# A Control Injection Attack against S7 PLCs - Manipulating the Decompiled Code

Wael Alsabbagh[1,2] and Peter Langendörfer[1,2]
[1]*IHP – Leibniz-Institut für innovative Mikroelektronik, Frankfurt (Oder), Germany*
[2]*Brandenburg University of Technology Cottbus-Senftenberg, Cottbus, Germany*
E-mail: (Alsabbagh,Langendoerfer)@ihp-microelectronics.com

IECON 2021 - 47th Annual Conference of the IEEE Industrial Electronics Society | 978-1-6654-3554-3/21/$31.00 ©2021 IEEE | DOI: 10.1109/IECON48115.2021.9589721

***Abstract*—In this paper, we discuss an approach which allows an attacker to modify the control logic program that runs in S7 PLCs in its high-level decompiled format. Our full attack-chain compromises the security measures of PLCs, retrieves the machine bytecode of the target device, and employs a decompiler to convert the stolen compiled bytecode (low-level) to its decompiled version (high-level) e.g. Ladder Diagram LAD. As the LAD code exposes the structure and semantics of the control logic, our attack also manipulates the LAD code based on the attacker's understanding to the physical process causing abnormal behaviors of the system that we target. Finally, it converts the infected LAD code to its executable version i.e. machine bytecode that can run on the PLC using a compiler before pushing the malicious code back to the PLC. For a real scenario, we implemented our full attack-chain on a small industrial setting using real S7-300 PLCs, and built the database (for our decompiler and compiler) using 108 different control logic programs of varying complexity, ranging from simple programs consisting of a few instructions to more complex ones including multi functions, sub-functions and data blocks. We tested and evaluated the accuracy of our decompiler and compiler on 5 random programs written for real industrial applications. Our experimental results showed that an external adversary is able to infect S7 PLCs successfully. We eventually suggest some potential mitigation approaches to secure systems against such a threat.***

***Index Terms*—Programmable Logic Controllers (PLCs), Control Injection Attack, Decompiler, Compiler, Ladder Diagram;**

## I. Introduction

Industrial Control Systems (ICSs) are used to automate critical control processes such as production lines, electrical power grids, gas plants and others. They consist of Programmable Logic Controllers (PLCs) which are directly connected to the physical processes. They are equipped with control logic that defines how to monitor and control the behavior of the processes. Thus, their safety, durability, and predictable response times are the primary design concerns. PLCs are offered by several vendors such as Siemens, Allen-Bradley, Schneider, etc. Each vendor has its own proprietary firmware, programming, communication protocols and maintenance software. However, the basic hardware and software architecture is similar, meaning that all PLCs contain variables, and logic to control their inputs and outputs. The PLC code is written on an engineering station in the vendor's control logic language. The control logic is then compiled into an executable format, and downloaded to the PLC. Unfortunately, the security features are largely absent in ICS components or ignored/disabled because security is often at odds with operations. Therefore, thousands of PLCs are directly reachable from the internet. Although only one PLC may be reachable from outside, this exposed PLC is likely to be connected to internal networks e.g. via PROFINET with many more PLCs [1]. This is what is called the deep industrial network, therefore attackers can leverage an exposed PLC to extend their access from the internet to the deep industrial network.

Stuxnet [2] is perhaps the most well-known attack on ICSs. This malware used a windows PC to target Siemens S7-300 PLCs that are specifically connected with variable frequency drives. It infects the control logic of the PLCs to monitor the frequency of the attached motors, and only launches an attack if the frequency is within a certain normal range (i.e. 807 Hz and 1,210 Hz). Another attacks on PLCs have been already conducted in the last decade. Most of them aimed at modifying the control logic in its compiled version e.g. MC7 bytecode for Siemens and RX630 bytecode for Schneider. In contrast, our attack manipulates the control logic program in its high-level format, precisely in its LAD format. We choose LAD over the other programming languages because LAD is a graphical language where each instruction is represented as a graphical symbol and the instructions are grouped into networks which makes reading and understanding the control logic program in LAD format very easy even for non-experts.

We also focus in this paper, as a part of our full attack-chain, on the capability of employing 1) a decompiler to obtain the LAD code from the stolen machine bytecode over the network, and 2) a compiler to recompile the infected LAD code into MC7 machine bytecode that the PLC can read. We evaluate the accuracy of our decompiler and compiler on 5 different control logic programs (chosen randomly). Finally, we performed our full attack on a real industrial example application based on S7 300 PLCs and TIA Portal software (see figure 1).

Please note that compromising the ICS network is out of the scope of this work and can be achieved via typical attack vectors in our IT world such as infected USB, vulnerable web server, etc. Our attack scenario is network based, and can be successfully launched by any attacker with a network access to the target PLC. However, finding PLCs connected directly to the Internet is an easy task using search engines such as Shodan, Censys, etc.

