Automated Reconstruction of Control Logic for Programmable Logic Controller Forensics

Syed Ali Qasim[1], Juan Lopez Jr[2], Irfan Ahmed[1]

[1] Virginia Commonwealth University, Richmond VA 23284, USA
{qasimsa, iahmed3}@vcu.edu
[2] Oak Ridge National Lab Oak Ridge, TN 37830
lopezj@ornl.gov

Abstract. This paper presents `Similo`, an automated scalable framework for control logic forensics in industrial control systems. `Similo` is designed to investigate denial of engineering operations (DEO) attacks, recently demonstrated to hide malicious control logic in a programmable logic controller (PLC) at field sites from an engineering software (at control center). The network traffic (if captured) contains substantial evidence to investigate DEO attacks including manipulation of control logic. `Laddis`, a state-of-the-art forensic approach for DEO attacks, is a binary-logic decompiler for the Allen-Bradley's RSLogix engineering software and MicroLogix 1400 PLC. It is developed with extensive manual reverse engineering effort of the underlying proprietary network protocol and the binary control logic. Unfortunately, `Laddis` is not scalable and requires similar efforts to extend on other engineering software/PLCs. The proposed solution, `Similo`, is based on the observation that engineering software of different vendors are equipped with decompilers. `Similo` is a virtual-PLC framework that integrates the decompilers with their respective (previously-captured) ICS network traffic of control logic. It recovers the binary logic into a high-level source code (of the programming languages defined by IEC 61131-3 standard) automatically. `Similo` can work with both proprietary/open protocols without requiring protocol specifications and the binary formats of control logic. Thus, it is scalable to different ICS vendors. We evaluate `Similo` on three PLCs of two ICS vendors, i.e. MicroLogix 1400, MicroLogix 1100, and Modicon M221. These PLCs support proprietary protocols and the control logics written in two programming languages: Ladder Logic and Instruction List. The evaluation results show that `Similo` can accurately reconstruct a control logic from an ICS network traffic and can be used to investigate the DEO attacks effectively.

Keywords: Control System · SCADA · Forensics · PLC · ICS

1 Introduction

Industrial control systems (ICS) monitor and control our critical infrastructures such as nuclear plant and gas pipelines. These systems were originally designed

to be isolated environments with limited access to the outer world. Increasingly, they are now connected to the Internet and corporate networks, thereby making them vulnerable to cyber attacks [3, 18, 10, 19].

Unfortunately, the current forensic capabilities are insufficient to investigate cyberattacks on ICS environments because these environments are significantly different from traditional IT systems [1, 2]. They are connected with physical processes, have the critical requirement of high availability, and use resource-constrained computing devices, legacy operating system, and proprietary network protocols.

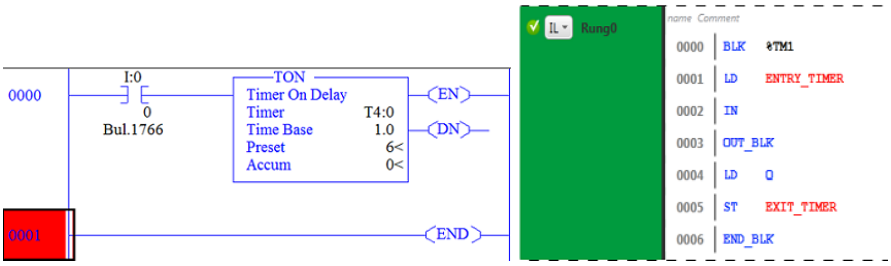
An ICS consists of control center and field sites. The control center runs ICS services such as human machine interface (HMI), historian, and engineering workstation. The fields sites have physical process, and computing devices such as sensors, actuators, and programmable logic controllers (PLCs). A PLC maintains a desired actuator state by a control logic and observing the current state of a physical processes using sensor data. The PLC also communicates the sensor data and actuator state to control center over a communication channel.

Recently, Senthivel *et al.* [16] present a new class of ICS attacks, namely, denial of engineering operations attack (DEO). In DEO I, an attacker compromises a control logic of a target PLC. When an engineering software attempts to retrieve the control logic from the compromised PLC, it intercepts the traffic via man-in-the-middle attack and replaces/removes the malicious logic from the control logic in the network traffic before forwarding it to the software. Hence, the engineering software receives a normal (non-malicious) control logic. In DEO II, which is a variant of DEO I, the attacker replaces a legitimate instruction in a control logic with noise data such as `0xFFFFFF` to make the engineering software malfunction.

The ICS network traffic contains substantial evidence to investigate DEO attacks including manipulation of control logic. The challenge is to reconstruct and transform the binary control logic (in the traffic dump) into its high-level source code. The closest effort in this direction is `Laddis` [16], which is a binary-logic decompiler for the Allen-Bradleys RSLogix engineering software and MicroLogix PLC series. Unfortunately, `Laddis` is not scalable and requires manual reverse engineering to extend on other engineering software/PLCs.

This paper presents `Similo` to recover a control logic from an ICS network traffic automatically. `Similo` is based on the observation that engineering software of different vendors are equipped with a decompiler that transforms a binary control logic into a high-level language source-code. `Similo` is an automated and scalable framework (for control logic forensics), which utilizes the *upload* function of an engineering software to integrate a previously-captured network traffic dump of a control logic with a decompiler in the engineering software. The framework does not require manual reverse engineering efforts for proprietary protocols and binary control logic. Thus, it is scalable.

We evaluate `Similo` on 113 control logic programs at three different levels: packet-level, functional-level and source-code-level of control logic. We use the engineering software of two different vendors, Allen-Bradley and Modicon, and



(a) Ladder Logic (b) Instruction List

Fig. 1: Different representations of a timer program

three PLCs supporting two IEC 61131-3 programming languages i.e., Instruction List and Ladder Logic and two proprietary protocols i.e., PCCC, and M221 proprietary layer encapsulated in Modbus. The evaluation results show that `Similo` can engage an engineering software using an ICS network traffic dump of a control logic including session establishment, echo messages and transferring of the control logic in the traffic dump to an engineering software. This results in a correct reconstruction and transformation of a control logic into a source-code. We further recreate DEO attacks on these PLCs and engineering software and utilize `Similo` to investigate them successfully.

