Constructing PLC Binary Program Model for Detection Purposes

Tianyou CHANG1*, Qiang WEI1, Yangyang GENG1, Hongwei ZHANG2

1 State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou Henan 450001, China
2 The people’s bank of china center branch of Zhoukou city, Zhoukou Henan 466000, China
641644683@qq.com

Abstract. Programmable Logic Controller (PLC) programs are vulnerable to tampering attacks with addition of malware, which can substantially cause severe physical destructions. In order to solve the problems, We propose a static analysis method of Binary PLC programs for the detection of PLC malwares. Firstly, disassemble the binary program into STL program. Secondly, constructed the control flow graph of STL program according to the characteristics of STL language. Thirdly, generate the program execution paths according to traverse the control flow graph and analysis loop paths. Finally the PLC program output and input mapping relations are obtained based on the program execution paths. The experiment shows that the proposed method has achieved the analysis of PLC programs and get each path predicate and the mapping relation between output and input.

1. Introduction

With the integration of informatization and industrialization and the introduction of Industry 4.0[1], the industrial control system has been upgraded from stand-alone to interconnected mode. Moreover, it has become open and intelligent instead of being closed and automatic. However, the risks in the industrial control system are also increased at the same time. It should be noted that an industrial controller is a computer that is dedicated to an industrial control system. The PLC can control the entire operation for the industrial control system by executing its codes. Meanwhile, PLC codes are vulnerable to tampering attacks such that it can cause physical destruction. Therefore, it is of great significance to protect the PLC codes from tampering attacks.

It has been shown by the search engine SHODAN[2] that thousands of industrial control systems are directly accessible via the Internet. However, there is no consideration of the information security in the phase of designing PLCs, which causes various vulnerabilities and makes PLC programs vulnerable to tampering attacks[3][4][5][6]. A tool that achieves SNMP scan in the local network and socks5 agent in PLCs for s7-300/400 by modifying the PLC codes[5]. A new worm is implemented which can spread on PLCs and perform any function by inserting itself into user programs[6]. In the protection of PLC programs, Daniel et al. converted SCL language to the NuSMV model[7] and then they proposed a method of formal verification of complex properties on PLC programs[8]. In 2016, Nannan proposed a method of mutation-based verification for Simulink design models to verify the PLC programs[9]. To a certain extent, these methods can perform the validation of the PLC codes, but they cannot analyze the functions provided by the configuration software. According to these attacks...
above, it can be observed that these tampering attacks on the industrial control system are implemented by tampering with the binary bytecode programs in PLCs. Therefore, these security detection methods for source code cannot be applied to malicious code in PLC.

In order to solve the problems, We propose a static analysis method of Binary PLC programs for the detection of PLC malwares. Fig. 1 illustrates the technical route in this paper. Firstly, the binary bytecode programs are extracted from PLCs and then they are compiled into STL programs. Secondly, the control flow graph is constructed according to the characteristics of STL language. Since STL syntax is obscure and there is much implicit data, it is difficult to analyze the data flow in STL language. In this paper, the implicit status word will be replaced with the displayed jump instructions, and timers will be considered as important factors during constructing PLC program model. Thirdly, The executable paths of the PLC program can be obtained preliminarily by traversing the control flow graph. Moreover, a method of summarization generation for loop statements is employed for the loop statements in loop paths. Subsequently, it will generate the new executable paths. Finally, the cumulative conditional constraints for every path and the corresponding mapping relation between input variables and output variables can be obtained by implementing the data flow analysis for every executable path.

2. Program model construction

2.1. Programmable Logic Controller

Industrial Control System (ICS) is the general term for control systems used in the industrial production. In particular, it includes supervisory control and data acquisition (SCADA), distributed control system (DCS), programmable logic controller (PLC), intelligent terminal, human machine interface (HMI) and so on.