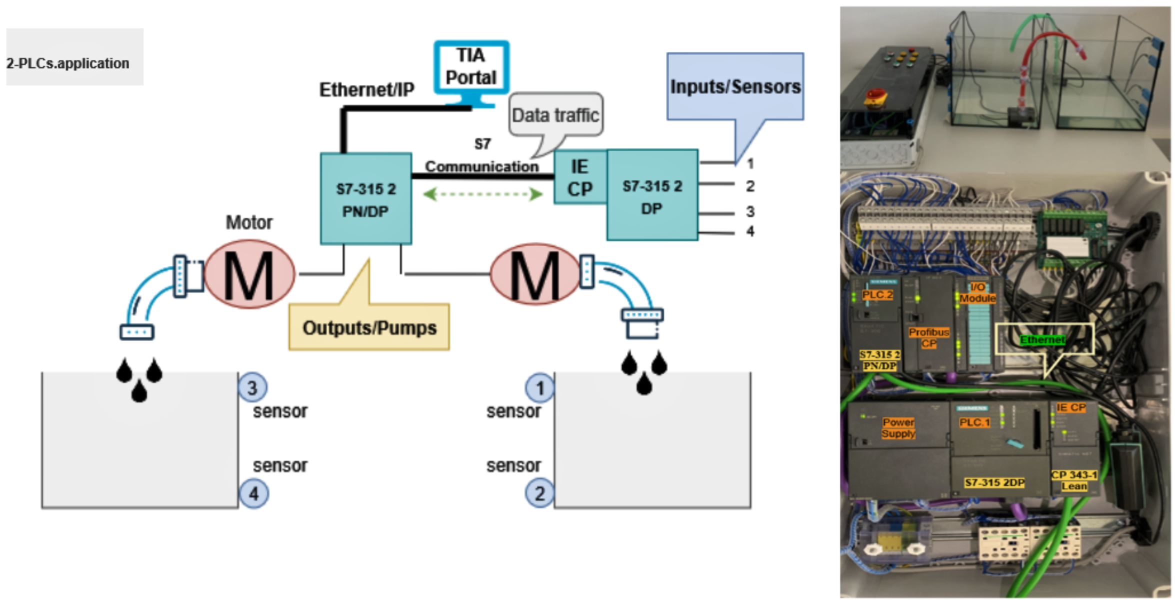


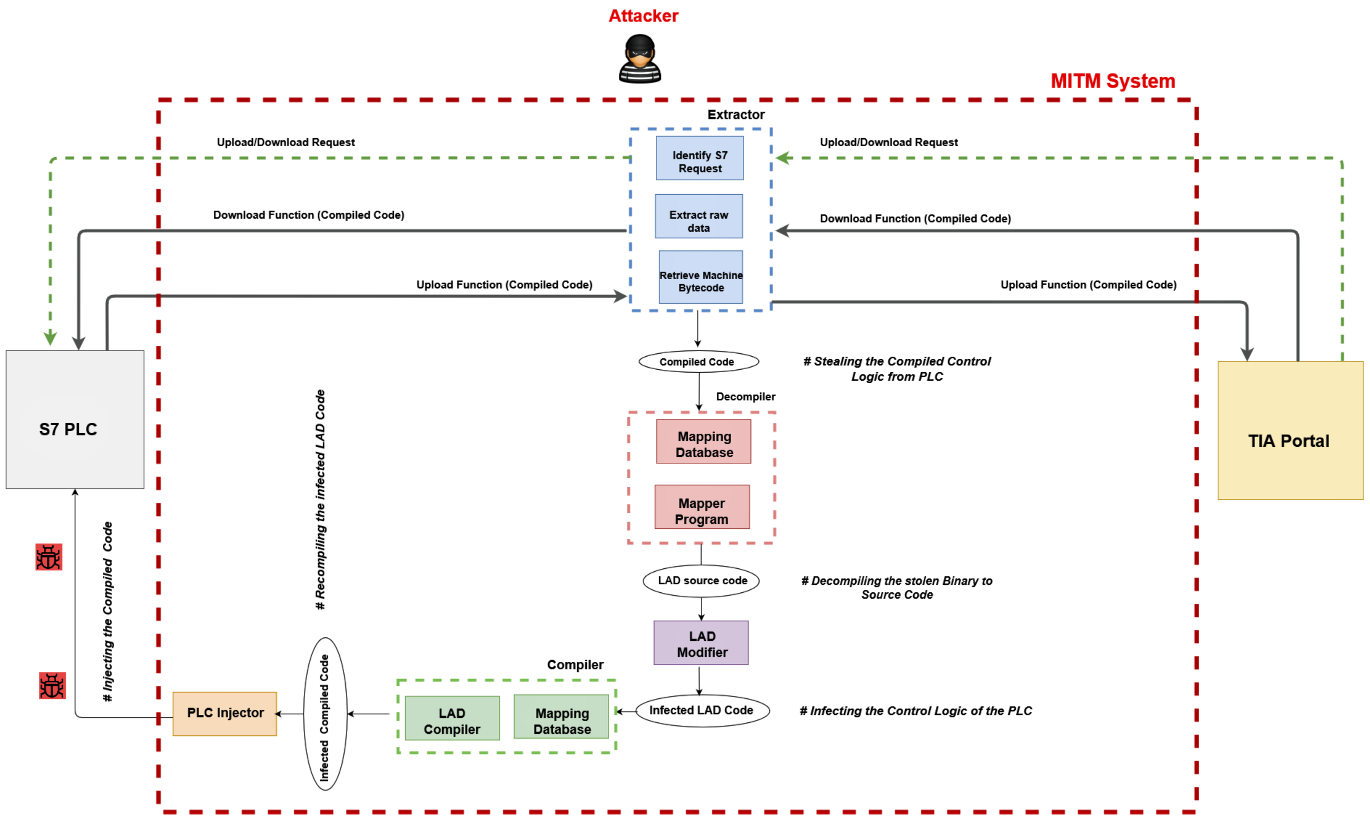
Fig. 1: Example application of our control process

The rest of the paper is organized as follows. Section II discusses related work, while our experimental setup is presented in section III. We illustrate our attack scenario in IV, and evaluate our decompiler and compiler in V, and section VI concludes this paper.

## II. Related Work

In the recent years, many vulnerabilities aimed at modifying control logic source code, by exploiting the engineering station [3], or by leveraging Ethernet design flaw and then using crafted packets to delete control logic programs [4], [5]. Other vulnerabilities modify the control logic at runtime, compromising firmware and authentication flaws, and triggering PLC fault states to overwrite the control logic [6], [7]. However, as real scenario attacks targeted ICSs, we can mention the ones that occurred in Ukraine [8], [9], and in Germany [10]. These attacks caused severe control distributions in the target facilities and a massive damage in the physical systems controlled by the PLCs attacked. In the following, we compare our approach to previous published efforts that focused on exploiting the control logic code of PLCs.

In 2018, A Ladder Logic Bomb malware written in ladder logic or one of the compatible languages was introduced in [11]. Such malware is inserted by an attacker into existing control logic on PLCs. Anyway, this scenario requires from the adversary to be familiar with the programming languages that the PLC is programmed with beforehand, which is not a common case for a real scenario. Another group of researchers presented a remote attack on the control logic of PLCs in [12]. They were able to infect the PLC and to hide the infection from the engineering software at the control center. They implemented their attack on Schneider Electric Modicon M221 PLCs, and its vendor-supplied engineering software (SoMachine-Basic). In opposite to their work, our attack allows the attacker to modify the control logic program in its high-level format, and on the wish of the attacker. Furthermore, we use S7 PLCs provided by Siemens which transfer the machine bytecode over S7 packets. At black Hat USA 2015 Klick et al. [13] demonstrated injection of malware into the control logic of a Simatic S7-300 PLC, without disrupting the service. The modification process of their attack is also done on the machine bytecode level. In a follow on work, Spenneberg et al [14] presented a PLC worm. The worm spreads internally from one PLC to other target PLCs. During the infection phase, the worm scans the network for new targets (PLCs). The authors hided the infected code in an organization block (OB9999) and then transferred from a PLC to another using their worm. Their attack manipulates the control logic of S7 PLCs successfully, but it is written with constraints such as the maximum cycle on one hand, and on other hand did not decompile the machine bytecode which required that the attacker has a TIA Portal installed on his machine. In 2021, we, in a former work, presented a stealthy injection attack on the control logic of S7 PLCs [15]. Our attack introduced a malicious logic in a target PLC. As a part of our attack scenario, we implemented an initial decompiler that takes the machine bytecode as an input and decompiles it into Statement List (STL) source code. Our decompiler used in [15] was very limited to only a few instructions, and utilized a small database that consists of 56 entries. In this work, we develop our mapping database to involve 3802 entries, 34 LAD instructions including inputs, outputs, function blocks, data blocks, organization blocks, timers, counters, etc. Moreover, our new approach allows an adversary to modify the control logic in its high-level code i.e. LAD format, and recompiles the infected code to its machine code again, using a compiler before pushing it back to the target PLC.