2 Background

2.1 Control Logic

Programmable Logic Controller. PLCs are embedded devices that reside on field-sites to control and monitor physical processes directly. A PLC has input and output modules. The input module connects input devices such as sensors that provide temperature and pressure in a pipeline, level of liquid in a tank, etc. The output module connects with actuators to maintain the desired state of a physical process. The control logic in a PLC processes the input to set the output. A PLC also supports network communication (such as Ethernet or serial port) to communicate with the ICS services in control center such as engineering software.

PLC Programming. IEC 61131-3 defines five languages to write a control logic. These languages can be divided into two categories, i) Textual, and ii) Graphical. Structured text, and Instruction list are textual while Ladder Logic, Functional Block Diagram, and Sequential Function Chart are graphical. Note that for the purpose of evaluation, we select one language from each category i.e Ladder logic (graphical) and Instruction List (textual).

Ladder Logic. Ladder logic is a graphical language and is derived from Relay Logic. The program is defined in the form of a graphical diagram. A horizontal line in a Ladder logic program is called *rung*. A rung comprises of a

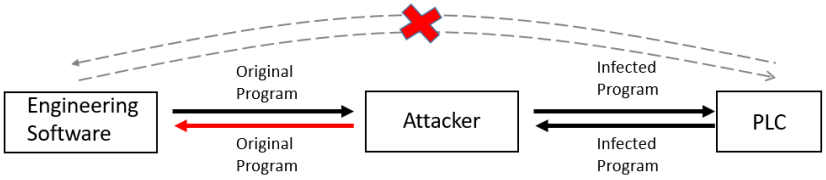


Fig. 2: DEO Attack I: Hiding infected ladder logic from the engineering software

number of input and output instructions. An instruction defines an operation to be performed by the processor [4].

Figure 1a is a ladder logic program consisting of one rung and two instructions: 1) *XIC* (Examine if closed) on left is associated with the input address *I:0/0*, 2) *TON* (timer on delay) on right. The timer instruction has three attributes, i) time base (the unit of time, 1.0 means one second). ii) Preset (maximum time to wait). iii) Accumulator (the time that has passed). It also has two control bits, *EN* (enable) and *DN*(Done).

When the program executes and the *XIC* is true, it will start the timer and *EN* will become true. The preset is 6 and the time-base is one second. When the timer completes 6 seconds, the *DN* bit turns to true and the accumulator is changed to the preset value.

Instruction List. Unlike Ladder Logic, Instruction List resembles assembly language consisting of sequence of instructions. Figure 1b shows an equivalent program in Figure 1a. The first instruction *BLK* is the start of the timer function block. The second instruction, *LD* (load operator) looks for close edge contact, which is associated with the input *%I0.0*. The contact is closed when bit *%I0.0* is 1. The following instructions are as follows: *IN* represents the input of Timer function block; *Out_BLK* wires the output of timer; *Q* represents the output of timer, and it becomes 1 when the timer expires; *ST* is store operator, which is equivalent to a coil in ladder logic and takes the value of previous logic and is used to store output. Finally, *END_BLK* represents the end of the timer function block [12].

When the program executes and *LD* is true, it sets *IN* true and starts the timer. The timer has a time-base of 1 second and preset of 6 second. When the timer completes 6 seconds, it sets *Q* (output of timer) true and then both *LD* and *Q* go into *ST*. *LD* and *Q* are in series. When both *LD* and *Q* are true, it will turn the output *ST* true.

2.2 Denial of Engineering Operations (DEO) Attack

Recently, Senthivel *et al.* [16] present denial of engineering operation (DEO) attacks that jeopardize an engineering software's capabilities to perform remote-maintenance on a PLC. They demonstrate the attacks on Allen-Bradley MicroLogix 1400-B and RSlogix 500 (engineering software).

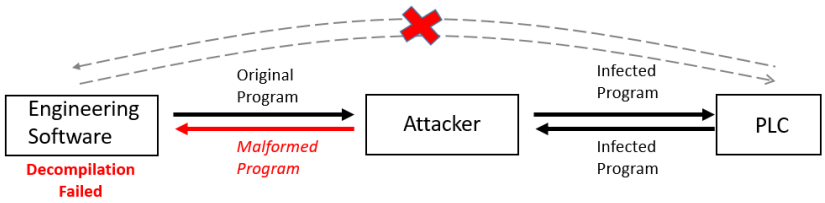


Fig. 3: DEO Attack II: Crashing the decompiler running on Engineering software

1) DEO Attack I. In DEO I (Figure 2), an attacker performs a man-in-the-middle between a target PLC and an engineering workstation (the computer running an engineering software). When the control engineer downloads a control logic program to a compromised PLC, the attacker intercepts the communications and infects this control logic by replacing some part of the code with malicious logic before forwarding it to the PLC. Similarly, when the control engineer tries to upload the control logic from the PLC, the attacker intercepts the traffic and replaces the infected logic with the original code. In this way, the control engineer remains unaware of the malicious control logic running on the PLC.

Consider the ladder logic program in Figure 1a, the timer controls the yellow light in a traffic light signal. The attacker modifies the preset value from 6 seconds to 80 seconds when the program is downloaded to the PLC of the signal. When a control engineer attempts to retrieve the program from the PLC, the attacker intercepts the traffic and change the preset back to its original value i.e., 6.

2) DEO Attack II. DEO II is similar to the DEO 1 in that the attacker performs a man-in-middle between the engineering workstation and PLC, intercepts the communication, and manipulate the traffic as it passes through the attacker's machine. However, in DEO II (Figure 3), the attacker replaces the original code with random (noise) data such as 0xFFFF. When an engineering software receives the malformed logic, it fails to decompile.

2.3 Challenges in DEO Forensic Investigation

For a forensic investigation of DEO attacks, the network traffic (if captured) contains substantial evidence including manipulation of control logic. The challenge is to reconstruct and transform the binary control logic (in the traffic dump) into its high-level source code. Unfortunately, binary control-logic does not have a standard open format (such as Linux ELF) to allow a generic decompiler. ICS vendors define their binary control-logic representations. Often, each vendor has multiple binary representations across their different engineering software to program different types of PLCs.

Recall that IEC 61131-3 standard defines five programming languages for PLCs (such as Structured Text, and Ladder Logic) [9]. An engineering software often supports only one or two languages. Thus, binary logic must be transformed into their respective high-level languages for forensic investigation, mak-

ing the transformation more challenging. Lastly, the engineering software and PLCs communicate using different ICS protocols that may be proprietary or may use an open protocol with an embedded proprietary protocol layer. Thus, reconstruction of binary control logic from a network traffic capture requires extensive manual reverse engineering of the proprietary protocols.