PLC is a computer dedicated to the industrial control, where the hardware structure is roughly the same with the micro-computer. In general, it includes the power supply, CPU, memory, IO module, communication module and other components. The engineer station can compile the PLC source program such that a binary bytecode program can be generated. Subsequently, it is sent to the PLC, which is executed in the form of feedback loops. Its working process can be divided into three phases. In particular, they are the input sampling phase, the implementation phase of user programs, and the output refreshing phase.

At the input sampling phase, PLCs sequentially read all input value of sensors and store them into the I/O area which forms a snapshot of sensors. At the implementation phase of user programs, the user programs are executed. Thus, the default scan cycle time should be less than 150ms. If the scan cycle time expires, an interrupt routine will be called. At the output refreshing phase, the CPU will refresh the data of I/O area into the output circuits and then drive the corresponding peripherals via the output circuits.

For Siemens PLC, PLC programs can be programed in Ladder Diagram (LAD), Function Block Diagram (FBD), structured control language (SCL) and Statement List (STL), respectively. In contrast to the text-based SCL and assembler-like STL, the LAD and FBD languages are graphical. It should be noted that there should be at least one organization block called OB1 in a PLC program. In addition, the OB1 is comparable to the main() function in a traditional C program. Moreover, PLC programs
written in any language can be compiled into MC 7.

In general, the input variables in PLC programs include multiple variables to receive the input parameters of devices. Meanwhile, the output variables generally include multiple variables to control the behavior of different actuators respectively. It should be noted that the numbers of input variables and output variables are fixed. In addition, timers, just as the output variables are often used in PLC programs to control the execution of the actuator and the number of timers of the same program is fixed. Thus, a group of input variables is treated as the input vector and a group of output variables and a group of timer are treated as the output vector. Moreover, the output vector will determine the actions performed by the industrial control system equipment. Therefore, industrial control equipment cannot perform the correct actions due to the change of the PLC output vector-value caused by most of the industrial control system attacks.

2.2 Disassemble binary programs into STL programs
Siemens S7Comm protocol is implemented for the communication between PLC and PC, with functions of uploading and downloading the program blocks. Therefore, binary program blocks from PLC by S7 protocol can be obtained. Meanwhile, the binary program blocks are encoded in MC7code for the Siemens PLC. Moreover, its structure is demonstrated in literature [5], as follows in Fig.2.

| Description                        | Byte | Offset |
|-----------------------------------|------|--------|
| Block signature                   | 2    | 0      |
| Block version                     | 1    | 2      |
| Block attribute                   | 1    | 3      |
| Block language                    | 1    | 4      |
| Block type                        | 1    | 5      |
| Block number                      | 2    | 6      |
| Block length                      | 4    | 8      |
| Block password                    | 4    | 12     |
| Block last modified date          | 6    | 16     |
| Block interface last modified date| 6    | 22     |
| Block interface length            | 2    | 28     |
| Block segment table length        | 2    | 30     |
| Block local data length           | 2    | 32     |
| Block data length                 | 2    | 34                 |
| Data (MC7 / DB)                   | X    | 36     |

| Block signature                   | 1    | 36+x   |
| Block number                      | 2    | 37+x   |
| Block interface length            | 2    | 39+x   |
| Block interface blocks count      | 2    | 41+x   |
| Block interface                   | y    | 43+x   |

It should be noted that a tool is provided in the open source software DotNetSiemensPLCToolBoxLibrary, which can substantially translate the PLC binary programs into STL programs. Therefore, the tool is employed to disassemble PLC binary programs into STL programs.