Attacker
MITM System
Extractor
Identify S7 Request
Extract raw data
Retrieve Machine Bytecode
Upload/Download Request
Download Function (Compiled Code)
Upload Function (Compiled Code)
Upload/Download Request
Download Function (Compiled Code)
Upload Function (Compiled Code)
S7 PLC
TIA Portal
Compiled Code
# Stealing the Compiled Control Logic from PLC
Decompiler
Mapping Database
Mapper Program
LAD source code
# Decompiling the stolen Binary to Source Code
LAD Modifier
Infected LAD Code
# Infecting the Control Logic of the PLC
Compiler
LAD Compiler
Mapping Database
Infected Compiled Code
PLC Injector
# Recompiling the infected LAD Code
# Injecting the Compiled Code

Fig. 2: High-level overview of our proposed attack scenario

## III. Experimental Set-up

In this section, we describe our experimental set-up used to test our full-chain attack presented in this paper. As shown in figure 1 (please note that we also used this setup in experiments run in our earlier publications [15], [16], [20]), there are two aquariums filled with water that is pumped from one to the other until a certain level is reached and then the pumping direction is inverted. The control process in this set-up is cyclically running as follows: PLC.1 (S7 315-2DP) reads the input signals coming from the sensors 1, 2, 3 and 4. The two upper sensors (Num. 1, 3) installed on both aquariums are reporting to PLC.1 when the aquariums are full. While the two lower sensors (Num. 2, 4) are reporting to PLC.1 when the aquariums are empty. After that, PLC.1 sends the sensors' readings to PLC.2 (S7 315-2 PN/DP) using an industrial Ethernet Communication Processor (IE-CP 343-1 Lean). Then PLC.2 powers the pumps on/off depending on the sensors' readings received from PLC.1.

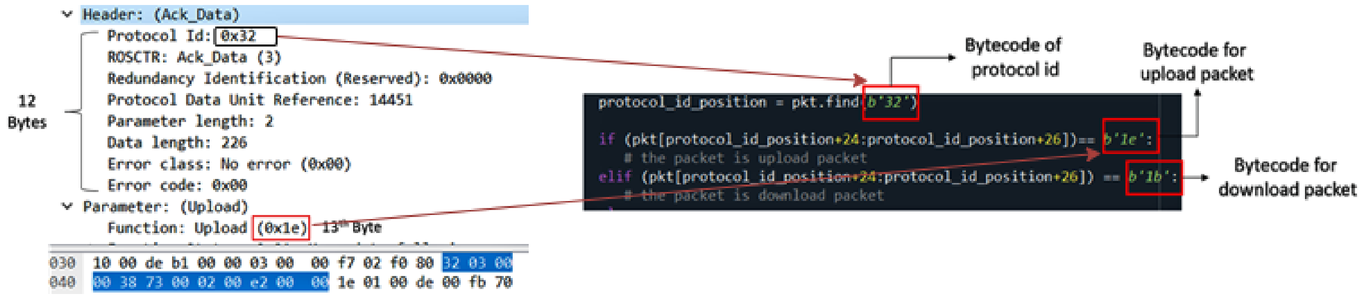
## IV. Attack Description

In this Paper, we present a full attack-chain on the control logic of an S7 PLC. We assume a realistic attack scenario where the TIA Portal software in the engineering station is not reachable for an attacker, thereby making our attack more challenging. After the attacker penetrated the system network, and can send/receive messages to/from a target PLC, he launches our MITM system presented in [15] between the engineering station (TIA Portal software) and the field side (S7 PLCs), so he is able to listen and record all the network traffic exchanged between the stations using Wireshark software. Figure 2 shows a high-level overview of our proposed control injection attack. It consists of six main phases:

1- Compromising the PLC security measures. In this work we skip this step as it is already achieved and illustrated in details in our former papers [15], [16], and focus only on the following phases.
2- Stealing the compiled machine bytecode program from the target PLC.
3- Decompiling the bytecode representation of the stolen control logic into its high-level source code (LAD code).
4- Modifying the control logic in its decompiled format by replacing/removing/adding entries from/to the original code.
5- Recompiling the infected code into its low-level representation (that can run on the PLC).
6- Pushing the infected machine bytecode back to the PLC.

### A. Extractor - Stealing the Machine Bytecode from the PLC

*1) Identify S7Comm requests:* As the PLC only sends/receives the control logic program by processing either upload or download request that sent from the engineering station, our extractor first determines all S7Comm packets



```
Header: (Ack_Data)
    Protocol Id: 0x32
    ROSCTR: Ack_Data (3)
    Redundancy Identification (Reserved): 0x0000
    Protocol Data Unit Reference: 14451
    Parameter length: 2
    Data length: 226
    Error class: No error (0x00)
    Error code: 0x00
Parameter: (Upload)
    Function: Upload (0x1e)

030  10 00 de b1 00 00 03 00  00 f7 02 f0 80 32 03 00
040  00 38 73 00 02 00 e2 00  00 1e 01 00 de 00 fb 70
```

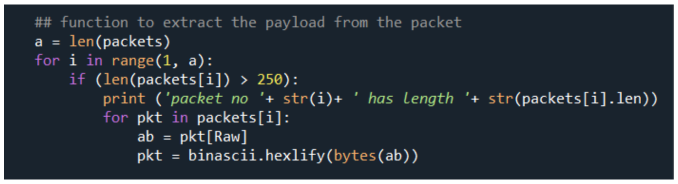
```
protocol_id_position = pkt.find(b'32')

if (pkt[protocol_id_position+24:protocol_id_position+26])== b'1e':
   # the packet is upload packet
elif (pkt[protocol_id_position+24:protocol_id_position+26]) == b'1b':
   # the packet is download packet
```

Fig. 3: Identify an S7 request's functionality

exchanged by checking the packet header, precisely the protocol ID (*0x32*) which is unique for S7Comm frames, and then reads the 13th byte which assigns the functionality of the S7 command request see figure 3. This byte is always set at *0x1e* and *0x1b* for upload and download request respectively.