The closest effort in this direction to develop forensic investigation capabilities for control logic is `Laddis` [16], which is a binary-logic decompiler for the Allen-Bradleys RSLogix engineering software and MicroLogix PLC series. `Laddis` is developed with the manual reverse engineering of the PCCC protocol and the binary representation of the high-level ladder logic program written in RSLogix. Unfortunately, `Laddis` is not scalable and requires similar efforts to extend on other engineering software/PLCs.

3 Problem Statement

Given an ICS network traffic dump of a control logic, our goal is to reconstruct and transform the binary control logic (in the traffic dump) into its high-level source code. Considering the challenges outlined in Section 2.3, a practical solution should address at least two basic requirements:

Automation. The solution must be automated to achieve a high-level source code of a low-level binary control logic in a network traffic without human intervention including reverse engineering of a proprietary ICS protocol and a binary representation of a high-level control logic.

Scalability. The solution must be scalable to multiple vendor products including engineering software (used to create a control logic), proprietary ICS protocols, and PLCs.

4 Similo - A virtual PLC Framework

4.1 Overview of Similo

We observe that engineering software of different vendors are equipped with decompilers that can transform a binary control logic into a high-level language source-code. We propose to integrate a decompiler in engineering software with a previously-captured network traffic dump of a control logic to obtain the source-code of the control logic. Our solution is `Similo`, an automated and scalable virtual-PLC framework that does not require manual reverse engineering. `Similo` utilizes the *upload* function of an engineering software to achieve the integration.

Upload function. The *upload* is a required functionality (used by control engineers) to retrieve a binary control logic from a PLC remotely, which further triggers a decompiler in engineering software to achieve high-level source code of the control logic.

Generally, when a control engineer runs the *upload* command in engineering software, it starts a series of request-response messages between a PLC and an engineering software such as session-establishment messages, echo messages,

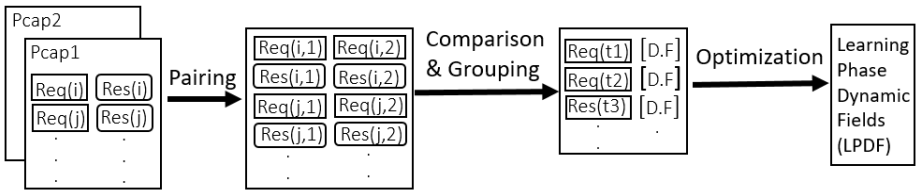


Fig. 4: Overview of Learning/Training phase

and control logic messages. Engineering software first establishes a session with a PLC and then, sends read-request messages to the PLC to read the memory locations of a control logic. In response, PLC sends the data on the requested memory locations (i.e., control logic) to the engineering software in the payload of response messages. After receiving an entire binary control logic, engineering software passes it to the decompiler to trigger decompilation process, which in turn produces the source code in a high-level language.

Virtual-PLC framework. To develop `Similo`, we assess the communication behavior of the *upload* function of two engineering software, RSLogix 500 and SoMachine-Basic with three PLCs, Allen-Bradley's MicroLogix 1400 and MicroLogix 1100, and Schneider Electric's Modicon M221. We make two interesting observations that show that the communication behavior is deterministic: first, an engineering software always makes a small number of unique requests to retrieve the control logic from a PLC; second, if we send an associated response message from a previous network dump as reply to a request message from engineering software, the next request message from the software will be same as the next request message in the network traffic dump.

Based on these observations, we design `Similo` using the *upload* function. Recall that engineering software uses the *upload* function to retrieve control logic from a PLC memory. `Similo` on the other hand retrieves control logic from a network traffic (captured during the transfer of the logic). It consists of a virtual-PLC that responds to the *upload* function queries using a previous network traffic dump of a control logic. It handles dynamic protocol fields in the request-response messages automatically, making it scalable to different PLCs, proprietary protocols and engineering software. For this paper, we test `Similo` successfully on three different PLCs (Micrologix 1400, MicroLogix 1100 and Modicon M221), two ICS protocols (ENIP, and Modbus) and two engineering software (RsLogix, and SoMachineBasic).

`Similo` consists of two phases: training, and testing. The training phase provides understanding of dynamic header fields of messages using benign pcap files while the testing phase engages an engineering software to respond to the request messages using the response messages in a network traffic (under investigation) including updating the header fields.

4.2 Learning/Training Phase

Figure 4 presents an overview of the training phase, which consists pairing, comparison and grouping, and optimization steps for identifying dynamic header fields in request-response messages.

Pairing. Pairing is the first step to identify an instance of a message in a set of two benign pcap files from different sessions that contain same control logic. We assume that the header values of dynamic fields change across multiple sessions. However, their contents (control logic) remain the same since same control logic is used on both pcap files. We use two properties of a message to find same message instance in the pcap files: 1) message length and 2) message content similarity.

Ideally, we have to compare each message of the first pcap file with all messages of the second pcap file to find the best match. However, we optimize this approach by finding a match with 85% threshold (based on our initial experiments) i.e if the length of two messages is same and the similarity is more than 85%, they are considered same and paired together. In our experience, this approach decreases the time taken for learning significantly without affecting the functionality of `Similo`. Note that pairing is used for initial screening of pcap files and does not assume to achieve 100% accuracy for finding same messages. The results of pairing are further refined in later stages. Figure 4 show the pairing process, where *Req (i,1)* and *Res (i,1)* is a request response pair from pcap1 and *Req (i,2)* and *Res (i,2)* is a pair from pcap2.

Comparison and Grouping. After pairing similar messages, `Similo` performs differential analysis on each pair, i.e comparing two messages character by character and records the indices (i.e., locations of bytes) where the values are different. During our experiments, we found that the length of header fields vary in different request messages due to which the offsets of dynamic fields also vary. In order to tackle this, `Similo`groups messages based on length such that all the messages in one group will have the same header size and structure. There after `Similo`find the differences of all message pairs in one group which are further processed to get the dynamic fields.