2.3 Control flow graph construction
STL language is in a compilation level, and it is close to IL language that complied with the IEC61131-1 standard. The STL instructions can be classified into different types based on effects of the STL instructions on the program control flow. They are shown in Tab.1.
Tab 1. STL instructions classification

| Type of instruction                      | Instructions                      |
|------------------------------------------|-----------------------------------|
| General instructions                     | all instructions except for changing the program flow class instructions |
| Unconditional jump instruction           | JU, JL                            |
| Conditional jump instruction             | JC, JCN, JCB, JNB, JBI, JNBI, JO, JOS, JZ, JN, JP, JM, JPZ, JMZ, JUO |
| Loop instruction                         | LOOP                              |
| Block unconditional end instruction      | BEU, BE                          |
| Block conditional end instruction        | BEC                              |
| Unconditional Call block instructions    | UC, CALL                         |
| Conditional Call block instructions      | CC                               |

It should be noted that the control flow graph is often used for intermediate representation of program analysis, which can indicate the control flow of the program. Moreover, it can directly affect the data flow analysis and the way of program model construction. The program control flow graph is the set of nodes and directed edges. It can be represented using the binary structure $\text{CFG} = (N, E)$, where $N$ represents the set of nodes of the control flow graph, and $E$ represents the set of directed edges of control flow graphs.

In addition, the control flow graph $\text{CFG}$ is stored in a cross-linked list. The nodes are used to record the characteristics of the STL instruction during the construction of control flow graph for the STL program. In particular, the data structure is as follows.

```c
class Node {
    int Nnum;
    string Type;
    int Next;
    int Next_Jump;
    string Instr;
}
class Edge {
    int Enum;
    bool Is_jump;
    int Node_hNum;
    int Node_tNum;
}
```

where $\text{Nnum}$ denotes the number of the instruction node, $\text{Type}$ denotes the node type, $\text{Next}$ denotes the number of next node arriving along the sequential edge, $\text{Next}_\text{Jump}$ denotes the number of next node arriving along the edge of jump type, and $\text{Instr}$ denotes the instruction of the node.

The directional edges are constructed for the connection among the associated nodes. Thus, the data structure can be shown as follows.

```c
class Node {
    int Nnum;
    string Type;
    int Next;
    int Next_Jump;
    string Instr;
}
class Edge {
    int Enum;
    bool Is_jump;
    int Node_hNum;
    int Node_tNum;
}
```

where the $\text{Enum}$ represents the number of the directed edge, $\text{Is_jump}$ represents whether the edge is the jump type or not, $\text{Node}_h\text{Num}$ represents the head node number of the directed edge, and $\text{Node}_t\text{Num}$ represents the tail node number of the directed edge.

According to the instruction features of STL language, the corresponding control flow graph can be constructed. Moreover, the process of construction can be described as follows.

1) An instruction node is created for every STL instruction, and the numbers for these nodes are set according to the location of instructions. Meanwhile, the node type is recorded. For instructions with label, their locations are caught in table A. Subsequently, it is easy to find the destination position of the jump instruction when a jump instruction is encountered.
2) A directed edge is created between two adjacent nodes, and the edge type is set as the sequential type. Next, the linear control flow graph can be obtained initially.

3) For each jump instruction, a direction edge e is added to the control flow graph. Next, the location of label instruction can be found in table A. Subsequently, it can be set as the head node number of edge e. Moreover, the tail node number of edge e is the number of the jump instruction node, and the type of edge e is marked as the jump type.

4) Each edge of the sequential type is traversed. If there is only one incoming edge and one outgoing edge of an edge, the edge will be deleted and the two nodes will be merged into one node. It should be noted that the node number is the smaller one between the numbers of both nodes. Meanwhile, the node type is the sequential type. The step is repeated until there is no edge in the control flow graph mentioned above.

5) Each unconditional jump node is traversed. Moreover, the types of two outgoing edges for the instruction node are analyzed and the outgoing edge of jump type is removed. Ultimately, the control flow graph CFG can be obtained.

By implementing the control flow graph generation method, the corresponding control flow graph can be constructed for the program, which is shown in Fig. 3(b).