*2) Extract the data payload:* After the PLC receives an upload/download request from the engineering station, it responds by sending either its code to the engineering station for an upload request, or an acknowledgement packet informing the user that it is ready to receive the code of a download request; and then the engineering station starts downloading the code into the connected PLC. In this step our extractor records all the response stream for any identified upload/download request which eventually contains the bytecode program that the PLC runs. As the network stream consists of different packets e.g. setup communication, job function, block start, block process, block end, etc. our extractor needs first to filter the stream keeping only the exact S7 packet that the bytecode is transferred with, ignoring the rest of the packets.

Our investigation shows that the S7comm packet that the PLC machine bytecode is existing in, has always a larger size than the others, precisely larger than 250 bytes. This is due to the fact that for a very simple control logic that comprises of only one LAD network i.e. only one input and one output, the size of the S7 packet that transfers the program is 254 bytes. Therefore to ensure a successful extraction, our extractor records and saves only the S7 packets that have a size larger than 250 bytes. Please note that the size of the machine bytecode differs significantly from each other depending on the complexity and the number of instructions and networks involved in the program, but as we set the filtering process at the minimum size that a PLC program might have, for the

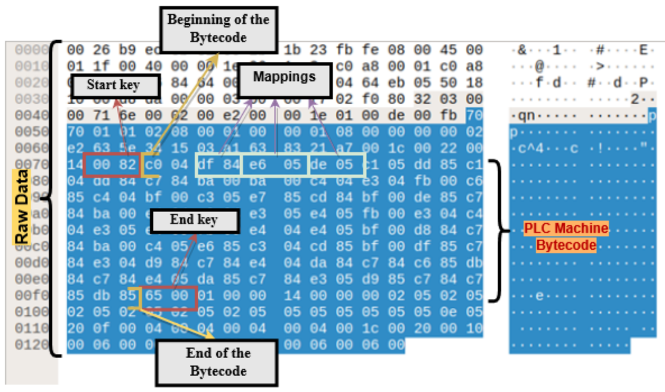


```
## function to extract the payload from the packet
a = len(packets)
for i in range(1, a):
    if (len(packets[i]) > 250):
        print ('packet no '+ str(i)+ ' has length '+ str(packets[i].len))
        for pkt in packets[i]:
            ab = pkt[Raw]
            pkt = binascii.hexlify(bytes(ab))
```

Fig. 4: Extract the data payload from S7Comm packets

example application given in section III, our extractor could successfully filter the network stream to retrieve all S7Comm packets that eventually transfer the machine bytecode. Figure 4 shows the snippet of python code that our extractor uses to filter the network stream and then to extract the raw data from the S7Comm packet. It is worth mentioning that the extracted raw data is in assembly format. Meaning that we still need to convert it into bytecode format for further computing. This is done by utilizing the binascii.hexlify function (as shown in the last line of figure 4).

*3) Retrieve the Machine Bytecode from the raw data:* Once our extractor has obtained the raw data (that contains the machine bytecode), it then filters this data to retrieve only the machine bytecode which eventually represents the low-level program that the PLC runs. Our findings show that the machine bytecode is always located between two bytecode keys in the extracted raw data: Start key *0x0082*, and End key *0x6500* as shown in figure 5.



```
0000  00 26 b9 ed [illegible]  1b 23 fb fe 08 00 45 00
0010  01 1f 00 40 00 00 1e [illegible]  c0 a8 00 01 c0 a8
0020  [illegible] 84 64 00 [illegible]  04 64 eb 05 50 18
0030  [illegible] 00 00 03 [illegible]  02 f0 80 32 03 00
0040  00 71 6e 00 02 00 e2 00  00 1e 01 00 de 00 fb 70
0050  70 01 01 02 08 00 01 00  00 01 08 00 00 00 00 02
0060  e2 63 5e 34 15 03 a1 63  83 21 a7 00 1c 00 22 00
0070  14 00 82 c0 04 df 84 e6  05 de 05 c1 05 dd 85 c1
0080  04 dd 84 c7 84 ba 00 ba  00 c4 04 e3 04 fb 00 c6
0090  85 c4 04 bf 00 c3 05 e7  85 cd 84 bf 00 de 85 c7
00a0  84 ba 00 [illegible] e3  05 e4 05 fb 00 e3 04 c4
00b0  04 e3 05 [illegible] e4  04 e4 05 bf 00 d8 84 c7
00c0  84 ba 00 c4 05 e6 85 c3  04 cd 85 bf 00 df 85 c7
00d0  84 e3 04 d9 84 c7 84 e4  04 da 84 c7 84 c6 85 db
00e0  84 c7 84 e4 05 da 85 c7  84 e3 05 d9 85 c7 84 c7
00f0  85 db 85 65 00 01 00 00  14 00 00 00 02 05 02 05
0100  02 05 02 [illegible] 05 02 05  05 05 05 05 05 05 0e 05
0110  20 0f 00 04 [illegible] 04 00 04  00 04 00 1c 00 20 00 10
0120  00 06 00 0[illegible]  00 06 00 06 00
```

Fig. 5: Raw Data of S7Comm packet captured by Wireshark

By extracting only the bytes located between these two bytecode keys, we managed successfully to get the machine bytecode which is the input for our decompiler in the next step.

## *B. Decompiler - decompiling the Machine Bytecode to Ladder Diagram*

Siemens provides its TIA Portal software for engineers to program PLCs in Ladder Diagram (LAD), Function Block Diagram (FBD), Structured Control Language (SCL), and Statement List (STL). In contrast to the text-based SCL and assembler-like STL, the LAD and FBD languages are graphical. The Ladder Diagram (LAD) consists of networks, each has elements e.g. inputs and outputs. Figure 6 shows a simple network example that has eight input entries, one output entry, and three parallel branches.