Optimization. In this process, the differences identified between the message pairs in each group are compared with one another and only those indices are selected that are present in more than 50% of messages. Since the initial pairing is not 100% accurate, there is a chance that other than the dynamic header fields, some paired messages may also have little differences in payload too. So the optimization process filters the differences present in payload. For example if the differential analysis of three message pairs of length X has resulted in the following dynamic field indices: *(0,1,4,9,19),(0,3,4,15),(1,3,4,22)* the resultant would be *(0,1,3,4)*, which will represent the offsets of dynamic fields in all the messages in group X. The optimized indices are further divided in different groups based on the adjacency and each group represents one dynamic field such as transaction ID, length etc. These dynamic fields might be incomplete/partially filled but that problem is solved during the testing phase.

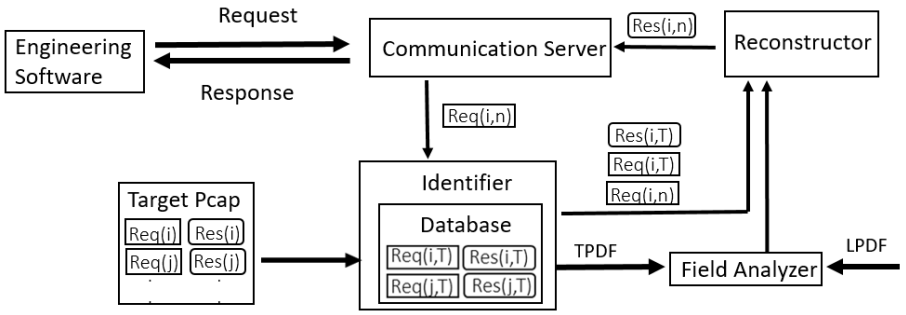


Fig. 5: Overview of Testing phase

After these steps, `Similo` gets the indices of dynamic fields in all the request-response messages present in one set of pcap files. The same process is repeated with other pcap files and finally the results of all the files are again compared and analyzed using the majority rule and the information and the resulting dynamic fields, referred to as Learning Phase Dynamic Fields (LPDF), are used in testing phase.

4.3 Testing Phase

Figure 5 shows an overview of the testing phase. After completing the training phase, `Similo` takes a target pcap file, extracts request and response messages, and then, stores them in database in the form of request and response pairs. Afterwards, it starts the communication server and waits for the message from the engineering software initiated by the *upload* function. Upon receiving a request, the communication server forwards it to the Identifier. The identifier performs two tasks. First, it finds the same request message (based on content) in the database. Second, it compares the two messages and identifies the dynamic fields between these two request messages. We call them training phase dynamic fields (TPDF).

The identification is similar to pairing since it uses message length and content similarity. However, at this stage, we have information about the dynamic fields from the learning phase. Note that the dynamic fields are present in the header and the later part contains the control logic. For every request message with the same length as of the new request message, instead of comparing the whole message, the identifier only compares the part that lies beyond the last dynamic field determined by learning phase.

The grouping of messages based on length helps `Similo` in performing the look up efficiently. The identifier selects the request message with highest similarity with a new request message. It then, passes the request along with the request-response pair from the database to *Reconstructor*. Similarly, the testing phase dynamic fields are passed to Field analyzer.

	Learning phase dynamic fields	Testing phase dynamic fields	Combined dynamic fields
Overlap	((0,1,2),(6,7))	((2,3),(9))	((0,1,2,3),(6,7))
Adjacent	((0,1,2),(6,7))	((3),(9))	((0,1,2,3),(6,7))
Confined	((0,1,2),(6,7))	((1),(9))	((0,1,2)(6,7))

Fig. 6: Accumulation example of dynamic fields in learning and testing phases

Field Analyzer. We know the location and tentative size of dynamic fields in a message, however we still have to ascertain the boundary of the fields. To find complete fields, field analyzer compares the dynamic fields from learning phase with the dynamic fields from the testing phase. Specifically, if any dynamic field from the TPDF overlaps, is adjacent to, or is confined in any dynamic field from the LPDF, Field analyzer combines it with the dynamic field from learning phase otherwise it discards it. Figure 6 explains the working of Field Analyzer. In the first case two fields were identified in the testing phase i.e *(2,3)* and *(9)*. Since *(2,3)* is overlapping one of the LPDF, it is combined with it resulting in *(0,1,2,3)* where as *(9)* is not overlapping, adjacent or confined in any of the LPDF so it is discarded. Similarly the second and third case explains the Adjacent and Confined scenarios.

It is still possible that even after this combination some fields are partly empty. That means the values at those indices remain same in both sessions, thus we do not need to change them for reconstructing the response message. The final dynamic fields are forwarded to Reconstructor.

Reconstructor. It is the last component in `Similo`, which takes request-response messages from the target pcap and the dynamic field offsets from the Field analyzer. The dynamic fields in a target request message are mapped to its paired response message. If the values are same, reconstructor changes the values of dynamic fields in the response message according to the values in the new request message and forwards this message to the communication server. The communication server then sends this response message to the engineering software and waits for next request and so on. This process finally makes the engineering software to recover the control logic from the network dump.

5 Implementation

We have implemented `Similo` in python and used scapy [14] for network packet manipulation. During the learning phase, `Similo` makes dictionaries from the pcap files. The request and response messages are filtered on the basis of IP address and port. The transport layer payloads of request and response messages are converted to hex streams and used as keys and values.

To calculate similarity, we use SequenceMatcher from difflib library [8]. Furthermore, `Similo` compares both sets of requests and response messages present in each tuple, character by character and the differences are stored in a dictionary, where length of request message represents the key and value is a list of

arrays of differences generating from each comparison. These differences are later processed to get the offsets of dynamic fields within a packet via Optimization.

The optimization uses a majority rule to separate the protocol related dynamic fields from the rest. For each message type (based on length), it calculates the number of instances of each offset. If an offset appears in majority (more than 50% or user defined threshold), it is considered as part of dynamic field and used in the testing phase, otherwise, it's ignored.

During our research, we found that generally, the PLCs have fixed ports for communicating with the engineering software e.g Allen-Bradley MicroLogix 1100 and 1400 use port 44818, Modicon M221 uses port 502. Thus, in the testing phase, using the socket library, `Similo` opens a server socket (communication server) using socket on the default ports of real PLCs and waits for message from the engineering software. After getting a target pcap file from the user, it generates the database i.e dictionary using the method explained.