![Control Flow Graph](image)

**2.3.1. status word conversion**

In PLC’s CPU, there is a status word known as the status word register. For the S7300 / 400 PLC, the status word is a 16-bit register. However, only 9 bits are actually used.

| 15-9 | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|------|----|----|----|----|----|----|----|----|----|
| Reserved bits | BR | CC1 | CC0 | OS | OV | OR | STA | RLO | FC |

It should be noted that STL syntax is obscure. There are more implicit data but no displayed control flow structures in high-level languages such as loops, branches, and function calls. Consequently, it is difficult to analyze the data flow in STL language. Moreover, whether the jump instruction jumps or not directly depends on the status word rather than variables. Subsequently, it is not conducive to analyze data flow of the program. In this paper, the status word will be replaced with the displayed jump instructions.
Two types of STL instructions are processed. In particular, one is the logic operation instructions or mathematical instructions, which can change the status word. The other is the jump instructions or return instructions, which can affect the control flow of the program.

In this paper, the binary logic operations are employed instead of Boolean logic operations. For Boolean logical operations, it is necessary to determine the termination instructions of the Boolean logic string. In STL instructions, assignment instruction "=" and the jump instructions can be classified as the termination instruction set in the Boolean logic string. In addition, the instructions with logic operations are replaced by binary logic operations. For example, the instruction "A" is replaced by "•", the instruction "O" is replaced by "+". Furthermore, the instructions "A 10.0; A 10.1; O 10.2; Q4.0" is replaced with " RLO = 10.0 · 10.1 + 10.2; Q4.0 = 10.0 · 10.1 + 10.2". For the mathematical operations, the mathematical operations are used to perform substitutions. For example, the instructions "L IB1; + 2; T IB1" is replaced with "IB1 = IB1 + 2".

isJump(Instr_control) is utilized to represent whether the instruction Instr_control satisfies the jump condition or not. In particular, its value becomes 1 when the jump condition is met; otherwise, its value is 0. Jump (Instr_control) can be defined as follows:

\[
\text{isJump}(\text{JU}|\text{JL}|\text{BE}|\text{CALL}|\text{UC}) = 1, \quad \text{all the time}
\]

\[
\text{isJump}(\text{JC}|\text{CB}|\text{CC}|\text{BEC}) = \begin{cases} 
1, & \text{if} \text{RLO} = 1 \\
0, & \text{otherwise}
\end{cases}
\]

\[
\text{isJump}(\text{JCN}|\text{JNB}) = \begin{cases} 
1, & \text{if} \text{RLO} = 0 \\
0, & \text{otherwise}
\end{cases}
\]

\[
\text{isJump}(\text{JBI}) = \begin{cases} 
1, & \text{if} \text{BR} = 1 \\
0, & \text{otherwise}
\end{cases}
\]

\[
\text{isJump}(\text{JO}) = \begin{cases} 
1, & \text{if} \text{OV} = 1 \\
0, & \text{otherwise}
\end{cases}
\]

\[
\text{isJump}(\text{JOS}) = \begin{cases} 
1, & \text{if} \text{OS} = 1 \\
0, & \text{otherwise}
\end{cases}
\]

According to the definition above, it can be seen that the control flow of program is controlled directly by the status word. Thus, it is necessary to analyze the status word.

Where OV is the overflow bit. If an overflow occurs when an arithmetic operation or floating point comparison instruction is executed, OV bit will be set; otherwise, OV bit will be cleared. Moreover, OS bit is the overflow status holding bit. In particular, OS bit is set when OV bit is set and OS bit still remains when OV bit is cleared. When jump instructions are encountered, the OS bit is reset.

BR bit is usually used at the end of the program block, which can indicate whether there are errors in STL instruction. Moreover, its value can be changed by the instructions such as saved, JCB and JNB. Thus, it is necessary to track and analyze these instructions when the BR instruction appears. Since these instructions all assign the current value of the RLO to the BR bit, the binary logic value of the RLO should be first calculated when these instructions appear. Subsequently, the binary logic value of the RLO can be assigned to the BR bit.

2.3.2 Clock analysis
Since the industrial control system generally requires high real-time processing capability, the timer is a very important object of the model. The time in the model consists of the accumulated time of timers and the execution cycle time of PLC programs. Since the timer is a global variable across all the cycles, it will increase the modeling difficulty and the detection time by considering time as a factor during modeling. However, it becomes impossible to detect time-related safety specifications without consideration of time.