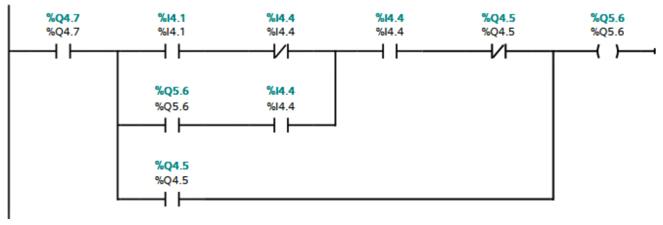


Fig. 6: An example of Ladder Diagram network in TIA Portal

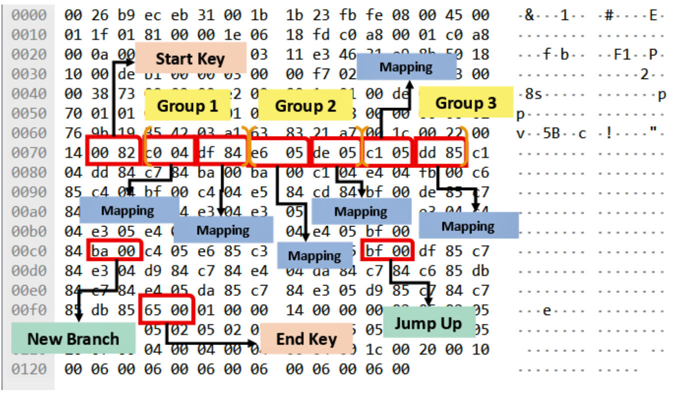
The TIA Portal software compiles the control logic program converting it from its high-level format e.g. LAD version into machine bytecode i.e. MC7 bytecode that the PLC can read and process. In this work, we developed a decompiler that takes the machine bytecode obtained from our extractor as an input and converts it into LAD source code. Our decompiler comprises of two main components: First, the database for mapping each hex-bytes set to their corresponding LAD entries. Second the mapper program, which utilizes the entries found to generate the final LAD source code. Please note that we mean by instructions: inputs, outputs, memory bits, function blocks, data blocks, timers, counters, etc. while entries mean the different types of instructions such as Boolean, bytes, word, double word, etc. e.g. %I0.1, %Q1.1, %DW0.1, etc.

*1) Mapping Database:* To create our mapping database, we needed to collect a good number of pairs: hex-bytes to their corresponding LAD entries. This was done by mapping 108 different control logic programs of varying complexity, ranging from simple programs consisting of a few inputs and outputs to more complex ones including multi inputs, outputs, function blocks, data blocks, organization blocks, timers, counters, etc. all the programs are written for different real physical processes such as traffic light, gas pipe, water tank, etc.

Creating our mapping database is done by applying an offline method as follows: we cleared the PLC memory and then opened the TIA Portal software, and programmed the PLC with a certain LAD code containing only a single network consisting of 10 times of the same entry. Here we used 10 times of the input %I0.0. Due to the fact that each LAD network must have at least one output located at the end, we concatenated the 10 inputs with %Q0.0 as an output to close the network. After that, we downloaded this program into the PLC and used our extractor to retrieve the machine bytecode representing the low-level code of this single LAD network. We could identify that an %I0.0 entry is always mapped to *0xC000* in the bytecode format. Afterwards we cleared the PLC memory again and opened each control logic program used in our experiments in the TIA Portal software separately, and inserted the %I0.0 input before and after each entry taking into account that each LAD network must be closed with an output. Then we downloaded each new modified program into the PLC, and retrieved the machine bytecode of each using our extractor. We eventually identified each pair of hex-bytes to its corresponding entry in LAD code. After repeating this process for all 108 programs, we managed to create a mapping database consisting of 3802 entries for 34 different instructions.

*2) Mapper Program:* Our mapper program reads the output of the extractor (the stolen machine bytecode) and calls then the mapping database to identify the corresponding LAD entries. It works in four steps as shown in figure 7.

1- it divides the entire bytecode into smaller groups of hex-bytes using rules-based approach i.e. it takes all the hex-bytes in one group until an output is reached. Then a new group of hex-bytes starts until reaching another output, and so on.
2- it divides each group into sets of hex-bytes, each represents a potential mapping (entry) in the LAD code.
3- it then compares each set of hex-bytes in each group to the pairs in our database.
4- After a successful decompiling, our program generates the LAD code using the following rules seen in figure 8:

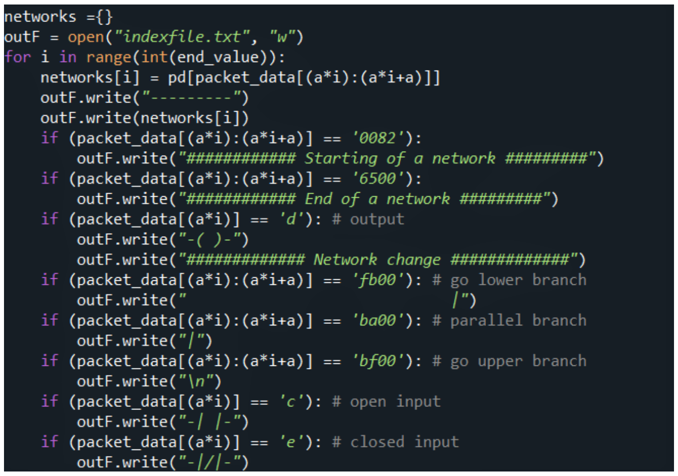


```
0000  00 26 b9 ec eb 31 00 1b  1b 23 fb fe 08 00 45 00   ·&···1·· ·#····E·
0010  01 1f 01 81 00 00 1e 06  18 fd c0 a8 00 01 c0 a8   ········ ········
0020  00 0a 00 [illegible] 03  11 e3 46 [illegible] 18   ···f·b·· ··F1··P·
0030  10 00 de [illegible] 00  00 f7 02 [illegible] 00   ········ ······2··
0040  00 38 73 [illegible] 00 de [illegible]             ·8s····· ·······p
0050  70 01 01 [illegible] 00 00 [illegible]             p······· ········
0060  76 9b 19 [illegible] 63  83 21 a7 00 1c 00 22 00   v··5B··c ·!····"·
0070  14 00 82 c0 04 df 84 e6  05 de 05 c1 05 dd 85 c1   ········ ········
0080  04 dd 84 c7 84 ba 00 ba  00 c1 04 e4 04 fb 00 c6   ········ ········
0090  85 c4 04 bf 00 c4 04 e5  84 cd 84 bf 00 de 85 c7   ········ ········
00a0  84 [illegible] 4 e3 04 e3  05 [illegible]          ········ ········
00b0  04 e3 05 e4 [illegible]  04 e4 05 bf 00 [illegible] ········ ········
00c0  84 ba 00 c4 05 e6 85 c3  [illegible] bf 00 df 85 c7 ········ ········
00d0  84 e3 04 d9 84 c7 84 e4  04 da 84 c7 84 c6 85 db   ········ ········
00e0  84 c7 84 e4 05 da 85 c7  84 e3 05 d9 85 c7 84 c7   ········ ········
00f0  85 db 85 65 00 01 00 00  14 00 00 00 [illegible] 05 ···e···· ········
[illegible] 05 02 05 02 0 [illegible] 05 [illegible] 05  ········ ········
[illegible] 04 00 04 00 0 [illegible] 1c 00 20 00 10     ········ ·· ··
0120  00 06 00 06 00 06 00 06  00 06 00 06 00            ········ ·····
```

Fig. 7: Our approach used in the mapper program

1- start a new network until an output is reached.
2- create a new parallel branch at the current entry being decompiled in case a parallel entry is found which is mapped to *0xba00* in bytecode. This new branch includes all the following hex-bytes until a jump entry is reached.
3- end the current branch and jump back down to the entry where *0xba00* is located, in case *0xfb00* is found.