The identifier is a search function that takes a request message from the server *Req (i,n)* and iterates on the database keys to finds same request message with different dynamic fields *(req_t)*. For this purpose, it uses the length and similarity of static fields. After finding same request from the target pcap *Req(i,T)*, it compares these two requests to find the differences. The Identifier then passes the *Req (i,n), Req (i,T)* and *Res (i,T)* to the reconstructor and *TPDF* to the Field analyzer.

The field analyzer function takes two inputs: LPDF, and *TPDF*. It iterates over both of them and if any *TPDF* fields are adjacent, overlap or one is confined in the boundary of a LPDF, it combines the two, otherwise, it ignores the *TPDF* (Figure 6). The output of this function is an array of arrays containing dynamic field offsets. Finally the Reconstructor function takes the *Req(i,n), Req(i,T)* and *Res (i,T)* from the Identifier and the final set of dynamic fields from the Field analyzer. It maps the dynamic fields in *Req (i,T)* on *Res (i,T)* to check if the values of dynamic field in the request and response message are same. If it is true, it edits the *Res (i,T)* by changing the value of dynamic fields according to the new request *R(i,n)* and forwards the new response message to the communication server, which then sends it to the engineering software.

6 Evaluation

Lab Setup. We evaluate `Similo` on three PLCs Allen-Bradley MicroLogix 1400 Series B, Allen Bradley MicroLogix 1100 Series B, and Schneider Electric Modicon M221. The engineering softwares used for the first two PLC is RSLogix 500 V9.2.01 and M221 is evaluated on SoMachine Basic v 1.6 and v 1.4. Both programming software run on Windows 7 virtual machine (VM) and the virtual-PLC runs on a VM with Ubuntu v 16.04. The engineering software, PLCs and virtual-PLC all were connected via Ethernet.

Experiment Methodology. A typical experiment includes capturing the network traffic when an engineering software uploads a control logic from a real

Table 1: Dataset summary of Ladder logic programs for MicroLogix 1100

File Information		Rung				Instruction			
File size (KB)	**# of Files**	**Min**	**Max**	**Total**	**Avg.**	**Min**	**Max**	**Total**	**Avg**
0-40	16	2	17	90	5.62	3	48	240	15
41-60	1	4	4	4	4	12	12	12	12
61-80	4	8	63	145	36.25	25	245	543	135.75
81-100	1	13	13	13	13	37	37	37	37
Total	**22**	-	-	**252**	-	-	-	**832**	-

Table 2: Dataset summary of Ladder logic programs for MicroLogix 1400

File Information		Rung				Instruction			
File size (KB)	**# of Files**	**Min**	**Max**	**Total**	**Avg.**	**Min**	**Max**	**Total**	**Avg**
20-40	21	1	17	99	4.71	1	48	276	13.14
41-60	8	4	48	93	10.33	4	53	344	38.88
61-80	7	8	63	149	22.57	28	245	577	96.166
81-100	2	13	15	28	14	15	37	52	26
101-120	1	10	10	10	10	23	23	23	23
Total	**39**	-	-	**379**	-	-	-	**1272**	-

PLC. `Similo` uses the pcap files and communicates with the engineering software to recover the control logic. At the end, two programs are compared in the engineering software manually to find accuracy of the virtual PLC.

Dataset. For the evaluation of the PLCs, Allen-Bradley MicroLogix 1400 and MicroLogix 1100, we use 39 and 22 different Ladder logic programs respectively. For Modicon M221, we use 52 Instruction List programs. These programs were written for different physical processes such as traffic light, hot water tank, elevator, gas pipeline, and vending machines, and are of varying complexity and sizes. Tables 1, 2 and 3 show the features of the datasets for MicroLogix 1400 and 1100, and Modicon M221 respectively.

6.1 Virtual PLC as a Device

`Similo` establishes and maintains a connection with engineering software as a real PLC. We evaluate it with two engineering software i.e RSlogix and SoMachine Basic and conclude that both software recognize `Similo` as a device and does not distinguish between real PLC and `Similo`. Figure 1 shows the outcome of the experiments where `Similo` is recognized as a real MicroLogix 1100, MicroLogix 1400 and Modicon M221 PLC. The experiments are performed as follows.

To connect Allen-Bradley MicroLogix 1100 and 1400 to the engineering workstation, the user has to manually configure a driver in the RSlinx Classic. For

Table 3: Dataset summary of Instruction List programs for Modicon M221

File Information		**Rung**				**Instruction**			
File size (KB)	**# of Files**	**Min**	**Max**	**Total**	**Avg.**	**Min**	**Max**	**Total**	**Avg**
60-80	30	1	3	72	2.4	2	23	793	26.4
80-100	14	2	27	107	7.64	7	112	463	33
100-130	4	8	14	43	10.75	20	72	153	38.2
130+	4	12	26	63	16	36	118	269	67.2
Total	**52**	-	-	**286**	-	-	-	**1678**	-

Ethernet communication the user can select either EtherNet/IP driver or Ethernet devices driver. In case of Ethernet device driver the user has to give the IP address of the PLC device while EtherNet/IP driver searches the subnet to discover the PLC devices. In our experiments we configured Ethernet devices driver (AB_ETH-1) and gave it the IP address of `Similo` as shown in red circles in Figure 7a and Figure 7b, Rslinx classic identified `Similo` as a real MicroLogix 1100 and MicroLogix 1400 PLCs.

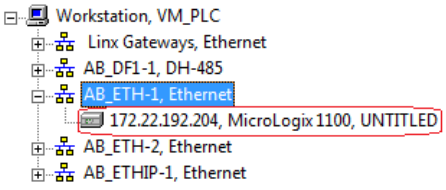
Similarly, in SoMachine Basic, user can either give the IP address of the PLC or browse the subnet with the help of refresh devices function available (marked in the figure). In our experiment we provide SoMachine Basic the IP address of `Similo`. Figure 7c shows that SoMachine Basic identified `Similo` as a real PLC (TM221CE16R).

6.2 Function-level Accuracy

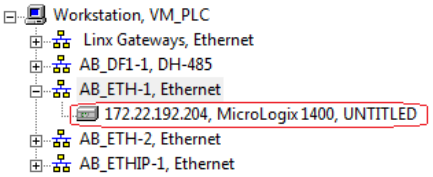
To successfully imitate a real PLC, `Similo` has to perform three tasks i.e i) establish a connection with the engineering software ii) handle non-control logic messages such as echo, and iii) upon receiving an *upload* request from the engineering software, correctly uploading the control logic (present in the pcap file). In this section, we evaluate the ability of `Similo` to establish and maintain a stable connection with the engineering software and upload the correct control logic to the engineering software.