There are five different timers used in the STL language, which are pulse timer S_PULSE, extended pulse timer S_PEXT, on-delay timer S_ODT, hold on-delay timer S_ODT and the off-delay timer S_OFFDT. In this paper, the on-delay S_ODT timer will be illustrated as a representative of analyzing the timer, and the other timers work in a similar way as the on-delay timer.

There are two inputs IN and PT for the connection delay timer, where IN represents whether the timer is enabled and its value is the current value of the RLO. Moreover, PT represents the preset value of the timer. The on-delay timer also has two outputs, where one is ET that indicates the elapsed
time since the activation of the timer and the other is the Boolean output Q of the timer. The on-delay timer has the following characteristics.

1) When the enable IN is on, ET starts to count, where output Q of the timer is 0. When the ET is greater than the preset value PT, output Q of the timer becomes 1.

2) When the enable is off, ET is cleared to be 0 and output Q of the timer is 0.

The function \( Q = T_{ODT}(IN, PT, ET) \) is defined, which can represent the mapping relation among the on-time delay timer output Q, the enable terminal IN, the current time value ET and the preset value PT.

\[
Q = T_{ODT}(IN, PT, ET) = \begin{cases} 
1, & \text{if } ET > PT, IN = 1 \\
0, & \text{otherwise} 
\end{cases}
\]

The value of the enable IN is the value of the current RLO, which can be represented by a Boolean logical operation. When the enable IN is on, the ET begins to count. The clock of the PLC processor cannot be simulated whether the dynamic analysis method or the static analysis method of programs is used. In order to express the elapsed time ET in this paper, the ET can be divided into two cases based on the impact of the timer on the clock counter. One is ET> PT and the other is ET<= PT. Subsequently, the corresponding control flow charts can be constructed according to these two scenarios.

For the determination of ET> PT or ET<= PT, it can be determined by the number of cycles that the program keeps on running after the enable IN is started. Since the scan cycle of PLC program is fixed and is set to be C, the scan cycle of PLC program is used for timing, such as \( 4 * C < PT < 5 * C \). In particular, the time greater than PT occurs for the first time after the enable IN is on for 5 scan cycles. The value of output Q for this timer changes from 0 to 1. Therefore, it is necessary to calculate the number of scan cycles after the time is larger than Pt, since each timer enable IN is started. As shown in the formula (1), the timer ET greater than PT occurs for the first time after \( n \)-th cycle since the enable IN is on.

\[
n = PT/C; \quad (1)
\]

Therefore, the output vector is set to be \( o = [o_1, o_2 ... o_n, (T_1.IN, T_1.Q, T_1.NPT, T_1.NET) ... (T_n.IN, T_n.Q, T_n.NPT, T_n.NET)] \), where the timer can be represented by a quaternion. In particular, the NPT can be calculated by formula (1) and can be utilized to represent the value PT preset. Moreover, NET represents the number of cycles that the PLC keeps on running after the enable IN of the timer is on. Since the time of scan cycle is fixed, NET can be utilized to represent the current time of the timer.

The logical operation instructions and mathematical operation instructions can be transformed into binary logical operations. And the corresponding status bit can be calculated according to the relation between jump instructions and status bits. In addition, clocks are take into account the process of constructing control flow graph. Ultimately, a new control flow graph can be obtained in Fig. 3(c).

2.4. executable path generation

The control flow graph of the program is traversed by a method of Depth-first and multiple backtracking. Subsequently, mapping relations between the output vector and the input vector in the STL program and the cumulative conditional constraints of paths can be obtained.

Firstly, the set of executable paths of the program can be obtained. Next, the loop paths are processed to get the new set of executable paths again. Finally, the mapping relations between the output vector and the input vector of the STL program and the cumulative conditional constraint of paths can be obtained.