4- end the current branch and jump back up to the entry where *0xba00* is located, in case *0xbf00* is found.



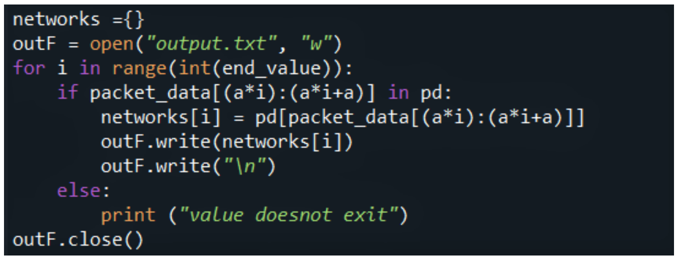
```
networks ={}
outF = open("indexfile.txt", "w")
for i in range(int(end_value)):
    networks[i] = pd[packet_data[(a*i):(a*i+a)]]
    outF.write("---------")
    outF.write(networks[i])
    if (packet_data[(a*i):(a*i+a)] == '0082'):
        outF.write("############ Starting of a network #########")
    if (packet_data[(a*i):(a*i+a)] == '6500'):
        outF.write("############ End of a network #########")
    if (packet_data[(a*i)] == 'd'): # output
        outF.write("-( )-")
        outF.write("############# Network change #############")
    if (packet_data[(a*i):(a*i+a)] == 'fb00'): # go lower branch
        outF.write("                              |")
    if (packet_data[(a*i):(a*i+a)] == 'ba00'): # parallel branch
        outF.write("|")
    if (packet_data[(a*i):(a*i+a)] == 'bf00'): # go upper branch
        outF.write("\n")
    if (packet_data[(a*i)] == 'c'): # open input
        outF.write("-| |-")
    if (packet_data[(a*i)] == 'e'): # closed input
        outF.write("-|/|-")
```

Fig. 8: Rules used in our Mapper Program to generate the final LAD code

Figure 9 shows a part of control logic program decompiled by our LAD decompiler compared to the original one displayed in the TIA Portal software.

## C. Modifier - Modifying the Ladder Diagram

As it is easier for an attacker to parse and manipulate the control logic for a LAD code than machine bytecode, our attack allows the adversary to modify the code in its LAD format on his wish i.e. he can modify/inject/delete entries or even networks based on his understanding of the exposed physical process. The modification in this attack is done by using our LAD modifier. Its functionality is to save all the entries used in the PLC program in a text file, precisely in an instructions list. So all what the attacker needs to modify the PLC program is to open the text file and easily to manipulate the entries listed in the instructions list. Figure 10 shows a snippet of the python code of our modifier.



```
networks ={}
outF = open("output.txt", "w")
for i in range(int(end_value)):
    if packet_data[(a*i):(a*i+a)] in pd:
        networks[i] = pd[packet_data[(a*i):(a*i+a)]]
        outF.write(networks[i])
        outF.write("\n")
    else:
        print ("value doesnot exit")
outF.close()
```

Fig. 10: Snippet code for generating instructions list in a text file

As seen, all the decompiled entries that the current PLC program uses are saved in an editable text file (output.txt), and due to the fact that the attacker is already familiar with the physical process from the previous step, he could modify the instructions list causing damage in the target system. Figure 11 shows an example of modifying the instruction list in the resulting text file. We replaced the output %Q4.7 with %Q4.1, and the input %I5.6 with the output %Q5.3.

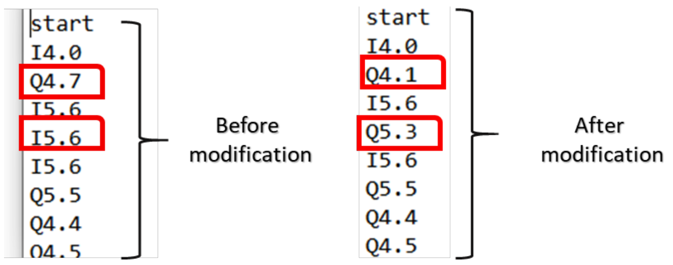


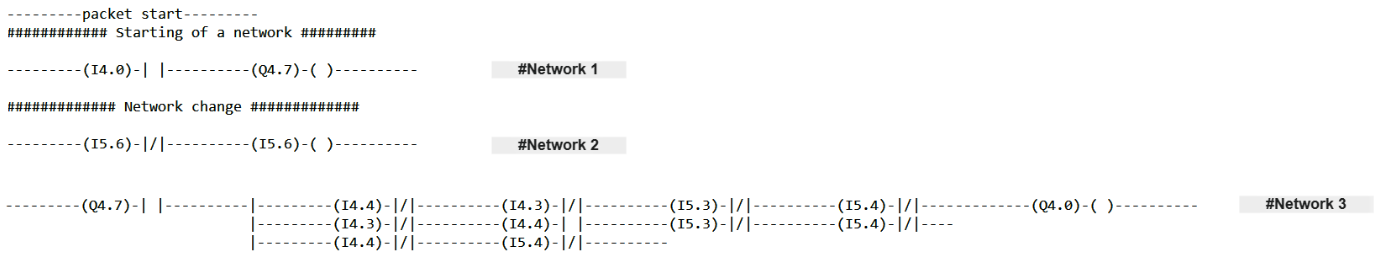
Fig. 11: Modifying the instructions list in the text file

## D. Compiler - Compiling the infected LAD source code to machine bytecode

After a successful modification of the control logic, we need to recompile the infected LAD code to machine bytecode before pushing it back to the target PLC. For achieving this, we designed a compiler which works similar to our above-mentioned decompiler but in a reverse process. It uses the same mapping database to get the equivalent hex-bytes of each entry in the LAD code. Our compiler reads the resulting output of the modifier (the infected LAD code that was saved in a text file), and then calls the mapping database and recompiles the entries into their corresponding hex-bytes. Figure 12 shows the output of our compiler which is the infected machine bytecode that we want the PLC to read and process.