Session establishment and maintenance. Note that apart from the transferring control logic, engineering software also sends ping (echo) messages and other functional commands to PLC. To test the robustness of `Similo` in establishing and maintaining the the connection, we perform the following experiment. Both RSLgix 500 and SoMachine Basic initiates a connection with `Similo` and keeps it open for few minute without requesting for an upload. During these experiments, `Similo` maintained the connection successfully in 113 cases.

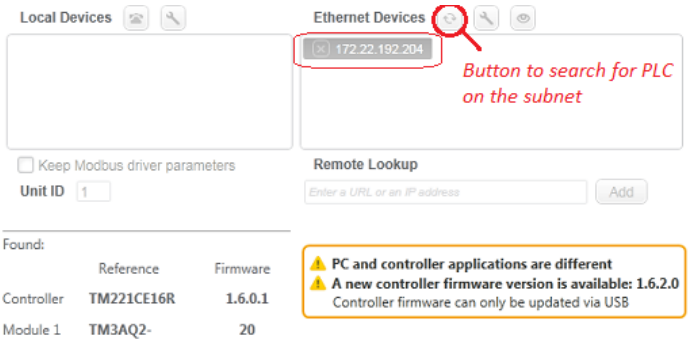
Transfer Accuracy. After establishing and maintaining the session successfully, the next task of `Similo` is to upload a given control logic in network traffic correctly. As mentioned in section 4, the *upload* function of engineering software sends a series of read requests to the PLC. In the beginning the engineering software gets the program storage information/metadata of the control logic from



(a) Similo recognized as MicroLogix 1100 by Rockwell Automation's RSLogix 500



(b) Similo recognized as MicroLogix 1400 by Rockwell Automation's RSLogix 500



(c) Similo recognized as Modicon M221 by Schneider Electric's SoMachine Basic

Fig. 7: Similo recognized as a real PLC (MicroLogix 1100 and 1400 and Modicon M221) by two engineering software, RSLogix 500 and SoMachine Basic

the PLC, then it starts reading the control logic binary from the PLC memory. During the upload process, upon receiving the request message, `Similo` searches for a response message in its database and sends the reply after editing the dynamic/session dependant fields.

At this stage, any changes other than the dynamic fields can disrupt the connection between the engineering software and PLC or damage the integrity of control logic. Our experiments show that `Similo` identified and edited the dynamic fields successfully while preserving the integrity of control logic being uploaded. Furthermore, we analyze `Similo's`capability to reverse-engineer the ICS proprietary protocols. To evaluate the accuracy of `Similo`, we manually calculated the number of rungs and instructions in each of the 113 control logic files and transferred them one by one with `Similo` to the engineering software. After each upload, the program was compared with the original files to see if the number of rungs and instructions are the same. To further check the integrity of control logic transferred by `Similo`, the instructions in original and `Similo`-transferred control logic were compared manually to check their order on the rung. Similarly, the values of other variables, such as timer preset, and timer base were also compared with the original program.

Table 4: Transfer accuracy of `Similo`

PLC	# of control logic files uploaded	Original Program		Similo Output		Accuracy
		Rungs	Instructions	Rungs	Instructions	%
MicroLogix 1100	22	252	832	252	832	100%
MicroLogix 1400	39	379	1272	379	1272	100%
Modicon M221	52	286	1678	286	1678	100%

MicroLogix 1400. For Allen-Bradley MicroLogix 1400 PLC, 39 ladder-logic programs containing 379 rungs and 1272 instructions were uploaded. `Similo` showed 100% accuracy in establishing connection, basic communication and control logic upload. Moreover, in all cases, the original programs and the ones uploaded by `Similo` were identical. Table 4 shows the transfer accuracy of `Similo`.

MicroLogix 1100. To evaluate the accuracy of `Similo` on MicroLogix 1100, we used 22 ladder-logic programs of varying complexities containing 252 rungs and 832 instructions. `Similo` was able to upload all programs with 100% transfer accuracy.

Modicon M221. Modicon M221 was evaluated using 52 different programs in Instruction-list, comprising of 286 rungs and 1678 instructions. These programs varied in terms of complexity ranging from as minimum as one rung and two instructions per program to more than 20 rungs and 100+ instructions per program. During our experiments `Similo` showed 100% accuracy in uploading the control logic from the pcap files.

6.3 Packet-Level Accuracy

One of the main heuristics in developing `Similo` is the deterministic behaviour of engineering software. The engineering software uses same set of messages to initiate a connection or request for upload. Thus, keeping the deterministic behaviour of the engineering software in mind, if `Similo` has a complete network traffic of a previous session, it can use it to communicate with the engineering software with a high probability that all request-messages from engineering software can be found in the network traffic. The results from our experiments strengthens this theory.

This section evaluates `Similo's` ability to identify a given request-message in the database (target pcap file). Table 5 shows the results of packet-level accuracy of `Similo`. During the process of uploading 52 control logic programs as Modicon M221, `Similo` received 8800 request messages from the engineering software. Out of these, 8776 message of same as length and average similarity of 99.99 % were present in the database. For the remaining 24 messages, `Similo` selected the request message with the closest length. In this case the average similarity of

Table 5: Packet-level accuracy of Similo

PLC	No. of files	Request messages received	Request messages present in DB		Request messages not present in DB	
			No.	Avg. Similarity %	No.	Avg. Similarity %
MicroLogix 1100	22	1639	1639	100%	0	-
MicroLogix 1400	39	4219	4219	100%	0	-
Modicon M221	52	8800	8776	99.99%	24	56.39%

the messages selected by `Similo` is 0.58%. Although this similarity is not perfect, but the engineering software accepted the response message from `Similo` without crashing or giving errors and the overall behaviour of the communication does not change. Similarly, for MicroLogix 1400, while uploading 39 control logic files, `Similo` received 4219 messages and all of these were present in the database with average similarity of 100%. For MicroLogix 1100, during the upload process, `Similo` received 1639 and all of them were present in the database (target pcap files) with 100% accuracy.

6.4 Forensic Analysis using Similo

To evaluate `Similo` for a forensic analysis of a real cyberattack, we used two denial of engineering operations (DEO) attacks for Allen-Bradley MicroLogix 1400, presented by Senthival *et al.* [16]. This section contains the attack summary and execution details along with a forensic analysis using `Similo`.