An algorithm of Depth-first and multiple backtracking can be summarized as follows.

**Algorithm 1: GetCFG_Paths**

**Input**: The control flow graph \( cfg \)

**Output**: The paths CFG_Paths of control flow graph \( cfg \)

1. \( S_1 \leftarrow \text{create_stack()}, S_2 \leftarrow \text{create_stack()}, S_3 \leftarrow \text{create_stack()} \)
2. \( Q \leftarrow \text{create_queue()} \)
3 set $n_1 \leftarrow -1$
4 push($S_1$) $\leftarrow$ start, push($Q$) $\leftarrow$ start, search(start.Next)
5 for each node in $cfg$ then
6 switch node:
7 case sequential node:
8 push($S_1$), push($Q$), search(node.Next_Jump)
9 case unconditional jump node:
10 push($S_1$), push($Q$), search(node.Next)
11 case conditional jump node:
12 if node.Nnum = $n_1$ then push($S_1$), push($Q$), search(node.Next)
13 else push($S_1$), push($S_2$), push($Q$), search(node.Next)
14 end if
15 case stop node:
16 push($Q$), path $\leftarrow$ popAll($Q$), add path to $CFG_Paths$
17 if empty($S_2$) then return
18 else $n_1 \leftarrow$ pop($S_2$).Nnum
19 repeat pop($S_1$) until top($S_1$).Nnum = $n_1$
20 pushAll($S_1$) $\leftarrow$ popAll($S_1$), pushAll($S_1$) and pushAll($Q$) $\leftarrow$ popAll($S_3$)
21 search(top($S_1$).Next_Jump)
22 end
23 end switch
24 end for

As shown in Algorithm 1, firstly, stacks $S_1$, $S_2$, and $S_3$ are created. In particular, the stack $S_1$ is utilized to save the nodes passing through during traversing the control flow graph. Stack $S_2$ is utilized to save the conditional jump instruction nodes encountered during traversing the control flow graph. Stack $S_3$ is utilized to cache the intermediate nodes popped from the stack $S_1$, which are pushed onto the stack $S_1$. Moreover, a queue $Q$ is created, which is used for recording a path of the program control flow. The set of control flow paths $CFG_paths$ are created.

Secondly, stack $S_1$, stack $S_2$ and $S_3$, queue $Q$ and the path set $CFG_paths$ are initialized, where the start node of the control flow graph is pushed into the stack $S_1$. Meanwhile, the node in the queue $Q$ is recorded. Search from the “Next” field of the starting node and define $n_1 = -1$, where $n_1$ is to record the number of the condition jump node in the latest visit.

Secondly, Search from the “Next” field of the nodes until a sequential node is encountered, where the node is pushed onto the stack $S_1$ and queue $Q$. Search from the “Next_Jump” field of the node until an unconditional jump node is encountered, where the node is pushed onto the stack $S_1$ and queue $Q$. Search from the “Next” field of the node until a conditional jump node is encountered. Meanwhile, the number of the conditional jump node is compared with $n_1$. If they are equal, it indicates that the node is popped from the stack $S_2$ and searching is performed along the “Next_Jump” field of the node to the node itself. Therefore, the node is pushed onto the stack $S_1$ and queue $Q$. If the number of the condition jump node is not equal to $n_1$, the node is pushed onto the stack $S_1$, stack $S_2$ and queue $Q$. If a stop node is encountered, it indicates that a program path has been searched, the stop node is pushed onto the queue $Q$. Then all nodes are popped from the queue $Q$ and a program path can be obtained, where the path is added to the path set $CFG_paths$. If stack $S_2$ is empty, the algorithm will be terminated. Otherwise, a node is popped from the stack $S_2$ and is assigned the number of the node to $n_1$. Subsequently, the nodes in stack $S_1$ will be popped from the stack $S_1$ in order until the number of top node on stack $S_1$ is equal to $n_1$. All the remaining nodes pop from stack $S_1$ and are pushed onto stack $S_3$, finally, all the nodes pop from stack $S_3$, and are pushed onto stack $S_1$ and queue $Q$. Continue searching along the “Next_Jump” field of the node whose number is $n_1$, where the process is repeated until stack $S_2$ becomes empty.