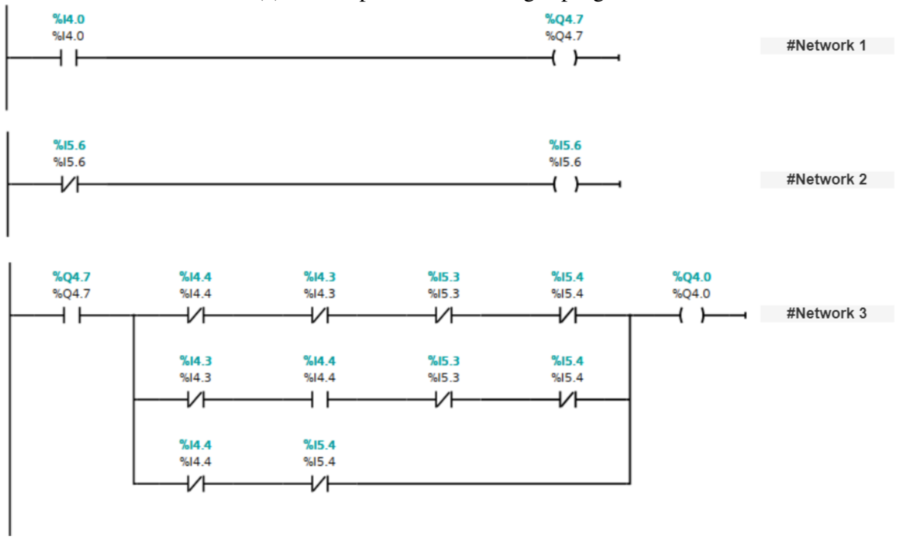
## E. Injector - Infecting the PLC

In the final step, an attacker has already the malicious code in its machine bytecode format, and all needed to corrupt the system is to push the infected control logic back to the PLC. Due to the lack of integrity checks in S7-300 PLCs, such controllers execute commands whether or not are delivered from a legitimate user. Therefore, our PLC injector crafts the full S7Comm packet that we want to send to the PLC by placing the malicious machine bytecode (obtained from the previous step) in as raw data and then adding the parameters and the proper S7 packet header. In this work, our PLC injector uses the same S7 packet that our extractor already identified (see section A.2) and replaces only the original machine bytecode located between the start and end keys with the malicious one (the output of our compiler). Afterwards it injects the crafted packet into the PLC using the well-known Python Snap7 library, precisely function Cli_Download as done in our former work [16]. For our example application given in section III, we managed successfully to alter the physical process controlled by the infected PLC causing a water overflow.



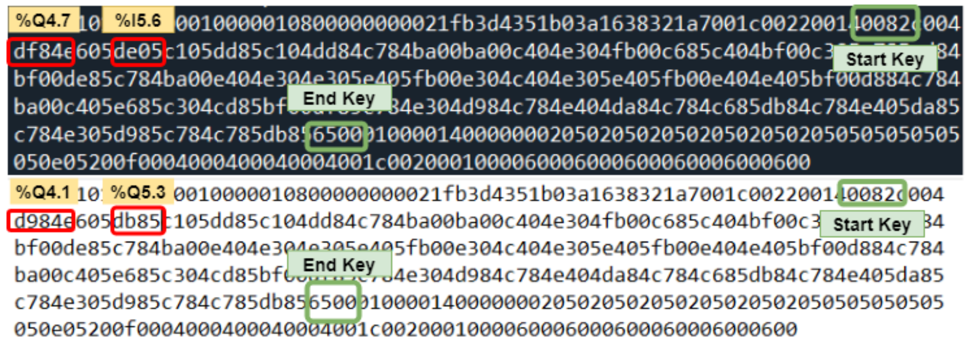
```
---------packet start---------
############ Starting of a network #########

---------(I4.0)-| |----------(Q4.7)-( )----------          #Network 1

############# Network change #############

---------(I5.6)-|/|----------(I5.6)-( )----------          #Network 2


---------(Q4.7)-| |----------|---------(I4.4)-|/|----------(I4.3)-|/|----------(I5.3)-|/|----------(I5.4)-|/|-------------(Q4.0)-( )----------          #Network 3
                             |---------(I4.3)-|/|----------(I4.4)-| |----------(I5.3)-|/|----------(I5.4)-|/|----
                             |---------(I4.4)-|/|----------(I5.4)-|/|----------
```

(a) Decompiled Control Logic program



(b) Original Control Logic displayed on TIA Portal software

Fig. 9: LAD code displayed on the attacker machine and TIA Portal respectively



```
%Q4.7 10 %I5.6 001000000108000000000021fb3d4351b03a1638321a7001c002200140082c004
df84e605de05c105dd85c104dd84c784ba00ba00c404e304fb00c685c404bf00c3[illegible]84
bf00de85c784ba00e404e304e305e405fb00e304c404e305e405fb00e404e405bf00d884c784
ba00c405e685c304cd85bf[illegible]4e304d984c784e404da84c784c685db84c784e405da85
c784e305d985c784c785db856500010000140000000205020502050205020502050505050505
050e05200f00040004000400040001c002000100006000600060006000600600
```

```
%Q4.1 10 %Q5.3 001000000108000000000021fb3d4351b03a1638321a7001c002200140082c004
d984e605db85c105dd85c104dd84c784ba00ba00c404e304fb00c685c404bf00c3[illegible]84
bf00de85c784ba00e404e304e305e405fb00e304c404e305e405fb00e404e405bf00d884c784
ba00c405e685c304cd85bf[illegible]4e304d984c784e404da84c784c685db84c784e405da85
c784e305d985c784c785db856500010000140000000205020502050205020502050505050505
050e05200f00040004000400040001c002000100006000600060006000600600
```

Fig. 12: The original and malicious bytecode respectively

## V. Evaluation of Decompiler and Compiler

To assess the accuracy of our decompiler and compiler, we downloaded 5 programs randomly[1] to an S7-300 PLC, captured their network traffic and then extracted them from the traffic using our extractor. Afterwards, we run our decompiler to decompile the programs into their LAD source codes, and then compared the decompiled and original LAD code to measure the accuracy of the decompiler. For evaluating the accuracy of our compiler, we also recompiled the decompiled version back to its machine bytecode and compared both (recompiled and original) versions. The experimental results presented in table 1 show that our decompiler and compiler work 100% correct.