DEO Attack I: Hiding infected ladder logic (running in the PLC) from engineering software. In the first attack, the attacker performs a man-in-the-middle between PLC and the engineering workstation (computer running the engineering software). When a ladder logic program is downloaded to the PLC, the attacker replaces some portion of the original program with malicious logic. When a control engineer attempts to retrieve the program running on the PLC, the attacker intercepts the communication and replaces the infected logic with the original logic. In this way, the engineering software shows the original program and the attacker deceives the engineer successfully.

Attack Execution. To achieve the man-in-the-middle, we used ARP poising via Ettercap. The program used for this attack was designed to control the traffic light. It contains three timers, each controlling one of the signal lights (red, orange, green). The goal of the attack is to make a change in the timing of green light. The timer instruction consists of three parameters i.e base, preset and accumulated. The preset value controls the amount of time. So to achieve the goal, when the Control engineer downloads this program to the PLC (MicroLogix 1400), using a custom built Ettercap filter, we change the value of preset from 20 to 80. Now the green light will stay ON for 80 seconds instead of 20. Similarly when the control engineer uploads the ladder logic program from the PLC, we

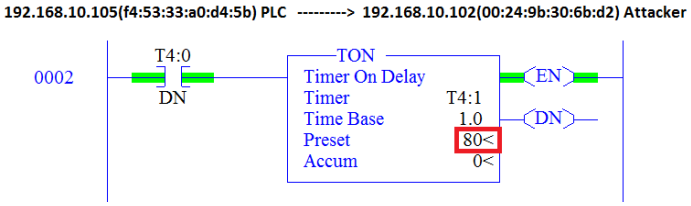


Fig. 8: Control logic from PLC to Attacker

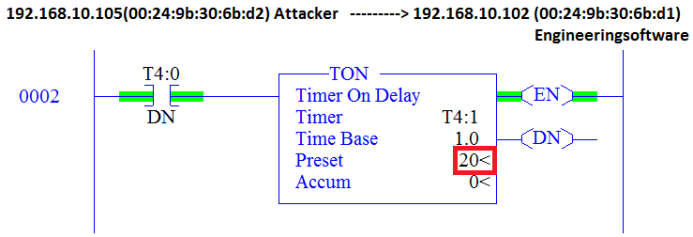


Fig. 9: Control logic from Attacker to Engineering Workstation

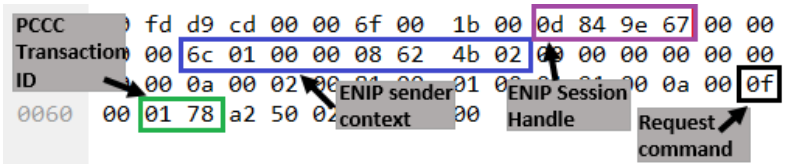
replace the preset value back to 20 and the control engineer only sees the original program on the engineering software. Thus, the PLC runs the infected ladder logic but the Control engineering is not aware of this infection.

Forensic analysis. To investigate the DEO attack, we use `Similo` to recover both instances of the control logic in a network traffic capture i.e., one between the engineering software to the attacker, and the other between attacker and the PLC. We utilize MAC addresses to separate the network traffic. Figure 8 and Figure 9 show the recovered instances of the control logic, one is original where the other is manipulated by the attacker by changing the timer preset value from 20 to 80.

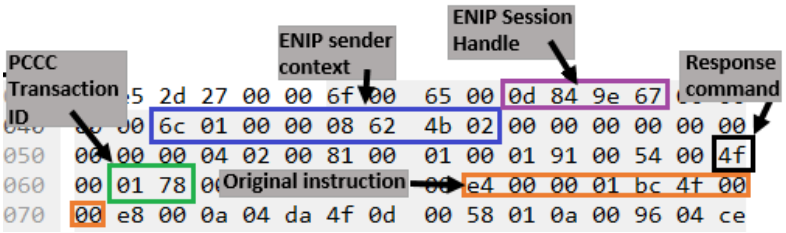
DEO Attack II: Crashing an engineering software. In the second DEO attack, the attacker performs a man-in-the-middle between the PLC and the workstation. Whenever a control engineer tries to upload a ladder logic program from a target PLC, the attacker intercepts the traffic and modifies the ladder logic instructions by adding random noise such as the sequence of 0xFF bytes. Apparently, it fails the decompilation process in engineering software. This DEO attack is a denial of service, which jeopardizes a control engineer's capability to retrieve the program running on the PLC using engineering software.

Attack Execution. Similar to the first attack, the attacker uses ARP poisoning (with Ettercap) to achieve man-in-the-middle. When a control engineer tries to upload the code from a target PLC, the attacker intercepts the communication and replaces a genuine instruction with a malfunctioning one. Figure 10b and Figure 10c show the original and malformed messages.

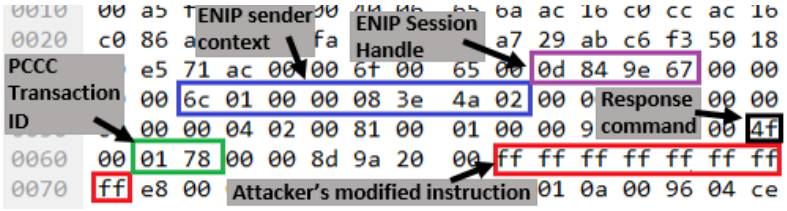
Forensic Analysis. To investigate the DEO attack, we use MAC addresses to separate the two instance of the control logic, and then utilize `Similo` to



(a) Request message from the engineering software



(b) Response message from PLC to attacker



(c) Malformed response from attacker to Engineering software

Fig. 10: Request and response packets that crash the decompiler

attempt to recover the control logic. To identify malformed control logic packet between engineering software and attacker, we initiate the *upload* function. However, the malformed packet disrupts the engineering software and makes it unavailable for further communication with `Similo`. `Similo` identifies the packet that has caused the disruption since no further communication is possible after the packet is transmitted. Figure 10 shows response message from both benign and manipulated control logic. Figure 10(c) is identified by `Similo`.

To recover the second (malicious) instance of the control logic between the attacker and infected PLC, `Similo` utilizes the *upload* function again and transmit the control logic to the engineering software successfully, resulting in the recovery of the logic to high-level source code.

