in searchNode. This method will call itself as long as tokens of dynamically generated query match with the stored one. If tokens do not match then this method will stop processing and control will go to next singly linked list of the same group and same process will execute.

searchValidCharacter ( ) returns true if a thread perform a successful match. In this method matching of inputted character with the character stored in character list is performed. During the processing of *startThreadSearch* method if a token is found which is actually an input data value then this method will executes. This method will extract each literal which is present in input data value and compare the data value with the stored character in the character list, if a match found then this method will return false and query will be treated as a SQL injection.

5. CONCLUSION AND FUTURE WORK

As it is a multi threaded implementation is fully utilized the newly available multi core processors and performs the search quickly. However, due to use of array indexing techniques the frequently generated SQL query structure parsing will be processed quickly which is a performance gain to the existing available solution. As in the proposed scheme, validates each dynamically generated SQL at runtime, this approaches increases the runtime overload of the system but reduces the possibility of SQLIA. The authors' proposed technique can also detect such type of attack which are may cause by only changing the data value of a query but not changing the structure of the query by *character matching technique* for the inputted value in the query. In the proposed method up to token matching stage all test cases works fine but in the character matching stage with some case such as when each character/literal of inputted data value is matched with characters stored in character list, there may possibly that the proposed technique will not result as optimal results. It may happen that inputted data value may contain some SQL keywords which correct according to an authenticated user point of view but since that input value is matched with character list content so it will be treated as an invalid query, furthermore, further in future authors will expand this solution to handle the character matching algorithm and develop a tool to enhance the efficiency of the proposed method. Currently, the proposed system approaches preventing from almost most of the attack of SQLIA.

6. AUTHORS CONTRIBUTION

All authors have contributed equally to his work and declare no conflict of interest.

7. REFERENCES

[1] Noor Ashitah Abu Othman, Fakariah Hani Mohd Ali and Mashyum Binti Mohd Noh: Secured Web Application Using Combination of Query Tokenization and Adaptive Method in Preventing SQL Injection Attacks. *2014 IEEE, 2014 International Conference on Computer, Communication, and Control Technology (l4CT 2014), September 2 - 4,2014 - Langkawi, Kedah, Malaysia*

[2] Anamika Joshi and Geetha V: SQL Injection Detection using Machine Learning. *2014 International Conference on Control, Instrumentation, Communication and Computational Technologies (ICCICCT) ©2014 IEEE.*

[3] Jaskanwal Minhas, Raman Kumar. Blocking of SQL Injection attack by Comparing Static and Dynamic queries. *International Journal of computer network and Information Security 2013*.

[4] A. Dasgupta, V. Narasayya, M. Syamala. A Static Analysis Framework for Database Applications. *IEEE 25th International Conference on Data Engineering*. Pages 1403 – 1414, March 2009.

[5] W. Halfond, J. Viegas and A. Orso. A Classification of SQL Injection Attacks and Countermeasures, *Proceedings of the IEEE International Symposium on Secure Software Engineering (ISSSE)*, 2006

[6] W. G. Halfond and A. Orso. AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks. In *Proceedings of the IEEE and ACM International Conference on Automated Software Engineering (ASE 2005), Long Beach, CA, USA, Nov 2005.*

[7] Wikipedia, "SQL injection" http://en.wikipedia.org/wiki/SQL_injection

[8] William G. J. Halfond, Alessandro Orso. Combining Static Analysis & Runtime Monitoring to Counter SQL-Injection Attacks. *SIGSOFT Software Engineering Notes Volume 30 Issue 4.* July 2005.

[9] Kumar, Anil, Rohit Agarwal, and Rahul Kala. "Temporal Logic based Motion Planning in Unstructured Environments."

[10] F. Valeur, D. Mutz, and G. Vigna. A Learning-Based Approach to the Detection of SQL Attacks. *In Proceedings of the Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, pages 123–140, 2005.

[11] Boyd and A. Keromytis. SQLrand: Preventing SQL injection attacks. *In Proceedings of the Applied Cryptography and Network Security (ACNS), pages 292–304, 2004.*

[12] G. Wassermann and Z. Su. An Analysis Framework for Security in Web Applications. In Proceedings of the FSE Workshop on Specification and Verification of Component-Based Systems (SAVCBS), pages 70–78, 2004.

[13] Kumar, Anil, and Rahul Kala. "Linear Temporal Logic-based Mission Planning." IJIMAI 3.7 (2016): 32-41.