[1]https://instrumentationtools.com

## VI. Discussion, Security Recommendations, and Future Work

We presented an advanced control logic injection attack for altering the program running in an S7 PLC to disrupt physical processes controlled by the compromised device. Our full attack-chain, including security measures exploitation, decompilation, Compilation, high-level code modification, and PLC injection, was implemented on a real industrial setting,

TABLE 1: The accuracy of the decompiler and compiler

| **Control Logic Program** | **Entries in TIA Portal** | **Entries in Decompiler** | **Entries in Compiler** | **Accuracy** |
|---|---|---|---|---|
| Bottle Detection | 7 | 7 | 7 | 100% |
| Automatic Mixing controlling in a Tank | 11 | 11 | 11 | 100% |
| Temperature control using pulse width modulation | 16 | 16 | 16 | 100% |
| Car parking | 16 | 16 | 16 | 100% |
| Fan control unit System for industry | 17 | 17 | 17 | 100% |

precisely on real S7 300 PLCs, and engineering software TIA Portal. As a part of this work, we used 108 different logic control programs to create a sufficiently large mapping database that consists of 3802 entries for 34 different instructions. Our decompiler and compiler were tested and evaluated for 5 different real industrial control logic programs. The experimental results show that our attack scenario managed successfully to alter the program running in the compromised PLC causing a water overflow for the example application used in this work.

From a security point of view, we highly suggest some countermeasures to our attack such as protection and detecting of control logic. The first step to protect our systems from various sort of attacks is to improve the isolation from other networks [17], combining this with standard security practices [18], and even defence-in-depth security in the control systems [19]. In addition, a digital signature should be employed not only to the firmware as most of the PLC vendors do but also to the control logic. Furthermore, a mechanism to check the protocol header which contains information about the type of the payload is also recommended as a solution to detect and block any potential unauthorized transfer of the control logic. Finally, Siemens provides the users with an MPI adaptor to upload and download the control logic between the TIA Portal and PLC safely. The MPI Protocol is so far not supported by any network sniffers. Taking into account the benefits of using Ethernet/Profinet connections related to cost and convenience, the MPI connection still provides a better secure communication between the control center and the remote devices. This helps to prevent attackers from snooping which in turn improves security as listening and capturing packets transferred over the network is the main base for attackers to perform most of the attacks against ICSs.

The exploit in this paper is efficient but not at all complicated as S7-300 PLCs still use the old version of S7 protocol which lacks of security mechanisms compared to the newer version (S7Comm Plus) that the modern S7 PLCs e.g. S7-1200 and S7-1500 PLCs use. So in our future work we will investigate if our control injection attack can be run successfully against the modern S7 PLCs. We are aware of the fact that this will be more challenging as S7comm plus protocol supports improved security implementing anti-replay mechanisms and integrity checks.

## References

[1] W. Alsabbagh and P. Langendörfer, “A Fully-Blind False Data Injection on PROFINET I/O Systems,“ Proc. 30th International Symposium on Industrial Electronics (ISIE 2021).

[2] N. Falliere. Exploring stuxnet’s PLC infection process, Sept. 2010.

[3] CVE-2010-5305. https://nvd.nist.gov/vuln/detail/CVE-2017-12088.

[4] CVE-2017-12088. https://nvd.nist.gov/vuln/detail/CVE-2017-12088.

[5] CVE-2019-10929. https://nvd.nist.gov/vuln/detail/CVE-2019-10929.

[6] CVE-2017-14468. https://nvd.nist.gov/vuln/detail/CVE-2019-14468.

[7] TALOS-2017-0443. https://talosintelligence.com/vulnerabilitiy_reports/TALOS-2017-0443.

[8] G. liang, S. R. Weller, J. Zhao, F. Luo, and Z.Y. Dong, "The 2015 Ukraine blackout: Implications for false data injection attacks," IEEE Transactions on Power Systems, 2016, doi: 10.1109/TPWRS.2016.2631891.

[9] R. M. Lee, M. J. Assante, and T. Conway, "Analysis of the cyber attack on the Ukrainian power grid," Technical report, SANSE-ISAC, March 18 2016. Available at: https://ics.sans.org/media/ESAC_SANS_Ukraine_DUC_5. Pdf.

[10] T. De Maizière. Die Lage Der IT-Sicherheit in Deutschland, The German Federal Office for Information Security,2014. https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/Lageberichte/ Lagebericht2014.pdf

[11] N. Govil, A. Agrawal, N. O. Tippenhauer, "On Ladder Logic Bombs in Industrial Control Systems," January, 2018, dio: 10.1007/978-3-31972817-9_8.

[12] K. Sushma, A. Nehal, Y. Hyunguk, and A. Irfan, "CLIK on PLCs! Attacking Control Logic with Decompilation and Virtual PLC," 2019, dio: 10.14722/bar.2019.23xxx.

[13] J. Klick, S. Lau, D. Marzin, J. Malchow, and V. Roth, "Internet-facing PLCs-a new back orifice," In Blackhat USA 2015, Las Vegas, USA, 2015.

[14] A. Spenneberg, M. Brüggemann, and H. Schwartke, "PLC-blaster: A worm living solely in the PLC," in Black Hat Asia, Marina Bay Sands, Singapore, 2016.

[15] W. Alsabbagh and P. Langendörfer, "A Stealth Program Injection Attack against S7-300 PLCs," 2021 22nd IEEE International Conference on Industrial Technology (ICIT), 2021, pp. 986-993, doi: 10.1109/ICIT46573.2021.9453483.

[16] W. Alsabbagh and P. Langendörfer, "A Remote Attack Tool against Siemens S7-300 Controllers: A Practical Report," presented at 11. Jahreskolloquium Kommunikation in der Automation (KommA 2020), Lemgo, Germany, Oct. 29, 2020.

[17] M. Stouffer, V. Pillitteri, “Guide to industrial control systems (ics) security,” NIST special publication, 2015.

[18] “Framework for improving critical infrastructure cybersecurity version 1.1,” National Institute of Standards and Technology, Tech. Rep., 2018, Available at: https://doi.org/10.6028/NIST.CSWP.04162018.

[19] “Recommended practice: Improving industrial control system cybersecurity with defense-in-depth strategies,” Department of Homeland Security, Tech. Rep., 2016.

[20] W. Alsabbagh and P. Langendörfer, "Patch Now and Attack Later - Exploiting S7 PLCs by Time-Of-Day Block," 2021 4th IEEE International Conference on Industrial Cyber-Physical Systems (ICPS), 2021, pp. 144-151, doi: 10.1109/ICPS49255.2021.9468226.