7 Related Work

When a PLC is compromised by an adversary, it can have disastrous effect on ICS infrastructures. For instance, Stuxnet damages a nuclear plant physically

and it does so by changing the frequency of a motor drive controlling the speed of a centrifuge [6].

Valentine *et al.* [17] mentioned that a simple intentional or unintentional error in ladder logic can disrupt the availability and integrity of a target PLC. For example, if output coil is removed in ladder logic, the code will still compile and run on the PLC. However, it will not trigger an intended alarm disrupting the functionality of the PLC.

Kotler *et al.* [11] say due to limited resources installing an additional program on PLC to check the control logic programs is difficult and it will put extra computational burden on the PLC. The authors propose a method using formal verification to detect malicious or faulty PLC programs. They used NuSMV to perform the formal verification. Cheung *et al.* proposed [7]a model based intrusion detection technique for SCADA network that monitors Modbus TCP.

Patel *et al.* recognize [13] that many private protocols exists in SCADA networks. Since most of these proprietary protocols are designed to maximize performance only few of them have built-in security features like message authentication. Attackers can use reverse engineering approach to get the communication and network information.

Beresford *et al.* [5] presented different way to attack Siemens Simatic S7 PLC by reverse engineering and modifying International Standards Organization Transport Service Access Point protocol (the standard protocol for communicating with and programming all S7 Programmable Logic Controllers made by Siemens). Using a replay attack the attacker can perform all the tasks a normal engineering software do like turning off the CPU, disabling memory protection and uploading new project files to the PLC.

After the attack there are very few tools that can analyze the network capture and extract the control logic transferred during the attack. Moreover the available tools focus on one protocol or control logic language. Senthivel *et al.* [15] made Cutter, a tool to extract the PCCC traffic from network capture. This tool parses the pcap files according to protocol specific tags. Cutter only extracts the PCCC data from the files so the authors made another tool Laddis, which decompiles the files extracted by cutter to get the high level control logic code from binaries. These tools are developed entirely on reverse engineering and are only limited to PCCC protocol extraction and decompilation. This raises the need of a generic tool which can use network dump to get the high level presentation of control logic and Similo tries to solve that problem

8 Conclusion

We presented a fully-automated framework `Similo` to recover a control logic from an ICS network traffic. `Similo` integrated a previously captured ICS traffic with the decompiler and handled dynamic header fields automatically without manual reverse engineering. We evaluated `Similo` on three different PLCs and two protocols of two different ICS vendors to show that `Similo` supported multiple vendors successfully. Furthermore, `Similo` was evaluated on the denial of

engineering operations attacks and recovered the malicious and original control logic from the network traffic accurately.

References

1. Ahmed, I., Obermeier, S., Naedele, M., Richard III, G.G.: SCADA Systems: Challenges for Forensic Investigators. Computer **45**(12), 44–51 (Dec 2012)
2. Ahmed, I., Obermeier, S., Sudhakaran, S., Roussev, V.: Programmable Logic Controller Forensics. IEEE Security Privacy **15**(6), 18–24 (November 2017)
3. Ahmed, I., Roussev, V., Johnson, W., Senthivel, S., Sudhakaran, S.: A SCADA System Testbed for Cybersecurity and Forensic Research and Pedagogy. In: Proceedings of the 2nd Annual Industrial Control System Security Workshop (ICSS) (2016)
4. AllenBradlry: User manual, `https://literature.rockwellautomation.com/idc/groups/literature/documents/um/1763-um001_-en-p.pdf`
5. Beresford, D.: Exploiting siemens simatic s7 plcs (2011)
6. Chen, T.M., Abu-Nimeh, S.: Lessons from stuxnet. Computer **44**(4), 91–93 (2011)
7. Cheung, S., Dutertre, B., Fong, M., Lindqvist, U., Skinner, K., Valdes, A.: Using model-based intrusion detection for scada networks. In: Proceedings of the SCADA Security Scientific Symposium. Miami Beach, Florida (Jan 2007)
8. diflib: `https://docs.python.org/3/library/difflib.html`
9. IEC: IEC 61131-3, `https://www.sis.se/api/document/preview/562735/`
10. Kalle, S., Ameen, N., Yoo, H., Ahmed, I.: CLIK on PLCs! Attacking Control Logic with Decompilation and Virtual PLC. In: Proceeding of the 2019 NDSS Workshop on Binary Analysis Research (BAR) (2019)
11. Kottler, S., Khayamy, M., Hasan, S.R., Elkeelany, O.: Formal verification of ladder logic programs using nusmv. In: SoutheastCon 2017. pp. 1–5 (March 2017)
12. Modicon: SoMachine Basic - Generic Functions Library Guide, `https://www.schneider-electric.com/en/download/document/EIO0000001474/`
13. Patel, S.C., Bhatt, G.D., Graham, J.H.: Improving the cyber security of scada communication networks. Commun. ACM **52**(7), 139–142 (Jul 2009)
14. Scapy: `https://scapy.net/`
15. Senthivel, S., Ahmed, I., Roussev, V.: SCADA Network Forensics of the PCCC Protocol. Digit. Investig. **22**(S), S57–S65 (Aug 2017)
16. Senthivel, S., Dhungana, S., Yoo, H., Ahmed, I., Roussev, V.: Denial of Engineering Operations Attacks in Industrial Control Systems. In: Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy. pp. 319–329. CODASPY '18, ACM, New York, NY, USA (2018)
17. Valentine, S., Farkas, C.: Software security: Application-level vulnerabilities in scada systems. In: 2011 IEEE International Conference on Information Reuse Integration. pp. 498–499 (Aug 2011). https://doi.org/10.1109/IRI.2011.6009603
18. Yoo, H., Ahmed, I.: Control logic injection attacks on industrial control systems. In: Dhillon, G., Karlsson, F., Hedström, K., Zúquete, A. (eds.) ICT Systems Security and Privacy Protection. pp. 33–48. Springer International Publishing, Cham (2019)
19. Yoo, H., Kalle, S., Smith, J., Ahmed, I.: Overshadow plc to detect remote control-logic injection attacks. In: Perdisci, R., Maurice, C., Giacinto, G., Almgren, M. (eds.) Detection of Intrusions and Malware, and Vulnerability Assessment. pp. 109–132. Springer International Publishing, Cham (2019)