The path set can be obtained by traversing the control flow graph in Fig. 2(c). By using Algorithm 1, the path set can be obtained as follows.
The above-described algorithm is able to traverse all branches of the control flow graph. However, it can only search for the loop path for one time. For example, there are some duplicate nodes in path 3, which constitute a loop in the control flow diagram. The algorithm only searches once for the loop paths. Therefore, it is necessary to process the loop paths.

2.4.1. Loop paths analysis

There exist some variables with varying values from the increased number of iteration for the loop path during the execution of the program. Due to the changes of these variables, the loop condition can no longer be met. Then, the execution stream of the program jumps out of the loop path. To analyze the circumstances where the execution stream of the program jumps out of the loop, it is necessary to extract these variables and analyze the changes of their values in the loop path. Consequently, the relation between these variable values and the number of cycles can be identified. And this relation should be brought into the condition of the cycle to obtain a summary of the loop statements, namely, the number of cycles for the loop and the values of these variables when the execution stream of the program jumps out of the loop path. Therefore, the loop statements can be extracted according to the loop path, and the scan cycles of loop can be obtained by calculating the summary of the loop. Finally, the loop path is expanded according to the scan cycles of loop statements. In this way, a new executable path set $C_{\text{paths}}$ can be obtained.

Subsequently, two terms should be explained before analyzing the executable path set $C_{\text{paths}}$. More specifically, they can be given as follows.

Variable defining occurrence. It means that a new value will be assigned to a variable when it appears in a basic block.

Variable usage occurrence. It means that a variable will be used in a basic block before the variable defining occurrence.

An example is given in the following code,

\[
A(\begin{array}{l}
O \quad I0.0 \\
O \quad Q4.0 \\
\end{array})
= \begin{array}{l}
AN \quad I0.1 \\
= \quad Q4.0
\end{array}
\]

In Q4.0 of the instruction, "O Q4.0" is used as usage occurrence, since the PLC program is in accordance with the pattern of input sampling stage, the implementation stage of the user program and, the output refreshing stage. For the S7 300/400 PLC, the value of Q4.0 is the value initialized by block OB100 if it is the first scan cycle of program. Otherwise, the value of Q4.0 would be assigned by block OB1 in the previous scan cycle. The appearance can be defined by Q4.0 in the instruction "= Q4.0", where its value is the output value of program execution for this cycle. This would be used in the next scan cycle of program.

The cumulative conditional constraint of the path can be obtained, and the corresponding mapping relations among the output vector, the input vector and the output vector in the previous scan cycle of program can also be obtained by analyzing every path in the set $C_{\text{paths}}$. Firstly, the data flow analysis is implemented for every path in the set $C_{\text{paths}}$. Secondly, the intermediate variable is replaced with the input variable and the usage occurrence output variable, where all usage occurrence output variables are considered as the output variables in the previous scan cycle. Finally, the cumulative conditional constraint for each path and the corresponding mapping relations can be generated among the output vector, the input vector and the output vector in the previous scan cycle of program, which
can be given as $c_{ij}(in, ob)$ and $o = \int c_{ij}(in, ob)$, respectively. In particular, $i$ represents the input vector-valued, $o$ represents the output vector-valued, and $ob$ represents the output vector-valued in the previous scan cycle.

3. Case studies
Test the PLC programs of five different scenes respectively, as follows.

Traffic lights control system: there are red, yellow and green traffic lights in four directions at road intersection. And each signal light is controlled by a Boolean output variables.

Elevator control system: the elevator carries the cage to the designated floor according to the control instructions from inside or outside of the cage. After the arrival of the designated floor, the elevator door opens and closes automatically after 10 seconds.

Stirring control system: stirring control system is connected to two inlet ports which are separately labeled A and B, and a discharge port. At the middle of the mixing car, there are three liquid level sensor switches to detect the liquid level is high, middle or low.

Water tank control system: the water tank connects a water inlet and a water outlet, and the two liquid level sensors divide the water level of the tank into low, medium and high water levels.

Sewage injection control system: the water inlet control system mainly includes the coarse grille and the cleaning machine, when the liquid level difference between the both sides of the coarse grille is more than a value, start the cleaning machine; when the liquid level difference between the both sides of the coarse grille is less than a value, turn off the cleaning machine.

These five kinds of scenes were tested before and after the program being tampered. In particular, we extracted the number of nodes in the control flow graph, the number of executable paths. The results are summarized in Tab.2 and Tab.3.

In addition, the time costs are counted for program model construction, state generation, specifications detection and generating sequence of input vector-valued, respectively. The results are presented in Tab.4.
It can be seen from Tab.4, the differences of the time spent on different processes are large for the same program, where the time spent on elevator is the longest. For the same program, the differences of the time spent on different processes are also large, where the time spent on state creation is the longest. According to Tab.2 and Tab.3, it can be seen that the time spent on executable paths generation depends on the number of nodes in the control flow graph and the number of executable paths.

At the same time, it can be seen that the time spent on the whole process is less than 2 minutes. Therefore, it can be concluded that the proposed method is practical and efficient.

4. Conclusion
In this paper, a PLC malicious code detection method is proposed, which analyze the binary program of the PLC and efficiently implement the model building of PLC binary program. However, our method is only suitable to construct PLC program model, there is no detection of PLC malwares. Therefore, the research for malicious behavior detection of PLC binary programs based on our model is implemented in future work.

Acknowledgment
This work is partially supported by the National key research and development Program of China (2016YFB0800203)

References
[1] Heng S. Industry 4.0: Upgrading of Germany's Industrial Capabilities on the Horizon[J]. Social Science Electronic Publishing, 2015.
[2] Valentine S E. PLC code vulnerabilities through SCADA systems[C]// University of South Carolina, 2013.
[3] Mclaughlin S. On dynamic malware payloads aimed at programmable logic controllers[C]// Usenix Conference on Hot Topics in Security. USENIX Association, 2011:10-10.
[4] Mclaughlin S, Medaniel P. SABOT: specification-based payload generation for programmable logic controllers[C]// ACM Conference on Computer and Communications Security. ACM, 2012:439-449.
[5] J Klick, S Lau, D Marzin, J Malchow, V Roth. Internet-facing PLCs - A New Back Orifice[C].blackhat,2015.
[6] R Spenneberg, M Brüggemann, H Schwartke.PLC-Blaster :A Worm Living Solely in the PLC[C].blackhat,2016.
[7] Darvas D, Adiego B F, Viñuela E B. Transforming PLC programs into formal models for
verification purposes[J]. 2013.

[8] Darvas D, Adiego B F, Vörös A, et al. Formal verification of complex properties on PLC programs[M]// Formal Techniques for Distributed Objects, Components, and Systems. 2016:284-299.

[9] He N, Oke V, Allen G. Model-based verification of PLC programs using Simulink design[C]// IEEE International Conference on Electro Information Technology. IEEE, 2016:0211-0216.

[10] Nie C J, Liu H F, Pu-Rui S U, et al. A Loop Summarization Method for Dynamic Binary Program Analysis[J]. Acta Electronica Sinica, 2014, 42(6):1110-1117.

[11] Xie X, Liu Y, Le W, et al. S-looper: automatic summarization for multipath string loops[C]// International Symposium on Software Testing and Analysis. ACM, 2015:188-198.

[12] Xie X, Chen B, Liu Y, et al. Proteus: computing disjunctive loop summary via path dependency analysis[C]// ACM Sigsoft International Symposium on Foundations of Software Engineering. ACM, 2016:61-72.